Lecture 2: Intro to Compilers and Program Analysis

I. HANDOUTS
   A. Handout1.txt

II. What is a compiler
   A. What do you think a compiler is?
   B. What is not a compiler
      1. Assembler
      2. Lisp Interpreter

III. Why study compilers?
   A. Compilers are very successful
      1. Model for other software systems
      2. Good combination of theory and practice
   B. Compiler are and will continue to be needed
      1. Essential for RISC processors
      2. Essential for new languages
      3. Essential for DSLs
   C. Compiler technology is used outside compilers
      1. Program analysis for software tools, understanding, security
      2. Database query languages
      3. Database (query) optimization and implementation
      4. Front ends for applications (command based)
      5. JIT compiling, portability, load-time compilation
      6. Processor emulation
      7. Other uses?

IV. Components of a compiler
   A. Pipeline approach
      1. Overall goal: translate from sequence of characters to runnable image
      2. Simplify compilation process by breaking apart translation
         a. Akin to divide and conquer but on algorithm rather than data
         b. Simpler translations are easier to understand, implement, debug, ...
      3. Show handout1.txt
4. Data flow diagram
   a. Source file => Lexical analysis => Token stream => Syntax Analysis =>
      Parse Trees => Syntax Processing => Abstract Syntax Trees =>
      Semantic Analysis => Annotated ASTs => Intermediate Code
      Generation => Intermediate Code => Optimization => Intermediate
      Code => Code Generation => Assembler Code => Assembler =>
      Machine Code
   b. Machine Code + Run Time Library => Runnable System

5. Importance of data representations

6. Is this how compilers really work?

B. Variations on a pipeline
   1. Narrow vs wide – how much to keep in memory
   2. Symbol table inside or outside
      a. What does this mean and entail
   3. More complex languages, e.g. C++
   4. Multiple semantic stages
   5. Multiple optimization stages
      a. Optimizations depend on each other
   6. Multiple code generation stages
      a. Register allocation
      b. Generating instructions
      c. Scheduling instructions
   7. Different types of intermediate code
      a. Simple code, basic blocks, flow graph
      b. Different levels of op codes
      c. Stack-based or operator based
   8. Complete front-back separation (language independence)
   9. One pass compilation
   10. JIT separation – byte codes and byte code compilers

C. Characteristics of a compiler
   1. Speed of the compiler
   2. Speed of the generated code
   3. Portability, retargetability
   4. Extensibility
   5. Run time environment

D. Theory behind a compiler
   1. Lexical analysis :: regular grammars (DFSA, NFSA)
2. Syntax analysis :: context free grammars
3. Semantic analysis :: attribute grammars, type theory, formal semantics
4. Optimization :: fast graph algorithms, set algorithms, SSA form
5. Code generation :: pattern matching, dynamic programming
6. Run time :: garbage collection, synchronization

E. Tools for building compilers
1. Grammar-based tools for lexical and syntactic analysis
2. Attribute grammars and other semantic tools, visitor patterns
3. Tree-pattern based code generators
4. Code generators from ISP

V. Program Analysis
A. Program analysis often uses the same information as a compiler
   1. Starting points
      a. Abstract syntax tree (for source analysis)
      b. Annotated abstract syntax tree (with bindings)
      c. Program dependence graph (close to intermediate code)
      d. JIT code
      e. Binary code (from loader image)
      f. Binary code (from memory)
   2. What is computed
      a. Types, data flow, information flow, other properties
         i. Many of these are the same as needed for compiling
         ii. Or needed for optimization
      b. Escape analysis – needed to understand whether something can be
         optimized or not (punt and let the programmer declare volatile)
      c. Flow analysis – akin to liveness analysis in optimization
      d. Type analysis – akin to expression analysis
         i. Even typed languages aren’t typed inside expressions
         ii. Most languages have overloaded operators
         iii. Need to model the type at each expression point
   B. Program analysis uses the same algorithms as a compiler
      1. To get to proper starting point (e.g. parsing)
      2. To compute the information
         a. Flow-based set algorithms over program graphs
         b. Implementing these efficiently is the same in both
   C. Sometimes program analysis is akin to compilation
      1. Nobbles needs to link uses and definitions
2. Easier to do by doing a complete flow analysis of the system than by traditional expression and type analysis

VI. Assignment
A. Read chapter 2, understand FSAs and regular expressions
   1. Exercises 2.1, 2.4c, 2.12b
B. Have a short presentation of your project idea ready