LAS: A scalable assembler

developed by Leo Liu(122090330)

Build & Use

```
#build
Proj> cmake -B build -G"Unix Makefiles"
Proj> cd build
Proj/build> make
```

```
#use
Proj/bin> ./las input.asm output.txt
```

Big picture thoughts and ideas

An assembler can be regarded as a very simple compiler.

Usually, a compiler is devided into frontend(tokenize, parse and semantic analyze), middle-end(optimize) and backend(code generate).

Since assembly is (almost) the same as machine code, the middle-end can be ignored and the backend is simple.

So the most important part of this assembler is the tokenizer and parser.

High level implementation ideas

The different phases should be indepent to each other.

For tokenizer, it should scan the asm file and produce a token stream.

For parser, it should consume the tokens from a token stream and produce a simple but informative structure (aka AST for high level language) for code generating.

Since different architectures and different instructions have different grammars, a domain-specific language should be designed to handle different structures and parse them.

Implementation details

The tokenizer has two parts: token and token stream.

A token is the simplest part in the language. In this project, the type of a token and the content can be discribed by the following code:

```
enum class tok_type
{
    ID, //For identfiers, including labels and instruction names
    REG, //For registers
    NUM, //For immediate numbers
    SYM, //For symbols
    STR, //For string literals
```

```
    CHAR, //For chars
    SEC, //For sections
    END //For EOF
};

struct token
{
    tok_type type;
    std::string content;
};
```

A token stream scans the bytes in an istream and devide the bytes into tokens.

```
class token_stream
{
    public:
    token_stream(istream&);

    struct iterator
    {
        iterator& operator++(); //Go to the next token
        const token& operator*(); //Get the current token
    };
};
```

Instead of exhausting the istream immediately, the token stream lazily get the next token everytime `iterator::operator++` is invoked.

C++ is a very complex language and have many features, especially about template.

Therefore, I designed a series of template classes to map the different terms in assembly.

Then for a specific asm language, we can customize the parser for different terms, and then use a language-free combinator to combine the parsers.

For example,

```
using iter=typename token_stream::iterator;

struct Data; //Store the result of parsing

struct rs
{
    Data parse(iter&);
};

template<typename Reg>
struct offset
{
    //...parse
};

//rd, rt, immediate, label, etc.

template<typename... Args>
struct args
{
    Data parse(iter&)
    {
        //Using template metaprogramming to generate the parser
        //that combines different arguments' parsers
    }
};
```

```
//The parser to parse the arguments of lw rt, immediate(rs)
using P=args<rt,offset<rs>>;
```

This can be extended for different asm languages such as x86 or arm, since C++ template itself is very scalable.

Though in this example the parser depends on the token_stream,

we can still decouple it by making token_stream a template argument.

After tokenizing and parsing, the code generator is very easy.

```
unsigned codegen(
    const MIPS::instr& instr,
    //The info about the instruction including opcode
    const MIPS::Data_field&,
    //The result of parsing, stores the different field such as registers, immediate numbers and labels
    unsigned pc
    //The address of the instruction, for relative-address jump like bne and so on.
    );
```