DAIMI-Scheme VML
File Formats

This document describes the file formats of the assembly files output by the DAIMI-Scheme compilers, and of the binary files read by the DAIMI-Scheme virtual machines.

DAIMI-Scheme assembler files have the filename extension `.dsa` (DAIMI-Scheme Assembly); binary DAIMI-Scheme files have the extension `.dsb` (DAIMI-Scheme Binary).

Part I
Assembly Language

1 Lexicographics

A few general notes:

Whitespace is ignored. Comments are Scheme-like, i.e. everything from a semicolon (which is not part of a string) to the end of the line is regarded as a comment and therefore ignored by the assembler.

Strings are delimited by double quotes (`"`) and may contain any character except double quote and newline. Backslashes are interpreted as escape characters in the usual way (among the valid escape sequences are `\\`, `\"`, `\n`, `\t` and `\r`, with the usual interpretations).

2 Grammar

The grammar of an assembly program:

`<program>`	::=	`( <magic-word> <limits> <lambda-table> <code> <signature> )`
`<id>`	::=	*a valid DAIMI-Scheme identifier*
`<string>`	::=	*a string, as specified above*
`<number>`	::=	*a decimal, signed integer*
`<magic-word>`	::=	`DAIMI-SchemeE03`
`<signature>`	::=	`<string>`
`<limits>`	::=	`( <max-glo> <max-tmp> <max-res> )`
`<max-`*xxx*`>`	::=	`<number>`
`<lambda-table>`	::=	`( <lambda-entry>`* `)`
`<lambda-entry>`	::=	`( <arity> <label> )`
`<arity>`	::=	`<number>`
`<code>`	::=	`( <instruction>`* `)`

```
<instruction>  ::=  ( nop )
                |   ( label <label> )
                |   ( load <data> <location> )
                |   ( move <location> <location> )
                |   ( new-vec <number> )
                |   ( extend )
                |   ( jump <label> )
                |   ( jump-if-false <location> <label> )
                |   ( tail-call <location> )
                |   ( call <location> <number> )
                |   ( return )
<label>        ::=  <id>
<location>     ::=  <scope> <number>
<scope>        ::=  lib | glo | res | tmp | vec | <number>
<data>         ::=  nil _
                |   bool <number>
                |   int <number>
                |   char <number>
                |   str <string>
                |   sym <id>
                |   close-flat <number>
                |   close-deep <number>
                |   void _
```

The arguments of the instruction have the same order and meaning as in the VM specification.

The three numbers in `<limits>` are, respectively, the number of global variables, the maximal number of temporary variables, and the maximal number of result values. That is, they are the lengths of the vectors to be allocated for `env-glo`, `env-tmp` and `aux-res`.

Note that the strings and symbols are specified directly in the code (rather than in a seperate table), while the lambda abstractions are placed in the table and referred to by their number in the table (starting with 0).

3 Example

A sample program:

```
(DAIMI-SchemeE03
  (2 0 1)
  ((0 f)
   (0 g))
  ((new-vec 0)
   (load close-flat 0 glo 0) ; Initalize f
   (new-vec 0)
   (load close-flat 1 glo 1) ; Initalize g
   (new-vec 0)               ; Args for f
   (call glo 0 0)    ; Call f
   (new-vec 0)
   (tail-call glo 1)    ; Call g
```

```
   (label f)
   (label g)
   (new-vec 1)
   (load str "Hello World!" vec 0)
   (tail-call lib 23))  ; A tail-call (to 'string->symbol') - return to caller
 "My Favorite Scheme Compiler")
```

Part II
Binary File Format

The following is the specification of the file format of DAIMI-Scheme binaries.

NB: Because the various numbers are stored in a fixed-size format in the file, this specification *implicitly* imposes certain restrictions on DAIMI-Scheme programs. For example,

- Lexical scope numbers must be in $[0; 127]$
- Indices scopes must be in $[0; 65535]$

et cetera.

4 File Format

A binary DAIMI-Scheme VML file consists of (in this order):

- Magic word
- Program limits
- String pool
- Symbol pool
- Lambda abstraction table
- Code
- Compiler signature
- Magic word

4.1 Magic Word

The binary file begins and ends with "magic word", that is, a fixed and easily recognizable byte sequence, to

a) identify the file as a binary DAIMI-Scheme VML file

b) help the VM detect if a file is malformed, before it tries to execute it

c) in the case of DAIMI-Scheme , also to identify the file as being encoded in little- or big-endian.

The magic word for binary DAIMI-Scheme files is the 32-bit value `0xDA15CE03` (for DA1mi-5Cheme Efteråret 03). The endian-ness of this value in the file corresponds to the endian-ness of the multi-byte integer values in the rest of the

file (which is in most cases the endian-ness of the machine on which the file was compiled).

Thus, in little-endian files, the byte sequence of the magic word is `0x03 0xCE 0x15 0xDA`, while in big-endian files it is `0xDA 0x15 0xCE 0x03`.

For further safety, some sections will end with another magic value. That value will be called `EOS` (end of section) and consists of a single byte with the value `0x80` (128 – chosen for its infrequent occurrence in the rest of the file).

4.2 Program Limits

The VM will need to know how much space to allocate for `env-glo`, `env-tmp` and `aux-res`. This section of the file contains this information.

It has the following stucture:

`<limits>` ::= `<max-glo> <max-tmp> <max-res>`
`<max-`*xxx*`>` ::= *16-bit unsigned integer*

4.3 String Pool

Structure of the string pool:

`<string-pool>` ::= `<string-count> <string-entry>`* `<EOS>`
`<string-entry>` ::= `<string-length>` *ascii-char**
`<string-count>` ::= *32-bit unsigned integer*
`<string-length>` ::= *32-bit unsigned integer*

The string entries are indexed from zero.

Remember that strings may contain any ASCII characters – including the NULL character.

4.4 Symbol Pool

The VM has a symbol table, in which all loaded and generated symbols are placed – to ensure that symbols with the same string representation are always the same object.

This is why string and symbols are placed in two different sections in the binary file: Symbols are inserted in the symbol table, strings are not. Otherwise, there is little difference between the two.

Structure of the symbol pool:

`<symbol-pool>` ::= `<symbol-count> <symbol-entry>`* `<EOS>`
`<symbol-entry>` ::= `<symbol-length>` *ascii-char**
`<symbol-count>` ::= *32-bit unsigned integer*
`<symbol-length>` ::= *32-bit unsigned integer*

The symbol entries are indexed from zero.

4.5 Lambda Abstraction Table

Structure of the lambda table:

```
<lambda-table> ::= <lambda-count> <lambda-entry>* <EOS>
<lambda-entry> ::= <arity> <lambda-code>
<lambda-count> ::= 32-bit unsigned integer
<lambda-code>  ::= 32-bit unsigned integer
<arity>        ::= 8-bit signed integer
```

The lambda entries are indexed from zero.

The `<arity>` n encodes the number of arguments expected by the function.
If $n >= 0$, it means that the function takes exactly n arguments.
If $n < 0$, it means that the function takes $-(n+1)$ arguments or more.
(Thus, $n \leq -2$ is used for variadic lambdas.)

`<lambda-code>` is the label of the first instruction in the function.

4.6 Code

The code section consists mainly of a sequence of instructions:

```
<code-section> ::= <code-length> <instruction>* EOS
<code-length>  ::= unsigned 32-bit integer
```

`<code-length>` is the sum of the lengths of the instructions, in bytes.

The formats of the instructions (`<instruction>`) are described in Section 5.

Other sections refer to "code labels". The label of an instruction is the byte index in the code section of the first byte of the instruction. Code bytes are indexed from zero.

Note that it is *not* legal to refer to bytes in the middle of an instruction; programs containing such references are illegal and should be rejected by the VM.

4.7 Compiler Signature

The compiler signature is a compiler-dependent string. It is encoded identically to a `string-entry` (see section 4.3 on the page before).

5 Instruction set

For information on the meaning of the instruction and valid values of their arguments, consult the VML specification.

5.1 Instruction Description Template

Each instruction is described like this:

Instruction:	(*assembly code instruction format*)
Opcode:	*The first byte of the instruction*
Arguments:	$a_1\ a_2\ \ldots$ *The sequence of the arguments following the opcode*
	$a_1 : t_1$ *The types of the arguments*
	a_2,$a_3 : t_2$
	$\ldots$
Length:	N bytes $(1+len(a_1) + len(a_2) + \ldots)$

5.2 Types and Values

Argument Types:

scope	a scope number (see below)	
type	a type number (see below)	
index	an unsigned 16-bit value	
label	an unsigned 32-bit value	(see also section 4.6)
size	an unsigned 16-bit value	
data	a data value (see below)	

Scope Numbers:
A scope number is a byte. The legal values are:
`lib` : −1
`glo` : −2
`res` : −3
`tmp` : −4
`vec` : −5
(or a non-negative value for the lexical scope corresponding to that number)

Type Numbers:
A type number is a byte. The legal values are:
0 : nil
1 : bool
2 : int
3 : char
4 : str
5 : sym
6 : close-flat
7 : close-deep
8 : void

Data Values:
A data value is a 32-bit integer. Its interpretation depends on the context – it is generally associated with a type number.

Type name	Data type	Interpretation of value
nil	The empty list	Ignored
bool	Boolean	0 for false, 1 for true
int	Integer	The integer value
char	Character	The ASCII code of the character
str	String	Index in the string table
sym	Symbol	Index in the symbol table
close-flat	A flat closure	Index in the lambda table
close-deep	A deep closure	Index in the lambda table
void	The void value	Ignored

In all cases except int, the value is unsigned. Table indices should be checked against the size of the table in question.

5.3 The Instructions

Instruction: (label the-label)
Opcode: None - it is simply a place in the code.
Length: 0 bytes

Instruction: (nop)
Opcode: 0
Arguments: -
Length: 1 byte

Instruction: (load V x T j)
Opcode: 1
Arguments: V x T j
V : type
x : data
T : scope
j : index
Length: 9 bytes (1+1+4+1+2)

Instruction: (move S i T j)
Opcode: 2
Arguments: S i T j
S,T : scope
i,j : index
Length: 7 bytes (1+1+2+1+2)

Instruction: (new-vec n)
Opcode: 3
Arguments: n
n : size
Length: 3 bytes (1+2)

Instruction: (extend)
Opcode: 4
Arguments: -
Length: 1 byte

Instruction: (jump L)
Opcode: 5
Arguments: L
L : label
Length: 5 bytes (1+4)

Instruction: (jump-if-false Q i L)
Opcode: 6
Arguments: Q i L
Q : scope
i : index
L : label
Length: 8 bytes (1+1+2+4)

Instruction: (tail-call Q i)
Opcode: 7
Arguments: Q i
Q : scope
i : index
Length: 4 bytes (1+1+2)

Instruction: (call Q i n)
Opcode: 8
Arguments: Q i n
Q : scope
i : index
n : size
Length: 6 bytes (1+1+2+2)

Instruction: (return)
Opcode: 9
Arguments: -
Length: 1 byte

6 Builtin Library Functions

The built-in procedures are enumerated as they occur in the DAIMI-Scheme specification. The list is repeated here for convenience:

Integers:
0 integer?
1 +
2 −
3 *
4 quotient
5 remainder
6 <
7 <=
8 =
9 >=
10 >

Booleans:
11 boolean?

Symbols:
12 symbol?

Characters:
13 char?
14 char–>integer
15 integer–>char

Strings:
16 string
17 make-string
18 string?
19 string-length
20 string-append
21 string=?
22 string-ref
23 string–>symbol
24 symbol–>string

Pairs:
25 pair?
26 cons
27 car
28 cdr
29 set-car!
30 set-cdr!

Lists:
31 null?

Vectors:
32 vector
33 make-vector
34 vector?
35 vector-length
36 vector-ref
37 vector-set!

Procedures:
38 procedure?
39 apply

Misc:
40 eqv?

Control:
41 call/cc
42 exit

Input:
43 open-input-file
44 input-port?
45 close-input-port
46 current-input-port
47 read-char
48 peek-char
49 eof-object?

Output:
50 open-output-file
51 output-port?
52 close-output-port
53 current-output-port
54 write-char