CSN-352

BRAIN-F*CK COMPILER

LLVM-LITE IMPLEMENTATION (GID 7)

Course Code: CSN-352
Course Name: Compiler Design
Course Instructor: Prof. Sudip Roy

Group Members

Name	Enrollment Number
Tanmay Mohit Shrivastav	**21114103**
Shrey Gupta	**21112103**
Raiwat Narendra Bapat	**21114078**

Contents

Introduction

Overview

The LLVM Compiler Infrastructure offers a robust set of modular technologies tailored for compiler development, featuring a platform-independent Intermediate Representation (IR) that allows for thorough code optimization, along with a flexible backend that facilitates the generation of optimized machine code across multiple architectures.

In the context of our project, we utilize LLVM's capabilities through the llvmlite library to develop a specialized compiler for the Brain-f*ck programming language. This project aims to leverage LLVM's optimization mechanisms to enhance the performance and efficiency of Brain-f*ck programs. By transforming Brain-f*ck code into LLVM's IR, we harness a standardized optimization framework that can adapt to different hardware configurations, ensuring our compiler is both versatile and powerful.

Problem Statement

Develop a versatile command-line utility in Python that enables users to efficiently compile Brain-f*ck programs into LLVM Intermediate Representation (IR), with additional functionalities for optimization, execution, and output management. The utility should leverage the LLVM compiler infrastructure to generate optimized machine code, offering a seamless bridge between the simplicity of Brain-f*ck and the performance benefits of modern compiler technologies.

Brain-F*ck Syntax

Brain-f*ck is a programming language renowned for its extreme minimalism and unconventional syntax, making it a favorite among enthusiasts seeking to delve into

esoteric languages. Its deliberately cryptic design encourages developers to think creatively and approach problems from a unique perspective.

At its core, Brain-f*ck operates on a tape-like data structure consisting of an array of cells, each capable of holding a single byte. The tape is initially set to a fixed length of 30,000 cells, with a data pointer positioned at the beginning of the array. Programmers manipulate this pointer and the values in the cells using a set of just eight commands.

Brain-F*ck has only 8-commands that are listed below:

- '>' increments the data pointer (to point to the next cell to the right).
- '<' decrements the data pointer (to point to the next cell to the left).
- '+' increment (increase by one) the byte at the data pointer.
- '–' decrement (decrease by one) the byte at the data pointer.
- '.' output the byte at the data pointer.
- ',' accepts one byte of input, storing its value in the byte at the data pointer.
- '[' if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.
- ']' if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

Despite its minimalism, Brain-f*ck challenges programmers to think in terms of low-level manipulation and control flow, offering a unique glimpse into the fundamental principles

of computation. While it may not be practical for everyday programming tasks, its role in pushing the boundaries of compiler design and language theory cannot be overstated.

About LLVM

LLVM is an open-source compiler infrastructure project designed to optimize compile-time, link-time, runtime, and idle-time performance across a variety of programming languages. It provides a modern SSA-based compilation framework, enabling sophisticated compiler transformations and easy code generation and analysis. LLVM supports a wide range of tools and libraries, enhancing code development and execution efficiency.

In our project in order to make use of LLVM functionalities we have used the LLVM lite library.

The LLVM lite library in Python, known as llvmlite, is a lightweight wrapper around the LLVM C++ library, focusing on providing a small subset of its functionality in a Python-friendly way. llvmlite is specifically designed to support JIT compilation and as a back-end for the Numba JIT compiler, allowing Python functions to be dynamically compiled to machine code.

Key Benefits of LLVM lite

The older llvmpy binding provided extensive access to LLVM APIs, but its translation of C++ memory management to Python was prone to errors. Many JIT compilers, including Numba, do not require the entire LLVM API suite. Instead, only the essential components such as the IR builder, optimizer, and JIT compiler APIs are necessary.

The most significant features of llvm-lite include the following:

- The IR builder is pure Python code and decoupled from LLVM's frequently-changing C++ APIs.

- Materializing a LLVM module calls LLVM's IR parser which provides better error messages than step-by-step IR building through the C++ API (no more segfaults or process aborts).
- Most of llvmlite uses the LLVM C API which is small but very stable (low maintenance when changing LLVM version).
- The binding is not a Python C-extension, but a plain DLL accessed using ctypes (no need to wrestle with Python's compiler requirements and C++ 11 compatibility).
- The Python binding layer has sane memory management.
- llvmlite is faster than llvmpy thanks to a much simpler architecture (the Numba test suite is twice faster than it was).

Implementation

Tokenization

Since each token in Brain-f*ck corresponds directly to a single character, tokenization does not require complex pattern matching or lexical analysis as with more syntactically dense languages. Instead, tokenization can be efficiently accomplished by simply splitting the input code into a stream of individual characters. Each character in Brain-f*ck directly represents a distinct command within the language, making this approach both straightforward and effective.

Furthermore, in Brain-f*ck, any characters that do not correspond to recognized commands are automatically treated as comments. This feature simplifies the parsing process, as the compiler can ignore these characters, focusing only on the valid command characters. This approach ensures that any extraneous text or symbols within the code, possibly used for readability or annotations by the programmer, do not affect the execution logic but are seamlessly integrated as part of the program's documentation.

Creating Abstract syntax trees

In programming languages, an AST is a tree representation of the abstract syntactic structure of source code written in a language. Each node of the tree denotes a construct occurring in the source code.

For each character, we determine the appropriate structure in the AST based on Brain-f*ck's syntax rules:

- If a character is an opening loop bracket [, it signifies the start of a nested structure. The function recursively calls itself to handle the inner loop, appending the result (a nested list representing the loop's content) to the top-level list.
- Characters other than brackets are added directly to the current list, as they represent direct operations within the current level of the program or loop.
- Upon encountering a closing loop bracket], the recursion unwinds, marking the end of the current loop's parsing.

The end result is a nested list structure where the top-level list represents the sequence of operations and loops at the outermost level of the Brain-f*ck program. Nested lists represent the content of loops, capturing the hierarchical structure of the code in a tree-like form.

Implementing the Brain-f*ck syntax in the LLVM-lite framework

- Our Python script is designed to transform Brain-f*ck code into LLVM's intermediate representation (IR) and, if desired, execute it using LLVM's just-in-time (JIT) compiler. The script begins by defining functions that create an abstract syntax tree (AST) from the Brain-f*ck code, which it then converts into LLVM IR. This setup involves initializing the LLVM environment and defining

specific LLVM types and functions that correlate with Brain-f*ck operations, such as putchar, getchar, and various memory manipulation commands.

- In the phase of LLVM IR generation, the script is responsible for managing resources such as allocating memory for the tape and index pointer, which are crucial components in Brain-f*ck programs. It includes defining helper functions that assist in calculating tape memory locations and transforming Brain-f*ck instructions into corresponding LLVM IR instructions.
- The core functionality of the script leverages LLVM's robust capabilities to dynamically compile and execute the generated IR code. This is facilitated through a user-friendly command-line interface that allows users to input a Brain-f*ck code file and, if chosen, run it using LLVM's McJIT engine. The McJIT engine dynamically compiles the LLVM IR into native machine code, thus executing it directly on the host machine.
- This script not only makes it possible to execute Brain-f*ck programs with greater efficiency but also utilizes LLVM's advanced optimization and compilation features to enhance performance. By bridging the gap between the simplicity of Brain-f*ck and the sophisticated architecture of LLVM, our tool offers a unique solution that harnesses the power of modern compiler technology to handle even the most minimalist programming languages effectively. This approach significantly boosts execution speeds and optimizes runtime performance, making it a valuable tool for developers exploring the capabilities of both Brain-f*ck and LLVM.
- Overall,our script bridges the gap between Brain-f*ck and LLVM, allowing Brain-f*ck programs to be efficiently executed through LLVM's optimization and compilation capabilities.

Test Cases

As previously mentioned in our Mid-term report, the following input test cases cover basic functionalities of a brainfuck compiler such as input and output stream along with increment and decrement operations without utilizing the concepts of loops.

- **Take_input.bf**: Takes input 3 numbers and prints them.

```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/take_input.bf
123
123
```

- **Cab_bois.bf**: Prints “Cab bois”.

```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/cab_bois.bf
CAB BOIS
```

- **String_rev.bf**: Reverses a 5 letter string.

```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/string_rev.bf
apple
elppa
```

- **Arithmetic.bf**: Takes input 2 numbers and adds a constant to one and subtracts the same from another and prints the final values for both the numbers.

```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/arithemetic.bf
46
91
```

The following test cases capture more advanced features of our compiler including the implementation of loops adhering to the goals stated in our mid way evaluation report. These test cases are one of the most complicated BF programs written by BF enthusiasts around the world hence showcasing the robust nature of our compiler.

- **helloworld_complex.bf:**Prints “Hello World!” using the loop functionality.

```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/helloworld_complex.bf
Hello World!
```

- **factor.bf:**Takes input an integer and outputs all prime factors for the same.

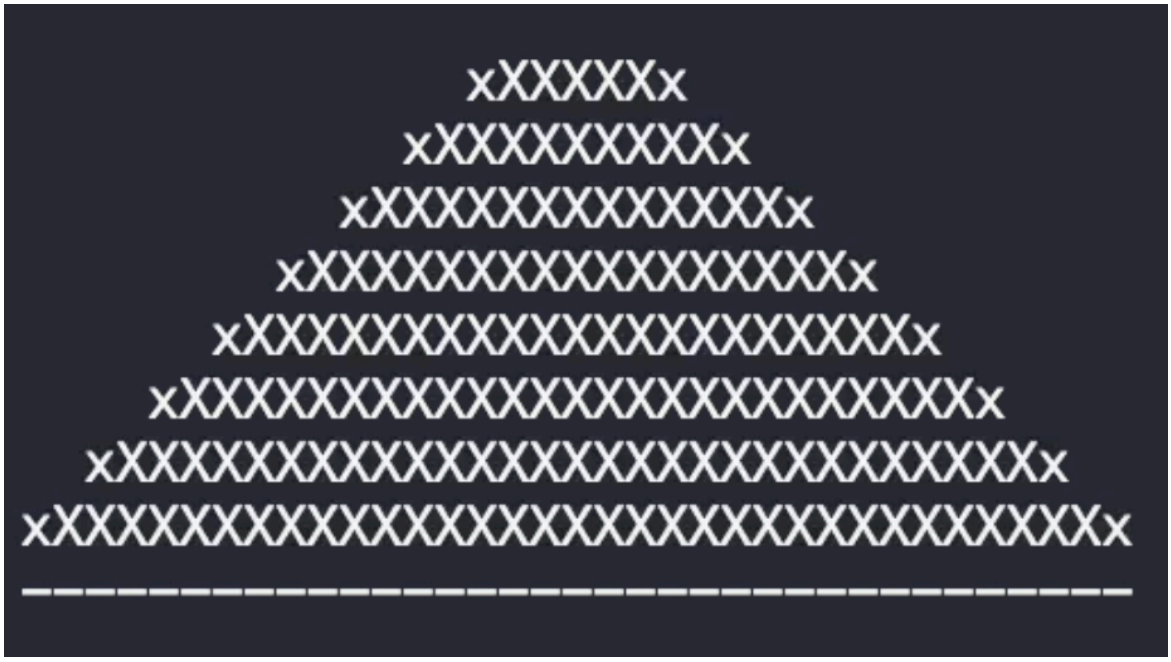
```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/factor.bf
8128
8128: 2 2 2 2 2 2 127
```

- **fibonacci.bf:**Prints first 11 terms of the fibonacci series**.**

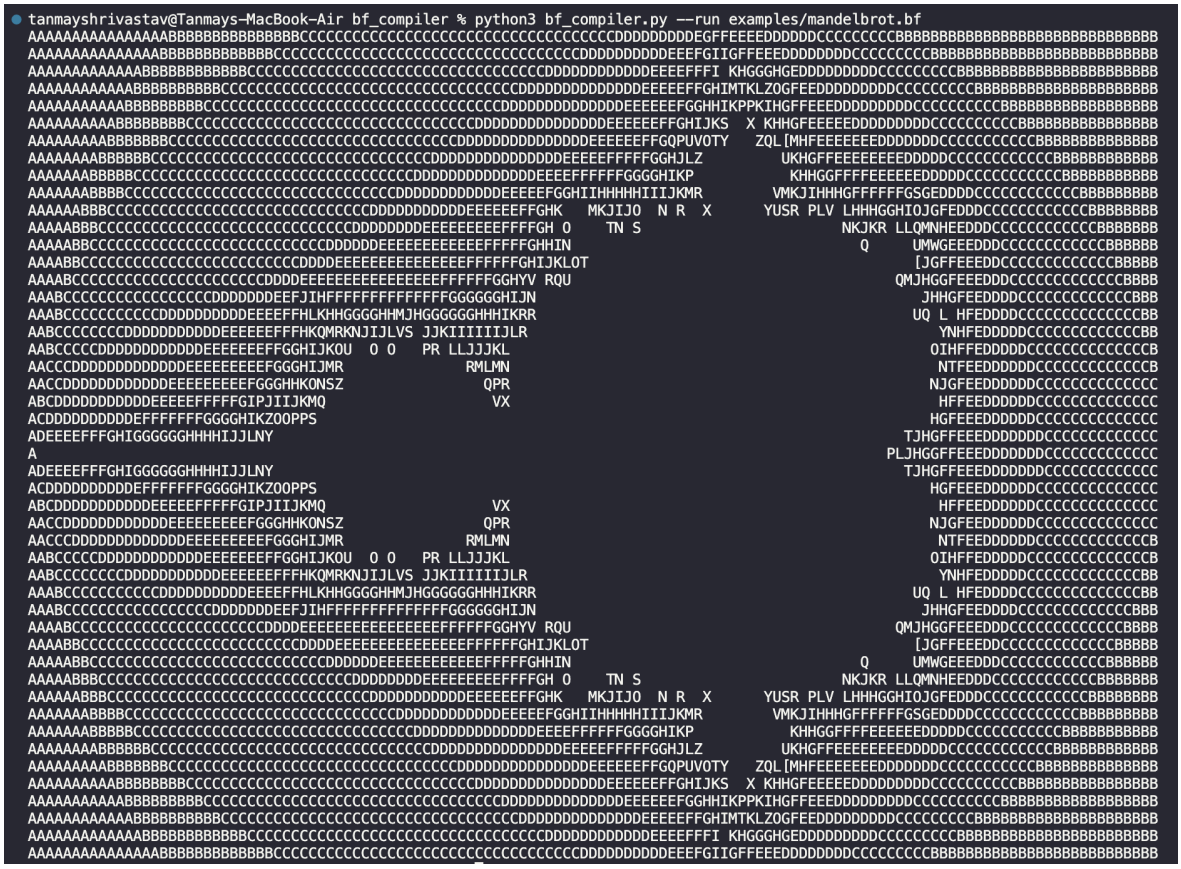
```
raiwat7@DESKTOP-LMO0CE6:/mnt/c/Desktop Folders/VSCode/BF-Compiler$ python3 bf_compiler.py --run examples/fibonacci.bf
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
```

- **hanoi.bf**:Solves the “Tower of Hanoi” problem and simultaneously simulates the entire solution.

  Click the link in the form of the image to see the simulation



- **mandelbrot.bf**: Prints the famous mandelbrot fractal as an ASCII image.



Repository

https://github.com/raiwat7/BF-Compiler