Syllabus for

“Compiling and Running of Program”

“程序编译与实现”教学大纲

Dr. Zheng Xiaojuan
Software College of Northeast Normal University.

Sep, 2019

The syllabus for the course “Compiler Construction Principle and Implementation Techniques” is given in this document, the contents include:

1. Prerequisite (前提课程)

- High-level Programming Languages (PASCAL, C)
- Assembly Language
- Data Structure
- Algorithm Design
- Computer Architecture

2. General Information

Credit (学分): 2

Hours(学时): 60

Class format: lectures + problem sets + exams + Projects

The lectures, problem sets and exams will be structured around compiler construction principle and implementation techniques. The instructor will introduce materials at class-time, some problems will be assigned for students as after-class assignments, and final examination will be taken at the end of the term. In addition, several projects will be suggested for practices.

Textbook:

Since bilingual teaching method will be used for this course, we are suggesting several optional textbooks for the course both in English and in Chinese.

[1] Aho, Alfred V., Ravi Sethi, and Jeffrey Ullman. Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesley, 1986. ISBN: 0201100886.

[2] Appel, Andrew W. Modern Compiler Implementation in Java. Cambridge, UK: Cambridge University Press, 1997. ISBN: 0521583888.

[3] Kenneth C. Louden. Compiler Construction: Principles and Practice. PWS Publishing Company, 1997. IBSN 0-534-3972-4.

[4] 金成植《编译程序构造原理与实现技术》.高等教育出版社.

3. Objectives

Compilers convert programs in "high level" languages to semantically equivalent machine code. Hence they constitute the bridge between hardware and software. Understanding how compilers work is essential for a deep understanding of the syntax of programming languages, efficiency and memory considerations of the available control structures and data types, issues in separate compilation, differences between programming languages, and the implications of processor architectures. The techniques of analysis and generation as used in compilers are useful to a much

broader class of problems like parsing command lines or search queries, analyzing or generating mark-up languages.

This course aims at introducing compiler construction principles and implementation techniques. Through studying the course students can more deeply understand basic concepts of programming languages, especially lexical, syntax and semantic conventions of high-level programming languages, get to know useful implementation techniques, form the ability to develop large-scale system software.

4. Main Contents

1. Introduction
   - 1.1 Introduction to the Course
     - 1.1.1 General Information
     - 1.1.2 Outline
     - 1.1.3 Course Goals
     - 1.1.4 How to Study the Course
   - 1.2 Introduction to Compiler
     - 1.2.1 Programming Languages
     - 1.2.2 Compiler and Interpreter
     - 1.2.3 Programs related to Compiler
     - 1.2.4 Design and Implementation of a Compiler
     - 1.2.5 Testing and Maintaining of a Compiler
   - 1.3 A Toy Compiler
     - 1.3.1 Functional Decomposition and Architecture of a Compiler
     - 1.3.2 General Working Process of a Compiler for a Toy Language

2. Scanning
   - 2.1 Overview
     - 2.1.1 General Function of a Scanner
     - 2.1.2 Some Issues about Scanning
   - 2.2 Finite Automata
     - 2.2.1 Definition and Implementation of DFA
     - 2.2.2 Non-Determinate Finite Automata
     - 2.2.3 Transforming NFA into DFA
     - 2.2.4 Minimizing DFA
   - 2.3 Regular Expressions
     - 2.3.1 Definition of Regular Expressions
     - 2.3.2 Regular Definition
     - 2.3.4 From Regular Expression to DFA
   - 2.4 Design and Implementation of a Scanner
     - 2.4.1 Developing a Scanner from DFA
     - 2.4.2 A Scanner Generator – Lex

3. Context-free Grammars and Parsing
   3.1 The Parsing Process
   3.2 Context-free Grammars
   3.3 Parse Trees and Abstract Syntax Tree
   3.4 Ambiguous
   3.5 Syntax of Sample Language

4. Top-down Parsing
   4.1 Overview of Top-down Parsing
   4.2 Three Important Sets
   4.2 Left Recursion Removal and Left Factoring (消除左递归和公共前缀)
   4.3 Recursive-Descent Parsing (basic method)
   4.4 LL(1) Parsing
      4.4.1 LL(1) Grammar
      4.4.2 LL(1) Parsing Table
      4.4.3 LL(1) Parsing Engine(驱动程序)
      4.4.4 LL(1) Parser Generator -- LLGen

5. Bottom-up Parsing
   5.1 Overview of Bottom-up Parsing
   5.2 Finite Automata for LR(0) Parsing
   5.3 LR(0) Parsing
   5.3 SLR(1) Parsing
   5.4 LR(1) Parsing
   5.5 LALR(1) Parsing
   5.5 Parser Generator

6. Semantic Analysis
   6.1 Overview of Semantic Analysis
   6.2 Symbol Table
   6.3 Semantic Analysis of Types
   6.4 Semantic Analysis of Declaration
   6.5 Semantic Analysis of Body
   6.6 Attribute Grammar and Action Grammar

7. Intermediate Code Generation
   7.1 Intermediate Representations
   7.2 Intermediate Code Generation of Expressions
   7.3 Intermediate Code Generation of Atomic Statements
   7.4 Intermediate Code Generation of Structural Statements

8. Intermediate Code Optimization
   8.1 Overview of Optimization Techniques
   8.2 Basic Block

8.3 Local Optimization of Constant Expressions
8.4 Local Optimization of Common Expressions
8.4 Loop Invariant Expressions
8.5 Global Optimizations

9. Runtime Environment
   9.1 Storage Organization and Allocation
   9.2 Activation Record and Stack
   9.3 Variable Access Environment (运行时变量访问环境)

10. Target Code Generation
    10.1 Overview of Target Code Generation
    10.2 Virtual Machine
    10.3 Temporary Varaibles
    10.4 Register Allocation
    10.5 Translation from Quadruples to Target Codes
    10.6 From AST to Target Codes

5. Grading

ACTIVITIES	PERCENTAGES
Final Examination	100%
Assignment	±5%
Attendance	±5%
Projects	Another course

6. Suggested Projects

Projects will be organized around designing and implementing a compiler for a sample language. There will be six segments, where each segment addresses a separate aspect of compiler construction.

Descriptions of the six parts of the compiler follow in the order that you will build them.

Scanner

This part scans the input stream (the program), and encodes it in a form suitable for the remainder of the compiler. You will need to decide exactly what you want the set of tokens to be and create the regular expressions for the scanner generator. The convention for this partitioning is quite standard in practice.

We'll supply a scanner generator. This will consist of a program that takes a specification of the set of token types and outputs a Java program, the scanner. This specification uses regular definitions to describe which lexical tokens, i.e., character sequences, are mapped to which token types. The resulting scanner

processes the input source code by interpreting the DFA (Deterministic Finite Automaton) that corresponds to the regular definition.

Parser

The parser checks the syntactic correctness of the token stream generated by the scanner, and creates an intermediate representation of the program that is used in code generation. You'll also need to build the symbol table, since you won't be able to build the code generator without it.
Both top-down parser and bottom-up parser should be developed.
Parser generator will be supplied. (Yacc & Bison)

Semantic Checker

This part checks that various non-context free constraints, e.g., type compatibility, are observed. We'll supply a complete list of the checks. It also builds a symbol table in which the type and location of each identifier is kept. The experience from past years suggests that many groups underestimate the time required to complete the static semantic checker, so you should pay special attention to this deadline.
It is important that you build the symbol table, since you won't be able to build the code generator without it. However, the completeness of the checking will not have a major impact on subsequent stages of the project. At the end of this project the front-end of your compiler is complete and you have designed the intermediate representation (IR) that will be used by the rest of the compiler.

Code Generator

In this assignment you will create a working compiler by generating unoptimized assembly code from the intermediate format you generated in the previous assignment.

Intermediate Code Generator

In this assignment you need to develop an intermediate code generator after semantic checking. It is suggested that quadruple be used.

Intermediate Code Optimizer

In this assignment you need to develop an intermediate optimizer for achieving both constant expressions and common expressions local optimization for intermediate code.

7. Calendar for 2019

Week	Day	Lectures	Assignments	Projects
1	2	Introduction to the Cours		
	5	Introduction to Compiler		
2	2			
	5			
3	2			
	5			
4	2			
	5			
5	2			
	5			
6	2			
	5			
7	2			
	5			
8	2			
	5			
9	2			
	5			
10	2			
	5			
11	2			
	5			
12	2			
	5			
13	2			
	5			
14	2			
	5			
15	2			
	5			
16	2			
	5			

8. Feedbacks