

# The minischeme project

Michel Schinz

Advanced Compiler Construction – 2009-02-20

# Project overview

What you get:

1. an interpreter and a compiler for minischeme, written in Scala,
2. a virtual machine, written in C.

What you have to do:

1. two non-graded “warm-up” exercises,
2. add a garbage collector to the virtual machine,
3. add support for closures to the compiler,
4. optimize tail calls in the compiler,
5. an advanced project of your choice.

# The minischeme language

# The minischeme language

Minischeme is a dialect of Scheme, itself a dialect of Lisp. Its main characteristics are:

- it is “dynamically typed”,
- it has few side effects (exceptions: arrays, input/output),
- it is functional: functions are first-class values and can be nested,
- it is very simple, with seven keywords (define, let, begin, lambda, if, and and or)
- it has three kinds of values: integers, vectors and functions,
- memory is freed automatically.

# Syntax

(define *name expr*)

Global value definition: *name* is bound to the value of *expr*.  
Only valid at the top level.

Global values are visible in the whole program, but  
initialized in the order in which they are written.

(let ((*name<sub>1</sub> expr<sub>1</sub>*) ...) *body<sub>1</sub> ... body<sub>n</sub>*)

Local value(s) definition: *name<sub>1</sub>* is bound to the value of  
*expr<sub>1</sub>*, *name<sub>2</sub>* to the value of *expr<sub>2</sub>*, ... in *body<sub>1...n</sub>*. The value  
of the whole expression is the value of *body<sub>n</sub>*. (Note: name  
*name<sub>j</sub>* is also visible in *expr<sub>k</sub>* for all  $k > j$ )

(begin *expr<sub>1</sub> expr<sub>2</sub> ... expr<sub>n</sub>*)

Sequential execution: *expr<sub>1</sub>...expr<sub>n</sub>* are evaluated in order.  
The value of the whole expression is the value of *expr<sub>n</sub>*.

# Syntax

(lambda (*name*<sub>1</sub> ... *name*<sub>*n*</sub>) *body*<sub>1</sub> ... *body*<sub>*m*</sub>)

Anonymous function, with parameters *name*<sub>1</sub> ... *name*<sub>*n*</sub> and body *body*<sub>1</sub> ... *body*<sub>*m*</sub>.

(if *expr*<sub>*c*</sub> *expr*<sub>*t*</sub> *expr*<sub>*e*</sub>)

Conditional: evaluates to the value of *expr*<sub>*e*</sub> iff *expr*<sub>*c*</sub> evaluates to 0, otherwise evaluates to the value of *expr*<sub>*t*</sub>.

(and *expr*<sub>1</sub> *expr*<sub>2</sub>)

Conjunction: evaluates to the value of *expr*<sub>1</sub> if *expr*<sub>1</sub> evaluates to 0, otherwise evaluates to the value of *expr*<sub>2</sub>.

(or *expr*<sub>1</sub> *expr*<sub>2</sub>)

Disjunction: evaluates to the value of *expr*<sub>1</sub> if *expr*<sub>1</sub> does not evaluate to 0, otherwise evaluates to the value of *expr*<sub>2</sub>.

# Syntax

$(expr_f\ expr_1\ \dots\ expr_n)$

Function application: apply the function resulting from the evaluation of  $expr_f$  with the values of  $expr_1\ \dots\ expr_n$  as arguments.

Arguments are evaluated from left to right.

1 2 3 ...  $n$

Integer constants.

$\#\backslash c$

Character constant (equivalent to an integer constant, see later).

$"string"$

String constant (equivalent to a vector, see later).

# Code example

Function to compute  $x^y$  on integers ( $y$  must be positive):

```
(define pow
  (lambda (x y)
    (if (= 0 y)
        1
        (if (= 0 (% y 2))
            (let ((z (pow x (/ y 2))))
              (* z z))
            (* x (pow x (- y 1))))))))
```

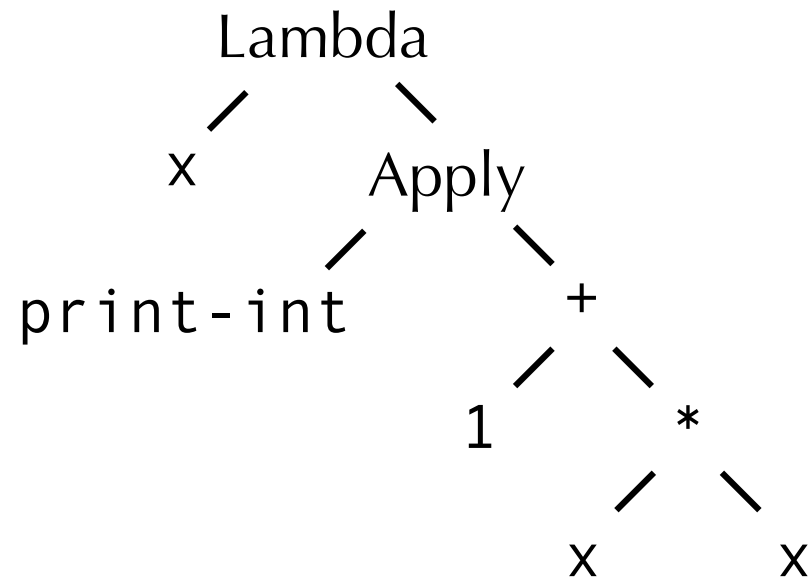


# Grasping the syntax

Minischeme “has no syntax”, in that its concrete syntax is very close to its abstract syntax.

For example, the minischeme expression on the left is an almost direct textual transcription of a pre-order traversal of its AST on the right, in which nodes are parenthesized, while leaves are unadorned.

```
(lambda (x)
  (print-int
    (+ 1 (* x x))))
```



# Syntactic equivalences

The following syntactic equivalences hold. Some of them are used in the compiler to simplify the input program.

$$\begin{array}{l} (\text{let } ((n_1 e_1) \\ \quad (n_2 e_2) \dots) \\ \quad \text{body}) \end{array} \equiv \begin{array}{l} (\text{let } ((n_1 e_1)) \\ \quad (\text{let } ((n_2 e_2)) \dots \\ \quad \text{body}) \end{array}$$
$$\begin{array}{l} (\text{let } ((n_1 e_1)) \\ \quad \text{body}) \end{array} \equiv \begin{array}{l} ((\text{lambda } (n_1) \text{ body}) \\ \quad e_1) \end{array}$$
$$\begin{array}{l} (\text{let } (\dots) \\ \quad b_1 \dots b_n) \end{array} \equiv \begin{array}{l} (\text{let } (\dots) \\ \quad (\text{begin } b_1 \dots b_n)) \end{array}$$
$$\begin{array}{l} (\text{lambda } (\dots) \\ \quad b_1 \dots b_n) \end{array} \equiv \begin{array}{l} (\text{lambda } (\dots) \\ \quad (\text{begin } b_1 \dots b_n)) \end{array}$$

# Primitives

Minischeme is equipped with the following primitives, most of which correspond directly to one VM instruction:

- Arithmetic primitives: `+`, `-`, `*`, `/`, `%`
- Logical primitives: `<`, `<=`, `=`, `>`, `>=`, `not`
- Vector primitives: `vector`, `vector-ref`, `vector-set!`
- Input/output primitives: `read-char`, `print-char`

Primitives are invoked using the syntax of function application, for example: `(* 6 (+ 4 3))`

However, primitives are not functions.

# Vectors

Minischeme provides three primitives to work with vectors (a.k.a. arrays):

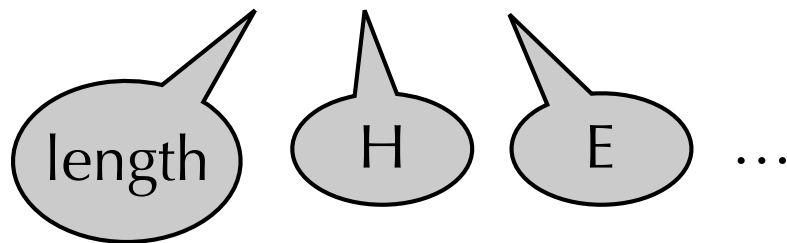
- `(vector e1 ... en)` allocates a vector of  $n$  elements, initialized with the values of  $e_1 \dots e_n$ .
- `(vector-ref v n)` returns the  $n^{\text{th}}$  element of vector  $v$ . Indexing is 0-based.
- `(vector-set! v n e)` sets the  $n^{\text{th}}$  element of vector  $v$  to the value of  $e$ .

# Characters and strings

Minischeme does not offer character and string values, but it offers syntactic sugar for character and string constants.

A character constant is written `#\c` and is translated to the ASCII code of `c`. For example, `#\H` is translated to `72`.

A string constant is written `"string"` and is translated to a vector. The first component of that vector contains the length of the string, while the following ones contain its characters encoded as above. For example, `"HELLO"` is translated to `(vector 5 72 69 76 76 79)`.



# Representing pairs

Pairs can easily be represented using vectors:

`:: construct a pair`

```
(define cons  
  (lambda (f s)  
    (vector f s)))
```

`:: get first component`

```
(define car (lambda (p) (vector-ref p 0)))
```

`:: get second component`

```
(define cdr (lambda (p) (vector-ref p 1)))
```

Note: the names `cons`, `car` and `cdr` are historical.

# Representing lists

Lists can easily be represented using pairs: the first component of the pair contains the head of the list, while the second component contains its tail – another list. The empty list is represented by a special value called `nil`.

This representation of lists by pairs is used in most functional languages: Scheme, Haskell, OCaml, Scala, etc.

For example, the list 1,2,3,4 can be constructed by the following code:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

and its second element can be accessed by the following code, where `lst` represents the list:

```
(car (cdr lst))
```

# The minivm virtual machine



# minivm

Minivm is a virtual machine designed for this project. Its main characteristics are:

- it is register-based: there are 32 general-purpose registers  $R_0 \dots R_{31}$ , and a program counter PC,
- it is very simple, with only 18 instructions,
- it accepts textual assembly code as input.

The design goals were:

- to have a simple, easy to implement machine,
- to have it resemble a real processor, to make the compiler realistic.

However, this machine is definitely not an ideal target for a Scheme compiler!

# Word size

The size of the various minimum storage elements (registers and memory blocks) is defined in terms of an abstract **word size**, expressed in bytes.

The word size of a given minimum instance is the size of a pointer in the architecture for which this instance was compiled. For example, it is 4 for a 32-bits architecture, and 8 for a 64-bits architecture.

Registers can contain exactly one word each, while memory blocks can contain an arbitrary number of words, depending on their capacity.

# Memory model

The memory of minivm is split in two parts:

1. the bottom one contains the code,
2. the top one contains the heap.

A block of heap memory can be allocated using the `ALOC` instruction. A block is an array of words placed consecutively in memory. The capacity of a block is the number of words it can contain.

There is no instruction to free a block, which means that a garbage collector is needed to run realistic programs.

# Instruction set

Minivm instruction set can be categorized as follows:

- Arithmetic: ADD, SUB, MUL, DIV, MOD
- Control: JLT, JLE, JEQ, JNE, JGE, JGT, HALT
- Memory: ALOC, LOAD, STOR, LINT
- Input/output: RCHR, PCHR

# Arithmetic instructions

ADD  $R_a R_b R_c$        $R_a \leftarrow R_b + R_c$

SUB  $R_a R_b R_c$        $R_a \leftarrow R_b - R_c$

MUL  $R_a R_b R_c$        $R_a \leftarrow R_b * R_c$

DIV  $R_a R_b R_c$        $R_a \leftarrow R_b / R_c$

MOD  $R_a R_b R_c$        $R_a \leftarrow R_b \bmod R_c$

# Control instructions

JLT  $R_a R_b R_c$       if  $R_b < R_c$  then  $PC \leftarrow R_a$

JLE  $R_a R_b R_c$       if  $R_b \leq R_c$  then  $PC \leftarrow R_a$

JEQ  $R_a R_b R_c$       if  $R_b = R_c$  then  $PC \leftarrow R_a$

JNE  $R_a R_b R_c$       if  $R_b \neq R_c$  then  $PC \leftarrow R_a$

JGE  $R_a R_b R_c$       if  $R_b \geq R_c$  then  $PC \leftarrow R_a$

JGT  $R_a R_b R_c$       if  $R_b > R_c$  then  $PC \leftarrow R_a$

HALT                      halt virtual machine

# Memory instructions

LINT  $R C$

$R \leftarrow C$

integer or label

LOAD  $R_a R_b R_c$

$R_a \leftarrow \text{Mem}[R_b + w * R_c]$

STOR  $R_a R_b R_c$

$\text{Mem}[R_b + w * R_c] \leftarrow R_a$

ALOC  $R_a R_b$

$R_a \leftarrow$  new block of  $R_b$  words

$w$  is the word size in bytes, memory is byte-addressable

# I/O instructions

RCHR  $R$                        $R \leftarrow$  read character from input

PCHR  $R$                       print char ( $R$ ) on output



# Implementation

You will be given a C implementation of `minivm`, with the following limitations:

- heap memory is never freed, and the VM exits when all available memory has been used,
- not as efficient as it could be.

Part of your job will be to improve it!

# Implementation overview

The implementation is composed of the following three main modules (C files):

- **loader**: parses textual assembly files and calls functions in the engine module to emit the corresponding instructions,
- **engine**: produces the representation of the program in memory, based on instructions from the loader, and executes it later,
- **memory**: allocates memory used to store the program and the data used by it.

# Code example

```
fact: LINT R2 else      ret:  LINT R3 2
      JEQ  R2 R1 R0    LOAD  R2 R30 R3
      LINT R2 3        MUL   R1 R1 R2
      ALOC R2 R2      LINT  R3 1
      STOR R30 R2 R0  LOAD  R2 R30 R3
      LINT R3 1       LOAD  R30 R30 R0
      STOR R29 R2 R3  JEQ   R2 R0 R0
      LINT R3 2      else:  LINT R1 1
      STOR R1 R2 R3  JEQ   R29 R0 R0
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JEQ  R2 R0 R0
```

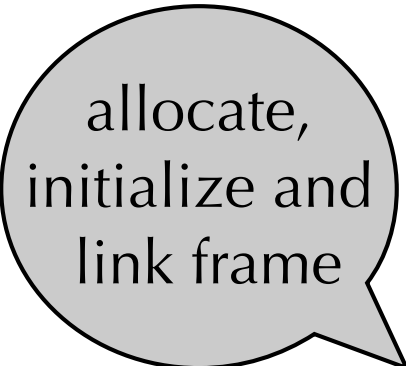
Note: R0 contains 0.

# Code example

```
fact: LINT R2 else
      JEQ  R2 R1 R0
```

```
LINT R2 3
ALOC R2 R2
STOR R30 R2 R0
LINT R3 1
STOR R29 R2 R3
LINT R3 2
STOR R1 R2 R3
ADD  R30 R2 R0
```

```
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JEQ  R2 R0 R0
else: LINT R1 1
      JEQ  R29 R0 R0
```



allocate,  
initialize and  
link frame

```
LINT R2 1
SUB  R1 R1 R2
LINT R29 ret
LINT R2 fact
JEQ  R2 R0 R0
```

Note: R0 contains 0.

# Code example

```
fact: LINT R2 else
      JEQ  R2 R1 R0
```

```
LINT R2 3
ALOC R2 R2
STOR R30 R2 R0
LINT R3 1
STOR R29 R2 R3
LINT R3 2
STOR R1 R2 R3
ADD  R30 R2 R0
```

allocate,  
initialize and  
link frame

```
LINT R2 1
SUB  R1 R1 R2
LINT R29 ret
LINT R2 fact
JEQ  R2 R0 R0
```

perform  
recursive  
call

```
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JEQ  R2 R0 R0
else: LINT R1 1
      JEQ  R29 R0 R0
```

Note: R0 contains 0.

# Code example

compute  
result

fact: LINT R2 else  
JEQ R2 R1 R0

ret: LINT R3 2  
LOAD R2 R30 R3  
MUL R1 R1 R2

allocate,  
initialize and  
link frame

LINT R2 3  
ALOC R2 R2  
STOR R30 R2 R0  
LINT R3 1  
STOR R29 R2 R3  
LINT R3 2  
STOR R1 R2 R3  
ADD R30 R2 R0

else: LINT R1 1  
JEQ R29 R0 R0

perform  
recursive  
call

LINT R2 1  
SUB R1 R1 R2  
LINT R29 ret  
LINT R2 fact  
JEQ R2 R0 R0

Note: R0 contains 0.

# Code example

```
fact: LINT R2 else
      JEQ  R2 R1 R0
```

```
LINT R2 3
ALOC R2 R2
STOR R30 R2 R0
LINT R3 1
STOR R29 R2 R3
LINT R3 2
STOR R1 R2 R3
ADD  R30 R2 R0
```

```
LINT R2 1
SUB  R1 R1 R2
LINT R29 ret
LINT R2 fact
JEQ  R2 R0 R0
```

```
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
```

```
LINT R3 1
LOAD R2 R30 R3
LOAD R30 R30 R0
JEQ  R2 R0 R0

else: LINT R1 1
      JEQ  R29 R0 R0
```

compute  
result

allocate,  
initialize and  
link frame

perform  
recursive  
call

unlink frame  
and return

Note: R0 contains 0.

# The minischeme interpreter and compiler



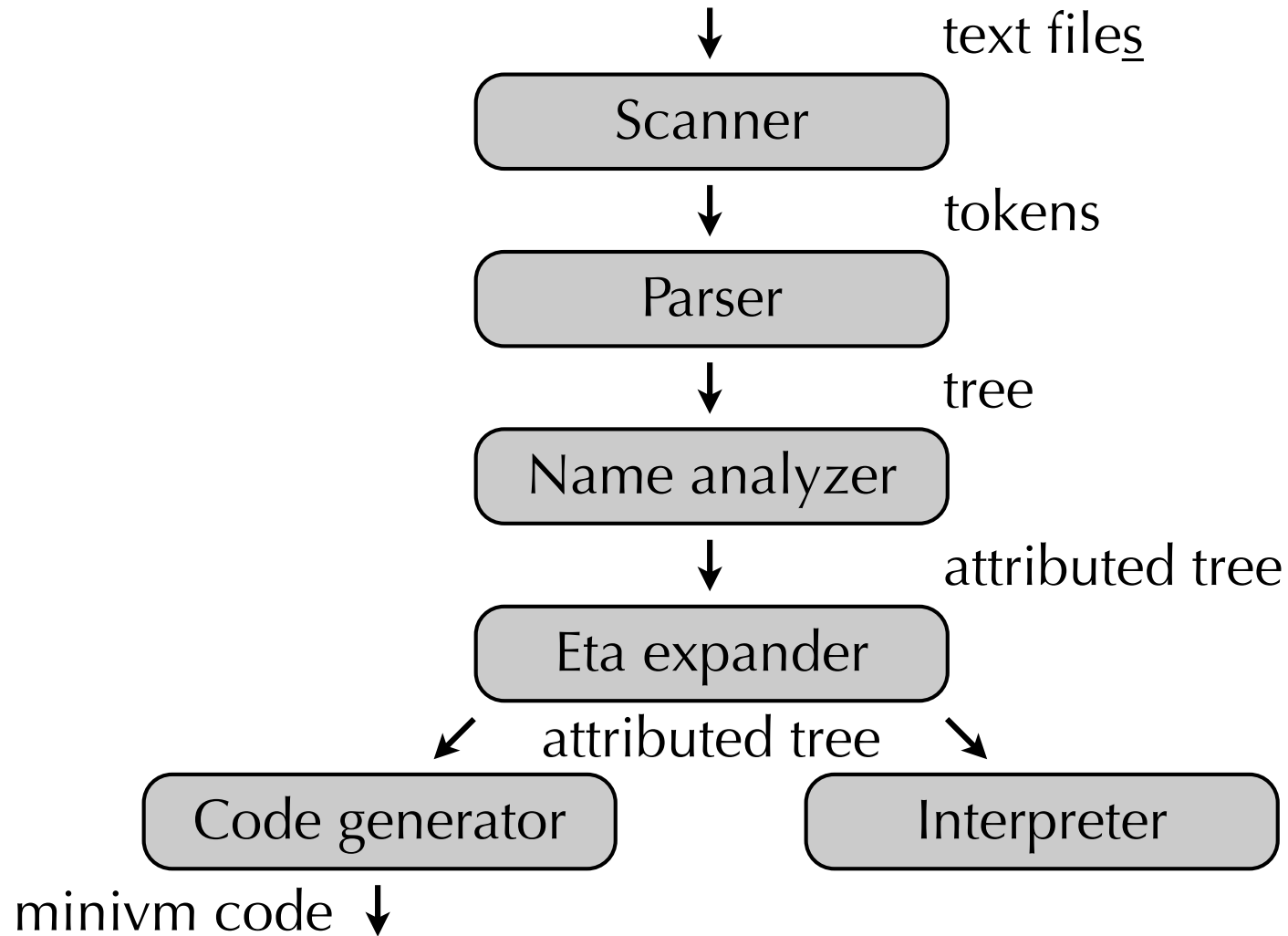
# Interpreter and compiler

You will be given a Scala implementation of a minischeme interpreter and compiler. The interpreter implements the full language, but the compiler has the following limitations:

- functions cannot refer to local values defined in an enclosing scope,
- no code is produced to perform dynamic checks, which means that most type errors or incorrect array indexing result in a VM crash (!),
- the produced code is not very good.

Your job will be to remove some of these limitations later.

# Organization



# Eta-expansion

The eta-expansion phase transforms the code to ensure that primitives are always applied to as many arguments as they expect. This guarantee simplifies later phases of the compiler.

This guarantee is obtained by wrapping primitives inside anonymous functions whenever they are used as values. For example, the following code:

```
(map not 1)
```

is transformed to:

```
(map (lambda (x1) (not x1)) 1)
```

Limitation: the `vector` primitive accepts a variable number of arguments, and since minischeme does not provide functions with variable number of arguments, it cannot be used as a value.

# Register usage

The compiler assigns specific roles to the following registers:

- $R_0$  – holds the constant 0,
- $R_{29}$  – holds the return address (LK),
- $R_{30}$  – holds the stack frame (FP),
- $R_{31}$  – holds the vector of global variables (GP).

Notice that these conventions are in no way enforced by the VM itself!

# Calling conventions

Function arguments are passed in registers  $R_1 \dots R_{29}$ .

Functions with more than 28 arguments are currently not supported. They could easily be supported by packing some arguments in a vector, though.

The return value is put in  $R_1$ .

Register  $R_{30}$  (FP) is callee-saved,  $R_1 \dots R_{29}$  are caller-saved.

# Stack

Stack frames are allocated from the heap, and a pointer to the stack frame of the currently-executing function is stored in  $R_{30}$  (*a.k.a.* the frame pointer FP).

The stack frame of a function  $f$  contains:

- the frame pointer of the function that called  $f$ ,
- the return address, saved from  $R_{29}$  (LK),
- the arguments passed to  $f$ , which are saved on the stack at function entry,
- all the local variables of  $f$ .

# Stack

Usually, the stack is a contiguous area of memory. In minivm, as we have seen, this is not the case.

The stack in minivm is a singly linked list of blocks: the head of that list is stored in the FP register, and each block but the last contains a pointer to its successor, in the form of the saved caller's FP.

