# Principles of Programming Languages - Scheme

William Osborne - u1603746

February 19, 2019

## 1 Introduction

Scheme is a dialect of Lisp, supporting a variety of programming paradigms. It uses Lisp's distinctive fully-parenthesised syntax, giving it the property of homoiconicity. This allows for a powerful syntactic extension system, making the language very expressive. Scheme also provides direct access to continuations as first-class objects, allowing for a wide variety of control structures to be implemented. Both articles to be critiqued focus on the conciseness and expressive power of Scheme. The first paper to be critiqued is the fourth revision of the Scheme language specification, which emphasises a minimal set of powerful language features. The second paper discusses the applications of the continuation system, demonstrating examples of its expressive power.

## 2 Revised$^4$ Report Language on the Algorithmic Language Scheme

The Revised$^4$ Report on the Algorithmic Language Scheme [1], widely referred to as R4RS, is the fourth revision of the de-facto language specification for Scheme. Its introduction presents a concise overview of the syntax and semantics for the language, as well as detailing common conventions such as indentation style. The main body of the report formally specifies the language in detail, providing a context-free grammar and full denotational semantics for the language. R4RS is not the latest version, but was chosen due to its simplicity compared to later iterations of the specification. It was also the first iteration to propose a hygienic macro system.

Scheme is known for its simplicity, and this is reflected in R4RS. The introduction to R4RS begins with the following quote: "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary." R4RS is extremely brief compared to the specification documents of most other languages - at only 55 pages long, it is shorter than the *contents page* for the latest Common Lisp specification. Scheme's core syntax is extremely brief, as its core is a dynamically-typed $\lambda$-calculus. An example of this minimalism is that Scheme lacks any dedicated syntax for loops; instead, it requires any implementation to be properly tail-recursive, so that recursion can be used without accumulating stack frames.

R4RS proposes in its appendices an abstraction to a hygienic macro system, `define-syntax`. Prior to this, the only *standard* macro facility was similar to the system used in older Lisp languages. Lisp macros are essentially functions; the difference being that instead of executing code, they return unevaluated Lisp forms. The macros are then expanded to their return value at compile-time. This system is more expressive than the convenient but limited `syntax-rules` system initially proposed by R4RS. Scheme did not catch up until the lower-level `syntax-case` system was added, as proposed by Hieb, Dybvig and Bruggeman in 1993 [5].

However, Lisp macros have a major downside. Since they works with raw Lisp forms ("S-expressions"), they cannot distinguish between bindings introduced inside and outside the macro and thus do not respect lexical scope. This can cause existing identifiers to be rebound or "shadowed" by a macro, which when combined with dynamic typing often produces elusive bugs. This problem can be avoided by generating new names for identifiers with the `gensym` procedure, which returns a new symbol guaranteed to be unused. However, proper use of `gensym` requires considerable care on the part of the programmer.

Scheme's macro system as proposed by R4RS is *hygienic*, which means Scheme macros are guaranteed to preserve existing bindings as part of the macro-expansion process. Scheme macros operate on *syntax objects* instead of raw S-expressions, which allows their origin to be preserved when transforming the syntax with $\alpha$-conversion.

Major criticisms of R4RS and the Scheme it defines stem from its minimal approach to specification. Several features considered crucial in other languages are deliberately omitted, passing the design work onto the plethora of Scheme implementations. R4RS does not define any kind of module system; indeed, the only provision for multi-file programs is the `load` function, which simply executes a specified file and loads any new definitions. The result of this is that different implementations tend to have incompatible module systems; modules in Guile Scheme cannot be used in CHICKEN programs, for example. Similarly, very few error handling systems are specified. This is less of an issue, since Scheme's support for continuations allow most of these mechanisms to be implemented in pure Scheme, but it still makes Scheme code less portable between implementations.

R6RS, published in 2009, attempts to resolve issues with these missing features by including them in the language. However, this has resulted in a huge size increase: by page count R6RS is 340% larger than R5RS. This has drawn widespread criticism and divided the Scheme community. A more successful endeavour to extend the language has been the *Scheme Requests For Implementation*, or SRFIs for short. Each SRFI precisely defines a standard for a language feature, such as hash-tables or a richer standard list library. Implementations can then support whichever SRFIs they deem worth including.

## 3 Applications of Continuations

In his paper *Applications of Continuations* [2], Friedman introduces the concepts of escape procedures and continuations. An escape procedure returns to the top level after it terminates, ignoring the rest of the call stack. Escape procedures are implemented using continuations, which are created with the `call-with-current-continuation` procedure - often abbreviated to `call/cc`. Friedman introduces the `lambda^` notation for such procedures and details their semantics. A variety of powerful control structures are then derived to demonstrate the expressive power of first-class continuations, including breakpoints and synchronous processes.

Scheme's minimal approach to language specification would not be practical if it were only as expressive as larger languages like C++ or even Common Lisp. Support for first-class continuations allows for the implementation of almost any control structure in pure Scheme. Friedman first demonstrates this with the Lisp procedure `BREAK`, which essentially provides the ability to set breakpoints, pause the code, and resume from the REPL at the programmer's convenience. This functionality is concisely implemented in six lines of Scheme, and demonstrates how Friedman's `lambda^` notation can be very expressive when used appropriately.

Friedman then details the implementation of `lambda^` using an intermediate function, `INVOKE/NO-CONT`. `INVOKE/NO-CONT` runs a Scheme function and then returns to the top level, ignoring the existing call stack. This highlights a very powerful detail of Scheme's continuations - that they can be stored as objects and activated later in the program, after the creating procedure has terminated.

Friedman briefly mentions *Continuation-Passing Style*, an alternative way of writing code commonly used as an Intermediate Representation in the implementation of compilers for functional programming languages [3]. In CPS, procedures never return, instead passing their result onto an extra continuation argument. Friedman makes the point that continuations need not be added formally to the language: CPS can be used to achieve the same results. Use of `call/cc`, however, allows the remainder of the program to be written in the more familiar *Direct Style*, which is much easier to understand and write.

A simple example of metaprogramming is provided with the `CYCLE` function, providing a clear example of continuations being used alongside Scheme code to add new language features. However, it is not a feature that would ever be used in a real Scheme program; adding `while` loops to a language specifically designed to avoid them is counter-intuitive.

A major criticism of the paper is that it does not discuss the performance implications of using continuations in Scheme code. Oleg Kiselyov argues that using `call/cc` alone is dangerous: "Offering `call/cc` as a core control feature in terms of which all other control facilities should be implemented turns out a bad idea. Performance, memory and resource leaks, ease of implementation, ease of use, ease of reasoning all argue against `call/cc`." [6] He instead promotes the use of *delimited* continuations, which capture the call stack only up to a certain point. This has several advantages, partly because they allow the continuation to be *reified* to what is essentially a function. Additionally, the `call/cc` function itself can be very expensive, depending on the execution model used by the underlying Scheme implementation. Some discussion of the downsides of continuation-based programming - such as confusing bugs it can cause - would be useful.

A limitation of *Applications of Continuations* as a learning tool in 2019 is due to its age: none of the Scheme example code (originally published in 1988) runs without modification in modern Schemes. Some of this is due to changes in scoping rules (defining new values with `set!` is not permitted), while all the macros used in the paper use the archaic `extend-syntax` function, proposed in 1986 by Kohlbecker [4]. The paper was written before the initial proposal of the modern Scheme macro system, the syntax of which was included as an appendix in R4RS [1] in 1991, and fully detailed by Diebvig, Hieb and Bruggeman [5] in 1993. However, this is not made clear in the article; some reference to the system used would aid the reader significantly in implementing the examples. That said, these issues are mostly unavoidable without revising the article, so this is more of a limitation than a fault of the writer.

## 4 Conclusion

These articles together provides a substantial introduction to the distinctive features of Scheme. R4RS presents a clear and concise specification of Scheme, offering language features with unmatched expressive power. This presents issues, however; the lack of an official module system or error-handling mechanism reduces the portability of Scheme code. It defines a hygienic macro system, improving on previous Lisps. The new syntax transformers do not entirely replace the existing macro system, since they lack the expressive power of raw procedures; this issue is addressed in later versions of the specification.

Applications of Continuations introduces a variety of concepts in continuation-based programming, including escape procedures and Continuation-Passing Style. Friedman implements a wide variety of common control constructs using continuations (including breakpoints and synchonous processes) in order to demonstrate their expressive power. The scope of the article is limited, however, in that it does not discuss the downsides of this approach to programming.

## References

[1] Harold Abelson, RK Dybvig, CT Haynes, GJ Rozas, NI Adams IV, DP Friedman, E Kohlbecker, GL Steele Jr, DH Bartley, R Halstead, et al. Revised 4 report on the algorithmic language scheme. *ACM SIGPLAN Lisp Pointers*, 4(3):1–55, 1991.

[2] Daniel P Friedman. Applications of continuations. In *Proceedings of the ACM Conference on Principles of Programming Languages*, 1988.

[3] Guy L Steele Jr. Rabbit: A compiler for scheme. 1978.

[4] E E Kohlbecker. *Syntactic Extensions in the Programming Language LISP*. PhD thesis, Indianapolis, IN, USA, 1986. UMI Order No. GAX86-27998.

[5] R Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and symbolic computation*, 5(4):295–326, 1993.

[6] O Kiselyov. An argument against call/cc. `http://okmij.org/ftp/continuations/against-callcc.html`. Accessed: 2019-02-13.

## 5 Discussion of Presentations

### 5.1 The History and Semantics of R - Amaris Paryag

In her presentation Amaris discussed the history of R, with a focus on its semantics and lexical scoping. Tools and R's package system were also described.

Amaris described the development of R - using semantics and scoping rules derived from Scheme, with syntax similar to the existing statistical language S. Indeed, the point was made that most S code runs unaltered in R. A timeline was used to clearly show the stages of R's conception, and the development of its package sytem, CRAN. Scheme's semantic model, lexical scope, was explained in detail, using examples to demonstrate the differences between static and lexical scoping as well as the advantages it provides.

One criticism of the presentation was that it did not discuss unique features of R; it primarily focused on the details of its semantics and history. The presentation was not aimed at listeners who were unfamiliar with R. While this is not an issue per se, many listeners in the session will not have used the language before.

### 5.2 An Introduction to Lua - Jacob Taylor

Jacob's presentation provided a comprehensive and clear introduction to the Lua language. A variety of programming paradigms were explained, with reference to the article "Programming with Multiple Paradigms in Lua". Finally, more advanced concepts such as coroutines were described.

After an initial primer on Lua's syntax and single-pass interpreter, Jacob described in detail the expressive power and efficiency of Lua's tables. Object-oriented programming in Lua was introduced, explaining the concept of metatables and colon syntax. Functional programming was explained with an summary of Lua's type system, highlighting its support for functions as first-class values.

Finally, more advanced topics such as coroutines (for co-operative threading) and true parallelism were discussed; the importance of combining Lua with another runtime such as C to "build your own language" was also highlighted.

Overall, all the points required to understand Lua were covered clearly and concisely; the strengths of the language and a variety of approaches to using it were both detailed.