Minischeme project

Michel Schinz & Iulian Dragos

2006-03-17

The project

What you get:

- a compiler for minischeme, written in Scala,
- a virtual machine (VM), written in C.

What you have to do:

- improve the compiler and the VM, *e.g.* by adding a garbage-collector and various optimisations.

The minischeme language

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

- untyped language,
- almost no side effects (one exception: I/O),
- functional: functions are first-class values,
- very simple: four keywords (`define`, `let`, `lambda` and `if`).

The minischeme language

`(define` *name expr*`)`

Global definition, only valid at top level. All global values are visible everywhere, but are initialised in written order.

`(let ((`*name*$_1$ *expr*$_1$`) …)` *body*$_1$ `…)`

Local value(s) definition: *name*$_1$ … *name*$_n$ are visible in *body*$_1$ … *body*$_m$, but **not** in *expr*$_1$ … *expr*$_n$.

The minischeme language

`(lambda (`*name*$_1$ …`)` *expr*$_1$ …`)`

Anonymous function definition.

`(if` *expr*$_{cond}$ *expr*$_{then}$ *expr*$_{else}$`)`

Conditional: evaluate *expr*$_{else}$ iff *expr*$_{cond}$ evaluates to 0, otherwise evaluate *expr*$_{then}$.

`(`*expr*$_{fun}$ *expr*$_1$ …`)`

Function application: call *expr*$_{fun}$ with *expr*$_1$ … *expr*$_n$ as arguments.

Minischeme example

Function to compute x^y on integers:

```
;; raise x to the power of y
(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)))))))
```

Minischeme primitives

Minischeme is equipped with a set of primitives. They are meant to be used to write "predefined" Scheme functions.

All primitives have a name starting with a dollar sign, *e.g.* `$print-int`.

Primitives are invoked using the syntax of a normal function call, but it is important to understand that primitives are **not** functions!

Minischeme primitives

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

- Arithmetic primitives: `$+`, `$-`, `$*`, `$/`, `$%`
- Logical primitives: `$<`, `$<=`, `$=`
- Array primitives: `$alloc`, `$set`, `$get`
- I/O primitives: `$read-int`, `$print-int`, `$read-char`, `$print-char`

Minischeme primitives

These primitives can for example be used to define the three basic operations on cells:

```
(define cons
  (lambda (f s)
    (let ((p ($alloc 2)))
      ($set p 0 f) ($set p 1 s) f)))
(define car (lambda (p) ($get p 0)))
(define cdr (lambda (p) ($get p 1)))
```

construct a cell

get first component

get second component

Syntactic sugar

The minischeme compiler defines some syntactic sugar for strings, translated to lists of integers: each character of the string is represented by its ASCII code.

For example, “`Hello`” is translated to `(cons 72 (cons 101 (cons 108 (cons 108 (cons 111 0)))))`

You will also add syntactic sugar for `and` and `or`.

The minivm virtual machine

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

- register-based,
- very simple (17 instructions),
- accepts text as input.

Minivm design goals

Minivm was designed to be:

- simple, and therefore easy to implement,
- relatively close to real processors, to make the compiler “interesting”.

It is certainly **not** the best design for a Scheme virtual machine!

Minivm registers

Minivm has 32 registers, named `R0`…`R31`. Only `R31` is special: it is the program counter (`PC`).

In the project, we will assign specific roles to:

- `R0` – holds the constant 0,
- `R28` – holds the return address (`LK`),
- `R29` – points to the current stack frame (`FP`),
- `R30` – points to the global variables area (`GP`).

Minivm memory management

Memory is composed of two areas:

1. the code area, containing the instructions making up the program, and
2. the heap, from which blocks can be allocated dynamically.

In particular, notice that there is **no** stack: “stack frames” are allocated in the heap, and linked together explicitly.

Minivm instructions

The minivm instruction set can be categorised as follows:

- Arithmetic: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`
- Control: `ISLT`, `ISLE`, `ISEQ`, `CMOV`
- Memory: `ALOC`, `LOAD`, `STOR`, `LINT`
- Input/output: `RINT`, `PINT`, `RCHR`, `PCHR`

Minivm
arithmetic instructions

`ADD` R_1 R_2 R_3	$R_1 \leftarrow R_2 + R_3$
`SUB` R_1 R_2 R_3	$R_1 \leftarrow R_2 - R_3$
`MUL` R_1 R_2 R_3	$R_1 \leftarrow R_2 * R_3$
`DIV` R_1 R_2 R_3	$R_1 \leftarrow R_2 / R_3$
`MOD` R_1 R_2 R_3	$R_1 \leftarrow R_2 \bmod R_3$

Minivm
control instructions

`ISLT` R_1 R_2 R_3	$R_1 \leftarrow R_2 < R_3$ [false: 0, true: 1]
`ISLE` R_1 R_2 R_3	$R_1 \leftarrow R_2 \leq R_3$ [false: 0, true: 1]
`ISEQ` R_1 R_2 R_3	$R_1 \leftarrow R_2 = R_3$ [false: 0, true: 1]
`CMOV` R_1 R_2 R_3	if $R_3 = 0$ then $R_1 \leftarrow R_2$

Minivm memory instructions

`LINT` R_1 C	$R_1 \leftarrow C$
`LOAD` R_1 R_2 C	$R_1 \leftarrow$ `Mem`[R_2 + C]
`STOR` R_1 R_2 C	`Mem`[R_2 + C] $\leftarrow R_1$
`ALOC` R_1 R_2	$R_1 \leftarrow$ new block of R_2 bytes

Minivm
Input/output instructions

`RINT` R	$R \leftarrow$ read integer from input
`PINT` R	print R on output
`RCHR` R	$R \leftarrow$ read character from input
`PCHR` R	print `char(`R`)` on output

Minivm calling conventions

Arguments are passed in registers $R_1 \ldots R_{27}$.

Functions with more than 27 (26, actually) arguments are not supported yet, but they easily could be.

The return value is put in R_1.

Minivm code example

```
fact:   LINT R2 else
        CMOV R31 R2 R1
        LINT R2 12
        ALOC R2 R2
        STOR R29 R2 0
        CMOV R29 R2 R0
        STOR R28 R29 4
        STOR R1 R29 8
        LINT R2 1
        SUB  R1 R1 R2
        LINT R27 fact
        LINT R28 ret
        CMOV R31 R27 R0
```

link and initialise frame

call fact recursively

```
ret:    LOAD R2 R29 8
        MUL  R1 R1 R2
        LOAD R28 R29 4
        LOAD R29 R29 0
        CMOV R31 R28 R0
else:   LINT R1 1
        CMOV R31 R28 R0
```

unlink frame

return

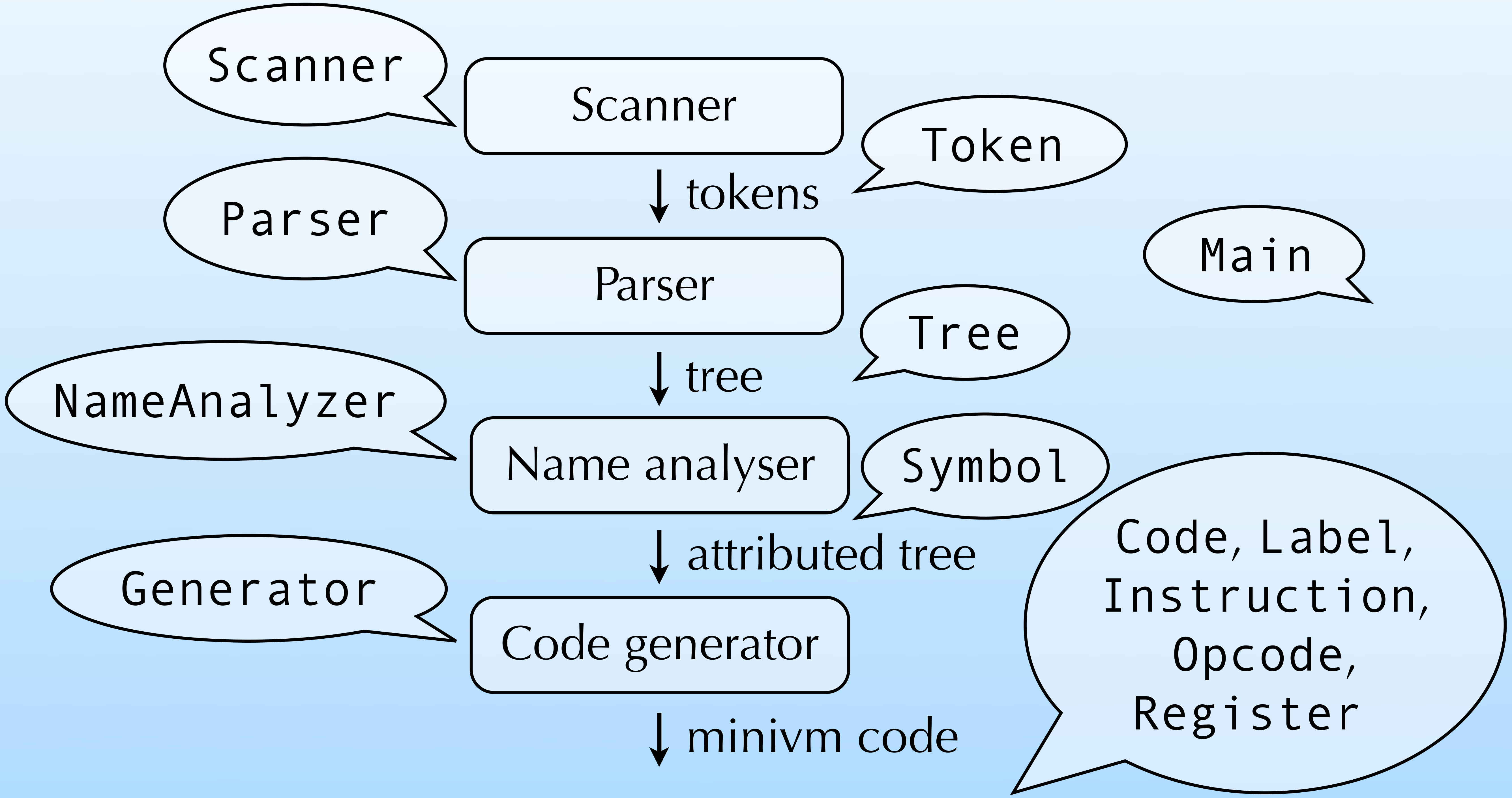
The minischeme compiler

We give you a working implementation (in Scala) of a minischeme compiler, with the following limitations:

- anonymous functions are only allowed at the top-level (*i.e.* no closures),
- the produced code is not very good.

Your job will be to remove those limitations (and others) later.

Minischeme compiler organisation



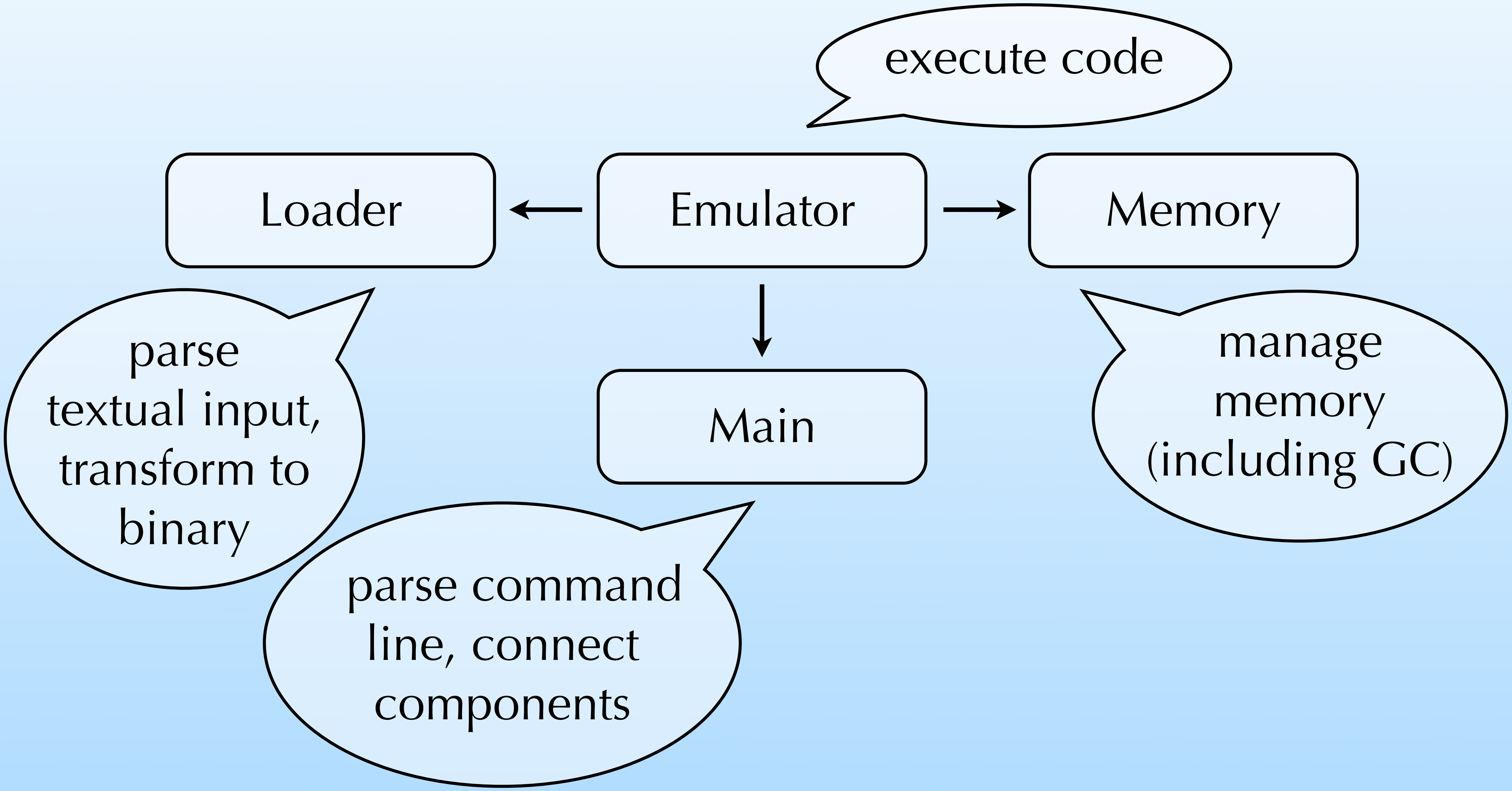
The minivm

We give you a working implementation (in C) of minivm, with the following limitations:

- no garbage collector: memory is never freed, and the VM exits when all available memory has been used,
- not as efficient as it could be.

Once again, your job will be to improve it!

minivm organisation



execute code
Loader
Emulator
Memory
parse textual input, transform to binary
Main
manage memory (including GC)
parse command line, connect components

minivm overview

The loader parses assembler files, resolve labels and produces a binary version of the program; that binary version is accessed by the emulator.

The emulator interprets the program. It can run in interactive mode, where it waits for user input after each step.

The memory manager allocates and reclaims (rather, *will* reclaim) memory in the heap area.

Project overview

The project will start with a set of assignments which all groups will have to complete :

- a small warm-up exercise (not graded),
- a threaded version of the emulator,
- a mark & sweep garbage collector,
- closure conversion,
- tail call elimination.

Project overview

After the assignments, every group will have to choose and complete one advanced project:

- a precise, copying garbage collector,
- a JIT compiler for the virtual machine,
- advanced optimisations,
- a linear-scan register allocator,
- etc.

Project evaluation

At the end of each assignment, you will have to send us your code electronically (using moodle).

At the end of the advanced project, you will have to present your work either through a small written report, or a short oral presentation (depending on the number of students attending the course).