Tamacola - A Meta Language Kit for the Web

Takashi Yamamiya, Yoshiki Ohshima

This paper was presented at the Workshop on Self-sustainign Systems (S3)2010,
The University of Tokyo, Japan, September 27-28, 2010, ACM Digital Library (to be published).

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Technical Report TR-2010-002

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Tamacola — A Meta Language Kit for the Web *

A Report on creating a self-hosting Lisp compiler on the Tamarin VM

Takashi Yamamiya Yoshiki Ohshima

Viewpoints Research Institute
1209 Grand Central Ave., Glendale, CA 91201
takashi@vpri.org yoshiki@vpri.org

Abstract

Tamacola is a dynamic, self-sustaining meta-language system grounded upon the Tamarin VM.[1] Tamacola compiles a Scheme-like S-expression language into ActionScript bytecodes, and contains meta-linguistic features, such as a PEG parser generator and macro system, which make it useful for defining new languages. In fact, Tamacola is written in itself, using its meta-linguistic features.

Since the Tamarin VM can load ActionScript bytecode files to extend and replace running programs, Tamacola can extend itself and define new languages while it is running. Furthermore, since the Tamarin VM is part of the ubiquitous Adobe Flash player, this self-modification can be accomplished while running in a web browser, with no extra installation requirement.

Objects in Tamacola are intimately tied to their ActionScript counterparts, providing good interoperability between Tamacola and the Flash Player. To show that the system is ready for practical use, we used Tamacola to implement both an interactive programming environment ("Workspace") and a simple particle language.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Extensible Languages; D.3.4 [*Programming Languages*]: Translator Writing Systems and Compiler Generators

General Terms Design, Languages

Keywords compiler, self hosting, Adobe Flash

1. Introduction

In the "STEPS" project [10], we have been writing various programming language processing systems and experimenting ideas with them. Among these, Piumarta's COLA system [12][13][14] aims to be the bottommost substrate that serves as the target language for the higher-level languages.

* This material is based upon work supported by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1] The source code of Tamacola is freely available under the MIT license at: http://www.vpri.org/vp_wiki/index.php/Tamacola.

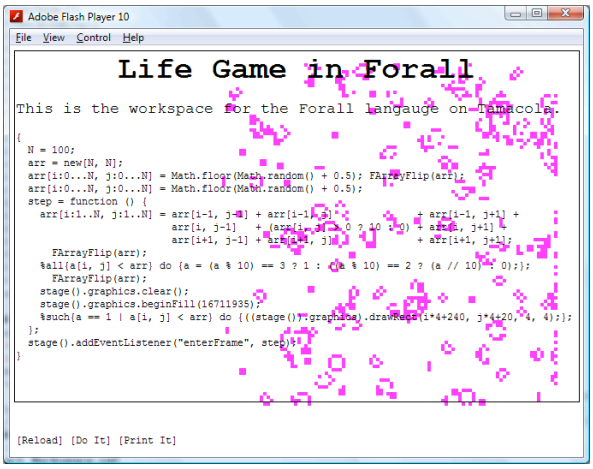

Figure 1. An interactive programming environment application done in Tamacola.

COLA's vision is to provide "Chains of Meanings" that successively transform structures and meaning through various forms to produce (for example) executable machine code as output. COLA provides the meta-linguistic facilities to define new languages. The meta-language of COLA is implemented in itself. It also implements its own execution runtime, so once properly bootstrapped, the entire system can stand on its own.

Our Tamacola work is inspired by COLA; the central idea of Tamacola is that the Chains of Meanings could be "grounded" onto other existing execution environments. Among such execution environments, we found the Tamarin VM [2] very attractive for a few reasons: a) the Tamarin VM is virtually ubiquitous (being used in the Adobe Flash Player, the vast majority of client computers have it); b) the Tamarin VM is open-source and the bytecode specification is well-documented; and c) the Tamarin VM allows dynamic loading of code and replacement of existing functions.

After about one-year of effort, our system (named "Tamacola" as a simple abbreviation of the related technologies) reached the point of being self-sustaining and ready for wider audience. Tamacola runs on top of the Tamarin VM and compiles a program to ActionScript bytecode, which can be loaded into the Tamarin VM. The meta-linguistic features, primarily a PEG parser generator with structural pattern matching, is sufficiently powerful that Tamacola is implemented in itself. Support libraries provide an interface with the Flash Player, so the environment can run in a Web-browser with zero installation.

We hope that Tamacola will enable students, hobbyists and others to try software built for the STEPS project. Also, our group

has been interested in the educational aspect of computing. This implementation could be a good vehicle for deploying dynamic software. Care was taken to keep the system simple (about 10,000 lines of code) and we also believe that the code serves as a good example of Tamacola programming itself.

The contributions of this work are summarized as follows:

- It demonstrates the feasibility of implementing a dynamic self-sustained meta-language system on top of Adobe's Tamarin VM and Flash Player.
- It provides a compact and zero-install-requirement implementation of a self-sustained system that students of programming languages can easily access.
- It illustrates a clean mapping from a higher-order functional language to the execution engine designed for ActionScript.

Section 2 presents the design principles of Tamacola along with the relationship to its parents, namely the COLA system and Scheme programming language. In Section 3, the Tamacola language is explained. In Section 4, the components, or the links in the Chain of Meaning, in Tamacola is described, along with an explanation of the terminology and technology used. In Section 5, the mapping of Tamacola function calls and variables to ActionScript is discussed in detail. Section 6 describes the bootstrap process and how the system becomes self-sustained. In Section 7, an interactive shell environment ("Workspace") is explained as an example. In Section 8, another example, a larger end-user array language called "Forall" is explained. Section 10 discusses related work, and Section 11 concludes.

2. Design of Tamacola

We explain the design principles of Tamacola in this section. In preparation, we first clarify some terminology used with the Tamarin VM and related Adobe Corporation technology.

2.1 Terminology and Technology around Tamarin VM

The Flash Player is a well-known, media-rich application environment that most users of Web-browsers often experience. The standard language for specifying Flash content is called "ActionScript 3"; in this paper, we refer to it simply as "ActionScript". Adobe has implemented a virtual machine called the "Tamarin VM" to run ActionScript, and released it under an open-source license. The Tamarin Project provides a standalone shell program called "avmshell". For developers' convenience, avmshell has a simple I/O API that lets a program running on it access the file system and interactive console.

While the implementation is referred to as the Tamarin VM, the specification of the VM is Adobe Virtual Machine 2 (AVM2) [1]. The input file format for AVM2 is ActionScript Byte Code (ABC). An ABC file (typically with file name ".abc") may contain function definitions (as sequences of bytecode instructions), class definitions, and a constant table.

Shockwave Flash Format (SWF) is the file format for the Flash Player. A SWF file serves as a container for various kinds of media data, such as vector graphics, video, and audio, in addition to ABC data.

While the VM technology used is the same, the Flash Player and avmshell work differently. The Flash Player does not have file I/O and can only load SWF files. Avmshell can load and execute both ABC and SWF from files.

2.2 Design Principles and Goals

Tamacola's primary goal is to provide a meta-linguistic facility for implementing a wide range of programming languages. The facility needs to be powerful enough to implement Tamacola itself. It is also essential for the implementation to be compact and clean, since we would like users to understand the whole system and make necessary extensions and changes without becoming lost in complexity.

Tamacola's design largely draws upon COLA. We also found inspiration from Scheme, a compact and consistent S-expression language, and we incorporated various Scheme syntax, function names, and features.

Four main design goals were:

A Dynamic Self-Supporting Environment Being a meta-language system implemented in itself, the entire system must be accessible to developers. All parts of the compiler must be modifiable and accessible so that adding different features and meta-features dynamically becomes feasible.

A Compact System The meta-implementation approach tends to lead to a compact artifact that does not require any external tools. This encourages developers to understand the whole system. Tamacola is about 10,000 lines of code, including compiler, assembler, test cases, and all libraries.

Heavy Use of PEG with Structural Matching Typically, the compilation of a program is implemented in stages that are pipelined together ("Chain of Meaning"). By mobilizing a powerful PEG parser generator that supports deep structural matching similar to OMeta [17], most such stages can be written using succinct declarative grammar definitions.

Familiar Environment By taking Scheme as an example, library names and syntax are familiar and consistent. Also, as the object and function models are based on those of ActionScript, features provided by Flash are accessible from Tamacola in a straightforward way.

In order to achieve these goals, on a third party's execution engine, certain things are left out. Tamacola does not aim to provide the ultimate flexibility that the COLA system has in mind; Tamacola cannot change the method lookup logic, nor can it change objects' layout in memory, etc. Also, modifying the execution engine is not possible. COLA implements a "kernel" for itself that provides abstract machine instructions, multi-processing, reified activation records, continuations and so on, whereas Tamacola cannot even have multi-processing since the Tamarin VM does not offer such a feature.

There are some Scheme features that are also out of the scope of Tamacola.

Tail Call Elimination Optimization Tamacola does not do tail call optimization, mainly because the AVM2 does not allow the `jump` instruction to jump beyond the enclosing function. This favors the use of loops with destructive assignments over the tail calls for implementing iteration. Note, however, that higher-order functions in the library (such as `fold` in SRFI-1) encapsulate their loops, so the use of such less-than-pure operations can at least be hidden.

Continuations Because the Tamarin VM does not allow direct access to the stack frame, Tamacola cannot create continuations. For error handling, ActionScript-style exception handling is provided.

Variable Arguments for a Function While it is possible to implement optional arguments in the Tamarin VM, the Tamacola language omits this on grounds of simplicity. Note that with the macro system, one can still write functions and special forms that take variable arguments, such as `and` or `cond`. Another workaround is to provide different functions for different argument counts, such as `map` and `map2`.

Hygienic Macros Tamacola's macro is the so-called traditional, non-hygienic macro, where a symbol in the macro definition can capture a variable of the same name in the caller. There are simplicity versus complexity trade-offs here, and again Tamacola opted for simplicity.

3. The Tamacola Language

The Tamacola language is an S-expression language which supports not only functional language features but also the kinds of object-oriented features provided by ActionScript. The basic feature set is confined to things that can be readily translated to ABC code. Such features include: arithmetic and comparison functions, control structures such as function application, `let`, `lambda`, `if`, `while` and `try-catch`, and object-oriented features such as creating a new class, creating a new object, testing types of objects, and object field access. A `define` at the top-level creates a binding in the "global" context for the compilation unit. There is a macro form that is also considered to be a basic feature. As explained in Section 4, these are mostly implemented as simple transformation rules in a PEG specification called `syntax.g`.

Tamacola uses ActionScript objects for its data representation. To access them, a few special language constructs are provided.

Free Variables Tamacola's functions are lexically scoped; that is, a function that refers to free variables "closes-over" these variables. When a function closure outlives the function activation that created it, any free-variables are preserved and still can be found in one of the lexically-enclosing function activations.

"`#self`" When a Tamacola function is used as a method for an ActionScript object, the receiver, or "`this`" in ActionScript, is accessed via a pseudo-variable called "`#self`".

"`#undefined`" Tamacola uses the special name `#undefined` to represent the "`undefined`" value in ActionScript. This is used as the initial value for slots in newly-created objects, and is also used as the return value from expressions with side-effects such as `define` or `set!`.

Numbers, Strings, Booleans Numbers, strings and Booleans are mapped to the corresponding ActionScript objects. A side-effect of this mapping is that unlike Scheme, where only #f is treated as false for conditionals, the number zero (0), an empty string, `'()`, and `#undefined` are all also (in Boolean expressions) treated as equivalent to false; this is the same way that ActionScript's conditional works. Similarly, the `eq?` function is mapped to strict equality `===` in ActionScript; but note that this is not strict enough to tell the difference between a floating point value and an integer; thus:

```
(eq? 1.0 1) => #t
```

Symbols, Pairs Symbols and pairs are ActionScript classes created in the Tamacola library. The null value (namely, `'()` in Tamacola) is the same as `null` in ActionScript.

eval `eval` takes a program in a string as argument and evaluates it in the top-level context. (See Section 4.6 for details of its behavior.)

There are object-oriented features provided to access ActionScript objects and to call the Flash API. With them, for instance, we can build Flash applications entirely within Tamacola.

Classes and Objects There is a form called `class` to create a new class with specified superclass, properties and constructor expressions. The `new` form creates a new object of a given class, and there are forms for testing type inclusion, and to test whether an argument is "`undefined`".

Slot-access There are a few forms to get and set a property of an object. For example, a form:

(`slot-get` *object string-or-number*)

looks up a property of the object (if the object is an Array, a numeric index can be used instead of property name). Similarly, to store a value into a property there is a form `slot-set!`.

Message-Sending The `send` form in:

(`send` *object selector arguments ...*)

invokes the specified method on the object with arguments.

3.1 Macros

Tamacola has two ways to define macros. `define-form` is used to define simple, traditional macros, while `define-pattern` provides macros that can use pattern matching. These macros are not hygienic, and symbols are evaluated in the caller's context.

- (`define-form` *name argument expander*)

  `define-form` transforms *argument* using *expander*. *Expander* is a Tamacola expression which returns an S-expression. *Argument* is expanded by *expander* and embedded at the "call" site. For example, Tamacola's `begin` form is defined as follows:

```
(define-form begin e (cons 'let (cons '() e)))
```

- (`define-pattern` (`template` `pattern`) ...)

  `define-pattern` is a pattern-template based macro similar to `syntax-rules` in Scheme. Any common variables found in both `template` and `pattern` are substituted by actual values in place of the template. The macro name is the `car` value of `template`. For example, Tamacola's `cond` form is defined as follows:

```
(define-pattern
  ((cond)                      ())
  ((cond (c0 . b0) . rest)  (if c0
                                (begin . b0)
                                (cond . rest))))
```

  Unlike `syntax-rule` in Scheme, ellipsis (`...`) is not supported.

One may wonder why we provide two kinds of macros, when theoretically `define-pattern` could be implemented with `define-form`? The major advantage of `define-pattern` is that it is implemented with simple list transformations, whereas `define-form` needs to evaluate the expander at compile time. In the development process of Tamacola, a macro system without dynamic evaluation was needed because we had to implement a PEG parser generator by macros before the `eval` function was ready to use. Even after the bootstrapping process was done, `define-pattern` is still used more often than `define-form` in Tamacola's source code because the mechanism and notation are simpler.

3.2 Command-Line Tools of Tamacola

For the programmer's convenience, a few commands are provided to invoke various Tamacola features to produce different types of files.

tamacc is the compiler that reads Tamacola code in `.k` files and generates corresponding ABC `.abc` files.

tamacola is a command line shell. It serves as a high-level driver; `.k` files and `.abc` files can be specified as its command-line arguments. It compiles the `.k` files and executes all arguments.

tamacola also provides the interactive shell within which programmers can evaluate expressions interactively. Expressions entered are evaluated in the global context, so global variables are visible to such expressions.

mkswf creates an SWF file from the list of `.abc` files specified as arguments. If `SymbolClass` is supplied as a command-line option, it generates a SWF file which can be loaded onto the Flash Player in web browsers.

mkpeg translates a PEG specification in a `.g` file to Tamacola code in a `.k` file. Tamacola's PEG is described in more detail in Section 4.

Figure 2 depicts the file types used in the Tamacola system and the commands to handle them.

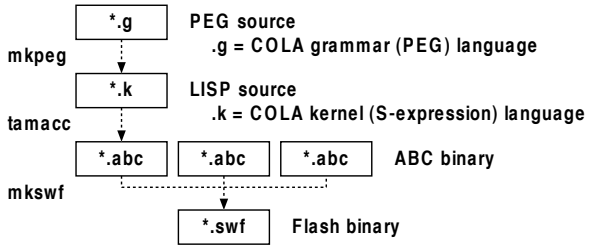

Figure 2. Tamacola tools

4. Implementation

In this Section we explain Tamacola's three main components, namely ABCSX, the PEG parser generator, and the compiler.

4.1 ABCSX: AVM2 assembler using S-expressions

ABCSX [18], written by author Yamamiya, is an assembler for AVM2. ABCSX generates ABC files from lists of instructions and metadata provided in S-expressions.

ABCSX has two input formats. One, called the "ABC-form", is a low-level format whose list structure corresponds to the logical layout of the ABC format. This form is used when low-level debugging or fine-grained manipulation of ABC files is necessary. ABC-form does a minimum of management of the jump table with labels, but does not handle constant table creation.

Another format, called the "ASM-form", is higher-level, concise and human-readable. When input is given in the ASM-form, ABCSX scans the constants used and then creates a constant table from the embedded literal expressions. This form is used for the Tamacola compiler as its output.

Following is a complete example of a "Hello, World" program in ASM-form:

```
(asm
 (method
  (((signature
     ((return_type *) (param_type ()) (name "hello")
      (flags 0) (options ()) (param_names ())))
    (code
     ((getlocal 0)
      (pushscope)
      (findpropstrict ((package "") "print"))
      (pushstring "Hello, World!!")
      (callproperty ((package "") "print") 1)
      (returnvoid))))))
 (script (((init (method 0)) (trait ())))))
```

The ASM-form starts with a symbol "`asm`", and may have several sections. There are mandatory sections such as "`method`" and "`script`". Also, there are optional sections such as "`instance`", "`class`", "`exception`", and "`metadata`". (No such optional sections were needed in this example.)

The "`method`" section contains method definitions. Each method definition consists of the method's signature and the code. The signature is metadata about the corresponding method's type, the debug name, information about optional arguments, and parameter names. In addition to regular methods, static methods, class constructors and anonymous functions can also be defined in the method section. The code consists of a list of ABC instructions. ABCSX analyzes the instructions at assembly time, and calculates necessary information required by the Tamarin VM such as the stack size and the register size used by the code. (More details on the stack and registers are in Section 5.)

The "`script`" section specifies the initialize script for the ABC data. In a typical case, avmshell runs the initialize script when an ABC is loaded, and binds functions and classes to the "global" context of the compilation unit.

ABCSX itself is an independent project. It is written in a subset of Scheme, and the same code base can be used for both COLA and Tamacola, along with other Scheme implementations such as Racket (a system formerly known as PLT-Scheme) and Gauche [9].

4.2 PEG Parser Generator

Tamacola heavily uses PEG [6] to specify most of the stages of the compiler. Since Tamacola's PEG is capable of matching deeply nested structures, transforming such structures can be concisely specified.

A PEG grammar consists of a series of rules. Each rule has a rule name on the left hand side and a parsing expression on the right hand side, separated by an equal sign `=`.

```
greeting = "Hello"
```

The PEG parser generator compiles each rule into a Tamacola function of the corresponding name (via macro expansions). The function needs to take two arguments, namely the input stream and the parser object, and the output is a boolean value which denotes whether the input matches or not. The following pseudo-code is the translated result from the rule above:

```
(define greeting
  (lambda (stream parser)
    (If the head of the stream matches "hello"?
      (Record "Hello" in parser, and answer #t)
      (Otherwise, Answer #f))))
```

The possible return values (#t and #f) indicate whether the rule succeeds or not; the actual value is recorded in the parser object.

The translation of a PEG rule to a function is straightforward. Programmers may write such functions by hand and use them in conjunction with rules defined in PEG.

We now show excerpts from the actual Tamacola compiler stage by stage.

4.3 PEG as the String Parser

A set of rules to recognize S-expressions comes first. The parser recognizes strings such as "42", "hello", and "(+ 3 4)" (a sequence of "(", "+", a space, "3", and so on) and constructs a corresponding list structure. The following is an excerpt from the S-expression parser specification:

```
char = [+-*/abcdefghijklmnopqrstuvwxyz]
dig  = [0123456789]
sp   = [ \t\r\n]*

sym  = char+ :s sp     -> (intern (->string s))
num  = dig+ :n sp      -> (string->number (->string n))
```

```
arity  = .*:x                          -> (length (->list x))
insts  = inst* :xs                     -> (concatenate (->list xs))

inst   = is-number:x                   -> `((pushint ,x))
       | is-symbol:x                   -> `((getlex ((ns "") ,(symbol->string x))))
       | '( '+ inst:x inst:y )         -> `(,@x ,@y (add))
       | '( '- inst:x inst:y )         -> `(,@x ,@y (subtract))
       | '( '* inst:x inst:y )         -> `(,@x ,@y (multiply))
       | '( '/ inst:x inst:y )         -> `(,@x ,@y (divide))
       | '( inst:f &arity:n insts:a ) -> `(,@f (pushnull) ,@a (call ,n))
```

Figure 3. An Excerpt of Tamacola Compiler (`syntax.g`)

```
sexp = sym
     | num
     | "(" sexp*:e ")" -> (->list e)
```

The `char` and `dig` rules match one of the characters enclosed in corresponding brackets. The `sp` rule matches any number of white space characters, because the star operator `*` repeats the previous expression zero-or-more times. (The plus operator `+` repeats the previous expression one-or-more times.) The `sym` rule matches a sequence of `char`s followed by white space. The sequence of `char`s is bound to the variable `s`, as `:` specifies a variable-binding action.

An arrow operator `->` specifies a semantic rule which creates Tamacola data and records it as the result of the rule. Any Tamacola expression can be used on the right hand side of the arrow operator. In this `sym` rule, the `->string` function converts its argument, which is the sequence of objects bound to `s`, to a string object. In turn, the `intern` function converts the string to a symbol object.

Similarly, the `num` rule returns a number object.

The vertical bar operator `|` specifies prioritized choices. Rules separated by `|` are tried in order, and the result from the first successful match is used. The `sexp` rules use operators to match an S-expression with symbols and numbers.

The PEG parser in Tamacola supports memorization of intermediate results and direct left-recursion optionally. For a complex grammar that requires a lot of backtracking, memorization provides good performance that is virtually linear to the input string. Left-recursion is also useful when writing a grammar that has multiple levels of operator precedence, for instance.

4.4 PEG as List Transformer

From the list structure that the parser creates, the compiler then generates code in ASM-form. Figure 3 shows the core part of this phase of the compiler (the `inst` rule), also written in PEG.

Before the `inst` rule, two helper rules are defined. `arity` counts all pending objects in the stream, and return the number. `insts` takes a list of lists (returned from `list*` expression), and concatenates them into a single list.

The `is-number` rule and the `is-symbol` match with a number value and a symbol value, respectively. If the compiler finds a number at the head of the stream, it generates the `pushint` instruction. Likewise, if a symbol is found, the `getlex` instruction is generated with the name space specification and the symbol to retrieve the value from a global variable.

Currently, predicate rules such as `is-number` and `is-symbol` can not be written in PEG directly; instead they are supplied as Tamacola functions. But recall that a hand-written function can serve as a rule in PEG, as explained above. For example, `is-number` is written as follows:

```
(define is-number
  (lambda (stream parser)
    (if (number? (stream/peek stream))
        (begin (stream/set-parser-result
                parser
                (stream/next stream))
               #t)
        #f)))
```

The `inst` rule is the core part of the transformer. The quoted-parenthesis pattern `'( ... )` matches a list structure, and the rules specified inside are tried, in an attempt to match the content of the list. For instance, a list beginning with a plus symbol `+` matches with a choice in the rule and emits the `add` instruction; e.g., `(+ 3 4)` emits `((push 3) (push 4) (add))`.

The last choice of the `inst` rule matches and emits a function application. The prefix `&` means following `arity` does not consume the stream, but just gets the result (the number of arguments). For example, `(print 42)` is translated to:

```
((getlex ((ns "") "print"))
 (pushnull)
 (pushint 42)
 (call 1))
```

This sequence of instructions would push the function named `print`, `null`, arguments, and call (with the `call` instruction with the number of arguments). In the case of a method call, the receiver object ("this" object in ActionScript) is specified before the arguments, but in this case, `null` is pushed instead because there is no receiver for a pure function.

4.5 The Whole Compiler Pipeline

The complete compiler is the combination of the parser and the compiler. The angle brackets operator "`<... >`" invokes a rule with specified arguments, and can be used to combine different sets of rules. Following is such an example:

```
compile = sexp:x <inst x>
```

In this case, the `sexp` rule converts the input text to a list structure, and the list is used as the input to the `inst` rule. For example, the rule `compile` converts "`(print (+ 3 4))`" to:

```
'((getlex ((ns "") "print"))
  (pushnull)
  (pushint 3)
  (pushint 4)
  (add)
  (call 1)))
```

This section has explained the basic compilation stages with PEG and ABCSX. But the mapping from a Lisp-like language to the Tamarin VM's model requires some more work. In the next section, we shall discuss the mapping of Tamacola language to Tamarin VM's stack model.

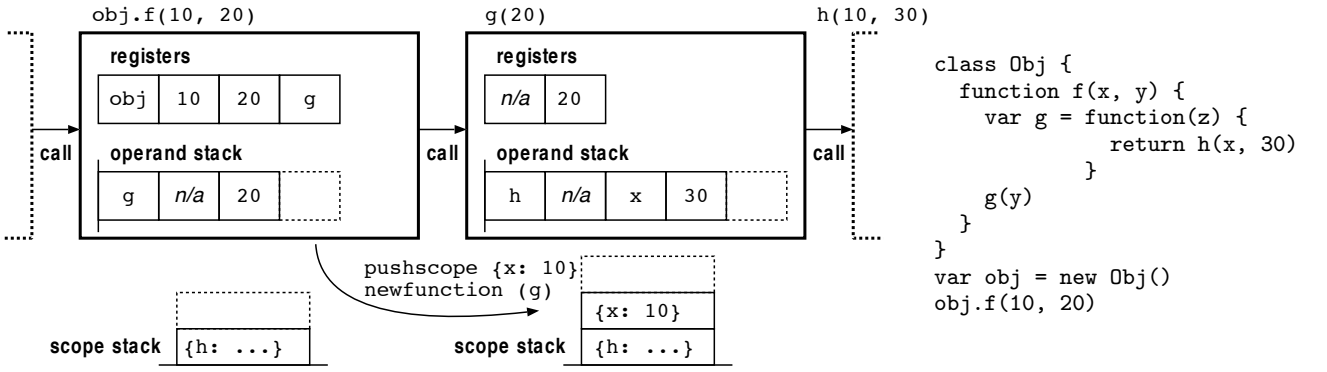

Figure 4. Runtime data structure on the Tamarin VM

4.6 The implementation of `eval`

Since the Tamacola compiler is available at runtime, compiling text as a program and generating the ABC data is simple. However, how the resulting ABC data is loaded has some implications. On avmshell, the shell provides the `loadBytes` primitive to load the ABC data from a byte array; so the compiler just generates the bytecode into an array and calls `loadBytes`, and executes the code. On the Flash Player however, SWF data is generated into a byte array, and then by using the Flash Player's asynchronous loading feature (`flash.display.Loader`), the SWF data is loaded from the data. When the loading is done, the code is evaluated and the sender of `eval` is notified via a callback function.

5. The Mapping to the Tamarin VM's Execution Context

In the previous section, we presented a simplified view of the Tamacola compiler. But the actual compiler needed some more work to deal with control structures, free variables, function creation, class creation, etc. Most importantly, we wanted to implement some Lisp features on top of the Tamarin VM, whose design is intimately tied to ActionScript, and such features needed to be properly mapped into the Tamarin VM's memory model and instruction set.

The Tamarin VM has three memory areas to store variables in a function; they are called registers, the operand stack, and the scope stack. Figure 4 shows the runtime execution context of the function calls in the Tamarin VM.

A set of registers are allocated for each function or method invocation. Registers provide the storage for local variables, arguments, temporary variables and the receiver of the message. The compiler determines the number of registers needed by a function, and upon a call to the function, the arguments and the receiver are stored at the compiler-determined locations in the registers.

Whether it is a method call with a receiver, or a function call without one, the zeroth register is reserved for the receiver (and is occupied by a nominal placeholder object when there is no receiver) and arguments start from the first register. Since accessing the zeroth register is no different from accessing the other registers for most of the instructions, Tamacola could have used the zeroth register to store the first argument and so on. However, for the sake of interoperability, it was decided to keep the calling convention in harmony with ActionScript conventions. Therefore, the arguments for a Tamacola function start from the first register and the Tamacola compiler generates an instruction to push `null` prior to other arguments for each function invocation (unlike the Adobe Flex compiler, which pushes the "global" context object for the compilation unit, even though it is not used in the function).

For example, in a method call `obj.f(10, 20)` of the example in Figure 4, the receiver `obj` is stored into the zeroth register, and the arguments 10 and 20 are stored in the first and second register, respectively. But for the function call such as `g(20)` in the example, the zeroth register is not used, and 20 is stored at the first register as the sole argument.

The operand stack is used to store operands, or intermediate results of the execution, for instructions in the function. Each instruction makes specific changes to the operand stack; For example, `add` takes two objects from the operand stack and pushes the result. Similarly, for a function call or a method call, the function object, the receiver (may be a nominal for a function call), and arguments are pushed onto the operand stack and the `call` instruction is executed. After `call`, these operands are popped and the return value is pushed at the top of stack.

The scope stack is used to access global variables or free variables from a closed-over inner function. This stack is necessary because neither the registers nor the operand stack could be accessed from other functions. The scope stack stores a chain of scope objects, each of which is either an activation object or a regular object and used as a dictionary for looking up the value from the variable name. At the bottom of the stack, the global context object that contains all global names stays, and a scope object is pushed to the stack when a new lexical scope is needed.

Unlike registers and the operand stack, whose lifetimes are limited to the corresponding function call, the scope stack works differently. A snapshot of the scope stack is remembered when a function is created by the `newfunction` instruction and the created function has an implicit reference to the snapshot so that the stack is available when the function is called, even if the enclosing function has already terminated.

In the example, when `g(20)` is invoked, the global function `h` is found in the bottommost entry in the scope stack. Then an object with the free variable `x` is stored on top of the global scope. This scope object with `x` is initialized by an instruction `pushscope` when the function `g` is created inside the method `obj.f`. (`x` must be on the scope stack because the function `g` cannot access the registers of the method `obj.f`.)

There is some consideration on how actually to use the scope stack. In theory, a `let` could be syntax sugar for a function call with newly introduced variables as arguments, so we could naively map an inner function in Tamacola to an inner function in ActionScript. However, this would impose a significant performance penalty because a function call in the Tamarin VM is slow. So, when the inner scope is introduced by a mere `let` and not a `lambda`, which may escape and outlive the enclosing function call, we would like to optimize it away.

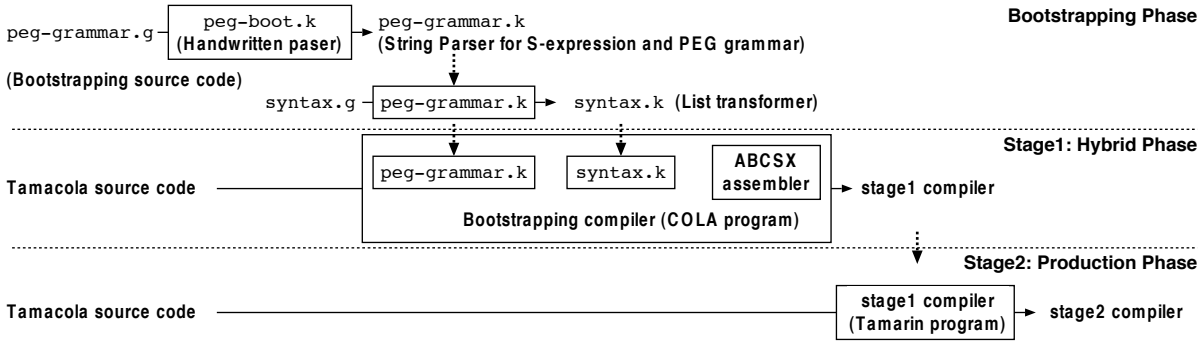

Figure 5. Building Process

To make an efficient variable mapping, the Tamacola compiler does the free-variable analysis when compiling a function to see if arguments for the function, or new variables introduced by `let`, are referenced by an inner function. Such variables are marked as "activation variables" and in the generated ABCSX code, activation variables are placed in an object on the scope stack. For all other variables are assigned to registers.

The regular object in the scope stack is also used to implement (unpopular) `with` syntax in ActionScript, for explanatory purpose we can show the compilation result in ActionScript with `with`: For example, a Tamacola function:

```
(lambda (x)
  (let ((y 10)
        (z 20))
    (lambda ()
      (print x z))))
```

is compiled into the the same bytecodes as the following ActionScript function would be compiled into:

```
function(x) {
  with ({"x" : x}) {
    var y = 10;
    with ({"z" : 20}) {
      return function() { print(x, z); };
    }
  }
}
```

6. Building Process

Because a primary goal of Tamacola was to write the system in itself, we needed to start the process of building the system before we had a compiler to compile the compiler's code. A typical approach to solve this situation of course is to write a compiler on another system (and possibly in another language) at first to compile the compiler.

However, one of our goals is to provide a compact and simple system, and we had an interesting opportunity to achieve the goal with minimal dependency on other systems because the central piece of Tamacola's compiler is written in a high-level PEG specification stored in a single file called `syntax.g`. `Syntax.g` specifies the transformation from S-expressions to the input for the ABCSX assembler. As OMeta has shown along the line of META-II [16], a grammar of PEG and translation of the grammar to procedural code can be written very compactly, especially with help from a unified pattern matcher (as we have shown in Section 4.2). This means that once we bootstrap the PEG parser generator, we can translate `syntax.g` to produce a compiler, and we can keep refining the process to reach to the point where the whole system can be compiled by itself.

Figure 5 shows the building process. In the following, we describe the process step by step.

6.1 The Bootstrapping Phase

Bootstrapping of Tamacola is done on the COLA system, because COLA has its own PEG parser generator and the bootstrap version of it has already been written by hand (by Piumarta). We borrow the bootstrap parser for COLA and feed the Tamacola's "real" PEG parser generator specification in to it via a file called `peg-grammar.g`. The resulting peg-grammar.k file contains a full-featured and working Tamacola PEG parser generator that runs on top of COLA. This PEG then processes `syntax.g` to generate the core of the Tamacola compiler that can run on COLA.

We needed to write some additional supporting code for the bootstrapping PEG, including code to absorb differences between COLA and avmshell, a stream library to handle byte arrays, and I/O.

6.2 Stage1: Hybrid Phase

We run `peg-grammar.k`, `syntax.k`, ABCSX, and such supporting code on top of COLA to parse all Tamacola code and generate corresponding .abc files that constitute the compiler that can run on top of the Tamarin VM. We call this phase "Hybrid" as the final Tamacola code is run on top of COLA. This is possible because the languages are almost identical. For macros in the Tamacola code, the expansion is done by the COLA macro system at this stage.

6.3 Stage 2: Production

The ABC files generated by the stage 1 can now be loaded onto avmshell. This "stage 2" compiler does run on the avmshell but is not quite "there" yet. For one thing, the macros in .k files, including syntax.k, from the previous stage were expanded by the COLA macro expander, so the Tamacola's macro systems are not exercised yet. And also we need to make sure that the environment can run in the form of the final product, including the different support code. So, we now compile the same `peg-grammar.g` but this time with different support code and with the macro system, then we move onto compiling `syntax.g` with the new `peg-grammar.k` by this compiler.

The resulting compiler is the "pure" Tamacola compiler; this compiler can also compile the entire system without needing COLA (even without using the macros expanded by COLA). Also, it can provide the interactive shell and dynamic code loading. In short, this is "full-featured" and self-sustained and can be used for further development.

```
arrayDecl = identifier:n (__ "[" identifier:i "]" -> `(,@ia-list ,i):ia-list)+
                                                                    -> `(array-decl ,n ,@ia-list)
...
query     = SUCH __ "{" expr:e __ "|" arrayDecl:n __ "<" expr:s __ "}"   -> `(such ,e ,n ,s)
          | ALL __ "{" identifier+:n __ "<" expr:s __ "}"                -> `(all ,(->list n) ,s)
...
lstmt     = IF lexpr:c THEN lstmt:t (ELSE lstmt:e)?                      -> `(if ,c ,t ,e)
          | expr:x __ "=" !("=") expr:v                                  -> `(assign ,x ,v)
          | compound
          | query:q (lDO | lDOINORDER):e compound:ss                     -> `(query ,e ,q ,ss)
```

Figure 6. An except of the parser specification of the Forall language.

7. Case Study #1: COLA Workspace

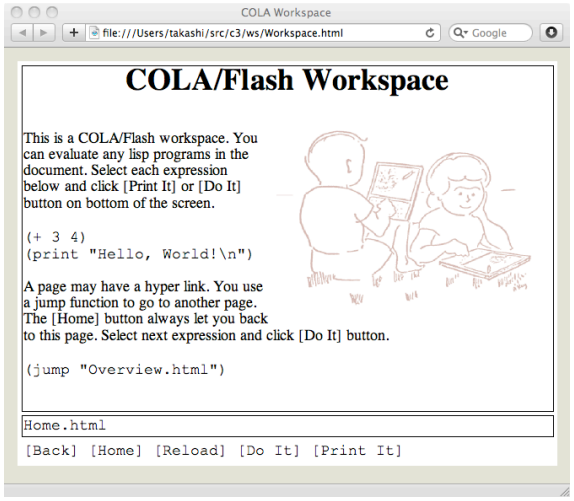

Figure 7. COLA Workspace

A workspace is a text editor in which you can enter, edit, evaluate, and inspect a program. Unlike a read-eval-print-loop usually provided in most Lisp implementations, a workspace works as documentation as well as as a debugging tool. A workspace is traditionally used in Smalltalk environments, and Emacs' scratch buffer and Mathematica's notebook provide similar features.

The COLA workspace (Figure 7) allows you to run a Tamacola program in a Web-browser. It has a user interface that found in a Smalltalk workspace, where you can simply eval an expression ("Do It") or print the evaluated result ("Print It"). The COLA workspace consists of several pages where each page has its topic and pages are connected by hyper links.

The contents of the page is stored using a subset of HTML format. The COLA workspace shows the content by the `htmlText` property in `flash.text.TextFiled` of the Flash API.

8. Case Study #2: A Particle Language

We wanted to implement a non-trivial language on top of Tamacola to validate its feasibility. Since the authors' group has been interested in programming languages for particles, and also since the language would be able to take advantage of Flash's graphics capabilities and the interactive programming of Tamacola, the language we decided to implement is a particle language with an interactive graphical environment. In analogy, the system would be like Processing [15] but using Flash as the execution environment.

The language is tentatively called "Forall". The basic syntax resembles JavaScript but has a few features to support arrays of data, i.e. particles:

- Multi-dimensional arrays are supported.
- To support simulations that repeatedly calculate the new state from the old state in the same buffer, the new state and the old state of the same array are accessible (double buffering).
- For array processing, new control structures to specify iteration concisely are introduced.

The multi-dimensional array in Forall uses comma-separated indices in brackets, i.e., "`arr[x, y]`" instead of "`arr[x][y]`". Internally, the actual buffer is a one-dimensional Tamacola `Array` with the dimension information called "format" attached. Double-buffer is done with a pair of such buffers, so that the user data can be used easily by the Flash APIs.

The syntax for "queries" in Forall looks like:

```
%all {a = arr[i:0..N, j:0..N]} do {a = f(a, i, j);}
%some{a = arr[i:0..N, j:0..N]} do {f(a, i, j);}
```

Notice that the user declares loop variables (in this example, `i` and `j`) and the variable to access the slot (in this example `a`). The idea of this syntax is that inside the compound statement after `do`, the code looks like it is simply describing the concern of a simple scalar variable a; but the code is "extended" by attaching a %-query clause. The variable `arr` is visible in the block as well, so if the code needs to access some other elements in `arr`, regular indexing syntax can also be used.

8.1 Translating Forall to Tamacola

The implementation of Forall is done in a "typical manner" when using PEG; namely, write a scannerless parser that constructs a parse tree from the source code string, and then transform the tree by another set of grammar rules, which effectively implement a visitor pattern on the tree.

The parser is written in about one hundred lines of code. The major part of it is used to specify the different levels of operator precedence, and also the specification of keyword tokens. The interesting addition is the syntax for the query clause. An excerpt of the parser is shown in Figure 6. As you can see, parsing the text and building the parse tree in S-expressions with quasi-quote is very straightforward in PEG. The operator precedence takes advantage of the left-recursion support and the memorization helps the performance of parser.

From the parse tree, another pass of processing generates the expression that Tamacola can execute. A part of the translator is shown in Figure 8. As the transformation for the query is somewhat complex, it uses helper functions (the long function names on the right hand side of `->` in the Figure) to create the index calculation from the "format" of the multi-dimensional array.

One reason that the transformation is complex is that because message-sending is relatively slow, care must be taken to avoid message-sending in the translated code.

```
make-query  = '( 'such trans:cond array-decl:t trans:a ) .:rhs
                        -> `(let ,t
                               (let ((format (FArray-format ,a)))
                                 (let ,(format-to-index-generator (reverse (cdr t)) 0 'format)
                                      ,(cross-product-iteration-generator
                                          (reverse (cdr t))
                                          (slot-access-generator-for-query
                                             (reverse (cdr t))
                                             0 a (car t)
                                             (if (null? cond) rhs `(if ,cond ,rhs '())))
                                          'format 1))))

query            = '( 'query 'do .:lhs trans :rhs ) <make-query lhs rhs>
```

Figure 8. An except of the translator from parse tree to Tamacola S-expression.

Assembler	1,687
SWF writer	267
Compiler	1,025
PEG processor	1,277
Libraries	1,260
Boot libraries	1,713
Miscellaneous	741
Total	7,970

Table 1. The source code line count

8.2 Interactive Environment

Since the Tamacola workspace (Section 7) lets us type in code interactively and compile the code down to the ABC bytecode and execute it, it is easy to hook up a new language to the workspace.

Figure 1 shows a screenshot of the execution.

9. Discussion

In the following, we show code size and performance benchmarks and discuss their implications.

9.1 Code size

We fully understand that the number of lines of code is a very inadequate measure of program complexity. Nonetheless, we provide some statistics on the Tamacola implementation in terms of LOC. At a minimum, it does tell us the relative weight of each component within Tamacola.

Table 1 shows the lines of code in each module of Tamacola, excluding test cases, documentation, and examples. The core part of the language consists of the ABCSX assembler, the SWF writer, compiler, and PEG parser generator, and these modules are written in 4,256 lines in total. The libraries include the stream library, the pretty printer, list functions, and etc. The boot libraries are all programs only necessary for the bootstrapping phase.

Comparison of code size with other languages is always fraught with difficulty. But just for reference, lua-5.1.4 includes 17,141 lines of code in `src` directory which includes all compiler, VM, and libraries.

9.2 Benchmarks

Table 2 shows a few micro benchmark tests on Tamacola, ActionScript and Squeak (the interpreter without Cog JIT compiler). Avmshell used for both Tamacola and ActionScript were built from the tamarin-central tree revision 714 and built with `--enable-debugger --target=x86_64-darwin` flags. The test code for ActionScript was built using the asc compiler in the Flex SDK 4.0. An Etoys 4 image and Squeak3.8.1 VM were used for the Squeak benchmarks. The platform was Mac OS X 10.6.4 on a MacBook Pro with Intel Core 2 Duo 2.5 GHz 4GB memory. The numbers are execution times in seconds.

	Fibonacci	Sieve	Sieve with `Vector`
Tamacola	8.90s	8.77s	4.73s
ActionScript	8.87s	5.09s	1.41s
Squeak (not Cog)	3.74s	2.57s	

Table 2. The results from micro-benchmarks.

The ActionScript versions of the benchmarks are ported from Squeak. The algorithm is the same, but the variables have type annotations, which are all `int`.

Fibonacci calculates the first 36 Fibonacci numbers recursively. This is a good test for measuring function call performance.

Sieve calculates the prime numbers up to 8,190 in the Eratosthenes Sieve algorithm. The test here repeats it 3,000 times from within one function. It is a good test of simple loops, arithmetic, and array accesses.

Sieve with Vector uses homogeneous `Vector` instead of `Array` to see the impact of parametric types in ActionScript.

By comparing the rows for Tamacola and ActionScript, we can see how efficient the code generated from Tamacola is, as they are on the same execution engine. For Fibonacci, Tamacola's result (8.90s) and ActionScript's (8.87s) are very close. The bytecode sequence for calling a function does not have much room for optimization and we can say that Tamacola is producing code that is as good as that produced by the Flex compiler. For the Sieve example, however, there is about 40% slow down. This is mainly due to the lack of type annotations in Tamacola.

Type annotations helps optimization in many ways. When the compiler can tell the type of operands for arithmetic is `int`, the asc compiler generates more specialized bytecode, e.g., `add_i` instead of `add`. The Tamacola compiler generates the `coerce_a` instruction prior to each and all stores into registers and to the operand stack, to satisfy the ActionScript code verifier, which requires that the types of the registers and operand stack slots be identical at a code location that may be reached by different code paths. The ActionScript compiler avoids this. Also, the ActionScript compiler does some peephole optimization, such as emitting the `increment_i` instruction when possible, using `ifle` instruction instead of `lessequal` and `iftrue` pair, eliminating redundant push and pop pairs.

Function call on the Tamarin VM is relatively slow compared to Squeak, for example. The difference is smaller for Sieve but it is still slower than Squeak. However, The Tamarin VM has some more potential because its homogeneous `Vector` uses a more efficient fixed-length buffer. Sieve with `Vector` is 3.6 times faster

than Sieve in ActionScript. The Forall language is implemented with `Array` as of this writing, but this shows the possibility of optimizing the Forall language, when Tamacola has type annotations added.

10. Related work

There are a few programming languages targeted to AVM2 besides ActionScript. HaXe by Nicolas Cannasse [4] is a popular multi-platform language written in OCaml. We referred to HaXe's ABC and SWF writer when implementing the ABCSX assembler.

Las3r by Aemon Cannon [5] is a Clojure-like compiler in ActionScript which can evaluate a program dynamically at runtime. Tamacola's dynamic code loader function for Flash Player is based on Las3r and As3 Eval library by Metal Hurlant [8].

Happy ABC by Hiroki Mizuno [11] is a Scheme compiler written in OCaml. Tamacola's stack layout in its let expression is derived from Happy ABC.

The Java Virtual Machine (JVM) and AVM2 share common aspects in various ways, and both are designed for object oriented languages. Languages in the Lisp family, such as Clojure by Rich Hickey [7] and Kawa by Per Bothner [3] gave us inspiration to design interoperability APIs for the VM.

11. Conclusions and Future Works

Thanks to the PEG parser generator and macros, the size of Tamacola source code is small, yet the system is powerful enough to build itself. A programming language designer can modify Tamacola or create a new language, or even make better tools with them. And the generated language is widely deployable on the Web with rich multimedia API on the Flash Player.

But there is more work ahead to further the vision. To date, the core compiler and only a few applications, the Forall language for massive manipulation and the COLA workspace for simple development within a Web browser, have been built. It is necessary to make more applications to test the usability of the framework.

Tamacola can build itself on avmshell but it still cannot bootstrap on the Flash Player alone because it needs a storage space for the generated ABC files outside the running Flash Player. To make a full development environment possible, it is necessary to write a Web-server application for accommodating such files.

There are important features to be implemented; most notably, the debugging support for this kind of generative language environment is always the big challange. A plausible path is to utilize the `fdb` debugger and debugging informations such as variable names and line numbers, but we would like to explore possibility of a self-implemented debugging facility.

Also, tail call elimination optimization is another key issue. While doing so for generic cases without incurring a performance penalty in AVM2 is very challenging, we could borrow ideas from existing JVM languages. Clojure has an explicit tail call expression `recur` which allows control to jump back to the "recursion point". Scala also supports the tail call optimization only if it can be converted to a jump instruction to the beginning of the same function.

The source code of Tamacola could be more compact and readable by using perhaps new domain specific languages more intensively. Notably, there is much room for improvement in the assembler, the SWF writer, and the pretty printer.

Designing a programming language is fun and practical. Sometimes choosing the right language makes your problem clear to describe and easy to understand. The harder we struggle with a problem, more and more the right language design is critical. Even today, a programming language is a kind of black magic. A compiler tends to be a large mystical chunk of code, and language designers and compiler creators need to be familiar with special programming tools.

What if designing a programming language were more like playing Etoys or Scratch? We could try and taste a language as if it were a work of handicraft or cooking. This idea may appear too radical considering that programming in an existing language is already magical enough, but a programming language can be thought of not only as instructions to a machine but also a communication tool among people in the first place. What if you could design a programming language using nothing more than a Web-browser ,and share your ideas using some simple mechanism? We believe that Tamacola is a tiny but significant step toward this goal.

Acknowledgments

The authors would like to thank Ian Piumarta for providing the basis of our work, Alex Warth for showing various implementation languages in OMeta. Scott Wallace, Alan Kay and all colleagues for feedback and support. We also would like to thank the program committee members for useful reviews.

References

[1] Adobe Systems Inc. ActionScript virtual machine 2 (AVM2) overview, . `http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf`.

[2] Adobe Systems Inc. The Tamarin Project, . `http://www.mozilla.org/projects/tamarin/`.

[3] P. Bothner. Kawa: compiling dynamic languages to the java VM. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 1998.

[4] N. Cannasse. haxe. `http://haxe.org/`.

[5] A. Cannon. Las3r. `http://github.com/aemoncannon/las3r`.

[6] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X.

[7] R. Hickey. Clojure. `http://clojure.org/`.

[8] M. Hurlant. As3eval. `http://eval.hurlant.com/`.

[9] S. Kawai. Gauche - a scheme interpreter. `http://practical-scheme.net/gauche/`.

[10] A. Kay, I. Piumarta, K. Rose, D. Ingalls, D. Amelang, T. Kaehler, Y. Ohshima, C. Thacker, S. Wallace, A. Warth, and T. Yamamiya. Steps toward the reinvention of programming (first year progress report). Technical report, Viewpoints Research Institute, 2007.

[11] H. Mizuno. Happy abc. `http://github.com/mzp/scheme-abc`.

[12] I. Piumarta. COLA kernel abstraction. Memo M-2009-007, Viewpoints Research Institute, 8 2009.

[13] I. Piumarta. Chains of Meanings in the STEPS system. Memo M-2009-011, Viewpoints Research Institute, 10 2009.

[14] I. Piumarta. PEG-based tree rewriter provides front-, middle- and back-end stages in a simple compiler. In *2nd ACM SIGPLAN Workshop on Self-Sustaining Systems (S3 2010)*, New York, NY, USA, 9 2010. ACM.

[15] C. Reas and B. Fry. *Getting Started with Processing*. Make, 2010. ISBN 144937980X.

[16] D. V. Schorre. META-II: a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM National Conference*, pages 41.301–41.3011, New York, NY, USA, 1964. ACM Press.

[17] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 Dynamic Languages Symposium*, pages 11–19, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[18] T. Yamamiya. An assembler for AVM2 using S-expression. Technical Memo M-2009-010, Viewpoints Research Institute, 2009.