ECE 3574: Building Cross-Platform Software using CMake

Changwoo Min

Meeting 8: Building Cross-Platform Software using CMake

The goal of today's meeting it to learn about building larger software projects that have multiple modules of code, unit tests, and main programs.

- Why CMake?
- Running CMake: GUI and command-line
- Writing a basic CMakeLists.txt configuration file

Software Configuration and Build tools

- You should be able to build all dependencies and the code itself, in debug and release mode, for all platforms supported in a single step.
- This can be done by a variety of means, including customs scripts and IDE tooling. We will be using a popular open source tool for this called `cmake`.

Why CMake? What problem does it solve?

- Once a project gets to a certain size, compilation and linking, setting compiler flags, etc becomes complicated.
- This is especially true for cross-platform projects. It is a pain to maintain build configuration for each platform (VS .sln, XCode .xcodeproject, makefiles, …)
- CMake is a build generator, it writes the files needed for the specific IDE or build tool
- There are other tools that do this as well, e.g. `scons`.

Running CMake

- Using the GUI
- Using the command line

Basic CMakeLists.txt Syntax

```
cmake_minimum_required(VERSION 3.5)
project(YOURPROJECTNAME CXX)

add_executable(exename1 file1.h file2.cpp ... )
add_executable(exename2 file3.h file4.cpp ... )

enable_testing()
add_test(test_name exename arguments)
```

More advanced CMake

- CMake is a very flexible tool. Some examples
  - perform different configurations based on platform
  - write source files at configure time
  - run external scripts and programs for memory checking, coverage analysis, documentation generation, etc.

CMake Tutorial

- CMake Tutorial
- Running CMake

CMake Examples

- Exercise 05
- Milestone 0
- Milestone 1

Exercise 08: CMake

- See Website

Milestone 1: Parsing

- Milestone 1
- State-machine-based Parsing
- Recursive descent parser
- Recursive Descent Parsing in C/C++

Backus-Naur Form (BNF)

- A notation technique for context-free grammar

```
<instruction> ::= [<label>] <operation> EOL
<operation>   ::= 'nop' | <load_word> | <load_add> | ...
<load_word>   ::= 'lw'   <register> SEP <memref>
<load_add>    ::= 'la'   <register> SEP <memref>
<memref>      ::= <label> | <register> | [offset] '(' <register> ')'
```

```
     .data
x:    .word 100
arr: .byte 10,11,12

     .text
main:
     # load word from location x into temporary register 0
     lw $t0, x
     # load address of arr into $t1
     la $t1, arr
```

Lexing and Parsing

- Lexical analysis (lexing): raw text → token list
- Paring: token list → AST (abstract syntax tree)
  - `<declaration>` from data section grammar
  - `<operation>` from text section grammar
- **Q: how to systematically apply BNF rules to the given token list?**
  - Bottom-up parser: token list → rule
  - Top-down parser: rule → token list

State machine based paring

- See website

Recursive descent parsing

- See website

Next Actions and Reminders

- Read Qt documents
  - Qt for Beginners
  - Qt Examples And Tutorials
  - Overview of Qt
- Install Qt on your host system
  - Please do this before class.
- Work on Milestone 1. We will discuss on the design of parser class next week.