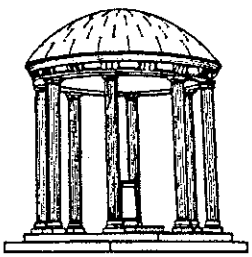
Three Implementation Models for Scheme

TR87-011

1987

R. Kent Dybvig



The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

UNC is an Equal Opportunity/Affirmative Action Institution.

Three Implementation Models for Scheme

by

R. Kent Dybvig

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1987

Approved by:

Gyula Magó
Advisor

Richard Snodgrass
Reader

David A. Plaisted
Reader

© 1987
R. Kent Dybvig
ALL RIGHTS RESERVED

R. KENT DYBVIG. Three Implementation Models for Scheme (Under the direction of GYULA A. MAGO.)

Abstract

This dissertation presents three implementation models for the Scheme Programming Language. The first is a heap-based model used in some form in most Scheme implementations to date; the second is a new stack-based model that is considerably more efficient than the heap-based model at executing most programs; and the third is a new string-based model intended for use in a multiple-processor implementation of Scheme. The heap-based model allocates several important data structures in a heap, including actual parameter lists, binding environments, and call frames. The stack-based model allocates these same structures on a stack whenever possible. This results in less heap allocation, fewer memory references, shorter instruction sequences, less garbage collection, and more efficient use of memory. The string-based model allocates versions of these structures right in the program text, which is represented as a string of symbols. In the string-based model, Scheme programs are translated into an FFP language designed specifically to support Scheme. Programs in this language are directly executed by the FFP machine, a multiple-processor string-reduction computer. The stack-based model is of immediate practical benefit; it is the model used by the author's *Chez* Scheme system, a high-performance implementation of Scheme. The string-based model will be useful for providing Scheme as a high-level alternative to FFP on the FFP machine once the machine is realized.

Acknowledgements

I would like to thank my advisor, Gyula A. Magó, for his assistance and guidance throughout my work on this project. His steadiness and patient support were essential to its completion. I appreciate his help more than he knows.

I would like to thank the other members of my committee as well: Dean Brock, Dave Plaisted, Rick Snodgrass, and Don Stanat. Each was willing to spend time discussing various facets of the research, and each offered challenges and suggestions that helped me along the way.

I would also like to thank Dan Friedman, who introduced me to Scheme and to many of the concepts of functional programming and parallel computing.

I would like to thank the many other people who have been helpful along the way, especially Bruce Smith, Dave Middleton, and Bharat Jayaraman.

I would like to thank my parents, Roger S. Dybvig and Elizabeth H. Dybvig, for their support throughout my education.

Finally, I would like to thank my wife, Susan, who deserves more appreciation than I can ever show for her support throughout my advanced education and for her assistance and patience during the writing of this dissertation.

Contents

Chapter 1 Introduction 1
1.1 Functional Programming Languages 4
1.2 Functional Programming Language Implementations 6
1.3 Multiprocessor Systems and Implementations 9
Chapter 2 The Scheme Language 13
2.1 Syntactic Forms and Primitive Functions 15
2.1.1 Core Syntactic Forms 16
2.1.2 Primitive Functions 18
2.1.3 Syntactic Extensions 23
2.2 Closures 29
2.3 Assignments 33
2.3.1 Maintaining State with Assignments 34
2.3.2 Lazy Streams 35
2.4 Continuations 36
2.5 A Meta-Circular Interpreter 39
Chapter 3 The Heap-Based Model 43
3.1 Motivation and Problems 44
3.2 Representation of Data Structures 46
3.2.1 Environments 46
3.2.2 Frames and the Control Stack 47
3.2.3 Closures and Continuations 49
3.3 Implementation Strategy 50
3.4 Implementing the Heap-Based Model 54
3.4.1 Assembly Code 55
3.4.2 Translation 56
3.4.3 Evaluation 59
3.5 Improving Variable Access 62
3.5.1 Translation 64
3.5.2 Evaluation 65

Chapter 4 The Stack-Based Model 69
4.1 Stack-Based Implementation of Block-Structured Languages . . . 71
4.1.1 Call Frames 71
4.1.2 Dynamic and Static Links 72
4.1.3 Functionals 74
4.1.4 Stack Operations 74
4.1.5 Translation 76
4.1.6 Evaluation 78
4.2 Stack Allocating the Dynamic Chain 80
4.2.1 Snapshot Continuations 81
4.2.2 Evaluation 82
4.3 Stack Allocating the Static Chain 84
4.3.1 Including Variable Values in the Call Frame 85
4.3.2 Translation and Evaluation 86
4.4 Display Closures 88
4.4.1 Displays 89
4.4.2 Creating Display Closures 90
4.4.3 Finding Free Variables 91
4.4.4 Translation 93
4.4.5 Evaluation 96
4.5 Supporting Assignments 98
4.5.1 Translation 101
4.5.2 Evaluation 105
4.6 Tail Calls 106
4.6.1 Shifting the Arguments 107
4.6.2 Translation 109
4.6.3 Evaluation 111
4.7 Potential Improvements. 113
4.7.1 Global Variables and Primitive Functions 113
4.7.2 Direct Function Invocations 114
4.7.3 Tail Recursion Optimization 114
4.7.4 Avoiding Heap Allocation of Closures 115
4.7.5 Producing Jumps in Place of Continuations 115
Chapter 5 The String-Based Model 117
5.1 FFP Languages and the FFP Machine 118
5.1.1 FFP Syntax 119
5.1.2 FFP Semantics 119

5.1.3 Examples . . . 123
5.1.4 The FFP Machine . . . 126
5.2 An FFP for Scheme . . . 129
5.2.1 Representation . . . 130
5.2.2 Compilation . . . 132
5.2.3 Evaluation . . . 134
5.3 Environment Trimming . . . 136
5.3.1 Translation . . . 137
5.3.2 Evaluation . . . 139
5.4 Assignments . . . 140
5.4.1 Representation . . . 140
5.4.2 Translation . . . 141
5.4.3 Evaluation . . . 143
5.5 Continuations . . . 144
5.5.1 Translation . . . 145
5.5.2 Evaluation . . . 146
Chapter 6 Conclusions . . . 149
Appendix A Heap-Based Vs. Stack-Based . . . 155
A.1 Empirical Comparison . . . 155
A.2 Instruction Sequences . . . 159
A.2.1 Variable Reference and Assignment . . . 161
A.2.2 Nested (Nontail) Call . . . 163
A.2.3 Tail Call . . . 165
A.2.4 Return . . . 166
A.2.5 Closure Creation . . . 167
A.2.6 Function Entry . . . 168
A.2.7 Continuation Creation . . . 170
A.2.8 Continuation Application . . . 171
Bibliography . . . 173

Chapter 1: Introduction

This dissertation presents three implementation models for Scheme programming language systems. These three models are referred to as *heap-based*, *stack-based*, and *string-based* models, because of the primary reliance of the first on heap allocation of important data structures, the reliance of the second on stack allocation, and of the third on string allocation. The heap-based model is well-known, having been employed in most Scheme implementations since Scheme's introduction in 1975 [Sus75]. The stack-based and string-based models are new, and are described here fully for the first time. The heap-based model requires the use of a heap to store call frames and variable bindings, while the stack-based and string-based models allow the use of a stack or string to hold the same information. The stack-based model avoids most of the heap allocation required by the heap-based model, reducing the amount of space and time required to execute most Scheme programs. The string-based model avoids both stack and heap allocation and facilitates concurrent evaluation of certain parts of a program. The stack-based model is intended for use on traditional single-processor computers, and the string-based model is intended for use on small-grain multiple-processor computers that execute programs by string reduction.

The author's *Chez* Scheme system, designed and implemented in 1983 and 1984, was the first to use the stack-based model. Other systems implemented since have employed some of the same techniques, including PC Scheme [Bar86] and Orbit [Kra86]. An implementation of ML [Car83, Car84], produced independently at about the same time as *Chez* Scheme, also employed some of the same techniques. The string-based model has yet to be implemented, though it has been

tested by interpretation on a sequential computer. It is expected to be employed in an implementation of Scheme for the FFP machine of Magó [Mag79, Mag79a, Mag84], as soon as this machine is realized. The FFP machine is a small-grained multiprocessor that directly executes programs written in Backus's FFP languages [Bac78].

Scheme is a variant of the Lisp programming language [McC60] based on the λ-calculus [Chu41, Cur58]. It was introduced by Steele and Sussman in 1975 and has undergone significant changes since [Sus75, Ste78, Ree86, Dyb87]. Unlike most Lisp dialects, Scheme is lexically-scoped, block-structured, supports functions as first-class data objects, and supports continuations as first-class data objects[1]. The popular Common Lisp dialect of Lisp [Ste84] was somewhat influenced by Scheme; it supports lexical scoping and first-class functions but not continuations. The ML programming language [Car83a, Mil84, Gor79] is similar in many respects to Scheme, supporting lexical scoping and first-class functions, but lacking continuations and variable assignments. Because of the similarities, many of the ideas presented in this dissertation apply to Common Lisp and ML as well as Scheme.

This dissertation presents several variants of each implementation model. These variants serve to simplify the presentation and to provide alternative models that might be useful for other languages similar, but not identical, to Scheme. Each model or variant addresses the representation of key data structures, the translation of source-level programs into object-level programs, and the evaluation of the object-level programs. The translation processes and most of the evaluation processes are described, in part, with working Scheme code, thus providing an executable specification of these processes. While it would be possible to base full implementations of Scheme on this code, most of the details have been suppressed to simplify and focus the presentation.

[1] A *first-class* object can be passed as an argument to a function, returned as the value of a function, and stored indefinitely. Most Lisp systems provide many types of first-class objects, including lists, symbols, strings, and numbers. Most other programming languages only provide scalar quantities, *e.g.*, numbers and characters, as first-class objects.

One contribution of this dissertation is practical and immediately useful. This is the description of the stack-based model for Scheme, which allows Scheme implementations on sequential computers to use a standard stack in much the same way as implementations of block-structured languages such Algol 60 [Nau63], Pascal [Jen74] and C [Ker78]. The heap-based model, used by most Scheme systems, requires call frames and binding environments to be heap-allocated, resulting in slower and more memory-intensive systems. Heap allocation of stack frames was thought to be necessary to support closures and continuations.

Another contribution is the description of a string-based model for Scheme that will allow Scheme to run efficiently on the FFP machine of Magó and on string-reduction machines in general. This will be useful once the FFP machine is realized, and may prove useful for other small-grain multiple-processors as well. A secondary contribution is the description of a new FFP to support Scheme and the corresponding implication for support of other languages using FFP and the FFP machine.

A third contribution is the detailed description and comparison of a set of alternative implementation models from the simplest heap-based systems through the most complex stack-based and string-based systems. Each may be better than the others under certain circumstances. Some are ideally suited to Scheme while others are better suited to languages that differ from Scheme in certain ways.

The stack-based and string-based models support efficient Scheme implementations partly because Scheme encourages *functional* rather than *imperative* programming techniques. That is, typical Scheme programs rely principally on functions and recursion rather than statements and loops, and they tend to use few variable assignments. Assignments are permitted, but they appear infrequently in Scheme code. The stack-based and string-based models exploit this, improving the speed of assignment-free code while possibly penalizing code that makes heavy use of assignments.

The remainder of this chapter gives background information on functional

programming languages, implementation techniques for functional languages, and related multiple-processor systems. Chapter 2 provides detailed information on Scheme, its syntax, and the features that make it worthy of study. Chapters 3, 4 and 5 present the heap-based, stack-based and string-based techniques. Chapter 6 presents the conclusions and ideas for further research. Finally, Appendix A completes the dissertation with a comparison of the stack-based model with the heap-based model. It presents the results of an empirical comparison of the two models on a set of four simple programs, and compares instruction sequences that might be generated by compilers for the two models.

1.1 Functional Programming Languages

Most Computer programs can be separated into two categories, *imperative* and *functional.* Imperative programs work by changing state in a statement by statement fashion, using statement-oriented loops and subroutines (procedures that perform side-effects and that do not necessarily return values) for program control. Functional programs work without changing state but by computing values in an expression-oriented fashion, using functions (procedures that simply compute and return values), recursion, and mapping for program control. A programming language can usually be placed into one of these two categories according to the style of the programs that can be written in the language, which is determined by the set of features provided by the language. The principal features of an imperative programming language are statements, including declarations (of procedures and variables), assignments, loop control statements, conditional statements, subroutine or function calls, and arithmetic expressions. In a functional programming language, the principal features are expressions, including binding expressions, conditional expressions, function calls, and arithmetic expressions.

Functional programming languages have several advantages over imperative programming languages:

1. Functional programs are simpler. Functional programs are built from expressions in a natural, recursive fashion. Imperative programs are built from complex statements, or commands, combined with expressions. Certain contexts require expressions while others require statements. Statements are almost never allowed within expressions.
2. Functional programs are generally easier to understand. Each piece of the program can be taken apart from its context and studied separately from the remainder of the program, because there is little or no state affecting that piece of the program.
3. Correctness proofs can be applied more easily to functional programs because of the regularity of structure and lack of state.
4. Local variables are simpler and never uninitialized in functional programs. A variable is a name for a value rather than a storage location. This value is established when the variable is bound (declared). In an imperative program, the initial assignment is typically separated from its binding (declaration).
5. Alternative evaluation orders are possible in functional programming languages. The order of execution of two independent expressions of a program is not important, and the absence of state ensures that many expressions are independent (for example, the arguments to a function application). Such expressions may even be executed in parallel. Furthermore, an expression whose value is never required need never be executed at all.

FFP, the related FP [Bac78], KRC [Tur79, Tur82], and Miranda [Tur86] are examples of functional languages.

In spite of their attractive semantic properties and potential ease of implementation on parallel computers, there are some programs that are not easily expressed in functional languages, *i.e.*, programs that require the concept of state. However, many programs are expressible and stylistically more attractive when written in a purely functional style. Some languages encourage a functional programming style but allow the use of imperative variable assignments when necessary. Such

languages support the features that make programming in a functional style possible and omit features that discourage functional programming (such as loops, gotos, and statement-oriented conditionals). Scheme is one such language, as are Lisp, ML, APL [Ive62], and ISWIM [Brg81].

Chapter 2 discusses and illustrates the use of a functional subset of Scheme, but gives some examples of problems that do not easily lend themselves to a functional style. The Scheme programs used to describe translation and evaluation in Chapters 3, 4, and 5 are written where possible in a functional style, using assignments only sparingly. They help to demonstrate that Scheme encourages programs to be written in a functional style while still allowing assignments when necessary.

1.2 Functional Programming Language Implementations

Because Scheme is closer in spirit to functional languages than to imperative languages, it is useful to consider methods commonly used to implement functional languages. Functional languages may be implemented in several different ways on a sequential computer. The most common way is the construction of an interpreter. An interpreter requires modeling of the variable environment (if any), handling of any special syntax, providing for function application, and providing any run time support (such as storage management) [Wis82]. A related alternative is to compile the source program into a lower-level language (perhaps machine code) and to interpret programs in the lower-level language.

For languages without variables, string or graph reduction is possible, either in a sequential or parallel processor. With string reduction [Ber75], the program is represented as a string of symbols. Evaluation proceeds by replacing each reducible expression with its value, working within the string itself. Graph reduction [Wad71, Rev84, Tur79] is similar, except that the program is represented by a graph with common subexpressions sharing the same node (that is, identical

subexpressions are not duplicated as they would be by the flat string representation). The main difference here is that such expressions are reduced only once.

For functional languages with variables, one alternative is the use of a combinator approach, such as described by Turner [Tur79]. Combinators are (typically) simple functions with no free variables. Two combinators, called S and K, are sufficient to describe any function in the λ-calculus [Cur58]. Once a program has been converted into a composition of S and K combinators, it may be reduced by a string or graph reduction machine. Hughes describes the use of more complex "super-combinators," to gain a more compact and efficient translation of the input program [Hug82].

When evaluating programs in a functional language, one has the opportunity to use *lazy evaluation* [Fri76, Hen76, Tur79]. Lazy evaluation, sometimes referred to as *demand-driven* or *call-by-need* evaluation, promises to evaluate only those subexpressions necessary to complete the problem. This approach is semantically valid because an expression whose value is not required, and that does not cause any side-effects (a requirement of functional languages) cannot affect the computation. Such an expression may as well be left unevaluated. Both the interpretation and combinator approaches lend themselves to lazy evaluation strategies.

Languages such as Scheme have somewhat different requirements. In particular, it is not generally possible to use lazy evaluation. Not only must every expression that may cause a side-effect be evaluated, but also the ordering of the evaluation must be preserved. With lazy evaluation, the order of evaluation is unpredictable, depending upon what is needed when. Because of the requirement for a binding environment where changes to the values of variables may be recorded, reduction mechanisms are not easily adapted to languages that allow assignments. Also, combinators cannot easily be used to remove variables if the variables can be assigned.

This leaves the direct interpretation and compilation (to traditional machine code) approaches. Many Scheme systems have been developed [Abe84, Bar86,

Cli84, Dyb83, Fri84, Kra86, Sus75, Ste77, Ste78], most of them using some combination of interpretation and compilation. There are also many descriptions of other Lisp implementations in the literature that use one or both of these approaches; these implementations are not relevant here since they do not address support for first-class closures, first-class continuations, or certain other Scheme features. Because of the need to support first-class functions and continuations, most implementations allocate call frames and binding environments in a heap.

An optimization of the typical environment structure used in Scheme programs was given by McDermott [McD80]. McDermott suggested that heap allocation of environments happen only when necessary, and was able to retain some variables on the stack. McDermott did not handle full continuations, but suggested that even in the presence of full continuations, a similar avoidance of heap allocation might be possible.

The T language developed at Yale was based on Scheme, but its designers avoided heap allocation of call frames by omitting full continuations from the early versions of the language. To quote from the 1982 Lisp Conference paper:

> "As a concession to efficient implementation on standard architectures, escape procedures are not valid outside the dynamic extent of the CATCH-expression which creates them; this ensures that the control stack behaves in a stack-like way, unlike in Scheme, where the control stack must be heap-allocated" [Ree82].

This dissertation shows that this need not be the case, as does a recent paper describing the latest implementation of T [Kra86].

Similar heap-allocated stack frames have been used in Smalltalk [Gol83, Ing78] implementations because stack frame objects may be retained indefinitely in a manner similar to general continuations.

Cardelli independently introduced a closure object nearly identical to the *display closures* described later in this dissertation [Car83, Car84]. The main difference is that Cardelli did not need to support assignment of variables. The use of *ref* cells in ML can replace the automatic generation of boxes for assigned variables described in this dissertation. There is no benefit in stack-allocating variable

bindings if the stack itself is implemented in a heap. ML does not support general continuations, so this was not a problem for Cardelli.

1.3 Multiprocessor Systems and Implementations

Various computer systems have been proposed that provide multiple processor support for the concurrent execution, or *parallel processing* of subparts of a computer program. Parallel processing is often simulated on a single processor by interleaving the execution of subparts; this is often referred to as *multi-tasking.* Parallel processing and multi-tasking systems are both considered to be distributed processing systems. These have been studied in various forms for almost two decades; the earliest works on the subject are those of Dijkstra, Brinch Hansen, and Hoare [Dij68, Bri73, Bri78, Hoa76]. A review of distributed processing languages and abstractions can be found in Filman and Friedman's text [Fil84].

General purpose[2] multiple-processor systems (termed *multiprocessors*) can be partitioned into two categories based on the size of the processor and the complexity of the communication network: *small-grain* and *large-grain.* Large-grain multiprocessors usually contain from several to several hundred processing elements (PEs), whereas small-grain processors contain anywhere from hundreds to millions of PE's. Each PE in a large-grain multiprocessor is typically the size of a minicomputer or powerful microprocessor. Each PE in a small-grain processor is typically the size of a small microprocessor or smaller, containing perhaps an ALU (arithmetic-logical unit), a few words of memory, and communication circuitry. Large-grain systems often employ a large central memory along with smaller local memories at each PE; small-grain systems usually do not have any central memory, relying only upon the local storage at the PEs and the temporary storage within the communication network.

[2] We will not discuss special-purpose multiprocessors, such as systolic arrays or SIMD (single-instruction, multiple-data) machines, since we plan to support general-purpose programming where parallelism is not obvious in the data structures or operations of the language.

One of the most important distinctions between large-grain and small-grain systems is the type of communication network. Large-grain systems tend to be well-connected (they can afford to be); that is, access to shared memory and to each other is typically through a complex communication network that allows relatively high-speed packet-switched communication between any two processors.

In contrast, small-grain systems provide high-speed, circuit-switched communication between adjacent or nearly adjacent processors, but slow communication in general between two distant processors (because the processors are not well-connected). Also, small amounts of information can be communicated efficiently; this is not typically the case in large-grain systems.

Large grain systems are most useful for programs where there is high level parallelism inherent in the structure of the program, that is, when a programmer or compiler can isolate several large sequential processes. It has yet to be seen whether there are effective methods for splitting a computation dynamically. Even with static analysis performed by a compiler, the results have not been impressive. As a result, most large-grain systems will be programmed by hand, a complex task that requires precious programmer time.

Small-grain systems should be more appropriate where the parallelism exists at a lower level, *e.g.*, at the expression level rather than at the procedure level. Decomposition, or process-splitting, may be performed dynamically, since the cost of communicating locally is minimal, and the cost of shifting within the network can be minimized by parallel movement. Although small-grain systems look more promising than large-grain systems for some purposes, they have not yet been around long enough for their value to be proven. Magó [Mag85] and Burton and Sleep [Bur81] provide convincing arguments for the viability of small-grain systems.

Several large-grain multiprocessors have been proposed for Lisp dialects. Marti and Fitch [Mar83] perform static analysis with a compiler to decompose Lisp programs; this seems to be one of the few attempts at executing Lisp programs

on a multiprocessor without explicit programmer control over parallelism. Two recent contributions, one by Halstead [Hal84] (see also [And82]) and the other by Gabriel and McCarthy [Gab84], propose languages with explicit parallel control structures of similar natures. Sugimoto, et al. [Sug83] propose to perform some automatic program decomposition dynamically, while allowing access to lower level primitives.

Many small-grain multiprocessors have been proposed to support functional languages. These include string reduction machines such as the FFP machine, graph reduction machines such as ALICE [Dar81] and AMPS [Kel79], dataflow machines [Arv77, Den79], and the shared-memory "ultracomputer" [Got83]. Although all of these multiprocessors might be considered to be small-grain systems, relative to the FFP machine the others are actually large-grain systems.

Chapter 2: The Scheme Language

Scheme is a programming language that is close in spirit to functional programming languages but similar in many ways to more traditional languages such as Algol 60. The principal similarity between Scheme and Algol 60 is that they are both *lexically-scoped, block-structured* languages. Lexical scoping means that the body of code, or *scope*, in which a variable is visible depends only upon the structure of the code and not upon the dynamic nature of the computation (as with *dynamic scoping*, which is employed by many Lisp dialects). Block structure means that scopes may be nested (in *blocks*); any statement (or expression, in Scheme) can introduce a new block with its own local variables that are visible only within that block. Scheme differs from Algol 60 in that it is an applicative order language, meaning that the subexpressions of a function application, *i.e.*, the function and argument expressions, are always evaluated before the application is performed. (In contrast, in Algol 60, evaluation of an argument passed *by name* does not occur until the argument is used.) Furthermore, Scheme supports first-class functions, or *closures.* A closure is an object that combines the function with the lexical bindings of its free variables at the time it is created. Closures are first class data objects because they may be passed as arguments to or returned as values from other functions, or stored in the system indefinitely.

Scheme also supports first-class *continuations.* A continuation is a Scheme function that embodies "the rest of the computation." The continuation of any Scheme expression (one exists for each, waiting for its value) determines what is to be done with its value. This continuation is always present, in any language implementation, since the system is able to continue from each point of the com-

putation. Scheme simply provides a mechanism for obtaining this continuation as a closure. The continuation, once obtained, can be used to continue, or restart, the computation from the point it was obtained, whether or not the computation has previously completed, *i.e.*, whether or not the continuation has been used, explicitly or implicitly. This is useful for nonlocal exits in handling exceptions, or in the implementation of complex control structures such as coroutines or tasks.

Scheme and Lisp are based on the *λ-calculus*. The λ-calculus is a mathematical language for studying functions. In 1982, Rosser summarized the history of the λ-calculus, which has it roots in the work of Frege in 1893 and Schonfinkel in 1924 [Ros82]. Since then, Church and Curry have worked extensively with the λ-calculus and related matters [Chu41, Cur58]. Knowledge of the λ-calculus is not essential for the reading of this dissertation, although it is certainly useful in understanding the full power and importance of Lisp and Scheme.

McCarthy introduced Lisp around 1960 (making it the second oldest computer programming language in use today, next to Fortran) [McC60]. McCarthy's original language has often been referred to as *pure* Lisp because it was entirely expression-oriented. Subsequent dialects of Lisp have included many different sorts of imperative control structures. Early Lisp dialects employed dynamic scoping and disallowed first-class functions, a definite departure from the λ-calculus.

Sussman and Steele introduced the Scheme dialect of Lisp in 1975 [Sus75]. Scheme more accurately reflects the λ-calculus by supporting lexical scoping and first-class functions. Because lexical scoping is employed rather than the traditional dynamic scoping, Scheme is similar in many ways to Algol 60 and other block-structured languages such as Pascal and C.

Steele describes Common Lisp as a combination of many different Lisp dialects [Ste84]. Although Common Lisp is a much larger dialect of Lisp, it is based in part on Scheme. Unlike all popular Lisp dialects except Scheme, Common Lisp provides lexical scoping and first class functions.