Tutorial

Implementing Brainfuck in COLA (Version 2)

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 (*combined object-lambda abstraction*) environment. To be able to focus on the introduction of COLA rather than the semantic 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. This version of the tutorial uses version 2 of the COLA environment, which allows for more concise language implementations that also fit more organically into the overall environment. 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. Detailed information about COLA grammars and parsers is given in section 4. Section 5 summarises the tutorial and discusses some possible extensions of the presented implementation. The appendix contains the complete source code of the brainfuck implementation.

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

1.1 COLA

COLA[1] 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].

Like the name suggests, COLA provides two kinds of abstraction. On the one hand, there is 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, and it is statically compiled. It provides seamless access to the "C level", thus supporting the implementation of primitives that cannot be expressed otherwise.

On the other hand, there is an environment providing *functional abstraction*. It also comes with a programming language; here, 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 [6] 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 the prototype-based object language mentioned above. From programs written in the function part, the object part can be accessed easily; thus, the programming model available here is very powerful, allowing the seamless integration of both abstractions.

The two parts of COLA are actually adjacent. The object part allows for concisely defining data structures in terms of objects and messages. The function part supports the definition of first-class behaviour through lambda abstraction.

An entirely malleable parser lies at the heart of the COLA function part. It can be instructed on the fly to accept code written in any language for which a grammar has been defined. Grammars can be defined in the function part itself by means of an implementation of PEGs (*parsing expression grammars*) [3]. 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 section 2. All explanations necessary for understanding the implementation will be given there, and section 4 will give a more detailed introduction to the concepts at work.

[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/`

`>`	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).

1.2 Brainfuck

The brainfuck[5,6] programming language is a minimalistic language. It actually implements a Turing machine [8, 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.

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.

[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.

[8] This listing was copied from the Wikipedia page on brainfuck (the URL was given above) as of 2009-03-18 16:38 CET.

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

```
++++++++++          initialises cell zero to 10
[
   >+++++++>++++++++++>+++>+<<<<-
]                   this loop sets the next four cells to 70/100/30/10
>++.                print   'H'
>+.                 print   'e'
+++++++.                    'l'
.                           'l'
+++.                        'o'
>++.                        space
<<+++++++++++++++.          'W'
>.                          'o'
+++.                        'r'
------.                     'l'
--------.                   'd'
>+.                         '!'
>.                          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 occasionally accesses the adjacent object environment, though, to achieve certain things. It also makes use of the PEG implementation available for COLA. The file containing the implementation is called `brainfuck.k`.

At first sight, the source code is divided into two main sections (ignoring the license), spanning lines 27–41 and 42–75, respectively. This separation into two sections is typical of such language implementation 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.

2.1 Preliminaries

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 operation:

```
28 (define putchar (dlsym "putchar"))
29 (define getchar (dlsym "getchar"))
```

The two well-known functions `putchar()` and `getchar()` 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. Note that `dlsym` is predefined in the COLA function environment and provides a basic wrapper to the standard library function of the same name.

It is important to note that the names bound to values by the code mentioned above indeed represent valid C function pointers. It is equally easy to *define* C functions in the COLA function environment. The code

```
(define myfunc (lambda (x) (printf "%d\n" (+ x 23))))
```

corresponds to the C function

```
int myfunc(int x) {
    return printf("%d\n", x + 23);
}
```

with the notable difference that the former is *dynamically* compiled as the COLA parser encounters it.

Having bound the `putchar` and `getchar` functions, the implementation moves on to defining the memory available to brainfuck programs:

```
(define mem-size 32768)
(define memory (calloc mem-size 1))
```

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 `calloc` and bound to the name `memory`. The `calloc` invocation initialises all array elements to zero.

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

At first sight, these two definitions look much like the definitions of ordinary functions, like seen above for `myfunc`: 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.[10]

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.

[10]One might, and quite rightfully so, get the idea that the COLA environment—supporting S-expression C ASTs, quasiquotation and dynamic compilation—is a very, *very* sophisticated C preprocessor and compiler.

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 runtime, 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 two parameters called `node` and `compiler`. The `node` parameter is a representation of the AST node currently being visited by the COLA function part compiler. It is an *object* to which messages can be sent—more precisely, it is a `SequenceableCollection`[11]. 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.

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 compiled.

It is important to note that calling syntax definitions "macros" is inaccurate, which has something to do with the presence of the second argument called `compiler`, which is not used in this 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.k` file contains the most important part: the definition of the language's grammar and behaviour. Lines 44–75 make up the entire thing. Given that brainfuck is such a simple language, this should 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

[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`.

more repetitions, and *, zero or more. Braces (()) are used to denote groups, and the vertical bar (|) marks alternatives. There are also some special elements that will be explained below.

Note that the entire grammar definition is written in the following form (heavily abbreviated and abstracted):

```
[ '{ ... definitions ... } name: grammar-name ]
```

This is nothing but a COLA message send (note the square brackets) where the `name:` message is sent with the name of the grammar as parameter. The name is quoted; this turns it into a symbol (i. e., a unique string). The grammar itself is also a quoted object represented by a set of definitions in curly braces. Details about these structures are given below in section 4.

2.2.1 Rules for Terminal Symbols

A bottom-up approach is best suited to provide a good understanding of the concepts at work in the brainfuck implemenattion 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      = ']'

    bfsymbol  =
        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 White Space

Before we can move on to processing brainfuck instructions, we need to clarify what white space is in a brainfuck program—recall that *any* character that is *not* one of the symbols listed in Table 1 is to be ignored by the implementation. So, in essence, anything that is not a brainfuck terminal symbol is considered white space:

```
    _ = (!bfsymbol .)*
```

The white space rule is given the name _. This is just a useful convention: that way, the name of the white space rule is quite unobtrusive.

The way the white space rule is constructed is more interesting than its name. Its purpose is to consume any sequence of zero or more characters that are *not* brainfuck symbols, i. e., input that is not consumed by the `bfsymbol` rule. This is achieved in the following way. The sequence is indicated by the star (`*`) attached to the entire expression. The dot (`.`) consumes a single character. In this special case, however, it only does so if a certain condition holds, which is expressed by the `!bfsymbol` expression. It applies the `!` (negation) predicate to the `bfsymbol` rule; the expression succeeds if `bfsymbol` does *not* match.

2.2.3 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 =
    _
    ( forward                        <- '(inc P)
    | backward                       <- '(dec P)
    | increment                      <- '(inc (char@ P))
    | decrement                      <- '(dec (char@ P))
    | put                            <- '(putchar (char@ P))
    | get                            <- '(set (char@ P) (getchar))
    | while _ instructions->0 _ wend <- `(while (!= 0 (char@ P)) ,[self @ '0])
    )
```

Looking at the rule definition from a high level, there is an alternative for each brainfuck symbol. The parts after left arrows (`<-`) are *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 do not actually *do* anything, 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 after the rule name,

```
instruction =
    _
```

consumes white space. Any occurrence of a brainfuck symbol may be preceded by white space. 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 _ instructions->0 _ wend <- `(while (!= 0 (char@ P)) ,[self @ '0])
        )
```

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 section 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[12].

The `put` and `get` symbols are handled 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 _ instructions->0 _ wend <- `(while (!= 0 (char@ P)) ,[self @ '0])
```

matches an entire "loop", starting with `while` and ending with `wend`, with a number of instructions (and possible white space) in between. This rule part refers to the `instructions` rule, which will be discussed below.

The parser maintains internal storage attached to the rule it currently evaluates. This "stack frame" can be accessed by index, starting from 0. Elements of a rule can store whatever is the result of their matching in these locations. The result of matching the `instructions` rule—an AST—is stored in parser "stack frame" at index 0, which is achieved by the `->` construct.

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.

The argument to the unquote operation is interesting: `[self @ '0]` is a message send. The receiver, `self`, is the current parser, which is sent the dereferencing message `@`. This realises an access to the parser's internal storage associated with the active rule, in this case, at index 0[13], and so retrieves the result of matching the `instructions` rule (see

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

[13] The number 0 is quoted (`'0`) to make it available as an object—message sends can only handle objects as both receivers and parameters.

above).

2.2.4 Matching Instruction Sequences and Building ASTs

The `instructions` rule was already mentioned above:

```
    instructions =
        instruction*->0 <- `(let () ,@[self @ '0])
```

While the `instruction` rule matches a single brainfuck instruction and returns a corresponding AST representing its behaviour, the `instructions` rule returns a sequence of ASTs for single instructions.

To achieve this, it makes use of the `instruction` rule, which it accepts multiple times. The result of these multiple matches is a sequence of ASTs[14], which is stored at index 0 in the parse node's internal storage. The AST returned from the `instructions` rule is a `let` expression (corresponding to a block in curly braces in C) containing all the AST bits generated for the matched expressions. The unquote syntax is different from above; `,@` unquotes a *collection* of elements and textually inlines them in the location where unquoting takes place.

2.2.5 Parsing and Executing a Program

The final rule of the brainfuck grammar is the `program` rule. When executing a brainfuck program, this rule is to be taken as the starting rule for the parser. The rule does not do much more than pass control to the `instructions` rule—a program *is* a sequence of instructions, after all—and consuming all remaining white space:

```
    program =
        instructions _ <- (let () [result eval] (printf "\n"))
```

The interesting part of the `program` rule is its action part, which consists of a single `let` expression—*not* a quoted or quasiquoted AST, which means that this action part is executed and its value returned. The result of parsing an entire program is implicitly available in the `result` object[15], which is sent the `eval` message. This triggers dynamic compilation of the entire AST and executes the resulting native code immediately.

The final instruction in the rule's action part just prints a newline to clean up the console.

3 Running Brainfuck Programs

In the following, it is assumed that the `brainfuck.k` file resides, along with some brainfuck program files with the suffix `.bf`, in the `function/examples2/brainfuck` directory of a complete COLA checkout. The version of COLA the presented implementation is

[14]The three EBNF operators `*`, `+`, and `?` return sequences of ASTs, which may be empty in case `*` matches zero times or `?` does not match at all.

[15]This object can also be sent the `print` message, which will dump the entire AST to `stdout`. Just give it a try.

tested on is SVN revision 646; the author does not guarantee it to work properly on any other revision.

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

```
$COLA2/main $COLA2/boot.k brainfuck.k hello.bf
```

where `$COLA2` references the directory where version 2 of the COLA function environment resides[16].

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 `brainfuck.k`, which contains the brainfuck implementation. Finally, `hello.bf` will be read and executed.

The COLA parser needs to be instructed to switch to a different input language as defined by some particular grammar. In the case of the brainfuck grammar, whose name is `brainfuck`, the command to switch is `{ brainfuck-program }`. All files containing brainfuck programs *must* begin with a line containing this command. The rationale behind this will be described in the next section.

4 COLA Grammars and Parsers

This section gives a more thorough introduction to the inner workings of COLA grammars and parsers. Both are represented by objects and are thus available as first-class entities in the COLA environment. For grammars, represented by `Grammar` objects, there is a special syntactic form—they are written down in curly braces as a set of rule definitions with, possibly, action parts. Evaluating a `Grammar` will yield a `Parser` object, which applies the grammar's rules to input passed to it.

There are four things to consider, which will be dealt with below, namely (1) the syntactic form that generates a `Grammar` object; (2) the protocol understood by `Grammar` objects; (3) evaluating a `Grammar` object to construct a `Parser` object; and (4) the side-effects of constructing a `Parser` object.

4.1 Grammar Literals

Recall the abstracted example for a "language definition" given in section 2.2:

```
[ '{ ... definitions ... } name: 'grammar-name ]
```

The part that actually defines the grammarthen the syntactic form of a grammar definition is `{ ... definitions ... }`, which is the special syntax in COLA to represent `Grammar` objects.

`Grammar` objects have a useful protocol, including the `name:` message used to give it a name, which was used in the brainfuck implementation. Table 2 gives a brief overview of the `Grammar` protocol.

[16] For the COLA SVN revision 646, this executable is `function/jolt2` relative to the COLA installation directory.

`startRule`	returns the start rule of this grammar
`name: aSymbol`	assigns the grammar a name
`named: aSymbol`	retrieves the grammar with the given name from all grammars
`startSymbol: aSymbol`	sets the start symbol for the grammar
`parserOn: aStream`	returns a parser for this grammar reading from the stream
`printOn: aStream`	prints the grammar on the given stream

Table 2: The protocol of `Grammar` objects.

```
... preliminaries ...
{
    forward   = '>'
    ...
    program =
        instructions _ <- (let () [result eval] (printf "\n"))
}
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<++++++++++++++
+.>.+++.------.--------.>+.>.
```

Listing 3: An instantly executing version of the brainfuck “Hello, world!” application.

Using the curly braces syntax, grammars can be represented literally in code, which is convenient. Unlike most other literals—e. g., strings or numbers—though, grammar literals do not evaluate to the corresponding objects one would expect—in this case, `Grammar` objects. Instead, they evaluate to `Parser` objects. For this reason, it is necessary to quote a grammar literal in order to assign it a name as in the sample brainfuck implementation, or in the abstracted example above and in section 2.2.

4.2 Parsing Details

Recapitulating what was said above, an expression `{ ... definitions ... }` will cause the COLA reader to create a grammar object; the read-eval-print loop will then evalate it to make a `Parser`. The last step of evaluating a `Grammar`, once the `Parser` has been created, is to invite that `Parser` to run its start rule to consume as much input as it wants to take from the current input stream.

By default, the start rule of a grammar is its *final* rule. In the case of the brainfuck grammar, this holds for the `program` rule. With this in mind, it occurs that the brainfuck version of “Hello, world!” could have been written as an “instant executable” carrying its own language implementation as shown in Listing 3. Note how the grammar is, in this case, *not* quoted, and how the brainfuck code immediately follows it.

The last rule in a grammar definition does not have to be named, and there is one special case that involves such unnamed rules. Consider a grammar literal of the form `{ x-y }`—it only consists of one unnamed rule of the form `x-y`. As mentioned above, this rule implicitly becomes the start rule of the grammar. Rules of the form `x-y` actually hand over control to a parser for the rule `y` in grammar `x`, leading to that parser consuming input until the rule has been processed. After that, control is returned to the outer parser from whose context `x-y` was started. Whatever parsing the `y` rule in grammar `x` returns will be consumed as input by the outer parser.

Putting it all together is best done with a concrete example: take any of the files containing brainfuck programs. The `{ brainfuck-program }` "statement", with which the file begins is an example of a grammar literal handing over control to another parser by means of an unnamed start rule. The `program` rule in the `brainfuck` grammar consumes all further input until the `program` rule finishes, which will happen when end-of-file is reached.

The result from parsing brainfuck instructions—a COLA AST in S-expression form—is returned from the brainfuck parser and consumed by the outer parser. This happens to be the standard parser of the COLA function environment, and what it does when it consumes ASTs is to generate native code and execute it.

This is how the brainfuck JIT compiler actually comes into being.

5 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.

Acknowledgements

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, Michael Grünewald, and Christopher Schuster for their suggestions for improvement.

A Complete Source Code

```
;;; brainfuck implementation using the function part of COLA
;;;
;;; This particular implementation runs in version 2 of the environment.

;;; License (MIT License)
; Copyright (c) 2009 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
; 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 functions
(define putchar (dlsym "putchar"))
(define getchar (dlsym "getchar"))

;;; the memory
(define mem-size 32768)
(define memory (calloc mem-size 1))

;;; the pointer
(define P memory)

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

;;; grammar and language implementation

['{
    forward   = '>'
    backward  = '<'
    increment = '+'
    decrement = '-'
    put       = '.'
    get       = ','
    while     = '['
    wend      = ']'

    bfsymbol  =
        forward | backward | increment | decrement | put | get | while | wend

    _ = (!bfsymbol .)*

    instruction =
        _
        ( forward                     <- '(inc P)
        | backward                    <- '(dec P)
        | increment                   <- '(inc (char@ P))
        | decrement                   <- '(dec (char@ P))
        | put                         <- '(putchar (char@ P))
        | get                         <- '(set (char@ P) (getchar))
        | while _ instructions->0 _ wend <- `(while (!= 0 (char@ P)) ,[self @ '0])
        )
```

```

    instructions =
        instruction*->0 <- `(let () ,@[self @ '0])

    program =
        instructions _ <- (let () [result eval] (printf "\n"))
} name: 'brainfuck]
```

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, Extensible Object Models. In *Self-Sustaining Systems*, volume 5146 of *LNCS*, pages 1–30. Springer, 2008.

[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] C. Petzold. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine.* Wiley, 2008.

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