Minischeme project

Michel Schinz & Iulian Dragos

2007–03–16

The project

What you get:

- a compiler for minischeme, written in Scala,
- a virtual machine, 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:

- it is untyped – unlike Scheme, which is dynamically typed,
- it has few side effects (exceptions: arrays, input/output),
- it is functional: functions are first-class values,
- it is very simple, with only four keywords (`define`, `let`, `lambda` and `if`).

The minischeme language

`(define` *name* *expr*`)`

Global value definition, binding the value of *expr* to the *name*, only valid at the top level.

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

`(let ((`$name_1$ $expr_1$`) …)` $body_1$ `…)`

Local value(s) definition: $name_1$ is bound to the value of $expr_1$, $name_2$ to the value of $expr_2$, etc. while $body_1$ … is evaluated. The value of the whole expression is the value of $body_m$.

Note: the names $name_{1…n}$ are only visible in $body_{1…m}$, not in $expr_{1…n}$

The minischeme language

`(lambda` (*name*$_1$ ...) *body*$_1$ ...)

Anonymous function, with parameters *name*$_1$ ... *name*$_n$ and body *body*$_1$ ... *body*$_m$.

`(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 (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)))))))
```

Minischeme primitives

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

- Arithmetic primitives: `+`, `-`, `*`, `/`, `%`
- Logical primitives: `<`, `<=`, `=`
- Vector primitives: `vector`, `vector-ref`, `vector-set!`
- Input/ouput primitives: `read-int`, `print-int`, `read-char`, `print-char`

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

However, it is important to understand that primitives are *not* functions. In particular, primitives cannot be manipulated as values, while functions can.

Eta-expansion

Since primitives cannot be manipulated as values, the following definition should in principle not be accepted:

```
(define plus +)
```

However, the minischeme compiler performs a transformation known as eta-expansion to transform the above code into the following, legal one:

```
(define plus (lambda (a1 a2) (+ a1 a2)))
```

In summary, the aim of eta-expansion is that whenever the programmer tries to use a primitive as a value, that primitive is replaced by an equivalent anonymous function. This guarantees that primitives are never used as values.

Minischeme vectors

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

- `(vector` e_1 … e_n`)` creates a vector of *n* elements, initialised with the values of e_1 … e_n.
- `(vector-ref` *v* *n*`)` returns the *n*th element of *v*. Indexing is 0-based, and no bounds checking is done!
- `(vector-set!` *v* *n* *e*`)` sets the *n*th element of *v* to the value of *e*.

Notice that `vector` accepts a variable number of expressions. Since minischeme does not provide the concept of functions with a variable number of parameters, it is the only primitive that cannot be eta-expanded.

Pairs in minischeme

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.

Lists in minischeme

Lists can easily be represented using pairs: the first component of the pair represents the head of the list, and the second component represents its tail, which is another list. The empty list is represented by 0.

This representation of lists by pairs is used in most functional languages.

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

```
(cons 1 (cons 2 (cons 3 (cons 4 0))))
```

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

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

Characters and strings

The minischeme compiler defines some syntactic sugar for characters and strings.

A character *c* is written `#\`*c* and is translated to the ASCII code of *c*. For example, `#\A` is translated to 65.

A string *s* is written `"`*s*`"` and is translated to the list of the ASCII codes of its characters. For example, `"Hello"` is translated to:

```
(cons 72
  (cons 101
    (cons 108
      (cons 108
        (cons 111 0)))))
```

The minivm virtual machine

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

- it is register-based,
- it is very simple, with only 17 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!

Minivm registers

Minivm has 32 general-purpose registers, named `R0`…`R31`, and a program counter (`PC`).

In the project, we will assign specific roles to:

- `R0` – holds the constant 0,
- `R29` – holds the return address (`LK`),
- `R30` – points to the current stack frame (`FP`),
- `R31` – points to the global variables area (`GP`), containing all global values.

Notice that these are just conventions used by the compiler, that are in no way enforced by the VM itself!

Calling conventions

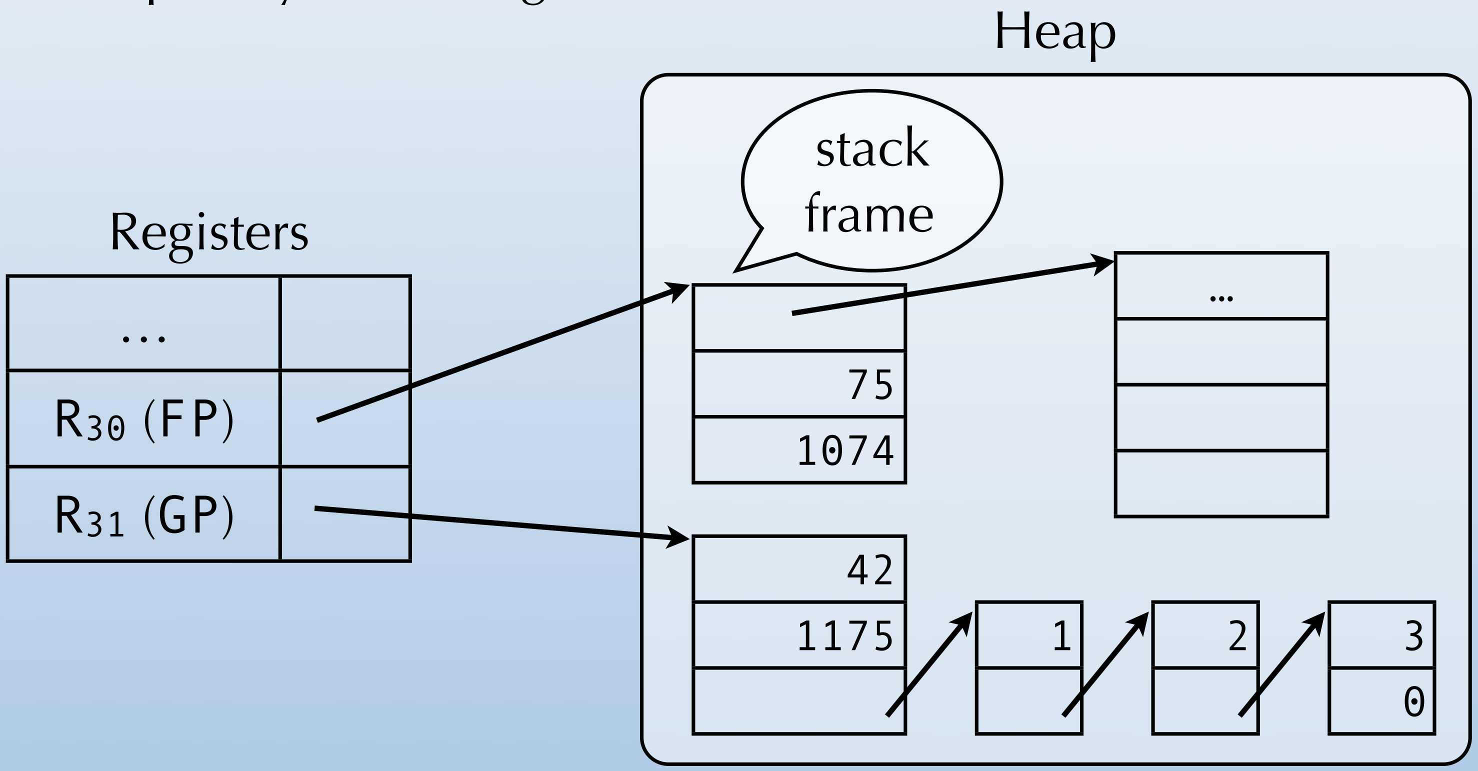
Function arguments are passed in registers R1…R28.

Functions with more than 28 – 27, actually – arguments are not supported yet. They could be supported by passing some of the arguments on the stack, though.

The return value is put in R1.

Memory organisation

All memory used by programs is dynamically allocated from a single heap. In other words, even stack frames used to store local variables are allocated from the heap, and explicitly linked together.



Minivm instructions

The minivm instruction set can be categorised as follows:

- Arithmetic: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`
- Control: `ISLT`, `ISLE`, `ISEQ`, `JMPZ`
- Memory: `ALOC`, `LOAD`, `STOR`, `LINT`
- Input/output: `RINT`, `PINT`, `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	$R_a \leftarrow R_b \text{ mod } R_c$

Control instructions

`ISLT` R_a R_b R_c	$R_a \leftarrow R_b < R_c$ [false: 0, true: 1]
`ISLE` R_a R_b R_c	$R_a \leftarrow R_b \leq R_c$ [false: 0, true: 1]
`ISEQ` R_a R_b R_c	$R_a \leftarrow R_b = R_c$ [false: 0, true: 1]
`JMPZ` R_a R_b	if $R_b = 0$ then `PC` $\leftarrow R_a$

Memory instructions

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

I/O 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 code example

```
fact: LINT R2 else
      JMPZ R2 R1
      LINT R2 12
      ALOC R2 R2
      STOR R30 R2 0
      STOR R29 R2 4
      STOR R1 R2 8
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
```

allocate, initialise and link frame

perform recursive call

```
ret:  LOAD R2 R30 8
      MUL  R1 R1 R2
      LOAD R2 R30 4
      LOAD R30 R30 0
      JMPZ R2 R0
else: LINT R1 1
      JMPZ R29 R0
```

compute result

unlink frame and return

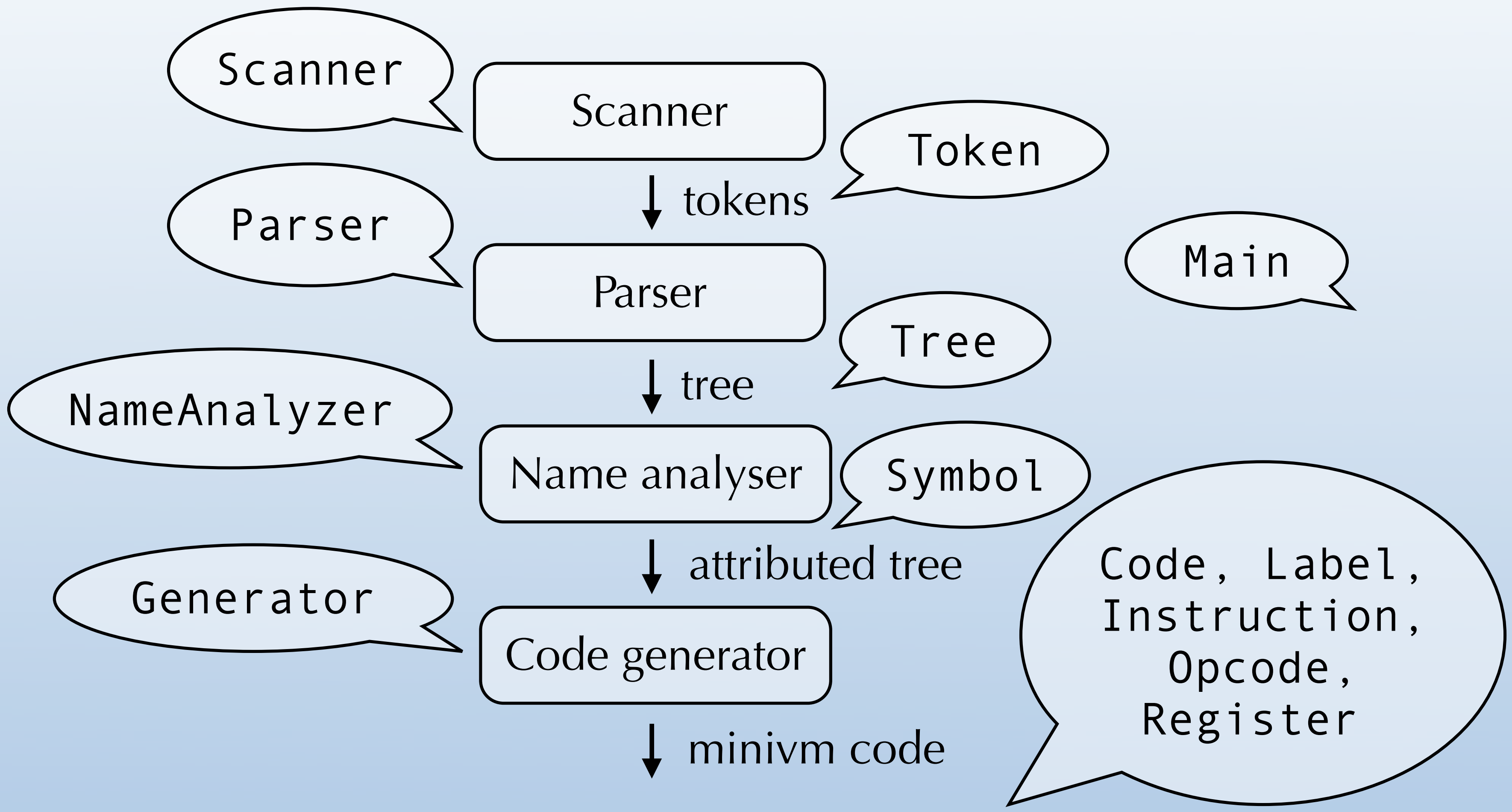
The minischeme compiler

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 these, and other, limitations later.

Compiler organisation



Minivm implementation

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 overview

The parser analyses assembler files, resolves labels and produces a binary version of the program in memory; that binary version is accessed by the emulator.

The emulator interprets the program. It can run interactively, and wait 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 :

- two small warm-up exercises (not graded),
- a “mark-and-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 grading

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.