Tutorial

Implementing Brainfuck in COLA

Michael Haupt
Software Architecture Group
Hasso-Plattner-Institut, University of Potsdam, Germany
michael.haupt@hpi.uni-potsdam.de

1 Introduction

This document provides a brief and very technical introduction to the basics of implementing programming languages using the COLA environment. To be able to focus on the introduction of COLA rather than semantics details of the language under implementation, the brainfuck language was chosen for its simplicity. The brainfuck implementation presented here is most certainly less than optimal in terms of performance, but then again, the latter is clearly not the core intention of this document.

The first section gives brief overviews of both COLA and brainfuck. Section 2 is the main part, describing the brainfuck implementation in detail. The author hopes that the reader will be able to pick up a great deal of interesting information about COLA while working through the text. In section 3, a brief manual for running brainfuck programs is given. Section 4 summarises the tutorial and discusses some possible extensions of the presented implementation. The appendix contains both the complete source code of the brainfuck implementation and the license.

The author will be happy to receive comments, suggestions for improvements, etc. by e-mail.

1.1 COLA

COLA[1] (*combined object-lambda abstraction*) is a programming platform centered on the idea of building complex systems using minimal abstractions. It originates from ongoing research on fundamental new computing technologies[2,3] [1] at Viewpoints Research Institute[4].

[1] http://piumarta.com/software/cola/
[2] http://vpri.org/html/work/ifnct.htm
[3] http://vpri.org/mailman/listinfo/fonc
[4] http://vpri.org/

Like the name suggests, COLA provides two kinds of abstraction. At the bottom of the system, there lies a minimal object model—minimal in that it is the smallest possible representation that allows for full dynamism, i. e., late binding and modification of virtual method tables through message sending—providing *object abstraction* [4] that can be used as the basis for programs using a prototype-based object-oriented programming language. The language has a Smalltalk-like syntax.

One level higher, there is a layer providing *functional abstraction.* It also comes with a programming language; this time, S-expressions [5] are used to represent programs. It is important to note that, even though programs written in the function part of COLA look much like Scheme code, their semantics are significantly different from Scheme. It is healthy to adopt, and early so, a view on these S-expressions that regards them as a convenient and easy-to-parse C AST representation.

COLA function part programs are compiled using a just-in-time compiler that performs, for optimisation purposes, tree pattern matching on the ASTs prior to generating native code, which is then executed. The *entire* function part of COLA is implemented in terms of the underlying object part, and using the object-oriented language pertaining to the latter.

From within the object part, it is easy to access the underlying C level, and from programs written in the function part, it is easy to access the underlying object level. In summary, the programming model made available at the level of the function part of COLA is extremely powerful and provides a great degree of control to the programmer.

On top of the function part, PEGs (*parsing expression grammars*) [3] have been realised. Using PEGs, it is easy to define language implementations in terms of a grammar and corresponding actions associated with matching parts of grammar rules. The PEG implementation in COLA has been used to provide the brainfuck implementation described in Sec. 2; all necessary explanations will be given there.

1.2 Brainfuck

The brainfuck[5,6] programming language is a minimalistic language. It actually implements a Turing machine [7] and is indeed Turing complete [2][7].

Brainfuck features an array of 30,000 cells, each of which contains an ASCII character. Initially, all cells are initialised to 0. There is a pointer—henceforth called `P`—referencing a given cell in the array. Initially, `P` references the leftmost cell (i. e., the one at the array index 0).

A set of eight commands is used to write brainfuck programs. They are given in Table 1. The six commands above the double horizontal line form the Turing complete instruction set of brainfuck. The two below only exist to allow for side effects also known as user interaction.

[5]`http://esoteric.voxelperfect.net/wiki/Brainfuck`

[6]`http://en.wikipedia.org/wiki/Brainfuck`

[7]The referenced paper does not prove Turing completeness for brainfuck, of course. It proves it for P'', programs written in which, however, can be trivially transformed to brainfuck. P'' lacks input and output capabilities, but otherwise uses a set of six symbols, just like brainfuck.

>	increase P by 1
<	decrease P by 1
+	increase the value of the cell referenced by P by 1
-	decrease the value of the cell referenced by P by 1
[	if the value of the cell referenced by P is 0, proceed after corresponding]
]	proceed at corresponding [
.	output the value of the cell referenced by P as ASCII value
,	read one character and put its ASCII value into the cell referenced by P

Table 1: All brainfuck commands.

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<++++++++++++++
+.>.+++.------.--------.>+.>.
```

Listing 1: Hello world in brainfuck (without comments).

Brainfuck programs consist of sequences of these characters. The classic hello world example looks like shown in Listing 1[8] when written in brainfuck.

All characters that do not correspond to a brainfuck command are, by definition, ignored, so that it is possible to write nicely commented brainfuck programs, such as in Listing 2[9].

Given the simplicity of the language, this text will not waste any more time on introducing its subtle concepts. Instead, the brainfuck implementation in COLA will be discussed at length.

[8]This listing was copied from the Wikipedia page on brainfuck (the URL was given above) as of 2007-11-12 17:19 CET.

[9]This listing was copied from the same source as Listing 1. Comments were slightly abridged.

```
++++++++++
[>+++++++>++++++++++>+++>+<<<<-] Initial loop to set up useful values in array
>++.                             Print 'H'
>+.                              Print 'e'
+++++++.                         Print 'l'
.                                Print 'l'
+++.                             Print 'o'
>++.                             Print ' '
<<+++++++++++++++.               Print 'W'
>.                               Print 'o'
+++.                             Print 'r'
------.                          Print 'l'
--------.                        Print 'd'
>+.                              Print '!'
>.                               Print newline
```

Listing 2: Hello world in brainfuck.

2 Implementing Brainfuck in COLA

The complete source code of the brainfuck implementation is given in Appendix A. This section will perform a complete walk-through by quoting and discussing, bit by bit, pieces of the source.

The brainfuck implementation is written in the language provided for the function part of COLA. It frequently accesses the object layer below, though, to achieve certain things. It also makes use of the PEG implementation available for COLA. The file containing the implementation is called `brainfuck.peg`.

At first sight, the source code is divided into three main sections, spanning lines 1–47, 48–85, and 86–100, respectively. This separation into three sections is typical of `.peg` files. Each of the sections plays a certain role in the brainfuck implementation, and each of them will be dealt with below in a dedicated subsection in detail.

In short, the first section is about setting up an environment for the language implementation. The second—and most important—contains the language definition and, so to speak, implementation. The third section contains further code that is needed to get the language implementation up and running.

2.1 Preliminaries

The preliminaries are included in between `%{` and `%}` in lines 25 and 47, respectively. They are copied *verbatim* to the output when the `.peg` file is processed.

As mentioned above, this section is for setting up an environment for the language implementation to live in. Hence, you can find all kinds of definitions here.

Taking a closer look, the first two lines that contain somethling looking like code define two names and bind them to the result of some `import` operation:

```
27 (define OrderedCollection (import "OrderedCollection"))
28 (define FileStream        (import "FileStream"))
```

This is already the first encounter of an access from the function part of COLA to its object part. Both `OrderedCollection` and `FileStream` are prototypes[10] defined in the underlying object library[11]. Each such prototype is, by default, not visible to the function part, but can be made available by `import`ing it. The `import` operation results in a reference to the prototype object, which can be bound to a name using `define`. In essence, these two lines make the two prototypes available to the function part and accessible as objects henceforth.

Binding *C functions* to names is equally easy, as the following code demonstrates:

```
31 (define putchar (dlsym "putchar"))
32 (define getchar (dlsym "getchar"))
33 (define memset  (dlsym "memset"))
```

[10]The object library is entirely prototype-based, so these are *not* classes. In everyday usage, they feel a lot like classes, though, which is a matter of convenience.

[11]In a complete installation of COLA, the object library accessible from the function part resides, in the form of `.st` files, in the directory `function/objects`.

The three well-known functions `putchar()`, `getchar()`, and `memset()` from the C standard library are bound via `dlsym` and thus made available as real functions. In the subsequent program, a bit of code like `(putchar 65)` will output the letter `A` to standard output. This brief example should have shown you that accessing the C layer is indeed very easy from within COLA programs.

After having bound some objects and functions, the implementation moves on to defining the memory available to brainfuck programs:

```
(define mem-size 32768)
(define memory (malloc mem-size))
```

Here, `mem-size` is a constant denoting that this particular brainfuck implementation boldly uses an array of 32 kB instead of only 30,000 bytes. That very array is allocated using `malloc` and bound to the name `memory`.

For convenience, there also exists a function that relies on `memset()` to zero out the entire array:

```
(define init-memory (lambda () (memset memory 0 mem-size)))
```

It is interesting to note that this single line of code actually defines a valid C function: the result of compiling the above code with the COLA JIT compiler is a pointer to a function adhering to the C ABI. Bearing in mind that the S-expressions used in COLA programs represent actual C ASTs, it is trivial to map the above code to C source code:

```
void* init-memory() {
    return memset(memory, 0, mem-size);
}
```

Of course, the two identifiers `init-memory` and `mem-size` do not conform to C syntax, but the point should be clear.[12]

The brainfuck pointer `P` is defined and made to point to the beginning of the array like this:

```
(define P memory)
```

The last two lines in the preliminaries section introduce several features of the COLA function part that have not been introduced yet, namely *syntax definitions* and *message sends*.

```
(syntax inc (lambda (node) `(set ,[node second] (+ ,[node second] 1))))
(syntax dec (lambda (node) `(set ,[node second] (- ,[node second] 1))))
```

At first sight, these two definitions look much like the definitions of ordinary functions, like seen above for `init-memory`: there is a name that is bound to a `lambda` expression. Moreover, the functions seem to apply *quasiquotation*—using backticks (`` ` ``) for quasiquoting and commas (`,`) for unquoting—as known from the Scheme programming language [6]. In fact, the quasiquote semantics of the function part of COLA are the same as in Scheme.

[12]One might argue that `init-memory` was not necessary at all, because the `memory` array could have been allocated and filled with zeros using `calloc()`. The present approach was chosen to illustrate the creation of C functions in Jolt.

The notable obvious differences to Scheme code are twofold: the two definitions of `inc` and `dec` are not made using `define`, but `syntax`. Moreover, there appear square brackets (`[]`) in the code, which is certainly not Scheme syntax.

The definitions are bound to their respective names using `syntax` to mark them as *syntax definitions*. Roughly speaking, they can be regarded as a COLA equivalent to macros (they are significantly more powerful, though—see below). Macros can be used just like functions in code, only that they are evaluated *immediately*, instead of at run-time, and that the results of their evaluation—usually a bit of AST—is *inlined* where their "invocation" was found by the compiler. Taking this into account, it will be immediately clear why `inc` and `dec` return quasiquoted ASTs: they return code to be inserted instead of the macro applications.

Looking at the code, the `lambda`s each accept a single parameter called `node`. This is a representation of the AST node currently being visited by the COLA function part compiler. The `node` is an *object* to which messages can be sent—more precisely, it is a `SequenceableCollection`[13]. Sending messages to objects is done using square brackets. Each pair of square brackets encloses one message send in, roughly, Smalltalk syntax. Nested message sends must be enclosed in nested square brackets, as will be demonstrated in Sec. 2.3 below.

The four appearances of `,[node second]` in the definitions of `inc` and `dec` all access (and unquote) the second element of the `node`. Given that `node` represents the AST node currently being visited by the compiler, the first element consequently is the name of the macro currently being evaluated. The second and subsequent elements reference AST parts passed as parameters to the macro application. Unquoting will lead to a textual representation of that AST part to be inlined in the newly created AST.

In essence, an application of the `inc` macro, e. g., `(inc q)`, will yield an AST snippet looking just like this: `(set q (+ q 1))`. Analogously, `(dec 42)` will yield the obviously nonsensical `(set 42 (- 42 1))` which will lead to an error when evaluated.

It is important to note that calling syntax definitions "macros" is inaccurate. A syntax definition is actually not only passed the single `node` argument that gives it access to the AST, but a second argument usually called `compiler`, which was omitted for simplicity in the example. The `compiler` argument is a reference to an object representing the actual *compiler* as part of whose compilation process the syntax definition application is met. Hence, syntax definitions can have far greater influence on the compilation process than macros, which essentially just replace text with different text—syntax definitions can immediately talk to the compiler and, for instance, influence the way it generates native code.

2.2 Parsing and Compiling

As mentioned above, the second section of the `brainfuck.peg` file contains the most important part: the definition of the language's grammar and behaviour. Lines 51–84 make up the entire thing. Given that brainfuck is such a simple language, this should

[13]Again, see the `function/objects` directory of your COLA installation for the sources.

not come as a surprise.

The grammar is defined in a convenient way: a non-terminal name is given, followed by an equals sign (`=`). After that, all production rules pertaining to the non-terminal are given in what looks much like an EBNF notation. In fact, the usual symbols as found in EBNF can be used to specify COLA PEG grammars: `?` denotes an option, `+`, one or more repetitions, and `*`, zero or more. Braces (`()`) are used to denote groups, and the vertical bar (`|`) marks alternatives.

2.2.1 Rules for Terminal Symbols

The description of the brainfuck implementation will start at the end and move to the front—this bottom-up approach is best suited to provide a good understanding of the concepts at work and how they are used together. So, definitions of rules for all the terminal symbols are considered first:

```
Forward         = '>'
Backward        = '<'
Increment       = '+'
Decrement       = '-'
Put             = '.'
Get             = ','
While           = '['
Wend            = ']'
BrainfuckSymbol = ; all of the above
    Forward | Backward | Increment | Decrement | Put | Get | While | Wend
```

These lines define a dedicated non-terminal for each terminal symbol in brainfuck, and another non-terminal that matches any brainfuck symbol. Terminal symbols are given in single quotes (`''`).

Before moving on, some clarification is advisable. Throughout this text, the brainfuck implementation has always been called a brainfuck *implementation* so far. Deliberately so: the true nature of the language implementation—interpreter or compiler?—should not be given away until just now. In fact, the implementation utilises the COLA function part capabilities of just-in-time compilation. This brainfuck implementation is *not* an interpreter: the brainfuck programs passed to it are indeed compiled to native code before they are executed.

2.2.2 Matching a Single Instruction

The next rule is already the one where the really interesting things happen. It matches and processes a single brainfuck instruction:

```
Instruction = ((!BrainfuckSymbol) .)* ; exclude all "illegal" characters
    ( Forward                       { `(inc P) }
    | Backward                      { `(dec P) }
    | Increment                     { `(inc (char@ P)) }
    | Decrement                     { `(dec (char@ P)) }
    | Put                           { `(putchar (char@ P)) }
    | Get                           { `(set (char@ P) (getchar)) }
    | While body:Instructions Wend { `(while (!= 0 (char@ P)) ,@body) }
    )
```

Looking at the rule definition from a high level, there is an alternative for each brainfuck symbol. The curly braces ({}) mark *action parts* of the grammar. Action parts are accumulated during parsing when the rule parts they are associated with match, and they are executed once the parser has finished parsing a document. In other words, whenever a `Forward` symbol is matched whilst parsing some brainfuck input, the corresponding action `{ '(inc P) }` is noted for later execution. That way, the parser generates, while parsing its input, an executable representation in terms of actions.

Each of the action parts contains a quasiquote expression. That is, these action parts to not actually *do* something, instead they return quasiquoted AST parts. This is how one can tell that the implementation actually first assembles a complete AST of the brainfuck input. An "interpreting" implementation would not use quasiquotation in these places; its action parts would immediately execute the logic associated with each rule of the grammar. Apart from that, implementing an interpreter would actually be more complicated than the present solution: parser input positions would have to be memorised to be able to realise loops, and the parser would have to parse its input over and over again, for each iteration.

Looking at the details of the `Instruction` rule, its structure becomes apparent. The first line,

```
Instruction = ((!BrainfuckSymbol) .)* ; exclude all "illegal" characters
```

contains some code that facilitates the brainfuck implementation's ignoring all non-brainfuck characters in the input. The expression `(!BrainfuckSymbol)` is a condition for the following dot (`.`). The dot matches any single character. The condition, in this case, restricts `.` to match only those characters that are not brainfuck symbols. An arbitrary number of these may be given, as denoted by the `*` attached to the code.

The rest of the `Instruction` rule,

```
    ( Forward                   { `(inc P) }
    | Backward                  { `(dec P) }
    | Increment                 { `(inc (char@ P)) }
    | Decrement                 { `(dec (char@ P)) }
    | Put                       { `(putchar (char@ P)) }
    | Get                       { `(set (char@ P) (getchar)) }
    | While body:Instructions Wend { `(while (!= 0 (char@ P)) ,@body) }
    )
```

defines a group of alternatives: for each brainfuck symbol, an alternative is given with the corresponding AST-generating action part.

The ASTs generated for `Forward` and `Backward` apply the `inc` and `dec` macros explained above in Sec. 2.1 to the brainfuck array pointer `P`. Consequently, the code returned from these rule parts will, when executed, increment or decrement `P`, correctly implementing the language semantics.

The `Increment` and `Decrement` actions are supposed to alter the value of the brainfuck array cell *pointed to* by `P`. This is achieved by applying the `inc` and `dec` macros to `(char@ P)`, which interprets `P` as a pointer to a C `char` and dereferences it[14].

[14]The equivalent C code for `(char@ P)` is `*P`.

Put and Get are implemented in non-surprising ways, invoking the putchar and getchar functions defined in the preliminaries section accordingly. For Get, the result of the getchar application is stored in the location pointed to by P using set.

For the loop constructs While and Wend, the case is more interesting. The corresponding rule part,

```
    | While body:Instructions Wend { `(while (!= 0 (char@ P)) ,@body) }
```

matches an entire "loop", starting with While and ending with Wend, with a number of instructions in between. This rule part refers to the Instructions rule, which will be discussed below. For now, it is important to know that the result of matching the Instructions rule—an AST—is bound to a variable called body.

The action part for loops returns an AST making use of the while construct available in the function part of COLA. The loop condition, adhering to the brainfuck language semantics, checks whether the value in the array cell pointed to by P is zero. The loop body is simply the AST returned from the matching of the Instructions rule. It is inlined into the generated AST by unquoting it to a textual representation using ,@. Other than unquoting via a single comma (,), ,@ expects a *list* of elements to be inlined in a quasiquote expression. Given that body is indeed a list of ASTs, this certainly makes sense.

2.2.3 Matching Instruction Sequences and Building ASTs

The Instructions rule was already mentioned above:

```
Instructions =
    insns:Instruction { (set insns [OrderedCollection with: insns]) }
    ( ins:Instruction { [insns add: ins] } )*
    { insns }
```

While the Instruction rule matches a single brainfuck instruction and generates a corresponding AST representing its behaviour, the Instructions rule matches an entire sequence of brainfuck instructions and returns the corresponding AST for it.

To achieve this, it makes use of the Instruction rule and adds the ASTs returned from matching single instructions to an OrderedCollection, which is finally returned:

```
    ( ins:Instruction { [insns add: ins] } )*
    { insns }
```

The first of these two lines matches, zero or more times, a single instruction, binds the resulting AST to the variable ins, and, in the corresponding action part, adds the AST contained in ins to the OrderedCollection named insns. This latter operation is done at the object level: the add: message is sent to insns with the parameter ins in square brackets. The second line just contains an action part that is executed when matching single instructions is finished; it merely returns the insns collection.

This is how ASTs are assembled. To explain how the OrderedCollection containing the assembled AST comes into play, it is important to point out how the first part of the Instructions rule is implemented:

```
    insns:Instruction { (set insns [OrderedCollection with: insns]) }
```

A single instruction is matched, and the corresponding AST is bound to `insns`. Later on in this rule, `insns` is expected to contain an `OrderedCollection`, which is achieved by the action part. It creates the desired `OrderedCollection`[15], fills it with the AST parts that are already there—at this point in time, they are contained in `insns`—and finally binds the newly created `OrderedCollection` to `insns`.

This is indeed a dirty trick, but the way COLA PEG grammars are formulated requires it to be applied this way—they do not support local variables yet. There is also a consequence for the semantics of the language: each program and loop body will consist of at least one instruction.

2.2.4 Parsing and Executing a Program

The final rule of the brainfuck grammar is the `Program` rule. It is the first one given in the grammar section of the `brainfuck.peg` file, which implies that it is considered to be the starting rule of any brainfuck program.

The rule does not do much more than passing control to the `Instructions` rule—a program *is* a sequence of instructions, after all—and binding the resulting AST to the `prg` variable:

```
Program =
    prg:Instructions {
        (let ((bf-ast ‘(let () ,@prg)))
            (printf "AST of the Brainfuck program:\n")
            [StdOut println: bf-ast]
            (printf "Running now...\n")
            [bf-ast eval])
    }
```

The interesting part of the `Program` rule is its action part, which consists of a single `let` expression. The `let` defines `bf-ast`:

```
        (let ((bf-ast ‘(let () ,@prg)))
```

The same idiom as observed above for brainfuck loop constructs is used here again: the AST resulting from the `Instructions` rule is spliced into a newly created expression. In this case, it is another `let` expression that does not bind any further names. A `let` expression of the form `(let () ...)` corresponds to a C block occurring somewhere in a method: `{ ... }`. In a nutshell, the resulting value of `bf-ast` is an executable AST.

The four remaining lines of the rule's action part,

```
            (printf "AST of the Brainfuck program:\n")
            [StdOut println: bf-ast]
            (printf "Running now...\n")
            [bf-ast eval])
```

mostly consist of convenience output. In fact, only the fourth line, where the `eval` message is sent to the `bf-ast` object, is actually required. For debugging purposes, the brainfuck implementation outputs the entire AST before it executes it.

[15]This is done by sending the `new` message to the `OrderedCollection` prototype.

2.3 Starting Up the Implementation

The final part of the `brainfuck.peg` file begins after the `%%` delimiter. Like for the preliminaries section, the contents of this last section are copied verbatim to the file generated when processing `brainfuck.peg`. This final section contains code that is used to start up the brainfuck implementation.

First of all, the brainfuck memory array is initialised to contain only zeroes:

```
(init-memory)
```

This is done by simply invoking the `init-memory` function that was defined in the preliminaries section (cf. Sec. 2.1).

The next few lines,

```
(let ((bf-input
          (if [[OS arguments] notEmpty]
              (yy-new [FileStream on: [File open: [[OS arguments] removeLast]]])
              (yy-new [FileStream on: StdIn]))))
    (or
        (yy-parse bf-input)
        (printf "Syntax error.\n")))
```

assemble an input stream for the brainfuck program to be parsed and pass it to the language implementation. The `let` expression binding `bf-input` handles command line arguments. The code `[[OS arguments] notEmpty]` accesses the object level[16] to determine whether any command line argument was given at all. If so, `bf-input` is bound to an input source corresponding to that file. Otherwise, brainfuck code is expected to be entered via `stdin`.

The ensuing `or` expression exploits the fact that parameters to logical disjunctions are lazily evaluated. If parsing—achieved by passing `bf-input` to `yy-parse`—fails, an error message is output.

Finally,

```
(printf "\n")
```

prints a newline character to flush output and to return cleanly to the operating system prompt.

3 Compiling and Running the Implementation

In the following, it is assumed that the `brainfuck.peg` file resides, along with some brainfuck program files with the suffix `.bf`, in the `function/examples/peg` directory of a complete COLA checkout. The version of COLA the presented implementation is tested on is SVN revision 350; the author does not guarantee it to work properly on any other revision.

To generate an actual COLA function part program out of the `.peg` file, it is necessary to preprocess it. This is done using the following command line:

```
$COLAF boot.k peg.k -o brainfuck.peg.k brainfuck.peg
```

[16]This is an example of nested square brackets use to handle multiple message sends.

where `$COLAF` references the COLA function part executable[17].

This command line will start up the COLA function part and read, in the given order, `boot.k`, which will set up a complete COLA environment, and `peg.k`, which defines the PEG processor. Finally, the name of the output file for the brainfuck implementation—`brainfuck.peg.k`—and the input file it is to be generated from are given.

To run a brainfuck program, e. g., a file `hello.bf` containing the hello world code shown in Sec. 1.2, the following command line is to be given:

```
$COLAF boot.k brainfuck.peg.k hello.bf
```

This will, again, set up a complete COLA environment before loading the brainfuck implementation, which will then read and execute `hello.bf`.

4 Summary

This tutorial has demonstrated how to implement a simple programming language in COLA using the available PEG implementation. The brainfuck programming language was chosen for its simplicity, which allowed for concentrating on the features of COLA instead of language features.

Various improvements are conceivable. For instance, it could be interesting to realise the brainfuck implementation as an *interpreter* that executes the language semantics as parsing goes along, instead of generating a complete AST of the input program and passing that to the just-in-time compiler.

It would also be nice to have an actual brainfuck *compiler* generating binary files that could be executed independently. COLA, in its current version, lacks the ability to serialise generated native code to files. This feature is planned for the near future, however.

Acknowledgments

The author is grateful to Ian Piumarta, who made many important remarks that helped improving this document's accuracy, and for various hints on COLA details that were made by Hans Schippers—they really helped getting the brainfuck implementation running. Thanks also go to Robert Feldt for his suggestions for improvement.

A Complete Source Code

```
;;; brainfuck implementation using the function part of COLA

;;; License (MIT License)
; Copyright (c) 2007-2008 Michael Haupt
; michael.haupt@hpi.uni-potsdam.de, http://www.hpi.uni-potsdam.de/swa/
;
; Permission is hereby granted, free of charge, to any person obtaining a copy
; of this software and associated documentation files (the "Software"), to deal
```

[17]For the COLA SVN revision 350, this executable is `function/jolt-burg/main`.

```
; in the Software without restriction, including without limitation the rights
; to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
; copies of the Software, and to permit persons to whom the Software is
; furnished to do so, subject to the following conditions:
;
; The above copyright notice and this permission notice shall be included in
; all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
; AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
; OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
; THE SOFTWARE.

%{
;;; bind library objects
(define OrderedCollection (import "OrderedCollection"))
(define FileStream        (import "FileStream"))

;;; bind library functions
(define putchar (dlsym "putchar"))
(define getchar (dlsym "getchar"))
(define memset  (dlsym "memset"))

;;; the memory
(define mem-size 32768)
(define memory (malloc mem-size))

(define init-memory (lambda () (memset memory 0 mem-size)))

;;; the pointer
(define P memory)

;;; convenience functions
(syntax inc (lambda (node) `(set ,[node second] (+ ,[node second] 1))))
(syntax dec (lambda (node) `(set ,[node second] (- ,[node second] 1))))
%}

;;; grammar and language implementation

Program =
    prg:Instructions {
        (let ((bf-ast `(let () ,@prg)))
            (printf "AST of the Brainfuck program:\n")
            [StdOut println: bf-ast]
            (printf "Running now...\n")
            [bf-ast eval])
    }

Instructions =
    insns:Instruction { (set insns [OrderedCollection with: insns]) }
    ( ins:Instruction { [insns add: ins] } )*
    { insns }

Instruction = ((!BrainfuckSymbol) .)* ; exclude all "illegal" characters
    ( Forward                     { `(inc P) }
    | Backward                    { `(dec P) }
    | Increment                   { `(inc (char@ P)) }
    | Decrement                   { `(dec (char@ P)) }
    | Put                         { `(putchar (char@ P)) }
    | Get                          { ‘(set (char@ P) (getchar)) }
    | While body:Instructions Wend { ‘(while (!= 0 (char@ P)) ,@body) }
    )

Forward         = ’>’
Backward        = ’<’
Increment       = ’+’
Decrement       = ’-’
Put             = ’.’
Get             = ’,’
While           = ’[’
Wend            = ’]’
BrainfuckSymbol = ; all of the above
    Forward | Backward | Increment | Decrement | Put | Get | While | Wend

%%

;;; start up brainfuck

(init-memory)

(let ((bf-input
          (if [[OS arguments] notEmpty]
              (yy-new [FileStream on: [File open: [[OS arguments] removeLast]]])
              (yy-new [FileStream on: StdIn]))))
    (or
        (yy-parse bf-input)
        (printf "Syntax error.\n")))

(printf "\n")
```

B License

References

[1] A. Kay and D. Ingalls and Y. Ohshima and I. Piumarta and A. Raab. Proposal to NSF. Granted on August 31, 2006. Technical Report VPRI Research Note RN-2006-002, Viewpoints Research Institute, 2006.

[2] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[3] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.

[4] I. Piumarta and A. Warth. Open Reusable Object Models. Technical Report VPRI Research Note RN-2006-003-a, Viewpoints Research Institute, 2006.

[5] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[6] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, Jr. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[7] A. M. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 1936.