A Small Scheme VM, Compiler, and REPL in 4K

Samuel Yvon
Université de Montréal
Canada
samuel.yvon@umontreal.ca

Marc Feeley
Université de Montréal
Canada
feeley@iro.umontreal.ca

Abstract

Compact language implementations are increasingly popular for use in resource constrained environments. For embedded applications such as robotics and home automation, it is useful to support a Read-Eval-Print-Loop (REPL) so that a basic level of interactive development is possible directly on the device. Due to its minimalistic design, the Scheme language is particularly well suited for such applications and several implementations are available with different tradeoffs. In this paper we explain the design and implementation of Ribbit, a compact Scheme system that supports a REPL, is extensible and has a 4 KB executable code footprint.

CCS Concepts: • **Computer systems organization → Embedded software**.

Keywords: Virtual Machines, Read-Eval-Print-Loop, Compiler, Dynamic Languages, Scheme, Small Footprint

ACM Reference Format:
Samuel Yvon and Marc Feeley. 2021. A Small Scheme VM, Compiler, and REPL in 4K. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '21), October 19, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486606.3486783

1 Introduction

The Scheme programming language is a member of the Lisp family that is known for its short specification, its minimalistic design and the expressive power of the constructs it offers. Scheme is often cited as a good choice when a language implementation with a small footprint is needed either as an extension language embedded in a larger software or to stand on its own in a resource constrained microcontroller.

The use case which has motivated our work is code mobility where an executable program can be embedded in a document, email, or website. In that use case the size of the program must be small to minimize the transmission or loading time or to satisfy space constraints, such as the size of a disk boot sector, the URL length limit and the UDP packet size. On the other hand, the space used while the program is executing is of secondary importance.

Implementations of Scheme subsets typically have an executable machine code footprint in the 20-200 KB range [13]. The main factors impacting the footprint are the size of the builtin library and the support for interactive development using a REPL. PICOBIT [26] is currently one of the most compact Scheme system at 5-15 KB on PIC microcontrollers (depending on the size of the source program and the configured features), however it achieves this with a whole-program compilation that removes unused parts of the builtin library and so it does not offer a REPL. In this paper we explain the design of Ribbit, a compact and extensible implementation of Scheme that can be used with and without a REPL.

To make things concrete, we set a target goal of fitting a REPL and as big of a library as possible in a 4 KB executable footprint, less than PICOBIT, which lacks a REPL. For host languages like JavaScript, Python and Scheme that aren't typically Ahead-Of-Time (AOT) compiled to an executable form, this is the target size of the minified source code.

This objective is challenging as a Scheme system must support non-trivial features: closures, tail calls, first-class continuations and automatic memory management.

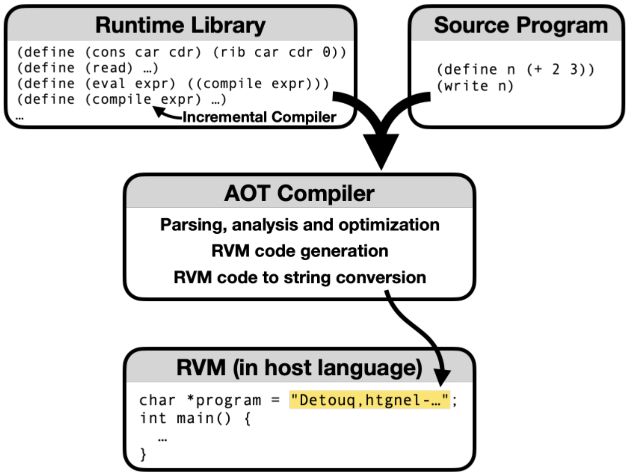
2 Design

In order to maximize the system's usefulness, the design must not be specialized to a given platform. It is desirable to write the system's source code using simple constructs that are portable to multiple languages. We use the term *portable* in its general sense since we are concerned about portability across languages. Thus the source code is portable if it can be translated manually with minimal effort (on the order of a day or two of work) to any host language that is appropriate for the end application (e.g. C in the case of microcontrollers and JavaScript for web apps). Moreover the system should be easy to modify and extend to support application specific features such as access to hardware ports, platform specific interfaces, and Scheme language extensions.

2.1 Overall Design

We achieve a compact executable footprint by a layering of languages. The compilation pipeline is shown in Figure 1. At the lowest level is a tiny VM, the Ribbit VM (RVM), a

VMIL '21, October 19, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9109-2/21/10…$15.00
https://doi.org/10.1145/3486606.3486783



```
(define (cons car cdr) (rib car cdr 0))
(define (read) …)
(define (eval expr) ((compile expr)))
(define (compile expr) …)
…
```

```
(define n (+ 2 3))
(write n)
```

```
char *program = "Detouq,htgnel-…";
int main() {
  …
}
```

Figure 1. The compilation pipeline: the AOT compiler produces a compact encoding of the program and runtime library which is then embedded in the RVM source code.

stack machine adapted to Scheme and implemented in a host language, currently one of C, JavaScript, Python, and Scheme. The RVM's core is roughly 150 lines of portable code.

The main parts of the REPL, including the parser, evaluator and runtime library, are written in Scheme and compiled to the RVM instruction set using a moderately optimizing AOT Scheme compiler. The optimizations most relevant to code size are constant propagation and dead code analysis that remove definitions of constants and library procedures unused by the program. To achieve this, both the main program and the library are read by the compiler and processed as a single unit. The compiler also creates a compact textual representation of the generated RVM instructions which is embedded as a string in the RVM source code and expanded at run time during the initialization phase using a short decompaction algorithm explained in Section 2.11. A key benefit of this language layering is that when implementing a given feature the compact representation of the RVM code is more compact than when using the host language.

A programmer may use the system through the REPL available in the runtime library or through the AOT compiler to benefit from its optimizations. Since the REPL is part of the runtime library, a compiled program may also make use of it at run time. This allows the programmer to adapt and extend the runtime library with application specific functionality. The VM implementation need only be extended with new primitives to access platform specific features.

Unlike several other Scheme systems offering a REPL, the evaluator that is used by the REPL and also the runtime library's `eval` procedure, is based on a non-optimizing *incremental compiler* that generates RVM code at run time. This offers better execution speed than the usual implementation of `eval` as an interpreter. The evaluator uses the runtime library's `compile` procedure to first compile the Scheme expression to a parameterless procedure and then calls this procedure to cause the expression's execution by the RVM:

```
(define (eval expr) ((compile expr)))
```

2.2 Memory Management and the RVM

When the host language does not manage memory using a garbage collector (GC), such as C and assembler, the RVM's code must include one. The choice of algorithm is important as it impacts the memory access speed, the space usage, and the executable footprint (due to the code required for the GC and memory accesses).

Four things need to be stored in memory: the Scheme objects, the Scheme global variables, the RVM's stack, and the instruction stream executed by the RVM. To avoid arbitrary limits, all of these are allocated in a garbage collected heap. This allows the depth of procedure call nesting and the size of the program's RVM code to be essentially unlimited (the only limit is the size of the heap). It also allows the space of unused code to be reclaimed at a fine granularity, for example when a procedure is redefined using the REPL the code of the old definition can be reclaimed if it is not currently executing or reachable from the stack or other live data structure.

A distinguising feature of the system is that all types of memory allocated structures are built out of fixed sized records called *ribs* (hence the name Ribbit). Immediate values are not memory allocated. The RVM currently only supports fixed precision integers (Scheme *fixnums*) as immediate values, but there is no fundamental reason the RVM could not be extended to support other types. Each *rib* contains three object references. The choice of using three fields for ribs offers a good compromise for representing Scheme objects, the stack and the instruction stream, as explained in the next sections. For now it suffices to understand that the stack and instruction stream use one field of the rib to explicitly chain the ribs. For its operation, the RVM maintains two variables: `stack`, which refers to the rib at the top-of-stack (TOS), and `pc`, which refers to the rib containing the RVM instruction being executed. These are the only garbage collection roots of the RVM (aside from the constants `#f` and `#t`), so any object not reachable from the stack or the instruction stream following the current point of execution can be reclaimed.

When the host language is C, an object reference is implemented with a machine word with the lower bit encoding the type: 1 when the object is an integer (the other bits represent the integer value), and 0 when the object is a rib (all the bits are the aligned address of the rib in memory). The fact that the memory manager only deals with a single type of memory allocated object means that a *header* field is not needed thus saving space. Moreover, a Cheney-style stop-and-copy GC [9] is used with the simplification that the scanning of the copied objects need not detect object boundaries because all fields of the copied objects must be updated.

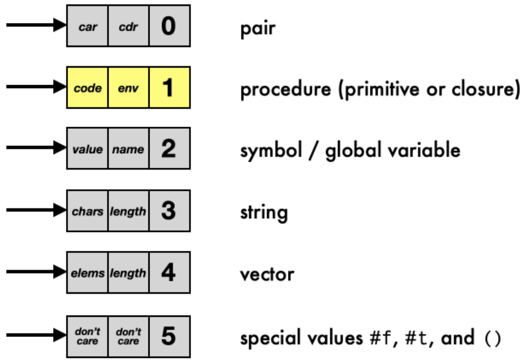


Figure 2. The representation of memory allocated Scheme objects using ribs. (Procedures are coloured yellow to help identify them in Figure 5)

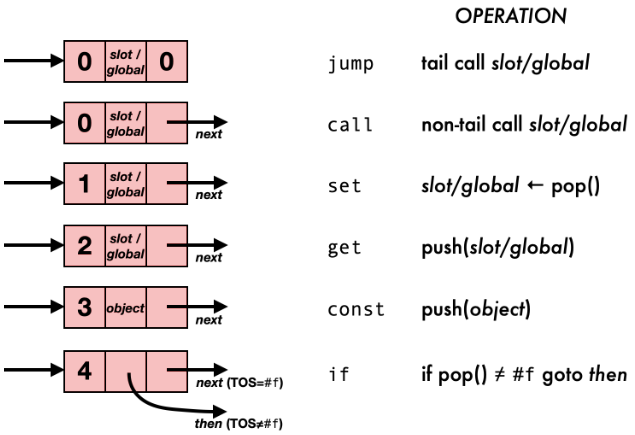


Figure 3. The representation of the six RVM instruction types using ribs. (Coloured red to help identify code in Figure 5)

When the host language is a dynamically typed language with garbage collected object arrays (e.g. JavaScript and Python), a rib is a three element host array and Scheme integers are mapped to host integers. The language's run time type tests can distinguish integers from ribs, for example in JavaScript `if(x instanceof Array)`... can be used, but also the more clever and compact `if(x.length)`... (when `x` is a rib `x.length` is 3 which acts like `true` and when `x` is an integer the result is `undefined` which acts like `false`).

2.3 Scheme Object Representation

For representing memory allocated Scheme objects, the last field of the rib contains a type indicator and the two other fields contain the rest of the object's attributes (see Figure 2). The type indicator could be any Scheme object as long as it is unique (according to `eqv?`), but currently a small Scheme integer is used for simplicity.

For pairs, the first two fields are the `car` and `cdr`.

The procedure representation contains the *code* and *env* attributes and is explained in greater detail in Section 2.6.

For symbols the second field is the symbol's name as a Scheme string and the first field is used to store the value of the Scheme global variable with that name. A separate table of global variables is not needed because a RVM instruction that accesses a global variable refers directly to the symbol. During program execution the RVM does not need to manage a symbol table. However, a symbol table is part of the implementation of the runtime library's `string->symbol` procedure, which itself is called by `read`. Consequently, a symbol table is managed by the runtime library only if the source program uses these library procedures or the REPL.

For Scheme strings and vectors, the second field is a Scheme integer indicating the length, and the first field is respectively the Scheme list of characters and elements. Note that a binary or ternary tree could have been used instead of a list, which would give logarithmic rather than linear time indexing. However the more complex indexing code would increase the runtime library's footprint.

The special values `#f`, `#t`, and `()` have the same type indicators and each is allocated once in the RVM initialization phase. This allows testing for these objects using a fast reference equality (`eqv?`). The first two fields are unused and are initialized to an implementation dependent convenient value (typically 0). For simplicity Scheme characters are not a distinct type and are represented as integers. It would be easy to add a distinct type at the cost of a larger runtime library footprint.

2.4 Instruction Graph Representation

Many VMs represent the instruction stream as a sequence of bytecodes in dedicated blocks of memory. This can complicate the garbage collection of unused code and the handling of return addresses that need to be distinguished from other values on the stack. In the RVM, instructions are stored in ribs with the last field referencing the rib of the next instruction (except the tail call instruction, `jump`, which has no next instruction). As a result, a return address is a reference to a rib and is handled by the GC like all other objects. The *instruction stream* is really an *instruction graph*. This graph representation could be used to express loops without explicit `jump` instructions, but currently the AOT compiler does not take advantage of this as explained in Section 2.11.

The RVM has only six instruction types. Each one corresponds to a basic construct of Scheme (except `lambda` which is handled specially), and with distinct instructions for tail and non-tail calls. As shown in Figure 3, the first field of the rib contains a small Scheme integer *opcode* indicating the instruction type (0=`jump` or `call`, 1=`set`, 2=`get`, 3=`const`, 4=`if`). For the `if` instruction, which pops the TOS and compares it to `#f`, the second field is a reference to the code

`rib`	0	*z*←pop();*y*←pop();*x*←pop();*r*←rib(*x*,*y*,*z*)
`id`	1	*x*←pop();*r*←*x*
`arg1`	2	*y*←pop();*x*←pop();*r*←*x*
`arg2`	3	*y*←pop();*x*←pop();*r*←*y*
`close`	4	*x*←pop();*r*←rib(*x*[0],`stack`,1)
`rib?`	5	*x*←pop();*r*←bool(*x* is a rib)
`field0`	6	*x*←pop();*r*←*x*[0]
`field1`	7	*x*←pop();*r*←*x*[1]
`field2`	8	*x*←pop();*r*←*x*[2]
`field0-set!`	9	*y*←pop();*x*←pop();*x*[0]←*y*;*r*←y
`field1-set!`	10	*y*←pop();*x*←pop();*x*[1]←*y*;*r*←y
`field2-set!`	11	*y*←pop();*x*←pop();*x*[2]←*y*;*r*←y
`eqv?`	12	*y*←pop();*x*←pop();*r*←bool(*x* is identical to *y*)
`<`	13	*y*←pop();*x*←pop();*r*←bool(*x*<*y*)
`+`	14	*y*←pop();*x*←pop();*r*←*x*+*y*
`-`	15	*y*←pop();*x*←pop();*r*←*x*-*y*
`*`	16	*y*←pop();*x*←pop();*r*←*x*`*`*y*
`quotient`	17	*y*←pop();*x*←pop();*r*←*x*//*y*
`getchar`	18	*x*←getchar();*r*←*x*
`putchar`	19	*x*←pop();putchar(x);*r*←*x*

bool(*x*) = `#t` if *x*, otherwise `#f`

Figure 4. The primitive procedures defined by the RVM. The result of the primitive is *r*.

jumped to when TOS≠`#f` and the last field is the reference to the code jumped to when TOS=`#f`. For the `const` instruction the second field is a reference to the object to push to the stack. For the `jump`, `call`, `set`, and `get` instructions the second field is the instruction's operand, which can be either a Scheme nonnegative integer or Scheme symbol. An integer indicates a stack slot relative to the top (i.e. 0 is the TOS) and a symbol indicates the global variable with that name.

2.5 Primitive Procedures

The data operations of the RVM are implemented using *primitive procedures*. Figure 4 gives the operations performed by the 20 available *primitives*. The Scheme object corresponding to a primitive is simply a rib whose first field (*code*) is a Scheme integer in the range 0 to 19, the second field (*env*) is unused, and the third field is 1, the type indicator for procedures. The RVM only initializes the `rib` global variable and it is up to the runtime library to create the Scheme objects corresponding to the other primitives it needs using calls to `rib`, for example the `id` primitive can be defined with:

```
(define id (rib 1 0 1)) ;; identity procedure
```

When a primitive is called through a `call` instruction it pops a certain number of arguments from the stack and pushes to the stack the operation's result. The number of arguments is fixed for a given primitive and the RVM does not check that there are enough arguments on the stack.

The same operations are executed when a primitive is called through a `jump` instruction, but before the result is pushed to the stack the RVM's `stack` and `pc` variables are updated according to the continuation in the current stack frame which contains the state of those variables when the `call` was executed (the details are given in Section 2.7). The `jump` instruction naturally corresponds to a tail call and all procedure activations end with a `jump`. A procedure which ends with a result that isn't a tail call in the source code can return the value by pushing it to the stack and then executing a `jump` to the `id` primitive.

2.6 Closures

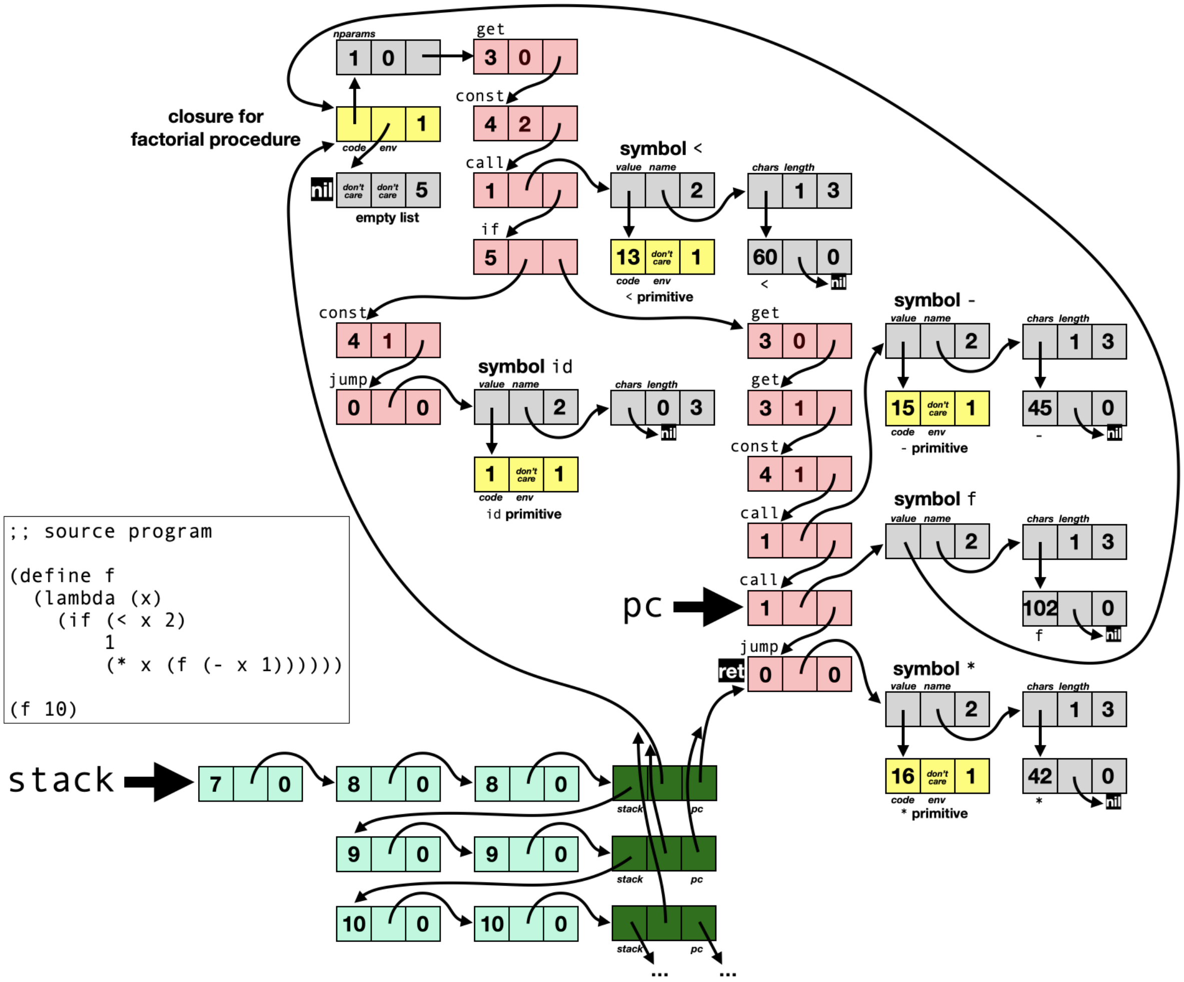
Closures are distinguished from primitive procedures by having a *code* field that refers to a rib. That rib's first field is a Scheme integer indicating the number of arguments expected by the procedure, and the third field is a reference to the rib that is the RVM instruction at the procedure entry point.

The *env* field of closures allows access to its free variables (disregarding global variables which are accessed through the corresponding symbol). When a procedure is defined in the global scope it has no free variables so the *env* field is not relevant. The AOT compiler handles this by creating a constant procedure with the empty list in the *env* field. For procedures defined in nested scopes, the AOT compiler uses the `close` primitive to construct the closure. The only argument is a constant procedure *template* whose *code* field will be copied to the *code* field of the allocated closure. The `close` primitive also saves the RVM's `stack` variable in the closure's *env* field. When the procedure is called, the *env* field will be used to access the variables accessible at the point where the closure was created. This is possible because a reference to the current procedure is put on the stack as part of the call protocol, as is explained in the next section.

2.7 The Call Protocol

A `call` to a non-primitive procedure does several things. First it allocates a continuation rib to be filled-in later. Then it pops from the stack as many values as the number of parameters expected by the procedure and accumulates the values in a list whose tail is the continuation rib. Consequently the list contains the values in the reverse order they were on the stack, which itself is the reverse of the order in the source code. The continuation rib is then initialized as follows: the RVM's `stack` variable is stored in the first field, a reference to the called procedure is put in the second field, and the *next* field of the `call` instruction is put in the third field. Finally the constructed list is assigned to the RVM's `stack` variable and the entry point rib of the procedure is assigned to the RVM's `pc` variable.

Now that control has been transferred to the called procedure all the information required to access local and free variables, and also to return control to the caller, is available through the RVM's `stack` variable. When a `jump` instruction is executed, the continuation rib is found by looping through the stack until a rib with a non-zero third field is



```
;; source program

(define f
  (lambda (x)
    (if (< x 2)
        1
        (* x (f (- x 1))))))

(f 10)
```

Figure 5. The state of the RVM during the execution of a call to the factorial procedure. The state shown corresponds to just before the recursive call (f (- x 1)) is executed when x is 8, which is after two recursive calls to f. The ribs coloured green are the stack frames. In dark green are the continuation ribs.

found. A local variable is accessed by using a get instruction whose integer index indicates a slot before the continuation rib. A free variable is also accessed by using a get instruction, but with an integer index that indicates a slot past the continuation rib. This works because the chaining of the stack ribs is in the second field, and in the second field of the continuation rib is a reference to the current procedure, and the second field of the procedure object is the *env* field containing a reference to the stack when the closure was created. The AOT compiler can determine the slot index by taking into account that each transition to a *parent* lexical scope requires adding 2 to the index (for the continuation rib and the current procedure closure that must be skipped).

A jump to a non-primitive procedure works in a similar way. The only difference is that the continuation rib's first and third fields are copied from the continuation rib of the current activation.

To help explain the call protocol we show in Figure 5 the state of the RVM during the execution of the recursive factorial procedure. The state shown is after having started two recursive calls and just before calling f in the third recursive call (f (- x 1)). The RVM's stack variable refers to the current *stack frame* which ends with a continuation rib that is linked to the caller's stack frame and *return address* (ret). In this example the factorial procedure is defined in the global scope, so the closure's *env* field is the empty list. If the closure had free variables it would refer to a rib in the stack frame that was current when the closure was created.

This example also shows other interesting features. Both stack slots and global variables store their values in the first field of a rib. This simplifies the implementation of the get and set instructions. The operand of those instructions always designates a rib (either a stack slot or a symbol) and then the first field is either read or written to. The factorial procedure returns the value 1 by pushing it to the stack with a `const` instruction and then executes a `jump` to the `id` primitive. Moreover, note that the symbol storing the `id` primitive has an empty name (thus saving space). This is possible thanks to an AOT compiler extension that allows renaming symbols (to an empty name in the case of the `id` symbol) and indicating which ones must be put in the runtime system's symbol table so that the programmer can use them at the REPL.

2.8 Variadic Procedures

The Scheme language supports creating variadic procedures that receive the rightmost arguments in the form of a list, the *rest parameter*. Some common standard procedures such as `+`, `append`, and `map`, are often called with exactly two arguments in practice but can take a different number of arguments when necessary.

Variadic procedures could be implemented in the RVM by passing the argument count on the stack and constructing the rest parameter on entry to the procedure. They could also be implemented without changing the RVM by adopting a calling convention that passes parameters using a Scheme list. Both of these approaches would negatively impact the system's footprint so variadic procedures are not currently supported by Ribbit.

2.9 Closure Space Safety

A consequence of saving the RVM's `stack` variable in the closure is that closures are not *safe-for-space* [21, 24] because they hold on to more data than strictly needed; not only lexical variables but all the call history. The AOT and incremental compilers could be modified to use a flat-closure representation that copies into the closure only its free variables. However this would add complexity to the RVM and/or adversely affect the RVM code compactness. The Scheme language specification does not require closures to be safe-for-space, so Ribbit currently avoids flat-closures in order to minimize the system's footprint.

2.10 First-Class Continuations

The implementation of first-class continuations can be done entirely in the runtime library as shown in Figure 6. The key idea is to obtain a reference to the stack through the `close` primitive and then to read and write the continuation rib.

When the `call/cc` procedure is called it is passed a single parameter, the *receiver* procedure. That procedure will be called with a single parameter *k*, a closure that represents the continuation of `call/cc`. The program will call *k* with a single parameter, the value that must be returned to the continuation of `call/cc`.

So both `call/cc` and *k* are procedures that take a single parameter. When executed at the very beginning of those procedure's bodies the expression `(field1 (field1 (close #f)))` will create a closure containing a reference to the stack which is extracted by a call to `field1` and then a second call to `field1` will skip one rib (containing the sole parameter) to get a reference to the continuation rib.

```
(define (call/cc receiver)
  ;; first get call/cc's continuation rib "c"
  (let ((c (field1 (field1 (close #f)))))
    (receiver
      (lambda (r)
        ;; get current continuation rib "c2"
        (let ((c2 (field1 (field1 (close #f)))))
          (field0-set! c2 (field0 c)) ;; set "stack" field
          (field2-set! c2 (field2 c)) ;; set "pc" field
          r))))) ;; return to continuation of call/cc
```

Figure 6. The definition of `call/cc` in the runtime library.

What needs to be done when *k* is called is to copy the content of the continuation rib of the `call/cc` activation (*c*) to the continuation rib of the current activation (*c2*). That way when *k* returns it will return to the continuation of `call/cc`. Note that the continuation rib *c2* was freshly created when *k* was called, whether with a `jump` or `call`, so it cannot be part of a previously captured continuation.

2.11 Compact Encoding of Instruction Graph

For portability reasons, in particular to support non AOT compiled host languages (e.g. JavaScript and Python), the RVM instruction graph generated by the Ribbit AOT compiler is embedded in the RVM source code in the form of a string. This string will be decoded during the initialization of the RVM to reconstruct the rib representation of the instruction graph. It is important for the decoding algorithm to be relatively short to not negatively impact the RVM's footprint.

Characters that must be escaped are not ideal to use in the embedded string because they need one extra byte to encode them compared to other characters and the source code size matters for non AOT compiled host languages. For this reason we restrict the characters to ASCII and avoid using `\` and `"`, and also the space character that might cause problems with some editors. This leaves 92 usable characters to encode the RVM instruction graph. Each character of the string is a *code* with 92 possible values.

The encoded instruction graph starts with the names of all the symbols and global variables used by the instruction graph separated by commas, in other words the program's *symbol table*. This allows referring to a symbol by its index in the symbol table. Some of the names may be blank if these symbols aren't meant to be accessible through `string->symbol`, `read` and the REPL.

The `jump`, `call`, `set`, and `get` instructions have one operand that can be a symbol, indicating the global variable of that name, or a nonnegative integer index of a stack slot.

To simplify the decoder a `const` instruction in the encoded instruction graph has a constant object that is a symbol, a nonnegative integer or closure (with empty *env* field). This

						closure	
	`jump`	`call`	`set`	`get`	`const`	`const`	`if`
code range	0..22	23..55	56..58	59..71	72..85	86..90	91
s	20	30	0	10	11	4	-

Figure 7. The range of codes used in the encoding of the RVM instruction graph. The number of short encodings is *s*.

allows the encoding to be like the `jump`, `call`, `set`, and `get` instructions when the operand is not a closure. Consequently, the AOT compiler must avoid generating `const` instructions with other types of constant objects. Instead the constant object is put in a fresh global variable at the start of the execution and a `get` instruction is used to access the constant (this is done with an appropriate computation, for example `(- 0 1)` to create the constant `-1`).

The encoding must contain the information on the type of operand and also a nonnegative integer *i* indicating the symbol's index in the symbol table or the slot's index on the stack. To avoid an arbitrary limit on the values of *i* that can be encoded, a variable length encoding is used. If *i* is less than 46 (half of 92) a single code containing *i* is used. For larger values of *i* multiple codes are used. If *k* codes are used to encode *i* the first $k-1$ codes contain a value between 46 and 91 and the *k*th code contains a value between 0 and 45. By computing each of these codes modulo 46, the *k* codes can be interpreted as a base-46 integer that encodes *i*. For example $k = 2$ codes can encode *i* up to 2115 and $k = 3$ codes can encode *i* up to 97335. The AOT compiler optimizes the compactness by sorting the symbol table to give small indexes to the symbols and global variables that are frequently referred to by the instruction graph. Slot indexes are naturally biased towards small values of *i*.

To achieve a high compactness each RVM instruction needs to be encoded by a small number of codes, ideally just one code for the most frequently occuring instructions. Each instruction type is allotted a certain range of values out of the 92 possible code values as shown in Figure 7. The ranges were determined experimentally to give a compact encoding for the REPL with runtime library.

Each range covers *r* code values and the first $s = r - 3$ values are for *short encodings*. For $i < s$ a short encoding gives in a single code the value of index *i*, which is a symbol index when the instruction is a `jump` or `call`, a slot index when the instruction is a `set` or `get`, and a literal integer when the instruction is a `const`. Of the remaining 3 code values one indicates that the index *i* is encoded in a variable length base-46 integer. The remaining 2 code values are for encoding an index *i* which is a slot index when the instruction is a `jump` or `call`, and a symbol index when the instruction is a `set`, `get` or `const`. The first code contributes a 0 or 1 as the first digit in a variable length base-46 integer.

For `const` instructions where the operand is a closure, the index *i* represents the number of arguments it expects. There are 4 short encodings for the frequent case of 0..3 arguments, otherwise a variable length encoding is used to encode *i*.

A single code value is needed to encode the `if` instruction.

The RVM supports instruction graphs that contain sharing and cycles. However a decoding algorithm that supports general graphs would be complex and adversely affect the footprint. For this reason the AOT compiler always creates an instruction graph that is a tree (no sharing and no cycles). To simplify the reconstruction of the rib representation of the instruction graph the string representation contains the encoded instructions in reverse order. As each instruction is decoded they are added to the front of the accumulated list of ribs. Consequently a `jump` instruction marks the beginning of a (reversed) branch of the tree, the `if` instruction marks the joining of 2 branches, and the closure `const` instruction marks the end of a branch of the tree. In other words, the decoded instructions contain enough information to reconstruct the tree without other structural markers. When a `jump` instruction is decoded the current instruction graph is pushed to a stack and a new instruction graph containing just the `jump` instruction is started. When an `if` or closure `const` instruction is decoded, the stack is popped to get the operand to put in the instruction.

The symbol table decoding and instruction graph decoding take roughly 25 lines of code each in the JavaScript version.

2.12 Runtime Library

The runtime library that is combined with the source program is a command-line argument of the AOT compiler. For convenience, two runtime libraries that are mostly subsets of the R4RS Scheme [1] standard procedures are supplied with the Ribbit system: **`min`** and **`max`**. The **`min`** runtime library has only basic procedures, but includes `eval`, `read`, `write`, `call/cc`, and enough to run all our benchmarks and a REPL. The **`max`** runtime library has all the features of the **`min`** runtime library and covers most of the other R4RS procedures. Both libraries restrict numbers to small integers and do not support a distinct character type or file system operations. The `eval` (and REPL) of the **`max`** runtime library supports more special forms including `define`, `set!`, `lambda`, `if`, `and`, `or`, `cond`, `let`, and `quote`, which are the same supported by the AOT compiler. A detailed list of features supported by these libraries is shown in Figure 8.

The AOT compiler supports an `(export` *symbol*...`)` form that allows the programmer to specify the set of symbols to be included in the run time symbol table, and consequently accessible through `string->symbol`, `read`, and the REPL. This affects the AOT compiler's procedure dependency graph analysis that determines the set of procedures that are part of the compiled program.

The "Ribbit REPL" is a program compiled by the AOT compiler with the **`min`** or **`max`** runtime library and containing nothing but a call to the `repl` procedure and an appropriate `export` form for all the procedures defined by that library.

The **min** library supports these predefined procedures (in bold are the special forms supported by eval):

*, +, -, <, =, cadddr, caddr, cadr, call/cc, car, cddr, cdr, cons, **define**, display, eof-object?, equal?, eqv?, eval, **if**, **lambda**, length, list->string, list->vector, list-ref, make-string, make-vector, newline, not, null?, pair?, peek-char, procedure?, **quote**, quotient, read, read-char, repl, **set!**, set-car!, set-cdr!, string->list, string->symbol, string-length, string-ref, string-set!, string?, symbol->string, symbol?, vector->list, vector-length, vector-ref, vector-set!, vector?, write

The **max** library adds support for:

<=, >, >=, abs, **and**, append, assoc, assq, assv, **begin**, boolean?, ca...r, cd...r, ceiling, **cond**, denominator, eq?, even?, expt, floor, for-each, gcd, integer?, lcm, **let**, map, max, member, memq, memv, min, modulo, negative?, number->string, numerator, odd?, **or**, positive?, remainder, reverse, round, string->number, string-append, string-copy, string-fill!, string<=?, string<?, string=?, string>=?, string>?, substring, truncate, vector-fill!, zero?

Figure 8. The features supported by the runtime libraries.

3 Evaluation

3.1 Footprint

The footprint of the system is the sum of the RVM's footprint and the length of the source program instruction graph's string representation produced by the AOT compiler. This later part is independent of the host language, and only depends on the source program and the runtime library.

For the Ribbit REPL the string representation of the instruction graph is 2068 and 4241 bytes, roughly 2 KB and 4 KB, respectively for the **min** and **max** runtime libraries. A detailed breakdown of this footprint is given in Figure 9. These numbers suggest the rule of thumb that each line of code (LOC) contributes 7 bytes to the total footprint.

The footprint when the RVM implementation is taken into account for the C, JavaScript, Python and Scheme host languages is given in Figure 10. The first column of the table is the footprint for an empty program (giving the "bare RVM" footprint). We can see that with the **min** runtime library both JavaScript and Python achieve our 4 KB target footprint and the others are close (in particular C with -Os at 4.7 KB).

For the C host language the RVM was compiled in two ways: with the -Os option (optimize for space) and the -O3 option (optimize for speed). In the case of C, the footprint refers to the executable file size on disk for a statically linked x86-32 ELF executable produced by the gcc compiler. Note that this RVM includes a garbage collector implementation and system calls for I/O to avoid linking with the C stdio library. Additional methods are employed to reduce code size, mainly stripping the ELF binary of unused sections and simplifying the produced executable. The result is an executable file with a footprint close to the incremental size cost of embedding the RVM in a complete C application.

For JavaScript, Python and Scheme, the footprint is the size of the source code after minification. The JavaScript version includes a simple DOM-based console emulation to support the REPL interaction in the web page. The full code is given in Figure 13 to give a visual understanding of the footprint breakdown and an appreciation for the small size of the RVM implementation.

Ribbit REPL +				
min lib		**max** lib		
bytes	LOC	bytes	LOC	Feature
456		993		symbol table
481	86	1040	149	eval with incr. compiler
323	58	409	69	read + string->number
258	43	360	50	write + number->string
550	162	1439	361	other procedures
2068	349	4241	629	string representation total

Figure 9. The breakdown of the footprint for various features of the runtime libraries. The lines of code are for the Scheme code stripped of comments and pretty-printed.

Bare	Ribbit REPL +		
RVM	**min** lib	**max** lib	Host
2.6 KB	4.7 KB	6.9 KB	C -Os
5.5 KB	7.6 KB	9.7 KB	C -O3
1.8 KB	3.8 KB	5.9 KB	JavaScript
2.0 KB	4.0 KB	6.1 KB	Python
2.8 KB	4.9 KB	7.0 KB	Scheme

Figure 10. The executable footprint for various host languages. The first column shows the footprint for an empty program.

3.2 Execution Speed

To evaluate the execution speed we compare Ribbit with Scheme systems implemented in C that aim to be embeddable on small devices or inside larger applications. For this comparison we use a suite of 10 standard Scheme benchmarks ranging from 15 to 280 LOC and with a mix of operations: integer arithmetic, list processing, recursion, and in one case, ctak, first-class continuations. The number of internal iterations in these benchmarks was adjusted to get an execution time with the C -O3 version of Ribbit that is on the order of 1-2 seconds and each benchmark is run 10 times to calculate the average and standard deviation. The tool hyperfine [22] is used perform time measurement. The test machine is a Intel i7-9750H (12) @ 4.5 GHz with 16 GB of RAM running Linux. All C compilations were done with gcc 10.3.0. The JavaScript Ribbit was run with Node.js v17.0.0 [12], the Python Ribbit was run with Pypy 7.3.5 [3], and the Scheme Ribbit was compiled with Gambit v4.9.3 [14].

A first use case of interest is the context where a REPL is needed. For this we have used the C version of Ribbit, compiled with -O3 and -Os, because they offer different space-time tradeoffs. A 240 KB heap is used in both cases. These are compared to other systems that provide a REPL: MiniScheme [20], TinyScheme [25], SCM [17], SIOD [8], and the Chicken interpreter [2].

The benchmark results are given in Figure 11. For each system the executable footprint is given and the execution time for each program. The execution time is absolute in the case of the C `-O3` version of Ribbit and for the other systems it is expressed relatively to that. A first observation is that the `-O3` version has a 60% larger footprint and it executes twice as fast as the `-Os` version. The footprint of Ribbit, even the `-O3` version, is considerably smaller than the other systems.

In terms of execution speed, the C `-O3` version of Ribbit is consistently faster than the other systems, with the exception of two programs where SCM is marginally faster. For several programs Ribbit is an order of magnitude faster than MiniScheme and TinyScheme. SCM has the next best performance, followed by SIOD, and then Chicken. Surprisingly most systems fail to run `ctak` which uses first-class continuations. Moreover, Chicken's performance on `ctak` is about 7.5× slower than Ribbit even though Chicken's design is based on a fast implementation of continuations. We find these results surprising because execution speed was not a specific design goal. We believe this good execution speed may be due to the RVM's extreme compactness that helps optimize the use of the processor's caches.

A second use case is when a REPL is not needed, for example when the development phase is finished and the program is put into production. For this situation we compare Ribbit's AOT compiler to the BIT [13] and PICOBIT [26] Scheme systems that only provide an AOT compilation mode. We also compare with the JavaScript, Python and Scheme versions of Ribbit. We are interested in both the total footprint and the execution time. The results are given in Figure 12. Again the execution time is absolute in the case of the C `-O3` version and is relative for the other systems.

Here too the C version of Ribbit compares favourably; several programs have an order of magnitude smaller footprint and execution time than BIT and PICOBIT. Unsurprisingly Ribbit's fastest version is the C version, but the other host languages also give reasonable performance, often faster than BIT and PICOBIT. With the C `-Os` version several benchmark programs have a footprint below 3 KB. Thanks to a compact RVM the JavaScript version gives the smallest footprints, in some cases below 2 KB.

4 Related Work and Conclusion

Language implementations fitting inside a single disk sector are not an uncommon programming challenge. Our work takes inspiration from *sectorlisp* [30], a 856 byte implementation of Lisp, which itself takes inspiration from *sectorforth* [4], an implementation of Forth that fits in a single 512 byte disk sector. Those systems use a direct (non-layered) implementation of the language itself and lack a garbage collector, while Ribbit implements a more featurefull language. Our work is closer in spirit to the Smalltalk-80 implementation, where we put emphasis on minimizing the footprint of the VM.

Using specialized machines to run languages in the Lisp family is also not a novel idea. Lisp machines use a specialized hardware platform to execute Lisp code efficiently. Similar to our work is the Scheme-79 chip [27, 29] which uses a linked representation of the code (called S-code). Porting our VM to a hardware description language is conceivable but would require adding other facilities to make it practical. The core logic would however stay the same. The PicoLisp [7] team claims [6] to have implemented a similar idea but few details can be found. Similar to Ribbit, PicoLisp implements all objects using only a two field pair structure. Ribbit's three field ribs are reminiscent of the Bigloo [23] *extended* pairs, which masquerade as pairs but have three fields.

Running Ribbit on *bare metal*, without a host operating system, is particularly attractive due to its small size. Mimosa [33] and Loko-Scheme [32] are examples of bare metal Schemes. The total system footprint could be reduced by implementing a low-level interface to the hardware in the VM and building the higher-level interface on top of that in the Scheme library which has a compact representation.

In recent years, there has been increased interest in providing high level languages to heavily constrained environments such as microcontrollers. The well known project MicroPython [11] provides a Python environment for popular microcontroller platforms. Espruino [10] is a similar effort to bring JavaScript to ARM based microcontrollers. Armpit Scheme [19] and uLisp [18] bring Scheme and Lisp to ARM based microcontrollers as well. Interestingly, all these projects enable the use of a REPL to facilitate development and provide an interactive interface to the devices. Darjeeling and uJ [5, 15] are Java implementations for microcontrollers which do not include a REPL, perhaps because until the addition of `jshell`, the REPL is not a traditional Java feature. OCaPIC [31], an OCaml port for PIC microcontrollers, and MicroScheme [28] are also without a REPL.

Having a language implementation running on top of a VM simplifies embedding into other projects, for example to add scripting to complex software, such as video games, or to provide a cross platform core for multiplatform software, such as mobile applications. Lua [16] is a typical example of such a language with a small footprint, around 250 KB. In this context, a much smaller footprint VM like the RVM with little platform dependencies is ideal to minimize the cost of embedding and the footprint overhead.

Acknowledgments

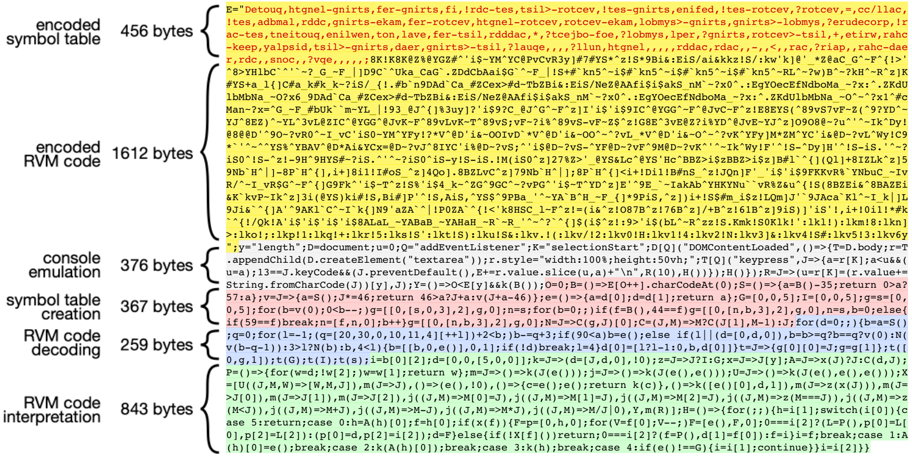
The authors would like to thank Nicolas Hurtubise for suggesting ideas to better compact the RVM code. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

	Ribbit REPL + **min** lib		MiniScheme	TinyScheme	SCM	SIOD	Chicken (csi)
	C -O3	C -Os	[20]	[25]	[17]	[8]	[2]
Footprint	7.6 KB	4.7 KB	57.6 KB	254.6 KB	266.3 KB	240.4 KB	4661.0 KB
ctak	1.18s ±1.2%	1.9× ±1.0%	FAIL	FAIL	FAIL	FAIL	7.5× ±2.8%
fib	1.32s ±1.5%	1.9× ±1.3%	13.1× ±1.2%	33.5× ±1.6%	1.6× ±3.4%	2.4× ±1.1%	4.8× ±4.3%
sum	1.05s ±1.3%	2.1× ±1.3%	17.6× ±1.9%	43.1× ±1.1%	1.9× ±2.3%	3.3× ±2.2%	5.8× ±2.3%
ack	1.47s ±2.6%	2.0× ±0.9%	28.8× ±0.9%	35.0× ±0.7%	2.3× ±2.8%	FAIL	4.7× ±8.8%
mazefun	1.20s ±2.4%	1.9× ±0.8%	FAIL	30.7× ±0.8%	1.2× ±2.0%	FAIL	3.3× ±1.4%
nqueens	0.91s ±1.0%	1.9× ±1.2%	12.3× ±1.5%	28.1× ±1.2%	1.4× ±2.9%	2.6× ±1.3%	3.9× ±5.1%
tak	0.94s ±1.4%	2.0× ±0.9%	9.3× ±2.1%	24.3× ±1.8%	1.2× ±2.7%	2.1× ±1.3%	3.1× ±4.6%
takl	1.18s ±1.6%	1.7× ±1.8%	8.4× ±1.3%	18.2× ±0.8%	0.9× ±2.9%	1.7× ±1.2%	2.3× ±4.0%
primes	1.03s ±1.6%	2.1× ±1.0%	13.9× ±1.5%	36.0× ±1.3%	1.7× ±2.4%	2.8× ±1.6%	4.8× ±1.8%
deriv	0.94s ±2.2%	2.0× ±1.0%	7.0× ±1.5%	16.2× ±1.3%	0.9× ±2.1%	1.5× ±1.0%	2.5× ±2.4%

Figure 11. Footprint and execution speed comparison when a REPL is used.

	Ribbit's AOT compiler + **min** lib					BIT	PICOBIT
	C -O3	C -Os	JavaScript	Python	Scheme	[13]	[26]
ctak	0.94s **5.7** KB	2.1× **2.8** KB	11.2× **2.0** KB	16.2× **2.2** KB	2.9× **3.1** KB	40.9× **39.5** KB	7.8× **113.4** KB
fib	1.06s **5.6** KB	2.0× **2.7** KB	9.5× **1.9** KB	5.5× **2.0** KB	3.2× **2.9** KB	22.2× **39.3** KB	25.5× **113.1** KB
sum	0.81s **5.6** KB	2.3× **2.7** KB	12.2× **1.9** KB	7.1× **2.0** KB	3.4× **2.9** KB	26.2× **39.3** KB	36.2× **113.1** KB
ack	1.11s **5.6** KB	2.1× **2.7** KB	9.2× **1.9** KB	6.7× **2.1** KB	2.6× **3.0** KB	51.5× **39.3** KB	FAIL
mazefun	1.04s **10.4** KB	2.0× **7.6** KB	7.7× **6.7** KB	8.3× **6.9** KB	2.5× **7.8** KB	FAIL	FAIL
nqueens	0.71s **5.7** KB	2.1× **2.9** KB	10.9× **2.1** KB	6.9× **2.2** KB	3.1× **3.1** KB	17.4× **39.5** KB	13.2× **113.4** KB
tak	0.74s **5.6** KB	2.1× **2.7** KB	10.5× **1.9** KB	7.0× **2.1** KB	3.0× **3.0** KB	18.5× **39.3** KB	10.5× **113.1** KB
takl	0.94s **5.7** KB	1.8× **2.8** KB	9.1× **2.0** KB	5.6× **2.1** KB	2.7× **3.1** KB	11.4× **39.5** KB	2.2× **113.2** KB
primes	0.77s **5.8** KB	2.3× **2.9** KB	11.5× **2.1** KB	7.2× **2.2** KB	3.8× **3.1** KB	28.6× **39.5** KB	22.1× **113.4** KB
deriv	0.72s **5.9** KB	2.1× **3.0** KB	10.1× **2.2** KB	6.9× **2.3** KB	2.9× **3.2** KB	FAIL	FAIL

Figure 12. Footprint and execution speed comparison when a Scheme AOT compiler is used.



```
E="Detouq,htgnel-gnirts,fer-gnirts,fi,!rdc-tes,tsil>-rotcev,!tes-gnirts,enifed,!tes-rotcev,?rotcev,=,cc/llac,
!tes,adbmal,rddc,gnirts-ekam,fer-rotcev,htgnel-rotcev,rotcev-ekam,lobmys>-gnirts,gnirts>-lobmys,?erudecorp,!r
ac-tes,tneitouq,enilwen,ton,lave,fer-tsil,rdddac,*,?tcejbo-foe,?lobmys,lper,?gnirts,rotcev>-tsil,+,etirw,rahc
-keep,yalpsid,tsil>-gnirts,daer,gnirts>-tsil,?lauqe,,,,?llun,htgnel,,,,,rddac,rdac,,-,,<,,rac,?riap,,rahc-dae
r,rdc,,snoc,,?vqe,,,,,;8K!K8K@Z%@YGZ#^'i$~YM^YC@PvCvR3y]#7#YS*^z!S*9Bi&:EiS/ai&kkz!S/:kw'k]@'_*Z@aC_G^~F^{!>'
^8>YHlbC`^'`~?_G_~F_|]D9C`^Uka_CaG`.ZDdCbAai$G`^~F_|!S+#`kn5^~i$#`kn5^~i$#`kn5^~i$#`kn5^~RL^~?w)B^~?kH^~R^z]K
#YS+a_l{]C#a_k#k_k~?iS/_{!.#b`n9DAd`Ca_#ZCex>#d~TbZBi&:EiS/NeZ@AAfi$i$akS_nM`~?x0^.:EgYOecEfNdboMa_~?x:^.ZKdU
lbMbNa_~O?x6_9DAd`Ca_#ZCex>#d~TbZBi&:EiS/NeZ@AAfi$i$akS_nM`~?x0^.:EgYOecEfNdboMa_~?x:^.ZKdUlbMbNa_~O^~^?x1^#c
Man~?x=^G_~F_#bUk``m~YL_|!93_@J^{]%3uy]?'i$9?C_@J^G^~F^z]I'i$'i$9IC^@YGG^~F^@JvC~F^z!E8EYS(^89vS7vF~Z(^9?YD^~
YJ^8EZ)^~YL^3vL@ZIC^@YGG^@JvK~F^89vLvK~T^89vS;vF~?i%^89vS-vF~Z$^z!G8E^3vE@Z?i%YD^@JvE~YJ^z]O9O8@~?u^'^~Ik^Dy!
@8@@D'^9O~?vR0^~I_vC'iS0~YM^YFy!?*V^@D'i&~OOIvD`*V^@D'i&~OO^~^?vL_*V^@D'i&~O^~^?vK^YFy]M*ZM^YC'i&@D~?vL^Wy!C9
*`'^~^^YS%^YBAV^@D*Ai&YCx=@D~?vJ^8IYC'i%@D~?vS;^'i$@D~?vS-^YF@D~?vF^9M@D~?vK^'^~Ik^Wy!F'^!S-^Dy]H'^!S-iS.'^~?
iS0^!S-^z!-9H^9HYS#~?iS.^'^~?iS0^iS-y!S-iS.!M(iS0^z]27%Z>'_@YS&Lc^@YS'Hc^BBZ>i$zBBZ>i$z]B#l`^{](Ql]+8IZLk^z]5
9Nb`H^|]-8P`H^{],i+]8i1!I#oS_^z]4Qo].8BZLvC^z]79Nb`H^|];8P`H^{]<i+!Di1!B#nS_^z!JQn]F'_'i$'i$9FKKvR%`YNbuC_~Iv
R/^~I_vR$G^~F^{]G9Fk^'i$~T^z!S%'i$4_k~^ZG^9GC^~?vPG^'i$~T^YD^z]E'^9E_`~IakAb^YHKYNu``vR%Z&u^{!S(8BZEi&^8BAZEi
&K`kvP~Ik^z]3i(@YS)ki#!S,Bi#]P'^!S,AiS,^YS$^9PBa_'^~YA`B^H_~F_{]*9PiS,^z])i+!S$#m_i$z!LQm]J'`9JAca`Kl^~I_k|]L
9Ji&`^{]A'^9AKl`C^~I`k{]N9'aZA`^|!P0ZA`^{!<'k8HSC_l~F^z!=(i&^z!O87B^z!76B^z]/+B^z!61B^z]9iS)]'iS'!,i+!0i1!*#k
`^{!/Qk!A'i$'i$'i$'i$8ALaL_~YABaB_~YAHaH_~R`~R_'^~^?`^{]$(i$^z!:9>'i$(bL^~R^zz!S.Kmk!S0Klk!':lkl!):lkm!8:lkn]
>:lko!;:lkp!1:lkq!+:lkr!5:lks!S':lkt!S):lku!S&:lkv.!(:lkv/!2:lkv0!H:lkv1!4:lkv2!N:lkv3]&:lkv4!S#:lkv5!3:lkv6y
";y="length";D=document;u=0;Q="addEventListener";K="selectionStart";D[Q]("DOMContentLoaded",()=>{T=D.body;r=T
.appendChild(D.createElement("textarea"));r.style="width:100%;height:50vh;";T[Q]("keypress",J=>{a=r[K];a<u&&(
u=a);13==J.keyCode&&(J.preventDefault(),E+=r.value.slice(u,a)+"\n",R(10),H())});H()});R=J=>(u=r[K]=(r.value+=
String.fromCharCode(J))[y],J);Y=()=>O<E[y]&&k(B());O=0;B=()=>E[O++].charCodeAt(0);S=()=>{a=B()-35;return 0>a?
57:a};v=J=>{a=S();J*=46;return 46>a?J+a:v(J+a-46)};e=()=>{a=d[0];d=d[1];return a};G=[0,0,5];I=[0,0,5];g=s=[0,
0,5];for(b=v(0);0<b--;)g=[[0,[s,0,3],2],g,0];n=s;for(b=0;;)if(f=B(),44==f)g=[[0,[n,b,3],2],g,0],n=s,b=0;else{
if(59==f)break;n=[f,n,0];b++}g=[[0,[n,b,3],2],g,0];N=J=>C(g,J)[0];C=(J,M)=>M?C(J[1],M-1):J;for(d=0;;){b=a=S()
;q=0;for(l=-1;(q=[20,30,0,10,11,4][++l])+2<b;)b-=q+3;if(90<a)b=e();else if(l||(d=[0,d,0]),b=b>=q?b==q?v(0):N(
v(b-q-1)):3>l?N(b):b,4<l){b=[[b,0,e()],0,1];if(!d)break;l=4}d[0]=[l?l-1:0,b,d[0]]}t=J=>{g[0][0]=J;g=g[1]};t([
0,g,1]);t(G);t(I);t(s);i=b[0][2];d=[0,0,[5,0,0]];k=J=>(d=[J,d,0],!0);z=J=>J?I:G;x=J=>J[y];A=J=>x(J)?J:C(d,J);
P=()=>{for(w=d;!w[2];)w=w[1];return w};m=J=>()=>k(J(e()));j=J=>()=>k(J(e(),e()));U=J=>()=>k(J(e(),e(),e()));X
=[U((J,M,W)=>[W,M,J]),m(J=>J),()=>(e(),!0),()=>{c=e();e();return k(c)},()=>k([e()[0],d,1]),m(J=>z(x(J))),m(J=
>J[0]),m(J=>J[1]),m(J=>J[2]),j((J,M)=>M[0]=J),j((J,M)=>M[1]=J),j((J,M)=>M[2]=J),j((J,M)=>z(M===J)),j((J,M)=>z
(M<J)),j((J,M)=>M+J),j((J,M)=>M-J),j((J,M)=>M*J),j((J,M)=>M/J|0),Y,m(R)];H=()=>{for(;;){h=i[1];switch(i[0]){c
ase 5:return;case 0:h=A(h)[0];f=h[0];if(x(f)){F=p=[0,h,0];for(V=f[0];V--;)F=[e(),F,0];0===i[2]?(L=P(),p[0]=L[
0],p[2]=L[2]):(p[0]=d,p[2]=i[2]);d=F}else{if(!X[f]())return;0===i[2]?(f=P(),d[1]=f[0]):f=i}i=f;break;case 1:A
(h)[0]=e();break;case 2:k(A(h)[0]);break;case 3:k(h);break;case 4:if(e()!==G){i=i[1];continue}}i=i[2]}}
```

Figure 13. The JavaScript implementation of the Ribbit REPL with **min** runtime library showing the amount of code for each major part of the system. About half of the code is for the string containing the representation of the instruction graph. Note that this code has gone through a minification process to make it more compact and it is all on a single line with no line breaks. This code can be executed by visiting the following link: https://udem-dlteam.github.io/ribbit/repl-min.html . The REPL with **max** runtime library can be executed by visiting the following link: https://udem-dlteam.github.io/ribbit/repl-max.html .

References

[1] Harold Abelson, R Kent Dybvig, Christopher Thomas Haynes, Guillermo Juan Rozas, Norman I Adams IV, Daniel Paul Friedman, Eugene Kohlbecker, Guy Lewis Steele Jr, David H Bartley, Robert Halstead, et al. 1991. Revised4 Report on The Algorithmic Language Scheme. *ACM SIGPLAN Lisp Pointers* 4, 3 (1991), 1–55.

[2] The Chicken Scheme Authors. 2021. Chicken Scheme. Website. (2021). https://www.call-cc.org/

[3] The PyPy Authors. 2021. PyPy. Website. (2021). https://www.pypy.org/

[4] Cesar Blum. 2020. sectorforth. Github Repository. (2020). https://github.com/cesarblum/sectorforth

[5] Niels Brouwers, Peter Corke, and Koen Langendoen. 2008. Darjeeling, a Java Compatible Virtual Machine for Microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion (Companion '08)*. Association for Computing Machinery, New York, NY, USA, 18–23. https://doi.org/10.1145/1462735.1462740

[6] Alexander Burger. 2014. Mailing List. (2014). https://www.mail-archive.com/picolisp@software-lab.de/msg04823.html

[7] Alexander Burger. 2021. Project Website. (2021). https://picolisp.com/wiki/?home

[8] George J. Carrette. 1997. SIOD v3.4. Website. (1997). https://www.nongnu.org/muesli/ulsiod.html

[9] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678. https://doi.org/10.1145/362790.362798

[10] The Espruino Developers. 2021. Javascript for Microcontrollers. (2021). https://www.espruino.com/

[11] The Micropython Developers. 2021. Python for Microcontrollers. (2021). https://micropython.org/

[12] The NodeJs Developers. 2021. NodeJS. Website. (2021). https://nodejs.org/en/

[13] Danny Dubé and Marc Feeley. 2005. BIT: A Very Compact Scheme System for Microcontrollers. *Higher-order and symbolic computation* 18, 3-4 (2005), 271–298.

[14] Marc Feeley. 2021. Gambit Scheme. Github Repository. (2021). https://github.com/gambit/gambit

[15] Dmitry Grinberg. 2013. uJVM. Website. (2013). https://dmitry.gr/index.php?r=05.Projects&proj=12.%20uJ%20-%20a%20micro%20JVM

[16] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The Evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2–1.

[17] Aubrey Jaffer. 2021. The SCM Implementation of Scheme. Website. (2021). https://people.csail.mit.edu/jaffer/SCM.html

[18] David Johnson-Davies. 2021. uLisp. Project Website. (2021). http://www.ulisp.com

[19] Hubert Montas. 2006. Armpit Scheme. SourceForge Repository. (2006). http://armpit.sourceforge.net/

[20] Atsushi Moriwaki. 1989. A Mini-Scheme Implementation. Website. (1989). ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/minischeme.tar.gz

[21] Zoe Paraskevopoulou and Andrew W Appel. 2019. Closure Conversion Is Safe for Space. *Proceedings of the ACM on Programming Languages* 3, ICFP, 1–29.

[22] David Peter. 2021. Hyperfine. Github Repository. (2021). https://github.com/sharkdp/hyperfine

[23] Manuel Serrano. 2021. Bigloo: a Practical Scheme Compiler. Website. (2021). https://www-sop.inria.fr/mimosa/fp/Bigloo/index.html

[24] Zhong Shao and Andrew W Appel. 2000. Efficient and Safe-for-Space Closure Conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1, 129–161.

[25] Dimitrios Souflis, Kevin Cozens, and Jonathan Shapiro. 2021. TinyScheme. SourceForge Repository. (2021). http://tinyscheme.sourceforge.net/home.html

[26] Vincent St-Amour and Marc Feeley. 2009. PICOBIT: a Compact Scheme System for Microcontrollers. In *International Symposium on Implementation and Application of Functional Languages*. Springer, 1–17.

[27] Guy Lewis Steele Jr and Gerald Jay Sussman. 1979. Design of Lisp-Based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode. (1979).

[28] Ryan Suchocki and Sara Kalvala. 2014. Microscheme: Functional programming for the Arduino. In *Scheme and Functional Programming Workshop, Washington, DC*. 21–29.

[29] Gerald J Sussman, Jack Holloway, GL Steele, and Alan Bell. 1981. SCHEME-79—LISP on a chip. *Computer* 14, 7 (1981), 10–21.

[30] Justine Tunney. 2020. SectorLisp. Github Repository. (2020). https://github.com/jart/sectorlisp

[31] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. 2015. Programming Microcontrollers in OCaml: The OCaPIC Project. In *Practical Aspects of Declarative Languages*, Enrico Pontelli and Tran Cao Son (Eds.). Springer International Publishing, Cham, 132–148.

[32] Göran Weinholt. 2021. Loko Scheme. GitLab Repository. (2021). https://scheme.fail/

[33] Samuel Yvon and Marc Feeley. 2021. Running Scheme On Bare Metal (Experience Report). In *Proceedings of the 2020 Scheme and Functional Programming Workshop*. 51–65.