Final year project

"A functional language and compiler for the Arduino micro-controller"

Author:
Ryan Suchocki

Supervisor:
Dr. Sara Kalvala

Year of Study:
2013-2014

Word Count:
12951 words

Keywords: Compiler, Functional Programming, Micro-controller, Memory management, Arduino

September 13, 2014

µs
microscheme

Abstract

This report describes the design and development of Microscheme: a functional language for the Arduino. The challenges of implementing a high level, functional language on an extremely resource-constrained platform are discussed throughout. Moreover, the particular nuances of the Microscheme compiler are justified, and its achievements presented. Microscheme is unique among compact Scheme implementations in that it uses direct compilation rather than a *virtual machine* approach; and uses an unconventional compiler architecture, in which the tree structure (of the input program) is determined during lexical analysis. The overall contribution is identified as the first micro-controller-targeting functional language to be released publicly, as a ready-to-use tool, with extensive supporting materials.

Acknowledgements

I'd like to make special acknowledgement of the tutorial paper: 'An Incremental Approach to Compiler Construction' [28], by Abdulaziz Ghuloum of Indiana University; which is simply the finest introductory work on Scheme compilation. Also, thanks to Dr. Sara Kalvala, whose guidance was generous and detailed; and finally to fellow student Matthew Hill for his critical inspection of this report.

Contents

1 Introduction **2**
1.1 Aims and Objectives . . . 3
1.2 Organisation . . . 4

2 Preparation **5**
2.1 Feasibility Study . . . 5
2.2 Literature Review . . . 8

3 Methodology **11**
3.1 Iterative Methodology . . . 11
3.2 Prototyping . . . 11
3.3 Development . . . 14
3.4 Testing . . . 15

4 Design **16**
4.1 Compiler Architecture . . . 16
4.2 Type System . . . 18
4.3 Memory Layout . . . 20
4.4 Register Allocation . . . 22
4.5 Calling Convention . . . 24
4.6 Tail Recursion . . . 28
4.7 Exception Handling . . . 30
4.8 Syntactic Sugar . . . 30

5 Results **32**
5.1 Example Programs . . . 32
5.2 Comparative Analysis . . . 37

6 Release **38**

7 Conclusions & Further Work **41**

References **45**

Appendices **48**

1 Introduction

The Arduino is an open-source development board, based on Atmel 8-bit micro-controllers, with associated software, libraries and accessories. The user-base consists mainly of "artists, designers, hobbyists and anyone interested in creating interactive objects or environments" [1]. Generally, the intended users are non-experts, from backgrounds other than computing and electrical engineering.

Arduino boards consist of an ATMega micro-controller, mounted on a credit-card-sized PCB (printed circuit board). The micro-controller chip is augmented with voltage regulators, power and USB sockets, general-purpose I/O pins, and LED lights to form a user-friendly prototyping platform. The layout of the board is standardised so that accessory boards (known as *shields*, made by third-parties, can be stacked directly onto the Arduino. Typically, ATMega chips are programmed using an external programmer device, but Arduinos are pre-loaded with a 'bootloader' which facilitates easy reprogramming via the USB port. The official Arduino software is an all-in-one IDE. It consists of a code editor, with an easy interface to the bundled debugger, compiler and uploader tools. The debugging and compiling features are provided by *avr-gcc* [2], and uploading by *avrdude* [3]. The only programming language available is C. The *avr-libc* [4] and *wiring* [5] libraries are built-in, providing a large wealth of library functions and definitions. Together with an impressive collection of example programs, these ensure that a programmer familiar with C can very quickly begin working with the Arduino.

The functional programming paradigm is characterised by the idea that all computation can be achieved through the evaluation of mathematical functions. In the functional style, the programmer is required to write definitions for these functions, and the computer is required to evaluate them. Based on Alonso Church's λ-calculus, functional languages provide a radically different way of thinking about computation, compared with the more prevalent imperative family. There are many reasons one might like to program in the functional style, ranging from a simple predisposition for mathematical reasoning, to an appreciation of the elegant and concise code that results. Some of its most compelling features include higher-order functions, by which new functions following familiar control patterns can be created, just by combining existing functions; and the abstraction (automation) of common data structures such as lists, trees and vectors.

1.1 Aims and Objectives

The C programming language is (for good reason) the *de facto* standard for all micro-controller programming. It is appropriate that the pre-eminent language should be C, because reasoning about C programs requires precisely the sort of low-level understanding of computer architecture that is likely to be held by users of micro-controllers. This should not, however, preclude the availability of alternatives; and one underlying motivation for this project is **to provide an alternative language for the popular Arduino project**. Since the Arduino is targeted at people without a background in electrical engineering, it is not reasonable to assume that they have any preference for C, or for imperative programming in general. A functional language will enable new users with some experience with mathematics to explore the platform more productively, as well as presenting a valuable learning opportunity for existing users.

As well as providing an alternative for the Arduino in particular; the overall academic aim of this project is **to show that functional programming for micro-controllers in general is not only feasible, but practical and even preferable**; which has not satisfactorily been shown in the literature. There are major challenges involved in implementing a functional language for a platform with only 8KB of RAM. Those same challenges are manifest as fine-grain resource savings in larger implementations, which presume gigabytes of RAM. The same extrapolation can be made for time-efficiency. Pushing implementations to the extreme low end of platform capabilities is therefore a necessary and fruitful mode of research for the field of programming language and compiler design.

In order to achieve the aforementioned aims, the following project objectives were codified early on, and formed the basis of the project timetable:

1. To design a functional language, based on Scheme, along with a runtime system which can be implemented on an Arduino device.
2. To implement a compiler that translates the above language into the relevant assembly code.
3. To implement, within the compiler, some degree of program optimization.
4. To make the compiler 'production ready' by providing error reporting, some documentation, and an easy interface with the existing GNU compiler stack so that a user can easily compile, assemble and upload code.

5. To produce working example programs, and by comparing their performance with equivalent programs written in C, conduct an analysis of the performance of the new language.

1.2 Organisation

The project was carried out in four major phases: preparatory research, prototyping, development and deployment. The preparatory research took place over the Summer of 2013, after initial discussions with the project supervisor. The findings of the preparatory research are presented here as a feasibility study and literature review. The feasibility study mainly documents the refinement of the project aim from *a functional language* to *a Scheme subset*, while the literature review gives a summary of the existing relevant work, and the projects that were influential in the design of Microscheme. The prototyping and development phases were carried out from October 2013 to January 2014, and are presented here in the Methodology and Design sections. From February to March 2014, the focus changed to providing supporting materials so that the compiler could be released to the public. These materials included a public website, comprehensive language guide, example programs and utility libraries. Since then, the website has been visited over 300 times, mostly through links on the Arduino and Scheme community forums. This work is detailed in the Results and Release sections.

The majority of the project work focussed on objectives 1, 2, and 4. Though aspects of program optimisation can be seen in the solution, no specific period was dedicated to it. It was decided later that working towards a public release of the non-optimising compiler was a more worthwhile aim; and a significant amount of time was needed to produce the supporting materials. Also, a comparative analysis—as suggested by Objective 5—is included in this report (See section 5.2). This performance analysis was not a major part of the project, because functional languages are generally 'slower' than low-level imperative ones, and this is largely accepted as the price for richer semantics and more advanced features. The results are, however, interesting. The analysis will show that Microscheme can compete with C in terms of execution time. However, the final usability and usefulness of the compiler was given more consideration than its relative performance. This project is about the applicability of functional languages to micro-controllers, and it seeks to demonstrate that claim through the variety of *working* example programs and the conceptual merits of those programs, as well as the measurable performance results.

2 Preparation

2.1 Feasibility Study

The applicability of the functional paradigm to low-powered micro-controller applications is not self-evident. The functional paradigm provides a perspective of computer programs as collections of (perhaps recursive) functions which need to be evaluated in some predictable order. It assigns to the computer the role of "evaluator or calculator: its job is to evaluate expressions and print the results" [6]. Seemingly, this is at odds with the conventional role of micro-controllers, which is explicitly to sequence input and output events, with some calculation *on the side*. One does not typically sit at a micro-controller terminal and use it interactively. Therefore, it is not at all clear that a functional language can express any useful program for a micro-controller. A significant amount of preparation time for this project was needed to determine what kind of functional language would be useful—and which of its features could be implemented—on the Arduino platform.

An important question is whether to aspire to implement a purely functional language. Pure functional programming entails *evaluation without side-effects*, meaning that any given expression should result in the same value each time it is evaluated; irrespective of the program state. "This relieves the programmer of the burden of prescribing the flow of control" [7]. However, the micro-controller programmer requires a great deal of control over the sequence of input/output events, which is inseparably determined by the flow of control. Furthermore, the programmer requires that a program has many *effects* other than to compute its result (usually real-world effects such as operating lights and motors). This also suggests that strict (eager) evaluation is more appropriate than lazy evaluation.

A consequence of functional purity is that expressions as simple as a single *variable reference* are required always to evaluate to the same value. Hence, there can be no assignment, and variables cannot be 'reused'. In an extremely memory-constrained environment, this is not workable (without implementing an exceptionally efficient garbage collector). Arguably, the key advantage of functional over imperative programming is the support (and indeed, encouragement) for higher order functions, which can be provided regardless of functional purity. Using higher order functions, "one can modularise programs in new and exciting ways" [7]. (Which amounts to incredibly concise source code.) Hence, it was decided to implement a non-pure language.

The LISP programming language [8] (in all its dialects), which is non-pure, interprets

the functional paradigm in a way conducive to micro-controller applications. It solicits the view of programs, and their constituent procedures, as lists of expressions to be evaluated in order. It allows input and output operations in a straightforward imperative way, without resorting to the bewildering 'Monad' system [9] espoused by pure functional languages. The 'Scheme' dialect [10] carries a high degree of minimalism, but is abundantly expressive for the type of programs one might reasonably want to run on low-powered devices. That minimalism also makes it suitable for a project on this time-scale. Finally, there is an active Scheme community, comprising researchers and enthusiasts who share the *hacking* and experimental values of the Arduino community. So, the implementation of Scheme for the Arduino is a particularly promising combination.

The essential feasibility of functional programming on the Arduino platform rests on the foundations for first-class functions. The ATMega micro-controllers—used on Arduino boards—are classified as modified Harvard architecture. In fact, they are tangibly Harvard architecture machines (with separate program and data memories, occupying separate address spaces), modified only in the sense that the minimal LMP (Load Program Memory) and SPM (Store Program Memory) instructions are provided. SPM is used by the Arduino bootloader for easy on-chip programming (i.e. uploading new programs). The equivalent 'data memory' instructions are faster (taking 1 clock cycle per instruction rather than 3) and are provided with useful variations such as displacement, pre-decrement and post-increment [11]. Clearly, the architecture is designed for static code storage and execution. Naively, this conflicts with Scheme's *code is data* philosophy, whereby procedures are created dynamically at runtime. This conflict, combined with the tiny data memory (2-8 kB) renders interpreted or just-in-time-compiled scheme impractical. For statically compiled Scheme, the distinction is irrelevant. Procedures which may be created at any point during the program's execution are compiled statically, and brought into memory by creating a heap-allocated *closure* object, which contains a reference to the statically compiled code, as well as variables *closed over* by the procedure. Analogously to object-oriented classes, a `(lambda ...)` form might be evaluated several times: resulting in several procedures sharing the same code, but with distinct *environments* (distinct heap-allocated closure objects). The IJMP instruction [11] allows program memory addresses, stored in data memory, to be sought. Therefore, first-class functions can be achieved within the AVR instruction set.

One of Scheme's characteristic features is first-class continuations [12]. 'Continuation' refers to the state of a program at a particular step of its execution. Conventionally, when a subroutine (c.f. *procedure*, *function*) is called, a temporary context switch occurs; and

the program will *continue* executing from where the subroutine was called when it is complete. With first-class continuations, the programmer is free to dictate arbitrarily *to which context* control will pass when a subroutine relinquishes control. This *continuation passing style* (CPS) is seldom used by non-expert programmers, because it demands a departure from the customary nested quality of subroutines. The usefulness of first-class continuations rather derives from expediences in multi-tasking. First-class continuations pave the way for elegant implementations of coroutines [13], lightweight threads [14], and concurrency libraries [15]. Some implementations, however, have omitted them [16].

Imperative languages are typically implemented using a call-stack [17], where continuations are embodied as activation records stored in a global *last-in first-out* data structure. First-class continuations eschew not only the *last-in first-out* principle, but also the idea that a subroutine can 'return' only once. In fact, first-class continuations cannot efficiently be implemented using a contiguous call-stack, as "the entire stack must be copied on each creation or invocation of a first-class continuation" [18].

Like most RISC architectures, Atmel AVR is clearly optimised [19, 20] for the provision of a global execution stack. Dedicated 'stack pointer' registers are provided, and special instructions for working with a global stack are relatively fast [11]. A first-class continuation implementation uses a heap-allocated continuation chain to store activation records. When a program is executing in a regular, non-CPS way, this continuation chain is traversed in a precisely *last-in first-out* manner. Therefore, in programs which do not make use of first-class continuations, the continuation-chain is effectively a very inefficient simulation of a call stack. The burden of garbage collection is increased, and the architectural bias for a call stack is wasted. Appel [21] shows that heap-allocation and garbage collection can match the efficiency of a stack, but only with the luxury of large physical memories. In this exceptionally resource-restrained environment, and in light of the fact that typical micro-controller applications do not make use of concurrency, it is acceptable to omit first-class continuations.

This study suggests that an implementation of a Scheme subset for the Arduino will be both practicable and useful. Scheme has neat functional features such as higher-order functions, but retains the sort of explicit sequencing control, and ease of input/output, upon which micro-controller programming depends. Scheme is also high-level enough (in the sense of automatic memory management) to provide an overall usability shift from C. The implemented language will be recognisably Scheme in syntax and semantics, with the omission of first-class continuations, and will be called Microscheme.

2.2 Literature Review

Scheme is deliberately suited for rapid implementation, and so there several relevant lightweight scheme implementations in existence. At the time of writing, the Scheme community website [22] lists over 70 public implementations, some of which are micro-controller-targeting. Most of these projects, however, are not well documented. While some insight can be gained by reading their respective source code, the most significant projects are the ones written up as papers for publication, or presented as tutorials. This review discusses the well-explained projects that were influential in the design of Microscheme, and are ultimately a relevant basis for comparison.

PICOBIT [16] is a relevant micro-controller-targeting Scheme implementation which is academically documented. It consists of a front-end compiler, and a virtual machine designed to run on micro-controllers comparable to the Arduino (less than 10 kB of RAM). This arrangement is interesting, because the implementation is portable to any micro-controller platform for which the virtual machine can be compiled. PICOBIT deliberately targets a *subset* of the Scheme standard, on the basis of “usefulness in an embedded context”. First-class continuations, threads and unbound precision integers are considered useful, while floating-point numbers and a distinct vector type[1] are left out. The usefulness of unbounded precision integers (a costly feature) is questionable, because the computations required of small micro-controllers rarely involve values beyond a predictable range. Correspondingly, the lack of a floating-point type is unsatisfactory, because micro-controllers commonly process sensor readings, which usually measure rational values. Correct processing of sensor readings is rarely attainable with integer arithmetic. Efficient vectors are also desirable in any memory-constrained scenario. While the aims of PICOBIT are closely aligned to this project, Microscheme will occupy quite a different Scheme subset.

BIT [23, 24] is another interesting micro-controller-targeting Scheme implementation. Like PICOBIT, it consists of a custom virtual machine and corresponding bytecode compiler. PIC’s aim was to demonstrate the feasibility of Scheme on the Motorola 68HC11 chip. Though the project was successful to the extent of off-chip testing, the implementation was not actually deployed on the 68HC11 device. This failure is attributed to the fact that “gcc expects many registers on the target machine”, while the Motorola chip has only one general purpose register. This exposes a fundamental problem with writing a virtual machine for a micro-controller: that one is reliant on some external compiler to deploy the virtual machine. PICOBIT was successful in this respect, because the

[1] Efficient vectors are contiguous arrays, rather than linked lists.

authors developed an optimising C compiler specifically for the task of compiling their virtual machine. PICBIT [25] is a derivative of BIT, specifically targeting the PIC micro-controller. PICBIT—which used the 'Hi-Tech PICC-18 C compiler' for virtual machine compilation—was experimentally successful. In summary, these projects show that the success of a virtual machine implementation is highly dependent on the quality of the virtual machine compilation, to the extent that an implementation can be unusable on the target micro-controller.

Unlike PICOBIT and BIT, which are research projects, ARMPIT Scheme [26] (targeting ARM micro-controllers) is a well-documented, open-source software project, with a large number of real-world working examples. Contrary to the argument given in the feasibility study, ARMPIT's designers *do* intend that the micro-controller is used interactively, with a user issuing expressions and awaiting results via a serial connection. Therefore, ARMPIT is a Scheme interpreter rather than a compiler. The technical aspects of ARMPIT are extensively documented online, and it appears to implement an impressively complete subset of the R5RS [27] standard, with good performance. Some of the ARM devices supported are comparable to the Arduino in terms of RAM size, and so it can be concluded that it is possible to implement Scheme fully under such memory constraints. On the other hand, the task of interpretation is inherently less complex than compilation, so this is not a fair means of comparison. ARMPIT is the most serious Scheme implementation for micro-controllers available, but its usefulness is fundamentally limited by the fact that it is an interpreter.

The tutorial paper "An Incremental Approach to Compiler Construction" [28] by Abdulaziz Ghuloum is an excellent instructional work on Scheme compilation, which influenced the design of the Microscheme compiler in several ways. This paper does not explicitly describe a micro-controller implementation, but rather a methodology, and a generally lightweight implementation which is applicable to many target platforms. The methodology set out in this paper—which consists of a series of working compilers, each implementing an increasingly expressive (and increasingly sugared) subset of the language—was adopted as the chief prototyping technique for this project. One interesting aspect of this tutorial is that it places no importance on first-class continuations, and hence implements a call-stack rather than a heap-allocated continuation chain, yet arrives at a very rich Scheme subset. It is also novel in that it describes direct compilation from Scheme to a low-level assembly language, rather than compilation for a virtual machine. The data type tagging arrangement, activation frame layout, calling convention, error checking strategy, closure representation and tail-call strategy suggested in [28] formed

the basis (though heavily modified) of the Microscheme runtime system. The online tutorial series 'Scheme from Scratch' [29] shows how the methodology presented in [28] can be used to write compilers targeting C (a relatively high level language) as well as assembly. Relevance to this particular project aside, these two documents provide an insight into *how lisp-like languages are compiled* which is unsurpassed in both accessibility and pace.

Although functional programming on micro-controllers is basically a reasonable aspiration, there is clearly a lack of such implementations, especially for the Scheme language. Also, there appears to be no effort whatsoever to bring the functional paradigm to the Arduino, or even to Atmel chips in general. This project, therefore, offers a truly original tool, and fills a distinct gap in the wealth of public Scheme implementations. While there is a core of literature regarding the compilation of modern LISP-like languages for micro-controllers; it is heavily biassed towards virtual machines and bytecode compilation. While that approach enables portability between different micro-controllers, portability can also be achieved by providing a compiler with multiple *back-ends*; and none of those projects have been released as working tools. By exploring the possibility that straightforward source-to-assembly compilation is more viable (and more successful in the real world) for these constrained environments, this project provides a novel contribution to the research literature.

3 Methodology

3.1 Iterative Methodology

The overall development methodology was iterative, consisting of a series of prototype compilers (written in Haskell) followed by a production compiler written in pure C. This approach combines the advantages of rapid prototyping in a mature, very high-level functional language, with the eventual efficiency, portability and ease of installation of a pure C binary. Although many works on Scheme interpretation [30, 31, 28] advocate the use of Scheme itself for the task (i.e. bootstrapping), Haskell has also been favoured [32]—and was chosen for prototyping in this project due to the Author's familiarity with it.

The iterative methodology is based on the suggestions of [28], which proposes a succession of working compilers, each translating an increasingly rich subset of the goal language. This exploits the hierarchical characteristic of Scheme, whereby a small number of *fundamental forms* describe the syntax of every valid Scheme program; and an implementation-specific collection of primitive procedures are provided for convenience. Thus, exploiting "how small Scheme is when the core structure is considered independently from its syntactic extensions and primitives." [31]. These primitive procedures, which add input/output capabilities and efficient implementations for low-level tasks, can all be compiled as special cases of the 'procedure call' form. (The nature of the *core language*, as implemented by Microscheme, is presented as a formal grammar in appendix I.) This methodology also simplifies the building of a type system, because most of the prototyping can be done with the *integer* and *procedure* types, while richer types such as *characters*, *strings*, *lists* and *vectors* are added on later.

3.2 Prototyping

Compiler prototyping in functional languages is rapid because programming language grammars can be expressed directly as functional expressions. Recursive-descent parsing is achieved naturally by the evaluation of these expressions. In the case of Haskell, the built-in pattern matching and tree-like data types are powerful enough that the programmer has to write very little code to arrive at a working parser. Furthermore, sophisticated built-in string manipulation features render lexical analysis and code generation trivial. Appendix A shows an early prototype iteration, where programs are 'compiled' to a sequence of natural-language instructions. The structure of the Abstract Syntax Tree (AST) is concisely defined by the following Haskell data type declaration:

```
1 data Form = Const Num' | Var_ref Var | Quote Form | Seq [Form] |
      ProcCall Form [Form] | PrimCall String [Form] | Asg Var Form | Def
      Var Form | Cond Form Form Form | ProcDef [Var] Form deriving (Show,
      Eq)
```

The parser function 'l_parse' is defined by seven separate cases, dealing with the seven fundamental forms of the Scheme syntax. The most complicated case is that of 'lambda', which occupies just 2 lines:

```
1 l_parse (ESub (Keyword k : ESub vars : body))
2     | k == 'lambda' = ProcDef (map l_parseId vars) (seq (map l_parse
          body))
```

Here, 'ESub' is a structure produced by the lexer, representing a matched pair of parentheses, containing a list of tokens. (For a more detailed explanation of this early parenthesis-matching, see section 4.1) The lambda form is applicable to sequences of tokens of the form `(lambda (tokens...)tokens...)`. The structural constraint is enforced entirely by Haskell's built-in pattern matching facility, while the keyword *lambda* is detected by a straightforward equality condition. Thus, a huge amount of type-checking and list processing is implicitly represented by this code. Finally, the constructor function 'ProcDef' is used to produce a suitable node in the AST.

```
1 emit_instr (Cond test conseq alt) =
2     'Evaluate: {' ++ (emit_instr test) ++ '}. Compare result. If true,
          do this: {' ++ (emit_instr conseq) ++ '}, otherwise do this: {'
          ++ (emit_instr alt) ++ '}.'
```

Just as the AST is easily constructed in Haskell, the code for its traversal is also very elegant. The above rule evaluates a 'Conditional Branch' (aka IF) node in the AST, and generates a natural-language description of *how to evaluate it.* This definition makes recursive calls for the 'test', 'consequent' and 'alternative' subexpressions, thus incorporating the appropriate instructions into the string. Together, these rules describe an in-order tree traversal algorithm.

The function generating natural language instructions is sufficient for verifying the lexer and parser functions, and provides an outline for the code generator. The natural

language instructions generated by the function 'emit_instr' describe a *stack machine* for evaluating Scheme expressions (based on intuitions given by [28] and [30]) where operands are stored on the evaluation stack, and evaluation results are stored in a special 'current result' register. Scheme's universal adoption of the Polish (prefix) notation means this evaluation strategy is suitable for all expressions.

In order to produce a functioning code generator, those natural language instructions must be substituted with real Atmel AVR assembly code. This is a complicated process, where the intricacies of the calling convention, type system and automatic memory management are woven into the code generator. At this stage, the prototype only deals with the numeric type, and the simplest case of the code generator function is a *constant* form. The instruction needed is "Move constant ... into the result register". Inspection of the Instruction Set Manual reveals the 'LDI' (Load Immediate) instruction does just that. The instruction is used twice, to achieve a 16-bit numeric type. The Current Result Register occupies two 8-bit registers: CRSh and CRSl. The corresponding cases of the natural language generator, and the working code generator, are relatively congruent:

```
1 emit_instr (Const n) = "Move constant '" ++ (show n) ++ "' into result
    register.\n"
2 emit_asm env unique (Const n) = "LDI CRSh, " ++ imrepih n ++ "\nLDI
    CRSl, " ++ imrepil n ++ "\n"
```

However, more complicated is the case of a procedure call expression. The natural language instructions are roughly 'evaluate the arguments then call the procedure'. In reality, a Scheme implementation is expected to check that the given procedure name is bound to a procedure at the time of calling; to check that the number of arguments given matches the number expected; as well as to hand over control gracefully according to the chosen *calling convention*. This amounts to around thirty assembly instructions for each procedure call. (For details of the actual instructions generated for a procedure call, see section 4.5) In summary, the process of writing a working code generator is aided by the iterative methodology; the task is reduced to writing a series of program fragments, each performing a low-level action described by the natural language template.

3.3 Development

The prototype in Haskell was extremely fast to write, and could be modified with very little work. This was ideal for experimenting with intricacies such as the type system, exception handling and memory management. These experiments led to an understanding of which Scheme features could be achieved on the platform, and how they could be compiled; but the prototype is unsatisfactory as a final product in many ways. Firstly, it is unreasonable to expect a user to install a Haskell runtime in order to compile their Scheme programs for the Arduino. Also, certain code generation tasks such as generating globally unique label names are impractical within Haskell, because of its rejection of *program state*. Furthermore, desirable features such as line-number tracking, compilation error reporting and syntactic sugar (i.e. comment lines and string notation) would corrupt the elegance of the fully recursive prototype. Finally, Haskell's general-purpose pattern matching and rich data types are easily out-performed by a hand-coded C program with tightly nested comparisons, closed iterative loops and strictly controlled static arrays. With these aims in mind, the prototype was manually translated into C. The prototype consisted of lexer, parser and code generator functions. This structure is mirrored in the C version, but the three functions are split into separate source files (lexer.c, parser.c and codegen.c), because the equivalent C code is considerably larger. This translation exercise formed the main development phase of the project.

Translated into C, the prototype formed a solid framework for the final compiler, but further development was necessary to complete the supported language. The only structural change to the program was the addition of a 'scoper' stage. The prototype provided crude local-only variable bindings by translating local variable names into parameter indices. For example, the expression `(lambda (x y z) (+ x y))` would be translated into `(lambda [3 arguments] (+ [1st argument] [2nd argument]))` during parsing. However, a more elaborate system is needed to provide global variables, closures and lexical scope. Therefore, a 'scoper' function was added, which performs transformations on the AST between parsing and code generation. The more sophisticated scoper classifies variable references as *local bindings*, *global bindings* or *closure references* according to the rules of lexical scope. In addition, it produces a list of variables which must be *closed over* when a procedure is created.

In order to support a useful language, extra data types and a built-in library of useful primitive procedures were added—simply by adding extra 'cases' within the code generator function. Convenience features such as compilation error reporting were added without difficulty.

3.4 Testing

The testing strategy employed was 'continuous integration testing'; where, at each iteration of the software, the functionality added by the new code is verified. This testing strategy is particularly productive when used with the distinctly iterative methodology described here. The initial prototype is very minimal, and it is very easy to reason about the computation, and manually debug. At each iteration, it is very clear what new class of programs should be supported, and verify the new functionality. Therefore, the correctness of the compiler is maintained throughout, but for an increasingly expressive subset language. In addition, the final iteration was checked for memory leaks using the common Valgrind tool, and input boundary conditions were manually tested.

A common testing strategy for compilers is to first produce an exhaustive set of test programs, and then develop the compiler with the aim of correctly supporting those programs. However, since the goal of this project is to determine what kind of functional language can be implemented on such a minimal platform, that strategy was not suitable. A process of formal verification was also unsuitable for the same reason. The overall 'proof' of the software's success is the variety of recognisable Scheme programs that are correctly compiled by it; which are described at length in the results section.

*

* *

In conclusion, the key interesting aspects of this methodology are the iterative process suggested by [28], combined with the idea of compiling to natural language instructions first, before implementing a working code generator. This has the effect of separating out the tasks of 'parsing a source language' and 'generating in a target language'. While it might not lead to the most efficient compilers, this separation is clearly an example of modular software development, which is an intrinsically good practice. Those concepts, allied with the notion of prototyping in a mature functional language, guaranteed a highly productive workflow, which resulted in a useful production-ready compiler.

4 Design

This section sets out the design of the Microscheme compiler and runtime system. The content here is an accurate description of the final working compiler, written in C, and submitted with this report. The decisions presented here are the result of the extensive prototyping phase, and represent an exceptionally compact partial Scheme implementation. This design is a compromise between completeness (in the sense of compliance with the published Scheme standard [27]) and feasibility on the Arduino platform. The design features described cover all noteworthy aspects of the solution, and are each justified in terms of those competing constraints.

4.1 Compiler Architecture

The compiler is a recursive-descent, 4-pass cross-compiler, hand-written in pure C. It will run on any platform for which there is a C (99) implementation; and is designed to run on any modern PC. The recursive descent parsing strategy was selected due to its unmatched programmatic elegance, and particular suitability for the Scheme syntax.

The three components that are found in virtually all compilers are the lexer, parser and code generator. The lexer performs *lexical analysis*, typically transforming the source file into a list (or stream) of tokens. Tokens are the smallest indivisible units of a program's syntax. The parser performs *syntactic analysis*, by applying the rules of the language's *grammar*, and producing an Abstract Syntax Tree. Commonly, these stages are implemented in a *pipeline*, such that the parser controls the overall process, and calls the lexer function whenever it is ready for a new token. However, it is also possible to implement the lexer and parser as distinct processes, where the entire program is *lexed* before it is parsed; which aids reasoning about the compiler as a succession of transformations on a program representation. This is how the Microscheme compiler is organised.

When the lexer and parser are implemented as distinct stages, it is possible to adjust the division of labour between the two processes. Traditionally, the lexer converts the source code (a linear structure) into a **list** of tokens (another linear structure). The parser is responsible for transforming this linear structure into a *tree structure*, as well as recognising expression types. (See figure 1.) In the case of hand-written compilers, this division of labour typically leads to incredibly simple lexer functions, and incredibly complex parser functions.

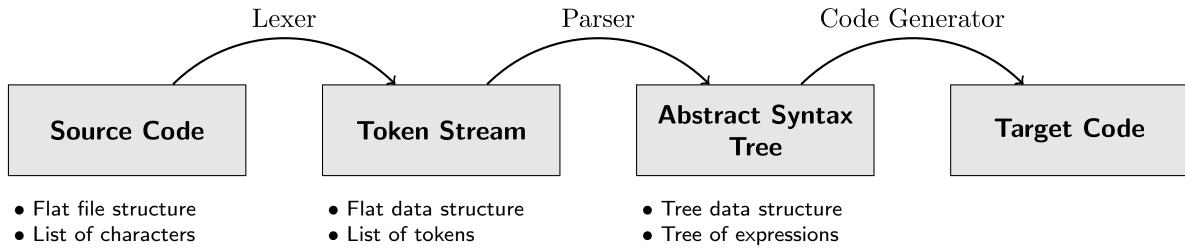


Figure 1: Conventional compiler architecture

This project explores a novel adaptation on the conventional architecture, made possible by the elegant syntax of LISP (and its dialects). Uniquely, the structure of the parse tree of a LISP program can be determined from the parentheses alone, right down to the level of arithmetic sub-expressions; without considering any of the actual keywords. Combined with the fact that Polish notation (prefix order) is enforced throughout, this means that the language has an LL(1) grammar. Looking at it another way: LISP's fully parenthesized syntax effectively makes *program nesting* a matter of lexical analysis, and reduces the burden on the syntactic analyser.

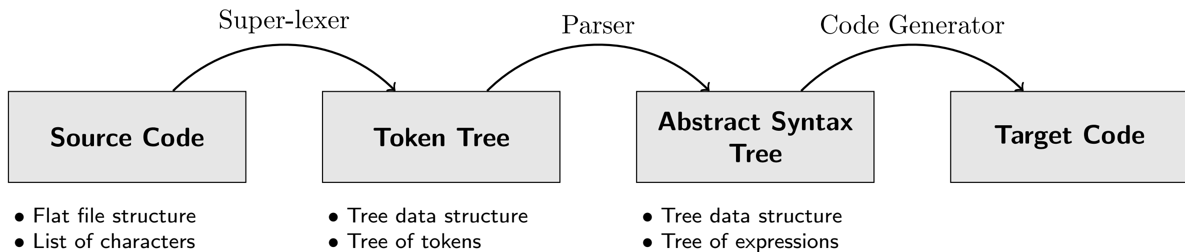


Figure 2: Revised compiler architecture

Figure 2 shows the amended compiler architecture, in which the tree structure of the AST is prepared as early as the lexing stage. (The lexer is referred to as a super-lexer to indicate its increased responsibilities.) Rather than producing a list of tokens, the super-lexer produces a Token Tree, made up of Token Nodes. A Token Node is either a keyword, an identifier, a numeric constant, a string constant or a subtree. A subtree node is itself a Token Tree, and represents a sub-expression in the program. The job of the parser is dramatically simplified because it has only to classify each node via relatively straightforward pattern matching, within a prepared tree structure. The parser never encounters any '(' or ')' tokens, as parentheses are matched and expanded by the super-lexer.

Finally—as shown by figure 3—a scoper function is included. Unlike the lexer and parser, the scoper does not generate a new data structure, but simply does work on the AST. It transforms variable references into explicit parameter, global or closure references, according to the rules of lexical scope.

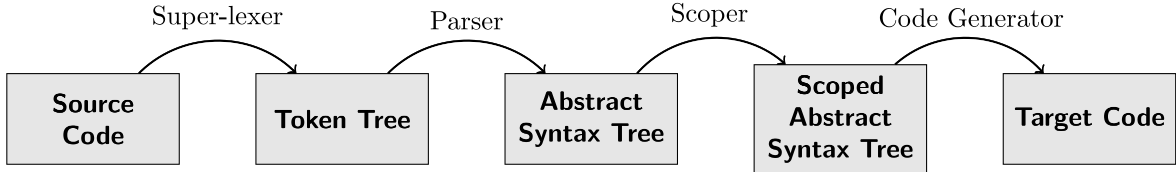


Figure 3: Full compiler architecture

4.2 Type System

Scheme has a strong, dynamic type system. It is strong in the sense that no type coercion occurs, any value stored in memory has a definite type, and procedures are only valid for a specific set of types. It is dynamic in the sense that variable names are not prescribed types by the programmer, and a given identifier can be bound to values of any type during runtime. Therefore, it is necessary for values to 'carry around' type information at runtime. The built-in types supported by Microscheme are procedures (functions), integers, characters, Booleans, vectors, pairs and *the empty list* (aka Null), which is considered to be a unique type. Linked lists are built recursively using pairs and *the empty list*. Strings are supported by the compiler, and are compiled as vectors of characters. Though this range of built-in types is minimal, it is powerful enough that richer types may be implemented 'on top'. For example, 'long integer' and 'fixed-point real' libraries have been developed for Microscheme, which use pairs of integers to represent numbers with higher precision. Providing a 'numerical stack' by combining simpler types is common for Scheme implementations, and Scheme is designed precisely to enable this kind of extension.

Type	Upper Byte								Lower Byte							
	$b7$	$b6$	$b5$	$b4$	$b3$	$b2$	$b1$	$b0$	$b7$	$b6$	$b5$	$b4$	$b3$	$b2$	$b1$	$b0$
Integer	0	$data \times 15$														
Pair	1	0	0	$address \times 13$												
Vector	1	0	1	$address \times 13$												
Procedure	1	1	0	$address \times 13$												
Character	1	1	1	0	0	-	-	-	$char \times 8$							
Null	1	1	1	0	1	-	-	-	-	-	-	-	-	-	-	-
Boolean	1	1	1	1	1	0	0	b	-	-	-	-	-	-	-	-
Unused	1	1	1	1	0	-	-	-	-	-	-	-	-	-	-	-

Table 1: Data tagging arrangement

Table 1 shows the data tagging scheme used. It was chosen to use fixed memory cells of 16 bits for all global variables, procedure arguments; closure, vector and pair fields. Cells of 16 bits are preferable because they can neatly contain 13-bit addresses (for 8KB

of addressable RAM), as well as 15-bit integers. Although 32-bits is the received standard for integer types; this is a 8-bit computer, and so a compromise was made. The instruction set contains some restricted 16-bit operations such as addition ‘ADDIW’ and subtraction ‘SBIW’, so 16-bit arithmetic is reasonably fast.

The tagging scheme is biassed to give maximum space for the numeric type. The most significant bit of every value held by Microscheme is dedicated to differentiating between ’integers’ and ’any other type’. It is important that the MSB is zero for integer values, rather than one, because this simplifies the evaluation of arithmetic expressions. Numeric values can be added together, subtracted, multiplied or divided without first removing the data tag. A mask must still be applied after the arithmetic, because the calculation could overflow into the MSB and corrupt the data tag. At the other end of the spectrum, Booleans are represented inefficiently under this scheme, with 16 bits of memory used to store a single Boolean value. The richer types are represented fairly efficiently, with 13-bit addressed pointing to larger heap-allocated memory cells. Overall, this system is extremely compact, and provides a very dense representation for the most commonly used data types; as necessitated by the constraints of the Arduino platform. There is some scope for the addition of extra built-in types in the future, as values beginning $11110-$ are currently unused.

Microscheme’s strong typing is achieved by type checking built-in to the primitive procedures. When bit tags are used to represent data types, checking is achieved by applying bit masks to data values, which corresponds directly with assembly instructions such as ‘ANDI’ (bitwise AND, immediate) and ‘ORI’ (bitwise OR, immediate). Therefore, low-level type checking is achieved in very few instructions.

4.3 Memory Layout

The available flash memory (RAM) is allocated the address range `0x200` to `0x21FF` for the Arduino MEGA (and `0x100` to `0x8FF` for the Arduino UNO). It was important to support both models, as they are both popular among the Arduino community. The differences between models are resolved by a model-specific assembly header files, included automatically at compile-time, containing definitions derived from the relevant technical data sheet. Different ATMega chips could easily be supported by writing equivalent definition files. Microscheme uses the first $2 \times n$ bytes for global variable cells, where n is the number of global variables in the program. The remainder of the space is shared between the *heap* and the *stack*. The heap starts immediately after the global variables, and grows 'upwards', while the stack starts at the upper RAM boundary, and grows 'downwards'; an arrangement which is essentially fixed by the nature of the Atmel instruction set.

Objects on the heap are not restricted to the two-byte cell size used elsewhere. The built-in procedures to work with heap-allocated objects determine their size from the information contained within the object. Procedures, pairs, and vectors are heap-allocated types. When a value of these types is held by a variable, the 2-byte variable cell contains the appropriate *data type tag*, followed by a 13-bit memory address, pointing to the start of the area of heap space allocated to that structure. Therefore, there is a built-in layer of indirection with these types. Figure 4, overleaf, shows the layout of the objects in detail.

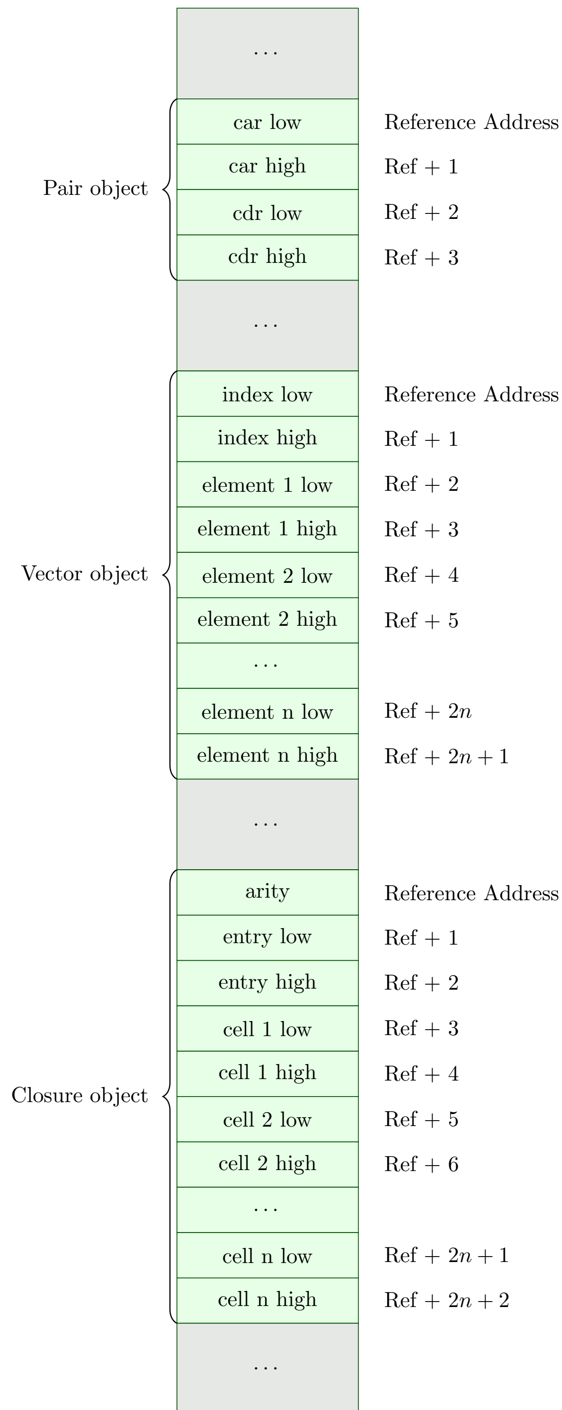


Figure 4: Heap-allocated object layout

For details of the stack layout, see section 4.5.

4.4 Register Allocation

The ATmega series of micro-controllers are purported to have 32 general-purpose registers [20, 19]. In reality, most of these registers are highly restricted in function, and the nuances in this allocation are crucial to the feasibility of Microscheme.

The 32 'general-purpose' registers are restricted in the following ways. The first 16 registers cannot have values loaded directly into them, i.e. the 'LDI' instruction is only valid for registers 16 to 31. This makes them unsuitable for many purposes, because any immediate values that need to be set programatically must first be loaded into an upper-page register, then 'MOV'ed into place. The instructions for performing 16-bit arithmetic such as 'ADDIW' and 'SBIW' are only valid on the final four register pairs: 24-31. Hence, it is highly desirable for any low-level counting or calculation of *offsets* to occur in this range. Furthermore, the instructions for indirect memory addressing are only valid on the final three register pairs: (26:27), (28:29) and (30:31). In other words, memory addresses held in the register pairs (r0:r1) to (r24:r25) cannot be dereferenced. Hence, any 'pointers' to heap or stack locations should ideally be placed in the final three pairs. Also, the 'IJMP' instruction, for loading a code address into the internal Program Counter, is only valid for the final register pair (r30:r31). This instruction is crucial in the provision of first-class functions; the essential quality of functional programming. Finally, the instructions 'LDD' and 'STD', for indirect memory access with displacement, are available for the final two register pairs: (28:29) and (30:31). In summary, these restrictions pose a significant challenge in the provision of a functional language runtime. This is primarily because of the amount of book-keeping required at runtime to provide the dynamic type system and automatic memory management; which in-turn rely on registers being reserved for specific low-level purposes.

Register	Use	Register	Use	Register	Use	Register	Use
r0 r1	MULX	r8 r9	- -	r16 r17	GP1	r24 r25	CCP
r2 r3	TCSS	r10 r11	- -	r18 r19	GP2	r26 r27	HFP
r4 r5	falseReg zeroReg	r12 r13	- -	r20 r21	GP3	r28 r29	CRS
r6 r7	- -	r14 r15	- -	r22 r23	PCR -	r30 r31	AFP

Table 2: Register Allocation Table

Table 2 shows the allocation of registers by Microscheme. The Microscheme runtime system requires 4 registers to be reserved for special purposes. The 'CCP' (Current Closure Pointer) stores a reference to the 'closure' or 'procedure object' of the currently executing procedure, if any. The 'HFP' (Heap Free Pointer) stores the address of the next available byte of heap-storage; where any new heap object should be allocated. The 'CRS' (Current ReSult) stores the result of the most recently evaluated expression, or sub-expression. Finally, the 'AFP' (Activation Frame Pointer) points to the first byte of the current 'Activation Frame' on the call stack. This is where procedure arguments are found. These four values require 16 bits each, and are placed in the register pairs (24:25) to (30:31) so that 16-bit arithmetic operations may be used, as discussed above.

The first major challenge with these allocations is that each of the CCP, HFP, CRS and AFP will—at some point—hold memory addresses which will need to be dereferenced. However, as described above, the instruction for indirect memory access is only valid on the final three register pairs. The chosen solution is to place the CCP in register pair (24:25). When the CCP is dereferenced, Microscheme swaps it into the pair (26:27), performs the necessary memory access, then swaps it back again. This choice is justified by the reasonable estimation that *closure lookup* will always be less frequent than *argument lookup*, *heap writing* or *using the result of the previous calculation*.

The allocation is further restricted by the fact that the IJMP instruction—for branching to a code address stored in memory—is only valid on the register pair (30:31). Ideally, therefore, this pair should be reserved for use when calling a procedure. This would mean relegating the HFP, CRS or AFP to another register pair, as with the CCP, and swapping them in when necessary. This is really not acceptable, because those registers are frequently used in all programs. The chosen solution is to temporarily 'break' the register allocation during a procedure call. When a procedure call is reached, the register pair (30:31) is temporarily overwritten with the target code address, and the *callee* is expected to restore the value. This arrangement works out neatly, because the value of the Activation Frame Pointer changes during a procedure call. Its new value is equal to that of the Stack Pointer, immediately after the context switch. Therefore, the callee procedure can restore the AFP with the instructions `IN AFPl, SPl` and `IN AFPh, SPh`.

The final restriction to the allocation table is that the instructions 'LDD' and 'STD', for indirect memory address with displacement, are only available on the final two register pairs. This instruction is crucial for working efficiently with heap-allocated objects. Figure 4 shows how heap-allocated objects are structured with a single reference address,

followed by data fields which appear at some calculable *displacement* from it. Using the 'LDD' and 'STD' instructions, those fields can be accessed with single instructions. Therefore, the CRS is allocated to register pair (28:29), because it will sometimes store references to heap-allocated objects. By elimination, the HFP must be allocated to register pair (26:27).

The registers 'GP1', 'GP2' and 'GP3' are reserved as general purpose storage cells, that are used internally by the efficient assembly code generated for primitive procedures. Other registers are reserved for minor functions.

Altogether, the register allocation is extremely dense, and deals with a large number of instruction set nuances to minimise the number of instructions generated. These restrictions are a significant limiting factor to the provision of high-level language features. By eschewing first-class continuations, a design has been found that can be implemented efficiently with the available registers; but it is hard to see that a fully-compliant Scheme implementation could ever be achieved on this platform without doubling, or tripling the number of instructions generated.

4.5 Calling Convention

Figures 5, 6 and 7 (overleaf) show typical assembly listings for a procedure call, and the layout of activation frames. Between them, these demonstrate the calling convention for standard (non tail-recursive) procedure calls. The caller-side listing can be summarised as follows:

1. Push the current AFP onto the stack.
2. Push the return address onto the stack.
3. Push the current CCP onto the stack.
4. Evaluate each of the outgoing arguments, and push them onto the stack.
5. Evaluate the procedure expression.
6. Check that the procedure expression evaluated to a procedure.
7. Check that the number of arguments given matches the number expected.
8. Load the procedure entry address from the procedure object.
9. Jump to the procedure entry address.

   ...

10. After the procedure has returned: Restore the AFP from the stack.

The code for a procedure call is rather long, in comparison to a typical C function call. This is because a Scheme procedure call is inherently more flexible. Scheme has a dynamic type system, and allows any expression to take the place of 'procedure name' in the procedure call form. This is a crucial part of the 'functions are first-class values' idea, as it allows for higher-order procedure calls: where the result of a procedure is itself a procedure, which is, in turn, called. However, it is not possible to determine, before runtime, whether that expression will in fact evaluate to a procedure. By the same token, it is not possible to determine beforehand whether the correct number of arguments are given for the procedure. These two conditions must be checked for every Scheme procedure call; costing in the order of 20 clock cycles. The procedure call code has been designed so that a large segment of it (including those two checks) is constant across all procedure calls. That code is defined as a subroutine at the assembly level, and is replaced with a single instruction `JMP proc_call` by the code generator; thus saving hundreds of lines of assembly code in a typical program.

The callee-side listing performs the following steps:

1. Copy the Stack Pointer to the Activation Frame Pointer
2. Execute the body of the procedure
3. Set the stack pointer just below the arguments in the current activation frame.
4. Restore the previous CCP from the activation frame
5. Temporarily place the return address, from the activation frame, into the AFP.
6. Jump to the return address.

The calling convention and activation frame are designed with tail recursion in mind (see section 4.6), but are also influenced by the register restrictions described in the previous section. The CCP and AFP are changed upon a procedure call, and must be restored when that procedure returns. The new CCP is set by the caller, while the AFP must be updated by the callee. The previous values are saved in the activation frame, along with the return address and arguments. The AFP is stored in register pair (30:31), which is also needed for jumping to instruction addresses held in memory. Therefore, it must be restored after by the caller, after the procedure has returned.

```
    PUSH AFPh
    PUSH AFPl
    LDI GP1, hi8(pm(proc_ret_χ))
    PUSH GP1
    LDI GP1, lo8(pm(proc_ret_χ))
    PUSH GP1
    PUSH CCPh
    PUSH CCPl
    [code for argument i]            } Repeat for each argument given
    PUSH CRSl                        }
    PUSH CRSh                        }
    [code for procedure expression]
    LDI PCR, α
    MOV GP1, CRSh                    }
    ANDI GP1, 224                    }
    LDI GP2, 192                     }
    CPSE GP1, GP2                    }
    RJMP error_notproc               }
    ANDI CRSh, 31                    }
    MOV CCPh, CRSh                   } This code is the same for every procedure call, so it is
    MOV CCPl, CRSl                   } outlined (at the assembly level) to a subroutine
    LD GP1, Y;CRS                    }
    CPSE GP1, PCR                    }
    RJMP error_numargs               }
    LDD AFPh, Y+1; Y=CRS             }
    LDD AFPl, Y+2; Y=CRS             }
    IJMP; context switch             }
proc_ret_χ:
    POP AFPl
    POP AFPh
```

Figure 5: Procedure Call Routine (Caller Side)

χ = an identifier unique to this procedure call
$\alpha = 2\times$ arity of this procedure

```
 1  proc_entry_χ:
 2       IN AFPl, SPl
 3       IN AFPh, SPh
 4       [code for procedure body]
 5       ADIW AFPl, α
 6       OUT SPl, AFPl
 7       OUT SPh, AFPh
 8       POP CCPl
 9       POP CCPh
10       POP AFPl
11       POP AFPh
12       IJMP
```

This code is actually omitted if the procedure body always ends with a tail-call, because it will never be reached

Figure 6: Procedure Call Routine (Callee Side)

χ = an identifier unique to this procedure
$\alpha = 2\times$ arity of this procedure

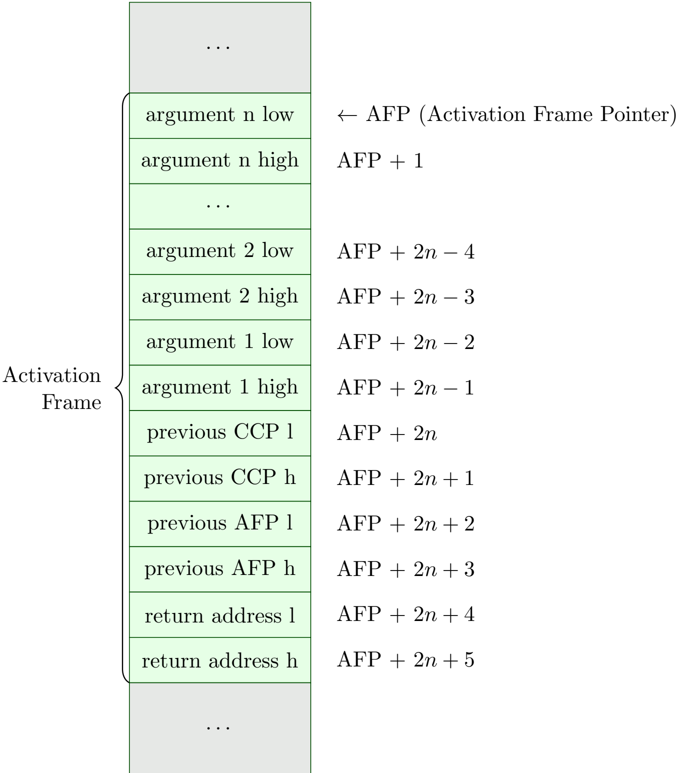


Figure 7: Activation Frame Layout

4.6 Tail Recursion

Scheme implementations are required [27] to be *properly tail recursive.* This means that they must perform tail-call-elimination whenever possible. While some C compilers perform tail-call-elimination as an ad-hoc optimisation, the C standard does not guarantee it. Efficient tail recursion is an important feature for functional language implementations; because it allows recursive algorithms to be used with space-efficiency asymptotically no worse than iteration. Because scheme has no built-in iteration constructs, and because the Arduino has extremely limited memory space; this requirement is especially pertinent for Microscheme.

Tail-call-elimination is performed by the compiler at the parsing stage. Procedure calls are eagerly transformed into tail-calls whenever they are in a *tail context.* An expression is in a tail context if it is the final expression in the body of some procedure. The 'tail context' attribute is inherited by other forms such as conditional branches and by sequences. Hence, in the expression `(lambda (x) (if (zero?  x) (p x) (q x)))`, the procedure calls to 'p' and 'q' are both in the tail context of the lambda expression—and will be compiled as tail calls.

Unlike ordinary procedure calls, tail calls reuse part of the current activation frame; thus ensuring constant-space performance for recursive calls, and releasing memory earlier for non-recursive calls. The activation frame (figure 7) is designed with this operation in mind. The information about the enclosing procedure is left in-tact, while the arguments are overwritten. This causes the callee procedure to 'return' to the enclosing context, instead of the current context. Since this call is in tail-context, nothing will be missed. The following listing gives the assembly code generated for a typical tail-call.

```
 1      [code for argument i]
 2      PUSH CRSl
 3      PUSH CRSh
 4      [code for procedure expression]
 5      MOVW TCSl, CRSl ; Save proc
 6      ADIW AFPl, [2*parent_numArgs]
 7      OUT SPl, AFPl
 8      OUT SPh, AFPh
 9      SBIW AFPl, [α + β]
10      LDD GP1, Z+[β - i]
11      PUSH GP1
12      ...
13      MOVW CRSl, TCSl ; Restore proc
14      LDI PCR, %i
15      MOV GP1, CRSh
16      ANDI GP1, 224
17      LDI GP2, 192
18      CPSE GP1, GP2
19      RJMP error_notproc
20      ANDI CRSh, 31
21      MOV CCPh, CRSh
22      MOV CCPl, CRSl
23      LD GP1, Y;CRS
24      CPSE GP1, PCR
25      RJMP error_numargs
26      LDD AFPh, Y+1; Y=CRS
27      LDD AFPl, Y+2; Y=CRS
28      IJMP; context switch
```

Lines 1–3: Repeat for each incoming argument

Lines 6–12: Shift all the incoming arguments down into the activation frame

Lines 15–28: This code is the same for every procedure call, so it is outlined (at the assembly level) to a subroutine

Figure 8: Tail Call Routine (Caller Side)

χ = an identifier unique to this procedure
$\alpha = 2\times$ arity of outgoing procedure
$\beta = 2\times$ arity of incoming procedure

4.7 Exception Handling

Due to Scheme's dynamic nature, runtime exceptions are unavoidable. As well as the procedure call exceptions described in section 4.5, there are type and range exceptions. If an arithmetic operator is applied to a non-numeric argument, the 'Not a Number' exception will be raised; and likewise for pair and vector operators. The runtime system also checks bounds for vector operations. For example, the expression `(vector-ref 4 "abc")` would result in an 'Out Of Bounds' exception, because the string `"abc"`—which is compiled as a vector of three characters—does not have a fourth element. Finally, there is a 'Divide by Zero' exception for arithmetic, and a 'Custom Exception' which can be raised using the `(error)` primitive, for example when writing libraries with additional internal constraints.

The arduino is a standalone device, with no direct text-based output. There is no guarantee that the user will connect any sort of output device to the Arduino. However, the Arduino standard does guarantee that an LED is connected to digital pin 13 on any compliant board; and so this is the only assured means of communicating with the user. Therefore, digital pin 13 is reserved by Microscheme as a status indicator. The LED is switched off during normal operation; but flashes in a predetermined pattern when an exceptional state is reached. (For a summary of the exception codes, see appendix B "Language Guide"). Conversely, there is no guaranteed means of input whatsoever; so Microscheme does not support any kind of exception recovery. When an exceptional state is reached, the device must be reset. There is no convenient way of reporting the location at which the exception occurred, so it is left to the programmer to determine the program fault by its behaviour up until the exception. Though minimal, this arrangement is the most that can be provided on the platform, and works well in practice.

4.8 Syntactic Sugar

The compiler supports three important syntactic extensions, which greatly increase its general usability. These are strings, comments and 'includes'. Strings are not a distinct type, but are compiled as vectors, where each element of the vector is a character constant. The expression `(define message "Hello!")` is *syntactic sugar* for the less convenient expression `(define message (vector #\H #\e #\l #\l #\o #\!))`. This is a syntactic extension in the sense that the two expressions result in the very same parse tree. The quotation marks are matched by the lexer, and the whole string is passed as a token to the parser. The parser automatically produces a call to the primitive `(vector ...)`

along with the appropriate character literals. Some implementations represent strings as cons-based lists, and some even represent vectors in general as lists. True vectors store elements contiguously; whereas cons-lists store them as linked lists, with individual pointers between each consecutive element. Hence, true vectors use approximately half the space, and were included in Microscheme specifically to enable the efficient storage of strings. The disadvantage with vectors is that they cannot easily be concatenated in-place; but since memory space is at such a premium on the Arduino, the more dense representation is highly preferable.

Any code following a semicolon, before the end of the line, is a considered a comment; and is ignored by the lexer. This is a simple extension, but it is an essential feature for a real-world compiler. Together with the `(include ...)` form, these extensions enable modular programming techniques, code re-use and code sharing. The `(include ...)` form is treated as an instruction to the parser to include an external program as a node in the abstract syntax tree (as is the nature of tree structures). The parser simply calls the 'lexer' and 'parser' functions separately on the included file, then makes then resultant Abstract Syntax Tree a node in the overall tree. These features are intended to facilitate the development of a suite of libraries and examples, and eventually an active userbase.

*

* *

In conclusion, the design decisions presented here represent an extremely compact functional language runtime, and a richly featured, working compiler. The language implemented is a maximal feasible subset of Scheme, with respect to the Arduino platform. It is recognisably a Scheme subset, in the sense that any valid Microscheme program is also a valid Scheme program. It is a 'maximal feasible' subset in the sense that to implement the remaining features would incur a steep performance reduction (due to the small memory and instruction set limitations) with no relevant benefit in terms of expressivity. The register allocation and calling convention are particularly well tailored to the platform, and make extremely good use of the available registers and instructions. The condition of *proper tail recursion* is fulfilled eagerly; which means that tail-recursive functions can be used to perform iteration with constant-space usage, and thus proving the overall viability of recursive algorithms even in low-memory contexts.

5 Results

The overall academic aim of this project is to show that functional programming—in the context of micro-controllers—is not only feasible, but also practical. The feasibility of Scheme in particular as a functional language for the Arduino is demonstrated here firstly by the variety of working example programs. Secondly, a comparative analysis is given to show how Microscheme programs compare with equivalent C programs in terms of execution time, and therefore how practical it is as an alternative. This analysis was performed on programs running on the Arduino hardware, and therefore represents a real-world, easily repeatable outcome.

5.1 Example Programs

The first example program is the canonical 'Hello World'. This is not the simplest possible version of the program; in fact it is contrived in order to demonstrate a range of the compiler's features. Firstly, the 'list' library is imported, which is another Microscheme source file containing common list-processing procedures. The library 'io_uno' is included, which contains digital I/O pin mappings for the 'Arduino UNO', which is the smallest supported model, with only 2KB of RAM. Finally, 'lcd.ms' provides procedures for interacting with a 'Hitachi HD44780' driver, which is extremely commonplace in small LCD modules. The strings "!emehcS" and "morf ,iH" are compiled as vectors, and each passed into the 'vector->list' procedure (so that the procedures defined in 'list.ms' may later be applied to them) and bound to the names *m1* and *m2*. Next, *m2* is appended to *m1*, and the result is reversed to produce the full message. This (unnecessarily complex) code demonstrates many aspects of the language from recursive procedures, in the form of list transformations, to its dynamic type system and syntactic extensions.

```
1  (include 'libraries/list.ms')
2  (include 'libraries/io_uno.ms')
3  (include 'libraries/lcd.ms')
4  (lcd_init)
5
6  (define m1 (vector->list '!emehcS'))
7  (define m2 (vector->list ' morf ,iH'))
8  (define message (reverse (append m1 m2)))
9
10 (for-each write message)
```

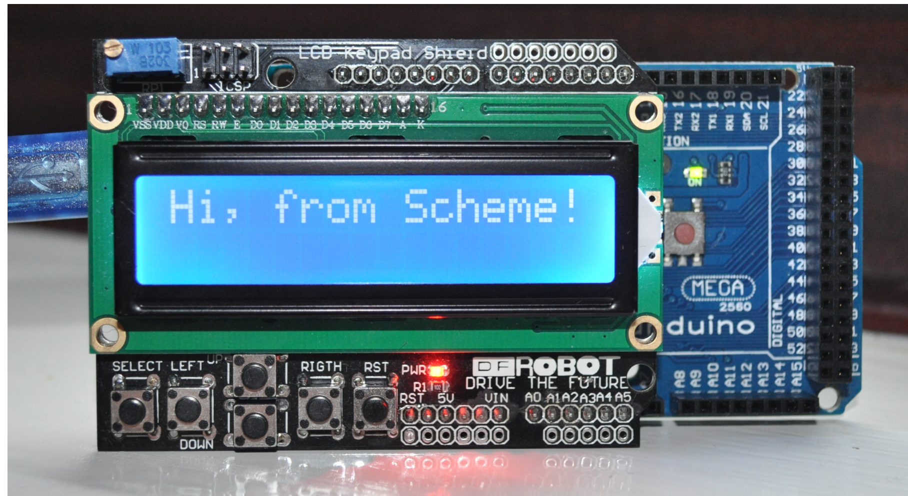


Figure 9: Example Program 1

In order to print the message to the LCD screen, the power of higher-order functions is employed. The 'lcd.ms' library provides a procedure 'write', which prints a single character to the screen, but no procedure for printing an entire string. This is because, in functional programming, it is normal to build functions using existing ones whenever possible. The 'list.ms' library provides a higher-order function 'for-each'. 'For-each' takes a procedure and a list as arguments; and applies the function to each element in the list. Combining these two procedures, a specific procedure for printing lists of characters is composed ad-hoc; without writing out an algorithm for it. In the example, the composed function is used anonymously; but one could bind it to a name (such as `print`) for convenience: `(define (print x) (for-each write x))`.

The second example program (given as appendix C due to its length) shows four procedures calculating factorials. This example is designed to demonstrate the tail-recursive behaviour of the language, as well as the 'long.ms' library, which shows how extended data types can be built on top of the base language.

The first procedure defined is `(factorial n)`, which is a naive recursive definition of the factorial function. It is a naive solution because the recursive call occurs as an operand to an arithmetic expression; and hence cannot be tail-call-eliminated. The recursive call is preceded by the expression `(serial-send (stacksize)`, which does not influence the computation, but serves as a benchmarking tool. The primitive procedure `(stacksize)` simply returns the current size of the call stack (in bytes). By sending this value, via the serial connection, to a serial terminal on a connected PC, the behaviour of the recursion is analysed. Here is the resulting serial transaction when calculating the factorial of 7.

```
8 16 24 32 40 48 56
5040
```

The first seven numbers are the results of `stacksize` at each layer of recursion, and the final number is the result of the calculation. The call-stack grew by 8 bytes at each step, which corresponds with the expected growth for this function. (An activation frame consists of $2n + 6$ bytes, where n is the number of parameters to the function.) While the correct answer was reached, this linear space usage is considered unacceptable for functional languages, which rely on recursion in place of iterative constructs. With this behaviour, if a recursive procedure needed to iterate over hundreds of items, the entire RAM might be occupied by the *call stack* before the base case is reached.

The next procedure, `tail-factorial`, uses a modified 'tail-recursive' algorithm, where the recursive call is the outermost operator in the tail context of the procedure, and so is a candidate for tail-call-elimination, as described in section 4.6. Since the requirement (by the Scheme standard [27]) for *proper* tail-recursion is fully implemented by Microscheme, the output for this procedure shows constant stack usage, while arriving at the same solution:

```
10 10 10 10 10 10 10
5040
```

The factorial function is fairly steep, and Microscheme's built-in integer type cannot store the factorial of 8. Microscheme uses 15-bit unsigned integers, and so the largest integer that can be stored is 32767, while 8! is 40320. However, the Microscheme library 'long.ms' shows how richer numeric types can be built from existing types. This library stores long integers (i.e. 30-bit unsigned integers) as pairs of base integers; i.e. using the built-in 'pair' and 'integer' types. 'long.ms' provides long versions of the basic arithmetic operators, prefixed with 'l' for clarity, such as 'l=', 'l+' and 'l*'. The procedure `lfactorial` uses exactly the same algorithm as `factorial`, but with the long operators in place of primitive ones. By making this simple substitution, the procedure can compute the factorial of 11:

```
14 22 30 38 46 54 62 70 78 86 94
39916800
```

And likewise for the tail-recursive version:

```
16 16 16 16 16 16 16 16 16 16 16
39916800
```

The previous examples demonstrate many interesting features of the compiler, but the programs are simple, and it is not clear how they relate to real-world micro-controller applications. One such application (for which the Arduino is particularly popular) is hobby-level robotics. Controlling physical actuators in real-time is a key micro-controller function, and section 2.1 explained why the LISP family of languages, and Scheme in particular, are well suited to the sequencing of input/output events.

The third and fourth example programs (Appendices D and E respectively) are robotic control programs for tracing fractal patterns. In the interest of speed and repeatability, the programs were tested using a simple robot simulator running on a PC. The control programs themselves were run on real Arduino devices; which sent instructions such as 'forward 10' and 'left 20' to the PC via serial link. The PC was running a simple Python script which interpreted those commands, and moved a simple trace around the screen; simulating a robot dragging a pen across a surface. These programs could trivially be adapted to send similar signals to real electric motors; and the HelloWorld example program has shown that Microscheme can easily communicate with real external hardware.

Fractal patterns were chosen because they are naturally recursive structures. In contrast to the previous example calculating factorials, fractal patterns involve hundreds of recursive procedure calls even for relatively small examples. Therefore, these programs really test the correctness of the runtime system in terms of control flow, and its stability during a long computation.

The first example implements the Koch snowflake. The Koch snowflake is generated by the following recursive procedure: Take a straight line, trisect it, and replace the middle segment with an equilateral triangle; then apply this procedure to each of the 4 resulting straight lines. This procedure corresponds to the sequence of movements: "Forward n, left $60°$, forward n, right $120°$, forward n, left $60°$, forward n", where n is one third of the length of the existing line. When this algorithm is applied to the three sides of an equilateral triangle, a 'snowflake' shape is produced. The infinite recursion is restricted by a simple 'depth' parameter, and a 'base case' which stops the recursive descent when the desired depth is reached. The second example implements the Sierpiński curve; which has a very similar recursive algorithm. Microscheme can run these programs, with a high 'depth' or 'order' setting, with ease. In fact, it was necessary to insert a deliberate pause of 100 milliseconds at the base case: `(pause 100)`, because the Python simulator program could not draw the trace quickly enough. Figure 10, overleaf, shows some large outputs from these programs.

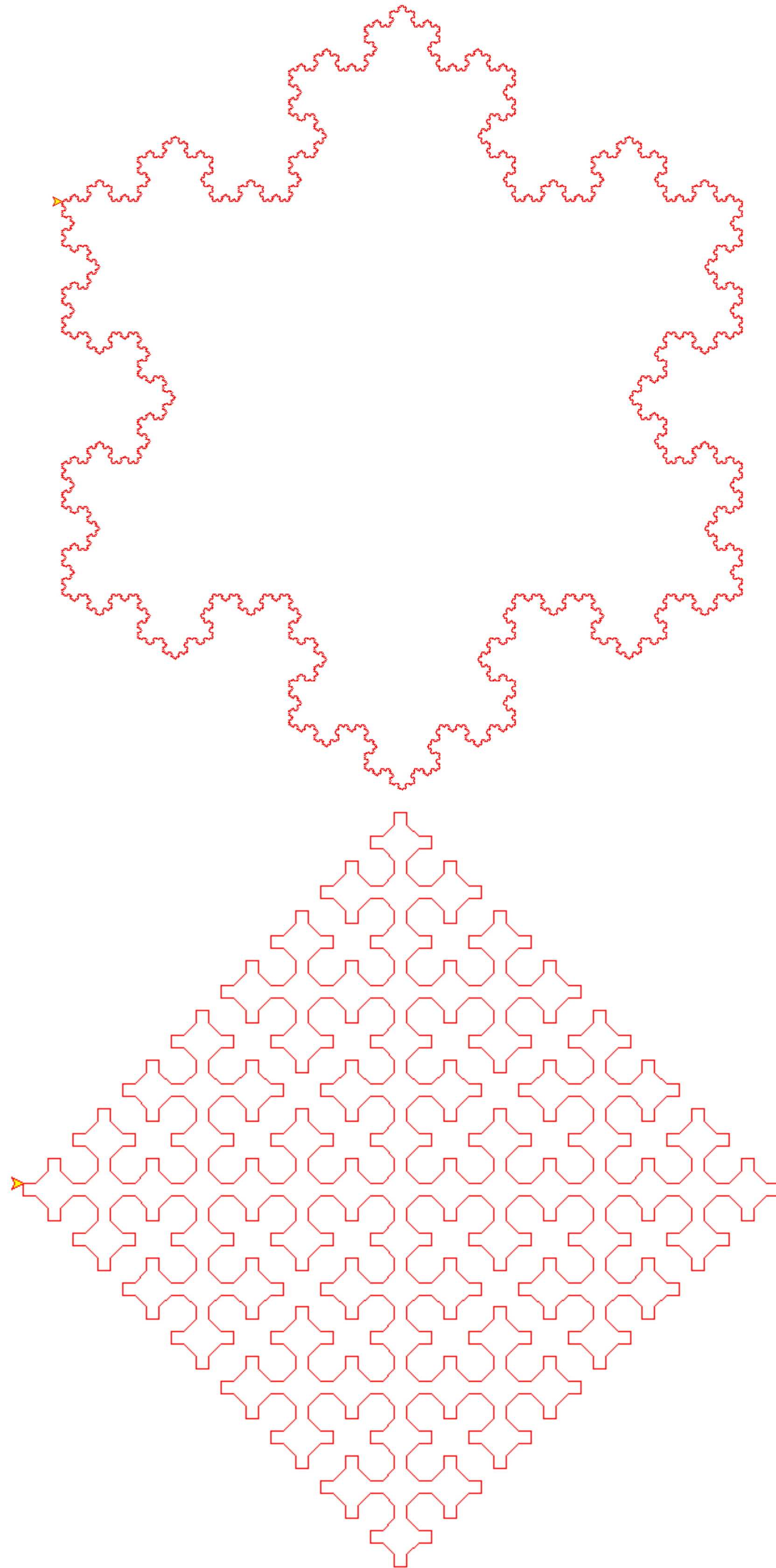


Figure 10: Koch Snowflake (Order 6) and Sierpiński curve (Order 4)

5.2 Comparative Analysis

The design efforts of this project have been focussed on minimising memory usage, rather than execution time. No specific time-optimisations have been implemented. However, in order to judge the usefulness of the implemented system, it is necessary to have a scientifically valid measurement for its performance.

The measurements were taken by running equivalent Microscheme and C specimen programs on an Arduino device, and measuring their execution times externally. Both programs were set to send distinct signals, via a serial connection, before and after the main program loop; while a simple Python script (appendix F) awaited the signals, and reported the time difference between them. The signals were sent immediately before and after the main loop, in order to exclude the initialisation activities of both systems.

The chosen specimen program (Appendices G & H) calculates the factorial of integers 1 to 7, 1000 times. This involves recursive procedure calls, arithmetic and iteration; and is a reasonable estimation for all computation. In order to make the comparison as fair as possible, the C program was written using the 'short' (16-bit) data type throughout, to match Microscheme's 15-bit arithmetic. Furthermore, the C program was written using recursive functions rather than *for loops*, so the two programs contain exactly the same algorithm. So, as far as possible, the test is comparing the performance of the two run-time systems, and not the respective merits of the functional and imperative paradigms.

In order to verify that this arrangement does indeed measure the execution time of the program, and is not corrupted by the asynchronous nature of the serial protocol, some calibration runs were made. First, using a *zero* program, where the *start* and *end* signals are sent with no delay. For this, both languages gave measurements of 0.0000 seconds, when rounded to 4 decimal places. Next, each language was measured using a 1-second program. This is achieved using the `(sleep 1000)` and `delay(1000);` constructs respectively, which use the known processor clockspeed to delay for an approximate number of milliseconds. The Microscheme and C programs took 0.999334 and 0.999453 seconds respectively. Therefore, the results are definitely meaningful up to the third decimal place (and probably more, considering the variance in `delay()` routines). Since the specimen program includes 1000×7 iterations, the results will fall well within this level of precision.

The average time of the C program, across 20 runs, was: 0.041 seconds. The Microscheme program took an average of 0.360 seconds. Therefore, Microscheme is approximately 9 times slower than C, as compiled by the official Arduino IDE.

6 Release

The last phase of the project was dedicated to making the compiler production-ready, and releasing it to the public. This involved creating binaries for Linux and Windows, example programs, libraries, and documentation. The compiler itself was also checked for memory leaks, and provisions for compilation error reporting were finalised, with a uniform table of error codes.

The development was carried out on a 64-bit Linux machine. In order to make the compiler available for a large audience, an automated script was produced to build both 64-bit and 32-bit binaries for Linux and Windows. Windows binaries are created trivially using the MinGW cross-compiler [33]. The four binaries are automatically archived, along with libraries and supporting materials, in either .tar.gz format or .zip format for Windows users. This setup allows for quick development and deployment of updated versions.

The libraries distributed with Microscheme, which are each example programs in their own right, provide basic functionality for a range of applications. The 'long.ms' and 'fixedpoint.ms' libraries implement double-precision integers and fixed-point real numbers respectively; each implemented using *pairs* of integers. Double-precision integers are useful for large calculations, and real (non-integer) numbers are crucial for scientific calculations, for example in sensor networks. The 'list.ms' library provides efficient list-processing procedures, including higher-order functions such as *map* and *fold*, as well as the type predicate `(list? ...)` and typecasting operators: `(vector->list ...)`. The library 'ascii.ms' provides the procedures `number->ascii` and `long->ascii` for printing numeric types as ascii-encoded strings.

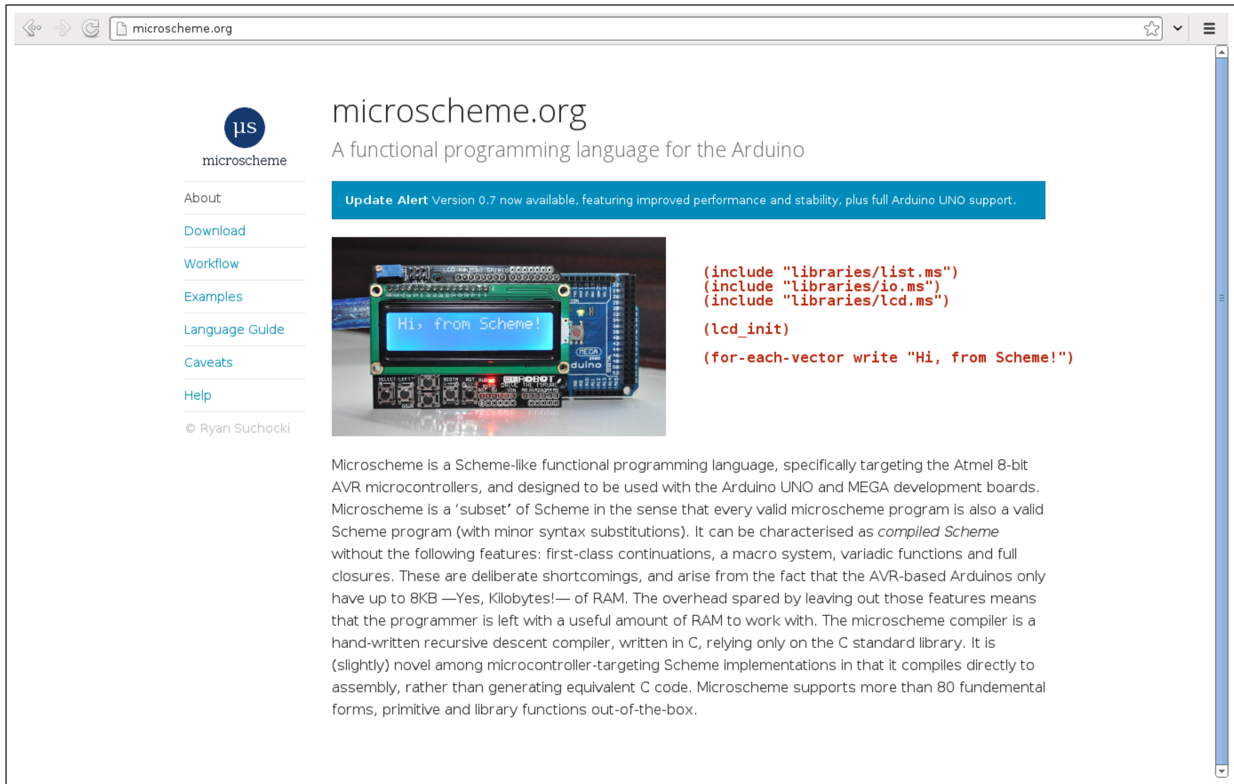
The libraries 'io.ms' and 'io_uno.ms' provide access to the *General Purpose Input/Output* (GPIO) pins for the Arduino MEGA and Arduino UNO models respectively. At the hardware level, these pins are used by reading and setting byte-sized registers, thus interfering with 8 pins at a time. Ideally, the user should be presented with a simple software interface, where individual pins are manipulated according to the numbers printed alongside them on the Arduino boards. The matter is complicated by the fact that the Arduino design does not maintain the on-chip ordering of these pins, so the printed pin numbers do not correlate with the appropriate hardware registers. The pin mappings are published by the Arduino project [34, 35], and must be implemented in software. This provision was made via a Microscheme library, rather than at the assembly level, because of the ease with which these mappings are expressed in Scheme:

```
(define arduino_ports
    (vector 41 41 41 41 41 41 41 41 35 35 35 35 35 35))
(define arduino_pins
    (vector 1  2  4  8  16 32 64 128 1 2  4  8  16 32))
```

Finally, the 'lcd.ms' library provides an interface for common LCD screens, as demonstrated in section 5.1. This is an example of the type of library that is developed for Arduino Shields. Shields are accessory circuits which plug directly onto the Arduino PCB. "The different shields follow the same philosophy as the original toolkit: they are easy to mount, and cheap to produce." [36]. The intention is that a novice user can purchase a shield, download the accompanying library, and start experimenting with little groundwork.

As well as some simple example programs, an extensive language guide (appendix B) was written. The language guide provides an elementary tutorial in Scheme programming, as well as an account of the particular characteristics of Microscheme. The guide contains explanations of compilation errors and runtime exceptions; and should be useful for new users and experienced Scheme programmers alike.

The supporting materials and examples are available on the Microscheme website: `http://microscheme.org`, along with download links and licensing details (See figure 11). The compiler is currently closed-source, and only the compiled binaries are released. In the future, the project will become fully *Open Source*, in the spirit of the larger Arduino project. In Febuary 2014, the project was made public, and initially publicised via a post on the Arduino community forum (`http://forum.arduino.cc/`). Later, interest was shown by the Scheme community (`http://community.schemewiki.org`). As of April 2014, the website has received around 300 unique visitors, which is overall a very positive response to this release effort.



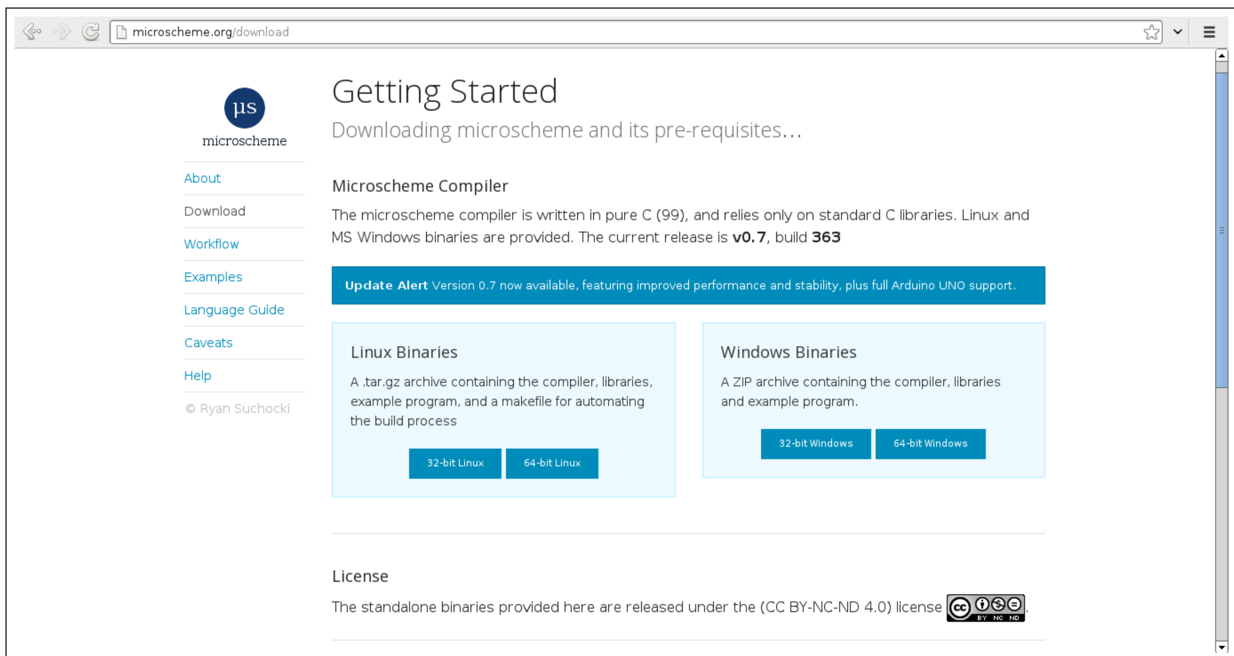


Figure 11: Public-facing microscheme.org website

7 Conclusions & Further Work

There were two underlying aims to this project. Firstly, to provide a working alternative language for the popular Arduino project. Secondly, to show that functional programming for micro-controllers in general is not only feasible, but practical and even preferable. The relevance of the project is therefore both practical and academic. The overall outcome is very positive, and the project is largely complete with respect to the initial specification. The few weaknesses of the project; and the future work which will be carried out to address them; are presented here in balance with the fulfilled objectives.

The first objective set out in the project specification was to design a suitable functional language for the Arduino. The result is the language Microscheme, which was defined formally (see appendix I) in the project progress report, which can broadly be summarised as 'Scheme, without first-class continuations'. A more elaborate, but less formal language definition is now provided by the Language Guide (appendix B). Unfortunately, there is no widespread existing Scheme/LISP dialect which eschews first-class continuations in this way; so Microscheme can only be described as a partial implementation of Scheme. In my contention, the design of Microscheme is sufficiently well thought-out and justified that it should be seen as a LISP dialect in its own right, which happens to be a subset of Scheme. The second objective: to implement a compiler for the above language; has been exceeded. Specifically, the compiler implements Microscheme as defined in the project specification; but it also supports extra syntactic sugar, and an extensive set of primitives.

The third objective: to implement a degree of program optimisation; was given less attention than originally intended. There are aspects of optimisation present in the compiler (tail-call elimination, dead-code elimination, procedure in-lining), but there is no dedicated optimising stage. It was decided to focus on preparing the compiler for a public release; rather than working on additional optimisations. Realistically, no-one is going to be using Microscheme in time-critical situations; and enough research has been done on functional language optimisations elsewhere, upon which this project could not conceivably improve.

The fourth and fifth objectives: to make the compiler production-ready, and to evaluate it; are the main indicators of the success of the project. The compiler is ready for real-world usage, as demonstrated by the variety of example programs and supporting materials, and is itself an efficient, memory-safe, maintainable piece of software. The

comparative analysis showed that, in general computation (i.e. function calls and arithmetic), Microscheme programs are approximately 10 times slower than C, as compiled by avr-gcc. For a functional language (which implicitly involves more type-checking and book-keeping at runtime) this is a modest result. A performance loss is generally accepted as the price of a richer, higher-level programming paradigm. However—considering the Microscheme compiler has received no specific optimisation effort, whereas GCC is one of the most popular, and therefore extensively developed compilers in the world—it is an extremely favourable result. The runtime system was designed with the Arduino (Atmel ATMega) architecture in mind, and is highly hand-tailored to the available hardware. Its performance is easily good enough for most Arduino use cases, as the timing of any program is likely to be limited by external factors such as serial communications and sensor readings, which are many *orders of magnitude* above the clock-speed of the micro-controller.

The completion of these objectives is the basis for the claim that functional programming on micro-controllers is feasible and practical. It has been shown that useful micro-controller programs can be expressed within the functional paradigm; and that the performance, even without a specific optimisation effort, is 'good enough' for typical applications. In the future, further optimisations will be implemented; thus reducing the performance gap, and making the system even more competitive. As argued in section 2.1: the idea that functional programming for the Arduino is *preferable* is a natural consequence of the stated target audience of the Arduino platform (artists, hobbyists), and the fact that Scheme is much more high-level than C.

This project provides a novel academic contribution. Section 2.2 described the most relevant related work, both published and online. Although there are some extremely compact Scheme implementations, they have not been shown to work well on real micro-controller hardware. Also, the existing literature focusses on virtual-machine implementations, rather than direct compilation; with obvious portability benefits. This project shows that, by focussing on a single (popular) platform, a system can be produced which is ultimately more useful in the real world. However, the academic contribution of this project goes beyond the practicality of direct compilation. This is the first functional micro-controller system which has been developed to completion, and to a state where it can be released to the public. It therefore proves, more satisfactorily than the existing literature, that functional programming *really works* in this context. Microscheme is, unequivocally, the first well-documented micro-controller-targetting Scheme implementation; and is the first alternative language designed specifically for the Arduino.

The results presented here are readily reproducible, as the compiler can be downloaded and verified, by anyone, at `http://microscheme.org`. The comparative analysis can be reproduced using the testing harness (appendix F) and specimen programs (appendices G & H). An Arduino device is required to verify the full competence of the compiler. As standalone software, however, the error-reporting behaviour of the compiler can be inspected, and the validity of the generated assembly code can be checked using the avr-gcc assembler.

The most serious weakness of this project is the lack of *garbage collection* (GC), which refers to the automatic recycling of heap memory space; and is the necessary counterpart to automatic memory allocation. This is not a deviation from the project objectives; but rather a long-standing deliberate omission. This decision was made because the development of a garbage collector would consume a significant amount of development time, and is simply beyond the scope of this project. The omission is not fatal, because it is possible in Scheme to program in a heap-conservative way. More specifically, Scheme allows data to be used in a mutable way, thus modifying data already on the heap rather than creating new immutable objects. This distinction was discussed in section 2.1, and was used to argue that LISP-like languages are appropriate for micro-controller applications (more so than purely functional languages). Therefore, a careful Scheme programmer can avoid the need for GC, and an implementation without it is not useless. However, the Scheme programmer ought to have the option of state-free programming. Furthermore, since the Arduino is an extremely memory-constrained platform, the challenges of GC implementation are particularly interesting. Therefore, an important future goal is to implement a garbage collector for Microscheme.

Since Microscheme is so closely tailored to the Atmel architecture, its future is dependant on that of the Atmel-based Arduino. There is currently a more powerful Arduino model (the Arduino Due) which uses an ARM core, with significantly more RAM and processing power. There are also similar products such as 'mbed' [37], using more powerful chips. These devices are significantly more expensive than the Atmel-based models. It is likely that the class of extremely-low-end Atmel chips will remain popular for some time, despite the ongoing miniaturisation of high-end processors, simply because of the extremely low price. One can purchase an ATmega328P chip with the Arduino bootloader pre-installed for around £3 online. Therefore—perhaps for the first time—Microscheme represents a way of deploying functional programs in inexpensive, even disposable devices.

In conclusion, this report has shown that the project is technically ambitious, and that the objectives of the project specification have been exceeded. The project contains several novel aspects, such as the unconventional *lexer* producing a *token tree*; and is sufficiently distinct from the related literature that it represents a new academic contribution. Microscheme is unique among micro-controller-targeting functional language implementations. It has been designed for real-world micro-controller applications, rather than as an experimental contrivance. Moreover, it is the first such project to be released to the public as a ready-to-use tool, with proper supporting materials. In its current state, it is capable of running impressive (and realistic) example programs, and is an asset to the Arduino community. With future development in the areas of optimisation and garbage collection, Microscheme could be an asset to the functional programming community also.

References

[6] R.J. Bird and P. Wadler. *An Introduction to Functional Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988. ISBN: 9780134841892.

[7] John Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[8] John McCarthy. *Recursive functions of symbolic expressions*. Springer, 1983.

[9] Philip Wadler. "Comprehending monads". In: *Mathematical Structures in Computer Science* 2.04 (1992), pp. 461–493.

[10] Gerald J Sussman and Guy L Steele Jr. "Scheme: A interpreter for extended lambda calculus". In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 405–439.

[12] Harold Abelson et al. "Revised4 report on the algorithmic language scheme". In: *ACM SIGPLAN Lisp Pointers* 4.3 (1991), pp. 1–55.

[13] Mitchell Wand. "Continuation-based multiprocessing". In: *Proceedings of the 1980 ACM conference on LISP and functional programming*. ACM. 1980, pp. 19–28.

[14] Christopher T Haynes and Daniel P Friedman. "Engines build process abstractions". In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM. 1984, pp. 18–24.

[15] John H Reppy. "CML: A higher concurrent language". In: *ACM SIGPLAN Notices*. Vol. 26. 6. ACM. 1991, pp. 293–305.

[16] Vincent St-Amour and Marc Feeley. "PICOBIT: a compact scheme system for microcontrollers". In: *Implementation and Application of Functional Languages*. Springer, 2011, pp. 1–17.

[17] Edsger W Dijkstra. "Recursive programming". In: *Numerische Mathematik* 2.1 (1960), pp. 312–318.

[18] Andrew W Appel and Zhong Shao. "Empirical and analytic study of stack versus heap cost for languages with closures". In: *Journal of Functional Programming* 6.1 (1996), pp. 47–74.

[21] Andrew W Appel. "Garbage collection can be faster than stack allocation". In: *Information Processing Letters* 25.4 (1987), pp. 275–279.

[23] Danny Dubé and Université de Montréal. "BIT: A very compact Scheme system for embedded applications". 2000.

[24] Danny Dubé and Marc Feeley. "Bit: A very compact scheme system for microcontrollers". In: *Higher-order and symbolic computation* 18.3-4 (2005), pp. 271–298.

[25] Danny Dubé. “Picbit: A scheme system for the pic microcontroller”. In: *Scheme 2003* (2003), p. 7.

[27] Harold Abelson et al. “Revised5 report on the algorithmic language Scheme”. In: *Higher-order and symbolic computation* 11.1 (1998), pp. 7–105.

[28] Abdulaziz Ghuloum. “An incremental approach to compiler construction”. In: *Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR*. Citeseer. 2006.

[30] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. MIT Press, 1996. ISBN: 9780262011532.

[31] R.K. Dybvig and J.P. Hébert. *The Scheme Programming Language*. University Press Group Limited, 2009. ISBN: 9780262512985.

[32] J. Tang. *Write Yourself a Scheme in 48 Hours*. Ioss, Llc, 2007.

[38] R.K. Dybvig. *The Scheme Programming Language*. MIT Press, 2003. ISBN: 9780262541480.

Technical Manuals

[11] Atmel Corporation. *AVR Instruction Set Manual*. 2010.

[19] Atmel Corporation. *Atmel ATmega328P Datasheet*. 2009.

[20] Atmel Corporation. *Atmel ATmega2560 Datasheet*. 2009.

Online Sources

[1] Various Authors. *Arduino Website*. URL: `http://www.arduino.cc`.

[2] Various Authors. *avr-gcc Website*. URL: `http://gcc.gnu.org/wiki/avr-gcc`.

[3] Various Authors. *avrdude Website*. URL: `http://www.nongnu.org/avrdude`.

[4] Various Authors. *avr-libc Website*. URL: `http://www.nongnu.org/avr-libc`.

[5] Various Authors. *wiring Website*. URL: `http://wiring.org.co`.

[22] Various Authors. *Scheme Community Wiki*. URL: `http://community.schemewiki.org/?scheme-faq-standards`.

[26] Hubert Montas. *ARMPIT Scheme*. 2006. URL: `http://armpit.sourceforge.net/`.

[29] Peter Michaux. *Scheme from Scratch*. 2010. URL: `http://michaux.ca/articles/scheme-from-scratch-introduction`.

[33] Various Authors. *MinGW*. 2012. URL: http://www.mingw.org/.

[34] Various Authors. *Arduino UNO Pin Mapping*. URL: http://arduino.cc/en/Hacking/PinMapping168.

[35] Various Authors. *Arduino MEGA Pin Mapping*. URL: http://arduino.cc/en/Hacking/PinMapping2560.

[36] Various Authors. *Arduino Shields*. URL: http://arduino.cc/en/Main/ArduinoShields.

[37] Various Authors. *mbed platform*. 2014. URL: https://mbed.org/.

Non-cited sources

These texts and tools are not cited directly in the text, but were influential in principle.

[39] S.L.P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall international series in computer science. Prentice/Hill International, 1987. ISBN: 9780134533339.

[40] Simon L. Jones and David R. Lester. *Implementing functional languages: A tutorial*. Prentice Hall. New York., 1992. ISBN: 9780137219520.

[41] D.P. Friedman and M. Felleisen. *The Little Schemer*. MIT Press, 1996. ISBN: 9780262560993.

[42] Henry G Baker. "CONS should not CONS its arguments, part II: Cheney on the MTA". In: *ACM Sigplan Notices* 30.9 (1995), pp. 17–20.

[43] Peter. Norvig. *How to Write a (Lisp) Interpreter (in Python)*. 2010. URL: http://norvig.com/lispy.html.

[44] Matt Godbolt. *Interactive GCC Compiler*. 2012. URL: http://gcc.godbolt.org/.

Appendices

Appendix A: Prototype Listing
Appendix B: Microscheme Language Guide
Appendix C: Microscheme Factorial Demo
Appendix D: Microscheme Fractal Demo (Koch Snowflake)
Appendix E: Microscheme Fractal Demo (Sierpinski Curve)
Appendix F: Benchmarking Timer Program (Python)
Appendix G: Benchmarking Test Program (Scheme version)
Appendix H: Benchmarking Test Program (C version)
Appendix I: Microscheme Formal Grammar

Appendix A: Prototype Listing

```
1  import Data.Bits
2  -- l_lexr (Parenthesized l_lexer)
3  data TokenNode = Keyword String | Primword String | Numeric String |
       Identifier String | ESub [TokenNode] deriving (Show, Eq)
4
5  l_lex :: String -> [TokenNode]
6  l_lex s = fst $ l_lex' [] [] s
7
8  l_lex' :: String -> [TokenNode] -> String -> ([TokenNode], String)
9  l_lex' acc acct (c:cs)
10     | c `elem` " \n\r\t" && (acc == "") = l_lex' acc acct cs
11     | c `elem` " \n\r\t" = l_lex' [] (if (acc /= "") then (acct ++ [
           classify acc] ) else acct) cs
12     | c == '(' = let (result, rest) = l_lex' [] [] cs
13          in l_lex' [] ((if (acc /= "") then (acct ++ [classify acc] )
              else acct) ++ [ESub result]) rest
14     | c == ')' = ((if (acc /= "") then (acct ++ [classify acc] ) else
           acct), cs)
15     | otherwise = l_lex' (acc ++ [c]) acct cs
16 l_lex' acc acct [] = ((if (acc /= "") then (acct ++ [classify acc] )
       else acct), "")
17
18 classify :: String -> TokenNode
19 classify raw_token
20     | raw_token `elem` ["quote", "if", "set!", "define", "lambda", "
           begin"] = Keyword raw_token
21     | raw_token `elem` ["=", "not", "+", "-", "*", "serial-send"] =
           Primword raw_token
22     | all (\c -> c >= '0' && c <= '9') raw_token = Numeric raw_token
23     | otherwise = Identifier raw_token
24
25 -- l_parser:
26
27 data Form = Const Num' | Var_ref Var | Quote Form | Seq [Form] |
       ProcCall Form [Form] | PrimCall String [Form] | Asg Var Form | Def
       Var Form | Cond Form Form Form | ProcDef [Var] Form deriving (Show,
       Eq)
28
29 type Var = String
30 type Num' = Int
31
32 parse_prog :: [TokenNode] -> [Form]
33 parse_prog = map l_parse
```

```
34
35 l_parse :: TokenNode -> Form
36
37 -- FUNDEMENTAL Forms:
38
39 l_parse (Numeric num)
40     = Const (read num)
41 l_parse (Identifier var)
42     = Var_ref var
43 l_parse (ESub (Keyword k : exps))
44     | k == "begin" = seqq (map l_parse exps)
45 l_parse (ESub [Keyword k, Identifier var, exp])
46     | k == "set!" = Asg var (l_parse exp)
47 l_parse (ESub [Keyword k, Identifier var, exp])
48     | k == "define" = Def var (l_parse exp)
49 l_parse (ESub [Keyword k, test, conseq, alt])
50     | k == "if" = Cond (l_parse test) (l_parse conseq) (l_parse alt)
51 l_parse (ESub (Keyword k : ESub vars : body))
52     | k == "lambda" = ProcDef (map l_parseId vars) (seqq (map l_parse
       body))
53
54 -- DERIVED FORMS:
55
56 l_parse (ESub (Identifier k : ESub binds : body))
57     | k == "let" = let (lhs, rhs) = unzip (map parseBind binds)
58         in ProcCall (ProcDef (lhs) (seqq (map l_parse body))) (rhs)
59
60 -- PROCEDURE CALL (Anything Else...)
61
62 l_parse (ESub (Primword p : exps))
63     = PrimCall p (map l_parse exps)
64 l_parse (ESub (proc : exps))
65     | otherwise = ProcCall (l_parse proc) (map l_parse exps)
66
67 -- This shouldn't occur:
68 l_parse tokens = error ("Malformed or Unknown Form: " ++ (show tokens))
69
70 seqq :: [Form] -> Form
71 seqq [only] = only
72 seqq more = Seq more
73
74 l_parseId :: TokenNode -> Var
75 l_parseId (Identifier i) = i
76 l_parseId i = error ("Only parameter names allowed here: " ++ (show i))
77
```

```
78 parseBind :: TokenNode -> (Var, Form)
79 parseBind (ESub ([e1, e2])) = (l_parseId e1, l_parse e2)
80 parseBind i = error ("Only binding pairs allowed here: " ++ (show i) ++
       " Remember, each binding must have its own parentheses, even if
      there's only one.")
81
82 ---------------------------------------------------
83
84 emit_instr :: Form -> String
85
86 emit_instr (Const n) = "Move constant '" ++ (show n) ++ "' into result
      register.\n"
87 emit_instr (Cond test conseq alt) = "Evaluate: {\n" ++ (emit_instr test
      ) ++ "}. Compare result. If true, do this: {\n" ++ (emit_instr
      conseq) ++ "}, otherwise do this: {\n" ++ (emit_instr alt) ++ "}.\n"
88 emit_instr (Var_ref var) = "Lookup the variable '" ++ (show var) ++ "
      '.\n"
89 emit_instr (Seq forms) = foldr (++) "" $ map emit_instr forms
90 emit_instr (ProcCall proc args) = (foldr (++) "" $ map (\arg -> (
      emit_instr arg) ++ "Copy result to outgoing call frame\n" ) args) ++
       "Evaluate this expression (should eval to a procedure): {\n" ++ (
      emit_instr proc) ++ "} fetch the corresponding closure, and call the
       procedure.\n"
91 emit_instr (Asg var expr) = "Evaluate: {\n" ++ (emit_instr expr) ++ "}
      And bind '" ++ (show var) ++ "' to the result."
92 emit_instr (Def var expr) = "Evaluate: {\n" ++ (emit_instr expr) ++ "}
      Allocate new storage for '" ++ (show var) ++ "', and bind it to the
      result."
93 emit_instr (ProcDef fargs body) = "Create a new closure on the heap.
      Closure contains pointer to the following instructions: {\n" ++ (
      emit_instr body) ++ "} Set result register to a procedure pointing
      to the new closure.\n"
94
95 ---------------------------------------------------
96
97 compile :: String -> IO ()
98 compile s = putStr $ foldr (++) "" $ map (emit_instr [] "") $ map
      l_parse $ l_lex s
```

Appendix B: Language Guide

A crash course in the particular workings of microscheme

Learning Scheme

The existing wealth of tutorials and crash-courses in Scheme are really very good, and I shall not attempt to better them. For example, Teach Yourself Scheme in Fixnum Days. On the other hand, I have included enough detail that the ambitious hacker could reasonably learn a lot by tinkering with the examples, and refering to this guide. I do recommend that the novice reader follows at least some general introduction to functional programming—which I cannot provide— before proceeding.

Fundemental Forms

A microscheme program is a list of expressions, which are evaluated in order when the program runs. Each expression—except for constants like 4, #t and "hello"—takes one of the ten fundemental forms described here. Each fundemental form is composed of a pair of parentheses (brackets), containing keywords, lists and subexpressions. Each subexpression must also be a constant or fundemental form, and so on.

Abstraction: (lambda (X Y Z . . .) B . . .)

Every fundemental form is surrounded by a pair of parentheses (brackets). The lambda form produces a procedure (a computational unit which models some function). Its name reflects that fact that it represents a lambda abstraction as found in Lambda Calculus. The keyword lambda is followed by a list of variable names, which are the arguments of the procedure; as well as one or more expressions which form the body of the procedure. When the procedure is applied (i.e. called, invoked) the body expressions are executed in order, and the value of the final expression is returned. Procedures produced in this way are intrinsically anonymous in Scheme, and they are first-class values. This means that the lambda form evaluates to an thing representing the procedure, which is not automatically given a name. We can bind that thing to a name using (define ...), but we are not obliged to. Instead, we could pass it directly to another procedure, or return it from an enclosing procedure. For example, (lambda (x) (+ x 1)) represents a procedure which takes one argument, assumes it has type number, and returns the number 1 higher.

Application: (<procedure> A B C . . .)

The procedure application form does not contain any keywords. It is composed by writing an expression <procedure>, followed by some number of arguments, all inside a pair of parentheses. The <procedure> expression must evaluate to a procedure in some way. That means it can either be a primitive procedure name (as listed below), a lambda expression, a variable name, or another procedure application. If you give a variable name, then that variable must be bound to a procedure by the time the application is reached. If you give another procedure application as <procedure>, then that application must return something of type procedure. In any case, the number of arguments (A B C . . .) given must match the number of arguments expected by <procedure>. '+' is the name of a primitive procedure, taking two arguments. Hence, (+ 3 7) is a valid procedure application which evaluates to 10. Since expressions of any complexity can be used as the two arguments to +, and likewise for other math operators, this form can be used to write any arithmetic expression in prefix notation. To give a richer example, the code given for the lambda form above is a valid expression, which evaluates to something of type procedure, expecting a single numeric argument. So, we can form an application like this: (<example from above> <any numeric argument>). Writing it out in full gives: ((lambda (x) (+ x 1)) 5) which evaluates to 6.

Definition: (define <name> <expr>)

The two forms we've seen so far are actually powerful enough to express any program that our electronic computers can compute. (See wikipedia, of course) But, in practice, we can get a lot more done if we give names to things, and use them over and over again. The define form takes a variable name and an expression. It evaluates the expression, and binds the result to the given name (i.e., it stores the result in the variable <name>). From that point onwards, <name> refers to the thing produced by the expression, which could be of any type, even procedure. In microscheme, definitions are only allowed at the top-level. Definitions within the body of some form must be achieved using (let . . .). Combining the three forms above, we can write the following program, which results in the variable 'theothernumber' being bound to the value 6:

```
(define plusone (lambda (x) (+ x 1)))
(define thenumber 5)
(define theothernumber (plusone 5))
```

Definition (again): (define (<proc> X Y Z . . .) B . . .)

Since the pattern for defining a named function: (define <procname> (lambda (. . .) . . .)) is so frequently used in Scheme programs, a shorthand notation is provided for it. The first definition of the program above can be rewritten as (define (plusone x) (+ x 1)). Under Scheme's semmantics, these expressions are precisely equivalent. It is (slightly) important that the programmer realises that this is a library form, i.e. it is compiled just as a lambda inside a define form.

Assignment: (set! <name> <expr>)

Assignment looks just like definition, but with the 'set!' keyword instead of 'define'. This is used to change the value to which some variable name is bound. That could be a global variable, which is introduced by (define . . .), a procedure argument, or a local variable introduced by (let . . .) The set keyword includes an exclamation mark to remind you that it is changing the state of the system; and this is why Scheme is considered not to be purely functional.

Conditional: (<predicate> <consequent> <alternative>)

The conditional form takes at least a predicate and a consequent, and optinally an alternative. Each of these are expressions of any kind. If the predicate evaluates to true (In Scheme, anything other than false, denoted #f, counts as true) then the consequent will be evaluated. If the predicate evaluates to false, and an alternative is given, then it will be evaluated. This is subtly different from the conditional branches of imperative programming. As well as making a decision about which expression to evaluate, the conditional itself inherits the value of whichever branch is chosen. This means you can use the whole expression as a subexpression, whose value depends on the predicate. e.g. (+ 1 (if (= 2 3) 7 13)) evaluates to 14.

Conjunction: (and A B C . . .)

The conjunction form takes any number of arguments, each of which is a subexpression. It will evaluate those expressions in order. If it reaches one that evaluates to false (denoted #f), then it will stop and return #f. If none of them evaluates to #f, then the value of the final expression will be returned (remember, anything other than #f is considered true). Using this form with zero arguments is equivalent to the true constant #t.

Disjunction: (or A B C . . .)

Like conjunction, this disjunction form evaluates each of its arguments in order. If any one of them evaluates to anything other than #f, it stops and returns that value. If it reaches the end of the list, and every expression evaluated to #f, then it returns #f. Using this form with zer oarguments is equivalent to the false constant #f.

When used with Boolean type values, the conjunctive and disjunctive forms work just like boolean operators in imperative languages. (or #f #t #f) evaluates to #t and so forth. In Scheme, however, these forms perform a much more powerful function. Since they are variadic, and will keep evaluating until a false or true subexpression is reached respectively, they can be used as control-flow mechanisms in place of nested (if . . .) forms.

Local Binding: (let ((a X) (b Y) (c Z) . . .) B . . .)

The let form is used to bind names to values only for a specific part of the program. The first argument to let is a list of binding pairs. Each binding pair is a pair of brackets containing a variable name and an expression. The expressions that are given as the body 'B . . . ' are evaluated with those names bound to their corrseponding values, and the value of the final expression is returned. Those bindings do not persist outside of the let form. For any code outside of the let's parentheses, the variables a b c . . . are unchanged, and may not be defined at all.

Even if you only give one binding pair, the parentheses around the list of binding pairs is still needed. Hence, you end up with double brackets: (let ((x 5)) (+ x 1)). Missing those is a common mistake. The variable bindings apply in the body, but not within other binding pairs in the list. i.e., the expression Y should not rely on X being bound to a.

Sequence: (begin B1 B2 . . .)

Finally, you can group together expressions with the sequencial form, using the begin keyword. The whole thing is treated as one expression, whose subexpressions are executed in sequence. As usual, the value of the final subexpression is returned for the overall expression. You can use this in cases where you want to guarantee a group of expressions will be evaluated, or where you want to give multiple expressions in a context where only one is expected. (+ 1 (begin 2 4 6)) evaluates to 7. The subexpressions 2 and 4 are evaluated, but they have no effect. 6 is evaluated and returned to the outer + procedure.

Primitive Procedures

Primitive procedures are procedures that are built-in to the language. This means that the compiler produces efficient low-level routines for them. Unlike full-blown Scheme, microscheme primitives are not first-class. i.e., they can only appear in the function application form. This is a problem when you want to pass a primitive function as the argument to a higher-order function such as map. For example, you may want to invert a list of Booleans: (map not list_of_booleans). The solution is to make a simple wrapper-function which is first-class but performs the same function as the primitive you want to work with: (define (not* x) (not x)). Then, you are free to use it as a value: (map not* list_of_booleans). This might seem annoying, but it is not without good reason. Making all primitive functions first-class would tie up around .5 KB of RAM. On the arduino, RAM is precious. This compromise ammounts to you, the programmer, telling the compiler exactly which primitives need to be loaded into RAM. For the vast majority of programs, this ammounts to a massive memory saving. The primitive procedures built-in to compiler version 0.6 are:

- =, >, >=, <, <=, not
- +, -, *, div, mod, zero?
- number?, pair?, vector?, procedure?, char?, boolean?, null?
- cons, car, cdr, set-car!, set-cdr!
- list, vector
- vector-length, vector-ref, vector-set!
- assert, error
- include, stacksize, heapsize, pause
- serial-send, digital-state, set-digital-state

Type System

Microscheme has a strong dynamic type system. It is strong in the sense that all values have a specific, definite type; no type coersion occurs; procedures are generally valid for a specific set of types; and type exceptions are raised when procedures are applied to values of the wrong type.

It is dynamic in the sense that a variable is not restricted to hold values of a certain type. The type of value to which a variable name will be bound is not known until run-time, and can change as the program progresses.

The built-in types are: Number, Char, Boolean, Pair, Vector, 'The Empty List' aka null, which is said to have a type of its own, and Procedure. From these basic types we can infer compound types. A List is defined to be something of the type null or pair where the value of the cdr field has type List. This definition is effectively implemented by the (list? ...) function in the 'list.ms' library. Even though the built-in numeric type is fairly restricted (15-bit unsigned integer), a richer numeric stack can be built using combinations of pairs, vectors and numbers. For example, the 'xtypes.ms' library provides types long and fp, which represent 8-digit unsigned integers, and 4+4 digit fixed-point real numbers respectively.

For every type, there is a predicate function which answers the question 'is this value of type X'. These predicates are consistently formed by appending a question mark to the type name. For example, (number? 4) evaluates to #t. (boolean? 4) evaluates to #f. (boolean? (number? 4)) evaluates to #t. Procedures for converting between types are formed with an arrow between the type names, e.g. (vector->list a b). These conversions are not provided for many types, but they can be written manually.

Microscheme Libraries

Microscheme supports the (include ...) primitive, which effectively loads the whole contents of another file into the program. This allows commonly used program segments to be saved in 'libraries' that can be included in any other program. Typically, libraries contain definitions, but do not perform any input or output, so including them simply makes a set of procedures and data structures available to the program. Some useful libraries are included with microscheme, and more will become available as the project matures:

libraries/io.ms provides digital Input/Output functionality. This allows you to work with the Arduino's digital I/O pins, using the indices given to them on the arduino board. It provides the procedures:

- (set-ddr N X) to set the DDR (data-direction-register) for a pin. N is the pin number. X is #f for 'input' and #t for 'output'.
- (get-ddr N) returns a boolean representing the DDR value for pin N. #t means 'output'.
- (set-pin N Y) sets the value (high or low) for pin N.
- (set-pin N Y) gets the value (high or low) of pin N.

libraries/list.ms provides various functions for working with lists, which are linear data structures built using pairs and null. Procedures provided include:

- (list? X) returns true if and only if X is a list.
- (reverse X) if X is a list, returns a new list which is the reverse of it.
- (map P X) returns a list formed by performing procedure P on every element of list X.
- foldr, foldl, for-each, all various common higher-order list procedures.
- (vector->list V) returns a list whose elements are identical to those of vector V.
- The primitive (list . . .) for building lists is built-in, and implemented efficiently by the compiler.

libraries/long.ms provides an implementation for 8-digit unsigned integers:

- (long hi lo) forms a long where hi represents the high four digits and lo represents the low four digits of the number. The number 994020 is produced by
- (long 99 4020).
- (hi X) and (lo X) extract the high and low parts of a long.
- (long? X) returns true if X is a valid long. Warning: any pair of numbers will satisfy this.
- l+ l- l* l/ standard arithmetic operators.
- l= l< l> l<= l>= standard numeric comparators.

libraries/fixedpoint.ms provides an implementation for 5+5 digit unsigned fixed-point reals:

- (fp whole frac) forms an fp (fixed-point-number) where whole represents the whole part and frac represents the five digits of the fractional part. The number 4.5 is produced by (fp 4 5000), which can be read as 4.5000.
- (whole X) and (frac X) extract the whole and fractional parts of an fp.
- (fp? X) returns true if X is a valid fp. Warning: any pair of numbers will satisfy this.
- fp+ fp- fp* fp/ standard arithmetic operators.
- fp= fp< fp> fp<= fp>= standard numeric comparators.

NB: including the xtypes library has the same effect as including long and fixed-point individually, but saves memory by taking advantage of the overlap between their functions.

Compiler Errors

- 0 Out of memory
- 1 Char buffer full
- 2 while lexing the file '...'. File could not be opened
- 3 Comment before end of token
- 4 Extraneous)
- 5 Missing)
- 6 Procedure '...' is primitive, and cannot be used as a value
- 7 Non-identifier in formal argument list
- 8 Malformed lambda. No formals given
- 9 Wrong number of operands to IF form
- 10 First operand to SET should be IDENTIFIER
- 11 Wrong number of operands to SET form
- 12 Wrong number of operands to DEFINE form
- 13 [Syntax] Non-identifier in formal argument list
- 14 First operand to DEFINE should be IDENTIFIER or PARENS
- 15 [Syntax] Definition not allowed here
- 16 Malformed Binding
- 17 Malformed LET?
- 18 First operand to INCLUDE should be STRING
- 19 Wrong number of operands to INCLUDE form
- 20 Unknown parenthesized form
- 21 Unknown form
- 22 Unexpected list of expressions
- 23 NOT IN SCOPE
- 24 Integer constant too large
- 25 Freevar refs of degree > 1 not supported yet
- 26 Internal Error

Runtime Exceptions

Like Scheme, microscheme is strongly, dynamically typed. Exceptions are semmantic errors that arise at runtime. Microscheme makes use of the Arduino's built-in LED on digital pin 13 to give on-device indications of these situations. Generally, exceptions are not recoverable, and the device will need to be reset if an exception is raised. While it is possible to use digital pin 13 for general input and output, it is highly recommended to leave it free for exception indication.

Status	Meaning	Indication
RUN	Program Running	No Light
NVP	Not a Valued Procedure	Single Flashes
NAR	Number of ARguments	2 Flashes
NAN	Not A Number	3 Flashes
NAP	Not A Pair	4 Flashes
NAV	Not A Vector	5 Flashes
OOB	Out Of Bounds	6 Flashes
DBZ	Divide By Zero	7 Flashes
ERR	Custom Exception	Continuous Flashes
HALT	Program Completed	Continuous Light

NVP: A procedure application takes the form (proc X1 X2 ... Xn) where proc is an expression. At the time of application, if proc does not evaluate to a procedure, such as the result of a (lambda . . .) form, or a variable bound to a procedure, NVP will be raised.

NAR: A procedure application takes the form (proc X1 X2 ... Xn) where X1 X2 ... Xn are arguments. At the time of application, if proc evaluates to a procedure taking m arguments, but m $\neq$ n, then NAR will be raised.

NAN: Indicates that an arithmetic operator (+, -, *, /, div, mod) received an argument that did not evaluate to a number.

NAP: Indicates that a pair operator (car, cdr, set-car!, set-cdr!) received an argument that did not evaluate to a pair.

NAV: Indicates that a vector operator (vector-ref, vector-set!) received an argument that did not evaluate to a vector.

OOB: Indicates that a vector operator (vector-ref, vector-set!) received an index that was outside the dimensions of the vector given.

DBZ: Indicates an attempt to divide by zero.

ERR: This exception is raised manually by the programmer. See (error) and (assert expr) in the language guide.

Appendix C: Microscheme Factorial Demo

```
1  ;; Factorial Demo
2  ;; microscheme.org
3
4  (include "libraries/list.ms")
5
6  (define (factorial n)
7    (if (zero? n)
8      1
9      (begin (serial-send (stacksize))(* (factorial (- n 1)) n))))
10
11 (serial-send (factorial 7))
12
13 (define (tail-factorial n acc)
14   (if (zero? n)
15     acc
16     (begin (serial-send (stacksize)) (tail-factorial (- n 1) (* acc n))
         )))
17
18 (serial-send (tail-factorial 7 1))
19
20 (include "libraries/long.ms")
21
22 (define (factorial n)
23   (if (l= n l0)
24     l1
25     (begin (serial-send (stacksize))(l* (factorial (l- n l1)) n))))
26
27 (let ((it (factorial (long 0 11))))
28   (serial-send (hi it))
29   (serial-send (lo it)))
30
31 (define (tail-factorial n acc)
32   (if (l= n l0)
33     acc
34     (begin (serial-send (stacksize)) (tail-factorial (l- n l1) (l* acc
         n)))))
35
36 (let ((it (tail-factorial (long 0 11) l1)))
37   (serial-send (hi it))
38   (serial-send (lo it)))
```

Appendix D: Microscheme Fractal Demo

```
;; Koch Snowflake
;; microscheme.org

(include "libraries/turtle.ms")

(define (segment length level)
    (if (zero? level)
        (begin
            (forward length)
            (pause 100))
        (begin
            (segment (div length 3) (- level 1))
            (left 60)
            (segment (div length 3) (- level 1))
            (right 120)
            (segment (div length 3) (- level 1))
            (left 60)
            (segment (div length 3) (- level 1)))))

(define (snowflake length level)
    (segment length level)
    (right 120)
    (segment length level)
    (right 120)
    (segment length level)
    (right 120))

(start)

(snowflake 400 4)

(end)
```

Appendix E: Microscheme Fractal Demo 2

```
1  ;; Sierpinski Curve
2  ;; microscheme.org
3
4  (include "libraries/turtle.ms")
5
6  (define (half_sierpinski size level)
7      (if (zero? level)
8          (begin
9              (forward size)
10             (pause 100))
11         (begin
12             (half_sierpinski size (- level 1))
13             (left 45)
14             (forward (div (* size 141) 100))
15             (left 45)
16             (half_sierpinski size (- level 1))
17             (right 90)
18             (forward size)
19             (right 90)
20             (half_sierpinski size (- level 1))
21             (left 45)
22             (forward (div (* size 141) 100))
23             (left 45)
24             (half_sierpinski size (- level 1)))))
25
26 (define (sierpinski size level)
27     (half_sierpinski size level)
28     (right 90)
29     (forward size)
30     (right 90)
31     (half_sierpinski size level)
32     (right 90)
33     (forward size)
34     (right 90))
35
36 (start)
37
38 (sierpinski 10 4)
39
40 (end)
```

Appendix F: Benchmarking Timer Program (Python)

```
1  # Arduino Benchmarking Program
2  # By Ryan Suchocki. http://microscheme.org
3
4  import serial
5  import time
6
7  ser = serial.Serial("/dev/ttyACM1", 9600)
8  ser.flushInput()
9  ser.close()
10
11 ser.open()
12
13 b1 = 0
14 b2 = 0
15 val = 0
16
17 t = time.time()
18
19 while 1:
20     b1 = ser.read();
21     b2 = ser.read();
22     val = (ord(b1) << 8) + ord(b2)
23     print (str(val))
24
25     if val == 10:
26         t = time.time()
27
28     elif val == 20:
29         print (str(time.time() - t))
```

Appendix G: Benchmarking Test Program (Scheme)

```
;; Benchmarking Specimen Program
;; Calculates the factorial of integers 1..7, 1000 times
;; microscheme.org

(define (tail-factorial-aux n acc)
  (if (zero? n)
    acc
    (tail-factorial-aux (- n 1) (* acc n))))

(define (tail-factorial n)
    (tail-factorial-aux n 1))

(define (for-range f a b)
    (f a)
    (if (not (= a b))
        (for-range f (+ a 1) b)))

(define (times n f)
    (if (zero? n)
        #t
        (begin
            (f)
            (times (- n 1) f))))

(serial-send 100) ; To identify that this is the Scheme version
(serial-send 10)   ; To start the timer
(times 1000 (lambda () (for-range tail-factorial 1 7)))
(serial-send 20)   ; To end the timer
```

Appendix H: Benchmarking Test Program (C)

```
1  /* Benchmarking Specimen Program | microscheme.org
2     Calculates the factorial of integers 1..7, 1000 times        */
3
4  short x = 0;
5
6  void setup() {
7    Serial.begin(9600);
8  }
9
10 void loop() {
11   sendWord(200); // To identify that this is the C version
12   sendWord(10);   // To start the timer
13   outerstep(1000);
14   sendWord(20);   // To end the timer
15 }
16
17 void outerstep (short n){
18   if (n == 0) return;
19   else {
20     innerstep(7);
21     outerstep(n-1);
22   }
23 }
24
25 void innerstep (short m){
26   if (m == 0) return;
27   else {
28    x = factorial(m);
29    innerstep (m-1);
30   }
31 }
32
33 short factorial(short i){
34   if (i == 0) return 1;
35   else return i * factorial(i-1);
36 }
37
38 void sendWord(short word) {
39   Serial.write(word >> 8);
40   Serial.write(word & 255);
41 }
```

Appendix I: Microscheme Formal Grammar

Here is a formal grammar generating the language of valid μScheme programs, notated in Extended Backus-Naur Form. This is a simplification of a grammar[38] corresponding to the R5RS Scheme standard.

⟨*program*⟩ ::= ⟨*form*⟩*

⟨*form*⟩ ::= ⟨*definition*⟩ | ⟨*expression*⟩

⟨*definition*⟩ ::= (`define` ⟨*variable*⟩ ⟨*expression*⟩)
  | (`define` (⟨*variable*⟩ ⟨*variable*⟩*) ⟨*expression*⟩+)

⟨*expression*⟩ ::= ⟨*constant*⟩
  | ⟨*variable*⟩
  | (`lambda` ⟨*formals*⟩ ⟨*expression*⟩+)
  | (`if` ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩) | (`if` ⟨*expression*⟩ ⟨*expression*⟩)
  | (`set!` ⟨*variable*⟩ ⟨*expression*⟩)
  | (⟨*expression*⟩ ⟨*expression*⟩*)

⟨*formals*⟩ ::= ⟨*variable*⟩ | (⟨*variable*⟩*)

⟨*variable*⟩ ::= ⟨*letter*⟩ ⟨*subsequent*⟩* | + | - | * | / | =

⟨*subsequent*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩ | . | + | - | : | . . .

⟨*constant*⟩ ::= ⟨*boolean*⟩ | ⟨*character*⟩ | ⟨*string*⟩ | ⟨*digit*⟩+

⟨*boolean*⟩ ::= #t | #f

⟨*character*⟩ ::= #\⟨*any ascii character*⟩ | #\newline | #\space

⟨*string*⟩ ::= " ⟨*string character*⟩* "

⟨*string character*⟩ ::= \" | \\ | ⟨*any ascii character other than " or *⟩