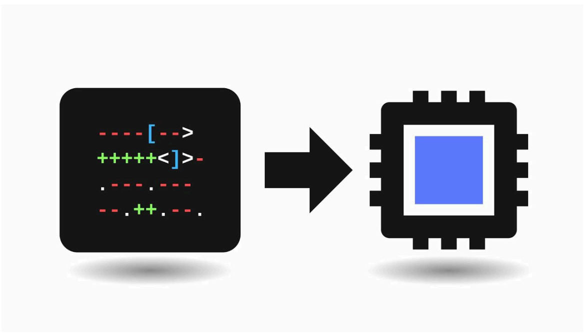
A Comprehensive Essay on tsoding's JIT Compiler for Brainf*ck

Campita, Renz Andrei O.
CD-1L
2025-05-03

Introduction



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

I made JIT Compiler for Brainf*ck lol (Tsoding Daily, 2024)

1. *What is the effect of old PLs to the one you have chosen to evaluate? Do you think the PL you have chosen has considerably improved the process of lexical analysis or has it preserved a lot of the characteristics defined by earlier PLs?*

Brainf*ck was invented by Urban Müller as an attempt to make language that would require the smallest compiler. It was inspired by Wouter van Oortmerssen's esoteric langauge in 1993 – FALSE (named after the author's favorite truth value).

The FALSE programming language had features such as: literals, a stack, arithmetic, comparison, lambda and flow control, identifiers, I/O, and comments. This is an extremely simple language that had a **1024-byte compiler**. In comparison, Brainf*ck was even simpler only having a pointer, an array of 30,000 memory cells, and 8 operations. Müller was able to write a compiler for this language using only **240 bytes**.

Brainf*ck is one of the most popular esoteric programming languages both due to its name and features. With its goal to be a language that would require an extremely small compiler, it has been a staple language to use for recreational programming.

This paper will analyze one of these imlementations: Alexey Kutepov[1]'s (aka. tsoding) Brainf*ck JIT Compiler[2]. This implementation has been chosen because it also has an accompanying YouTube video[3], a recording of the livestream when tsoding created the compiler from scratch. This shows the whole process of creating the compiler, and all of the decisions tsoding made.

To have a general grasp of the language, here is how to display "Hello World!" in Brainf*ck:

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

Code Snippet 1: Hello World! in Brainf*ck

[1] https://github.com/rexim
[2] https://github.com/tsoding/bfjit
[3] https://youtu.be/mbFY3Rwv7XM

Lexical Analysis

```
typedef struct {
  Nob_String_View content;
  size_t pos;
} Lexer;


bool is_bf_cmd(char ch)
{
  const char *cmds = "+-<>,.[]";
  return strchr(cmds, ch) != NULL;
}

char lexer_next(Lexer *l)
{
  while (l->pos < l->content.count
&& !is_bf_cmd(l->conent.data[1->pos])
  {
    l->pos += 1;
  }
  if (l->pos >= l->content.count)
return 0;
  return l->content.data[l->pos++];
}
```

Code Snippet 2: tsoding's Lexer implementation

In tsoding's implementation of the Brainf*ck JIT compiler, the lexer is extremely minimal. Its main purpose is only to clean the scoure code by removing all invalid characters.

Character	Operation
+	Increment Data
-	Decrement Data
>	Move Pointer Right
<	Move Pointer Left
.	Output Data
,	Input Data
[	Jump Forward
]	Jump Backward

Table 1: All valid characters in Brainf*ck

Comments

Since there is a fixed set of valid characters, it is trivial to implement comments. All other chracters are treated as comments and are removed by the lexer. In tsoding's implementation, all comments cannot include any of the valid

Tokens

Since in Brainf*ck, each of the operations are represented by a single character, tokenization is not needed. The lexer will just give the parser the cleaned up source code.

In the life span of Lexical Analyzers, we have mentioned that some of its functions and features are already merged in other software besides lexical analyzers themselves. What characteristics of the lexical analyzer makes it easy to merge it with these other software? and do you think it would be possible to do it with other phases of the compiler?

"functions and features already merged in other software -> text editors; linters"

It is easy to merge with other software since the grammar of the language is already defined and the underlying semantics of the language is not yet processed at this stage, making implementation of these features lightweight enough for supporting software such as text editors or linters to execute said features in real time.

Do you think it would be possible to do it with other phases of the compiler?

It could be possible to also do it with other phases of the compiler. An example of this is Language Server Protocol (LSP) servers that the most popular text editors or IDEs use. Well implemented LSPs can give programmers warnings about unused variables, bounds checking, type mismatches, etc.

For the later phases of the compiler, these are harder to implement but some do. Just in time (JIT) compilers are examples of these other software that merges compilation steps into its own process. JITs generate intermediate representation before it generates machine code.

Syntax Analysis

tsoding's implementation of the Brainf*ck JIT compiler does not produce a parse tree. Since Brainf*ck source code basically processes each character as its own "expression" – as what other programming languages may treat them, the compiler does not need to generate a parse tree for each expression.

For example, here is a comparison between C and Brainf*ck.

C:

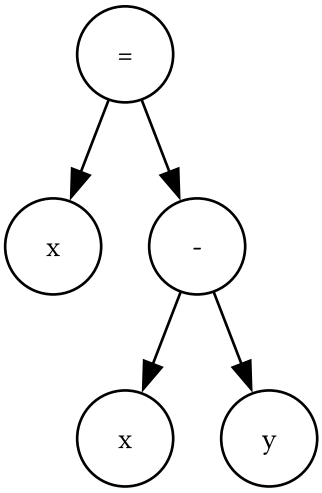
```
int main() {
  int x = 0;
  while (x < 5) {
    x++;
  }

  print("%s", x);

  return 0;
}
```

Code Snippet 3: Printing "5" in C

Parse tree for the statement `x = x - y`:



Brainf*ck:

```
+
+
+
+
+
.
```

Code Snippet 4: Printing "5" in Brainf*ck

Since Brainf*ck does not require a parse tree, tsoding's implementation of the parser only has these functions:

- Transform the source code into Intermediate Representation
- Check for unbalanced brackets

3. *Syntax Analysis can be implemented with varying complexity. What are the impacts of this varying complexity in Syntax Analysis in the other phases of the compiler?*

The work that the syntax analyzer or the parser does with the tokens –or in Brainf*ck's case, the source code tself– can greately affect the proceeding steps in compilation. In general, the more processing the parser does: syntax errors, intermediate code generation, parenthesis balancing, etc., the less the following steps has to take care of.

4. *How about the usage/popularity of the compiler? Does the complexity of the compiler play a role in that?*

The complexity of the compiler also affects its popularity. Giving a simple interface that programmers can interact with, with good documentation will also increase the communication between programmers using the same language.

Semantic Analysis

tsoding's implementation does not have semantic analysis. This is not required as there are no semantics that has to be analyzed. If anything, maybe this step can optimize commonly used patterns in Brainf*ck such as the `[-]` pattern, which sets the current memory cell to zero.

1. Many of the programming languages that have been created over the years have improved and made steps to automate some parts of the compilation process to both improve performance and lessen errors in the side of the programmers. These are easily implementable in some of the phases of the compiler. How about in Semantic Analysis? is this also the case? Is it different?

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

2. Type Checking has been around for quite some time. Looking at older PLs, type checking has been considered to be less of a priority compared to other parts of the compiler. Discuss how this increase in importance in typse checking came about.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Intermediate Representation

```c
    typedef enum {
        OP_INC             = '+',
        OP_DEC             = '-',
        OP_LEFT            = '<',
        OP_RIGHT           = '>',
        OP_OUTPUT          = '.',
        OP_INPUT           = ',',
        OP_JUMP_IF_ZERO    = '[',
        OP_JUMP_IF_NONZERO = ']',
    } Op_Kind;

    typedef struct {
        Op_Kind kind;
        size_t operand;
    } Op;
```

Code Snippet 5: tsoding's Operation structure

3. Intermediate Representation that we have discussed include DAGs and Three-Address Code Representations. Are these the only representations that exist in PLs or are there others? If there are, what advantages do these other approaches bring to the table? If there are non, why do you think PLs have stuck with this approach?

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Code Generation

1. The determination of Basic Blocks and Creation of Flow Graphs is one prominent example of an algorithm for code generation. Are there different approaches done in other PLs? IF so, give an example.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

Code Optimization

2. Code Optimization is a very broad topic but we only focus on code-improving transformations that are machine-independent. Besides proper memory management, what other code optimization techniques are existing? Use your chosen PL to discuss these techniques

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet, ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum defuturum, quas natura non depravata desiderat. Et quem ad me accedis, saluto: 'chaere,' inquam, 'Tite!' lictores, turma omnis chorusque: 'chaere, Tite!' hinc hostis mi Albucius, hinc inimicus. Sed iure Mucius. Ego autem mirari satis non queo unde hoc sit tam insolens domesticarum rerum fastidium. Non est omnino hic docendi locus; sed ita prorsus existimo, neque eum Torquatum, qui hoc primus cognomen invenerit, aut torquem illum hosti detraxisse, ut aliquam ex eo est consecutus? – Laudem et caritatem, quae sunt vitae.

3. We talk generally about the necessity of each of the phases of the compiler. Nowadays, with the improvemnt of hardware, what would be the effect of taking off or skipping this phase of the compiler? Would it be beneficial or not?

The essence of a compiler is to translate source code into machine code that a particular hardware can execute. This Brainf*ck JIT compiler shows that not all phases of the compiler is strictly necessary. Due to the simplicity of the language itself, it is almost a trivial step to translate it to machine code.

Considering constant improvements in hardware with newer processors and optimizations in processors such us out of order execution, it is difficult to tell the effect of removing steps in a compilation process. It is possible that the CPU itself would recognize hotspots in the program it is executing and optimize it on the fly, or even do optimizations that the programmer or the compiler would catch. The only real and reliable way to know the effects of modifying the compiler is to empirically test and measure performance in specific hardware configurations.