CSN-352

BRAIN-F*CK COMPILER
LLVM-LITE IMPLEMENTATION (GID 7)

Group Information

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

Introduction

Overview

The LLVM Compiler Infrastructure offers a set of modular technologies for developing compilers, providing a platform-independent Intermediate Representation (IR) for code optimization and a flexible backend for generating optimized machine code across various architectures.

Problem Statement

Develop a versatile command-line utility in Python that enables users to efficiently compile Brainf*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 Brainfck and the performance benefits of modern compiler technologies.

Brain-F*ck Syntax

Brain-f*ck language is an esoteric programming language designed to test the boundaries of compiler design and as a proof of concept. It intentionally refuses to follow the common ways of simplifying the complexity and clarifying the obscurity. Instead, it offers a challenge of breaking new ground within deliberately obscure rules. Brainf*ck stores data in the form of cells of 8-bit size(cells are bytes).In the classic distribution, the array of cells has 30,000 cells, and the pointer begins at the leftmost cell(Cell 0).

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.

Implementation

Tokenization

As each token in the Brain-F*ck language is a single character, tokenization can be achieved by simply splitting the input code into a character stream.

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.

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 translates Brainf*ck code into LLVM intermediate representation (IR) and optionally executes it using LLVM's just-in-time (JIT) compiler.
- The script first defines functions to generate an abstract syntax tree from Brainf*ck code and convert it into LLVM IR. It then initializes LLVM and defines LLVM types and functions corresponding to Brainf*ck operations such as `putchar`, `getchar`, and memory manipulation.

- During LLVM IR generation, it allocates memory for the tape and index pointer used in Brainf*ck programs. It also defines helper functions to calculate tape memory locations and compile Brainf*ck instructions into LLVM IR instructions.
- The script utilizes LLVM's capabilities to dynamically compile and execute the generated IR code. It provides a command-line interface to specify a Brainf*ck code file and optionally run it using LLVM's McJIT engine. The McJIT engine dynamically compiles the LLVM IR into native machine code and executes it.
- Overall,our script bridges the gap between Brainf*ck and LLVM, allowing Brainf*ck programs to be efficiently executed through LLVM's optimization and compilation capabilities.

Test Cases

At present, our Python script is equipped with functionalities limited to addition, subtraction, moving the pointer left or right, printing, and taking input. Consequently, we are restricted to testing our code to basic programs that avoid the use of loops and perform basic input,output and arithmetic operations.The subsequent output is as follows:

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

- **Cab_bois.bf**: Prints “Cab bois”.

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

```
tommy@LAPTOP-H4GUDQIF:/mnt/c/Users/Gupta/Desktop/IITsemwise/IIT STUDY MATERI
ng_rev.bf
Shrey
yerhStommy@LAPTOP-H4GUDQIF:/mnt/c/Users/Gupta/Desktop/IITsemwise/IIT STUDY M
```

- **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.

```
tommy@LAPTOP-H4GUDQIF:/mnt/c/Users/Gupta/Desktop/IITsemwise/IIT STUDY MATERIAL SEM-
ples/arithemetic.bf
45
90tommy@LAPTOP-H4GUDQIF:/mnt/c/Users/Gupta/Desktop/IITsemwise/IIT STUDY MATERIAL SE
```

The rest of the outputs will be documented comprehensively in the final report, encompassing the examination of all provided test cases alongside their respective outputs.

Future Work

1. - Include loop functionality ('[' and ']') into the compiler.
2. - Introduce a feature to translate Brainf*ck code into bitcode.
3. - Develop a capability to convert Brainf*ck code into an Intermediate Representation, saving it in a .ll file.
4. - Establish error handling to identify and handle all potential error cases efficiently.

Repository

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