Assembler Overview

The ByteFrost assembler is a complex piece of software that contains multiple pipeline stages.

Definitions

Set Definitions

These set definitions are used in the parsing stages of the assembler pipelines.

1. $ASCII$ is the set of ASCII characters.
2. A is the set of upper and lower case English characters.
3. D is the set of 10 numerical digits.
4. $TEXT$ is the set $(A \cup {_})~\circ~(A \cup {_} \cup D)^*$
    - A string s is in $TEXT$ iff the first character of s is an underscore or in A, and any following characters may be an underscore, in A, or in D.
5. ND is the set of unsigned decimal number strings.
    - $ND = {0} \cup (D \ {0})D^*$
6. NH is the set of unsigned hexadecimal number strings.
    - $NH = {0x} \circ D^+$
7. N is the set of unsigned number strings.
    - $N = ND \cup NH$
8. $NUMBER$ is the set of (hexadecimal and decimal) number strings.
    - $NUMBER = {\epsilon, -, +} \circ N$
9. $FILE$ is the set of file names.
    - $FILE = TEXT \circ {.} \circ TEXT$

Other Terms

`.asm` **file** - a ByteFrost assembly file; each line of such a file may contain at most one assembly statement.

Pipeline

When the ByteFrost assembler is executed, its input is a string from the command-line `a` which contains the name of the input `.asm` file followed by an array of optional command-line arguments.

The ByteFrost assembler then passes the command-line string `a` into the following pipeline:

0. **Command-Line Argument Parser**. Given the command-line arguments as specified by `int argc`, `char ** argv`, the Command-Line Argument Parser (CLAP) generates a `CommandLineArguments` object that contains the specified input files and any additional command-line argument flags / values. The CLAP will throw an error or warning for any command-line argument misuse. The `CommandLineArguments` object is saved as a field of the `Assembler`.

~~Given the command-line string `a`, the Command-Line String Parser (CLSP) gleans important information from `a`, i.e., the input `.asm` file and any other command-line arguments. It computes a `CommandLineArguments` object which contains the relevant input paramters from `a`, and throws an error or warning for any command-line argument misuse. The `CommandLineArguments` object is saved as a field of the `Assembler`.~~

1. **Parser**. Given the `CommandLineArguments` object in the `Assembler`, the Parser will attempt to open the input `.asm` file for reading. The parser then reads lines from the `.asm` file. Each line is a string `s`, which is parsed in the following way:

    1. Compute a string `s'` which is a prefix string of `s` until the character immediately before the first instance of the comment string `//`. If `s` contains no instance of the comment string, `s' = s`.
    2. Given the string `s'`, the Parser now splits `s'` into a vector of token strings `vector<string> tokenStrings` such that every token string is non-empty and contains no delimiter characters, defined by the Parser's set of delimiter characters (`std::unordered_set<char> delimiters`).
    3. Given the list of token strings `vector<string> tokenStrings`, the Parser computes a list of `Token`s `vector<Token> tokens` by mapping each token string `w` to a `TokenType t`. Each `TokenType` has its own regular expression; if a string `w` matches a `TokenType`'s regular expression, then it will be mapped to that `TokenType`. If a string `w` cannot be mapped to any `TokenType` the ByteFrost Assembler understands, then an error is thrown as an invalid token is encountered. Mapping a string `w` to a `TokenType` can fail even if `w` matches the `TokenType`'s regular expression. For instance, if `w` is mapped to a `PreProcessorDirective TokenType` since `w[0] = '.'` and `w[1:...]` can be mapped to a `Text TokenType`, if `w[1:...]` is not the name of any recognized preprocessor directive, the Parser will throw an error (e.g., `.notadirective` will cause this error, while `.define` will not). Once `w` is mapped to a `TokenType`, the Parser will create a `Token` object with this `TokenType` and contain any relevant information; note that `Token` is a base class so the actual `Token` object can be a derived class. e.g., `.define` -> `PreProcessorDirectiveToken(directive = DefineDirective)` **NOTE: need to change the syntax for this token depending on how preprocessor directive objects are stored / implemented**
    4. Given the list of `Token`s `vector<Token> tokens`, the Parser will compute a `Line` object which derives from the `Line` base class. To identify the derived class, the `vector<Token> tokens` list will be mapped to a `LineType` enum (just as token strings are mapped to a `TokenType` enum). Mapping `tokens` to `LineType` is done by looking at the length of `tokens` and the `TokenType`s of each token. E.g., if `tokens.size() == 0`, then the `LineType` is `EmptyLine`. if the first token has `TokenType PreProcessorDirective`, the parser will examine the particular directive (e.g., `.define`) to identify the expected number of tokens and their arguments (e.g., for `.define`, the expected `Token` sequence is `TEXT NUMBER NUMBER` (for constant name, size, and value)). If the number of tokens / types of token arguments don't match the expected ones for the `LineType` and its derived class, then an error will be thrown. The `Line` object is then added to a list of `Line`s in the `Assembler`, e.g. `vector<Line> lines`.

2. **Preprocessor**.

3. **Derived Instruction Conversion**.

4. **Code Generation**.

5. **Output File Generation**.

Class Structure

The entire assembler will be contained within the `Assembler` class, which will contain each pipeline stage as its own class and any "global" data will be similarly stored in the `Assembler` class itself.

I.e., the `Assembler` class looks like this:

```
class Assembler {
public:
    //  Constructor that reads in the command-line arguments.
    Assembler(int argc, char **argv);
private:
    //  Global data

    //  CommandLine Arguments
    CommandLineArguments args;

    //  Line vector
    std::vector<Line> lines;

    //  Pipeline Stage 0 - Command-Line String Parser (CLSP)
    CommandLineParser clsp;

    //  Pipeline Stage 1 - Parser
    Parser parser;

    //  ...
};
```