# Microscheme: Functional programming for the Arduino

Ryan Suchocki
ryan@ryansuchocki.co.uk

Dr. Sara Kalvala
sara.kalvala@warwick.ac.uk

## Abstract

The challenges of implementing high level, functional languages on extremely resource-constrained platforms such as micro-controllers are abundant. We present Microscheme: a functional language for the Arduino micro-controller. The constraints of the Arduino platform are discussed, and the particular nuances of Microscheme are justified. Microscheme is novel among compact Scheme implementations in that it uses direct compilation rather than a *virtual machine* approach; and an unconventional compiler architecture in which the tree structure of the input program is determined during lexical analysis.

***Keywords*** Scheme, Arduino, Functional programming, Micro-controllers, Compilers

## 1. Introduction

Micro-controllers are becoming increasingly popular among hobbyists, driven by the availability of low-cost, USB-programmable micro-controller boards, and by interest in areas such as robotics and home automation. The Arduino project [9]—which provides a range of Atmel and ARM-based development boards—is notable for its active community and extensive wealth of supporting materials. The official Arduino IDE supports only C and C++, relying on the *avr-gcc* [10] compiler, and the *avr-libc* [11] and *wiring* [14] libraries. The Arduino community, however, consists largely of hobbyists and hackers, who have no overriding predisposition for working in C. Therefore, by providing a functional language targeting the Arduino hardware, there is an opportunity to introduce a new group of users to the world of functional programming.

We present Microscheme: a functional programming language for the *Arduino* micro-controller. Microscheme is predominantly a subset of R5RS Scheme [1]. Specifically, every syntactically valid Microscheme program is a syntactically valid Scheme program (up to primitive naming). Microscheme is tailored specifically to micro-controller applications, targeting the 8-bit ATMega chips used on most Arduino boards.

The targeted controllers are 8-bit, Harvard architecture machines (meaning code and data occupy physically separate storage areas), with between 2KB and 8KB of RAM, running at 16 MHz. Implementing high-level, dynamic, functional features on such a constrained platform is a significant challenge, and so the Microscheme language has been designed to accommodate realistic micro-controller programs, rather than achieving standard-compliance. Microscheme is currently lacking first-class continuations, garbage collection, and a comprehensive standard library. Also, its treatment of *closures* is slightly unsatisfactory. Nonetheless, it has reached a state where useful functional programs can be run natively and independently on real Arduino hardware, and is novel in that respect.

The contents of this short paper are focussed on the design of the language and runtime system, and the particular difficulties of the target platform. Far more detail can be found in the report [8] or via the project website `www.microscheme.org`. Following in the spirit of the work of Ghuloum [5], it is hoped that this project might help to de-mystify the world of functional language compilation. The compiler was constructed using the methodology set out in [5], by producing 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." [4] 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. 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 bolted on later.

There are a number of great materials on Scheme implementation in Scheme, and indeed there are many shortcuts to be enjoyed by writing a self-hosting Scheme processor. However, the implementation of Scheme is itself an exercise of great educational worth, even to those who are not proficient Scheme programmers. Therefore, we posit that there is a place in our field for Scheme implementations in languages other than Scheme. The Microscheme compiler is a recursive-descent, 4-pass cross-compiler, hand-written in pure C (99), directly generating AVR assembly code which

is in turn assembled by the 'avr-gcc' assembler, and uploaded using the 'avrdude' tool [12]. It is designed to run on any platform on which the avr-gcc/avrdude toolchain will run. (And therefore, any platform on which the official Arduino IDE will run.)

## 2. The Language

Microscheme is based around ten fundamental forms: constants, variable references, definitions, 'set!', 'begin', 'if', lambda expressions and procedure calls as well as the 'and' and 'or' control structures. 'Let' blocks are compiled as lambda expressions via the canonical equivalence. An 'include' form is provided to enable code re-use. The available primitive procedures include arithmetic operators, type predicates, vector and pair primitives, Arduino-specific IO and utility functions. Microscheme has comments and strings (compiled as vectors of chars). There is no provision for hygienic macros, of a 'foreign function interface'. Microscheme has no Symbol type and no 'quote' construct, but has a variadic `(list a b c ...)` primitive for building lists.

The following code listing shows a successful Microscheme program for driving a four-wheeled robot using stepper motors. The speed and direction of stepper motors are controlled by sending pulses to a number of 'coils'. Stepper motor control is achieved in Microscheme by defining a list of 4 integers for each motor, corresponding to digital I/O pins, to which pulses are sent in sequence to achieve rotation. The 'list.ms' Microscheme library provides common higher-order functions such as 'for-each' and 'reverse', which are used throughout this program.

The following program runs comfortably inside even the leanest Arduino device. But, due to its lack of garbage collection, some Microscheme programs will simply run out of available RAM before completion. In fact, it is possible to program in a heap-conservative style. By making sure that expressions causing new heap space to be allocated are placed outside of "loops" in the program, and using mutable data structures, programs can easily be written which do not run out of RAM. Even programs which run indefinitely can be designed to survive without garbage collection. This is unsatisfactory, however, because such a style is a departure from the established and intended character of Scheme.

Microscheme currently contains an unorthodox, last-resort primitive for memory recovery of the form `(free! ...)`. The expressions within the free! form are evaluated, but any heap space allocated by them is re-claimed afterwards. This feature is a bad idea for all sorts of reasons, and it is best characterised as "avoiding garbage collection by occasionally setting fire to the trash can". On the other hand, since Microscheme has no provision for multi-threading, it is possible to use it safely.

```
;; Include UNO pin number mappings
(include "libraries/io_uno.ms")
;; And common list processing functions
(include "libraries/list.ms")

; The left and right stepper motors are
    defined as lists of four I/O pins,
    set as outputs.
(define mleft (list 4 5 6 7))
(define mright (list 11 10 9 8))
(for-each output mleft)
(for-each output mright)

; This procedure takes two lists of pins,
    and sends pulses to them in sequence
(define (cycle2 m1 m2)
  (or (null? m1) (null? m2)
    (begin
      (high (car m1))
      (high (car m2))
      (pause 4)
      (low (car m1))
      (low (car m2))
      (cycle2 (cdr m1) (cdr m2)))))

; To move the robot forward X units,
    cycle both motors 32*X times.
(define (forward x)
  (for 1 (* x 32) (lambda (_)
    (cycle2 mleft mright))))

; To rotate the robot to the right, cycle
    the left motor fowards and the right
    motor in reverse. &vv.
(define (right x)
  (for 1 (div (* x 256) 45) (lambda (_)
    (cycle2 mleft (reverse mright)))))
(define (left x)
  (for 1 (div (* x 256) 45) (lambda (_)
    (cycle2 (reverse mleft) mright))))

; This procedure recursively defines one
    side of a Koch snowflake
(define (segment level)
  (if (zero? level)
    (forward 1)
    (begin
      (segment (- level 1))
      (left 60)
      (segment (- level 1))
      (right 120)
      (segment (- level 1))
      (left 60)
      (segment (- level 1)))))

; Drive the robot around one side of a
    third order Koch snowflake
(segment 3)
```

| 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

## 3. Runtime System Design

### 3.1 Type System

Scheme has a strong, dynamic, latent 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, which is referred to as *latent typing*. A consequence of dynamic typing that functions might be presented with values at runtime which are not of the expected type, and so runtime exceptions must be caught and reported. The built-in types supported by Microscheme are procedures (functions), integers[1], characters, Booleans, vectors, pairs and the empty list (a.k.a. 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 precisely in the spirit of Scheme minimalism.

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 or more is the modern expectation 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. (Those instructions are restricted in the sense that they can only be used on certain register pairs.)

The tagging scheme is biassed to give maximum space for the numeric type. The MSB (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 provides a compact representation for the most commonly used data types; as necessitated by the constraints of the Arduino platform. There is 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, type 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. The tagging scheme allows for 'number' type checking in even fewer operations, using a special instruction which tests a single bit within a register:

```
SBRC CRSh, 7
; skip next if bit 7 of CRS is clear
JMP error_notnum
; jump to the 'not a number' error
```

This is precisely how type checking is achieved on the arguments to arithmetic primitives.

[1] We choose to say 'integer' rather than 'fixnum', to maximise familiarity for all readers

### 3.2 The Stack

By eschewing first-class continuations, it is possible to implement Scheme using activation frames allocated in a last-in first-out data structure, as in a conventional call stack, rather than a heap-allocated continuation chain; thus exploiting the efficient built-in stack instructions with which most microprocessor architectures are equipped. Microscheme uses a call stack in this way.

Since a program without first-class continuations will always be evaluated by a predictable traversal of the nested constructs of the language, activation frames on the stack can safely be interleaved with other data, providing a pointer to the current activation frame (AFP = Activation Frame Pointer) is maintained. Therefore, the stack is also used freely within lower-level routines such as arithmetic primitives, so the stack is used at once as a *call stack* and an *evaluation stack*. Any Microscheme procedure takes its arguments from the stack, and stores a single result in a special register (CRS = Current ReSult).

### 3.3 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). Such differences are handled by model-specific assembly header files, included automatically at compile-time, containing definitions (such as RAM start/end addresses, dependant on the installed memory size) derived from the relevant technical data sheets. Different ATMega chips could easily be supported by writing equivalent definition files. Microscheme uses the first $2 \times n$ bytes of RAM 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*, in the familiar "heap upwards, stack downwards" arrangement.

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 the size of each particular object from the information contained within it. 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 1 shows the layout of the objects in detail. Note that the closure object contains a 'parent closure' reference. This forms a traversable chain of closures for each procedure object to its enclosing procedures, as required for lexical scoping.

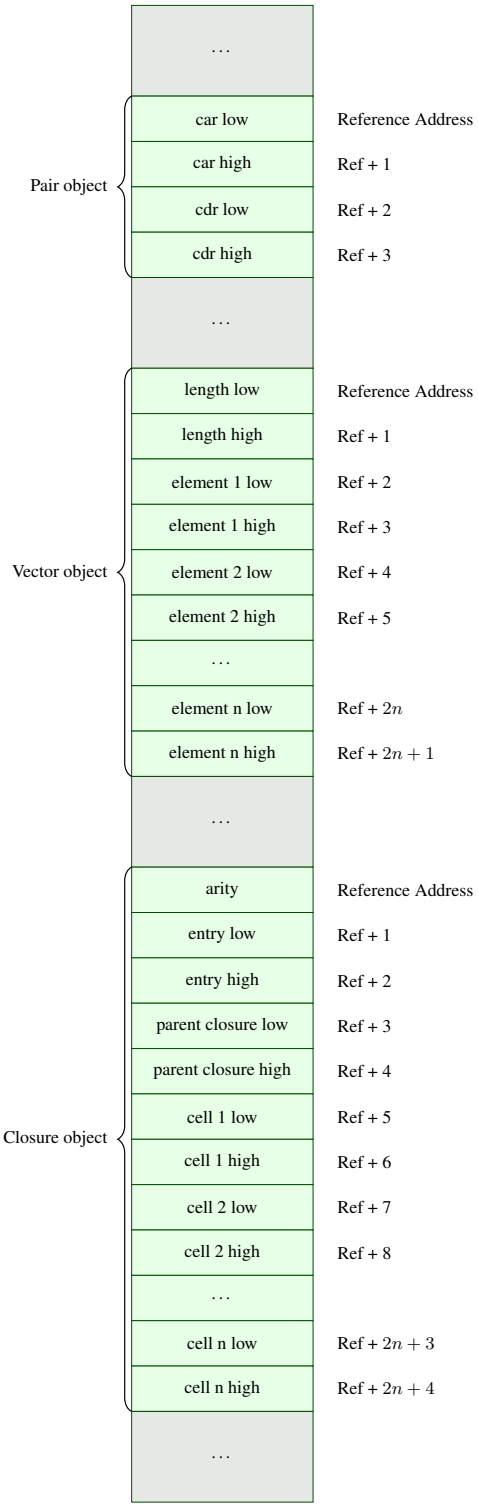


Figure 1: Heap-allocated object layout

For details of the stack layout, see section 3.5.

### 3.4 Register Allocation

The ATmega series of micro-controllers are purported to have 32 general-purpose registers [2]. In reality, most of these registers are highly restricted in function, and the nuances in the following allocation are crucial to the feasibility of Microscheme.

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

Table 2: Register Allocation Table

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 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 in section 3.1.

The first major challenge with these allocations is that each of the CCP, HFP, CRS and AFP will—at some point—hold memory addresses to be dereferenced. However, 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 is based on the plausible estimation that *closure lookup* is less frequent than *argument lookup*, *writing to the heap* 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 two simple 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 constant displacement, are only available on the final two register pairs. This instruction is crucial for working efficiently with heap-allocated objects. Figure 1 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 a single instruction. 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 registers (26:27).

Altogether, the register allocation is extremely dense, and deals with a large number of instruction set nuances to minimise the number of instructions generated. Some of the remaining registers (with restricted uses) are used to speed up certain low-level routines, and registers 6 thru 15 are available for use by future features such as a garbage collector. The register/instruction set 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 produces reasonably few instructions, while retaining a nucleus of functional features (including first class functions, higher order functions, lexical scope and closures) and is recognisably a subset of Scheme.

### 3.5 Calling Convention

Figures 2, 3 and 4 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 code for a procedure call is rather long, in comparison to a typical C function call, because a Scheme procedure call is a rather more sophisticated act. 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 practicable 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

```
PUSH AFPh                          ; Push the current AFP onto the stack
PUSH AFPl
LDI GP1, hi8(pm(proc_ret_χ))       ; Push the return address onto the stack
PUSH GP1
LDI GP1, lo8(pm(proc_ret_χ))
PUSH GP1
PUSH CCPh                          ; Push the current CCP onto the stack
PUSH CCPl
; Repeat for each argument:
  [code for argument i]            ; Evaluate each outgoing argument
  PUSH CRSl                        ; and push it onto the stack
  PUSH CRSh
[code for procedure expression]    ; Evaluate the procedure expression
MOV GP1, CRSh                      ; Mask out the lower 7 bits of the
ANDI GP1, 224                      ; upper byte of the result
LDI GP2, 192                       ; Check that we're left with the type
CPSE GP1, GP2                      ; tag for a procedure. Otherwise:
RJMP error_notproc                 ; jump to the 'not a procedure' error
ANDI CRSh, 31                      ; Mask out the data tag from the procedure
MOV CCPh, CRSh                     ; The remaining value is the address
MOV CCPl, CRSl                     ; of the incomming closure object.
LD GP1, Y; Y=CRS                   ; Fetch the expected number of arguments
LDI PCR, α                         ; from the closure object.
CPSE GP1, PCR                      ; Check against the given number. Otherwise:
RJMP error_numargs                 ; jump to the 'number of args' error
LDD AFPh, Y+1; Y=CRS               ; Load the procedure entry address
LDD AFPl, Y+2; Y=CRS               ; from the closure into register Z (AFP)
IJMP; context switch               ; Jump to that address.
proc_ret_χ:                        ; On return from the procedure:
POP AFPl                           ; restore the AFP.
POP AFPh
```

Figure 2: Procedure Call Routine (Caller Side)

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

```
proc_entry_χ:
  IN AFPl, SPl                     ; The new activation frame starts wherever
  IN AFPh, SPh                     ; the stack pointer is now
  [code for procedure body]
  ADIW AFPl, α                     ; Set the AFP just below the arguments
  OUT SPl, AFPl                    ; Set the stack pointer just below the arguments
  OUT SPh, AFPh
  POP CCPl                         ; Restore the old CCP from the stack
  POP CCPh
  POP AFPl                         ; Pop the return address, from the stack,
  POP AFPh                         ; into register Z (AFP)
  IJMP                             ; Jump to that address
```

Figure 3: Procedure Call Routine (Callee Side)

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

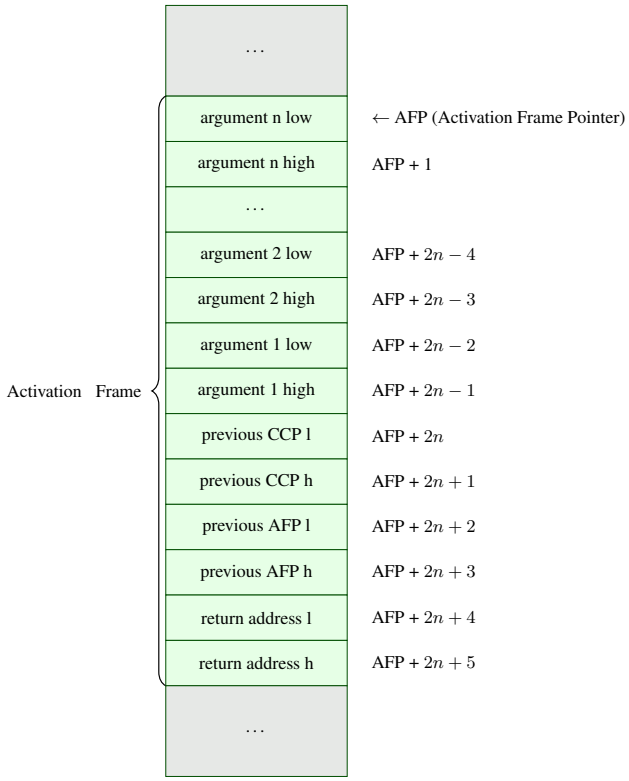


Figure 4: Activation Frame Layout

```
[code for argument i]
PUSH CRSl
PUSH CRSh
[code for procedure expression]
MOVW TCSl, CRSl ; Save proc
ADIW AFPl, [2*α]
OUT SPl, AFPl
OUT SPh, AFPh
SBIW AFPl, [α + β]
LDD GP1, Z+[β - i]
PUSH GP1
...
MOVW CRSl, TCSl ; Restore proc
LDI PCR, %i
MOV GP1, CRSh
ANDI GP1, 224
LDI GP2, 192
CPSE GP1, GP2
RJMP error_notproc
ANDI CRSh, 31
MOV CCPh, CRSh
MOV CCPl, CRSl
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
```

Repeat for each incoming argument

Shift all the incoming arguments down into the activation frame

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

Figure 5: Tail Call Routine (Caller Side)

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

for the procedure. These two conditions must be checked at runtime; costing in the order of 20 clock cycles per procedure call. The procedure call code has been designed so that a large segment of it (including those two checks) is constant across all procedure calls, and can be 'outlined' to a subroutine at the assembly level, saving hundreds of lines of assembly code (i.e. hundreds of bytes) in the generated executable.

The calling convention and activation frame are designed with tail recursion in mind, 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.

### 3.6 Tail Recursion

Scheme implementations are required [1] to be *properly tail recursive*. Tail-call-elimination is performed by the Microscheme compiler at the parsing stage. Procedure calls are eagerly transformed into tail-calls whenever they are in a *tail context.* 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 4) is designed with this operation in mind. The 'return' information for 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. Figure 5 shows the caller-side calling convention listing for a tail call.

### 3.7 Exception Handling

Due to Scheme's dynamic nature, runtime exceptions are unavoidable. As well as the procedure call exceptions described in section 3.5, there are type, bounds and arithmetic exceptions, and a 'custom' exception that may be raised programatically (for example, to constrain the domain of a function). The arduino is a standalone device, with no direct text-based output, and 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. (One flash for 'not a procedure', two flashes for 'wrong number of arguments', and so on.) 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.

### 3.8 Syntactic Sugar

The compiler supports 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 #\!))`. True vectors use approximately half the space of cons-based lists, 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 denser representation is preferable. 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, and makes the resultant Abstract Syntax Tree a node in the overall tree. 'Include' and commenting allow for the development of a suite of libraries, and a richer *numerical stack*.

## 4. Related Work

Other notable micro-controller-targeting Scheme implementations include PICOBIT, BIT and ARMPIT Scheme. PICOBIT [7] 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[2] are left out. While the aims of PICOBIT are closely aligned to this project, Microscheme will occupy quite a different Scheme subset.

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

Another impressive virtual-machine based implementation is BIT [3], which features real-time garbage collection, and has been ported to different micro-controllers.

ARMPIT Scheme [6] (targeting ARM micro-controllers) is a well-documented, open-source software project, with a large number of real-world working examples. Unusually, ARMPIT's designers intend that the micro-controller is used interactively, with a user issuing expressions and awaiting results via a serial connection. In other words, ARMPIT turns the micro-controller into a physical REPL (read-eval-print-loop) machine.

There are other projects which allow control of a micro-controller via a functional language running on some connected PC, such as via the Firmata library [13]. Though these tools present a way of 'controlling an Arduino from a functional language', they are clearly an altogether different kind of tool than a native compiler. Using such a library, one could never program an autonomous machine that strays away from its creator's workstation.

## 5. Conclusion

While Microscheme requires further work (notably: research into the feasibility of garbage collection and provision of a full suite of libraries) before it can be considered a complete programming tool, a significant amount of ground has been covered, and the compiler is in a usable state. By programming in a memory-conservative style (which, in any case, is an inevitability with this class of device) the adventurous Scheme programmer or Arduino hacker can very rapidly start writing programs to run natively on the Arduino, and such programs have proven successful, including:

- Robotic control programs, such as in section 2
- Programs recursively drawing fractals (~200 LOC) Demonstrating the correctness of the calling convention over thousands of recursive calls
- Library programs providing functions for digital I/O, long and fixed-point numeric types, standard higher-order list functions, ASCII manipulation and interfacing with LCD modules (~400 LOC)
- 'Countdown' program driving a multi-segment LED display (~200 LOC)
- Program for testing vintage SRAM chips (~300 LOC)
- Program for reading RPM signal from a car engine, driving a bar-graph LED display (~200 LOC)
- Various programs using an LCD module for text display

Moreover, the details presented here will hopefully find some educational use, or otherwise fill a gap in the literature surrounding Scheme implementation.

## References


[1] H. Abelson, R. K. Dybvig, C. T. Haynes, et al. Revised[5] report on the algorithmic language scheme. *Higher-order and symbolic computation*, 11(1):7–105, 1998.

[2] A. Corporation. *Atmel ATmega2560 Datasheet*, 2009.

[3] D. Dubé and M. Feeley. Bit: A very compact scheme system for microcontrollers. *Higher-order and symbolic computation*, 18(3-4):271–298, 2005.

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

[5] A. Ghuloum. An incremental approach to compiler construction. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR*. Citeseer, 2006.

[6] H. Montas. Armpit scheme, 2006. URL `http://armpit.sourceforge.net/`.

[7] V. St-Amour and M. Feeley. Picobit: a compact scheme system for microcontrollers. In *Implementation and Application of Functional Languages*, pages 1–17. Springer, 2011.

[8] R. Suchocki. A functional language and compiler for the Arduino micro-controller. Dissertation (u/g), University of Warwick. Available at `www.ryansuchocki.co.uk`.

[9] Various. Arduino website, 2014. URL `http://www.arduino.cc`.

[10] Various. Avr-gcc website, 2014. URL `http://gcc.gnu.org/wiki/avr-gcc`.

[11] Various. Avr-libc website, 2014. URL `http://www.nongnu.org/avr-libc`.

[12] Various. Avrdude website, 2014. URL `http://www.nongnu.org/avrdude`.

[13] Various. Firmata website, 2014. URL `http://www.firmata.org`.

[14] Various. Wiring website, 2014. URL `http://wiring.org.co`.