Stage 1 - Parser

This is the second stage of the ByteFrost Assembler pipeline. The **Parser** takes as input the `CommandLineArguments` object pointer generated by the `CLAP` and returns a `std::vector<Line>` vector that contains a `Line` object for each line in the input `.asm` file (assuming this file exists).

Definitions

Delimiter Set `delimiters`

The delimiter set is an `std::unordered_set<char>` stored by the `Parser` class that contains all characters that are understood to be delimiters in a line and which should be used as separators between token strings. The delimiter set `std::unordered_set<char> delimiters` is an implementation of the following set:

$DELIMITERS = {\texttt{SPACE}, \texttt{TAB}, \texttt{,}}$

`Token`s

A `Token` is a struct containing a `TokenType` enum and a token string:

```
struct Token {
    TokenType type;
    string token_string;
}
```

The Parser recognizes the following `Token` types, which are the values of the `TokenType` enum:

```
enum TokenType = {INSTRUCTION, GREGISTER, TEXT, SREGISTER, NUMBER, IMMEDIATE,
DIRECTIVE, LABEL, INVALID};
```

`INSTRUCTION`

A token string `w` is mapped to `TokenType::INSTRUCTION` if

1. `w` $\in TEXT$.
2. `w` $\in INSTRUCTIONS$.

`GREGISTER`

A token string `w` is mapped to `TokenType::GREGISTER` if

1. `w` $\in TEXT$.
2. `w` $\in GREGISTER$.

`TEXT`

A token string `w` is mapped to `TokenType::TEXT` if

1. `w` $\in TEXT$.
2. `w` cannot be mapped to `TokenType::INSTRUCTION` or `TokenType::GREGISTER`.

SREGISTER

A token string `w` is mapped to `TokenType::SREGISTER` if

1. The length of `w` is `>= 2`.
2. `w[0] = '%'`.
3. `w[1]...w[len(w) - 1]` $\in SREGISTER$.

NUMBER

A token string `w` is mapped to `TokenType::NUMBER` if

1. `w` $\in NUMBER$.

IMMEDIATE

A token string `w` is mapped to `TokenType::IMMEDIATE` if

1. The length of `w` is `>= 2`.
2. `w[0] = '#'`.
3. `w[1]...w[len(w) - 1]` $\in NUMBER$.

DIRECTIVE

A token string `w` is mapped to `TokenType::DIRECTIVE` if

1. The length of `w` is `>= 2`.
2. `w[0] = '.'`.
3. `w[1]...w[len(w) - 1]` $\in TEXT$.

LABEL

A token string `w` is mapped to `TokenType::LABEL` if

1. The length of `w` is `>= 2`.
2. `w[0] = ':'`.
3. `w[1]...w[len(w) - 1]` $\in TEXT$.

INVALID

A token string `w` is mapped to `TokenType::INVALID` if

1. `w` cannot be mapped to any `TokenType` that is not `TokenType::INVALID`.

Lines

A `Line` is a struct that contains all the information needed by the ByteFrost Assembler about a line in the input `.asm` file to generate an assembled file.

A `Line` object is defined in the following way:

```
struct Line {
    LineType type;
    string original_string;
    vector<Token> tokens;
    //  Maybe add a Union of possible line-related data here, such as
    //      Instruction, Directive, LabelDefinition, Empty
    //  the access format depends on the `LineType` (e.g., access the union as
    //  an Instruction if the LineType is LineType::INSTRUCTION)
    //  Alternatively, can handle this using inheritance from Line
}
```

The Parser recognizes the following `LineType`s:

```
enum LineType = {INSTRUCTION, DIRECTIVE, LABEL_DEFINITION, EMPTY, INVALID};
```

`INSTRUCTION`

A `vector<Token> tokens` vector is mapped to `LineType::INSTRUCTION` if

1. `tokens.size() >= 1`
2. `tokens[0].type == TokenType::INSTRUCTION`
3. There exists an instruction that has the name `tokens[0].token_string`.
    - Technically, this is a redundant condition given 4., but it may be simpler to check this first
4. For exactly one instruction that has the name `tokens[0].token_string`, the tokens in the rest of the `tokens` vector (i.e., `tokens[1]...tokens[tokens.size() - 1]`) match in type the expected token arguments of this instruction (and the number of arguments must match as well).

`DIRECTIVE`

A `vector<Token> tokens` vector is mapped to `LineType::DIRECTIVE` if

1. `tokens.size() >= 1`
2. `tokens[0].type == TokenType::DIRECTIVE`
3. There exists a preprocessor directive with the name `tokens[0].token_string.substr(1)` (i.e., the directive name doesn't include the `.` at the start of the directive token string)
4. Each `Token`'s `TokenType` in `tokens[1]...tokens[tokens.size() - 1]` matches with the corresponding `TokenType` in the directive's expected `TokenType` list, and the number of tokens (i.e., `tokenss.size() - 1`) must match the number of expected token arguments of the directive.

`LABEL_DEFINITION`

A `vector<Token> tokens` vector is mapped to `LineType::LABEL_DEFINITION` if

1. `tokens.size() == 1`
2. `tokens[0].type == TokenType::LABEL`

`EMPTY`

A `vector<Token> tokens` vector is mapped to `LineType::EMPTY` if

1. `tokens.size() == 0`

`INVALID`

A `vector<Token> tokens` vector is mapped to `LineType::INVALID` if

1. `tokens` cannot be mapped to any `LineType` that is not `LineType::INVALID`.

Parsing

The Parser generates the `std::vector<Line>` vector that represents the input `.asm` file's contents with the following steps:

1. Open the input file.

The Parser will attempt to open the input file for reading. If this step fails, the Parser will throw an error.

2. Generate a `std::vector<string> line_strings` vector

The Parser will create a `std::vector<string> line_strings` object and add each line read from the input file to this vector.

3. Generate a `std::vector<Line> lines` vector

This is the main step of the Parser. To accomplish it, the Parser will do the following for each line string `s` in the `std::vector<string> line_strings` vector:

1. Generate a string `s'` that contains no comments.

Given a line string `s`, `s'` is the substring of `s` from index 0 to the index of the start of the first instance of the comment string `//`. If `s` contains no comment strings, then `s' = s`.

2. Generate a `std::vector<string> token_strings` vector.

Allocate a `std::vector<string> token_strings` vector. The Parser iterates through each character in `s'`. For each character `c` in `s'`:

1. If $c \in DELIMITERS$, then set `isTokenString` to `false`.
2. Otherwise, set `isTokenString` to `true`.
3. If `isTokenString` is `true`: append `c` to `currTokenString`
4. Else if `isTokenString` is `false` and `currTokenString.length() > 0`:
    1. Add `currTokenString` to `token_strings`
    2. Set `currTokenString` to an empty string.

After running through every character in `s'`, if `isTokenString` is still `true`, then add the current `currTokenString` to the `token_strings` vector.

3. Generate a `std::vector<Token> tokens` vector.

Allocate a `std::vector<Token> tokens` vector. Iterate through each token string in `token_strings`. For each token string `w` in `token_strings`:

1. If $w \in TEXT$:
    1. If $w \in INSTRUCTIONS$, then add a token with `TokenType::INSTRUCTION` to the `tokens` vector.
    2. Else if $w \in GREGISTERS$, then add a token with `TokenType::GREGISTER` to the `tokens` vector.
    3. Else, add a token with `TokenType::TEXT` to the `tokens` vector.
2. etc. (map `w` to a `TokenType` and add a `Token` containing this type and `w` to the `tokens` vector.)

If a token string `w` cannot be mapped to a `TokenType`, throw an error (as an invalid token has been encountered).

4. Generate a `Line` object.

1. Allocate a `Line` object.
2. Map the `std::vector<Token> tokens` vector to a `LineType`.
3. Add the `Line` object to the `vector<Line> lines` vector.

4. Return the `std::vector<Line> lines` vector

After the `std::vector<Line> lines` vector has been generated, it is returned by the Parser to the `Assembler`.