B Compiler in Rust

1. Introduction

1.1. Context

In the late 1960s, Ken Thompson and Dennis Ritchie were working on the Multics operating system at Bell Labs. This was written in PL/I, an extremely complex language that required a heavy compiler. When Bell Labs pulled out of Multics, Thompson wanted to create a much more lightweight and portable operating system that could run on the much smaller PDP-7 computers. This system would later be developped into Unix.
The PDP-7 only had 8K words of memory, meaning that Thompson needed a new language that was small enough to fit on the computer, but powerful enough to be capable of writing an OS kernel. He took the already existing Basic Combined Programming Language (BCPL) and stripped away everything that he thought was unneccessary, resulting in the B language.
The key characteristic of B is that it is typeless. There exists only one data type, the machine word (called a "cell"). This made the entire computer essentially a giant array of words, making pointer arithmetic (such as `ptr + 5`) a key feature of the language.
While B is rarely used today, it's legacy is still hugely felt in the field of computer science. Dennis Ritchie would go on to invent the C language, who's syntax was based off of B. Many syntax features (such as `==`, `++`, `-`, and `+=`) were derived from the B language.

This project implements a lightweight modular compiler for the B programming language. The Rust programming language was used, leveraging its ownership model and ensuring performance. The target language is a custom three address code intermediate representation (TAC IR), which is executed on the B Virtual Machine (BVM).

1.2. Compiler Pipeline

Source Code

```
auto a, b, c;
a = 5;
b = 7;
c = a + b;
```

↓

Lexer

```
('a', '=', '5', ';', 'b'...)
```

↓

AST Generation

```
  c
 /\
a  b
|  |
5  7
```

↓

TAC IR

```
STACK[0] = 5
STACK[1] = 7
t1 = STACK[0]
t2 = STACK[1]
t3 = t1 + t2
STACK[2] = t3
```

This compiler follows the conventional framework for most modern modular compilers. The first process is lexing, where the source code (written in B) is tokenized. Note that the `auto` keyword at the beginning allocates memory (on the **`stack`**) for `a, b, c`. Then, the list of tokens are passed into a parser, where an abstract syntax tree (AST) is generated. From the AST, the TAC IR is created, which is finally passed to the VM.

1.3. Project Structure

Below is the directory tree of the project. Note that the `codegen/` (To translate TAC IR into x86 Assembly) is to be implemented in the future.

```
b-compiler/
├─ Cargo.toml
├─ src/
│  ├─ main.rs
│  ├─ common/
│  │  ├─ mod.rs
│  │  ├─ span.rs
│  │  └─ error.rs
│  ├─ lexer/
│  │  ├─ mod.rs
│  │  ├─ token.rs
│  │  └─ scanner.rs
│  ├─ parser/
│  │  ├─ mod.rs
│  │  └─ precedence.rs
│  ├─ ast/
│  │  ├─ mod.rs
│  │  └─ visitor.rs
│  ├─ sema/
│  │  ├─ mod.rs
│  │  └─ symbol_table.rs
│  ├─ ir/
│  │  ├─ mod.rs
│  │  ├─ tac.rs
│  │  └─ builder.rs
│  ├─ vm/
│  │  ├─ mod.rs
│  │  ├─ cpu.rs
│  │  └─ memory.rs
│  └─ codegen/
│     ├─ mod.rs
│     └─ x86_64.rs
└─ tests/
   ├─ hello_world.b
   └─ pointer_arithmetic.b
```

2. Lexer - (Ch. 6.1, 5.1, 3.5, 6.2, 18.3, 8.2, 4.3)