50.051 PLC

Compiler part – Outline

# Week 8

Lecture 1: Introduction to compilers and their components.

* Definition of compilers
* Purpose of compilers
* Types of compilers: source-to-source, cross, just-in-time (JIT)
* Overview of compiler architecture: front-end, middle-end, back-end
* Brief history of compilers
* Key concepts: syntax, semantics, tokens, parse tree, intermediate code, machine code, executable code

Lecture 1 (end) and 2: Overview of the compilation process and its stages.

* High-level overview of the compilation process
* Source code preprocessing
* Lexical Analysis (Tokenization) phase: recognizing tokens
* Syntax Analysis (Parsing) phase: building a parse tree
* Semantic Analysis phase: adding meaning to the parse tree
* Intermediate Code Generation phase: generating intermediate code
* Code Optimization phase: improving the intermediate code
* Machine Code Generation phase: generating machine code
* Assembly Code Generation phase: generating assembly code
* Object file generation
* Linking and loading

Lecture 3: A Reminder on Finite State Machines.

# Week 9

Lecture 1: Coding an FSM in C, everything you need to know.

Lecture 2: About Regular Expressions and how to run a regular expression analyser in C.

* Regular expressions definitions and notations
* Finite Automata and regular expressions equivalence

Lecture 3: Lexical Analysis/Tokenization phase: Understanding how the compiler scans and identifies tokens in the source code

* Overview of lexical analysis
* Scanning and tokenizing source code
* Lexical Analysis tools: Flex, Lex, ANTLR, etc.
* Hands-on coding exercises: implementing a simple lexical analyser using regular expressions

# Week 10

Lecture 1: Syntax Analysis/Parsing phase: Understanding how the compiler builds the parse tree from the tokens

* Overview of syntax analysis
* Context-Free Grammars
* BNF notations

Lecture 2: Syntax Analysis/Parsing phase: Understanding how the compiler builds the parse tree from the tokens – Part 2

* Parsing algorithms: Top-Down (LL) and Bottom-Up (LR)
* Parsing tables: First and Follow

Lecture 3: Syntax Analysis/Parsing phase: Understanding how the compiler builds the parse tree from the tokens – Part 3

* Hands-on coding exercises: implementing a simple LL parser

# Week 11

Lecture 1: Syntax Directed Translation, Understanding how to add semantics to the parse tree

* Overview of syntax directed translation
* Attributes, synthesized and inherited
* Abstract Syntax Trees (AST)

Lecture 2: Syntax Directed Translation, Understanding how to add semantics to the parse tree – Part 2

* Type checking and type coercion
* Code generation from AST
* Hands-on coding exercises: implementing a simple AST-based translator

Lecture 3: Intermediate Code Generation: Understanding how the compiler generates intermediate code from the parse tree

* Overview of intermediate code generation
* Three-address code representation
* Quadruples and triples
* Control flow graphs
* Translation of control structures
* (If time allows, hands-on coding exercises: generating three-address code from a simple program)

# Week 12

Lecture 1: Machine Code Generation: Understanding how the compiler generates machine code from the intermediate code

* Overview of machine code generation
* Target machine architecture
* Code generation for basic operations and expressions
* Register allocation and assignment
* Memory allocation and addressing modes
* (If time allows, hands-on coding exercises: generating machine code from intermediate code for a simple program)

Lecture 2: Assembly Code Generation: Understanding how the compiler generates assembly code from the machine code

* Overview of assembly code generation
* Assembly code syntax and directives
* Generating assembly code for basic operations and expressions
* Generating assembly code for control structures
* (If time allows, hands-on coding exercises: generating assembly code from machine code for a simple program)

Lecture 3: Linking and loading: Understanding how the compiler links the object files and generates the final executable

* Overview of linking and loading
* Object files and symbol tables
* Static and dynamic linking
* Library linking
* Executable file format

# Week 13

Lecture 1: A quick word on Debugging and profiling: Understanding how to debug and profile compiled code

* Overview of debugging and profiling
* Debugging tools: gdb, Visual Studio Debugger, etc.
* Breakpoints, watchpoints, and tracepoints
* Stack trace and call stack
* Profiling tools: gprof, perf, etc.

Lecture 2: Modern compilers, bootstrapping and compiler optimizations, and end

* Introduction to modern compiler design
* Compiler frameworks and tools: LLVM, GCC, Clang, etc.
* Intermediate Representations: LLVM IR, GCC IR, etc.
* Compiler optimizations: loop unrolling, constant propagation, etc.
* Code generation for modern architectures: SIMD, VLIW, etc.
* Conclusion and what to study next

Lecture 3: Optional, hands-on coding exercises to reinforce concepts covered in previous lectures

* In this lecture, students will work on hands-on coding exercises to reinforce the concepts covered in the previous lectures, to prepare for the exam.

Given that you've already covered the front-end, three-address code generation, and three-address code optimization, it's time to move on to the back-end of the compiler. In your final two-hour lecture, you can focus on the following topics:

1. Code Generation: Explain how the compiler translates the intermediate representation (IR), in this case, three-address code, into target machine code or assembly language. Discuss the challenges of generating efficient and optimized code for the target architecture.
2. Register Allocation: Describe the process of assigning variables and temporary values to registers in the target machine. Explain different register allocation strategies, such as graph coloring, linear scan, and priority-based allocation.
3. Instruction Scheduling: Discuss how compilers reorder instructions to maximize instruction-level parallelism and improve performance on pipelined and superscalar processors. Explain techniques such as list scheduling, trace scheduling, and software pipelining.
4. Peephole Optimization: Explain how compilers perform local optimizations by examining a small sequence of instructions and replacing them with a more efficient sequence, if possible. Provide examples of common peephole optimizations, such as redundant load/store elimination and strength reduction.
5. Assembler and Linker: Briefly describe the role of the assembler in translating assembly code into machine code and the linker in combining object files into a single executable. Explain how the compiler interacts with these tools to produce the final executable program.

To make the lecture engaging, consider incorporating examples and case studies to illustrate the concepts. Additionally, you can include comparisons between different optimization techniques and the trade-offs associated with each.