IMPLEMENTING CONTINUATIONS FOR A VIRTUAL MACHINE ENVIRONMENT

_____

A Project

Presented

to the Faculty of

California State University, Chico

_____

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

_____

by

Anthony J. Lorelli

Spring 2006

IMPLEMENTING CONTINUATIONS FOR A VIRTUAL MACHINE ENVIRONMENT

A Project

by

Anthony J. Lorelli

Spring 2006

APPROVED BY THE INTERIM DEAN OF THE SCHOOL OF GRADUATE, INTERNATIONAL, AND INTERDISCIPLINARY STUDIES:

_____

Susan E. Place, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

_____

Ralph Hilzer, Chair

_____

Benjoe Juliano, Ph.D.

ACKNOWLEDGMENTS

I would like to thank the members of my graduate committee, Prof. Ralph Hilzer and Prof. Benjoe Juliano. I valued their continuing support while I worked on this project.

This document would still be just an idea without the help of my wife, Emily, and my children, Elizabeth and Jacob. I deeply appreciate their frequent encouragement.

As I wrote, I often thought of my parents and grandparents. My grandfather and grandmother, Joseph and Lettie McGie, both received degrees from Chico Normal School in 1930. My parents attended California State University, Chico as well. My mother, Lynnette McGie, received a Bachelor's degree in 1966 and a teaching credential in 1967. My father, John Lorelli, received a Bachelor's degree in 1973 and a Master's degree in 1977. I hope this work is a worthwhile addition to their legacy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

IMPLEMENTING CONTINUATIONS FOR A VIRTUAL MACHINE ENVIRONMENT

by

Anthony J. Lorelli

Master of Science in Computer Science

California State University, Chico

Spring 2006

Scheme is a multiparadigm programming languages that lacks an extensive standard library. Virtual machines such as the Java Virtual Machine and Common Language Infrastructure are attractive application development platforms with large standard libraries. Implementing Scheme for these virtual machines is difficult because Scheme features first-class continuations. Virtual machine security restrictions prevent the use of several traditional continuation implementation strategies.

This project investigates the best means of implementing first-class continuations for a virtual machine platform. Nine continuation implementation strategies are surveyed. Five of these strategies are implemented using a Scheme interpreter built for the Common Language Infrastructure. The five strategies are simple exception, heap, frame recycling, stack reconstruction, and thread. The three criteria used to judge these

implementations are benchmark performance, ease of implementation, and integration with the target platform. Stack reconstruction is shown to be the preferable implementation based on these criteria.

CHAPTER I

INTRODUCTION

Virtual machines such as Sun's Java Virtual Machine (JVM) [1] and Microsoft's Common Language Infrastructure (CLI) [2] are attractive platforms for application development. They feature large standard libraries that include extensive support for frequent application needs, such as TCP/IP networking and database access. However, the most common programming languages that target these runtime environments, such as Java, C#, and Visual Basic.NET, are object-oriented in focus, rendering the use of other popular programming idioms awkward. Scheme, a dialect of Lisp, is a multiparadigm programming language that supports a variety of common programming idioms, like functional and logic programming, but lacks an extensive standard library [3]. Therefore, creating a Scheme interpreter or compiler that targets the JVM or CLI is a logical objective. The virtual machine (VM) developer gains support for common programming idioms beyond object orientation from Scheme, while the Scheme user gains access to the resources provided by the VM. However, a full implementation of Scheme requires support for first-class continuations. This is problematic because the operational requirements of first-class continuations do not always align well with the runtime execution model assumed by the VM. To explore this problem space, this project constructs model realizations of common continuation implementation strategies. Each

strategy is then critiqued to see which is the most effective general case solution and, by extension, the best means of providing a feature-complete Scheme implementation.

Foundational Knowledge

Before discussing the investigative scope of this project, a foundational review of the core concepts is presented.

Continuations

Continuations are control abstractions that represent the remainder of the current computation. They are referred to as first-class values in a programming language if they can be passed to functions as parameters, returned from functions as values, or stored in data structures. Languages with first-class continuations can implement a wide variety of features that are often built directly into an interpreter or runtime library [4], [5], [6], [7], [8]. In addition, continuation-passing style (CPS) is a heavily-studied representation strategy sometimes used as an intermediate form in compilers [9].

With such broad applicability, continuations have been a popular topic in programming language research for decades [10]. Scheme has included first-class continuations since its creation at MIT in the 1970s [11]. Usage and implementation of continuations are common topics in Scheme-related research [7], [12].

Virtual Machines

Virtual machines are a software representation of an abstract computer, complete with its own memory model and instruction set architecture. The VM can be implemented for multiple operating systems and hardware architectures. Therefore, an application that targets the VM architecture, instead of the lower-level OS or hardware on

which the VM runs, more easily gains portability across supported VM platforms. In addition, the VM enforces conditions on program execution that decrease the ability to maliciously affect other applications running on the same computer.

The CLI is the VM used in the implementation portion of this project. The main reason is that a production-quality CLI implementation is available for the desktop as both commercial and open-source distributions. Microsoft, the original developer of the CLI, makes it freely available for its Microsoft Windows operating system [13]. The Mono Project [14] distributes an open-source version that works on most UNIX-like operating systems. None of the open-source JVM implementations targeting the desktop environment are as fully-realized as Mono.

Investigative Scope

A review of common continuation implementations identifies appropriate strategies for use in the CLI environment. One common strategy copies, clears, and reinstalls the runtime call stack. Many early implementations of Scheme and Smalltalk, another programming language that supports first-class continuations, use this method [15], [16], [17]. Java and C# implementations of Scheme often use exceptions to model continuations because programmatic manipulation of the stack area of memory is not allowed. Exceptions are an error-handling mechanism present in both languages. They are used to implement continuations because their behavior is equivalent to a single-use continuation used as an escape procedure. Generally, such a strategy is limited to modeling upward, or escaping, continuations. However, recent work indicates that the exception handling facility can be used to implement all possible forms of continuations

by making a logical model of the stack and reconstructing it on demand [18], [19]. Threads are an alternative way of implementing continuations in any language whose thread system meets a reasonable set of requirements [20].

Allocating call frames on the heap instead of the stack avoids the necessity of stack modification. Then the language depends on the system garbage collection facility to free the allocated storage once the frames are no longer used [21], [22]. Various optimizations reduce the overhead generated by allocating call frames on the heap. One strategy recycles previously-allocated frames [12]. There are also hybrid heap/stack techniques that achieve excellent general-case performance, because they add no overhead for programs that do not capture a continuation [23].

Having gained a perspicacious view of what continuations are, what they are good for, and how they are traditionally modeled, a means of modeling continuations on the CLI is developed. Before doing so, however, a system is needed to judge the efficacy of each strategy. Strategies that perform well in common application usage scenarios are most important because the primary impetus in providing a Scheme interpreter or compiler for the CLI is to make available a multiparadigm programming language for application development. Two common application-level uses of continuations provide simple benchmarks. In addition, several benchmarks are provided that more narrowly exercise continuation capture and continuation invocation, which are the two primary continuation operations in Scheme.

A simple Scheme interpreter provides a foundation upon which the merits of the various strategies mentioned above are tested. The benchmarks used for measuring

continuation performance also prescribe exact language features and library functions to be included with the test bed interpreter.

The three criteria used to evaluate the implementation strategies are ease of implementation, integration with the CLI runtime environment, and benchmark performance. In some cases, a preliminary review of these criteria eliminates a strategy before it is implemented. For instance, strategies that directly modify the stack area of memory integrate poorly with the CLI runtime environment because stack modification is disallowed by the CLI security architecture. Others, such as those based on exception handling, integrate well with the CLI, but their benchmark performance depends heavily on CLI performance characteristics.

Significance

A limitation-free implementation of first-class continuations that interacts well with the JVM or CLI runtime environment is a recognized objective. Several recent implementation techniques have been developed for just this purpose [18], [19]. The closeness to this goal may be judged by surveying the feasibility of traditional implementation techniques, as well as more recent techniques such as [18] and [19]. Should it be met, other languages featuring first-class continuations could conceivably benefit from access to CLI resources. Examples include Smalltalk, Standard ML, Haskell, and Ruby. Building an interpreter or compiler for these languages is simplified if one or more worthy continuation implementation strategies are found. Beyond this practical benefit, there is satisfaction derived from solving thorny problems, such as how

to engineer a solution to a complex problem under constraints that prevent use of the most obvious answer.

Limitations

Five continuation strategies are implemented in Chapter V. Each uses the Scheme interpreter presented in Chapter IV as its foundation. A compiler-based implementation of the same five strategies would provide useful performance data, but building a compiler is beyond the scope of this paper. On a related note, programs that target the CLI are compiled down to an intermediate representation similar to assembly language. The CLI calls this format Common Intermediate Language (CIL). High-performance interpreters, such as IronPython (an implementation of the programming language Python for the CLI) compile the most frequently used procedures in a program to CIL [24]. While this is an intriguing technique for optimizing interpreter performance, runtime code generation is left as a potential enhancement. In addition, the implemented subset of Scheme is confined to the functionality necessary to run the continuation benchmark suite. This excludes certain language features required by the language standard. For instance, arithmetic is limited to integers since that is the extent required by the benchmarks.

Conventions

The following conventions are used:

- The name of a class, function, or variable and source code examples are written in a monospaced font. For example, the `Interpreter` class contains the `Evaluate` method. Executable source code is set off from the main text by indentation.

- Ellipses (…) in a source code example indicate the presence of unrelated material that has been omitted for clarity.
- The > character is the command line prompt of the interpreter. The interpreter response to input appears on subsequent lines. The following example shows the result of typing an arithmetical expression at the command prompt of a Scheme interpreter.

```
> (+ 5 7)
12
```

Project Content

The project is organized as follows. Chapter II explores the nature of continuations and their use in Scheme. The most common continuation implementation strategies found in the literature are surveyed. Chapter III presents a series of benchmarks based on common application-level uses of continuations and the design of Scheme continuation objects. Chapter IV presents a simple Scheme interpreter. Chapter V presents CLI-compatible implementations of five strategies presented in Chapter II and individually evaluates them based on the benchmarks in Chapter III. Chapter VI summarizes the project and, in conclusion, compares individual strategies to determine the most attractive overall strategy.

CHAPTER II

CONTINUATIONS

A continuation is a control abstraction that represents the remainder of the current computation. It is difficult to infer from such an abstract definition how continuations are used and why they are useful. To clarify both issues, this chapter presents common continuation uses, progressing from simple arithmetical examples to complex control structures. These examples then provide context for a discussion of common continuation implementation strategies.

What is a Continuation?

Programming languages that do not feature first-class continuations sometimes have structures that are analogous to implicit continuations managed by the language runtime. For instance, function invocations are tracked by means of function call frames stored in the stack area of memory in languages like C. These pending function call frames represent the work remaining to be done in any given program. Consider a naive implementation of the factorial function in C.

```
int factorial(int n) {
    return (n < 2) ? 1 : n * factorial(n-1);
}
```

If this `factorial` function is called with an argument of 5, a backlog of pending multiplications builds until the condition `n < 2` is met.

```
factorial(5)

   ( 5 * ) factorial(4)
   ( 5 * 4 * ) factorial(3)
   ( 5 * 4 * 3 * ) factorial(2)
   ( 5 * 4 * 3 * 2 * ) factorial(1)
   ( 5 * 4 * 3 * 2 * 1 )
```

On each line, the portion in the first set of parentheses represents the current control context, or continuation, of the computation. The C runtime environment encodes control context as a stack of function call frames to which the programmer is not given convenient access. In Scheme [3] and Standard ML of New Jersey (SML/NJ) [25], however, continuations are first-class structures. This project will focus on examples that use continuations in Scheme.

Scheme represents continuations as functions of a single argument. These functional representations are created by means of the constructor function `call-with-current-continuation` (`call/cc`) from Scheme's standard library. `Call/cc` takes a single-argument function as its only parameter. In return, `call/cc` calls the function and passes as its argument an object representing the current continuation. For example,

```
(* 3 (+ 4 1))
```

evaluates to 15. However,

```
(call/cc
  (lambda (k)
    (* 3 (k (+ 4 1)))))
```

evaluates to 5 because the continuation captured by the call to `call/cc` returns to the top level of control. The multiplication is superfluous. Control always escapes past it when the continuation object `k` is invoked on the result of the addition (`k` is a common continuation variable name in Scheme).

Continuations can also capture a particular series of operations. In the example

```
(* 3 (call/cc
       (lambda (k)
         (+ 4 (k 1)))))
```

control escapes the addition and resumes the pending multiplication, resulting in a final value of 3. It is tempting to save a reference to such a continuation and treat it as a named procedure. However, choosing a method to accomplish this must be done carefully. For instance, unexpected consequences will result if assignment is used to modify an existing name binding.

```
(define mult-by-three #f)

(* 3 (call/cc
       (lambda (k)
         (set! mult-by-three k)
         (k 3))))

(mult-by-three
  (mult-by-three 3))
```

First, add the name `mult-by-three` to the global namespace and bind it to an arbitrary value. Next, begin a multiplication expression. The first operand is the integer 3. For the second operand, capture the current continuation, a pending multiplication by three, with `call/cc`. Modify the binding of `mult-by-three` in the function passed to `call/cc` to refer to the captured continuation. The continuation object is then invoked on the integral value 3, resulting in a value of 9 for the complete expression. In the following expression, the continuation is reused, once again passing it 3 as an argument. The first call to `mult-by-three` is nested within a second call to the same function, which implies that the expression value will be 27. However, the final value of

the second expression is also 9 because the captured continuation occurs at the top level of control. Invoking it causes control to return to that level.

Scheme's interface for capturing continuations, `call/cc`, is partly to blame for how easy it is to make this mistake [26]. The use of `call/cc` only specifies the endpoint of the continuation to capture (i.e. the continuation active when `call/cc` is invoked). The starting point for the continuation is implied by context. Enough must be known about Scheme's execution model to deduce where this might be. Also, the function-style interface for continuation objects implies that continuations are composable, as many functions are. The example above shows that this is false. Subcontinuations, a kind of composable continuation, attempt to resolve these two issues with traditional Scheme continuations [27].

Why Are Continuations Useful?

First-class continuations can model features that are often built directly into the interpreter or runtime library of a language that does not expose continuations to the programmer. Such features include coroutines [4], exceptions [5], threads [6], nonblind backtracking [7], and nondeterministic computations [8]. This project concentrates on exceptions and coroutines because they are the most generally applicable. Threads are left unexplored because there are complicated issues involved in how continuation-based thread implementations interact with `dynamic-wind`, a required Scheme library function introduced in the latest Scheme specification [28]. Nonblind backtracking and nondeterministic computation are useful within specific domains, but are omitted because they are not widely applicable.

More recently, continuations have provided a new model for structuring web applications that abstracts away many of the protocol-level details exposed in common web application frameworks [29]. The Seaside framework [30], written in Smalltalk, is a notable example of how continuations are used to implement the application logic of a production-quality web application. Demonstrating these techniques requires a working HTTP implementation, which is beyond the scope of this paper.

Exceptions

Exceptions tie error handling logic to specific error conditions and signal when those conditions have occurred. C++, C#, and Java, among others, provide the statements `try`, `catch`, and `throw` as language-level facilities for manipulating exceptions.

- `try` introduces a block in which exception handling may be used.
- `catch` binds a specific error type with an error handler.
- `throw` signals that a particular type of error has occurred and transfers control to its lexically-nearest handler, clearing the stack of call frames until the handler is found.

For example, a division routine that protects against division by zero is implemented in C++ as follows.

```
#include <string>
#include <iostream>

class DivisionException {
private:
    std::string _msg;
public:
    DivisionException(const std::string& msg) :
        _msg(msg) {}
    std::string GetMessage() const { return _msg; }
};
```

```
int SafeDiv(int lhs, int rhs) {
    if ( rhs == 0 ) {
        throw DivisionException(
            "Error: division by zero.");
    }

    return lhs / rhs;
}

int main(int argc, char* argv[]) {
    try {
        std::cout << "Result: " <<
            SafeDiv(3, 0) << std::endl;
    } catch ( const DivisionException& divEx ) {
        std::cerr << divEx.GetMessage() << std::endl;
        return 1;
    }

    return 0;
}
```

The placement of `try` within `main` indicates that exception handling is in effect. `SafeDiv` uses `throw` to signal that an exception has occurred. Control passes from the `throw` statement back up the call stack to the nearest `catch` block specified for `DivisionException`.

Scheme defines no explicit exception handling features at the language level. However, a simple `try/catch/throw` facility is added using continuations and Scheme's support for creating new syntactic forms, also known as macros. The implementation is derived from the `raise/with-handler` facility [31]. In its current form, it is limited to one `catch` clause, but in principle an arbitrary number of `catch` clauses could be specified.

```
(define *handlers* '())

(define-syntax try
  (syntax-rules ()
    ((try exp (catch name handler))
     (lambda ()
       (call/cc
        (lambda (k)
          (let ((saved-handlers *handlers*))
            (set! *handlers* (cons (list name k handler)
                                   *handlers*))
```

```
            exp
            (set! *handlers* saved-handlers))))))))

(define throw
  (lambda (exception)
    (let ((handler-record (assq exception *handlers*)))
      (if handler-record
        (let ((error-k (cadr handler-record))
              (error-handler (caddr handler-record)))
          (error-k (error-handler exception)))
        "Error"))))
```

The implementation of `try` begins by capturing the current continuation with an invocation of `call/cc`. Control returns to the continuation after the exception is handled. The exception name and handler specified as arguments in the `catch` clause are added to the start of a global list of exception handlers, along with the captured continuation. The implementation of `throw` performs a linear search through the global `*handlers*` list to find the handler, extracts the handler and escape continuation saved with it, and passes the value generated by the handler to the continuation.

Coroutines

Coroutines are similar to subroutines that can suspend their own execution at any point, either by resuming another coroutine or by yielding control back to their caller. Knuth [32] argues that subroutines are a special form of coroutine. They are a useful abstraction that can profitably model many producer/consumer and parallel processing scenarios [33], though they have not been included as a feature in many recent languages. As with exceptions, Scheme does not include coroutines in the base language, but they can be modeled with continuations.

There are distinct similarities between coroutines and subcontinuations (the composable type of continuation mentioned above [27]). De Moura, Rodriguez, and Ierusalimschy [34] claim that asymmetric coroutines are functionally equivalent to

subcontinuations. Thus, in a notable instance of conceptual recursion, it may be possible to model first-class continuations in a language that already features asymmetric coroutines.

Characterizing coroutines as functions that can be paused and later resumed necessarily implies that they have some means of capturing their current state when they yield control back to their caller or to another coroutine. The current continuation is captured and saved when the coroutine yields control, then invoked when the coroutine is resumed. The implementation of coroutines in Scheme [4] is surprisingly short. In it, a constructor function creates new coroutine instances. The macro system defined by R5RS is hygienic [3], which means that any new name bindings introduced within the body of a macro are guaranteed not to capture or shadow a binding that exists outside of the macro definition. However, the coroutine constructor macro introduces a `resume` function that must be visible outside of the macro definition itself. Therefore, `define-macro`, a non-standard but widely available means of defining non-hygienic macros in Scheme, is used.

```
(define-macro coroutine
  (lambda (x . body)
    '(letrec ((local-control-state
               (lambda (,x) ,@body))
              (resume
               (lambda (c v)
                 (call/cc
                  (lambda (k)
                    (set! local-control-state k)
                    (c v))))))
       (lambda (v)
         (local-control-state v)))))
```

The coroutine consists of two functions. One represents the current state of the coroutine itself, `local-control-state`; the other, `resume`, pauses the current coroutine by resuming another coroutine, a reference to which is passed as the first argument of the

function. When `resume` is called, it assigns the current continuation to `local-control-state`, erasing the reference to the previous start state of the coroutine; this continuation is invoked if the coroutine in question is resumed later. Benchmarks that use both continuation-based exceptions and coroutines are provided in Chapter III.

Implementation Strategies

This section presents nine continuation implementation strategies. Some strategies map continuations onto an existing implementation language feature with similar behavior. Others manipulate function call frames in various ways, depending on the design objectives of the strategy. Strategies of both types are presented, beginning with the former.

Continuation-Based Continuations

Assuming that a complete implementation of Scheme is available for the target platform, the most straightforward implementation of continuations is a simple mapping from target language to implementation language. Target language continuations become proxies for implementation language continuations.

While this may seem unlikely, the scenario gains plausibility if the existing Scheme implementation is a compiler and the desired implementation is an interpreter. There is, in fact, a Scheme interpreter that fits this scenario. Scheme48 [35] is implemented using Pre-Scheme, a subset of Scheme that is translated into C for efficient compilation [36]. While Pre-Scheme does not include first-class continuations, Kelsey [36] describes plans to add continuations to the language. Such an arrangement might

also be possible using an alternate implementation language with first-class continuations, such as SML/NJ or Smalltalk.

As a potential downside, the design trade offs made in the implementation cannot be controlled, rendering performance unpredictable.

Exceptions

Continuations are commonly used to implement exceptions. By the same token, exceptions, such as those used in the `try/catch/throw` facility described above, behave like a one-shot continuation used as an escape procedure, which makes them natural for modeling continuations.

There are two common variations in how exceptions are used to model continuations. In the simplest, each continuation operation corresponds directly to an exception operation. Capturing a continuation causes a new context to be marked through the introduction of a new `try/catch` block; each continuation invocation corresponds to a `throw` operation. JScheme [37] and Kawa [38], both Java implementations of Scheme, implement continuations in this manner.

A more complicated technique is discussed in Sekiguchi, Sakamoto, and Yonezawa [18] and Pettyjohn, Clements, Marshall, Krishnamurthi, and M. Felleisen [19]. Instead of introducing a single demarcation point through the use of one `try/catch` pair, each function call performed is separated into its own exception handling context with a new `try/catch`. Capturing a continuation uses `throw` to send an exception object back through each of these handlers, making a logical copy of the stack and clearing the call frames currently present. When a continuation is invoked, special resumption logic uses the captured stack data to rebuild the previous context.

Continuation capture invokes the same resumption logic since performing the capture clears the stack of its contents.

The simple exception strategy has a straightforward implementation, but is limited to modeling upward continuations that escape from their current context and travel up the call stack. The stack reconstruction variation encompasses all possible continuation types, but its implementation is significantly more complicated. The need to separate all operations into atomic steps and introduce variables to hold temporary values where necessary suggests that, as an implementation strategy, it is more appropriate for a source-to-source translation solution. Both Sekiguchi, Sakamoto, and Yonezawa and Pettyjohn, Clements, Marshall, Krishnamurthi, and M. Felleisen [19] assume the technique will be used by a compiler.

Threads

Threads are another potential application of continuations. Therefore, as with exceptions, it is appropriate that threads provide a vehicle for modeling continuations. Kumar, Bruggeman, and Dybvig [20] show how they can model subcontinuations in any language whose thread system meets a minimum set of requirements. Feeley [39] applies the technique as a means of facilitating interoperability between Scheme and C, using one of the popular C-based thread APIs, such as POSIX threads or Win32 threads.

Thread-based continuations are limited to a single invocation unless the thread system allows for creating an exact clone of an existing thread. The usefulness of such an implementation is also limited by the speed with which the thread system allocates and initializes new threads, and by the number of threads that may be allocated at any given time.

Stack

To track function calls, languages such as C and Java create implicit continuations in the form of function call frames in the stack area of memory. As a function is entered, a new frame is allocated and pushed onto the stack. As the function exits, the corresponding call frame is discarded. If direct access to the stack area of memory is allowed, these continuations can be made explicit by copying all current call frames to the heap when a continuation is captured and restoring them to the stack when the continuation is invoked. Many early implementations of Scheme and Smalltalk use the stack strategy [15], [16], [17].

The indirect costs of the strategy are minimal. Programs that never capture a continuation pay no price and have the advantage of deterministic allocation and deallocation of their call frames. The direct costs are more substantial. A stack-to-heap copy is incurred for each capture and a heap-to-stack copy is incurred for each invocation. The strategy is unworkable in the CLI and JVM environments because they deny arbitrary manipulation of the stack.

Heap

In the heap strategy, all call frames are allocated on the heap and deallocated by a garbage collection (GC) algorithm. The increased amount of heap allocation and time spent performing GC raise the indirect cost of the strategy, but the direct cost of continuation capture and invocation are low. Capture retains a reference to the current call frame; invocation updates the frame pointer to reference that captured frame instead of its current value. Several implementations of Scheme and Smalltalk have used the

heap strategy [21], [22]. SISC, a Scheme interpreter for the JVM, is a recent example [40].

One common optimization of the heap strategy recycles call frames, lowering the total number of frames allocated [12].

Stack/Heap

In the hybrid stack/heap strategy for implementing continuations, capturing a continuation copies all current call frames stored on the stack into the heap. Unlike the pure stack technique, however, the stack is then cleared to insure that each call frame exists in only one location. Also, when a continuation is invoked, the frame pointer is simply updated to point into the heap. In a variation on the stack/heap technique that [12] calls the incremental stack/heap strategy, each frame is copied back to the stack just before it is needed. Several Smalltalk implementations have used the standard stack/heap strategy [12]. Scheme48 uses the incremental strategy [35].

CHAPTER III

EVALUATION CRITERIA

This chapter develops evaluation criteria for continuation implementations. The three criteria discussed are execution speed, ease of implementation, and integration with the host environment.

To judge execution speed, this chapter presents a series of benchmarks designed to simulate performance in common usage scenarios. Two benchmarks are created that use the coroutine and exception implementations demonstrated in Chapter II. They provide realistic models of how continuations might be manipulated by users of the language. However, the variety of actions performed in these tests makes it difficult to isolate the performance difference due to the continuations alone. Also, due to their inherent limitations, not all continuation implementations execute both the coroutine and exception examples. Therefore, several targeted benchmarks are provided that more specifically exercise Scheme's representation of continuations.

The quickest program imaginable is useless if it returns the wrong result. This implies correctness is also a desirable attribute. Given that code complexity is correlated with a higher error rate [41], a simpler implementation aids correctness. The two criteria for ease of implementation are implementation size and resemblance to familiar programming models.

The CLI is an attractive application development platform because its resources increase programmer productivity. These resources include extensive standard libraries, development tools such as debuggers and profilers, and VM-level security services. Giving Scheme access to these resources is a primary motivation for this project, so a strategy that facilitates this access is preferred. The extent to which this criterion applies depends on the language implementation type. An interpreter like the example in Chapter IV gains access to all available platform resources because it is written in one of the standard VM application implementation languages, such as C#. Programs written in the language it interprets can access the standard library, perhaps using a calling convention like JScheme's Java reflector variables [37], but they are not directly profiled or debugged by the platform development tools. They also fall outside the coverage of the VM security manager, except to the extent the interpreter operations are covered. On the other hand, a compiler that generates the VM intermediate form potentially gains access to all available resources for the language it implements, not just itself. This is only true, however, if the compiled programs use the native call stack, since most of the resources mentioned expect to find runtime data encoded there.

Benchmarks

The remainder of the chapter discusses performance benchmarks. To run each benchmark, the convention in Clinger, Hartheimer, and Ost [12] is followed and the function `run-benchmark` from Gambit-C [42] is used. In order, its arguments are the benchmark name, the number of times to run the benchmark, the function that starts the

benchmark running, and a predicate function to test whether the benchmark completes successfully. A sample run of the `invoke-k` benchmark presented below looks like

```
(run-benchmark "INVOKE-K" 1 (lambda () (invoke-k 10000))
               (lambda (result) (eq? result 'done)))
```

Exceptions

Chapter II presents a Scheme implementation of exception handling. The `SafeDiv` function from that chapter is reimplemented in Scheme as `safe-div` to create a new benchmark. The `test-driver` function creates an exception handling block and calls the division function. The combination of arguments used guarantees the generation of an exception.

```
(define safe-div
  (lambda (x y)
    (if (= y 0) (throw 'divide-by-zero) (/ x y))))

(define test-driver
  (try
   (let loop ((dividend 10))
     (let ((decr-div (- dividend 1)))
       (safe-div dividend decr-div)
       (loop-decr div)))
   (catch 'divide-by-zero
          (lambda (exception) "Divide-by-zero error"))))
```

Running `test-driver` causes the error handler specified in the `catch` clause to execute.

```
> (test-driver)
"Divide-by-zero error"
```

Coroutines

A simple producer/consumer scenario is provided as an application of the coroutine implementation in Chapter II. One coroutine produces the Fibonacci sequence.

The producer coroutine passes each newly calculated value to a consumer coroutine. The consumer coroutine appends the value to a list.

```
(define make-fibo-coroutine
  (lambda (limit consumer-cor)
    (coroutine no-init
      (let fibo* ((curr 1) (prev 0))
        (let ((new-val (+ curr prev)))
          (if (> new-val limit)
              (resume consumer-cor 'done)
              (begin
                (resume consumer-cor new-val)
                (fibo* new-val curr)))))))))
```

The producer signals its completion by passing the symbol `done` to the consumer.

```
(define make-list-coroutine
  (lambda (producer-cor)
    (coroutine no-initial-val
      (let loop ()
        (let ((val (resume producer-cor 'get-next-val)))
          (if (eq? 'done val)
              '()
              (cons val (loop)))))))))
```

A short driver function integrates the producer and consumer.

```
(define get-fibo-list
  (lambda (limit)
    (letrec ((fibo-cor
              (make-fibo-coroutine limit
                (lambda (v) (list-cor v))))
             (list-cor
              (make-list-coroutine
                (lambda (v) (fibo-cor v)))))
      (list-cor 'start))))
```

The driver returns the list generated by the consumer.

```
> (get-fibo-list 2000)
(1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597)
```

Capture and Invocation

Continuation capture and continuation invocation are the two actions of first-class continuations in Scheme. Continuations are captured with each call to `call/cc`. Continuations are invoked when the continuation object created by `call/cc` is applied

to an argument. Since it is difficult to isolate the changes in the performance of these operations in the exception and coroutine benchmarks, three additional benchmarks are provided. The first balances its use of captures and invocations, the second focuses on captures, and the third focuses on invocations.

The Takeuchi function (TAK) is a traditional benchmark for Lisp systems [12], [40], [43]. The first benchmark uses a modified version of TAK first presented in [43].

```
(define tak
  (lambda (x y z)
    (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y)))))
```

Function calls dominate any use of TAK. For instance, the traditional call of `(tak 18 12 6)` performs 63,609 function calls and 47,706 subtractions [43].

As of yet, TAK makes no explicit use of continuations. However, [43] also presents a modified version of TAK, called CTAK, that uses an exception handling facility to emulate continuations. With the amendments to CTAK in Clinger, Hartheimer, and Ost [12], each return operation explicitly uses a dedicated continuation to create a more useful benchmark.

```
(define ctak
  (lambda (x y z)
    (call/cc
      (lambda (k)
        (ctak+k k x y z)))))

(define ctak+k
  (lambda (k x y z)
    (cond ((not (< y x))
           (k z))
          (else
           (call/cc
             (lambda (k)
               (ctak+k
```

```
                k
                (call/cc
                  (lambda (k) (ctak+k k (- x 1) y z)))
                (call/cc
                  (lambda (k) (ctak+k k (- y 1) z x)))
                (call/cc
                  (lambda (k)
                    (ctak+k k (- z 1) x y)))))))))
```

Instead of implicitly returning the value of `z`, it is explicitly passed as the argument to a continuation. Reusing the arguments from TAK, the call `(ctak 18 12 6)` generates 63,609 continuation captures and 47,707 continuation invocations.

The next benchmark exercises continuation capture. It applies `call/cc` an arbitrary number of times without invoking the captured continuation object.

```
(define capture-k
  (lambda (limit)
    (letrec ((capture-k*
              (lambda (n)
                (if (< n limit)
                    (begin
                      (call/cc (lambda (k) 'ignored))
                      (capture-k* (+ n 1)))
                    'done))))
      (capture-k* 0))))
```

Tail-recursive function calls substitute for iteration. Each invocation of `call/cc` provides an anonymous function that does no real work as its argument. The continuation object passed as the function parameter `k` is ignored.

The final benchmark exercises continuation invocation. In it a continuation simulates a `goto` statement in order to emulate a `while` loop.

```
(define invoke-k
  (lambda (limit)
    (let ((n 0)
          (loop 0))
      (call/cc (lambda (k) (set! loop k)))
      (if (< n limit)
          (begin (set! n (+ n 1))
                 (loop 'ignored))
          'done))))
```

`Set!` updates `loop` with the value of the continuation object created by `call/cc`. This is the only continuation capture performed. The continuation is invoked the number of times specified by `limit`. See Appendix A for the implementation of `run-benchmark` and a contiguous listing of all benchmarks.

CHAPTER IV

SCHEME

This chapter presents an implementation of Scheme [11], a popular dialect of Lisp created in the 1970s at MIT by Guy Steele and Gerald Sussman. Notable features include first-class functions, macros, and first-class continuations. The language specification has been revised multiple times, including ratification as an IEEE standard [44]. The most recent revision R5RS [3]. In this chapter, an interpreter is implemented for a subset of the language defined by R5RS. The interpreter is the foundation for the five continuation implementations discussed in Chapter V.

Scheme is a multiparadigm programming language. It supports functional, imperative, logic, and object-oriented programming, either by means of its core forms or by means of a library [8], [45], [46]. Scheme programs are composed of expressions. Expressions are either atomic or complex. Atomic expressions are literal values, such as integer or string literals, or variable references. Complex expressions are special forms composed from one or more subexpressions.

Existing Implementations

While the Scheme defined by R5RS is no longer as simple as that defined in Sussman and Steele [11], the conceptual clarity of the design has encouraged numerous implementations over the past three decades, with a large range of target platforms and

implementation languages [9], [17], [21], [35], [42]. JVM and CLI are the target for multiple Scheme implementations that vary in usefulness and conformance to the language standard [37], [38], [40], [47]. Most of these target JVM because it has been available for a longer time. Nonetheless, the architecture of both platforms is similar enough that the design decisions made for a JVM-compatible implementation are also relevant for one targeting the CLI. Of the implementations mentioned, JScheme [37] and SISC [40] are the most relevant comparisons, since both generate an abstract syntax tree (AST), and then walk its nodes to perform an evaluation. The interpreter in this chapter adopts a similar approach. Kawa [38] and Bigloo.NET [47] are compilers that generate executables using the intermediate bytecode representation of the target platform.

One major difference between SISC and JScheme is AST evaluation. Both construct a direct in-memory representation of a program's s-expressions using pair-based lists. SISC then generates a customized representation of the s-expressions using an `Expression` class hierarchy, similar to the Interpreter pattern in Gamma, Helm, Johnson, and Vlissides [48]. Each `Expression` class contains an `eval` method that performs the evaluation logic specific to that type of expression. JScheme evaluates the s-expressions directly. The two Scheme implementations also differ in how they represent and manage continuations. SISC uses the heap strategy with the frame recycling optimization, whereas JScheme uses the simple exception strategy.

Scanning and Parsing

The grammatical structure of Scheme is simple enough that hand-written scanners and parsers are often used instead of a scanner or parser generation tool such as

lex or yacc. For instance, both JScheme and SISC use a hand-written scanner and parser. The regular, self-similar structure of the language facilitates this. As with most forms of Lisp, all complex expressions use fully-parenthesized prefix notation. These expressions are generally called symbolic expressions (s-expressions), a name that comes from the original definition of Lisp [49]. S-expressions are a textual representation of a program's AST. Parentheses and whitespace are the only expression delimiters defined in R5RS. There are no operators apart from the quotation shorthand notation, which is generally expanded into an equivalent function call when encountered by the scanner and parser.

All complex expressions have the same structure. They begin with an open left parenthesis, followed by the function or special form name. The function name is followed by zero or more parameters, depending on the arity of the function. Parameters are delimited by whitespace. The expression concludes with a right parenthesis.

Literal values create the majority of the lexical complexity in the language. R5RS defines literal forms for integers, rational numbers, complex numbers, characters, Boolean constants, string constants, and vectors (similar to arrays in other languages). For simplicity, rational and complex numbers are omitted, since the benchmarks in Chapter III only require support for integers.

The input, the scanner, the parser, and the interpreter have a producer/consumer relationship. The scanner divides the input into lexical tokens. The parser checks the order of the tokens and assembles a convenient in-memory representation of the expressions. The interpreter analyzes this in-memory representation to produce the program result.

The `Scanner` class represents the scanner. The class constructor takes a reference to an instance of `TextReader`.

```
using System.IO;

public class Scanner {
    private TextReader _reader;

    public Scanner(TextReader reader) {
        _reader = reader;
    }

    public Token NextToken() { … }
}
```

The `Token` class returned by `NextToken` pairs the token text with a value that represents the token type.

```
public class Token {
    private readonly string _text;
    private readonly TokenType _type;

    public Token(string text, TokenType type) {
        _text = text; _type = type;
    }

    public string Text { get { return _text; } }

    public TokenType Type { get { return _type; } }
}
```

The `Parser` class represents the parser. `Parser` includes an instance of `Scanner` as a member variable and builds a tree-based representation of the sequence of tokens generated by calls to that instance's `NextToken` method.

```
public class Parser {
    private Scanner _s;

    public Parser(Scanner s) { _s = s; }

    public object Read() {
        return _Read(_s.NextToken());
    }

    private object _Read(Token t) {
        switch (t.Type) { … }
    }
}
```

Parsing Atoms

`Parser.Read` chooses the representation for all recognized literal constants. Strings and integers are mapped to a reasonable CLI equivalent. Booleans, characters, and vectors require functionality not provided by the nearest CLI type. For these, new classes implement the desired behavior. `Read` examines the type enumeration value contained in the token instance to determine the lexeme type.

```
private object Read(Token t) {
    switch (t.Type) {
        …
        case TokenType.String:
            return t.Text;
        case TokenType.Integer:
            return long.Parse(t.Text);
        case TokenType.Boolean:
            return Boolean.Create(t.Text);
        case TokenType.Character:
            return new Character(t.Text);
        case TokenType.Vector:
            return List.ToVector((Pair)Read());
        …
    }
}
```

The parser also chooses an appropriate representation for variables names. Keywords and variable names are both instances of what R5RS calls symbols. Symbols are immutable, globally-unique strings. Two symbols are considered identical if spelled the same way. A dedicated `Symbol` class represents symbols within the interpreter. A simple symbol factory class enforces the uniqueness requirement.

```
public class Symbol {
    private readonly string _str;

    public Symbol(string str) { _str = str; }

    public override string ToString() { return _str; }
}

public class SymbolFactory {
    private static System.Hashtable _symTable;

    static SymbolFactory() {
        _symTable = new System.Hashtable();
```

```
    }

    public static Symbol Create(string str) {
        if (_symTable.Contains(str)) {
            return (Symbol)_symTable[str];
        } else {
            Symbol s = new Symbol(str);
            _symTable[str] = s;
            return s;
        }
    }
}

public class Parser {
    …
    private object Read(Token t) {
        switch (t.Type) {
            …
            case TokenType.Symbol:
                return SymbolFactory.Create(t.Text);
            …
        }
    }
    …
}
```

Parsing Lists

Scheme programs can also be viewed as lists of values. Modifying these lists prior to evaluating them provides the basis for macros. Scheme lists are composed of pairs of references, such as those constructed by the library function `cons`. For instance, the in-memory representation of a conditional expression such as `(if (eq? a b) c d)` is shown in Figure 1.

The value `()` represents the empty list, a special value used as a list terminator.

`Parser` represents complex expressions with a simple `Pair` class.

```
public class Pair {
    public static readonly Pair EmptyList;

    private object _head;
    private object _tail;

    static Pair() {
        EmptyList = new Pair(EmptyList, EmptyList);
    }
```

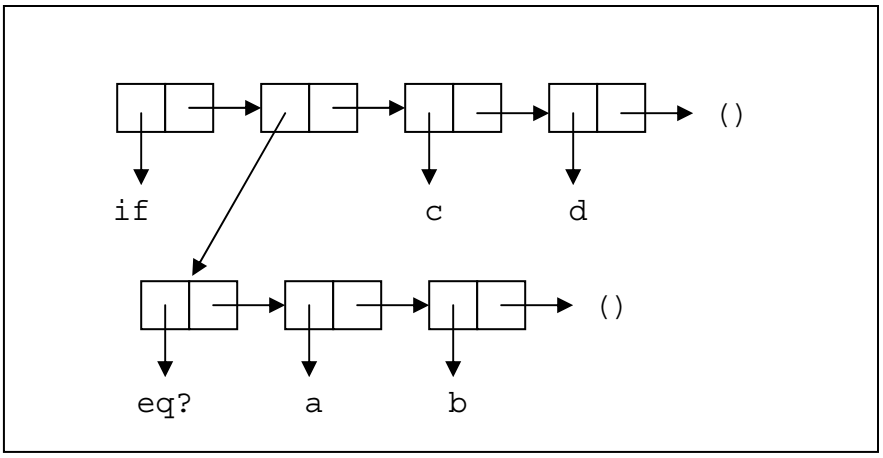

Fig. 1. List representation of conditional expression using pairs.

```
        public Pair(object head, object tail) {
            _head = head; _tail = tail;
        }

        …
    }

    public class Parser {
        …
        private object Read(Token t) {
            switch (t.Type) {
                …
                case TokenType.LeftParen:
                    return ReadList();
                …
            }
        }

        private object ReadList() {
            Token t = _s.NextToken();
            switch (t.Type) {
                case TokenType.RightParen:
                    return Pair.EmptyList;
                default:
                    return new Pair(Read(t), ReadList());
            }
        }
        …
    }
```

The result is an AST representation of each complex expression. The root node is evaluated to provide the function value. The child nodes are evaluated to provide the function arguments.

Evaluating Expressions

An interpreter is now implemented to evaluate the expression representations produced by the parser. Each instance of `Interpreter` is applied to a text stream and includes an instance of `Parser` as a member variable. When called, its `Evaluate` method returns the value of each successive Scheme expression contained in the referenced input stream.

```
using System.IO;

public class Interpreter {
    private Parser _parser;

    public Interpreter(TextReader reader) {
        _parser = new Parser(new Scanner(reader));
    }

    public object Evaluate() {
        return Evaluate(_parser.Read());
    }

    public object Evaluate(object expr) { … }
}
```

`Evaluate` must determine whether the value returned by the parser represents an atomic or complex expression. The simplest method is to test the type of variable returned. It is complex if it is an instance of `Pair`. It is atomic if not.

```
…
public object Evaluate(object expr) {
    if (expr is Pair) {
        // complex
    } else {
        // atomic
    }
}
…
```

Atomic Expressions

The two types of atomic expressions are literal constants and variable references. The evaluation policy for literal constants is simple. Since the parser already

created the CLI-native representation of the value, that representation is returned as the value of the expression.

To evaluate a variable reference, the value bound to the name being referenced is retrieved and returned as the value of the expression. Environments associate names with values. The interpreter uses them to create a new association or modify an association that already exists. The `Environment` class represents an environment.

Certain expressions, such as `lambda`, establish new name bindings in the current environment. The bindings introduced by `lambda` are valid within the body of the expression and invisible outside of it. In addition, one `lambda` can be used as a subexpression inside another, in which case both sets of bindings are accessible. For instance, in

```
(lambda (a)
   (lambda (b) (+ a b)))
```

the second `lambda` forms the body expression of the first. Both name bindings are visible because of this lexical nesting, allowing the second `lambda` to refer to both variables. This implies that the environment implementation must have some way of creating a distinct set of bindings with each new name binding expression used. At the same time it retains the ability to reach the bindings established in the surrounding environment. This behavior is modeled by creating a chain of environments. Each new environment maintains a link back to its parent environment. This parent link is always the null value for the outermost global environment. Environments associate names with values by means of a hashtable that maps the symbol name to the specified value.

```
using System.Collections;
```

```
public class Environment {
    private Hashtable _symTable;
    private Environment _parentEnv;

    public Environment(Environment parentEnv) {
        _parentEnv = parentEnv;
        _symTable = new Hashtable();
    }

    public Environment() {
        _parentEnv = null;
        _symTable = new Hashtable();
    }

    public object Lookup(Symbol s) {
        if (_symTable.Contains(s)) {
            return _symTable[s];
        } else if (_parentEnv != null) {
            return _parentEnv.Lookup(s);
        } else {
            return null;
        }
    }
}
```

The `Evaluate` method in `Interpreter` is now expanded. An instance of `Environment` is added to serve as the global environment for variable bindings.

```
public class Interpreter {
    private Environment _globalEnv;
    private Parser _parser;

    public Interpreter(TextReader reader) {
        _globalEnv = new Environment();
        _parser = new Parser(new Scanner(reader));
    }

    public object Evaluate() {
        return Evaluate(_parser.Read(), _globalEnv);
    }

    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            // complex
        } else {
            if (expr is Symbol) {
                // variable reference
                object val = env.Lookup((Symbol)expr);
                if (val == null) {
                    throw new System.Exception(
                        "No value associated with id " +
                        expr
                    );
                } else {
                    return val;
```

```
                }
            } else {
                // literal constant
                return expr;
            }
        }
    }
}
```

Complex Expressions

Complex expressions are made up of one or more subexpressions. The five basic types are conditional, abstraction, quotation, assignment, and function application.

Conditionals. Conditional expressions in Scheme have the form

```
(if condition consequent alternative)
```

The `alternative` expression is optional. If it is omitted and the condition evaluates to false, the return value of the `if` expression is unspecified. This functionality is added to `Interpreter` using the ternary conditional expression in C#.

```
…
public object Evaluate(object expr, Environment env) {
    if (expr is Pair) {
        Pair p = (Pair)expr;
        object head = p.Head;
        object tail = p.Tail;

        if (head == SymbolCache.IF) {
           return Boolean.IsTrue(
               Evaluate(List.First(tail)), env) ?
                   Evaluate(List.Second(tail), env) :
                   (List.Length(tail) == 3) ?
                       Evaluate(List.Third(tail), env) :
                       null;
        } else {
            …
        }
    }
    …
}
…
```

The `Boolean` class models Boolean values in Scheme. The only Scheme value that evaluates to false is `#f`, the Boolean false constant. All other values evaluate to true. `Boolean.IsTrue` maps Scheme Boolean values to C# Boolean values.

```
public class Boolean {
    public static readonly Boolean TRUE = new Boolean();
    public static readonly Boolean FALSE = new Boolean();

    private Boolean() { }

    public static Boolean Create(string s) {
        return (s.ToLower() == "#t") TRUE : FALSE;
    }

    public static Boolean Create(bool b) {
        return (b) ? TRUE : FALSE;
    }

    public static bool IsTrue(object o) {
        return (o == FALSE) ? false : true;
    }

    public override string ToString() {
        return (this == TRUE) ? "#t" : "#f";
    }
}
```

Abstraction. The abstraction expression, `lambda`, creates new functions. `Lambda` returns an anonymous function value when used by itself. `Lambda` creates named procedures when used in conjunction with a name binding expression such as `define`. The functions created by `lambda` are often called closures because they are said to *close over* their environment. In other words, closures retain a reference to the environment in which they are created, allowing them to retrieve the value of any free variable they include. (A free variable is a variable not mentioned in the function argument list.) In the example

```
(define make-accumulator
  (lambda (n)
    (lambda (i)
      (set! n (+ n i))
      n)))

(define acc (make-accumulator 10))

> (acc 1)
11
> (acc 1)
12
```

the inner `lambda` retains the ability to refer to the argument `n` used in the outer `lambda`. It can also update that binding with a new value.

The `Closure` class models closures. The interpreter creates an instance of this class when it evaluates a `lambda` expression.

```
public class Closure {
    private Pair _args;
    private Pair _expr;
    private Environment _env;

    public Closure(Pair args, Pair expr,
        Environment env) {
        _args = args; _expr = expr; _env = env;
    }

    …
}

public class Interpreter {
    …
    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            …
            } else if (head == SymbolCache.LAMBDA) {
                return
                    new Closure((Pair)List.First(tail),
                        (Pair)List.Rest(tail), env);
            }
            …
        }
        …
    }
    …
}
```

Both the sequence of expressions forming the body of the closure and the environment in which the closure is created are saved within the instance of `Closure` created by the interpreter. They are used when the function is applied to a set of arguments.

Assignment. `Set!` assigns a new value to a name that already exists in the current environment. For example, after the interpreter executes

```
(define a 100)
(set! a 200)
```

the variable `a` has the value 200, not 100. The value returned by `set!` is unspecified.

```
public class Environment {
    …
    public bool Assign(Symbol s, object v) {
        if (_symTable.Contains(s)) {
            _symTable[s] = v;
            return true;
        } else if (_parentEnv != null) {
            return _parentEnv.Assign(s, v);
        } else {
            return false;
        }
    }
    …
}

public class Interpreter {
    …
    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            …
            } else if (head == SymbolCache.SET) {
                Symbol id = (Symbol)List.First(tail);
                if (!env.Assign(id,
                    Evaluate(List.Second(tail), env))) {
                    throw new System.Exception(
                        "Cannot set undefined symbol " +
                        id
                    );
                }
            }
            …
        }
        …
    }
    …
}
```

`Set!` only applies to preexisting name bindings. `Define` introduces new name bindings without using `lambda`. The implementations of `define` and `set!` are similar. The primary difference is that if the current environment lacks a binding that uses the specified name, the interpreter creates a new binding for that name instead of signaling an error. The return value of `define` is unspecified.

```
public class Environment {
    …
    public void Define(Symbol s, object v) {
        if (Lookup(s) != null) {
            Assign(s, v);
        } else {
            _symTable[s] = v;
        }
```

```
    }
    …
}

public class Interpreter {
    …
    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            …
            } else if (head == SymbolCache.DEFINE) {
                env.Define((Symbol)List.First(tail),
                    Evaluate(List.Second(tail), env));
                return null;
            }
            …
        }
        …
    }
    …
}
```

Quotation

Scheme expressions can be viewed as executable code, or they can be viewed as data, values to be manipulated by other Scheme expressions. Quotation controls whether an expression is treated as code or data. An expression is treated as data when it is quoted using the `quote` expression. Specifically, when an expression is quoted, the return value is the in-memory representation of that expression built by the parser. For instance, when the variable `a` is referred to unquoted, the interpreter dereferences the variable and return the associated value. Quoting `a` returns the variable name as a first-class symbol value.

```
> (define a 10)
> a
10
> (quote a)
a
```

The implementation of `quote` is trivial.

```
public class Interpreter {
    …
    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            …
            } else if (head == SymbolCache.QUOTE) {
```

```
                return List.First(tail);
            }
            …
        }
        …
    }
}
```

Function Application. The interpreter is now given the ability to invoke functions. Function applications use prefix notation as with all complex expressions in Scheme. The function invoked is specified by the value at the start of the expression list. Any Scheme expression whose evaluation produces a function value is allowed. Functional arguments are evaluated before the function is applied.

The two types of functions the interpreter supports are the closures created by `lambda` expressions and the native functions created directly in the implementation language. Native functions perform actions not directly expressible in Scheme. A `Procedure` interface is introduced that allows the interpreter to treat both types of functions in the same way.

```
public interface Procedure {
    public object Apply(Interpreter i, Pair args);
}
```

The final type of complex expression is added to the interpreter evaluation loop.

```
public class Interpreter {
    …
    public object Evaluate(object expr,
        Environment env) {
        if (expr is Pair) {
            …
            } else {
                Procedure proc =
                    (Procedure)Evaluate(head, env);
                Pair args = EvalList((Pair)tail, env);
                object result = proc.Apply(this, args);
                return result;
            }
        }
        …
    }
```

```
    public Pair EvalList(Pair p, Environment env) {
        return (p != Pair.EmptyList) ?
            new Pair(Evaluate(p.Head, env),
                EvalList((Pair)p.Tail, env)) : p;
    }
    …
}
```

Arguments are passed to the procedure as a list of values. Closures map this list onto their list of argument names, then pass control back into the interpreter with the closure environment as the operative context.

```
public class Closure : Procedure {
    …
    public object Apply(Interpreter i, Pair args) {
        Environment closureEnv =
            new Environment(_args, args, _env);
        return i.Evaluate(_expr, closureEnv);
    }
}
```

Native functions are similar, but generally do not call an interpreter method when they execute. Consider `cons`, the pair constructor.

```
public class Cons : Procedure {
    public object Apply(Interpreter i, Pair args) {
        return new Pair(List.First(args),
            List.Second(args));
    }
}
```

Macros

The Scheme standard specifies other complex expressions, but they can be reduced to some combination of the five basic expressions described above. For instance, `let` expressions are used to introduce new local name bindings.

```
(let ((x 3)
      (y 4)
      (z 5))
  (+ x (+ y z)))
```

Given the obvious utility of such a construct, why doesn't the interpreter explicitly support it? In a sense, it *does* because all `let` expressions can be translated into equivalent `lambda` expressions. The example above becomes

```
((lambda (x y z) (+ x (+ y z))) 3 4 5)
```

Such transformations are implemented using macros. If Scheme expressions are treated as data, that data is manipulated in the same way any Scheme value is manipulated. In other words, the AST corresponding to a particular expression is transformed into an entirely different AST.

Efficient and correct implementation of macros is a complex subject and a frequent topic in Scheme-related research [50], [51], [52]. The benchmarks presented in Chapter III require their use, but their implementation is tangential to the continuation implementation strategies that are the focus of this project. As a result, the interpreter uses the portable macro engine provided by Dybvig and Waddell [53]. See Appendix A for details on its integration into the interpreter.

CHAPTER V

IMPLEMENTING CONTINUATIONS

This chapter presents five CLI-compatible continuation implementations. Each implementation is followed by a discussion of its direct and indirect execution costs and a table of benchmark results. Strategies incompatible with VM security restrictions, such as the stack and stack/heap methods, are omitted.

Complete source code for each interpreter is available in Appendix A. Counting the number of lines present in the source code files for the five implementations produces the results shown in Table 1.

TABLE 1
LINES OF SOURCE CODE IN CLI-COMPATIBLE CONTINUATION IMPLEMENTATIONS

Implementation Strategy	Lines of Source
Simple Exception	1776
Heap	2133
Frame Recycling	2207
Stack Reconstruction	2605
Threads	2220

The two CLI implementations used to conduct the performance tests are the .NET Framework 2.0 from Microsoft [13] and Mono 1.1.13 from the Mono Project [14]. They are installed on the computer systems described in Table 2.

TABLE 2
BENCHMARK SYSTEMS

CLI Version	Operating System	Processor	RAM
.NET Framework 2.0	Windows XP SP2	AMD Athlon 1 GHz	256 MB
Mono 1.1.13	Mac OS X 10.4.6	Motorola PowerPC G4 1.25 GHz	512 MB

The benchmarks from Chapter III are managed using `run-benchmark`. The following parameters are used.

```
(run-benchmark "exception" 1000
               (lambda () (test-driver)) #t)
(run-benchmark "coroutine" 1000
               (lambda () (get-fibo-list 2000)) #t)
(run-benchmark "tak" 20
               (lambda () (tak 18 12 6)) #t)
(run-benchmark "ctak" 20
               (lambda () (ctak 18 12 6)) #t)
(run-benchmark "capture-k" 1000
               (lambda () (capture-k 100)) #t)
(run-benchmark "invoke-k" 1000
               (lambda () (invoke-k 100)) #t)
```

Exceptions

Chapter II discusses using exceptions to model continuations. The two variations implemented are the simple exception method and the stack reconstruction method. The first, in which continuation capture and invocation directly correspond to exception handling operations, is presented here. The second, in which exception handling facilitates construction of a logical model of the call stack, is presented later in this chapter.

In the simple exception method, capturing a continuation corresponds to the introduction of a new exception handling block using `try`. Invoking a continuation corresponds to raising an exception using `throw`.

```
public class CallCC : Procedure {
```

```
    public object Apply(Interpreter i, Pair args) {
        ContinuationException contEx =
            new ContinuationException();
        Continuation k = new Continuation(contEx);
        Procedure p = (Procedure)List.First(args);
        try {
            return p.Apply(i,
                new Pair(k, Pair.EmptyList));
        } catch (ContinuationException kEx) {
            if (kEx == contEx) {
                return kEx.ReturnValue;
            } else {
                throw;
            }
        }
    }
}
```

The continuation object passed to the procedure called by `call/cc` must also act as a procedure, so it implements the same `Procedure` interface as `Closure` and the native library procedures, such as `CallCC`. When invoked, it throws the instance of `ContinuationException` created in `CallCC.Apply` as a way of passing its argument back up the call stack. The `catch` clause checks for identity between the two instances of `ContinuationException` in order to allow nested applications of `call/cc`.

When `Continuation.Apply` is called, a `throw` statement transfers control back to the point in the call stack marked by the original `try` block.

```
public class Continuation : Procedure {
    private ContinuationException _contEx;

    public Continuation(ContinuationException contEx) {
        _contEx = contEx;
    }

    public object Apply(Interpreter i, Pair args) {
        _contEx.ReturnValue = List.First(args);
        throw _contEx;
    }
}
```

`ContinuationException` is used instead of the more generic `System.Exception` because it contains a reference to the value of the continuation argument.

```
public class ContinuationException : System.Exception {
    private object _val;

    public ContinuationException() { _val = null; }

    public object ReturnValue {
        get { return _val; }
        set { _val = value; }
    }
}
```

The simple exception method is attractive from an execution cost standpoint. There are no indirect costs because other than the implementation of `CallCC`, `Continuation`, and `ContinuationException`, the interpreter is not modified. Programs that do not use continuations pay no cost for the interpreter's ability to use them. The direct costs are also limited. Capturing a continuation creates an exception handling block. Invoking a continuation instigates a `throw` operation that clears a portion of the native call stack. If nested continuations are used, the interpreter performs a linear search through the registered exception handlers until the originating handler is found. The overall speed of the simple exception method, then, depends largely on the speed with which the virtual machine establishes a new exception handling context and clears the stack when an exception is thrown. Simple exception benchmark results are shown in Tables 3 and 4. Times are given in microseconds. Smaller times are better. The value "NA" in the Time column indicates that the benchmark did not complete, so a time is not available.

TABLE 3
MONO BENCHMARK RESULTS FOR THE SIMPLE EXCEPTION STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,240
Coroutine	NA
TAK	2,030,568
CTAK	5,035,791
Capture-k	4,121
Invoke-k	NA

TABLE 4
.NET 2.0 BENCHMARK RESULTS FOR THE SIMPLE EXCEPTION STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,492
Coroutine	NA
TAK	1,406,522
CTAK	8,602,870
Capture-k	5,658
Invoke-k	NA

Predictably, the simple exception strategy succeeds in benchmarks requiring the partial erasure of the current call stack, but fail in those requiring complete erasure of the call stack and the restoration of an alternative set of call frames.

Even though it does not use continuations, the TAK benchmark is included for two reasons. First, it indicates how the current implementation strategy affects function call speed when continuations are not used. Second, it is a useful point of comparison for the performance of the CTAK benchmark, which emulates a function returning a value to its caller by creating and invoking continuations.

Heap

The heap method allocates call frames on the heap instead of the stack. The simple exception method implicitly uses the C# call stack to maintain the context of any given expression evaluation. In order to implement the heap method, the context data previously kept on the stack must be programmatically accessible and capturable at any given point.

Interpreter evaluation methods are registerized to make interpreter state explicit [54]. Registerization converts functions from using stack-allocated parameters to using a shared set of register variables [54]. `Evaluate` and `EvalList` are the two interpreter methods transformed. The arguments of `Evaluate` are an object reference representing the current expression to be evaluated and a reference to the current environment. The arguments of `EvalList` are a reference to the list of parameters being evaluated in preparation for the application of a Scheme function and a reference to the current environment. Both methods also return the evaluation result.

The three distinguishable method parameters are collected and reinterpreted as shared registers in a `Frame` class. The current environment parameter serves the same purpose in each method, so it is replaced by a single register. A fourth register is included to represent the return value. A reference to another instance of `Frame` is included as a fifth register to act as a return address once the current expression evaluation is complete. `Frame` represents the current state of the evaluation at any given point.

```
public class Frame {
    public object Expr;
    public Environment Env;
    public object Value;
    public Pair ValueRib;
    public Frame Context;
```

```
}
```

The interpretation strategy presented in Chapter IV uses the C# call stack whenever it is necessary to evaluate an expression within a context. For instance, to evaluate a conditional expression, `Evaluate` recursively calls itself to evaluate the condition. Evaluation of the consequent or alternative waits until the condition evaluation completes.

```
…
return Boolean.IsTrue(Evaluate(List.First(tail), env)) ?
    Evaluate(List.Second(tail), env) :
    Evaluate(List.Third(tail), env);
…
```

Now that evaluation state is encoded in instances of `Frame`, such an approach is infeasible. Instead, all evaluation logic is converted into operations that use the data available in current frame registers. The interpreter allocates additional instances of `Frame` to preserve context.

To facilitate this change, the interpreter adds a compilation step between parsing and evaluating the input. Once the macro expander completes its pass on the s-expression abstract syntax tree (AST) generated by the parser, the compiler transforms the AST into a graph of instructions that operate on registers. The instruction set used is similar to the set presented in the heap implementation portion of Dybvig [55].

The instruction set consists of the following:

- `(conditional consequent alternative)` If the current value in the `Value` register is false, the alternative expression is placed in the `Expr` register. Otherwise, the consequent expression is placed in `Expr`.

- `(closure args expr next)` Executing the `Closure` instruction creates an instance of the `Closure` class using `args` as its argument list and `expr` as its body. The instance is placed in the `Value` register. The `Expr` register is updated with `next`.

- `(assign id next)` The current value in the `Value` register is assigned to the symbol specified by `id`. An error is raised if that symbol does not have a preexisting binding in the current environment. The `Expr` register is updated with `next`.

- `(define id next)` A new name binding is created in the current environment using the symbol specified by `id` and the current value in the `Value` register. The `Expr` register is updated with `next`.

- `(variable id next)` The value associated with the variable specified by `id` is retrieved from the environment and placed in the `Value` register.

- `(constant expr next)` The constant value `expr` is placed directly in the `Value` register.

- `(context appexpr next)` The frame pointer is updated with a new instance of `Frame` allocated in anticipation of a function call. The new frame `Expr` register is given the value `appexpr`. The new `Context` register refers to the frame instance active when the instruction is executed. The original frame `Expr` register is updated with `next`.

- `(argument next)` The contents of the `Value` register are appended to the list of values contained in the `ValueRib` register.

- `(apply)` The contents of `Value` are treated as an implementer of the `Procedure` interface and applied to the list of values in `ValueRib`.
- `(end)` If `Context` refers to a valid instance of `Frame`, the contents of `Value` are copied into the `Context` frame `Value` register. If there are no more valid frames, the contents of `Value` are returned as the result of the evaluation.

The heap strategy easily implements proper tail recursion. If the next instruction following an `apply` is an `end`, then the value of the entire expression becomes the result of the application. There is no need to issue a `context` instruction for allocating a new call frame. This technique for detecting tail position is taken from Dybvig [55].

The heap method interpreter iterates over the frame registers instead of recursively evaluating the subtrees and leaves of the AST. It processes and updates the contents of the `Frame` registers until the final `end` instruction is encountered and the contents of the `Value` register are returned. As an example of instruction set use, consider a simple conditional expression.

```
(if #t 1 2)
```

The interpreter from Chapter IV finds the `if` symbol, recursively calls `Evaluate` to determine the value of the condition expression, and then chooses to evaluate either the consequent or the alternative based on the result. In contrast, compiling the expression above into the heap method instruction set produces the following structure.

```
(constant #t
  (conditional (constant 1 (end))
               (constant 2 (end))))
```

Note that the evaluation of the condition expression occurs before the interpreter encounters the `conditional` instruction. When the `conditional` instruction is executed the value of the expression is already available in the `Value` register.

Capturing and invoking continuations are elegant operations in the heap architecture. To capture the current continuation, the implementation of `call/cc` need only copy the current value of the `Context` register.

```
public class CallCC : Procedure {
    public void Apply(Interpreter I, Pair args) {
        i.Context.ValueRib =
            List.Create(
               new Continuation(i.Context.Context)
            );
        i.Context.Expr = InstructionCache.Apply;
        i.Value = List.First(args);
    }
}
```

Similarly, to restore the continuation, the continuation procedure copies its argument to the `Value` register and the captured `Frame` reference to the `Context` register.

```
public class Continuation : Procedure {
    private Frame _context;

    public Continuation(Frame context) {
        _context = context;
    }

    public void Apply(Interpreter i, Pair args) {
        i.Context = new Frame(List.First(args),
            InstructionCache.End, i.Context);
        i.Context.Context = _context;
    }
}
```

The heap method costs are the inverse of the simple exception method costs. The heap method establishes a new execution architecture, in which the current state of the computation is managed in terms of the shared register variables that belong to `Frame`. As a result, even programs that do not use continuations incur overhead caused by this explicit interpreter state management. On the other hand, continuation capture and

invocation are simpler and depend less on the performance characteristics of the virtual machine.

Running the benchmarks against the heap implementation produces the following results (Tables 5 and 6).

TABLE 5
MONO BENCHMARK RESULTS FOR THE HEAP STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,699
Coroutine	5,824
TAK	5,193,533
CTAK	7,148,289
Capture-k	7,084
Invoke-k	5,611

TABLE 6
.NET 2.0 BENCHMARK RESULTS FOR THE HEAP STRATEGY

Benchmark	Time (Microseconds)
Exceptions	520
Coroutine	2,403
TAK	1,601,302
CTAK	2,490,080
Capture-k	2,513
Invoke-k	1,832

Unlike the simple exception method, the heap strategy implementation completes all six benchmarks.

Frame Recycling

Frame recycling is an optimization for the heap strategy. Frames that have not been captured in a continuation are reused, which in turn reduces the total number of frames allocated and the amount of memory reclaimed by the garbage collector. To implement the optimization, `Frame` adds a private Boolean variable indicating whether the frame has been captured in a continuation. When a continuation is captured, the interpreter marks all the frames currently on the call stack, setting this `_captured` flag to true.

```
public class Frame {
    private bool _captured;
    …
    public void Mark() {
        _captured = true;
        if (Context != null) {
            Context.Mark();
        }
    }
    …
}

public class Continuation : Procedure {
    …
    public Continuation(Frame context) {
        _context = context;
        _context.Mark();
    }
    …
}
```

The frame recycling interpreter allocates new frames by means of a factory class instead of allocating frames directly. The factory class returns a previously allocated frame if one is available on the free list it maintains. It creates and returns a new instance of `Frame` if no free frames are available. For instance, the evaluation of the `constant` instruction using `FrameFactory` to enact the frame allocation policy looks like

```
…
if (instr == Instruction.Constant) {
    _context = FrameFactory.Create(
        List.Create(_context.Expr),
```

```
            List.Third(_context.Expr), _context);
    }
    …
```

The `Create` method takes a reference to the new contents of the `Value` and `Expr` registers, as well as a reference to the previous instance of `Frame`. `FrameFactory` adds this instance of `Frame` to the free list if it is not captured in a continuation.

```
public class FrameFactory {
    private static System.Collection.Queue _freeList;
    …
    public static Frame Create(object val,
        object expr, Frame context) {
        Frame f;
        if (_freeList.Count > 0) {
            f = (Frame)_freeList.Dequeue();
            f.Expr = expr;
            f.Value = val;
            f.ValueRib = context.ValueRib;
            f.Env = context.Env;
            f.Context = context.Context;
        } else {
            f = new Frame(val, expr, context);
        }

        if (!context.Captured) {
            _freeList.Enqueue(context);
        }

        return f;
    }
}
```

`Interpreter` now calls `FrameFactory.Create` instead of directly allocating frames.

The interpretation methodology is unchanged. Only the frame allocation policy is different. The benchmark results are as follows (Tables 7 and 8).

Using Mono, the increase in performance is dramatic for such a small change, particularly in the TAK and CTAK benchmarks. Average TAK performance improves by 41%. Average CTAK performance improves by 31%. Unfortunately, .NET 2.0 runtime

TABLE 7
MONO BENCHMARK RESULTS FOR THE FRAME RECYCLING STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,155
Coroutine	4,711
TAK	3,683,330
CTAK	5,425,516
Capture-k	5,628
Invoke-k	3,864

TABLE 8
.NET 2.0 BENCHMARK RESULTS FOR THE FRAME RECYCLING STRATEGY

Benchmark	Time (Microseconds)
Exceptions	630
Coroutine	2,283
TAK	1,673,406
CTAK	2,515,617
Capture-k	3,034
Invoke-k	2,032

performance is slightly worse in everything but the coroutine test, indicating that the cost of recycling frames outweighs the cost of reducing object allocations.

Stack Reconstruction

Most stack-related continuation implementation strategies depend on the ability to modify the call stack. CLI and JVM security restrictions make such approaches untenable. However, it is possible to make a logical copy of the call stack and reconstruct

it on demand by means of an artful manipulation of the VM exception handling facility [18], [19].

The stack reconstruction interpreter maintains the compilation step from the heap-based interpreter. However, the instruction set no longer requires explicit inclusion of the next instruction now that the native stack is available for maintaining context. The `context` instruction is omitted since native call frames replace instances of `Frame`. The stack-based instruction set is as follows:

- `(apply expr args)` The procedure that results from evaluating `expr` is applied to the list of values that results from evaluating `args`.
- `(argument first rest)` The result of evaluating `first` is prefixed to the list of values generated by evaluating `rest`. The resulting list is returned to the caller.
- `(assign id expr)` The result of evaluating `expr` is assigned to the symbol `id` in the current environment. The specified symbol must already have a name binding established. An exception is raised if it does not.
- `(closure args body)` A closure is allocated and returned. `Args` specifies the formal arguments. `Body` specifies the body expression.
- `(conditional expr texpr fexpr)` If the result of evaluating `expr` is false, `fexpr` is executed and its value returned. Otherwise, `texpr` is executed and its value returned.
- `(constant value)` The constant `value` is returned.

- `(define id expr)` The result of evaluating `expr` is associated with the symbol `id` in the current environment.
- `(sequence first rest)` The `first` instruction is executed. The result of executing `first` is returned if `rest` refers to the empty list. Otherwise, the result is discarded and the `rest` instruction is executed.
- `(variable id)` The value bound to the symbol `id` in the current environment is returned. If no binding exists, an exception is raised.

The execution of each instruction creates a series of `try` blocks, depending on how many separate control contexts the execution creates. If a continuation is captured, an exception is thrown back to the root of the call stack. As it passes through the exception handling block created for each instruction, a logical representation of the work remaining to be done for that instruction is constructed and passed to the exception. By the time it reaches the root of the stack, it has collected a complete logical representation of all active call frames.

Consider the evaluation of the `conditional` instruction.

```
public class Interpreter {
    …
    public object Execute(object expr, Environment env) {
        object instr = List.First(expr);
        …
        } else if (instr == Instruction.Conditional) {
            object cond;
            try {
                cond = Execute(List.Second(expr), env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new CondContext(this,
                    List.Third(expr),
                    List.Fourth(expr), env));
                throw;
            }

            object condResult =
                Execute(Boolean.IsTrue(cond) ?
                    List.Third(expr) :
```

```
                    List.Fourth(expr), env);
            return condResult;
        }
        …
    }
    …
}
```

`Interpreter` creates a new `CondContext` instance if it catches a `CaptureException` while it evaluates the condition expression. `CondContext` retains references to the consequent, alternative, and environment and is invoked later to execute the remainder of the computation.

The interpreter uses the stack model that belongs to the instance of `CaptureException` to immediately resume the current continuation so that `call/cc` can apply its argument. `Call/cc` passes the procedure to be applied to the instance of `CaptureException` before it is thrown.

```
public class CallCC : Procedure {
    public object Apply(Interpreter i, Pair args) {
        throw new CaptureException(
           (Procedure)List.First(args));
    }
}
```

`Evaluate` traps each `CaptureException` thrown during an evaluation. It also catches `InvokeException`, the exception thrown when a continuation is invoked.

```
…
private Evaluate(object expr, Environment env) {
    for (;;) {
        try {
            return Dispatch(expr, env);
        } catch (CaptureException capEx) {
            _capEx = capEx;
        } catch (InvokeException invEx) {
            _invEx = invEx;
        }
    }
}
…
```

`Evaluate` reestablishes its exception handlers after it catches a capture or invocation exception so that the next capture or invocation is also handled correctly.

`Dispatch` resumes a continuation if a continuation-holding exception was caught in the `Evaluate` exception trap. Otherwise, the standard expression evaluation logic is used.

```
…
private object Dispatch(object expr, Environment env) {
    if (_capEx != null) {
        CaptureException capEx = _capEx;
        _capEx = null;
        if (capEx.Continuation == null) {
            capEx.AddContext(new HaltContext());
        }
        Continuation k =
            new Continuation(capEx.Continuation);
        return ResumeFromCapture(capEx.Continuation,
            capEx.Procedure, k);
    } else if (_invEx != null) {
        InvokeException invEx = _invEx;
        _invEx = null;
        return ResumeFromInvoke(invEx.Continuation,
            invEx.ReturnValue);
    } else {
        return Execute(expr, env);
    }
}
…
```

`ResumeFromCapture` and `ResumeFromInvoke` have the same structure as the `Resume` method shown in Pettyjohn, Clements, Marshall, Krishnamurthi, and M. Felleisen [19]. First, they travel back up the list of contexts built by the exception as it traveled through the native call stack. Then they call the `Invoke` method of the top context and return the result to the next context. `ResumeFromCapture` adds an additional context to the top of the list to account for the application of the function passed as an argument to `call/cc`.

```
…
private object ResumeFromCapture(ContextList cList,
    Procedure proc, Continuation k) {
    object returnValue;
```

```
        if (cList == null) {
            try {
                returnValue =
                    proc.Apply(this, List.Create(k));
            } catch (CaptureException capEx) {
                capEx.AppendContext(cList);
                throw;
            }
        } else {
            returnValue = ResumeFromCapture(cList.Newer,
                proc, k);
        }

        try {
            return cList.Current.Invoke(returnValue);
        } catch (CaptureException capEx) {
            capEx.AppendContext(cList.Older);
            throw;
        }
    }
    …
```

The capture process must be handled differently if another continuation is captured during the execution of the current `Context.Invoke`. The current continuation can be appended in its entirety to the context list built by the `CaptureException` since it is already available in the list of contexts.

Both the direct and indirect costs of the stack reconstruction method are complex. Programs that do not use continuations still incur continuation management overhead because the interpreter frequently establishes exception handling blocks during the execution of its instruction set. Most instructions create at least one `try` block. Some create two. Continuation capture and invocation are also heavyweight operations in comparison to previous continuation strategies. Capturing a continuation throws an exception that clears the current call stack back to its root, allocating a new instance of `Context` in each exception handler it encounters along the way. The interpreter immediately rebuilds the stack through recursive calls to `ResumeFromCapture`. In most programs, the number of calls to `ResumeFromCapture` roughly corresponds to

the stack size at the time of continuation capture. Invocation is more efficient, because the continuation is already created. No representation of the current context is built before the interpreter reconstructs the stack through recursive calls to `ResumeFromInvoke`.

Running the benchmarks using the stack reconstruction method produces the following results (Tables 9 and 10).

TABLE 9
MONO BENCHMARK RESULTS FOR THE STACK RECONSTRUCTION STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,517
Coroutine	13,350
TAK	2,475,356
CTAK	32,430,465
Capture-k	12,085
Invoke-k	7,653

TABLE 10
.NET 2.0 BENCHMARK RESULTS FOR THE STACK RECONSTRUCTION STRATEGY

Benchmark	Time (Microseconds)
Exceptions	1,832
Coroutine	26,528
TAK	1,641,360
CTAK	52,424,883
Capture-k	23,183
Invoke-k	15,382

The CTAK, capture, and invocation times are significantly slower than the other implementations. However, this is expected, given that the cost of capture and invocation is linear with respect to the depth of the current call stack.

Threads

Kumar, Bruggeman, and Dybvig [20] show how threads implement subcontinuations. Subcontinuations can implement `call/cc`-style continuations, but the implementation is more involved than necessary if the primary target is `call/cc`-style continuations alone.

A simpler implementation may be developed if, following Feeley [39], threads are treated as containers for stack segments. The interpreter allocates a new thread each time an evaluation starts. The thread contains the evaluation call stack. The CLI `Monitor` class synchronizes execution and alerts the caller when a return value is available, either because a continuation was invoked or the last call frame on the stack was reached.

```
private static object Evaluate(object expr,
    Environment env) {
    Executor ex =
        new Executor(List.Create(Instruction.Resume,
            expr), env);
    Thread th = new Thread(new ThreadStart(ex.Execute));

    Monitor.Enter(ex);

    th.Start();

    Monitor.Pulse(ex);
    Monitor.Wait(ex);

    return ex.ReturnValue;
}
```

`Executor` now contains the `Execute` method from `Interpreter`. It is moved there to provide a `Thread`-compatible interface to the interpretation logic (CLI threads start running by calling a `void` method with zero arity). `Execute` uses the same instruction set as the stack reconstruction interpreter. The exception handling blocks used for

capturing contexts are removed. `Execute` interprets the specified expression once it gains the lock on itself released by `Evaluate`.

```
public class Executor {
    private object _expr;
    private Environment _env;
    private object _returnValue;

    public Executor(object expr, Environment env) {
        _expr = expr; _env = env; _returnValue = null;
    }

    public object ReturnValue {
        get { return _returnValue; }
        set { _returnValue = value; }
    }

    public void Execute() {
        Monitor.Enter(this);
        try {
            Execute(_expr, _env);
        }
        …
    }

    public object Execute(object expr, Environment env) {
        …
    }
}
```

The `Resume` instruction is added to allow `Executor` to return a value to its creator.

```
public object Execute(object expr, Environment env) {
    …
    } else if (instr == Instruction.Resume) {
        _returnValue = Execute(List.Seconds(expr), env);

        Monitor.Pulse(this);
        Monitor.Exit(this);

        throw new QuitException();
    }
    …
}
```

`Executor` signals that a value is now available via the `ReturnValue` property by calling `Monitor.Pulse`.

The same treatment of stack segments and return values extends to continuations. A new stack segment (e.g., a new thread) is created and started when

`call/cc` is applied. At the same time, execution of the current stack segment is suspended, allowing the interpreter to capture the continuation currently present in the native call stack. The continuation is invoked by resuming execution of the suspended call stack and ceasing execution of the current thread.

```
public class CallCC : Procedure {
    public object Apply(Executor i, Pair args,
        Environment env) {
        Continuation k = new Continuation();

        Pair newInstr = List.Create(Instruction.Apply,
            List.Create(Instruction.Constant,
                List.First(args)),
            List.Create(Instruction.Argument,
                List.Create(Instruction.Constant, k),
                Pair.EmptyList));

        Executor ex =
            new Executor(List.Create(Instruction.Resume,
                newInstr),
                env);
        k.JoinPoint = ex;
        Thread t =
            new Thread(new ThreadStart(ex.Execute));

        Monitor.Enter(ex);

        t.Start();

        Monitor.Pulse(ex);
        Monitor.Wait(ex);

        return ex.ReturnValue;
    }
}

public class Continuation : Procedure {
    private Executor _ex;

    public object Apply(Executor I, Pair args,
        Environment env) {
        _ex.ReturnValue = List.First(args);

        Monitor.Pulse(_ex);
        Monitor.Exit(_ex);

        throw new QuitException();
    }

    public Executor JoinPoint {
        set { _ex = value; }
    }
}
```

There are several limitations inherent in this strategy. First, the lifetime of the continuation object is limited to the dynamic extent of the call to `call/cc`. Once `call/cc` returns, the suspended parent thread resumes, invalidating the continuation. Second, continuations captured in this manner can only be used once. After a thread resumes, it modifies the call stack, either by returning from functions or calling new ones. Third, the `Monitor` class can only release locks held by the current thread. If there are nested calls to `call/cc`, a continuation object from one of the parent `call/cc` applications cannot be used, because `Continuation.Apply` attempts to release the lock on an instance of `Executor` belonging to a previous thread.

The indirect costs of the thread method are low. Programs that do not use continuations incur the cost of starting the thread that holds the initial stack segment and the synchronization cost of returning the value from that thread to its parent. However, these are one-time events during the execution of the program. The direct execution costs are significantly higher. Each continuation capture spawns a new thread and suspends the current one. Each continuation invocation exits the current thread and resumes the suspended continuation thread. The same is true each time `call/cc` returns a value.

Running the benchmarks using the thread method produces the following results (Tables 11 and 12).

While the TAK results are acceptable for both runtimes, all other results are poor. The CTAK results under Mono are notably degenerate, which illustrates the dependence of the thread strategy on the performance of the underlying thread system. The coroutine benchmark fails because a thread attempts to release a lock it does not

own. The invocation benchmark fails because it attempts to invoke the same continuation more than once.

TABLE 11
MONO BENCHMARK RESULTS FOR THE THREAD STRATEGY

Benchmark	Time (Microseconds)
Exceptions	2,638
Coroutine	NA
TAK	2,538,211
CTAK	1,331,550,487
Capture-k	140,565
Invoke-k	NA

TABLE 12
.NET 2.0 BENCHMARK RESULTS FOR THE THREAD STRATEGY

Benchmark	Time (Microseconds)
Exceptions	4,005
Coroutine	NA
TAK	1,711,961
CTAK	53,486,910
Capture-k	70,851
Invoke-k	NA

CHAPTER VI

SUMMARY AND CONCLUSIONS

Summary

Scheme is a multiparadigm programming language that lacks an extensive standard library. JVM and CLI are attractive application development platforms with large standard libraries, but the common languages that target them focus on object-oriented programming. Implementing Scheme for JVM or CLI is difficult, however, because Scheme features first-class continuations. There is no consensus on how to implement continuations for these platforms.

First-class continuations are a worthwhile feature in programming languages because they can implement control structures that are generally built directly into an interpreter or runtime library. Two such control structures, exceptions and coroutines, are presented in Chapter II, followed by nine continuation implementation strategies. A series of six continuation performance benchmarks is developed in Chapter III. The three evaluation criteria established for all continuation implementations are benchmark performance, ease of implementation, and access to CLI resources. A CLI-compatible Scheme interpreter is presented in Chapter IV. This interpreter is used as a foundation for testing continuation implementations. Five CLI-compatible continuation implementations are presented and benchmarked in Chapter V.

Four of the strategies presented in Chapter II are not implemented. Three require operations that are not permitted by JVM or CLI. The final unimplemented strategy, continuation-based continuations, requires the presence of continuations in the implementation language. Since one of the five completed strategies would likely be used to make this possible, it is assumed that continuation-based continuations do not perform better than the other five strategies discussed.

Conclusions

Relative performance must be measured with real-world tests because most continuation implementation strategies depend heavily on the speed of CLI implementation details, such as the speed of memory allocation and method dispatch. Similarly, ease of implementation is demonstrated by building a working example of each strategy. Only potential integration with VM resources can be inferred from the form of the implementation strategy without building a concrete example. Therefore, evaluation criteria developed in Chapter III are applied to the results from Chapter 5 to determine the worthiest continuation implementation strategy.

The first criterion used to judge the five CLI-compatible implementation strategies is benchmark performance. Benchmark results for all strategies are presented in Chapter V. Results are presented for both the Mono and .NET 2.0 CLI implementations. The frame recycling optimization for the heap strategy exhibited the best overall performance. Frame recycling causes a slight performance decrease using the .NET 2.0 runtime. However, the performance increase using Mono is large enough and the percentage decrease under .NET small enough that the change is worthwhile. The simple

exception strategy performs well in benchmarks that favor raw function call speed, presumably because it uses the CLI native call stack. However, its failure to complete two benchmarks for architectural reasons disqualifies it in the face of three other strategies that complete all six tests. Stack reconstruction performance is poor on CTAK, continuation capture, and continuation invocation. Its performance is adequate on the more important TAK, exception, and coroutine benchmarks.

The second evaluation criterion is ease of implementation. The larger the implementation, the more difficult it is to implement, both because of the increased level of detail and the raw time it takes to complete.

Implementation size for each interpreter is presented in Chapter V. The simple exception strategy has the smallest implementation with approximately 350 fewer lines of code than the closest alternative. Once again, its failure to execute all benchmarks disqualifies it from consideration. The same is true for the thread method. Among the three remaining, the two heap methods have a smaller implementation. While the stack reconstruction interpreter has approximately 400 more lines of source code than the heap or frame recycling interpreter, it recursively evaluates its instruction set. This is more familiar to programmers accustomed to programming languages that store control context on a call stack. The two heap methods use an iterative register-based approach that has more in common with processor architecture than programming languages. Consequently, there is no preferable strategy. The implementations of the heap and stack reconstruction strategies are approximately equal in their level of difficulty.

Access to CLI resources is a third criterion discussed in Chapter III. The heap method intrinsically forfeits access to CLI resources other than the standard library

because it manages its own independent set of call frames in the heap. The stack reconstruction method is slower, but it is the only stack-based method that has no limit on the type of continuations it can represent. Therefore, the stack reconstruction method has the greatest potential for CLI integration, particularly when used in conjunction with compiler-based implementations of the desired programming language.

Heap methods are the most attractive from a performance standpoint. Implementing them requires an amount of effort similar to implementation of the stack reconstruction method. The stack reconstruction method is better suited to integration with the host environment. The third criterion is paramount because the project focus is providing a CLI-compatible Scheme implementation. Therefore, stack reconstruction is the preferred method for implementing continuations.

Recommendations

For simplicity, the continuation implementations presented in Chapter V use an interpreter-based approach. It would be advantageous to build a compiler that generates CIL-based representations of Scheme programs using both the heap and stack reconstruction strategies to see whether the same general execution speed relationship still holds between the two.

The structure of the stack reconstruction strategy is suggested by Sekiguchi, Sakamoto, and Yonezawa [18], and Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen [19], but only a few implementation details are provided. As a result, there could well be suboptimal design choices in the current implementation that cause performance to be unnecessarily poor. There are areas where significant amounts of

optimization are possible. For instance, arrays of captured `Context` instances might perform better than the linked lists presented in Chapter V. Any performance improvements further cement stack reconstruction as the preferred implementation technique.

REFERENCES

REFERENCES

[1] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification Second Edition.* Reading, Mass.: Addison-Wesley Professional, 1999.

[2] "Standard ECMA-335: Common Language Infrastructure (CLI)," *ECMA International,* June 2005. Retrieved November 14, 2005 from the World Wide Web: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[3] H. Abelson, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, M. Wand, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams Iv, D.P. Friedman, E. Kohlbecker, G.L. Steele, Jr., and D.H. Bartley, "Revised^5 Report on the Algorithmic Language Scheme," *Higher-Order and Symbolic Computation,* vol. 11, no. 1, pp. 7-105, 1998.

[4] C.T. Haynes, D.P. Friedman, and M. Wand, "Obtaining Coroutines with Continuations," *J. Computer Languages,* vol. 11, no. 3/4, pp. 143-153, 1986.

[5] C. Queinnec, *Lisp In Small Pieces.* Cambridge, UK: Cambridge University Press, 1996.

[6] M. Wand, "Continuation-Based Multiprocessing," *Higher-Order and Symbolic Computation,* vol. 12, no. 3, pp. 285-299, 1999.

[7] D. Ferguson and D. Deugo, "Call With Current Continuation Patterns," *8th Conference on Pattern Languages of Programs,* 2001. Retrieved January 19, 2006 from the World Wide Web: http://hillside.net/plop/plop2001/accepted_submissions/PLoP2001/dferguson0/PLoP2001_dferguson0_1.pdf.

[8] C.T. Haynes, "Logic Continuations," *J. Logic Programming,* vol. 4, no. 2, pp. 157-176, 1987.

[9] G.L. Steele, Jr., "RABBIT: A Compiler for SCHEME," AI Lab Technical Report AITR-474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[10] J.C. Reynolds, "The Discoveries of Continuations," *Lisp and Symbolic Computation,* vol. 6, no. 3-4, pp. 233-248, 1993.

[11] G.J. Sussman and G.L. Steele Jr., “Scheme: An Interpreter for Extended Lambda Calculus,” AI Lab Memo AIM-349, MIT AI Laboratory, Cambridge, Mass., Dec. 1975.

[12] W.D. Clinger, A.H. Hartheimer, and E.M Ost, “Implementation Strategies for First-Class Continuations,” *Higher-Order and Symbolic Computation*, vol. 12, no. 1, pp. 7-45, Apr. 1999.

[13] Microsoft Corporation, “Microsoft .NET Framework Developer Center: Getting Started,” *.NET Framework Developer Center: Getting Started*, 2006. Retrieved April 17, 2006 from the World Wide Web: http://msdn.microsoft.com/netframework/gettingstarted/default.aspx.

[14] Mono Project, “About Mono,” *About Mono – Mono*, 2005. Retrieved April 17, 2006 from the World Wide Web: http://www.mono-project.com/Mono:About.

[15] L.P. Deutsch and A.M.Schiffman, “Efficient Implementation of the Smalltalk-80 System,” *Conf. Record Eleventh Annual ACM Symp. Principles of Programming Languages*, pp. 297-302. Salt Lake City: ACM Press. 1984.

[16] D. McDermott, “An Efficient Environment Allocation Scheme in an Interpreter for a Lexically-Scoped Lisp,” *Conf. Record 1980 Lisp Conf.*, pp. 154-162. New York: ACM Press. 1980.

[17] D.H. Bartley and J.C. Jensen, “The Implementation of PC Scheme,” *Proc. 1986 ACM Conf. Lisp and Functional Programming*, pp. 86-93. New York: ACM Press. 1986.

[18] T. Sekiguchi, T. Sakamoto, and A. Yonezawa, “Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling,” *Lecture Notes in Computer Science*, vol. 2022, pp. 217-233, 2001.

[19] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen, “Continuations from Generalized Stack Inspection,” *Proc. Tenth ACM SIGPLAN Conf. Functional Programming*, pp. 216-227. New York: ACM Press. 2005.

[20] S. Kumar, C. Bruggeman, and R.K. Dybvig, “Threads Yield Continuations,” *Lisp and Symbolic Computation*, vol. 10, no. 3, pp. 223-236, 1998.

[21] G.J. Sussman, J. Holloway, G.L. Steele Jr., and A. Bell, “The Scheme-79 Chip,” AI Memo 559, MIT AI Laboratory, Cambridge, Mass., Jan. 1980.

[22] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Reading, Mass.: Addison-Wesley, 1983.

[23] R. Hieb, R.K. Dybvig, and C. Bruggeman, "Representing Control in the Present of First-Class Continuations," *SIGPLAN Notices,* vol. 25, no. 6, pp. 66-77, June 1990. Retrieved December 11, 2005 from the World Wide Web: http://www.cs.indiana.edu/~dyb/pubs/stack.pdf.

[24] J. Hugunin, "IronPython: A Fresh Look at Python on .Net," *Proc. PyCon DC 2004,* 2004. Retrieved February 9, 2006 from the World Wide Web: http://www.python.org/pycon/dc2004/papers/9/.

[25] A.W. Appel and D.B. MacQueen, "Standard ML of New Jersey," *Proc. Third International Symp. on Programmming Language Implementation and Logic Programming,* pp. 1-13. Passau: Springer-Verlag. 1991.

[26] M. Feeley, "A Better API for First-Class Continuations," *2nd Workshop on Scheme and Functional Programming,* 2001. Retrieved January 18, 2006 from the World Wide Web: http://kaolin.unice.fr/Scheme2001/feeley.pdf.

[27] R. Hieb, K. Dybvig, and C.W. Anderson, "Subcontinuations," *Lisp and Symbolic Computation,* vol. 7, no. 1, pp. 83-110, 1994.

[28] M. Gasbichler, E. Knauel, M. Sperber and R.A. Kelsey, "How to Add Threads to a Sequential Language Without Getting Tangled Up," *Scheme Workshop 2003,* 2003. Retrieved October 8, 2005 from the World Wide Web: http://www-pu.informatik.uni-tuebingen.de/users/sperber/papers/adding-threads.pdf

[29] C. Queinnec, "The influence of browsers on evaluators or, continuations to program web servers," *ACM SIGPLAN Notices,* vol. 35, no. 9, pp. 23-33, 2000.

[30] S. Ducasse, A. Lienhard and L. Renggli, "Seaside — a Multiple Control Flow Web Application Framework," *Proc. ESUG Research Track 2004,* pp. 231-257, submitted for publication. Retrieved January 18, 2006 from the World Wide Web: http://www.iam.unibe.ch/~scg/Archive/Papers/Duca04eSeaside.pdf

[31] D. Friedman, "An Introduction to Scheme," *Indiana University Scheme Educational-infrastructure Project,* 1994. Retrieved November 18, 2005 from the World Wide Web: http://www.cs.indiana.edu/eip/dfried.ps.

[32] D.E. Knuth, *Fundamental Algorithms,* Vol. 1, *The Art of Computer Programming.* Reading, Mass.: Addison-Wesley, 1997.

[33] C.D. Marlin, *Coroutines.* Berlin: Springer-Verlag, 1980.

[34] A.L. de Moura, N. Rodriguez, and R. Ierusalimschy, “Coroutines in Lua,” *J. Universal Computer Science,* vol. 10, no. 7, pp. 910-925, Jul. 2004. Retrieved October 10, 2005 from the World Wide Web: http://www.inf.puc-rio.br/~roberto/docs/corosblp.pdf.

[35] R.A. Kelsey and J.A. Rees, “A Tractable Scheme Implementation,” *Lisp and Symbolic Computation,* vol. 7, no. 4, pp. 315-335, 1994. Retrieved March 5, 2006 from the World Wide Web: http://citeseer.ist.psu.edu/cache/papers/cs/889/ftp:zSzzSzftp.cs.indiana.eduzSzindrazSzscheme-repositoryzSzinstructzSzwandzSzvlispzSzlasczSzscheme48.pdf/kelsey93tractable.pdf.

[36] R. Kelsey, “Pre-Scheme: A Scheme Dialect for Systems Programming,” unpublished manuscript. Retrieved March 5, 2006 from the World Wide Web: http://mumble.net/~kelsey/papers/prescheme.ps.gz.

[37] K.R. Anderson, T.J. Hickey, and P. Norvig, “SILK: A Playful Blend of Scheme and Java,” *Proc. Workshop on Scheme and Functional Programming 2000,* pp. 13-22. Houston: Rice University. 2000.

[38] P. Bothner, “Kawa – Compiling Dynamic Languages to the JVM,” *Proc. USENIX 1998 Technical Conf., FREENIX Track,* June 1998. Retrieved January 8, 2006 from the World Wide Web: http://sources.redhat.com/kawa/papers/Freenix98.ps.gz.

[39] M. Feeley, “A Portable Implementation of First-Class Continuations for Unrestricted Interoperability with C in a Multithreaded Scheme,” *Proc. Workshop on Scheme and Functional Programming 2000,* pp. 65-66. Houston: Rice University. 2000.

[40] S.G. Miller, “SISC: A Complete Scheme Interpreter in Java,” *SISC: Second Interpreter of Scheme Code,* Jan. 2002. Retrieved January 27, 2006 from the World Wide Web: http://citeseer.ist.psu.edu/miller02sisc.html.

[41] S. McConnell, *Code Complete.* Redmond, Wash.: Microsoft Press, 2004.

[42] M. Feely, “Gambit-C, a portable implementation of Scheme,” *Gambit,* 2006. Retrieved January 18, 2006 from the World Wide Web: http://www.iro.umontreal.ca/~gambit/doc/gambit-c_toc.html.

[43] R.P. Gabriel, *Performance and Evaluation of Lisp Systems.* Cambridge, Mass.: MIT Press, 1985.

[44] “IEEE Std 1178-1990 IEEE Standard for the Scheme Programming Language – Description,” *IEEE Standards Organization,* Mar. 2005. Retrieved January 26, 2006 from the World Wide Web: http://standards.ieee.org/reading/ieee/std_public/description/busarch/1178-1990_desc.html.

[45] C. Queinnec, “Designing Meroon V3,” *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP’93 Workshop,* C. Rathke, J. Kopp, H. Hohl, and H. Bretthauer, eds., 1993.

[46] D.P. Friedman, “Object-Oriented Style,” *International Lisp Conference,* 2003. Retrieved January 26, 2006 from the World Wide Web: http://www.cs.indiana.edu/hyplan/dfried/ooo.pdf.

[47] Y. Bres, B.P. Serpette, and M. Serrano, “Bigloo.NET: Compiling Scheme to .NET CLR,” *Journal of Object Technology,* vol. 3, no. 9, pp. 71-94, Oct. 2004. Retrieved January 27, 2006 from the World Wide Web: http://www.jot.fm/issues/issue_2004_10/article4.pdf.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley, 1995.

[49] J. McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1,” *Communications of the ACM,* vol. 3, no. 4, pp. 184-195, Apr. 1960. Retrieved January 30, 2006 from the World Wide Web: http://www-formal.stanford.edu/jmc/recursive.pdf.

[50] R.K. Dybvig, R. Hieb, and C. Bruggeman, “Syntactic Abstraction in Scheme,” *Lisp and Symbol Computation,* vol. 4, no. 5, pp. 83-110, Dec. 1993. Retrieved March 22, 2006 from the World Wide Web: http://www.cs.indiana.edu/~dyb/papers/syntactic.ps.gz.

[51] R.K. Dybvig, D.P. Friedman, and C.T. Haynes, “Expansion-Passing Style: A General Macro Mechanism,” *LISP and Symbolic Computation,* vol. 1, no. 1, pp. 53-75, Jun. 1988. Retrieved March 22, 2006 from the World Wide Web: http://www.brics.dk/~hosc/local/LaSC-1-1-pp53-75.pdf.

[52] E. Hilsdale and D.P. Friedman, “Writing Macros in Continuation-Passing Style,” *Proc. Workshop on Scheme and Functional Programming 2000,* pp. 53-60. Houston: Rice University. 2000. Retrieved March 22, 2006 from the World Wide Web: http://www.ccs.neu.edu/home/matthias/Scheme2000/hilsdale.ps.

[53] R.K. Dybvig and O. Waddell, “Portable syntax-case,” *Portable syntax-case,* 2005. Retrieved February 15, 2006 from the World Wide Web: http://www.cs.indiana.edu/chezscheme/syntax-case/.

[54] D.P. Friedman, M. Wand, and C.T. Haynes, *Essentials of Programming Languages.* Cambridge, Mass.: The MIT Press, 2001.

[55] R.K. Dybvig, “Three Implementation Models for Scheme,” PhD dissertation, Dept. of Computer Science, University of North Carolina, Chapel Hill, North Carolina, 1987. Retrieved March 16, 2006 from the World Wide Web: http://www.cs.indiana.edu/~dyb/papers/3imp.pdf.

[56] R.K. Dybvig, *The Scheme Programming Language.* Cambridge, Mass.: The MIT Press, 2003.

APPENDIX A

CONTINUATION IMPLEMENTATIONS

The appendix is split into four sections. The first two sections contain the source code for the five continuation implementations described in Chapter V. The first section contains the source code files common to all five strategies. The second section holds the files specific to each strategy. The third section contains the initialization files needed by the Scheme interpreter. The fourth section contains the collected benchmarks from Chapter III, as well as the top-level file used to orchestrate them.

A.1 Common Files

```
//
// Boolean.cs
//
namespace Scheme {
    public class Boolean {
        public static readonly Boolean TRUE =
            new Boolean();

        public static readonly Boolean FALSE =
            new Boolean();

        private Boolean() { }

        public static Boolean Create(string s) {
            return ( s.ToLower() == "#t" ) ? TRUE : FALSE;
        }

        public static Boolean Create(bool b) {
            return ( b ) ? TRUE : FALSE;
        }

        public static bool IsTrue(object o) {
            return ( o == FALSE ) ? false : true;
        }

        public override string ToString() {
            return ( this == TRUE ) ? "#t" : "#f";
        }
    }
}

//
// Character.cs
//
namespace Scheme {
    public class Character {
        private char _ch;

        public Character(char ch) { _ch = ch; }

        public static bool operator ==(Character x,
            Character y) {
            return (x == null || y == null) ?
                false : x._ch == y._ch;
        }
```

```
        public static bool operator !=(Character x,
            Character y) {

            return (x == null || y == null) ?

                true : x._ch != y._ch;;
        }

        public char NativeChar {
            get { return _ch; }
        }

        public override int GetHashCode() {
            return (int)_ch;
        }

        public override bool Equals(object o) {
            return (o is Character) ?
                this == (Character)o : (object)this == o;
        }

        public override string ToString() {
            switch (_ch) {
                case ' ':
                    return "#\\space";
                case '\n':
                    return "#\\newline";
                default:
                    return "#\\" + _ch;
            }
        }
    }
}

//
// Closure.cs
//
namespace Scheme {
    public class Closure : Procedure {
        private Pair _args;
        private Pair _exp;
        private Environment _env;
        private bool _variableArity;
        private long _argListLength;

        public Closure(Pair args, Pair exp,
            Environment env) {
            _variableArity = !List.IsProper(args);
            _args = (_variableArity) ?
                _MakeArgList(args) : args;
            _argListLength = List.Length(_args);
            _exp = exp;
            _env = env;
        }

        private static Pair _MakeArgList(Pair args) {
            return (args == Pair.EmptyList) ? args :
                new Pair(args.Head,
                    (args.Tail is Pair) ?
                        _MakeArgList((Pair)args.Tail) :
                        new Pair(args.Tail,
                            Pair.EmptyList));
        }

        public object Apply(Interpreter i, Pair args,
            Environment env) {
            Pair p = (_variableArity) ?
                _SeparateValues(args, 0) : args;
            Environment cenv = new Environment(_args, p,
                _env);

            return i.EvaluateSequence(_exp, cenv);
        }
```

```
        private Pair _SeparateValues(Pair args,
            int depth) {
            return (depth < _argListLength - 1) ?
                new Pair(args.Head,
                    _SeparateValues((Pair)args.Tail,
                        depth + 1)) :
                new Pair(args, Pair.EmptyList);
        }

        public override string ToString() {
            string s = "<#function>";
            s += " Args: " + _args;
            return s;
        }
    }
}

//
// Environment.cs
//
namespace Scheme {
    public class Environment {
        private System.Collections.Hashtable _symbolTable;
        private Environment _parentEnv;

        public Environment(Pair syms, Pair vals,
            Environment parentEnv) {
            _symbolTable =
                new System.Collections.Hashtable();
            _parentEnv = parentEnv;
            CollectValues(syms, vals);
        }

        public Environment(Pair syms, Pair vals) {
            _symbolTable =
                new System.Collections.Hashtable();
            _parentEnv = null;
            CollectValues(syms, vals);
        }

        public Environment(Environment parentEnv) {
            _symbolTable =
                new System.Collections.Hashtable();
            _parentEnv = parentEnv;
        }

        public Environment() {
            _symbolTable =
                new System.Collections.Hashtable();
            _parentEnv = null;
        }

        private void CollectValues(Pair syms, Pair vals) {
            if (syms != Pair.EmptyList) {
                _symbolTable[syms.Head] = vals.Head;
                CollectValues((Pair)syms.Tail,
                    (Pair)vals.Tail);
            }
        }

        public object Lookup(Symbol s) {
            if (_symbolTable.Contains(s)) {
                return _symbolTable[s];
            } else if (_parentEnv != null) {
                return _parentEnv.Lookup(s);
            } else {
                return null;
            }
        }

        public bool Assign(Symbol s, object v) {
            if (_symbolTable.Contains(s)) {
                _symbolTable[s] = v;
                return true;
```

```
            } else if (_parentEnv != null) {
                return _parentEnv.Assign(s, v);
            } else {
                return false;
            }
        }

        public void Define(Symbol s, object v) {
            if (Lookup(s) != null) {
                Assign(s, v);
            } else {
                _symbolTable[s] = v;
            }
        }
    }
}

//
// InterpreterShell.cs
//
namespace Scheme {
    using System.IO;

    public class InterpreterShell {
        private System.IO.TextWriter _writer;
        private Interpreter _i;

        public InterpreterShell(TextWriter writer,
            Interpreter i) {
            _writer = writer; _i = i;
        }

        public static void Main(string[] args) {
            InterpreterShell shell =
                new InterpreterShell(System.Console.Out,
                    new Interpreter(System.Console.In));

            string appDir =
                System.AppDomain.
                    CurrentDomain.BaseDirectory;

            shell._i.Read(Primitives.Reader);
            shell.Read(Path.Combine(appDir, "psyntax.pp"));
            shell.Read(Path.Combine(appDir, "init.scm"));

            if (args.Length == 2 && args[0] == "-f") {
                shell.Read(args[1]);
                shell.REPL();
            } else if (args.Length == 1) {
                shell.Read(args[0]);
            } else if (args.Length == 0) {
                shell.REPL();
            }
        }

        public void Read(string file) {
            try {
                System.IO.StreamReader reader =
                    new System.IO.StreamReader(file);
                _i.Read(reader);
            } catch (System.Exception e) {
                _writer.WriteLine(e.Message);
                _writer.WriteLine(e.StackTrace);
            }
        }

        public void REPL() {
            for (;;) {
                _writer.Write("> ");
                try {
                    object result = _i.Evaluate();
                    if (result != null) {
                        _writer.WriteLine(result);
                    }
```

```
                } catch (System.Exception e) {
                    _writer.WriteLine(e.Message);
                    _writer.WriteLine(e.StackTrace);
                }
            }
        }
    }
}

//
// List.cs
//
namespace Scheme {
    using System.Collections;

    public class List {
        public static object First(object o) {
            Pair p = o as Pair;
            return (p != null) ? p.Head : null;
        }

        public static object Second(object o) {
            Pair p = o as Pair;
            return (p != null) ? First(p.Tail) : null;
        }

        public static object Third(object o) {
            Pair p = o as Pair;
            return (p != null) ? Second(p.Tail) : null;
        }

        public static object Fourth(object o) {
            Pair p = o as Pair;
            return (p != null) ? Third(p.Tail) : null;
        }

        public static object Rest(object o) {
            Pair p = o as Pair;
            return (p != null) ? p.Tail : null;
        }

        public static Pair Last(object o) {
            Pair p = o as Pair;
            if (p != null) {
                return (p.Tail == Pair.EmptyList) ?
                    p : (p.Tail is Pair) ?
                        Last(p.Tail) : p;
            } else {
                throw new System.Exception(
                    "Error: attempt to perform list " +
                    "operation on non-list."
                );
            }
        }

        public static long Length(object o) {
            if (o == Pair.EmptyList) return 0L;
            Pair p = o as Pair;
            if (p != null) {
                return 1L + Length(p.Tail);
            } else {
                throw new System.Exception(
                    "Attempt to take length of non-list."
                );
            }
        }

        public static Pair Reverse(Pair p) {
            return Reverse(p, Pair.EmptyList);
        }

        private static Pair Reverse(Pair orig,
            Pair revList) {
            return (orig == Pair.EmptyList) ? revList :
```

```
                Reverse((Pair)List.Rest(orig),
                    new Pair(List.First(orig), revList));
        }

        public static Pair Flatten(Pair args) {
            return (args.Tail == Pair.EmptyList) ?
                (Pair)args.Head :
                new Pair(args.Head,
                    Flatten((Pair)args.Tail));
        }

        public static Pair Create(System.Array a) {
            return Create(a.GetEnumerator());
        }

        private static Pair Create(IEnumerator e) {
            return ( e.MoveNext() ) ?
                new Pair(e.Current, Create(e)) :
                Pair.EmptyList;
        }

        public static Pair Create(object e1) {
            return new Pair(e1, Pair.EmptyList);
        }

        public static Pair Create(object e1, object e2) {
            return new Pair(e1, new Pair(e2,
                Pair.EmptyList));
        }

        public static Pair Create(object e1, object e2,
            object e3) {
            return new Pair(e1,
                new Pair(e2, new Pair(e3,
                    Pair.EmptyList)));
        }

        public static Pair Create(object e1,
            object e2, object e3, object e4) {
            return new Pair(e1, new Pair(e2,
                new Pair(e3, new Pair(e4,
                    Pair.EmptyList))));
        }

        public static Vector ToVector(Pair p) {
            long l = Length(p);
            object[] vec = new object[l];
            Pair currentCons = p;

            for (int i = 0; i < l; ++i) {
                vec[i] = currentCons.Head;
                currentCons = (Pair)currentCons.Tail;
            }

            return new Vector(vec);
        }

        public static bool IsProper(Pair p) {
            Pair last = Last(p);
            return (last.Tail == Pair.EmptyList);
        }
    }
}

//
// Pair.cs
//
namespace Scheme {
    public class Pair {
        public static readonly Pair EmptyList;

        private object _head;
        private object _tail;
```

```
        static Pair() {
            EmptyList = new Pair(EmptyList, EmptyList);
        }

        public Pair(object head, object tail) {
            _head = head; _tail = tail;
        }

        public object Head {
            get { return _head; }
            set { _head = value; }
        }

        public object Tail {
            get { return _tail; }
            set { _tail = value; }
        }

        public override string ToString() {
            if (this == EmptyList) {
                return "()";
            } else {
                System.Text.StringBuilder buf =
                    new System.Text.StringBuilder();
                buf.Append('(');
                buf.Append(_head.ToString());
                object rest = _tail;
                while (rest is Pair && rest != EmptyList) {
                    buf.Append(" ");
                    buf.Append(((Pair)rest).
                        Head.ToString());
                    rest = ((Pair)rest).Tail;
                }
                if (rest != EmptyList) {
                    buf.Append(" . ");
                    buf.Append(rest.ToString());
                }
                buf.Append(')');

                return buf.ToString();
            }
        }
    }
}

//
// Parser.cs
//
namespace Scheme {
    using System.Text;

    public class Parser {
        private Scanner _s;

        public Parser(Scanner s) { _s = s; }

        public object Read() {
            return _Read(_s.NextToken());
        }

        private object _Read(Token t) {
            switch (t.Type) {
                case TokenType.Boolean:
                    return Boolean.Create(t.Text);
                case TokenType.Character:
                    return new Character(t.Text[0]);
                case TokenType.EOF:
                    return null;
                case TokenType.Integer:
                    try {
                        return long.Parse(t.Text);
                    } catch (System.FormatException) {
                        throw new System.Exception(
                            "Invalid integer format: " +
```

```
                    t.Text
                );
            }
        case TokenType.LeftParen:
            return ReadList();
        case TokenType.Quote:
            object[] quoteArr = new object[2];
            quoteArr[0] = SymbolCache.QUOTE;
            quoteArr[1] = Read();
            return List.Create(quoteArr);
        case TokenType.Quasiquote:
            object[] qquoteArr = new object[2];
            qquoteArr[0] = SymbolCache.QUASIQUOTE;
            qquoteArr[1] = Read();
            return List.Create(qquoteArr);
        case TokenType.Unquote:
            object[] unquoteArr = new object[2];
            unquoteArr[0] = SymbolCache.UNQUOTE;
            unquoteArr[1] = Read();
            return List.Create(unquoteArr);
        case TokenType.UnquoteSplicing:
            object[] unqspliceArr = new object[2];
            unqspliceArr[0] =
                SymbolCache.UNQUOTESPLICING;
            unqspliceArr[1] = Read();
            return List.Create(unqspliceArr);
        case TokenType.RightParen:
            throw new System.Exception(
                "Unmatched right parenthesis."
            );
        case TokenType.Dot:
            throw new System.Exception(
                "Unrecognized use of dot notation"
            );
        case TokenType.String:
            return t.Text;
        case TokenType.Symbol:
            return SymbolFactory.Create(t.Text);
        case TokenType.Vector:
            Token nt = _s.NextToken();
            if (nt.Type == TokenType.LeftParen) {
                return List.ToVector(
                    (Pair)ReadList());
            } else {
                throw new System.Exception(
                    "Invalid vector literal"
                );
            }
        default:
            throw new System.Exception(
                "Unrecognized token"
            );
    }
}

private object ReadList() {
    Token t = _s.NextToken();
    switch (t.Type) {
        case TokenType.RightParen:
            return Pair.EmptyList;
        case TokenType.Dot:
            object o = Read();
            Token nt = _s.NextToken();
            if (nt.Type != TokenType.RightParen) {
                throw new System.Exception(
                    "Missing close paren after " +
                    o
                );
            } else {
                return o;
            }
        default:
            return new Pair(_Read(t), ReadList());
    }
```

```
        }
    }
}

//
// Procedure.cs
//
namespace Scheme {
    public interface Procedure {
        object Apply(Interpreter i, Pair args,
            Environment env);
    }
}

//
// Scanner.cs
//
namespace Scheme {
    using System.IO;
    using System.Text;

    public class Scanner {
        private TextReader _reader;
        private int _ch;
        private bool _haveChar;

        public Scanner(TextReader reader) {
            _reader = reader;
            _haveChar = false;
        }

        private int PopChar() {
            _haveChar = false;
            return _ch;
        }

        private void PushChar(int ch) {
            _ch = ch; _haveChar = true;
        }

        private int GetChar() {
            return _haveChar ? PopChar() : _reader.Read();
        }

        private Token PoundLiteral() {
            int ch = GetChar();

            switch (ch) {
                case 't': case 'T':
                    return
                        new Token("#t", TokenType.Boolean);
                case 'f': case 'F':
                    return
                        new Token("#f", TokenType.Boolean);
                case '(':
                    PushChar(ch);
                    return
                        new Token("#", TokenType.Vector);
                case '\\':
                    Token t = NextToken();
                    if (t.Text.Length == 1) {
                        return new Token(t.Text,
                            TokenType.Character);
                    } else {
                        string s = t.Text.ToLower();

                        if (s == "space") {
                            return new Token(" ",
                                TokenType.Character);
                        } else if (s == "newline") {
                            return new Token("\n",
                                TokenType.Character);
                        } else {
                            throw new System.Exception(
```

```
                        "Unrecognized " +
                        "#\\... literal: " +
                        "#\\" + t.Text
                    );
                }
            }
        default:
            throw new System.Exception(
                "Badly formed #... literal: " +
                "#" + (char)ch
            );
    }
}

private Token StringLiteral() {
    StringBuilder strBuffer = new StringBuilder();
    int ch;

    while ((ch = _reader.Read()) != '"' &&
        ch != -1) {
        if ( ch != '\\') {
            strBuffer.Append((char)ch);
        }
    }

    return
        new Token(strBuffer.ToString(),
            TokenType.String);
}

public Token NextToken() {
    StringBuilder strBuffer = new StringBuilder();
    int ch = GetChar();

    while (char.IsWhiteSpace((char)ch)) {
        ch = _reader.Read();
    }

    switch (ch) {
        case -1:
            return new Token("", TokenType.EOF);
        case '(':
            return new Token("(",
                TokenType.LeftParen);
        case ')':
            return new Token(")",
                TokenType.RightParen);
        case '\'':
            return new Token("'", TokenType.Quote);
        case '`':
            return new Token("`",
                TokenType.Quasiquote);
        case ',':
            ch = _reader.Read();
            if (ch == '@') {
                return new Token(",@",
                    TokenType.UnquoteSplicing);
            } else {
                PushChar(ch);
                return new Token(",",
                    TokenType.Unquote);
            }
        case '"':
            return StringLiteral();
        case '#':
            return PoundLiteral();
        case ';':
            _reader.ReadLine();
            return NextToken();
        case '0': case '1': case '2': case '3':
        case '4': case '5': case '6': case '7':
        case '8': case '9':
            strBuffer.Append((char)ch);
```

```
                    while (char.IsDigit(
                        (char)(ch = _reader.Read()))) {
                        strBuffer.Append((char)ch);
                    }

                    if (char.IsWhiteSpace((char)ch) ||
                        ch == ')') {
                        PushChar(ch);
                        return
                            new Token(strBuffer.ToString(),
                                TokenType.Integer);
                    } else {
                        strBuffer.Append((char)ch);
                        throw new System.Exception(
                            "Unrecognized sequence: " +
                            strBuffer.ToString()
                        );
                    }
                default:
                    do {
                        strBuffer.Append((char)ch);
                        ch = _reader.Read();
                    } while (
                        !char.IsWhiteSpace((char)ch) &&
                            ch != -1 && ch != '(' &&
                            ch != ')' && ch != '\'' &&
                            ch != '"' && ch != ';'
                    );

                    PushChar(ch);

                    string s =
                        strBuffer.ToString().ToLower();

                    return
                        new Token(s, (s == ".") ?
                            TokenType.Dot :
                            TokenType.Symbol);
            }
        }
    }
}

//
// Symbol.cs
//
namespace Scheme {
    using System.Collections;

    public class Symbol {
        private readonly string _str;
        private Hashtable _propTable;
        private readonly bool _isGenSym;

        public Symbol(string str) {
            _str = str;
            _propTable = new Hashtable();
            _isGenSym = false;
        }

        public Symbol(string str, bool isGenSym) {
            _str = str;
            _propTable = new Hashtable();
            _isGenSym = isGenSym;
        }

        public bool IsGenSym() { return _isGenSym; }

        public override string ToString() { return _str; }

        public Hashtable Properties {
            get { return _propTable; }
        }
    }
```

```
public class SymbolFactory {
    private static Hashtable _symTable;
    private static int _symbolNumber;
    private static readonly string _genSymPrefix =
        "__%sym";

    static SymbolFactory() {
        _symTable = new Hashtable();
        _symbolNumber = 0;
    }

    public static Symbol Create(string str) {
        if (_symTable.Contains(str)) {
            return (Symbol)_symTable[str];
        } else {
            Symbol s = new Symbol(str);
            _symTable[str] = s;
            return s;
        }
    }

    public static Symbol Create() {
        string uniqueName = _GenerateUniqueName();
        Symbol s = new Symbol(uniqueName, true);
        _symTable[uniqueName] = s;
        return s;
    }

    private static string _GenerateUniqueName() {
        string newSymbol = _genSymPrefix +
            _symbolNumber++;
        while (_symTable.Contains(newSymbol)) {
            newSymbol = _genSymPrefix +
            _symbolNumber++;
        }

        return newSymbol;
    }
}

public class SymbolCache {
    public static readonly Symbol IF;
    public static readonly Symbol SET;
    public static readonly Symbol DEFINE;
    public static readonly Symbol LAMBDA;
    public static readonly Symbol QUOTE;
    public static readonly Symbol LETREC;
    public static readonly Symbol BEGIN;
    public static readonly Symbol AND;
    public static readonly Symbol OR;
    public static readonly Symbol EXPANDER;
    public static readonly Symbol QUASIQUOTE;
    public static readonly Symbol UNQUOTE;
    public static readonly Symbol UNQUOTESPLICING;
    public static readonly Symbol CALLWITHVALUES;
    public static readonly Symbol
        CALLWITHCURRENTCONTINUATION;
    public static readonly Symbol CALLCC;

    static SymbolCache() {
        IF = SymbolFactory.Create("if");
        SET = SymbolFactory.Create("set!");
        DEFINE = SymbolFactory.Create("define");
        LAMBDA = SymbolFactory.Create("lambda");
        QUOTE = SymbolFactory.Create("quote");
        LETREC = SymbolFactory.Create("letrec");
        BEGIN = SymbolFactory.Create("begin");
        AND = SymbolFactory.Create("and");
        OR = SymbolFactory.Create("or");
        EXPANDER = SymbolFactory.Create("_expander");
        QUASIQUOTE =
            SymbolFactory.Create("quasiquote");
        UNQUOTE = SymbolFactory.Create("unquote");
```

```
            UNQUOTESPLICING =
                SymbolFactory.Create("unquote-splicing");
            CALLWITHVALUES =
                SymbolFactory.Create("call-with-values");
            CALLWITHCURRENTCONTINUATION =
                SymbolFactory.Create(
                    "call-with-current-continuation"
                );
            CALLCC = SymbolFactory.Create("call/cc");
        }
    }
}

//
// Token.cs
//
namespace Scheme {
    public enum TokenType {
        Boolean,
        Character,
        Dot,
        EOF,
        Integer,
        LeftParen,
        Quasiquote,
        Quote,
        RightParen,
        String,
        Symbol,
        Unquote,
        UnquoteSplicing,
        Vector
    }

    public class Token {
        private readonly string _text;
        private readonly TokenType _type;

        public Token(string text, TokenType type) {
            _text = text; _type = type;
        }

        public TokenType Type {
            get { return _type; }
        }

        public string Text {
            get { return _text; }
        }
    }
}

//
// Vector.cs
//
namespace Scheme {
    public class Vector {
        private object[] _val;

        public Vector(object[] val) { _val = val; }

        public Vector(long n) { _val = new object[n]; }

        public int Length {
            get {
                return _val.Length;
            }
        }

        public object this[long index] {
            get { return _val[index]; }
            set { _val[index] = value; }
        }
```

```
        public override string ToString() {
            string s = "#" + _val.Length + "(";
            for (int i = 0; i < _val.Length; ++i) {
                if (i != 0) s += " ";
                s += _val[i];
            }
            s += ")";

            return s;
        }
    }
}
```

A.2 Implementation-Specific Files

A.2.1 Simple Exception Strategy

```
//
// Continuation.cs
//
namespace Scheme {
    public class ContinuationException : System.Exception {
        private object _val;

        public ContinuationException() { _val = null; }

        public object ReturnValue {
            get { return _val; }
            set { _val = value; }
        }
    }

    public class Continuation : Procedure {
        private ContinuationException _contEx;

        public Continuation(ContinuationException contEx) {
            _contEx = contEx;
        }

        public object Apply(Interpreter i, Pair args,
            Environment env) {
            _contEx.ReturnValue = List.First(args);
            throw _contEx;
        }
    }
}

//
// Interpreter.cs
//
namespace Scheme {
    using System.IO;

    public class Interpreter {
        private Environment _globalEnv;
        private MacroExpander _expander;

        public Interpreter(TextReader reader) {
            StandardLibrary sl =
                StandardLibrary.Instance();
            _globalEnv =
                new Environment(sl.Symbols, sl.Values);
            _expander = new MacroExpander(
                new Parser(new Scanner(reader)), this,
                    _globalEnv);
        }

        public object Evaluate() {
            object o = _expander.Read();
            return o == null ?
                null : Evaluate(o, _globalEnv);
```

```
}

public object Evaluate(object expr,
    Environment env) {
    if (!(expr is Pair)) {
        if (expr is Symbol) {
            object val = env.Lookup((Symbol)expr);
            if (val == null) {
                throw new System.Exception(
                    "No value associated " +
                    "with symbol " +
                    expr
                );
            }
            return val;
        } else {
            return expr;
        }
    } else {
        Pair p = (Pair)expr;
        object head = p.Head;
        object tail = p.Tail;

        if (head == SymbolCache.IF) {
            return Boolean.IsTrue(
                Evaluate(List.First(tail), env)) ?
                    Evaluate(List.Second(tail),
                        env) :
                    (List.Length(tail) == 3) ?
                        Evaluate(List.Third(tail),
                            env) :
                        null;
        } else if (head == SymbolCache.DEFINE) {
            env.Define((Symbol)List.First(tail),
                Evaluate(List.Second(tail), env));
            return null;
        } else if (head == SymbolCache.SET) {
            Symbol id = (Symbol)List.First(tail);
            if (!env.Assign(id,
                Evaluate(List.Second(tail),
                    env))) {
                throw new System.Exception(
                    "Cannot set undefined " +
                    "identifier " + id
                );
            }
            return null;
        } else if (head == SymbolCache.QUOTE) {
            return List.First(tail);
        } else if (head == SymbolCache.BEGIN) {
            return
                EvaluateSequence((Pair)tail, env);
        } else if (head == SymbolCache.LAMBDA) {
            return
                new Closure((Pair)List.First(tail),
                    (Pair)((Pair)tail).Tail, env);
        } else if (head == SymbolCache.LETREC) {
            Pair letrecTail = (Pair)tail;
            Pair nameValList =
                (Pair)letrecTail.Head;
            Pair letrecBody =
                (Pair)letrecTail.Tail;
            Environment letrecEnv =
                ExtendRecEnvironment(nameValList,
                    new Environment(env));
            return EvaluateSequence(letrecBody,
                letrecEnv);
        } else if (head == SymbolCache.AND ||
            head == SymbolCache.OR) {
            Procedure pred =
                (Procedure)Evaluate(head, env);
            return pred.Apply(this, (Pair)tail,
                env);
        } else {
```

```
                Procedure proc =
                    (Procedure)Evaluate(head, env);
                Pair args = EvalList((Pair)tail, env);
                object result =
                    proc.Apply(this, args, env);
                return result;
            }
        }
    }

    public object EvaluateSequence(Pair expr,
        Environment env) {
        if (expr == Pair.EmptyList) {
            return null;
        } else {
            object val = Evaluate(expr.Head, env);
            return (expr.Tail == Pair.EmptyList) ?
                val :
                EvaluateSequence((Pair)expr.Tail, env);
        }
    }

    public Pair EvalList(Pair p, Environment env) {
        return (p != Pair.EmptyList) ?
            new Pair(Evaluate(p.Head, env),
                EvalList((Pair)p.Tail, env)) : p;
    }

    public void Read(TextReader reader) {
        Read(reader, _globalEnv);
    }

    public void Read(TextReader reader,
        Environment env) {
        MacroExpander expander = new MacroExpander(
            new Parser(new Scanner(reader)),
                this, env);
        object o;
        while ((o = expander.Read()) != null) {
            try {
                Evaluate(o, env);
            } catch (System.Exception) { }
        }
    }

    private Environment ExtendRecEnvironment(
        Pair nameValList, Environment env) {
        if (List.Length(nameValList) > 0) {
            Pair head = (Pair)List.First(nameValList);
            Symbol s = (Symbol)List.First(head);
            object v =
                Evaluate(List.Second(head), env);
            env.Define(s, v);
            return ExtendRecEnvironment(
                (Pair)List.Rest(nameValList),
                env);
        } else {
            return env;
        }
    }
  }
}

//
// MacroExpander.cs
//
namespace Scheme {
    public class MacroExpander {
        private Parser _parser;
        private Interpreter _i;
        private Environment _env;

        public MacroExpander(Parser parser, Interpreter i,
            Environment env) {
```

```
            _parser = parser;
            _i = i;
            _env = env;
        }

        public object Read() {
            object expr = _parser.Read();

            object expander =
                _env.Lookup(SymbolCache.EXPANDER);
            Procedure p = expander as Procedure;

            return (expr != null && p != null) ?
                p.Apply(_i,
                    new Pair(expr, Pair.EmptyList), _env) :
                expr;
        }
    }
}

//
// StandardLibrary.cs
//
namespace Scheme {
    public class ValueList {
        private Pair _valList;

        public ValueList(Pair valList) {
            _valList = valList;
        }

        public Pair Values { get { return _valList; } }
    }

    public class StandardLibrary {
        private Pair _syms;
        private Pair _vals;
        private static StandardLibrary _stdlib = null;

        private static Symbol[] _names = {
            SymbolFactory.Create("cons"),
            SymbolFactory.Create("car"),
            SymbolFactory.Create("cdr"),
            SymbolFactory.Create("display"),
            SymbolFactory.Create("integer->char"),
            SymbolFactory.Create("+"),
            SymbolFactory.Create("-"),
            SymbolFactory.Create(">"),
            SymbolFactory.Create("<"),
            SymbolFactory.Create("zero?"),
            SymbolFactory.Create("load"),
            SymbolFactory.Create("vector"),
            SymbolFactory.Create("vector-ref"),
            SymbolFactory.Create("string-length"),
            SymbolFactory.Create("set-car!"),
            SymbolFactory.Create("set-cdr!"),
            SymbolFactory.Create("procedure?"),
            SymbolFactory.Create("symbol?"),
            SymbolFactory.Create("pair?"),
            SymbolFactory.Create("null?"),
            SymbolFactory.Create("eq?"),
            SymbolFactory.Create("eqv?"),
            SymbolFactory.Create("boolean?"),
            SymbolFactory.Create("vector?"),
            SymbolFactory.Create("void"),
            SymbolFactory.Create("putprop"),
            SymbolFactory.Create("getprop"),
            SymbolFactory.Create("remprop"),
            SymbolFactory.Create("values"),
            SymbolFactory.Create("call-with-values"),
            SymbolFactory.Create("="),
            SymbolFactory.Create("vector-length"),
            SymbolFactory.Create("append"),
            SymbolFactory.Create("string?"),
```

```
            SymbolFactory.Create("string=?"),
            SymbolFactory.Create("and"),
            SymbolFactory.Create("or"),
            SymbolFactory.Create("apply"),
            SymbolFactory.Create("string"),
            SymbolFactory.Create("list"),
            SymbolFactory.Create("number?"),
            SymbolFactory.Create("_native-eval"),
            SymbolFactory.Create("gensym"),
            SymbolFactory.Create("gensym?"),
            SymbolFactory.Create("list->vector"),
            SymbolFactory.Create("make-vector"),
            SymbolFactory.Create("vector-set!"),
            SymbolFactory.Create("_get-ticks"),
            SymbolFactory.Create("/"),
            SymbolFactory.Create(
                "call-with-current-continuation"
            )
        };

        private static Procedure[] _procs = {
            new Cons(), new Car(), new Cdr(),
            new Display(), new IntegerToChar(), new Add(),
            new Subtract(), new GreaterThan(),
            new LessThan(), new ZeroPred(), new Load(),
            new VectorCtor(), new VectorRef(),
            new StringLength(), new SetCar(), new SetCdr(),
            new ProcedurePred(), new SymbolPred(),
            new PairPred(), new NullPred(), new EqPred(),
            new EqvPred(), new BooleanPred(),
            new VectorPred(), new VoidProc(),
            new PutProp(), new GetProp(), new RemProp(),
            new ValuesProc(), new CallWithValues(),
            new NumericEqualsPred(), new VectorLength(),
            new AppendProc(), new StringPred(),
            new StringEqualsPred(), new ApplyProc(),
            new StringProc(), new ListProc(),
            new NumberPred(), new NativeEval(),
            new GenSymProc(), new GenSymPred(),
            new ListToVector(), new MakeVector(),
            new VectorSet(), new GetTicksProc(),
            new DivProc(), new CallCC()
        };

        private StandardLibrary() {
            _syms = List.Create(_names);
            _vals = List.Create(_procs);
        }

        public static StandardLibrary Instance() {
            if (_stdlib == null) {
                _stdlib = new StandardLibrary();
            }
            return _stdlib;
        }

        public Pair Symbols {
            get { return _syms; }
        }

        public Pair Values {
            get { return _vals; }
        }
    }

    public class PutProp : Procedure {
        public object Apply(Interpreter i, Pair args,
            Environment env) {
            Symbol s = (Symbol)List.First(args);
            Symbol prop = (Symbol)List.Second(args);

            s.Properties[prop] = List.Third(args);
            return null;
        }
```

```
}

public class AppendProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return _Append(Pair.EmptyList, args);
    }

    private object _Append(object ls, object args) {
        return (args == Pair.EmptyList) ?
            ls : _BuildList(ls, args);
    }

    private object _BuildList(object ls, object args) {
        return (ls == Pair.EmptyList) ?
            _Append(List.First(args),
                List.Rest(args)) :
            new Pair(List.First(ls),
                _BuildList(List.Rest(ls), args));
    }
}

public class VectorLength : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Vector vec = (Vector)List.First(args);
        return (long)vec.Length;
    }
}

public class NumberPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return List.First(args) is int;
    }
}

public class NumericEqualsPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        long len = List.Length(args);
        if (len < 2) {
            throw new System.Exception(
                "= expects at least 2 arguments"
            );
        } else {
            long currentInt = (long)args.Head;
            Pair currentPair = (Pair)args.Tail;
            while (currentPair != Pair.EmptyList) {
                long nextInt =
                    (long)List.First(currentPair);
                if (currentInt != nextInt) {
                    return Boolean.Create(false);
                }
                currentPair = (Pair)currentPair.Tail;
            }

            return Boolean.Create(true);
        }
    }
}

public class GetProp : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Symbol s = (Symbol)List.First(args);
        Symbol prop = (Symbol)List.Second(args);

        return (s.Properties.Contains(prop)) ?
            s.Properties[prop] : Boolean.Create(false);
    }
}

public class RemProp : Procedure {
```

```
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Symbol s = (Symbol)List.First(args);
        s.Properties.Remove(List.Second(args));
        return null;
    }
}

public class ValuesProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        long len = List.Length(args);
        if (len > 1) {
            return new ValueList(args);
        } else if (len == 1) {
            return List.First(args);
        } else {
            return null;
        }
    }
}

public class CallWithValues : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object producer = List.First(args);
        object consumer = List.Second(args);

        object producerResult =
            ((Procedure)producer).Apply(i, null, env);
        Pair consumerArgs;
        if (producerResult is ValueList) {
            ValueList vl = (ValueList)producerResult;
            consumerArgs = vl.Values;
        } else if (producerResult != null) {
            consumerArgs =
                new Pair(producerResult,
                    Pair.EmptyList);
        } else {
            consumerArgs = Pair.EmptyList;
        }

        return ((Procedure)consumer).Apply(i,
            consumerArgs, env);
    }
}

public class VoidProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return null;
    }
}

public class EqPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) ==
            List.Second(args));
    }
}

public class StringPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) is
            string);
    }
}

public class StringEqualsPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        string s1 = (string)List.First(args);
```

```
        string s2 = (string)List.Second(args);
        return Boolean.Create(s1 == s2);
    }
}

public class VectorPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) is
            Vector);
    }
}

public class BooleanPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object o = List.First(args);
        return Boolean.Create(o == Boolean.TRUE ||
            o == Boolean.FALSE);
    }
}

public class EqvPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object o1 = List.First(args);
        object o2 = List.Second(args);
        return Boolean.Create(
            (o1 is long && o2 is long) ?
                (long)o1 == (long)o2 :
                (o1 is Character && o2 is Character) ?
                    (Character)o1 == (Character)o2 :
                    o1 == o2
        );
    }
}

public class NullPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) ==
            Pair.EmptyList);
    }
}

public class SymbolPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) is
            Symbol);
    }
}

public class PairPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object o = List.First(args);
        return Boolean.Create(o != Pair.EmptyList &&
            o is Pair);
    }
}

public class ProcedurePred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return Boolean.Create(List.First(args) is
            Procedure);
    }
}

public class SetCdr : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Pair p = List.First(args) as Pair;
```

```
        if (p == null) {
            throw new System.Exception(
                "Error: attempt to set cdr of " +
                "non-pair argument"
            );
        }

        p.Tail = List.Second(args);

        return null;
    }
}

public class SetCar : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Pair p = List.First(args) as Pair;

        if (p == null) {
            throw new System.Exception(
                "Error: attempt to set car of " +
                "non-pair argument"
            );
        }

        p.Head = List.Second(args);

        return null;
    }
}

public class StringLength : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        string s = List.First(args) as string;
        if (s == null) {
            throw new System.Exception(
                "Error: cannot apply string-length " +
                "to non-string value"
            );
        } else {
            return s.Length;
        }
    }
}

public class VectorCtor : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return List.ToVector(args);
    }
}

public class VectorRef : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object vec = List.First(args);
        long index = (long)List.Second(args);

        Vector v = vec as Vector;
        if (v == null) {
            throw new System.Exception(
                "Attempt to index non-vector value: " +
                vec
            );
        } else if (index < v.Length) {
            return v[index];
        } else {
            throw new System.Exception(
                "Index " + index +
                " out of range [0, " +
                (v.Length - 1) + "] in vector " + v
            );
```

```
        }
    }
}

public class Load : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        System.IO.StreamReader reader =
            new System.IO.StreamReader(
                List.First(args).ToString());
        i.Read(reader, env);
        return null;
    }
}

public class ZeroPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        int n = (int)List.First(args);
        return Boolean.Create(n == 0);
    }
}

public class Cons : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return new Pair(List.First(args),
            List.Second(args));
    }
}

public class Car : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object o = List.First(args);
        if (o == Pair.EmptyList) {
            throw new System.Exception(
                "Cannot take car of empty list"
            );
        } else {
            Pair p = o as Pair;
            if (p == null) {
                throw new System.Exception(
                    "Cannot take car of non-pair"
                );
            } else {
                return p.Head;
            }
        }
    }
}

public class Cdr : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object o = List.First(args);
        if (o == Pair.EmptyList) {
            throw new System.Exception(
               "Cannot take cdr of empty list"
            );
        } else {
            Pair p = o as Pair;
            if (p == null) {
                throw new System.Exception(
                    "Cannot take cdr of non-pair"
                );
            } else {
                return p.Tail;
            }
        }
    }
}

public class Display : Procedure {
```

```
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        System.Console.Write(List.First(args));
        return null;
    }
}

public class IntegerToChar : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return (char)((int)List.First(args));
    }
}

public class Add : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object n1 = List.First(args);
        object n2 = List.Second(args);
        if (n1 is int && n2 is int) {
            int i1 = (int)n1;
            int i2 = (int)n2;
            return i1 + i2;
        } else if (n1 is long && n2 is long) {
            long l1 = (long)n1;
            long l2 = (long)n2;
            return l1 + l2;
        }

        throw new System.Exception(
            "Unrecognized addition"
        );
    }
}

public class Subtract : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object n1 = List.First(args);
        object n2 = List.Second(args);
        if (n1 is int && n2 is int) {
            int i1 = (int)n1;
            int i2 = (int)n2;
            return i1 - i2;
        } else if (n1 is long && n2 is long) {
            long l1 = (long)n1;
            long l2 = (long)n2;
            return l1 - l2;
        }

        throw new System.Exception(
            "Unrecognized subtraction"
        );
    }
}

public class GreaterThan : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object n1 = List.First(args);
        object n2 = List.Second(args);
        if (n1 is long && n2 is long) {
            long i1 = (long)n1;
            long i2 = (long)n2;
            return Boolean.Create(i1>i2);
        }

        throw new System.Exception("Unrecognized >");
    }
}

public class LessThan : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
```

```
        object n1 = List.First(args);
        object n2 = List.Second(args);
        if (n1 is long && n2 is long) {
            long i1 = (long)n1;
            long i2 = (long)n2;
            return Boolean.Create(i1<i2);
        }

        throw new System.Exception("Unrecognized <");
    }
}

public class ApplyProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Procedure p = (Procedure)List.First(args);
        return p.Apply(i,
            List.Flatten((Pair)List.Rest(args)), env);
    }
}

public class StringProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        System.Text.StringBuilder strBuffer =
            new System.Text.StringBuilder();

        Pair currentPair = args;

        while (currentPair != Pair.EmptyList) {
            strBuffer.Append(
                ((Character)List.First(currentPair)).
                    NativeChar
            );
            currentPair = (Pair)currentPair.Tail;
        }

        return strBuffer.ToString();
    }
}

public class ListProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return args;
    }
}

public class NativeEval : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return i.Evaluate(List.First(args), env);
    }
}

public class GenSymProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return SymbolFactory.Create();
    }
}

public class GenSymPred : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Symbol s = (Symbol)List.First(args);
        return Boolean.Create(s.IsGenSym());
    }
}

public class ListToVector : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return List.ToVector((Pair)List.First(args));
```

```
    }
}

public class MakeVector : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        long n = (long)List.First(args);
        Vector v = new Vector(n);

        if (List.Length(args) == 2) {
            object o = List.Second(args);
            for (long index = 0; index < n; ++index) {
                v[index] = o;
            }
        }

        return v;
    }
}

public class VectorSet : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        Vector v = (Vector)List.First(args);
        long n = (long)List.Second(args);
        object val = List.Third(args);

        v[n] = val;

        return null;
    }
}

public class GetTicksProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        return System.DateTime.UtcNow.Ticks;
    }
}

public class DivProc : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        object n1 = List.First(args);
        object n2 = List.Second(args);
        if (n1 is long && n2 is long) {
            long i1 = (long)n1;
            long i2 = (long)n2;
            return i1 / i2;
        }

        throw new System.Exception(
            "Unrecognized division"
        );
    }
}

public class CallCC : Procedure {
    public object Apply(Interpreter i, Pair args,
        Environment env) {
        ContinuationException contEx =
            new ContinuationException();
        Continuation k = new Continuation(contEx);
        Procedure p = (Procedure)List.First(args);
        try {
            return p.Apply(i,
                new Pair(k, Pair.EmptyList), env);
        } catch (ContinuationException kEx) {
            if (kEx == contEx) {
                return kEx.ReturnValue;
            } else {
                throw kEx;
            }
        }
```

```
        }
    }
}
```

A.2.2 Heap Strategy

```
//
// Closure.cs
// The implementation of Closure follows the common file
// with the exception of its Apply method.
//
namespace Scheme {
    public class Closure : Procedure {
        //…
        public void Apply(Interpreter i, Pair args) {
            Pair p = (_variableArity) ?
                _SeparateValues(args, 0) : args;
            Environment cenv =
                new Environment(_args, p, _env);
            i.Context.Env = cenv;
            i.Context.Expr = _exp;
        }
        //…
    }
}

//
// Compiler.cs
//
namespace Scheme {
    public class Compiler {
        private MacroExpander _macroExp;

        public Compiler(MacroExpander macroExp) {
            _macroExp = macroExp;
        }

        public Compiler() { _macroExp = null; }

        public object Compile() {
            object o = _macroExp.Expand();
            return (o == null) ?
                o : Compile(o, InstructionCache.End);
        }

        public object Compile(object expr) {
            return Compile(expr, InstructionCache.End);
        }

        public object Compile(object expr,
            object nextExpr) {
            if (expr is Pair) {
                Pair p = (Pair)expr;
                object head = p.Head;
                object tail = p.Tail;

                if (head == SymbolCache.IF) {
                    object alternative =
                        (List.Length(tail) == 3) ?
                            Compile(List.Third(tail),
                                nextExpr) :
                            List.Create(
                                Instruction.Constant,
                                null, nextExpr);
                    return Compile(List.First(tail),
                        List.Create(
                            Instruction.Conditional,
                            Compile(List.Second(tail),
                                nextExpr),
                            alternative));
                } else if (head == SymbolCache.LAMBDA) {
```

```
            object bodyExpr = List.Rest(tail);
            if (bodyExpr == Pair.EmptyList) {
                return nextExpr;
            } else {
                object body =
                    CompileSequence((Pair)bodyExpr,
                        InstructionCache.End);
                return List.Create(
                    Instruction.Closure,
                    List.First(tail), body,
                    nextExpr);
            }
        } else if (head == SymbolCache.SET) {
            return Compile(List.Second(tail),
                List.Create(Instruction.Assign,
                    List.First(tail), nextExpr));
        } else if (head == SymbolCache.DEFINE) {
            return Compile(List.Second(tail),
                List.Create(Instruction.Define,
                    List.First(tail), nextExpr));
        } else if (head == SymbolCache.QUOTE) {
            return List.Create(Instruction.Quote,
                List.First(tail), nextExpr);
        } else if (head == SymbolCache.BEGIN) {
            return (tail == Pair.EmptyList) ?
                nextExpr :
                CompileSequence((Pair)tail,
                    nextExpr);
        } else if (head == SymbolCache.LETREC) {
            object letrecExp =
                List.Create(Instruction.Closure,
                    Pair.EmptyList,
                    CompileLetrec(tail),
                    InstructionCache.Apply);
            return (IsTail(nextExpr)) ? letrecExp :
                List.Create(Instruction.Context,
                    letrecExp, nextExpr);
        } else if (head ==
            SymbolCache.CALLWITHVALUES) {
            object producerExpr =
                Compile(List.First(tail),
                InstructionCache.Apply);
            object consumerExpr =
                List.Create(
                    Instruction.ArgMultiple,
                    Compile(List.Second(tail),
                        InstructionCache.Apply));
            object callSeq =
                List.Create(Instruction.Context,
                    producerExpr, consumerExpr);
            return (IsTail(nextExpr)) ? callSeq :
                List.Create(Instruction.Context,
                    callSeq, nextExpr);
        } else {
            object applicationExpr =
                CompileApplication(head, tail);
            return (IsTail(nextExpr)) ?
                applicationExpr :
                List.Create(Instruction.Context,
                    applicationExpr, nextExpr);
        }
    } else {
        return (expr is Symbol) ?
            List.Create(Instruction.Variable,
                expr, nextExpr) :
            List.Create(Instruction.Constant,
                expr, nextExpr);
    }
}

private bool IsTail(object nextExpr) {
    return (List.First(nextExpr) ==
        Instruction.End);
}
```

```
        private object CompileLetrec(object expr) {
            return CompileDefinitions(List.First(expr),
                List.Rest(expr));
        }

        private object CompileDefinitions(object defExprs,
            object exprSeq) {
            if (defExprs == Pair.EmptyList) {
                return CompileSequence((Pair)exprSeq,
                    new Pair(Instruction.End,
                        Pair.EmptyList));
            } else {
                object def = List.First(defExprs);
                return Compile(List.Second(def),
                    List.Create(Instruction.Define,
                        List.First(def),
                        CompileDefinitions(
                            List.Rest(defExprs),
                            exprSeq)));
            }
        }

        private object CompileSequence(Pair exprList,
            object nextExpr) {
            return (exprList == Pair.EmptyList) ?
                nextExpr :
                Compile(List.First(exprList),
                    CompileSequence(
                        (Pair)List.Rest(exprList),
                        nextExpr));
        }

        private object CompileApplication(object rator,
            object rands) {
            return CompileArguments(rands,
                Compile(rator, new Pair(Instruction.Apply,
                    Pair.EmptyList)));
        }

        private object CompileArguments(object rands,
            object nextExpr) {
            return (rands == Pair.EmptyList) ? nextExpr :
                Compile(List.First(rands),
                    List.Create(Instruction.Argument,
                        CompileArguments(List.Rest(rands),
                            nextExpr)));
        }
    }
}

//
// Continuation.cs
//
namespace Scheme {
    public class Continuation : Procedure {
        private Frame _context;

        public Continuation(Frame context) {
            _context = context;
        }

        public void Apply(Interpreter i, Pair args) {
            i.Context = new Frame(List.First(args),
                InstructionCache.End, i.Context);
            i.Context.Context = _context;
        }
    }
}

//
// Frame.cs
//
namespace Scheme {
```

```
public class Frame {
    public object Expr;
    public Environment Env;
    public object Value;
    public Pair ValueRib;
    public Frame Context;

    public Frame() {
        Expr = InstructionCache.End;
        Env = null;
        Value = null;
        ValueRib = null;
        Context = null;
    }

    public Frame(object expr, Frame context) {
        Expr = expr;
        Env = context.Env;
        Value = context.Value;
        ValueRib = context.ValueRib;
        Context = context.Context;
    }

    public Frame(object val, object expr,
        Frame context) {
        Expr = expr;
        Env = context.Env;
        Value = val;
        ValueRib = context.ValueRib;
        Context = context.Context;
    }

    public Frame(Frame context) {
        Expr = InstructionCache.End;
        Env = context.Env;
        Value = null;
        ValueRib = null;
        Context = context;
    }

    public Frame(Environment env) {
        Expr = InstructionCache.End;
        Env = env;
        Value = null;
        ValueRib = null;
        Context = null;
    }

    public Frame Clone() {
        Frame f = new Frame();

        f.Expr = Expr;
        f.Env = Env;
        f.Value = Value;
        f.ValueRib = ValueRib;
        f.Context = Context;

        return f;
    }

    public override string ToString() {
        string s = "Frame\n" +
            "Expr: " + ((Expr == null) ?
                "[null]" : Expr) +
            "\n" + "Env: " +
            ((Env == null) ?
                "[null]" : Env.ToString()) + "\n" +
            "Value: " + ((Value == null) ?
                "[null]" : Value) +
            "\n" + "ValueRib: " +
            ((ValueRib == null) ?
                "[null]" : ValueRib.ToString()) +
            "\n" + "Context: " +
            ((Context == null) ?
```

```
                    "[null]" : Context.ToString()) + "\n";
            return s;
        }
    }
}

//
// Instruction.cs
//
namespace Scheme {
    public class Instruction {
        public static readonly Instruction Apply;
        public static readonly Instruction ArgMultiple;
        public static readonly Instruction Argument;
        public static readonly Instruction Assign;
        public static readonly Instruction Closure;
        public static readonly Instruction Constant;
        public static readonly Instruction Context;
        public static readonly Instruction Define;
        public static readonly Instruction Quote;
        public static readonly Instruction Variable;
        public static readonly Instruction Conditional;
        public static readonly Instruction End;

        static Instruction() {
            Apply = new Instruction();
            ArgMultiple = new Instruction();
            Argument = new Instruction();
            Assign = new Instruction();
            Closure = new Instruction();
            Constant = new Instruction();
            Context = new Instruction();
            Define = new Instruction();
            Quote = new Instruction();
            Variable = new Instruction();
            Conditional = new Instruction();
            End = new Instruction();
        }

        public override string ToString() {
            return (this == Apply ) ? "Apply" :
                (this == ArgMultiple) ? "ArgMultiple" :
                (this == Argument) ? "Argument" :
                (this == Assign) ? "Assign" :
                (this == Closure) ? "Closure" :
                (this == Constant) ? "Constant" :
                (this == Context) ? "Context" :
                (this == Define) ? "Define" :
                (this == Quote) ? "Quote" :
                (this == Variable) ? "Variable" :
                (this == Conditional) ? "Conditional" :
                "End";
        }
    }

    public class InstructionCache {
        public static Pair Apply =
            new Pair(Instruction.Apply, Pair.EmptyList);
        public static Pair End =
            new Pair(Instruction.End, Pair.EmptyList);
    }
}

//
// Interpreter.cs
//
namespace Scheme {
    using System.IO;

    public class Interpreter {
        private Environment _globalEnv;
        private Compiler _compiler;
        private Frame _context;
```

```
public Interpreter(TextReader reader) {
    StandardLibrary sl =
        StandardLibrary.Instance();
    _globalEnv =
        new Environment(sl.Symbols, sl.Values);
    _compiler =
        new Compiler(new MacroExpander(new Parser(
            new Scanner(reader)), this));
    _context = new Frame();
    _context.Env = _globalEnv;
}

public Frame Context {
    get { return _context; }
    set { _context = value; }
}

public object Evaluate() {
    object o = _compiler.Compile();

    if (o == null) return o;

    _context.Expr = o;

    return Execute();
}

public object Execute() {
    for (;;) {
        object instr = List.First(_context.Expr);

        if (instr == Instruction.Constant) {
            _context = new Frame(
                List.Second(_context.Expr),
                List.Third(_context.Expr),
                _context);
        } else if (instr == Instruction.Variable) {
            object sym =
                List.Second(_context.Expr);
            object val =
                _context.Env.Lookup((Symbol)sym);
            if (val == null) {
                _context = new Frame(_globalEnv);
                throw new System.Exception(
                    "No value associated " +
                    "with symbol " + sym
                );
            } else {
                _context = new Frame(val,
                    List.Third(_context.Expr),
                    _context);
            }
        } else if (instr ==
            Instruction.Conditional) {
            _context = new Frame(
                (Boolean.IsTrue(_context.Value)) ?
                    List.Second(_context.Expr) :
                    List.Third(_context.Expr),
                    _context);
        } else if (instr == Instruction.Closure) {
            _context = new Frame(new Closure(
                (Pair)List.Second(_context.Expr),
                (Pair)List.Third(_context.Expr),
                _context.Env),
                List.Fourth(_context.Expr),
                _context);
        } else if (instr == Instruction.Quote) {
            _context = new Frame(
                List.Second(_context.Expr),
                List.Third(_context.Expr),
                _context);
        } else if (instr == Instruction.Assign) {
            Symbol id =
                (Symbol)List.Second(_context.Expr);
```

```
            if (_context.Env.Assign(id,
                _context.Value)) {
                _context = new Frame(null,
                    List.Third(_context.Expr),
                    _context);
            } else {
                _context = new Frame(_globalEnv);
                throw new System.Exception(
                    "Cannot set undefined " +
                    "identifier " + id
                );
            }
        } else if (instr == Instruction.Define) {
            Symbol id =
                (Symbol)List.Second(_context.Expr);
            _context.Env.Define(id,
                _context.Value);
            _context.Value = null;
            _context.Expr =
                List.Third(_context.Expr);
        } else if (instr == Instruction.Context) {
            object fn = List.Second(_context.Expr);
            _context = new Frame(
                List.Third(_context.Expr),
                _context);
            _context = new Frame(_context);
            _context.Expr = fn;
        } else if (instr == Instruction.Argument) {
            if (_context.ValueRib == null) {
                _context.ValueRib =
                    new Pair(_context.Value,
                        Pair.EmptyList);
            } else {
                List.Last(_context.ValueRib).Tail =
                    new Pair(_context.Value,
                        Pair.EmptyList);
            }
            _context = new Frame(
                List.Second(_context.Expr),
                _context);
        } else if (instr == Instruction.Apply) {
            Procedure proc =
                (Procedure)_context.Value;
            Pair args = _context.ValueRib;
            _context.ValueRib = null;
            proc.Apply(this, args);
        } else if (instr ==
            Instruction.ArgMultiple) {
            _context.ValueRib =
                (_context.Value is ValueList) ?
                    ((ValueList)_context.Value).
                        Values :
                    (_context.Value != null) ?
                        List.Create(
                            _context.Value) :
                        Pair.EmptyList;
            _context.Expr =
                List.Second(_context.Expr);
        } else if (instr == Instruction.End) {
            if (_context.Context != null) {
                _context.Context.Value =
                    _context.Value;
                _context = _context.Context;
            } else {
                break;
            }
        } else {
            System.Console.WriteLine(
                "Unexpected value: " +
                ((instr == null) ?
                    "[null]" : instr));
            break;
        }
    }
```

```
            return _context.Value;
        }

        public void Read(TextReader reader) {
            Frame context = _context;
            _context = new Frame(context.Env);
            Compiler compiler =
                new Compiler(new MacroExpander(
                new Parser(new Scanner(reader)), this));
            object o;
            while ((o = compiler.Compile()) != null) {
                _context.Expr = o;
                try {
                    Execute();
                } catch (System.Exception e) {
                    System.Console.WriteLine(
                        e.Message + "\n" +
                        e.StackTrace);
                }
            }
            _context = context;
        }
    }
}

//
// MacroExpander.cs
//
namespace Scheme {
    public class MacroExpander {
        private Parser _parser;
        private Interpreter _i;

        public MacroExpander(Parser parser,
            Interpreter i) {
            _parser = parser;
            _i = i;
        }

        public object Expand() {
            object expr = _parser.Read();

            object expander =
                _i.Context.Env.
                    Lookup(SymbolCache.EXPANDER);
            Procedure p = expander as Procedure;

            if (expr != null && p != null) {
                _i.Context = new Frame(_i.Context);
                _i.Context.Expr = InstructionCache.Apply;
                _i.Context.ValueRib =
                    new Pair(expr, Pair.EmptyList);
                _i.Context.Value = expander;
                return _i.Execute();
            } else {
                return expr;
            }
        }
    }
}

//
// Procedure.cs
//
namespace Scheme {
    public interface Procedure {
        void Apply(Interpreter i, Pair args);
    }
}

//
// StandardLibrary.cs
//
```

```
// The standard library implementation for the heap
// strategy is similar to the simple exception strategy
// with the following exceptions.
//
// All standard library classes implement the new
// procedure interface shown in Procedure.cs.
//
// *Apply* uses the following protocol to pass its result
// to its caller instead of returning a value directly.
// *i* is the instance of *Interpreter* passed to *Apply*.
//
// i.Context.Value = <return_value>;
// i.Context.Expr = InstructionCache.End;
//
// New classes or classes whose implementation has changed
// substantially are shown below.
//
namespace Scheme {
    public class ValuesProc : Procedure {
        public void Apply(Interpreter i, Pair args) {
            long len = List.Length(args);
            i.Context.Value = (len > 1) ?
                new ValueList(args) :
                (len == 1) ? List.First(args) : null;
            i.Context.Expr = InstructionCache.End;
        }
    }

    public class ApplyProc : Procedure {
        public void Apply(Interpreter i, Pair args) {
            i.Context.Expr =
                new Pair(Instruction.Apply,
                    Pair.EmptyList);
            i.Context.Value = List.First(args);
            i.Context.ValueRib =
                List.Flatten((Pair)List.Rest(args));
        }
    }

    public class CallCC : Procedure {
        public void Apply(Interpreter i, Pair args) {
            i.Context.ValueRib =
                new Pair(new Continuation(
                    i.Context.Context), Pair.EmptyList);
            i.Context.Expr = InstructionCache.Apply;
            i.Context.Value = List.First(args);
        }
    }

    // CompileProc is bound to the name "_compile"
    // in the global env
    public class CompileProc : Procedure {
        public void Apply(Interpreter i, Pair args) {
            object expr = List.First(args);
            Compiler c = new Compiler();
            i.Context.Value = c.Compile(expr);
            i.Context.Expr = InstructionCache.End;
        }
    }
}
```

A.2.3 Frame Recycling Strategy

```
//
// Continuation.cs
//
namespace Scheme {
    public class Continuation : Procedure {
        private Frame _context;

        public Continuation(Frame context) {
```

```
            _context = context;
            _context.Mark();
        }

        public void Apply(Interpreter i, Pair args) {
            i.Context = new Frame(List.First(args),
                InstructionCache.End, i.Context);
            i.Context.Context = _context;
        }
    }
}

//
// Frame.cs
//
namespace Scheme {
    public class Frame {
        public object Expr;
        public Environment Env;
        public object Value;
        public Pair ValueRib;
        public Frame Context;
        private bool _captured;

        public Frame() {
            Expr = InstructionCache.End;
            Env = null;
            Value = null;
            ValueRib = null;
            Context = null;
            _captured = false;
        }

        public Frame(object expr, Frame context) {
            Expr = expr;
            Env = context.Env;
            Value = context.Value;
            ValueRib = context.ValueRib;
            Context = context.Context;
            _captured = false;
        }

        public Frame(object val, object expr,
            Frame context) {
            Expr = expr;
            Env = context.Env;
            Value = val;
            ValueRib = context.ValueRib;
            Context = context.Context;
            _captured = false;
        }

        public Frame(Frame context) {
            Expr = InstructionCache.End;
            Env = context.Env;
            Value = null;
            ValueRib = null;
            Context = context;
            _captured = false;
        }

        public Frame(Environment env) {
            Expr = InstructionCache.End;
            Env = env;
            Value = null;
            ValueRib = null;
            Context = null;
            _captured = false;
        }

        public Frame Clone() {
            Frame f = new Frame();

            f.Expr = Expr;
```

```
        f.Env = Env;
        f.Value = Value;
        f.ValueRib = ValueRib;
        f.Context = Context;
        f._captured = _captured;

        return f;
    }

    public bool Captured { get { return _captured; } }

    public void Mark() {
        _captured = true;
        if (Context != null) {
            Context.Mark();
        }
    }
}

public class FrameFactory {
    private static System.Collections.Queue _freeList;

    static FrameFactory() {
        _freeList = new System.Collections.Queue();
    }

    public static Frame Create(object val,
        object expr, Frame context) {
        Frame f;
        if (_freeList.Count > 0) {
            f = (Frame)_freeList.Dequeue();
            f.Expr = expr;
            f.Value = val;
            f.ValueRib = context.ValueRib;
            f.Env = context.Env;
            f.Context = context.Context;
        } else {
            f = new Frame(val, expr, context);
        }

        if (!context.Captured) {
            _freeList.Enqueue(context);
        }

        return f;
    }

    public static Frame Create(object expr,
        Frame context) {
        Frame f;
        if (_freeList.Count > 0) {
            f = (Frame)_freeList.Dequeue();
            f.Expr = expr;
            f.Value = context.Value;
            f.ValueRib = context.ValueRib;
            f.Env = context.Env;
            f.Context = context.Context;
        } else {
            f = new Frame(expr, context);
        }

        if (!context.Captured) {
            _freeList.Enqueue(context);
        }

        return f;
    }

    public static void Release(Frame f) {
        if (!f.Captured) {
            _freeList.Enqueue(f);
        }
    }
}
```

```
}

//
// Interpreter.cs
//
// *Frame* is no longer directly allocated with *new*.
// *Interpreter* calls *FrameFactory.Create* instead.
// Evaluation of the quote instruction is provided as
// an example.
//
namespace Scheme {
    using System.IO;

    public class Interpreter {
        //…
        public object Execute() {
            for (;;) {
                object instr = List.First(_context.Expr);
                //…
                } else if (instr == Instruction.Quote) {
                    _context =
                        FrameFactory.Create(
                            List.Second(_context.Expr),
                            List.Third(_context.Expr),
                            _context
                        );
                }
                //…
            }
        }
        //…
    }
}
```

A.2.4 Stack Reconstruction Strategy

```
//
// Compiler.cs
//
namespace Scheme {
    public class Compiler {
        private MacroExpander _macroExp;

        public Compiler(MacroExpander macroExp) {
            _macroExp = macroExp;
        }

        public Compiler() { _macroExp = null; }

        public object Compile() {
            object o = _macroExp.Expand();
            return (o == null) ? o : Compile(o);
        }

        public object Compile(object expr) {
            if (expr is Pair) {
                Pair p = (Pair)expr;
                object head = p.Head;
                object tail = p.Tail;

                if (head == SymbolCache.IF) {
                    object alternative =
                        (List.Length(tail) == 3) ?
                            Compile(List.Third(tail)) :
                            List.Create(
                                Instruction.Constant,
                                null);
                    return List.Create(
                        Instruction.Conditional,
                        Compile(List.First(tail)),
```

```
                Compile(List.Second(tail)),
                alternative);
        } else if (head == SymbolCache.LAMBDA) {
            Pair bodyList = (Pair)List.Rest(tail);
            object bodyExpr =
                CompileSequence(bodyList);
            return List.Create(Instruction.Closure,
                List.First(tail), bodyExpr);
        } else if (head == SymbolCache.SET) {
            return List.Create(Instruction.Assign,
                List.First(tail),
                Compile(List.Second(tail)));
        } else if (head == SymbolCache.DEFINE) {
            return List.Create(Instruction.Define,
                List.First(tail),
                Compile(List.Second(tail)));
        } else if (head == SymbolCache.QUOTE) {
            return List.Create(
                Instruction.Constant,
                List.First(tail));
        } else if (head == SymbolCache.BEGIN) {
            return (tail == Pair.EmptyList) ?
                List.Create(Instruction.Constant,
                    null) :
                CompileSequence(List.Reverse(
                    (Pair)tail));
        } else if (head == SymbolCache.LETREC) {
            object letrecExp =
                List.Create(Instruction.Apply,
                    List.Create(
                        Instruction.Closure,
                        Pair.EmptyList,
                        CompileLetrec(tail)),
                    Pair.EmptyList);
            return letrecExp;
        } else if (head ==
            SymbolCache.CALLWITHVALUES) {
            object producerExpr =
                List.Create(Instruction.Apply,
                    Compile(List.First(tail)),
                    Pair.EmptyList);
            object consumerExpr =
                List.Create(Instruction.Apply,
                    Compile(List.Second(tail)),
                    List.Create(
                        Instruction.ArgMultiple,
                        producerExpr,
                        Pair.EmptyList));
            return consumerExpr;
        } else {
            object applicationExpr =
                CompileApplication(head, tail);
            return applicationExpr;
        }
    } else {
        return (expr is Symbol) ?
            List.Create(
                Instruction.Variable, expr) :
            List.Create(
                Instruction.Constant, expr);
    }
}

private object CompileLetrec(object expr) {
    object letrecExpr =
        ArrangeDefinitions(List.First(expr));
    if (letrecExpr != Pair.EmptyList) {
        List.Last(letrecExpr).Tail =
            List.Rest(expr);
    } else {
        letrecExpr = List.Rest(expr);
    }
    return CompileSequence(
        List.Reverse((Pair)letrecExpr));
```

```
        }

        private object
            ArrangeDefinitions(object defExprs) {
            if (defExprs == Pair.EmptyList) {
                return defExprs;
            } else {
                object def = List.First(defExprs);
                return (def == Pair.EmptyList) ?
                    Pair.EmptyList :
                    new Pair(List.Create(
                        SymbolCache.DEFINE,
                        List.First(def), List.Second(def)),
                        ArrangeDefinitions(
                            List.Rest(defExprs)));
            }
        }

        private object CompileSequence(Pair exprList) {
            if (exprList == Pair.EmptyList) {
                return exprList;
            } else {
                object expr = List.First(exprList);
                object rest = List.Rest(exprList);

                return List.Create(
                    Instruction.Sequence, Compile(expr),
                    CompileSequence((Pair)rest));
            }
        }

        private object CompileApplication(object rator,
            object rands) {
            return List.Create(Instruction.Apply,
                Compile(rator), CompileArguments(rands));
        }

        private object CompileArguments(object rands) {
            return (rands == Pair.EmptyList) ? rands :
                List.Create(Instruction.Argument,
                    Compile(List.First(rands)),
                    CompileArguments(List.Rest(rands)));
        }
    }
}

//
// Context.cs
//
namespace Scheme {
    public abstract class Context {
        protected Interpreter _i;
        protected object _expr;
        protected Environment _env;

        public Context() {
            _i = null;
            _expr = null;
            _env = null;
        }

        public Context(Interpreter i, Environment env) {
            _i = i;
            _expr = null;
            _env = env;
        }

        public Context(Interpreter i, object expr,
            Environment env) {
            _i = i;
            _expr = expr;
            _env = env;
        }
```

```
    public abstract object Invoke(object val);
}

public class ContextList {
    private Context _first;
    private ContextList _newer;
    private ContextList _older;

    public ContextList(Context first) {
        _first = first; _newer = null; _older = null;
    }

    public ContextList(Context first,
        ContextList newer) {
        _first = first; _newer = newer; _older = null;
    }

    public ContextList(Context first,
        ContextList newer, ContextList older) {
        _first = first; _newer = newer; _older = older;
    }

    public ContextList CopyStack() {
        return Copy(null);
    }

    private ContextList Copy(ContextList newer) {
        ContextList newStack =
            new ContextList(_first, newer);
        newStack._older = (_older == null) ?
            _older : _older.Copy(newStack);

        return newStack;
    }

    public ContextList Bottom() {
         return (_older == null) ?
             this : _older.Bottom();
    }

    public Context Current {
        get { return _first; }
    }

    public ContextList Newer {
        get { return _newer; }
        set { _newer = value; }
    }

    public ContextList Older {
        get { return _older; }
        set { _older = value; }
    }

    public override string ToString() {
        string s = _first.ToString();
        if (_newer != null) {
            s += "\n" + _newer.ToString();
        }
        return s;
    }
}

public class HaltContext : Context {
    public HaltContext() { }

    public override object Invoke(object val) {
        return val;
    }

    public override string ToString() {
        return "HaltContext";
    }
}
```

```
public class CondContext : Context {
    private object _texpr;
    private object _fexpr;

    public CondContext(Interpreter i, object texpr,
        object fexpr, Environment env) : base(i, env) {
        _texpr = texpr;
        _fexpr = fexpr;
    }

    public override object Invoke(object val) {
        return _i.Execute(Boolean.IsTrue(val) ?
            _texpr : _fexpr, _env);
    }

    public override string ToString() {
        return "CondContext: [T] " +
            _texpr.ToString() +
            " [F] " + _fexpr.ToString();
    }
}

public class AssignContext : Context {
    private Symbol _id;

    public AssignContext(Interpreter i, Symbol id,
        Environment env) : base(i, env) {
        _id = id;
    }

    public override object Invoke(object val) {
        if (_env.Assign(_id, val)) {
            return null;
        } else {
            throw new System.Exception(
                "Cannot set undefined identifier " +
                _id
            );
        }
    }

    public override string ToString() {
        return "AssignContext: " + _id.ToString();
    }
}

public class DefineContext : Context {
    private Symbol _id;

    public DefineContext(Interpreter i, Symbol id,
        Environment env) : base(i, env) {
        _id = id;
    }

    public override object Invoke(object val) {
        _env.Define(_id, val);
        return null;
    }

    public override string ToString() {
        return "DefineContext: " + _id.ToString();
    }
}

public class ArgContext : Context {
    public ArgContext(Interpreter i, object expr,
        Environment env) : base(i, expr, env) { }

    public override object Invoke(object val) {
        object result;
        try {
            result = _i.Execute(_expr, _env);
        } catch (CaptureException capEx) {
```

```
            capEx.AddContext(new RibContext(val));
            throw;
        }
        return new Pair(result, val);
    }

    public override string ToString() {
        return "ArgContext: " + _expr.ToString();
    }
}

public class RibContext : Context {
    private object _valueRib;

    public RibContext(object valueRib) {
        _valueRib = valueRib;
    }

    public override object Invoke(object val) {
        return new Pair(val, _valueRib);
    }

    public override string ToString() {
        return "RibContext: " + _valueRib.ToString();
    }
}

public class FuncContext : Context {
    public FuncContext(Interpreter i, object expr,
        Environment env) : base(i, expr, env) { }

    public override object Invoke(object val) {
        Pair vRib = (Pair)val;
        object fn = _i.Execute(_expr, _env);
        Procedure proc = (Procedure)fn;
        return proc.Apply(_i, vRib, _env);
    }

    public override string ToString() {
        return "FuncContext: " + _expr.ToString();
    }
}

public class ApplyContext : Context {
    private Pair _valueRib;

    public ApplyContext(Interpreter i, Pair valueRib,
        Environment env) : base(i, env) {
        _valueRib = valueRib;
    }

    public override object Invoke(object val) {
        Procedure proc = (Procedure)val;
        return proc.Apply(_i, _valueRib, _env);
    }

    public override string ToString() {
        return "ApplyContext: " + _valueRib.ToString();
    }
}

public class SeqContext : Context {
    public SeqContext(Interpreter i, object expr,
        Environment env) : base(i, expr, env) { }

    public override object Invoke(object val) {
        return _i.Execute(_expr, _env);
    }

    public override string ToString() {
        return "SeqContext: " + _expr.ToString();
    }
}
```

```
    public class MultArgContext : Context {
        public MultArgContext() { }

        public override object Invoke(object multArgVal) {
            Pair newRib = (multArgVal is ValueList) ?
                ((ValueList)multArgVal).Values :
                (multArgVal != null) ?
                    new Pair(multArgVal, Pair.EmptyList) :
                    Pair.EmptyList;
            return newRib;
        }

        public override string ToString() {
            return "MultArgContext";
        }
    }
}

//
// Continuation.cs
//
namespace Scheme {
    public class Continuation : Procedure {
        private ContextList _cList;

        public Continuation(ContextList cList) {
            _cList = cList;
        }

        public object Apply(Interpreter i, Pair args,
            Environment env) {
            InvokeException invEx =
                new InvokeException(
                    List.First(args), _cList);
            throw invEx;
        }
    }

    public class InvokeException : System.Exception {
        private object _val;
        private ContextList _cList;

        public InvokeException(object val,
            ContextList cList) {
            _val = val;
            _cList = cList;
        }

        public object ReturnValue {
            get { return _val; }
        }

        public ContextList Continuation {
            get { return _cList; }
        }
    }

    public class CaptureException : System.Exception {
        private ContextList _cList;
        private Procedure _proc;

        public CaptureException(Procedure proc) {
            _cList = null;
            _proc = proc;
        }

        public void AddContext(Context c) {
            ContextList cList = new ContextList(c, _cList);
            if (_cList != null) {
                _cList.Older = cList;
            }
            _cList = cList;
        }
```

```
        public void AppendContext(ContextList cShared) {
            if (cShared != null) {
                ContextList cList = cShared.CopyStack();

                cList.Newer = _cList;

                if (_cList != null) {
                    _cList.Older = cList;

                    _cList = cList;

                } else {
                    _cList = cList;
                }

                while (_cList.Older != null) {
                    _cList = _cList.Older;
                }
            }
        }

        public ContextList Continuation {
            get { return _cList; }
        }

        public Procedure Procedure {
            get { return _proc; }
        }
    }
}

//
// Interpreter.cs
//
namespace Scheme {
    using System.IO;

    public class Interpreter {
        private Environment _globalEnv;
        private Compiler _compiler;
        private CaptureException _capEx;
        private InvokeException _invEx;

        public Interpreter(TextReader reader) {
            StandardLibrary sl =
                StandardLibrary.Instance();
            _globalEnv = new Environment(
                sl.Symbols, sl.Values);
            _compiler =
                new Compiler(new MacroExpander(new Parser(
                    new Scanner(reader)), this));
            _capEx = null;
            _invEx = null;
        }

        public Environment GlobalEnvironment {
            get { return _globalEnv; }
        }

        public object Evaluate() {
            object o = _compiler.Compile();

            if (o == null) return o;

            return Evaluate(o, _globalEnv);
        }

        private object Evaluate(object expr,
            Environment env) {
            for (;;) {
                try {
                    return Dispatch(expr, env);
                } catch (CaptureException captureEx) {
                    _capEx = captureEx;
```

```
        } catch (InvokeException invEx) {
            _invEx = invEx;
        }
    }
}

private object Dispatch(object expr,
    Environment env) {
    if (_capEx != null) {
        CaptureException capEx = _capEx;
        _capEx = null;
        if (capEx.Continuation == null) {
            capEx.AddContext(new HaltContext());
        }
        Continuation k =
            new Continuation(capEx.Continuation);
        return
            ResumeFromCapture(capEx.Continuation,
                capEx.Procedure, k);
    } else if (_invEx != null) {
        InvokeException invEx = _invEx;
        _invEx = null;
        return ResumeFromInvoke(invEx.Continuation,
            invEx.ReturnValue);
    } else {
        return Execute(expr, env);
    }
}

private object ResumeFromCapture(ContextList cList,
    Procedure proc, Continuation k) {
    object returnValue;
    if (cList.Newer == null) {
        try {
            returnValue =
                proc.Apply(this, List.Create(k),
                    null);
        } catch (CaptureException capEx) {
            capEx.AppendContext(cList);
            throw;
        }
    } else {
        returnValue =
            ResumeFromCapture(cList.Newer, proc,
                k);
    }

    try {
        return cList.Current.Invoke(returnValue);
    } catch (CaptureException capEx) {
        capEx.AppendContext(cList.Older);
        throw;
    }
}

private object ResumeFromInvoke(ContextList cList,
    object val) {
    object returnValue;
    if (cList.Newer == null) {
        returnValue = val;
    } else {
        returnValue =
            ResumeFromInvoke(cList.Newer, val);
    }

    try {
        return cList.Current.Invoke(returnValue);
    } catch (CaptureException capEx) {
        capEx.AppendContext(cList.Older);
        throw;
    }
}

public object Execute(object expr,
```

```
Environment env) {
object instr = List.First(expr);

if (instr == Instruction.Constant) {
    return List.Second(expr);
} else if (instr == Instruction.Variable) {
    object sym = List.Second(expr);
    object varValue = env.Lookup((Symbol)sym);
    if (varValue == null) {
        throw new System.Exception(
            "No value associated with id " +
            sym
        );
    } else {
        return varValue;
    }
} else if (instr == Instruction.Conditional) {
    object cond = null;
    try {
        cond = Execute(List.Second(expr), env);
    } catch (CaptureException capEx) {
        capEx.AddContext(new CondContext(this,
            List.Third(expr),
            List.Fourth(expr),
            env));
        throw capEx;
    }

    object condResult =
        Execute(Boolean.IsTrue(cond) ?
            List.Third(expr) :
            List.Fourth(expr), env);
    return condResult;
} else if (instr == Instruction.Closure) {
    return new Closure((Pair)List.Second(expr),
        (Pair)List.Third(expr), env);
} else if (instr == Instruction.Assign) {
    object assignVal = null;
    Symbol id = (Symbol)List.Second(expr);
    try {
        assignVal =
            Execute(List.Third(expr), env);
    } catch (CaptureException capEx) {
        capEx.AddContext(
            new AssignContext(this,
                id, env));
        throw;
    }
    if (env.Assign(id, assignVal)) {
        return null;
    } else {
        throw new System.Exception(
            "Cannot set undefined id " + id
        );
    }
} else if (instr == Instruction.Define) {
    object defineVal = null;
    Symbol id = (Symbol)List.Second(expr);
    try {
        defineVal =
            Execute(List.Third(expr), env);
    } catch (CaptureException capEx) {
        capEx.AddContext(
            new DefineContext(this,
                id, env));
        throw;
    }
    env.Define(id, defineVal);
    return null;
} else if (instr == Instruction.Argument) {
    Pair p = null;
    try {
        object argRest = List.Third(expr);
        p = (argRest == Pair.EmptyList) ?
```

```
                Pair.EmptyList :
                (Pair)Execute(argRest, env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new ArgContext(this,
                    List.Second(expr), env));
                throw;
            }
            object argVal;
            try {
                argVal =
                    Execute(List.Second(expr), env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new RibContext(p));
                throw;
            }
            return new Pair(argVal, p);
        } else if (instr == Instruction.Apply) {
            Pair vRib = null;
            try {
                object fnArgs = List.Third(expr);
                vRib = (fnArgs == Pair.EmptyList) ?
                    Pair.EmptyList :
                    (Pair)Execute(fnArgs, env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new FuncContext(this,
                    List.Second(expr), env));
                throw;
            }
            object fn = null;
            try {
                fn = Execute(List.Second(expr), env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new ApplyContext(this,
                    vRib, env));
                throw;
            }
            Procedure proc = (Procedure)fn;
            return proc.Apply(this, vRib, env);
        } else if (instr == Instruction.ArgMultiple) {
            object multArgVal;
            try {
                multArgVal =
                    Execute(List.Second(expr), env);
            } catch (CaptureException capEx) {
                capEx.AddContext(new MultArgContext());
                throw;
            }
            Pair newRib = (multArgVal is ValueList) ?
                ((ValueList)multArgVal).Values :
                (multArgVal != null) ?
                    new Pair(multArgVal,
                        Pair.EmptyList) :
                    Pair.EmptyList;
            return newRib;
        } else if (instr == Instruction.Sequence) {
            try {
                object restSeq = List.Third(expr);
                if (restSeq != Pair.EmptyList) {
                    Execute(restSeq, env);
                }
            } catch (CaptureException capEx) {
                capEx.AddContext(new SeqContext(this,
                    List.Second(expr), env));
                throw;
            }
            return Execute(List.Second(expr), env);
        } else {
            throw new System.Exception(
                "Unexpected value: " +
                ((instr == null) ? "[null]" : instr)
            );
        }
    }
```

```
        public void Read(TextReader reader) {
            Read(reader, _globalEnv);
        }

        public void Read(TextReader reader,
            Environment env) {
            Compiler compiler =
                new Compiler(new MacroExpander(
                new Parser(new Scanner(reader)), this));
            object o;
            while ((o = compiler.Compile()) != null) {
                try {
                    Evaluate(o, env);
                } catch (System.Exception e) {
                    System.Console.WriteLine(
                        e.Message + "\n" +
                        e.StackTrace);
                }
            }
        }
    }
}

//
// StandardLibrary.cs
//
// The stack reconstruction standard library implementation
// is the same as the implementation for the
// simple exception strategy with the following changes.
//
namespace Scheme {
    public class CallCC : Procedure {
        public object Apply(Interpreter i, Pair args,
            Environment env) {
            throw new CaptureException(
                (Procedure)List.First(args));
        }
    }

    // CompileProc is bound to the name _compile in
    // the global environment
    public class CompileProc : Procedure {
        public object Apply(Interpreter i, Pair args,
            Environment env) {
            object expr = List.First(args);
            Compiler c = new Compiler();
            return c.Compile(expr);
        }
    }
}
```

A.2.5 Thread Strategy

```
//
// Continuation.cs
//
namespace Scheme {
    using System.Threading;

    public class Continuation : Procedure {
        private Executor _ex;

        public object Apply(Executor i, Pair args,
            Environment env) {
            _ex.ReturnValue = List.First(args);

            Monitor.Pulse(_ex);
            Monitor.Exit(_ex);

            throw new QuitException();
        }
```

```
        public Executor JoinPoint {
            set { _ex = value; }
        }
    }
}

//
// Interpreter.cs
//
namespace Scheme {
    using System.IO;
    using System.Threading;

    public class QuitException : System.Exception { }

    public class Executor {
        private object _expr;
        private Environment _env;
        private object _returnValue;

        public Executor(object expr, Environment env) {
            _expr = expr; _env = env; _returnValue = null;
        }

        public object ReturnValue {
            get { return _returnValue; }
            set { _returnValue = value; }
        }

        public Executor() {
            _expr = null; _env = null;
        }

        public void Execute() {
            Monitor.Enter(this);
            try {
                Execute(_expr, _env);
            } catch (QuitException) {
            } catch (System.Exception e) {
                System.Console.WriteLine(e.Message);
                System.Console.WriteLine(e.StackTrace);
            }
        }

        public object Execute(object expr,
            Environment env) {
            object instr = List.First(expr);

            if (instr == Instruction.Constant) {
                return List.Second(expr);
            } else if (instr == Instruction.Variable) {
                object sym = List.Second(expr);
                object varValue = env.Lookup((Symbol)sym);
                if (varValue == null) {
                    throw new System.Exception(
                        "No value associated " +
                        "with symbol " + sym
                    );
                } else {
                    return varValue;
                }
            } else if (instr == Instruction.Conditional) {
                object cond =
                    Execute(List.Second(expr), env);
                object condResult =
                    Execute(Boolean.IsTrue(cond) ?
                        List.Third(expr) :
                        List.Fourth(expr), env);
                return condResult;
            } else if (instr == Instruction.Closure) {
                return new Closure((Pair)List.Second(expr),
                    (Pair)List.Third(expr), env);
            } else if (instr == Instruction.Assign) {
                object assignVal;
```

```
            Symbol id = (Symbol)List.Second(expr);

            assignVal = Execute(List.Third(expr), env);

            if (env.Assign(id, assignVal)) {
                return null;
            } else {
                throw new System.Exception(
                    "Cannot set undefined " +
                    "identifier " + id
                );
            }
        } else if (instr == Instruction.Define) {
            Symbol id = (Symbol)List.Second(expr);
            object defineVal =
                Execute(List.Third(expr), env);
            env.Define(id, defineVal);
            return null;
        } else if (instr == Instruction.Argument) {
            object argRest = List.Third(expr);
            Pair p = (argRest == Pair.EmptyList) ?
                Pair.EmptyList :
                (Pair)Execute(argRest, env);
            object argVal =
                Execute(List.Second(expr), env);

            return new Pair(argVal, p);
        } else if (instr == Instruction.Apply) {
            object fnArgs = List.Third(expr);
            Pair vRib = (fnArgs == Pair.EmptyList) ?
                Pair.EmptyList :
                (Pair)Execute(fnArgs, env);
            object fn =
                Execute(List.Second(expr), env);
            Procedure proc = (Procedure)fn;
            return proc.Apply(this, vRib, env);
        } else if (instr == Instruction.ArgMultiple) {
            object multArgVal =
                Execute(List.Second(expr), env);
            Pair newRib = (multArgVal is ValueList) ?
                ((ValueList)multArgVal).Values :
                (multArgVal != null) ?
                    new Pair(multArgVal,
                        Pair.EmptyList) :
                    Pair.EmptyList;
            return newRib;
        } else if (instr == Instruction.Sequence) {
            object restSeq = List.Third(expr);
            if (restSeq != Pair.EmptyList) {
                Execute(restSeq, env);
            }
            return Execute(List.Second(expr), env);
        } else if (instr == Instruction.Resume) {
            _returnValue =
                Execute(List.Second(expr), env);

            Monitor.Pulse(this);
            Monitor.Exit(this);

            throw new QuitException();
        } else {
            throw new System.Exception(
                "Unexpected value: " +
                ((instr == null) ? "[null]" : instr));
        }
    }
}

public class Interpreter {
    private Environment _globalEnv;
    private Compiler _compiler;

    public Interpreter(TextReader reader) {
        StandardLibrary sl =
```

```
                StandardLibrary.Instance();
            _globalEnv =
                new Environment(sl.Symbols, sl.Values);
            _compiler = new Compiler(
                new MacroExpander(new Parser(
                    new Scanner(reader)), _globalEnv));
        }

        public Environment GlobalEnvironment {
            get { return _globalEnv; }
        }

        public object Evaluate() {
            object o = _compiler.Compile();

            if (o == null) return o;

            return Evaluate(o, _globalEnv);
        }

        private static object Evaluate(object expr,
            Environment env) {
            Executor ex = new Executor(
                List.Create(Instruction.Resume,
                    expr), env);
            Thread th =
                new Thread(new ThreadStart(ex.Execute));

            Monitor.Enter(ex);

            th.Start();

            Monitor.Pulse(ex);
            Monitor.Wait(ex);

            return ex.ReturnValue;
        }

        public void Read(TextReader reader) {
            Read(reader, _globalEnv);
        }

        public static void Read(TextReader reader,
            Environment env) {
            Compiler compiler =
                new Compiler(new MacroExpander(
                new Parser(new Scanner(reader)), env));
            object o;
            while ((o = compiler.Compile()) != null) {
                try {
                    Evaluate(o, env);
                } catch (System.Exception e) {
                    System.Console.WriteLine(
                        e.Message + "\n" + e.StackTrace
                    );
                }
            }
        }
    }
}

//
// StandardLibrary.cs
//
// The thread strategy standard library implementation
// is the same as the stack reconstruction strategy
// with the exception of *call/cc*.
//
namespace Scheme {
    public class CallCC : Procedure {
        public object Apply(Executor i, Pair args,
            Environment env) {
            Continuation k = new Continuation();
```

```
            Pair newInstr = List.Create(Instruction.Apply,
                List.Create(Instruction.Constant,
                    List.First(args)),
                List.Create(Instruction.Argument,
                    List.Create(Instruction.Constant, k),
                    Pair.EmptyList));

            Executor ex = new Executor(
                List.Create(Instruction.Resume,
                    newInstr), env);
            k.JoinPoint = ex;
            Thread t =
                new Thread(new ThreadStart(ex.Execute));

            Monitor.Enter(ex);

            t.Start();

            Monitor.Pulse(ex);
            Monitor.Wait(ex);

            return ex.ReturnValue;
        }
    }
}
```

A.3 Initialization Files

There are three interpreter initialization files. The first, Primitives.cs, introduces names into the global namespace needed by the macro implementation and defines several basic library functions. The second, psyntax.pp, is the portable syntax-case implementation of the R5RS macro system from R.K. Dybvig and O. Waddell, "Portable Syntax-Case," 2005 (http://www.cs.indiana.edu/chezscheme/syntax-case/). Version 7.0 should be downloaded from the reference address and stored in the same directory as the interpreter executable. The third, init.scm, defines Scheme functions required by the benchmarks.

```
//
// Primitives.cs
//
namespace Scheme {
    using System.IO;

    public class Primitives {
        public static readonly StringReader Reader;

        static Primitives() {
            string s = @"

(define $sc-put-cte #f)
(define sc-expand #f)
(define $make-environment #f)
(define environment? #f)
(define interaction-environment #f)
(define identifier? #f)
(define datum->syntax-object #f)
(define syntax->list #f)
(define syntax-object->datum #f)
(define generate-temporaries #f)
(define free-identifier=? #f)
(define bound-identifier=? #f)
(define literal-identifier=? #f)
(define syntax-error #f)
(define $syntax-dispatch #f)

(define _expander
```

```
  (lambda (expr) expr))

(define eval
  (lambda (expr)
    (_native-eval
      (_compile
        (_expander expr)))))

(define caar
  (lambda (x)
    (car (car x))))

(define cadr
  (lambda (x)
    (car (cdr x))))

(define not (lambda (x) (if x #f #t)))

;
; The following functions should also be defined here:
;
; (andmap f first . rest)
; (map f ls . more)
; (reverse ls)
; (memv x ls)
; (memq x ls)
; (assq x ls)
; (equal? x y)
;
; For a definition of the first, see [53].
;
; For standard definitions of the remaining
; six functions, see a reference such as [56].
;

            ";
            Reader = new StringReader(s);
        }
    }
}

;;;
;;; init.scm
;;;

(define call/cc call-with-current-continuation)

(set! _expander sc-expand)

; Already defined: cadr caar
(define cdar
  (lambda (ls) (cdr (car ls))))
(define cddr
  (lambda (ls) (cdr (cdr ls))))
(define caddr
  (lambda (ls) (car (cddr ls))))

(define time
  (lambda (proc)
    (let ((start (_get-ticks)))
      (let ((result (proc)))
        (let ((stop (_get-ticks)))
          (values result (- stop start)))))))

(define run-benchmark
  (lambda (name limit proc pred)
    (display "----------------------------------------")
      (newline)
      (display "Benchmark: ") (display name) (newline)
      (display "Iterations: ") (display limit) (newline)
      (let ((total 0)
            (high 0)
            (low 999999999999))
        (let loop ((n 0))
```

```
          (if (< n limit)
              (begin
                (call-with-values (lambda () (time proc))
                  (lambda (result ticks)
                    (display result) (newline)
                    (set! total (+ total ticks))
                    (if (< high ticks)
                        (set! high ticks))
                    (if (< ticks low)
                        (set! low ticks))))
                (loop (+ n 1)))
              (begin (set! total (/ total 10))
                     (display "Total: ")
                     (display total) (newline)
                     (display "Lowest: ")
                     (display (/ low 10)) (newline)
                     (display "Highest: ")
                     (display (/ high 10)) (newline)
                     (display "Average: ")
                     (display (/ total n))
                     (newline)))))))

(define newline
  (lambda ()
    (display (string #\newline))))

;
; init.scm must include a definition of
;
; (define-macro x)
;
; where define-macro is defined in terms of the
; portable syntax-case implementation from [53]. The
; source code for SISC [40] provides a good example
; of how this may be done.
;
```

A.4 Benchmarks

```
;;;
;;; callcc.scm
;;;

(load "../test/coroutines.scm")
(load "../test/exceptions.scm")
(load "../test/benchmark.scm")

(run-benchmark "exception" 1000
               (lambda () (test-driver)) #t)
(run-benchmark "coroutine" 1000
               (lambda () (get-fibo-list 2000)) #t)
(run-benchmark "tak" 20
               (lambda () (tak 18 12 6)) #t)
(run-benchmark "ctak" 20
               (lambda () (ctak 18 12 6)) #t)
(run-benchmark "capture-k" 1000
               (lambda () (capture-k 100)) #t)
(run-benchmark "invoke-k" 1000
               (lambda () (invoke-k 100)) #t)

;;;
;;; coroutines.scm
;;;

(define-macro coroutine
  (lambda (x . body)
    '(letrec ((local-control-state
               (lambda (,x) ,@body))
              (resume
                (lambda (c v)
                  (call/cc
                    (lambda (k)
```

```
                      (set! local-control-state k)
                      (c v))))))
       (lambda (v)
         (local-control-state v)))))

;;;
;;; exceptions.scm
;;;

(define *handlers* '())

(define-syntax try
  (syntax-rules ()
    ((try exp (catch name handler))
     (lambda ()
       (call/cc
         (lambda (k)
           (let ((saved-handlers *handlers*))
             (set! *handlers* (cons (list name k handler)
                                    *handlers*))
             exp
             (set! *handlers* saved-handlers)))))))))

(define throw
  (lambda (exception)
    (let ((handler-record (assq exception *handlers*)))
      (if handler-record
          (let ((error-k (cadr handler-record))
                (error-handler (caddr handler-record)))
            (error-k (error-handler exception)))
          "Error"))))

;;;
;;; benchmark.scm
;;;

(define safe-div
  (lambda (x y)
    (if (= y 0) (throw 'divide-by-zero) (/ x y))))

(define test-driver
  (try
    (let loop ((dividend 10))
      (let ((decr-div (- dividend 1)))
        (safe-div dividend decr-div)
        (loop decr-div)))
  (catch 'divide-by-zero
    (lambda (exception) "Divide-by-zero error"))))

(define make-fibo-coroutine
  (lambda (limit consumer-cor)
    (coroutine no-init
      (let fibo* ((curr 1) (prev 0))
        (let ((new-val (+ curr prev)))
          (if (> new-val limit)
              (resume consumer-cor 'done)
              (begin
                (resume consumer-cor new-val)
                (fibo* new-val curr)))))))))

(define make-list-coroutine
  (lambda (producer-cor)
    (coroutine no-initial-val
      (let loop ()
        (let ((val (resume producer-cor 'get-next-val)))
          (if (eq? val 'done)
              '()
              (cons val (loop)))))))))

(define get-fibo-list
  (lambda (limit)
    (letrec ((fibo-cor
               (make-fibo-coroutine
                 limit
```

```
                (lambda (v) (list-cor v))))
            (list-cor
              (make-list-coroutine
                (lambda (v) (fibo-cor v)))))
      (list-cor 'start))))

(define tak
  (lambda (x y z)
    (if (not (< y x))
        z
        (tak (tak (- x 1) y z)
             (tak (- y 1) z x)
             (tak (- z 1) x y)))))

(define ctak
  (lambda (x y z)
    (call/cc
     (lambda (k)
       (ctak+k k x y z)))))

(define ctak+k
  (lambda (k x y z)
    (cond ((not (< y x))
           (k z))
          (else
           (call/cc
            (lambda (k)
              (ctak+k
               k
               (call/cc
                (lambda (k) (ctak+k k (- x 1) y z)))
               (call/cc
                (lambda (k) (ctak+k k (- y 1) z x)))
               (call/cc
                (lambda (k) (ctak+k k (- z 1) x y)))))))))))

(define capture-k
  (lambda (limit)
    (letrec ((capture-k*
          (lambda (n)
        (if (< n limit)
            (call/cc (lambda (k) (capture-k* (+ n 1))))
            'done))))
      (capture-k* 0))))

(define invoke-k
  (lambda (limit)
    (let ((n 0)
          (loop 0))
      (call/cc (lambda (k) (set! loop k)))
      (if (< n limit)
          (begin (set! n (+ n 1))
                 (loop 'ignored)))
          'done)))
```