A R4RS Compliant REPL in 7 KB

By
Léonard Oest O'Leary
Mathis Laroche
Marc Feeley

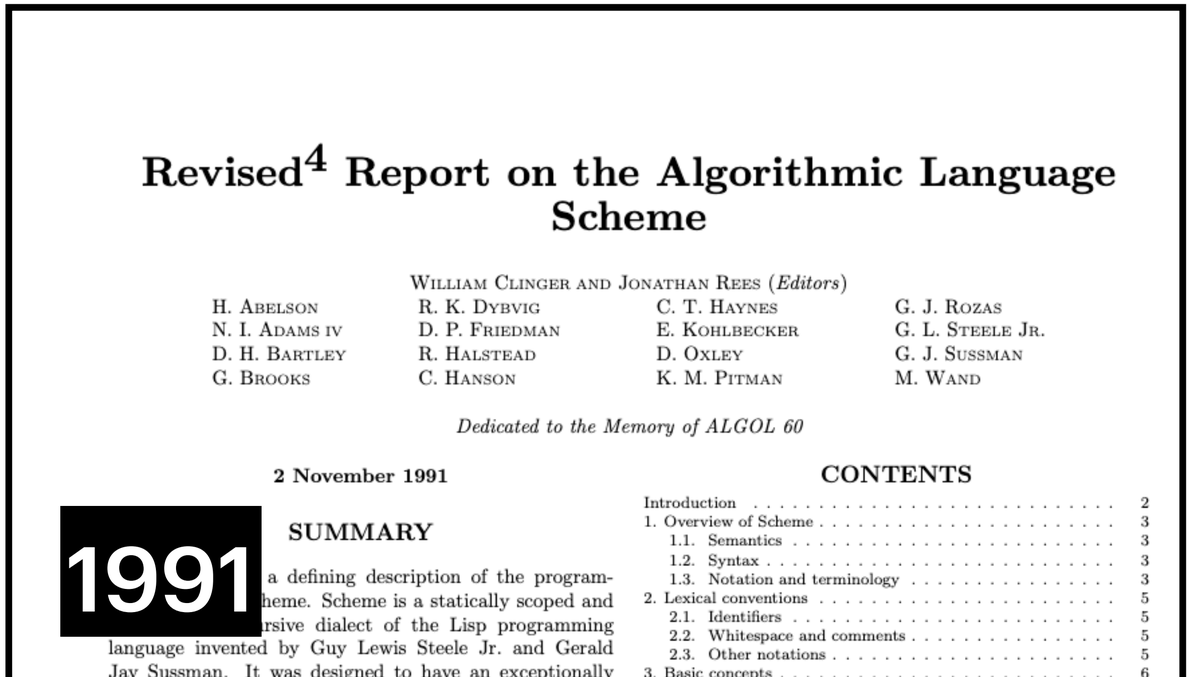
Trends in Functional Programming
12th of January 2024

Université de Montréal

What is R4RS ?

And how big is it ?

- 5th Scheme standard
- First "modern Scheme"



Revised^4 Report on the Algorithmic Language Scheme

William Clinger and Jonathan Rees (*Editors*)

H. Abelson, N. I. Adams iv, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele Jr., G. J. Sussman, M. Wand

Dedicated to the Memory of ALGOL 60

2 November 1991

1991

SUMMARY

a defining description of the program-
heme. Scheme is a statically scoped and
rsive dialect of the Lisp programming
language invented by Guy Lewis Steele Jr. and Gerald
Jay Sussman. It was designed to have an exceptionally

CONTENTS

- 164 essential procedures
- 18 essential syntaxes
- file I/O procedures
- rest parameters
- Quasiquote, unquote unquote-splicing
- call/cc, load (eval)

How small is 7KB ?

A R4RS Compliant REPL in 7 KB

LÉONARD OEST O'LEARY, Université de Montréal, Canada
MATHIS LAROCHE, Université de Montréal, Canada
MARC FEELEY, Université de Montréal, Canada

The Ribbit system is a compact Scheme implementation running on the Ribbit Virtual Machine (RVM) that has been ported to a dozen host languages. It supports a simple Foreign Function Interface (FFI) allowing extensions to the RVM directly from the program's source code. We have extended the system to offer conformance to the R4RS standard while staying as compact as possible. This leads to a R4RS compliant REPL that fits in an 7 KB Linux executable. This paper explains the various issues encountered and our solutions to make, arguably, the smallest R4RS conformant Scheme implementation of all time.

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

Additional Key Words and Phrases: Virtual Machines, Compiler, Dynamic Languages, Scheme, Compactness

ACM Reference Format:
Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. 1, 1 (September 2023), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Ribbit Scheme system [9, 14] is portable, extensible, and compact. It is based on a Virtual Machine (VM) that is portable to a dozen host languages including: JavaScript, C, Assembly (x86), Shell, Haskell, and Prolog. It is extensible, enabling programmers to add their own host-level primitives in Scheme code or using annotations within the VM's code. It is compact by design, with an extremely simple VM and with an AOT compiler that removes dead code from the program, library, and VM itself.

This paper explains how Ribbit has been extended to maintain a small size while adding conformance to the R4RS specification. The main enhancements to the previous Ribbit system are:

(1) Support for variadic procedures and rest parameters.
(2) Implementation of all required file I/O procedures.
(3) Various measures to better compact the generated code, including a new approach for encoding programs and a compact implementation of the standard library.

Authors' addresses: Léonard Oest O'Leary, Université de Montréal, Canada, leonard.oest.oleary@umontreal.ca; Mathis Laroche, Université de Montréal, Canada, mathis.laroche@umontreal.ca; Marc Feeley, Université de Montréal, Canada, feeley@iro.umontreal.ca.

These changes have allowed us to fit an interactive REPL fully conforming to R4RS in a 7 KB Linux executable program with no external dependencies. We chose to support the R4RS Scheme standard because it combines practicality and small size. Also, there is lots of existing code that can run in an R4RS system including most of the SLIB Portable Scheme Library [8]. Subsequent standards added features that would increase the size of the system substantially: hygienic macros and multiple values are required starting at R5RS, and libraries and Unicode support are required starting at R6RS. A more detailed reasoning for our choice can be found in Section 4.

The paper is organized as follows: Section 2 provides an overview of the Ribbit system. Section 3 explains the encoding optimizations. Section 4 describes the implementation of the R4RS library to achieve compactness and portability across host languages. Section 5 describes the x86 assembly host which is our most compact and fast implementation of the RVM. Section 6 evaluates the effectiveness of our approach through benchmarks that measure the space and execution time using multiple compilation settings. Finally, the paper concludes with related work.

2 RIBBIT

Ribbit has three main components: the Ribbit VM (RVM) implemented in multiple host languages, the Ribbit Scheme Compiler (RSC), and the standard library. RSC, an Ahead Of Time (AOT) compiler, combines the source program with the standard library to generate a standalone specialized RVM in the chosen host language. Every RVM source program contains annotations that attach meaning to portions of its code. This lets the compiler selectively include, exclude or adapt sections of the code leading to a RVM uniquely tailored to the program.

The compiler will embed in the RVM source code the RVM code it has generated for the program in an encoded form: the Ribbit Intermediate Byte Notation (RIBN), pronounced *ribbon*. The RIBN has two parts: the symbol table and the encoded sequence of RVM instructions. The symbol table is represented as a list, where the position of a symbol in the list is its index. When encoding the symbol table inside the RIBN, the list, as well as the string representation of symbols, are encoded in reverse order for decoding simplicity. The encoded program uses a specialized encoding discussed in Section 3.

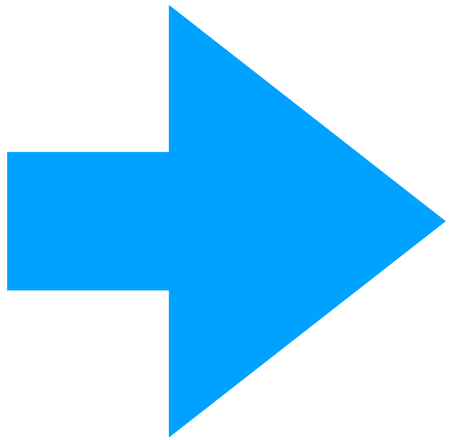
2.1 Ribbit VM

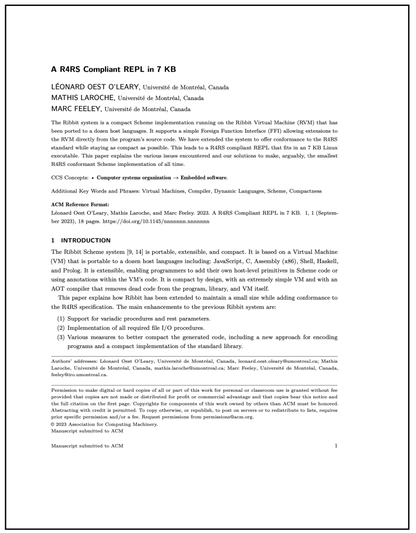
The Ribbit VM was designed with simplicity in mind, to minimize the VM's code size and allow porting it to new host languages with low effort. It is a stack machine with 6 available instructions loosely corresponding to the fundamental Scheme constructs: `jump` (tail call), `call` (non-tail call), `set` (writing a variable), `get` (reading a variable), `const` (literal data), and `if` (conditional execution).

To simplify memory management, the only data that is managed by the RVM is the *rib*: a three field structure where each field can be an integer or a reference to a rib. The code executed by the RVM, the Scheme data, and the stack are all represented using ribs. When a rib represents Scheme data, the last field is an integer indicating the type: 0 for pair, 1 for procedure, 2 for symbol, 3 for string, etc. In the rest of the paper we will use the notation `[a,b,c]` to mean a rib with the fields `a`, `b`, and `c`. This also happens to be the implementation of ribs in the Python and JavaScript RVMs, among others. As an example, the Scheme improper list `(1 2 . 3)` is represented using two ribs: `[1, [2, 3, 0], 0]`.

How small is 7KB ?

- 164 essential procedures
- 18 essential syntaxes
- file I/O procedures
- rest parameters
- Quasiquote, unquote unquote-splicing
- call/cc, load (eval)





A R4RS Compliant REPL in 7 KB

LÉONARD OEST O'LEARY, Université de Montréal, Canada
MATHIS LAROCHE, Université de Montréal, Canada
MARC FEELEY, Université de Montréal, Canada

The Ribbit system is a compact Scheme implementation running on the Ribbit Virtual Machine (RVM) that has been ported to a dozen host languages. It supports a simple Foreign Function Interface (FFI) allowing extensions to the RVM directly from the program's source code. We have extended the system to offer conformance to the R4RS standard while staying as compact as possible. This leads to a R4RS compliant REPL that fits in an 7 KB Linux executable. This paper explains the various issues encountered and our solutions to make, arguably, the smallest R4RS conformant Scheme implementation of all time.

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

Additional Key Words and Phrases: Virtual Machines, Compiler, Dynamic Languages, Scheme, Compactness

ACM Reference Format:
Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. 1, 1 (September 2021), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Ribbit Scheme system [9, 14] is portable, extensible, and compact. It is based on a Virtual Machine (VM) that is portable to a dozen host languages including: JavaScript, C, Assembly (x86), Shell, Haskell, and Prolog. It is extensible, enabling programmers to add their own host-level primitives in Scheme code or using annotations within the VM's code. It is compact by design, with an extremely simple VM and with an AOT compiler that removes dead code from the program, library, and VM itself.

This paper explains how Ribbit has been extended to maintain a small size while adding conformance to the R4RS specification. The main enhancements to the previous Ribbit system are:

(1) Support for variadic procedures and rest parameters.
(2) Implementation of all required file I/O procedures.
(3) Various measures to better compact the generated code, including a new approach for encoding programs and a compact implementation of the standard library.

Authors' addresses: Léonard Oest O'Leary, Université de Montréal, Canada, leonard.oest.oleary@umontreal.ca; Mathis Laroche, Université de Montréal, Canada, mathis.laroche@umontreal.ca; Marc Feeley, Université de Montréal, Canada, feeley@iro.umontreal.ca.

Manuscript submitted to ACM

These changes have allowed us to fit an interactive REPL fully conforming to R4RS in a 7 KB Linux executable program with no external dependencies. We chose to support the R4RS Scheme standard because it combines practicality and small size. Also, there is lots of existing code that can run in an R4RS system including most of the SLIB Portable Scheme Library [8]. Subsequent standards added features that would increase the size of the system substantially: hygienic macros and multiple values are required starting at R5RS, and libraries and Unicode support are required starting at R6RS. A more detailed reasoning for our choice can be found in Section 4.

The paper is organized as follows: Section 2 provides an overview of the Ribbit system. Section 3 explains the encoding optimizations. Section 4 describes the implementation of the R4RS library to achieve compactness and portability across host languages. Section 5 describes the x86 assembly host which is our most compact and fast implementation of the RVM. Section 6 evaluates the effectiveness of our approach through benchmarks that measure the space and execution time using multiple compilation settings. Finally, the paper concludes with related work.

2 RIBBIT

Ribbit has three main components: the Ribbit VM (RVM) implemented in multiple host languages, the Ribbit Scheme Compiler (RSC), and the standard library. RSC, an Ahead Of Time (AOT) compiler, combines the source program with the standard library to generate a standalone specialized RVM in the chosen host language. Every RVM source program contains annotations that attach meaning to portions of its code. This lets the compiler selectively include, exclude or adapt sections of the code leading to a RVM uniquely tailored to the program.

The compiler will embed in the RVM source code the RVM code it has generated for the program in an encoded form: the Ribbit Intermediate Byte Notation (RIBN), pronounced ribbon. The RIBN has two parts: the symbol table and the encoded sequence of RVM instructions. The symbol table is represented as a list, where the position of a symbol in the list is its index. When encoding the symbol table inside the RIBN, the list, as well as the string representation of symbols, are encoded in reverse order for decoding simplicity. The encoded program uses a specialized encoding discussed in Section 3.

2.1 Ribbit VM

The Ribbit VM was designed with simplicity in mind, to minimize the VM's code size and allow porting it to new host languages with low effort. It is a stack machine with 6 available instructions loosely corresponding to the fundamental Scheme constructs: `jump` (tail call), `call` (non-tail call), `set` (writing a variable), `get` (reading a variable), `const` (literal data), and `if` (conditional execution).

To simplify memory management, the only data that is managed by the RVM is the *rib*: a three field structure where each field can be an integer or a reference to a rib. The code executed by the RVM, the Scheme data, and the stack are all represented using ribs. When a rib represents Scheme data, the last field is an integer indicating the type: 0 for pair, 1 for procedure, 2 for symbol, 3 for string, etc. In the rest of the paper we will use the notation `[a,b,c]` to mean a rib with the fields `a`, `b`, and `c`. This also happens to be the implementation of ribs in the Python and JavaScript RVMs, among others. As an example, the Scheme improper list `(1 2 . 3)` is represented using two ribs: `[1, [2, 3, 0], 0]`.

Presentation **Outline**

1 What is Ribbit ?

A Scheme implementation based on a custom VM that is:

- **Compact**
- **Extensible**
- **Portable**

2 How we enhanced Ribbit to fit a REPL in 7Kb ?

- Fully R4RS compliant
- Novel encoding
- Compression
- x86 Host

Origins of Ribbit

- Original motivation: fascinating sectorlisp project
  - LISP interpreter + GC that fits in a 512 byte bootsector!

https://github.com/jart/sectorlisp

More a curiosity than practical because it is not portable or extensible

Ribbit is Portable

- Portability of a VM can be defined as:

The effort required by a programmer to get the VM working in a different environment

- **Environment** usually means a combination of operating system and hardware architecture

Our approach

Implement the Ribbit VM (RVM) in several host languages and make it easy to port the VM to a new one (2-3 days)

Ribbit is Portable

rvm.js

```
rvm_code = "…";
…
while (true) {
  …
}
```

rvm.asm

```
rvm_code: "…",0
…
loop:
            …
            jmp
loop
```

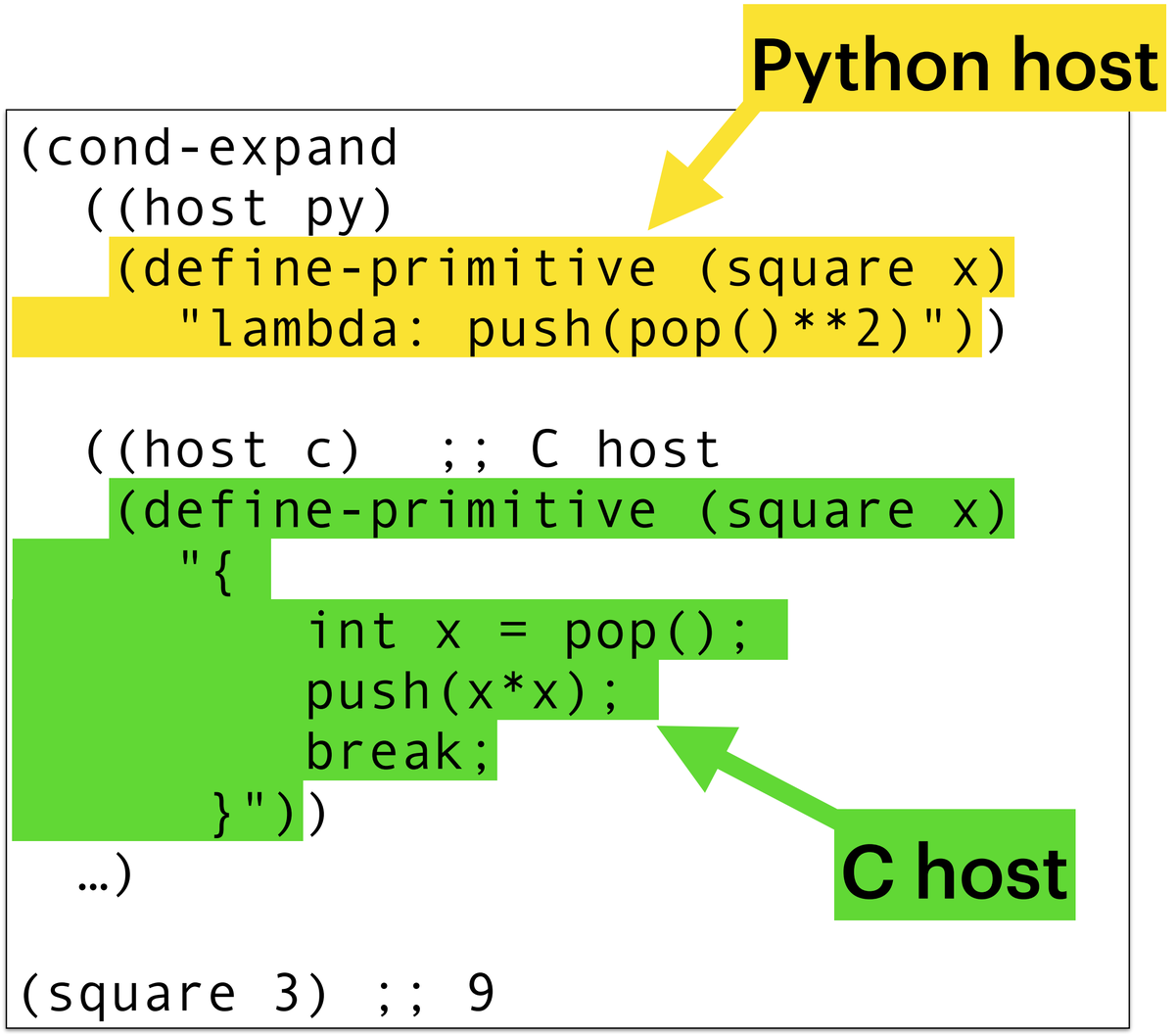
rvm.c

```
char *rvm_code = "…";
…
while (1) {
  …
}
```

rvm.clj … **rvm.lisp** … **rvm.hs** … **rvm.jl** … **rvm.lua** … **rvm.ml** … **rvm.scala** … **rvm.scm** … **rvm.zig** … **rvm.sh** … **rvm.py** …

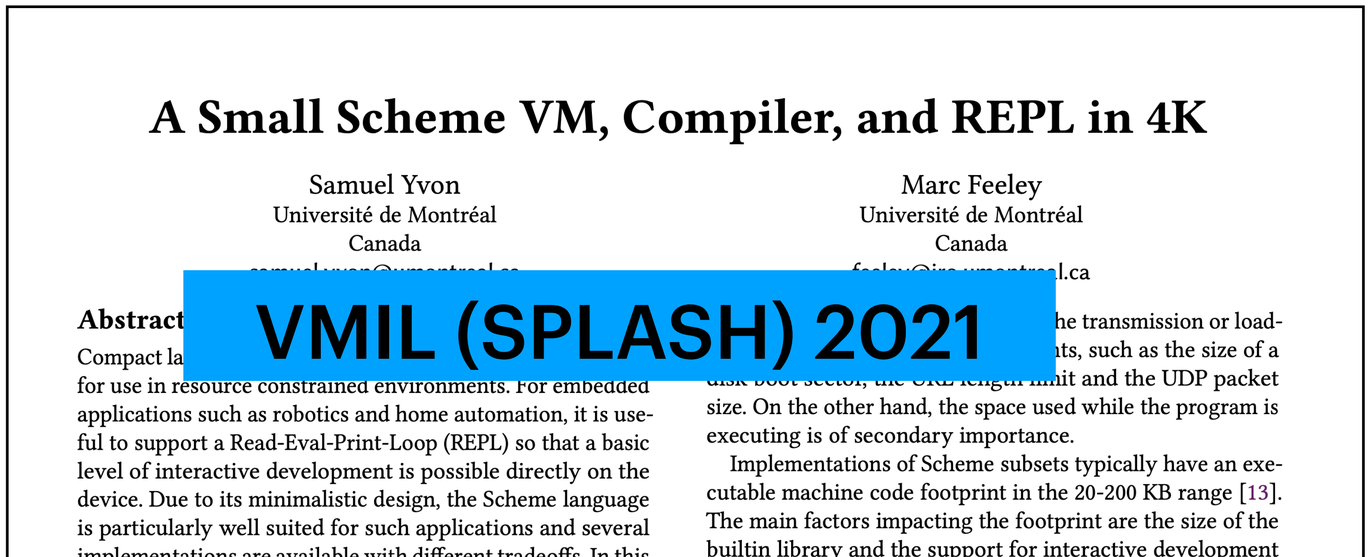
Ribbit is Extensible

- Primitives can be defined with a special form `define-primitive`
- Can be used as an Foreign Function Interface (FFI)



Ribbit is Compact

Ribbit has already **shown** its compactness :



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

Samuel Yvon
Université de Montréal
Canada

Marc Feeley
Université de Montréal
Canada

VMIL (SPLASH) 2021

Abstract

Compact la

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

he transmission or load-
nts, 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

However ...

- Subset of R4RS
- No file I/O, no rest parameters, no characters
- 4KB is with the JavaScript host
- Not user-friendly (no type checking)

Ribbit is Compact

R4RS library

Previous REPL

```
6.1: not, boolean?

6.2: eqv?, eq?, equal?

6.3: pair?, cons, car, cdr, set-car!, set-cdr!, caar, cadr,
caddr, cdaar, cdadr, cddar, cdddr, caaaar, cadddr, list,
length, list-ref

6.4: symbol?, symbol->string, string->symbol

6.5: =, <, >, <=, >=, +, *, -, quotient, number->string,
string->number

6.7: string?, make-string, string-length, string-ref,
string-set!, string->list, list->string

6.8: vector?, make-vector, vector, vector-length,
vector-ref, vector-set!, vector->list, list->vector

6.9: procedure?, call/cc, call-with-current-continuation

6.10: read, read-char, peek-char, eof-object?, write,
display, newline
```

59 procedures

New REPL

```
6.1: not, boolean?

6.2: eqv?, eq?, equal?

6.3: pair?, cons, car, cdr, set-car!, set-cdr!, caar, cadr, cdar, cddr, caaar, caadr,
cadar, caddr, cdaar, cdadr, cddar, cdddr, caaaar, caaadr, caadar, caaddr, cadaar, cadadr,
caddar, cadddr, cdaaar, cdaadr, cdadar, cdaddr, cddaar, cddadr, cdddar, cddddr, null?,
list?, list, length, append, reverse, list-ref, memv, memq, member, assv, assq, assoc

6.4: symbol?, symbol->string, string->symbol

6.5: number?, complex?, real?, rational?, integer?, exact?, inexact?, =, <,
>, <=, >=, zero?, positive?, negative?, odd?, even?, max, min, +, *, -, /,
abs, quotient, remainder, modulo, gcd, lcm, floor, ceiling, truncate, round,
number->string, string->number

6.6: char?, char=?, char<?, char>?, char<=?, char>=?, char-ci=?, char-ci<?, char-ci>?,
char-ci<=?, char-ci>=?, char-alphabetic?, char-numeric?, char-whitespace?,
char-upper-case?, char-lower-case?, char->integer, integer->char, char-upcase,
char-downcase

6.7: string?, make-string, string, string-length, string-ref, string-set!, string=?,
string<?, string>?, string<=?, string>=?, string-ci=?, string-ci<?, string-ci>?,
string-ci<=?, string-ci>=?, substring, string-append, string->list, list->string

6.8: vector?, make-vector, vector, vector-length, vector-ref, vector-set!, vector->list,
list->vector

6.9: procedure?, apply, map, for-each, call/cc, call-with-current-continuation

6.10: call-with-input-file, call-with-output-file, input-port?, output-port?,
current-input-port, current-output-port, open-input-file, open-output-file,
close-input-port, close-output-port, read, read-char, peek-char, eof-object?, write,
display, newline, write-char, load, eval
```

164 procedures

Ribbit is Compact

Liveness analysis

Inside the source code

```
;; Library
(define (display obj . rest)
  …)

(define (read . rest)
  …)

;; Source program
(display "hello world")
```

Inside the RVM

```
primitives = [
 prim3(lambda z,y,x:[x,y,z]), # Primitive ##rib
 prim1(lambda x:x),           # Primitive ##id
 ...
 prim1(putchar), # Primitive ##*
 ...
]
```

And other techniques, such as :

Not including the name of unneeded symbols
Having a compact encoding...

Let’s look closer…

Overview of Ribbit system

prog.scm

```
(write "hello")
```

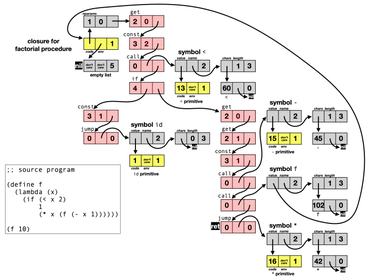
+

```
(define (write obj)
  …)

(define (eval expr)
  …)
```

lib.scm

Ribbit AOT compiler

- parsing
- expansion
- dead code elimination
- Optimizations



Serialization

```
#esolc##,2gra##,*##,di#
#,1gra##,<##,-##,f,,,
,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn
!.:lko!):lkp!,:lkr!(:l
kq{
```

1. Parsing & optimization

2. RVM code graph

3. Ribbit Intermediate Representation (RIBN)

rvm.js

```
rvm_code = "<rvm code here>";
…
while (true) {
   …
}
```

TEMPLATE

prog.js :

```
rvm_code = "#esolc##,2gra…";
…
while (true) {
   …
}
```

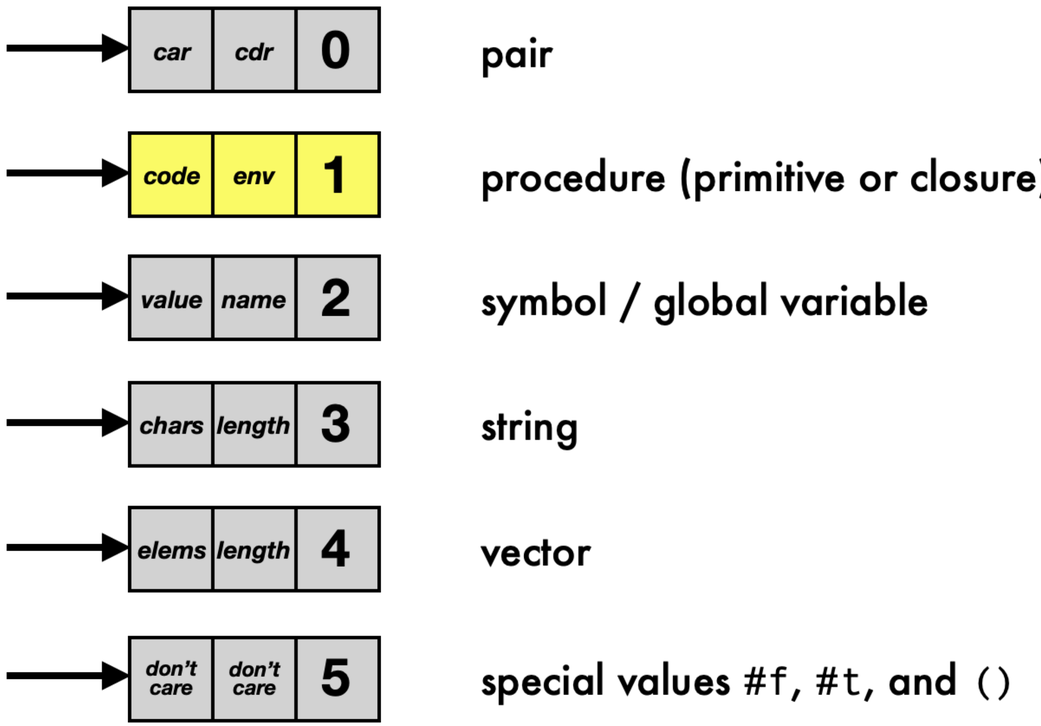
Ribbit VM overview

- To simplify memory management, all data is built out of triplets called **ribs** including Scheme objects, the RVM code, and the RVM stack

- Simple **stack machine with 5 instructions:**
  `jump/call` *`var`*, **`get`** *`var`*, **`set`** *`var`*, **`const`** *`value`*, **`if`** *`label`*

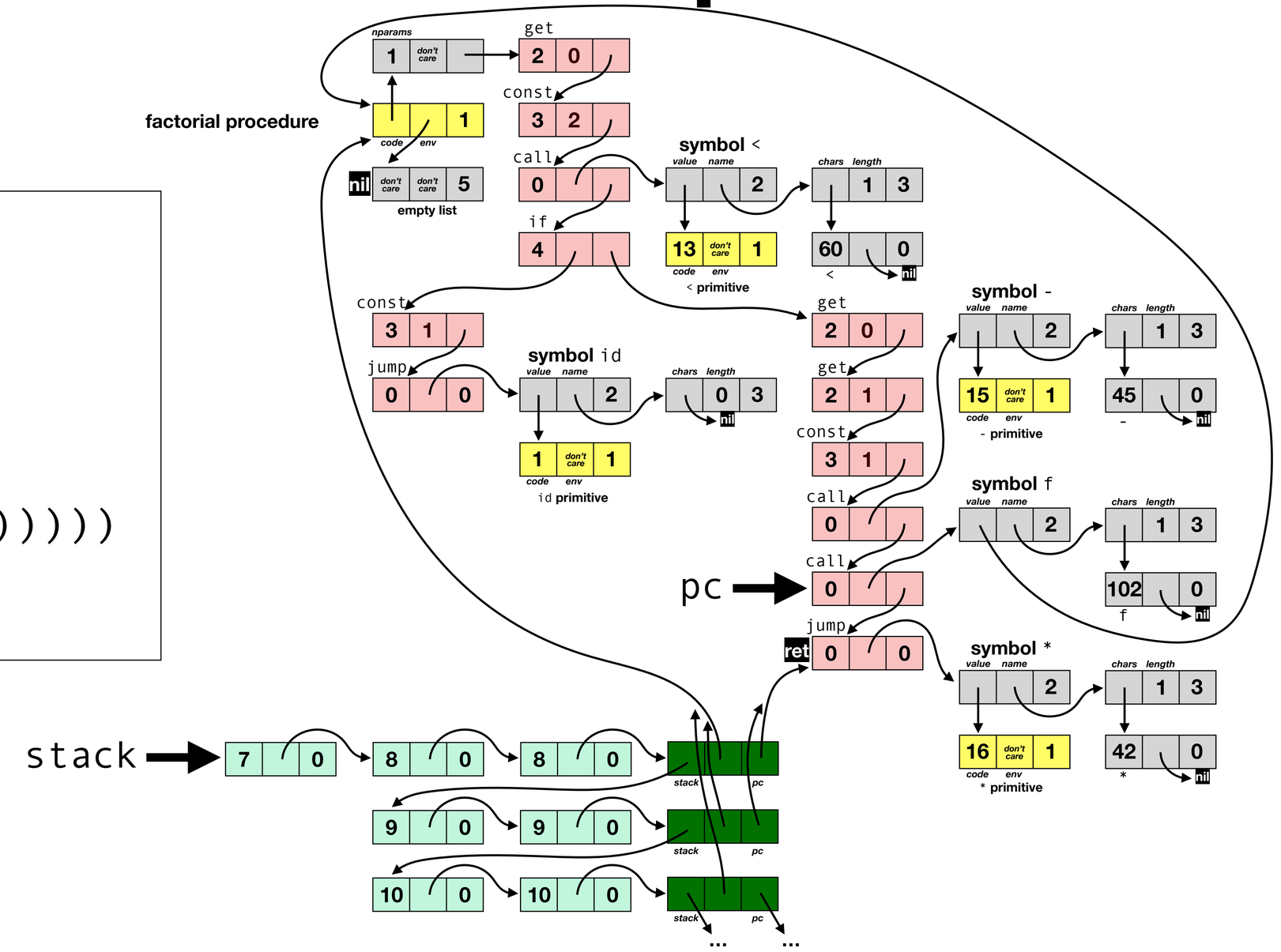
- Small set of **18 primitive procedures:**

```
rib?   field[0/1/2]
rib    field[0/1/2]-set!
id     arg1  arg2  close
eqv?   + -   * <   quotient
```



Execution example

```
;; source program

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

(f 10)
```



Typical RVM: `rvm.py`

```
rvm_code = ");'u?>vD?>vRD?>vRA?>vRA?>vR:?>vR=!(:lkm!':lkv6y"

import sys

stdo=sys.stdout

putchar=lambda c:[stdo.write(chr(c)),stdo.flush(),c][2]

def getchar():
 c=sys.stdin.read(1)
 push(ord(c) if len(c) else -1)

pos=-1
def get_byte():
 global pos
 pos+=1
 return ord(rvm_code[pos])

FALSE=[0,0,5]
TRUE=[0,0,5]
NIL=[0,0,5]

to_bool=lambda x:TRUE if x else FALSE
is_rib=lambda x:type(x) is list

stack=0

get_opnd=lambda o:(o if is_rib(o) else list_tail(stack,o))

def get_cont():
 s=stack
 while not s[2]:s=s[1]
 return s

def push(x):
 global stack
 stack=[x,stack,0]

def pop():
 global stack
 x=stack[0]
 stack=stack[1]
 return x

prim1=lambda f:lambda:push(f(pop()))
prim2=lambda f:lambda:push(f(pop(),pop()))
prim3=lambda f:lambda:push(f(pop(),pop(),pop()))

def arg2():x = pop();pop();push(x)
def close():push([pop()[0],stack,1])
```

```
def f0s(y,x):x[0]=y;return y
def f1s(y,x):x[1]=y;return y
def f2s(y,x):x[2]=y;return y

primitives = [
 prim3(lambda z,y,x:[x,y,z]),
 prim1(lambda x:x),
 pop,
 arg2,
 close,
 prim1(lambda x:to_bool(is_rib(x))),
 prim1(lambda x:x[0]),
 prim1(lambda x:x[1]),
 prim1(lambda x:x[2]),
 prim2(f0s),
 prim2(f1s),
 prim2(f2s),
 prim2(lambda y,x:to_bool(x is y if is_rib(x) or
                          is_rib(y)
                          else x==y)),
 prim2(lambda y,x:to_bool(x<y)),
 prim2(lambda y,x:x+y),
 prim2(lambda y,x:x-y),
 prim2(lambda y,x:x*y),
 prim2(lambda y,x:int(x/y)),
 getchar,
 prim1(putchar),
 prim1(exit),
]

def get_code():
 x=get_byte()-35
 return 57 if x<0 else x

def get_int(n):
 x=get_code()
 n*=46
 return n+x if x<46 else get_int(n+x-46)

list_tail=lambda lst,i:lst if i==0 else list_tail(lst[1],i-1)

# build the initial symbol table

symtbl=NIL
n=get_int(0)
while n>0:
 n-=1
 symtbl=[[FALSE,[NIL,0,3],2],symtbl,0]

accum=NIL
```

```
n=0
while 1:
 c=get_byte()
 if c==44:
  symtbl=[[FALSE,[accum,n,3],2],symtbl,0]; accum=NIL; n=0
 else:
  if c==59: break
  accum=[c,accum,0]
  n+=1

symtbl=[[FALSE,[accum,n,3],2],symtbl,0]
symbol_ref=lambda n: list_tail(symtbl,n)[0]

# decode the RVM instructions

while 1:
 x=get_code()
 n=x
 d=0
 op=0
 while 1:
  d=[20,30,0,10,11,4][op]
  if n<=2+d:break
  n-=d+3;op+=1
 if x>90:
  n=pop()
 else:
  if op==0:stack=[0,stack,0];op+=1
  n = (get_int(0)if n==d
       else symbol_ref(get_int(n-d-1)) if n>=d
       else symbol_ref(n)if op<3
       else n)
  if 4<op:
   n=[[n,0,pop()],NIL,1]
   if not stack:break
   op=4
 stack[0]=[op-1,n,stack[0]]

pc = n[0][2]

def set_global(val):
 global symtbl
 symtbl[0][0]=val
 symtbl=symtbl[1]

set_global([0,symtbl,1]) # primitive 0
set_global(FALSE)
set_global(TRUE)
set_global(NIL)
```

```
stack=[0,0,[5,0,0]] # primordial continuation (executes halt instr.)

while 1:
 o=pc[1]
 i=pc[0]
 if i<1: #-------------------------------------- jump/call
  o=get_opnd(o)[0]
  c=o[0]
  if is_rib(c):
   c2=[0,o,0]
   s2=c2
   nargs=c[0]
   while nargs:s2=[pop(),s2,0];nargs-=1
   if is_rib(pc[2]): # call
    c2[0]=stack
    c2[2]=pc[2]
   else: # jump
    k=get_cont()
    c2[0]=k[0]
    c2[2]=k[2]
   stack=s2
  else:
   primitives[c]()
   if is_rib(pc[2]): # call
    c=pc
   else: # jump
    c=get_cont()
    stack[1]=c[0]
  pc=c[2]
 elif i<2: #-------------------------------------- set
  get_opnd(o)[0]=stack[0]; stack = stack[1]
  pc=pc[2]
 elif i<3: #-------------------------------------- get
  push(get_opnd(o)[0])
  pc=pc[2]
 elif i<4: #-------------------------------------- const
  push(o)
  pc=pc[2]
 elif i<5: #-------------------------------------- if
  pc=pc[2 if pop()is FALSE else 1]
 else:
  break
```

Python RVM is
~200 LOC

Typical RVM: `rvm.py`

```
rvm_code = ");'u?>vD?>vRD?>vRA?>vRA?>vR:?>vR=!(:lkm!':lkv6y"
```

RIBN that prints “HELLO!”

Typical RVM: `rvm.py`

```
# decode the RVM instructions

while 1:
 x=get_code()
 n=x
 d=0
 op=0
 while 1:
  d=[20,30,0,10,11,4][op]
  if n<=2+d:break
  n-=d+3;op+=1
 if x>90:
  n=pop()
 else:
  if op==0:stack=[0,stack,0];op+=1
  n = (get_int(0)if n==d
       else symbol_ref(get_int(n-d-1)) if n>=d
       else symbol_ref(n)if op<3
       else n)
  if 4<op:
   n=[[n,0,pop()],NIL,1]
   if not stack:break
   op=4
 stack[0]=[op-1,n,stack[0]]

pc = n[0][2]
```

RIBN decoder

Ty vm.py

```
while 1:
 o=pc[1]
 i=pc[0]
 if i<1: #------------------------------------ jump/call
  o=get_opnd(o)[0]
  c=o[0]
  if is_rib(c):
   c2=[0,o,0]
   s2=c2
   nargs=c[0]
   while nargs:s2=[pop(),s2,0];nargs-=1
   if is_rib(pc[2]): # call
    c2[0]=stack
    c2[2]=pc[2]
   else: # jump
    k=get_cont()
    c2[0]=k[0]
    c2[2]=k[2]
   stack=s2
  else:
   primitives[c]()
   if is_rib(pc[2]): # call
    c=pc
   else: # jump
    c=get_cont()
    stack[1]=c[0]
  pc=c[2]
 elif i<2: #------------------------------------- set
  get_opnd(o)[0]=stack[0]; stack = stack[1]
  pc=pc[2]
 elif i<3: #------------------------------------- get
  push(get_opnd(o)[0])
  pc=pc[2]
 elif i<4: #------------------------------------- const
  push(o)
  pc=pc[2]
 elif i<5: #------------------------------------- if
  pc=pc[2 if pop()is FALSE else 1]
 else:
  break
```

Code graph interpreter

Typical RVM: `rvm.py`

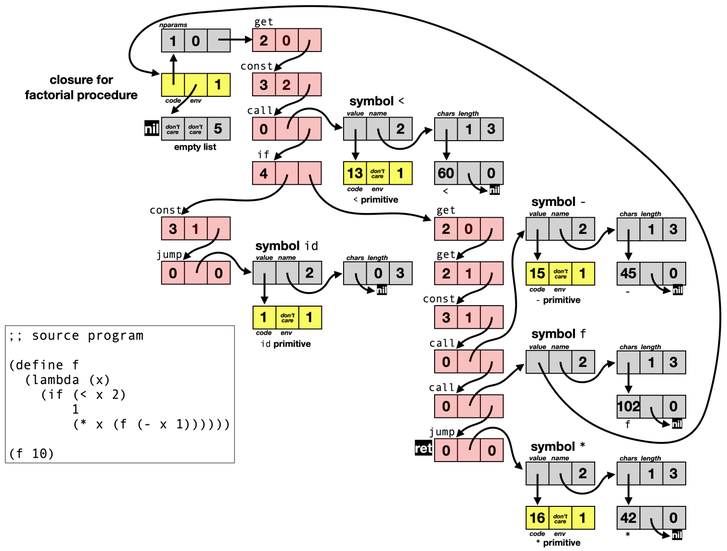
```
rvm_code = ");'u?>vD7>vRD?>vRA?>vRA?>vR:?>vR=!(:lkm!':lkv6y"

import sys

stdo=sys.stdout

putchar=lambda c:[stdo.write(chr(c)),stdo.flush(),c][2]

def getchar():
 c=sys.stdin.read(1)
 push(ord(c) if len(c) else -1)

pos=-1
def get_byte():
 global pos
 pos+=1
 return ord(rvm_code[pos])

FALSE=[0,0,5]
TRUE=[0,0,5]
NIL=[0,0,5]

to_bool=lambda x:TRUE if x else FALSE
is_rib=lambda x:type(x) is list

stack=0

get_opnd=lambda o:(o if is_rib(o) else list_tail(stack,o))

def get_cont():
 s=stack
 while not s[2]:s=s[1]
 return s

def push(x):
 global stack
 stack=[x,stack,0]

def pop():
 global stack
 x=stack[0]
 stack=stack[1]
 return x

prim1=lambda f:lambda:push(f(pop()))
prim2=lambda f:lambda:push(f(pop(),pop()))
prim3=lambda f:lambda:push(f(pop(),pop(),pop()))

def arg2():x = pop();pop();push(x)
def close():push([pop()[0],stack,1])
```

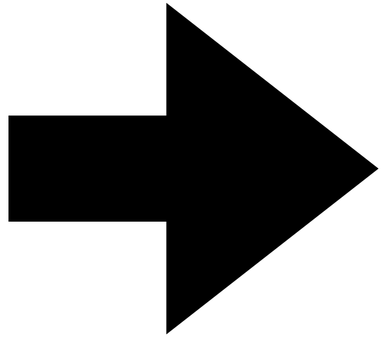
```
def f0s(y,x):x[0]=y;return y
def f1s(y,x):x[1]=y;return y
def f2s(y,x):x[2]=y;return y

primitives = [
 prim3(lambda z,y,x:[x,y,z]),
 prim1(lambda x:x),
 pop,
 arg2,
 close,
 prim1(lambda x:to_boo
 prim1(lambda x:x[0]),
 prim1(lambda x:x[1]),
 prim1(lambda x:x[2]),
 prim2(f0s),
 prim2(f1s),
 prim2(f2s),
 prim2(lambda y,x:to_bool(x is y if is_rib(x) or
                          is_rib(y)
                    else x==y)),
 prim2(lambda y,x:to_bool(x<y)),
 prim2(lambda y,x:x+y),
 prim2(lambda y,x:x-y),
 prim2(lambda y,x:x*y),
 prim2(lambda y,x:int(x/y)),
 getchar,
 prim1(putchar),
 prim1(exit),
]

def get_code():
 x=get_byte()-35
 return 57 if x<0 else x

def get_int(n):
 x=get_code()
 n*=46
 return n+x if x<46 else get_int(n+x-46)

list_tail=lambda lst,i:lst if i==0 else list_tail(lst

# build the initial symbol table

symtbl=NIL
n=get_int(0)
while n>0:
 n-=1
 symtbl=[[FALSE,[NIL,0,3],2],symtbl,0]

accum=NIL
```

```
primitives = [
 prim3(lambda z,y,x:[x,y,z]),
 prim1(lambda x:x),
 pop,
 arg2,
 close,
 prim1(lambda x:to_bool(is_rib(x))),
 prim1(lambda x:x[0]),
 prim1(lambda x:x[1]),
 prim1(lambda x:x[2]),
 prim2(f0s),
 prim2(f1s),
 prim2(f2s),
 prim2(lambda y,x:to_bool(x is y if is_rib(x) or
                                    is_rib(y)
                              else x==y)),
 prim2(lambda y,x:to_bool(x<y)),
 prim2(lambda y,x:x+y),
 prim2(lambda y,x:x-y),
 prim2(lambda y,x:x*y),
 prim2(lambda y,x:int(x/y)),
 getchar,
 prim1(putchar),
 prim1(exit),
]
```

Table of primitives

Ribbit's Encoding





```
#esolc##,2gra##,*##,di##,1gra##,<
##,-##,f,,,,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn!.:lko!):lk
p!,:lkr!(:lkq{
```

Ribbit's Encoding

Symbol Table

```
#esolc##,2gra##,*##,di##,1gra##,<
##,-##,f,,,,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn!.:lko!):lk
p!,:lkr!(:lkq{
```

Decoding instructions

Ribbit's Encoding

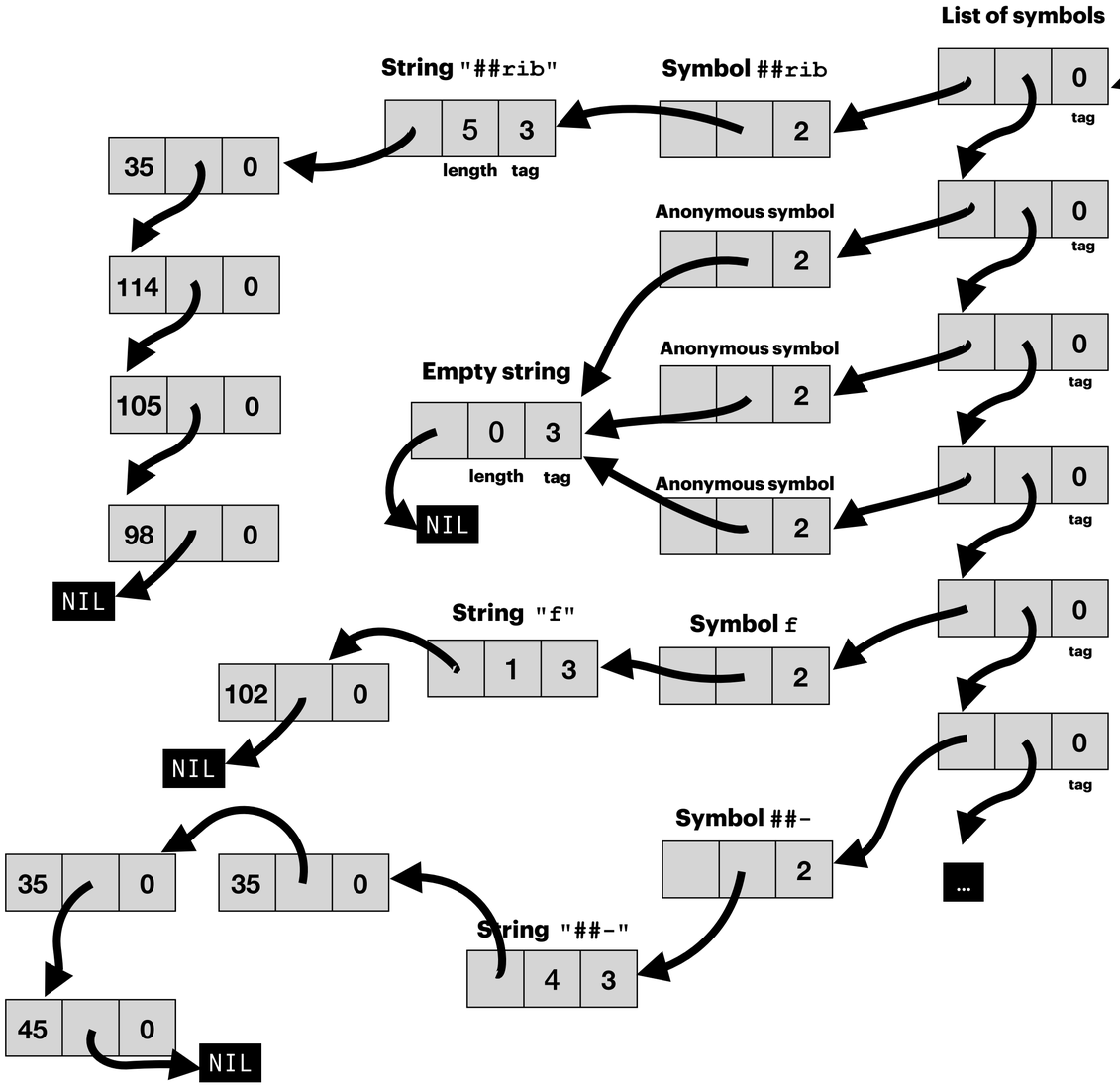
- The symbol-table is constructed from left to right
- Symbol's names are encoded in reverse order, separated by " , "

Symbol Table

```
#esolc##,2gra##,*##,di##,1gra##,<
##,-##,f,,,,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn!.:lko!):lk
p!,:lkr!(:lkq{
```

Decoding instructions

Ribbit's Encoding

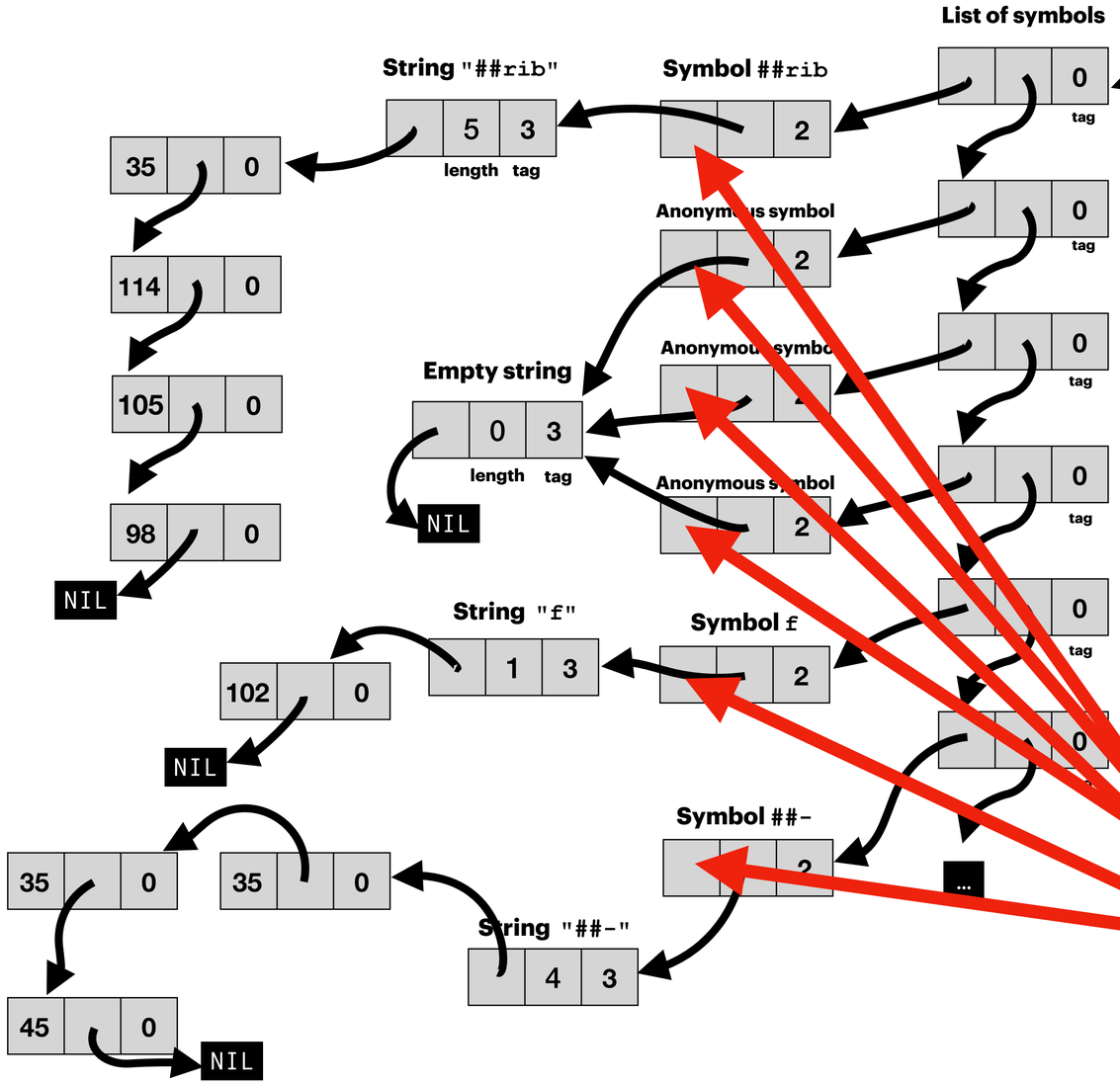


Symbol Table

```
#esolc##,2gra##,*##,di##,1gra##,<
##,-##,f,,,,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn!.:lko!):lk
p!,:lkr!(:lkq{
```

Decoding instructions

Ribbit's Encoding



Symbol Table

```
#esolc##,2gra##,*##,di##,1gra##,<
##,-##,f,,,,bir##;'u!',>?l_^+l~@m
^{Ak!+:lkl!*:lkm!-:lkn!.:lko!):lk
p!,:lkr!(:lkq{
```

Decoding instructions

Global variables (to be set later on)

Ribbit's Encoding

Instructions of the decoding stack machine

Decoding instruction type	Argument (*arg*)	RVM instruction generated	Effect on decoding stack state using the notation $\langle$*current-stack-state*$\rangle \rightarrow \langle$*next-stack-state*$\rangle$
$PUSH_0$	`int` or `sym`	`jump`	*stack...* $\rightarrow$ `[0,` *arg*`, 0]` *stack...*
$LINK_0$	`int` or `sym`	`call`	*x stack...* $\rightarrow$ `[0,` *arg*`,` *x*`]` *stack...*
$LINK_1$	`int` or `sym`	`set`	*x stack...* $\rightarrow$ `[1,` *arg*`,` *x*`]` *stack...*
$LINK_2$	`int` or `sym`	`get`	*x stack...* $\rightarrow$ `[2,` *arg*`,` *x*`]` *stack...*
$LINK_3$	`int` or `sym`	`const`	*x stack...* $\rightarrow$ `[3,` *arg*`,` *x*`]` *stack...*
$MERGE_3$	`int`	`const`	*y x stack...* $\rightarrow$ `[3, [[`*arg*`, 0,` *y*`], 0, 1],` *x*`]` *stack...*
$MERGE_4$	*none*	`if`	*y x stack...* $\rightarrow$ `[4,` *y*`,` *x*`]` *stack...*

RVM code encoding

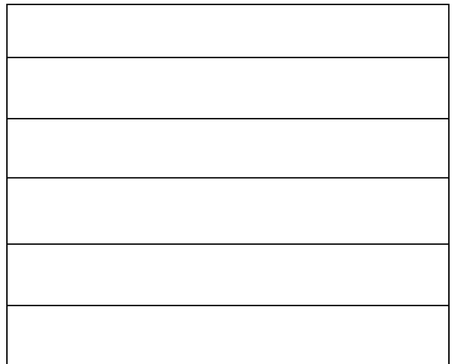
Serialization

```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!',>?l_^+l~@m^{Ak!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!,:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
`if`	91-91	1	$MERGE_4$		

STACK



Ribbit's Encoding

Decoding instructions

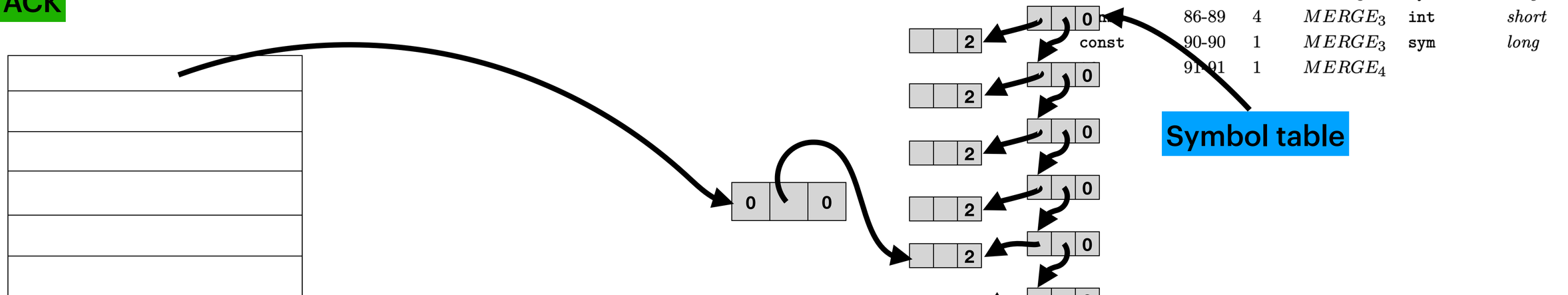
```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!',>?l_^:l^@m^{AK!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!,:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	2	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
	91-91	1	$MERGE_4$		

arg = 4 - 0 = 4

STACK



Ribbit's Encoding

Decoding instructions

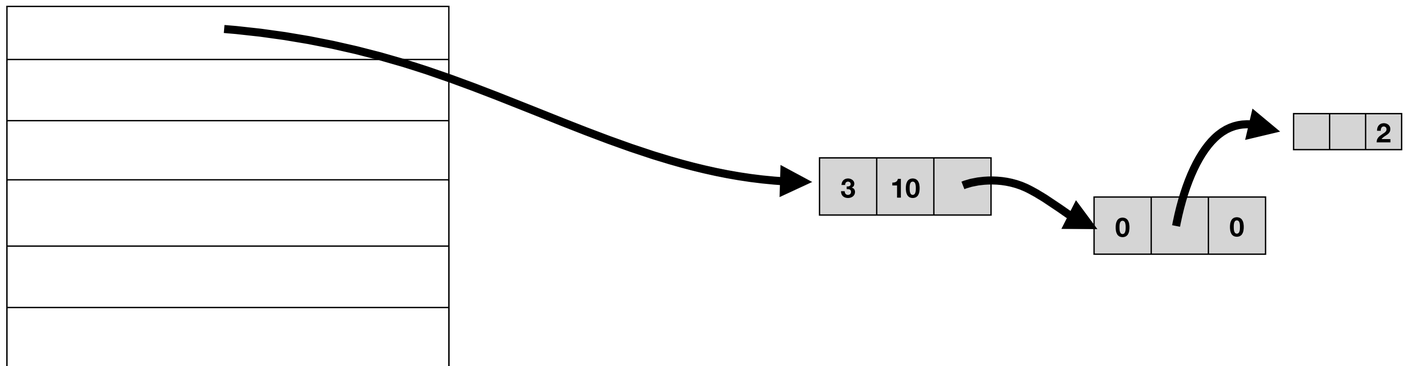
```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!',>?l_^+l~@m^{Ak!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!,:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
`if`	91-91	1	$MERGE_4$		

arg = 72 - 82 = 10

STACK



Ribbit's Encoding

Decoding instructions

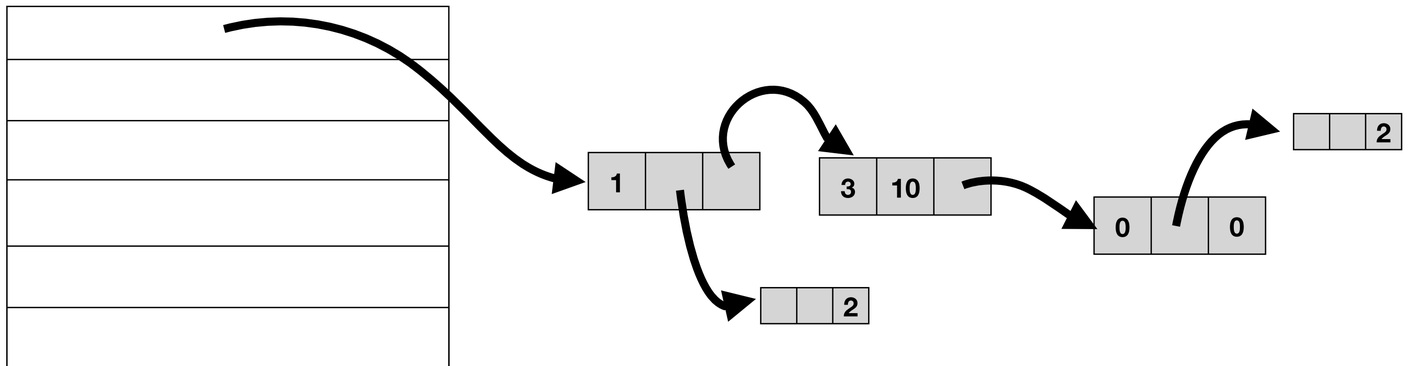
```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!!_>?l_^+l~@m^{Ak!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!;:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
`if`	91-91	1	$MERGE_4$		

arg = 4

STACK



Ribbit's Encoding

Decoding instructions

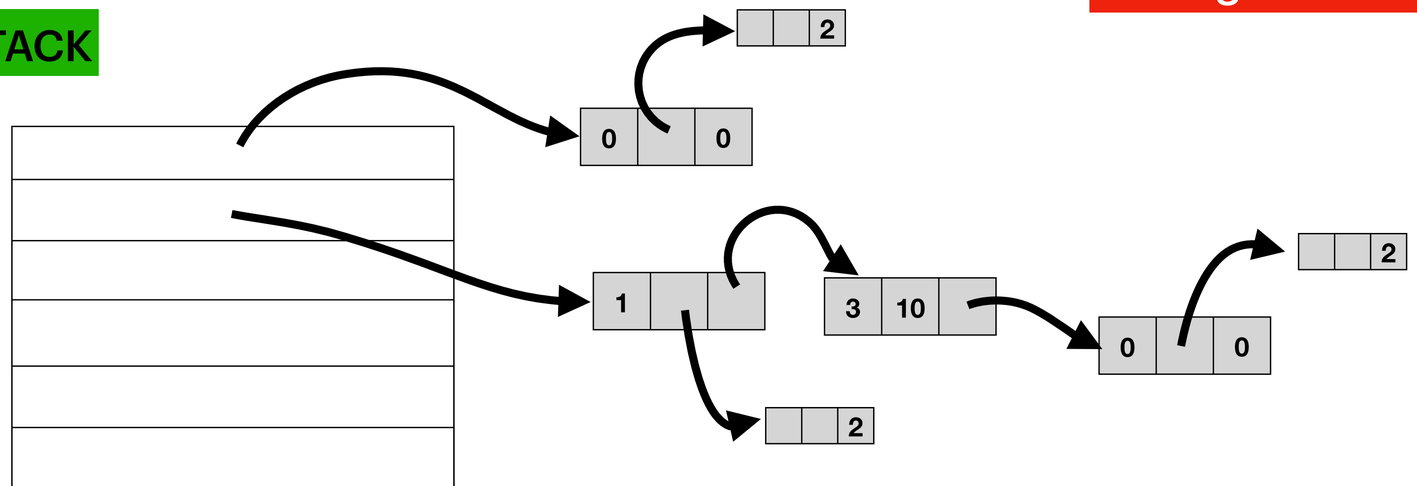
```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!',>?l_^+l_@m^{Ak!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!,:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
`if`	91-91	1	$MERGE_4$		

arg = 9 - 0 = 9

STACK



Ribbit's Encoding

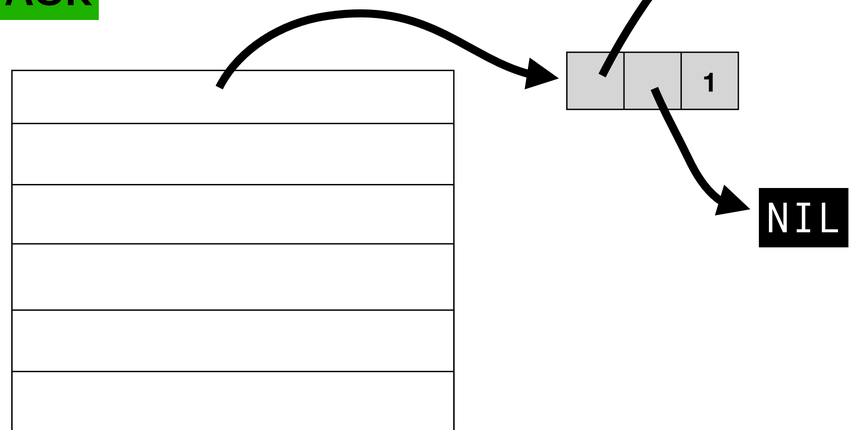
Decoding instructions

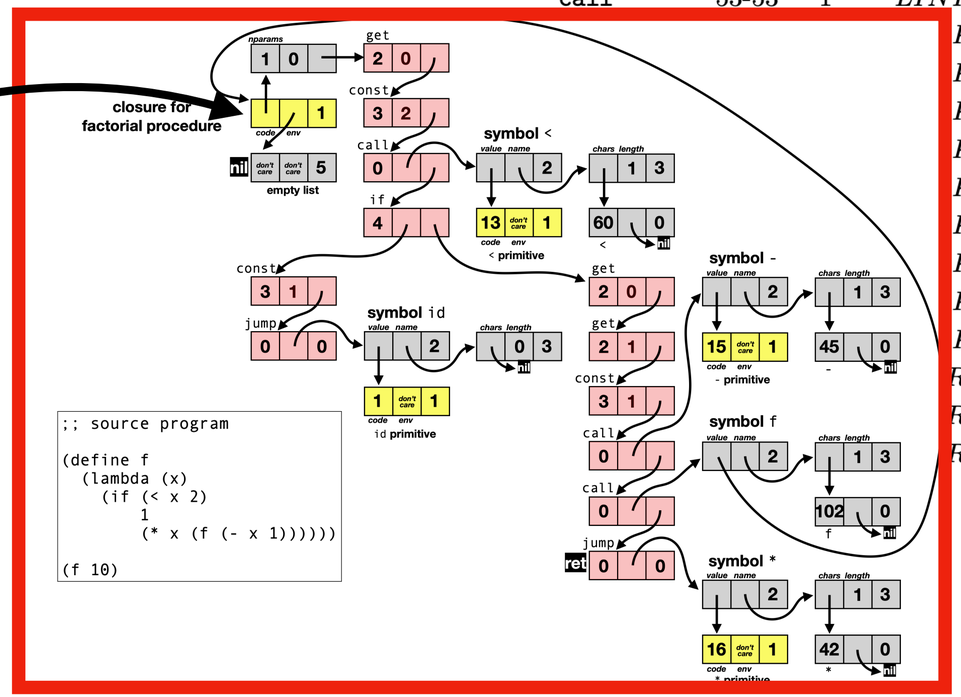
```
#esolc##,2gra##,*##,di##,1gra##,<##,
-##,f,,,,bir##;'u!',>?l_^+l~@m^{Ak!
+:lkl!*:lkm!-:lkn!.:lko!):lkp!,:lkr!
(:lkq{
```

Decoding instructions

RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
			K_0	`sym`	*long*
			K_1	`int`	*long*
			K_1	`sym`	*long*
			K_2	`int`	*short*
			K_2	`int`	*long*
			K_2	`sym`	*long*
			K_3	`int`	*short*
			K_3	`int`	*long*
			K_3	`sym`	*long*
			RGE_3	`int`	*short*
			RGE_3	`sym`	*long*
			RGE_4		

STACK





Presentation **Outline**

1 What is Ribbit ?

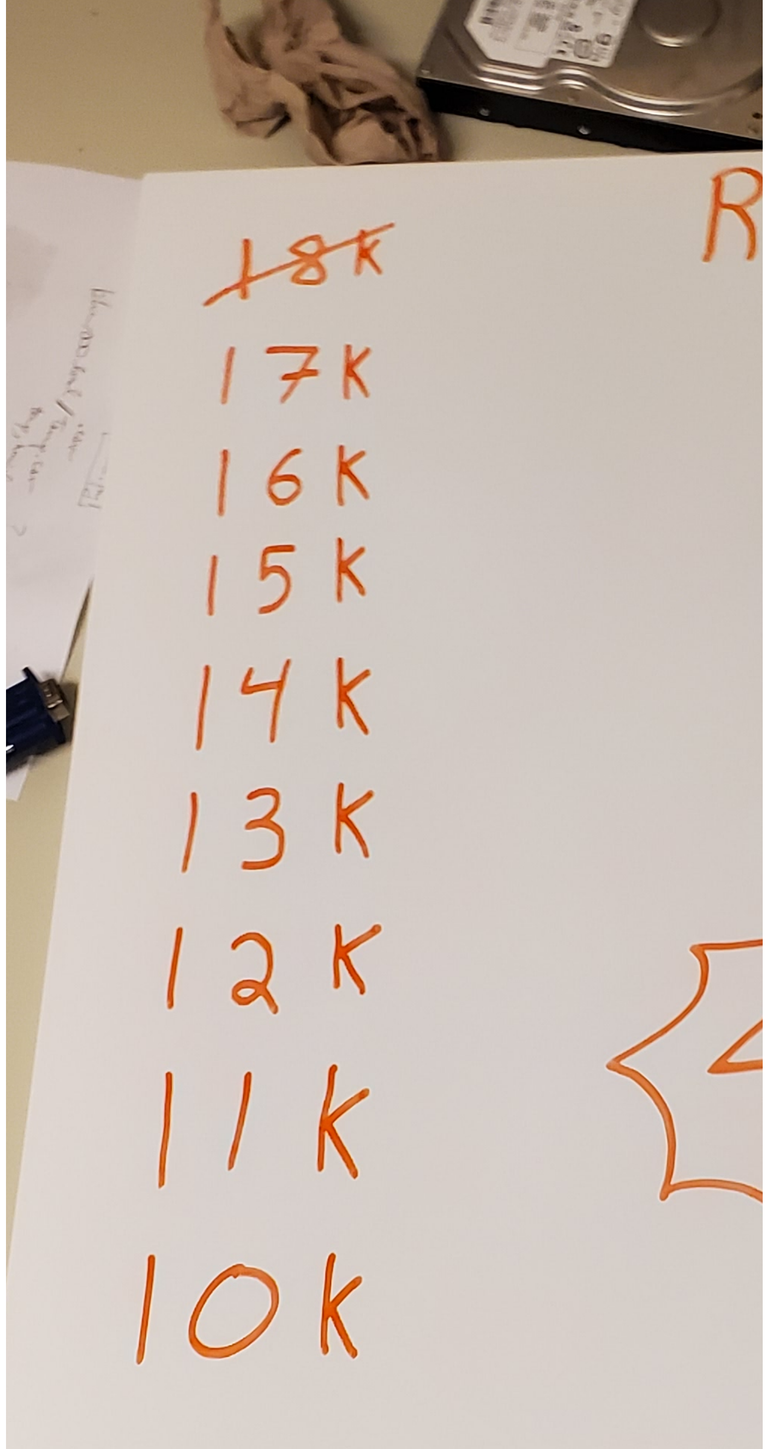
A Scheme implementation based on a custom VM that is :

- Compact
- Extensible
- Portable

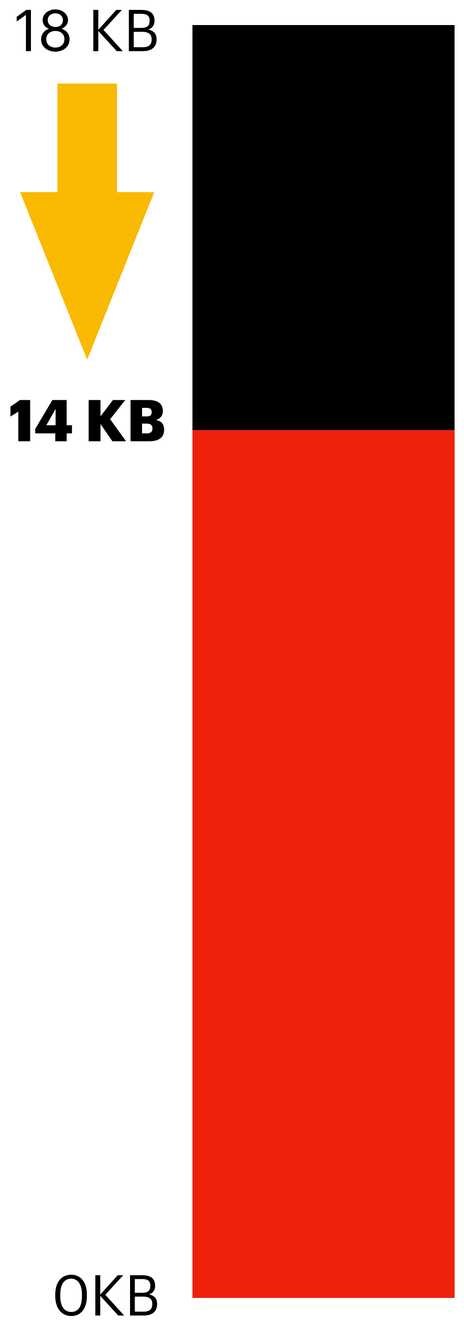
2 **How we enhanced Ribbit to fit a REPL in 7Kb ?**

- **Fully R4RS compliant**
- **Novel encoding**
- **Compression**
- **x86 Host**

The journey to 7KB



Writing R4RS
with style



We improved the way we wrote R4RS :

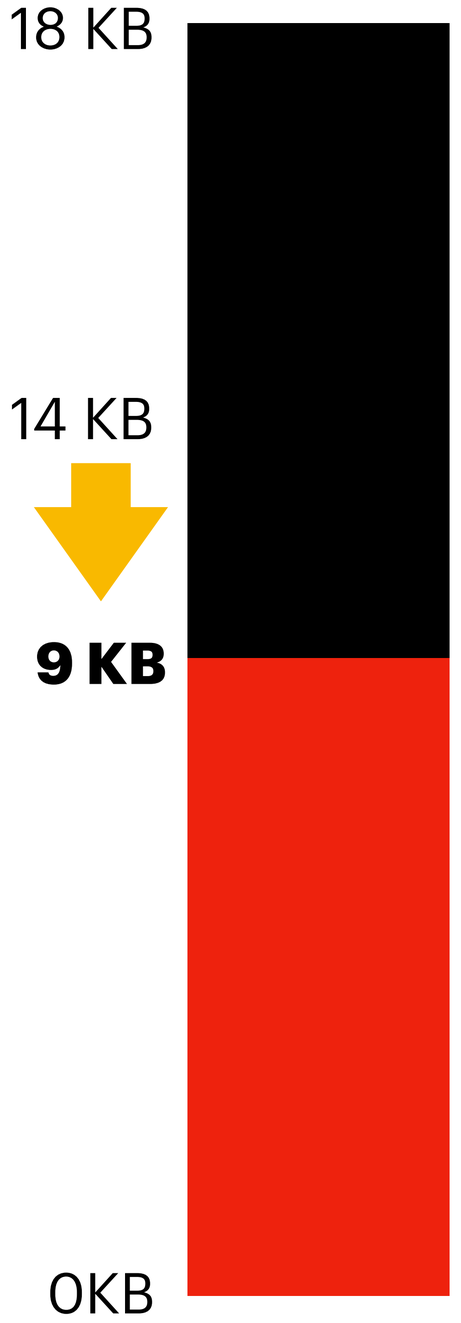
- Higher order procedures
- Using numbers instead of character literals (boxed)

Ribbit's Encoding

Enhancements

The enhanced encoding is :

- Using all 256 codes per byte rather than 92
- DAG code graph no longer tail duplicated
- The encoding is specialized to the source code



18 KB

Ribbit's Encoding

SHARE instruction

Decoding instruction type	Argument (*arg*)	RVM instruction generated	Effect on decoding stack state using the notation $\langle$*current-stack-state*$\rangle \rightarrow \langle$*next-stack-state*$\rangle$
$PUSH_0$	`int` or `sym`	`jump`	*stack...* → `[0,` *arg*`, 0]` *stack...*
$LINK_0$	`int` or `sym`	`call`	*x stack...* → `[0,` *arg*`,` *x*`]` *stack...*
$LINK_1$	`int` or `sym`	`set`	*x stack...* → `[1,` *arg*`,` *x*`]` *stack...*
$LINK_2$	`int` or `sym`	`get`	*x stack...* → `[2,` *arg*`,` *x*`]` *stack...*
$LINK_3$	`int` or `sym`	`const`	*x stack...* → `[3,` *arg*`,` *x*`]` *stack...*
$MERGE_3$	`int`	`const`	*y x stack...* → `[3, [[`*arg*`, 0,` *y*`], 0, 1],` *x*`]` *stack...*
$MERGE_4$	*none*	`if`	*y x stack...* → `[4,` *y*`,` *x*`]` *stack...*
$SHARE$	`int`	*none*	*x stack...* → *list-tail(x, arg) x stack...*

Table 1. The decoding instructions and their effect on the decoding stack.

9 KB

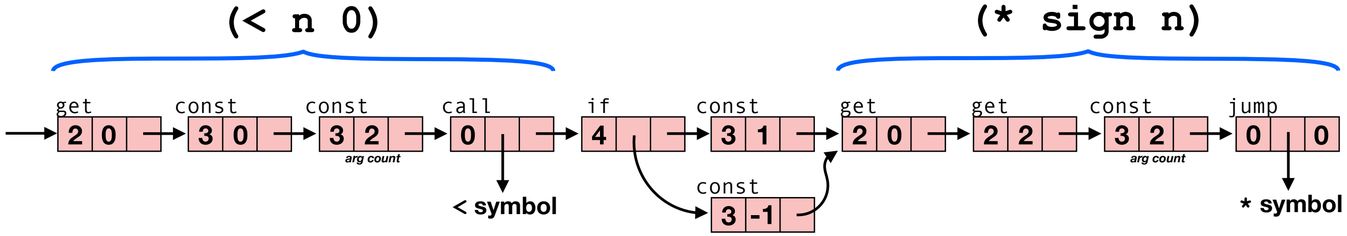
0KB

18 KB

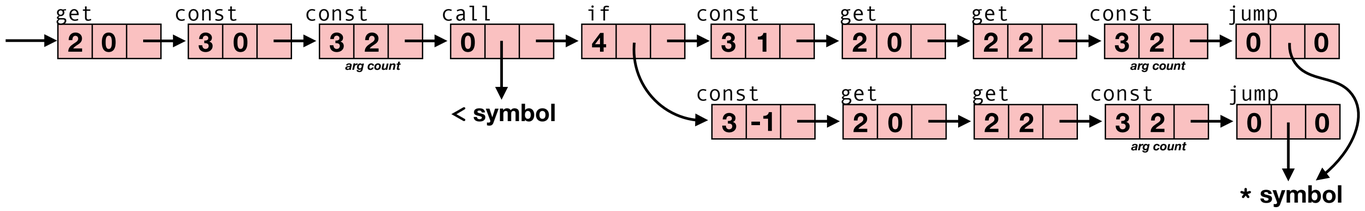
Ribbit's Encoding

SHARE instruction

With SHARE



Without SHARE



Decoding instruction type	Effect on decoding stack state using the notation $\langle current\text{-}stack\text{-}state \rangle \rightarrow \langle next\text{-}stack\text{-}state \rangle$
SHARE	x *stack...* $\rightarrow$ *list-tail(x, arg) x stack...*

9 KB

```
(lambda (n)
  (let ((sign (if (< n 0) -1 1)))
    (* sign n)))
```

0KB

Ribbit's Encoding

Instruction specialization

```
// @@(feature encoding/optimal
stack=0;
while(1){
    x = get_code();
    n=x;
    op=-1;
    // @@(replace "[0,1,2]" (list->host encoding/optimal/sizes "[" "," "]")
    while((d=[0,1,2][++op])<=n) n-=d;
    // )@@

    if (op<4) stack = [0,stack,0];
    if (op<24) n=op%2>0?get_int(n):n;

    if(op<20){ // jump call set get const
        i=op/4-1;
        i=i<0?0:i>>0;
        n=op%4/2<1?n:symbol_ref(n);
    }

    ...
    stack[0]=[i,n,stack[0]];
}
// )@@
```

Ribbit's Encoding

Instruction specialization

```
// @@(feature encoding/optimal
stack=0;
while(1){
    x = get_code();
    n=x;
    op=-1;
    // @@(replace "[0,1,2]" (list->host encoding/optimal/sizes "[" "," "]")
    while((u=[0,1,2][++op])<=n) n-=u;
    // )@@

    if (op<4) stack = [0,stack,0];
    if (op<24) n=op%2>0?get_int(n):n;

    if(op<20){ // jump call set get const
        i=op/4-1;
        i=i<0?0:i>>0;
        n=op%4/2<1?n:symbol_ref(n);
    }
    ...
    stack[0]=[i,n,stack[0]];
}
// )@@
```

18 KB

9 KB

0KB

Ribbit's Encoding

Instruction specialization

```
  INSTRUCTION       SIZES   START
(((jump int short)    5      0)
 ((jump int long)     1      5)
 ((jump sym short)    11     6)
 ((jump sym long)     2      17)
 ((call int short)    5      19)
 ((call int long)     1      24)
 ((call sym short)    15     25)
 ((call sym long)     2      40)
 ((set int short)     4      42)
 ((set int long)      1      46)
 ((set sym short)     0      47)
 ((set sym long)      2      47)
 ((get int short)     7      49)
 ((get int long)      1      56)
 ((get sym short)     4      57)
 ((get sym long)      2      61)
 ((const int short)   11     63)
 ((const int long)    3      74)
 ((const sym short)   0      77)
 ((const sym long)    1      77)
 ((const proc short)  7      78)
 ((const proc long)   1      85)
 ((skip int short)    3      86)
 ((skip int long)     1      89)
 (if                  1      90))
```

```
// @@(feature encoding/optimal
stack=0;
while(1){
    x = get_code();
    n=x;
    op=-1;
    // @@(replace "[0,1,2]" (list->host encoding/optimal/sizes "[" "," "]")
    while((d=[0,1,2][++op])<=n) n-=d;
    // )@@

    if (op<4) stack = [0,stack,0];
    if (op<24) n=op%2>0?get_int(n):n;

    if(op<20){ // jump call set get const
        i=op/4-1;
        i=i<0?0:i>>0;
        n=op%4/2<1?n:symbol_ref(n);
    }

    ...
    stack[0]=[i,n,stack[0]];
}
// )@@
```

Ribbit's Encoding

Instruction specialization

INSTRUCTION	SIZES	START
(((jump int short)	5	0)
((jump int long)	1	5)
((jump sym short)	11	6)
((jump sym long)	2	17)
((call int short)	5	19)
((call int long)	1	24)
((call sym short)	15	25)
((call sym long)	2	40)
((set int short)	4	42)
((set int long)	1	46)
((set sym short)	0	47)
((set sym long)	2	47)
((get int short)	7	49)
((get int long)	1	56)
((get sym short)	4	57)
((get sym long)	2	61)
((const int short)	11	63)
((const int long)	3	74)
((const sym short)	0	77)
((const sym long)	1	77)
((const proc short)	7	78)
((const proc long)	1	85)
((skip int short)	3	86)
((skip int long)	1	89)
(if	1	90))

Encodes range information

test.js

```
stack=0;
while(1){
    x = get_code();
    n=x
    op=-1;

    while((d=[5,1,11,2,5,1,15,2,4,1,0,2,7,1,4,2,11,3,0,1,7,1,3,1,1][++op])<=n)

    if (op<4) stack = [0,stack,0];
    if (op<24) n=op%2>0?get_int(n):n

    if(op<20){ // jump call set get const
        i=op/4-1;
        i=i<0?0:i>>0
        n=op%4/2<1?n:symbol_ref(n)
    }

    ...
                                    stack[0]];
```

```
> ./rsc -t js fibo.scm -o test.js
```

Ribbit's Encoding

Decoding instruction specialization

- The best encoding can be determined by the compiler and injected into the RVM with annotations
- The compiler can compute the ranges that generate the fewest `long` encoding instructions
- This minimizes the code overall

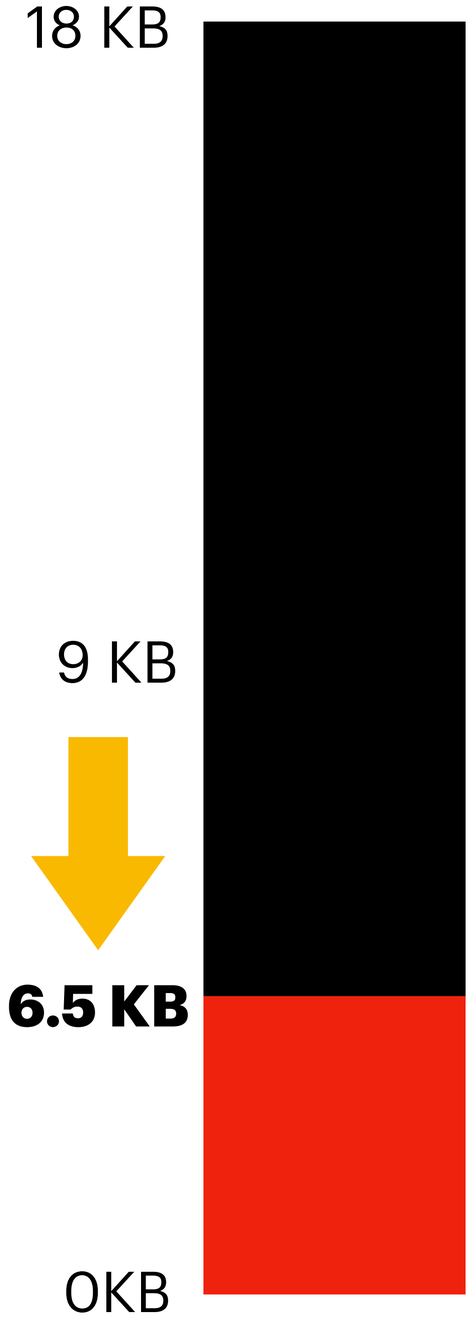
Ribbit's Compression

LZSS

- LZSS is a compression algorithm that uses back pointers to refer to part of text that has already been seen

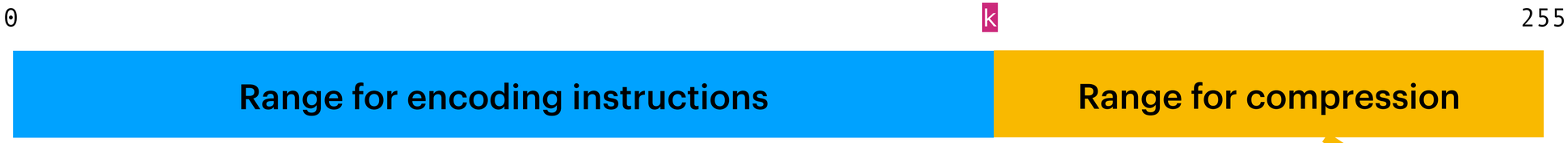
`I am Sam` → `I am S(4,2)`

The difficulty : Encoding the backpointers !



Ribbit's Compression

LZSS encoding



RVM instruction	Range	Size	Decoding instruction	Argument	Form
`jump`	0-19	20	$PUSH_0$	`sym`	*short*
`jump`	20-20	1	$PUSH_0$	`int`	*long*
`jump`	21-22	2	$PUSH_0$	`sym`	*long*
`call`	23-52	30	$LINK_0$	`sym`	*short*
`call`	53-53	1	$LINK_0$	`int`	*long*
`call`	54-55	2	$LINK_0$	`sym`	*long*
`set`	56-56	1	$LINK_1$	`int`	*long*
`set`	57-59	2	$LINK_1$	`sym`	*long*
`get`	59-68	10	$LINK_2$	`int`	*short*
`get`	69-69	1	$LINK_2$	`int`	*long*
`get`	70-71	2	$LINK_2$	`sym`	*long*
`const`	72-82	11	$LINK_3$	`int`	*short*
`const`	83-83	1	$LINK_3$	`int`	*long*
`const`	84-85	2	$LINK_3$	`sym`	*long*
`const`	86-89	4	$MERGE_3$	`int`	*short*
`const`	90-90	1	$MERGE_3$	`sym`	*long*
`if`	91-91	1	$MERGE_4$		

Encoding of back-pointers

```
(length, offset) => offset * size-base + length
```

+ Next byte

Back-pointers are encoded on 2 bytes

Decoding of back-pointers

```
offset = (quotient backpointer-value size-base)
length = (+ 3(modulo backpointer-value size-base))
```

Decided by the compiler

Ribbit's Compression

LZSS decoding

```
;; @@(feature compression/lzss/2b
%define BYTE_BASE 00            ;; @@(replace "00" compression/lzss/2b/byte-base)@@
%define SIZE_BASE 00            ;; @@(replace "00" compression/lzss/2b/size-base)@@
%define RIBN_BASE 00            ;; @@(replace "00" compression/lzss/2b/ribn-base)@@
%define RIBN_SIZE 00            ;; @@(replace "00" compression/lzss/2b/ribn-size)@@
%define COMPRESSED_RIBN_SIZE 00 ;; @@(replace "00" compression/lzss/2b/compressed-ribn-size)@@

decompress:
    mov  rvm_code_ptr, rvm_code

    sub  esp, RIBN_SIZE
    mov  edi, esp
    mov  ebp, esp

decompress_loop:
    mov ebx, esi
    sub ebx, rvm_code
    cmp ebx, COMPRESSED_RIBN_SIZE
    jns decompress_end

    movC eax, 0
    mov al, [rvm_code_ptr]
    inc rvm_code_ptr

    cmp eax, RIBN_BASE
    ...
;; )@@
```

Ribbit's Compression

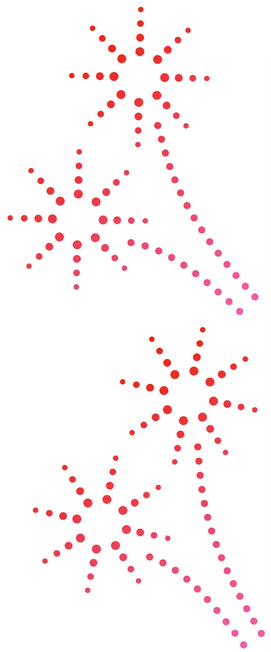
LZSS decoding

```
;; @@(feature compression/lzss/2b
%define BYTE_BASE 00            ;; @@(replace "00" compression/lzss/2b/byte-base)@@
%define SIZE_BASE 00            ;; @@(replace "00" compression/lzss/2b/size-base)@@
%define RIBN_BASE 00            ;; @@(replace "00" compression/lzss/2b/ribn-base)@@
%define RIBN_SIZE 00            ;; @@(replace "00" compression/lzss/2b/ribn-size)@@
%define COMPRESSED_RIBN_SIZE 00 ;; @@(replace "00" compression/lzss/2b/compressed-ribn-size)@@

decompress:
    mov  rvm_code_ptr, rvm_code

    sub  esp, RIBN_SIZE
    mov  edi, esp
    mov  ebp, esp

decompress_loop:
    mov ebx, esi
    sub ebx, rvm_code
    cmp ebx, COMPRESSED_RIBN_SIZE
    jns decompress_end

    movC eax, 0
    mov al, [rvm_code_ptr]
    inc rvm_code_ptr

    cmp eax, RIBN_BASE
    ...
;; )@@
```

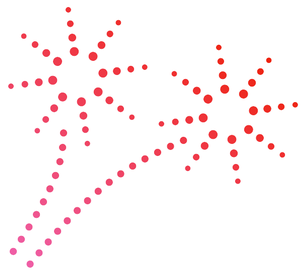
Injected by the compiler

50 LOC in x86 assembly !

Results



```
$ ./repl
> (define x (open-output-file "test_file"))
> (display “hello TFP !" x)

$ cat test_file
hello TFP !

$ ls -la ./repl
-rwxr-xr-x 1 501 lp 6624 Sep  2 09:37 ./repl
```



Results

	gsi (secs)	pna		pa		tc pna		tc pa		x86 REPL
ctak	**0.2** s	**0.9**×	2.1 KB	**1.0**×	2.2 KB	**1.5**×	8.8 KB	**1.8**×	9.3 KB	**2.8**×
fib	**26.2** s	**0.3**×	2.0 KB	**0.4**×	2.0 KB	**1.9**×	8.6 KB	**2.2**×	9.1 KB	**4.9**×
ack	**2.2** s	**0.4**×	2.0 KB	**0.4**×	2.0 KB	**1.6**×	8.6 KB	**1.9**×	9.1 KB	**5.7**×
tak	**2.3** s	**0.6**×	2.0 KB	**0.6**×	2.0 KB	**1.5**×	8.6 KB	**1.7**×	9.1 KB	**3.5**×
takl	**2.5** s	**0.9**×	2.2 KB	**1.0**×	2.2 KB	**0.8**×	8.7 KB	**1.0**×	9.2 KB	**1.0**×
primes	**1.4** s	**0.8**×	2.3 KB	**1.0**×	2.3 KB	**1.5**×	8.8 KB	**1.8**×	9.3 KB	**2.6**×
deriv	**0.7** s	**6.8**×	2.7 KB	**8.2**×	2.7 KB	**26.3**×	9.2 KB	**32.6**×	9.8 KB	**7.3**×
mazefun	**1.4** s	**0.7**×	4.0 KB	**0.8**×	4.1 KB	**1.7**×	9.9 KB	**2.0**×	11 KB	N/A
nqueens	**2.0** s	**0.8**×	3.4 KB	**0.9**×	3.5 KB	**1.7**×	8.8 KB	**2.0**×	9.2 KB	N/A
sum	**19.4** s	**0.3**×	2.0 KB	**0.4**×	2.0 KB	**2.1**×	8.6 KB	**2.5**×	9.1 KB	N/A

Table 4. Execution time when using the Gambit Scheme Interpreter and for Ribbit the relative execution time and footprint of the x86 assembly host on different benchmarks.

Future work

R7RS in 15 to 30 KB ?

Generating RVMs with LLM ?