Introduction

We chose exercise category A: compilation. As there were two of us in the group, we decided to use imperative and object-oriented languages as our "destination" languages. We chose Rust as the imperative language and Javascript as the object-oriented one. Each of these programs were implemented and can be compiled as source-to-source from MiniImpPlus and tested on the command line with commands provided later.

The translator program is also implemented in Rust as we think it's an interesting language and we wanted to test how this kind of task could be done with it.

The documentation is generated using mdbook with optional mdbookPdf backend. Docs folder contains documentation in pdf and html formats. You can also build the documentation with mdbook.

```
# Build mdbook
mdbook build
# View in browser
mdbook serve --open
```

The MiniImpPlus test program can be found in project root in `testprogram.mip`-file. It's implementation follows the exercise description.

Authors / group members

- Tuomas Rinne, tumrin@utu.fi
- Teemu Salonen, tpsalo@utu.fi

Setup

This program is written in rust and therefore Rust compiler is needed to compile and run it. Rust can be installed from The Rust website. Both the translator program and the output of the Rust flag of the program have only been tested to compile with the newest stable(version 1.69) Rust compiler. Older versions may work but are not guaranteed.

Compiling

```
# In the project root
cargo build # build in debug mode
```

Using the translator

You can see the help menu with:

```
cargo run -- --help
```

If you want to see how the MiniImpPlus code is compiled to either of the exercise languages, it can be done by running the command.

```
cargo run -- javascript
# or
cargo run -- rust
```

Parser implementation

Parser uses the visitor pattern to traverese each node and produce code in the destination language matching that node. We select the language by using command line argument parsed by Clap crate. The argument is mapped to languages enum listing all available languages. The file type is also selected with this enum.

The parsing is done by antlr_rust crate. Every language supported implements ***ParseTreeVisitorCompat*** trait and the antlr generated ***MiniImpVisitorCompat*** trait which allows running code when certain type of node is visited. We only need to change the nodes that are different in target language. Other nodes are left with default implementation which is just calling visit children and returning the output of that Antlr itself does not support generating Rust code so we used a fork with Rust support.

The traits are implemented for a struct which has a single string field. This string is the return value of the visitor which contains translated source code of that language.

The end result of the parsing is written to a file titled **output.<filetype>**.

Issues

The parser gives some debug warnings, but they don't affect the output source code and the programs work fine.

Rust

After running the translator with the rust flag.

```
cargo run -- --rust
```

an output.rs file is created. This file can then be compliled by invoking the rustc directly.

```
rustc output.rs
```

This results in a binary file that can be run as any other program.

```
./output
```

Implementation details

As Rust requires variables to explicitly state that they are mutable with "mut" keyword, all variables are assumed to be mutable when translating from MiniImpPlus to allow set statement to change variable value.

Javascript

To compile the output.mjs file, the translator has to be run with a javascript flag.

```
cargo run -- javascript
```

The Javascript program can be run from the command line using NodeJS. Please make sure to use node major version 18, as the compiled program may not work on older versions as it requires support for top level async.

Running the program can be done by using the command

```
node output.mjs
```

Due to the eventloop of node, the readline was hard to use with synchronously and had to be implemented with async functions.