EECS 483 – Compiler Construction

Winter 2008, University of Michigan
Trevor Mudge

January 7th, 2008

GSIs and Office Hours

- Contacting Trev
  - tnm@umich.edu
  - Office – 4757 CSE
  - Office hours – tbd
    * best to make an appointment

- GSI – tbd!
Class Overview

- The goal is for you to understand the major parts of a modern compiler (but simplified, no gcc!)
  - Focus on the concepts in class
  - Put concepts to work in projects (learn by doing)
- Emphasis - 1/2 frontend, 1/2 backend
- Caveats
  - Underlying infrastructure for projects is being redone this semester – again 😇
  - The class will be a bit disorganized, so hang in there

Background You Should Have

- 1. Programming (essential)
  - Good C/C++ programmer (essential)
  - Linux, gcc, gdb, emacs, stl
  - Compiler system not ported to Windows or Mac
- 2. Basics of computer organization (helpful)
  - EECS 370
  - RISC architecture, pipelining, registers, assembly code
- 3. Basic theory of computation (helpful)
  - EECS 376
  - Finite automata, regular expressions, context-free grammars
Textbook and Other Classroom Material

- Class textbook

- Other useful books
  - *Lex & Yacc*, Levine, Mason and Brown
  - *Advanced Compiler Design & Implementation*, Steven Muchnick
  - *Building an Optimizing Compiler*, Robert Morgan
  - *Modern Compiler Implementation in Java/C/etc*, Andrew Appel

- Course webpage + course newsgroup
  - Will be set up next week

What the Class Will be Like?

- Class meeting time – 1:30 – 3:00 MW
  - LONG – *Bring caffeine!!*
  - No regular Friday meeting, but keep it open for make ups

- Graduate class model
  - No preset grading curve
  - No memorizing facts, algorithms, formulas, etc.
  - But, you will have to be independent in this class and figure some things out yourself
    - RTFM
    - Ask when you get stuck on something
    - But, staring at some code for 2 hours should not seem unreasonable
What the Class Will be Like (2)

- Learning compilers
  - Learn by doing – writing code
  - Substantial amount of programming
  - Substantial amount of debugging
  - Reasonable amount of reading

- Classroom
  - Attendance – you should be here
  - Print out lecture notes beforehand – notes incomplete
  - Examples to work out in class

Course Grading

- Yes, everyone will get a grade ...
  - No preset distribution of grades, scale, etc.
  - Most (hopefully all) will get A’s and B’s
  - Projects are essential part of class

- Components – subject to change
  - Exams (2) – 40% (25% each)
  - Projects (3) – 55% (break down – tbd)
  - Homeworks – Practice problems only
  - Class participation and involvement – 5%
Exams

- 2 exams in class
- Format
  » Open book/notes – but that doesn’t mean studying is unnecessary
  » 1.5 hrs
  » Lecture material, projects
- No final exam

Homeworks

- A couple of these given out during the semester
  » Not sure how many yet
  » Optional – not to be turned in
- Goals
  » Learn the important concepts
  » Practice questions for the exams
Projects

- Build most of the important parts of a simple compiler divided into 3 parts
  1. Preprocessing, parsing, syntax check
  2. Building syntax trees/symbol table
  3. Creating assembly code

Notes
- 1-3: Spec of what you need to do
- All done individually

Project Grading

- Different from what you may be used to
  - You will set up an appointment
  - You will give a 5-10 min demo of your code and explain the implementation
  - Show program running on given inputs, run on unseen inputs
  - You will be expected to understand the implementation and be able to answer questions about it
Class Participation

- Interaction and discussion is important, but hard even with a medium size class
  - Be here and don’t just stare at the wall
  - Be prepared to discuss the material
- Opportunities for participation
  - Solving class problems
  - Posting insights on the newsgroup
  - Feedback to me about the class
- Not a large part of your grade
  - But it will have an affect, particularly in the borderline cases

Why Compilers?

- Compiler
  - A program that translates from 1 language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language
- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly – still used in embedded and DSPs
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages
Compiler Structure

- Source language
  - Fortran, Pascal, C, C++
  - Verilog, VHDL, Tex, Html
- Target language
  - Machine code, assembly
  - High-level languages
- Compile time vs run time
  - Compile time or statically – Positioning of variables
  - Run time or dynamically – SP values, heap allocation

General Structure of a Modern Compiler

- Source Program
  - Lexical Analysis
    - Scanner
    - Syntax Analysis
      - Parser
      - Build high-level IR
  - Semantic Analysis
    - Controlflow/Dataflow
  - Optimization
  - Code Generation
- Context
  - Symbol Table
  - CFG
- Assembly Code
- Front end
  - High-level IR to low-level IR conversion
- Back end
  - Machine independent asm to machine dependent
Lexical Analysis (Scanner)

-Extracts and identifies lowest level lexical elements from a source stream
  - Reserved words: for, if, switch
  - Identifiers: “i”, “j”, “table”
  - Constants: 3.14159, 17, “%d\n”
  - Punctuation symbols: “(, ), “, “,”, “+”

- Removes non-grammatical elements from the stream – ie spaces, comments

- Implemented with a Finite State Automata (FSA)
  - Set of states – partial inputs
  - Transition functions to move between states

Lex/Flex

- Automatic generation of scanners
  - Hand-coded ones are faster
  - But tedious to write, and error prone!

- Lex/Flex
  - Given a specification of regular expressions
  - Generate a table driven FSA
  - Output is a C program that you compile to produce your scanner
Parser

- Check input stream for syntactic correctness
  - Framework for subsequent semantic processing
  - Implemented as a push down automaton (PDA)
- Lots of variations
  - Hand coded, recursive descent?
  - Table driven (top-down or bottom-up)
  - For any non-trivial language, writing a correct parser is a challenge
- Yacc (yet another compiler compiler)/bison
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Static Semantic Analysis

- Several distinct actions to perform
  - Check definition of identifiers, ascertain that the usage is correct
  - Disambiguate overloaded operators
  - Translate from source to IR (intermediate representation)
- Standard formalism used to define the application of semantic rules is the Attribute Grammar (AG)
  - Graph that provides for the migration of information around the parse tree
  - Functions to apply to each node in the tree
Backend

- Frontend –
  - Statements, loops, etc
  - These broken down into multiple assembly statements

- Machine independent assembly code
  - 3-address code, RTL
  - Infinite virtual registers, infinite resources
  - “Standard” opcode repertoire
    - load/store architecture

- Goals
  - Optimize code quality
  - Map application to real hardware

Dataflow and Control Flow Analysis

- Provide the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal

- Dataflow analysis
  - Identify when variables contain “interesting” values
  - Which instructions created values or consume values
  - DEF, USE, GEN, KILL

- Control flow analysis
  - Execution behavior caused by control statements
  - If’s, for/while loops, goto’s
  - Control flow graph
Optimization

- How to make the code go faster
- Classical optimizations
  - Dead code elimination – remove useless code
  - Common subexpression elimination – recomputing the same thing multiple times
- Machine independent (classical)
  - Focus of this class
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

Code Generation

- Mapping machine independent assembly code to the target architecture
- Virtual to physical binding
  - Instruction selection – best machine opcodes to implement generic opcodes
  - Register allocation – infinite virtual registers to N physical registers
  - Scheduling – binding to resources (ie adder1)
  - Assembly emission
- Machine assembly is our output, assembler, linker take over to create binary
Why are Compilers Important?

- **Computer architecture**
  - Build processors that software can be automatically mapped to efficiently
  - Exploiting hardware features

- **CAD tools**
  - Behavioral synthesis / C-to-gates tools are hardware compilers
  - Use program analysis/optimization to generate cheaper hardware

- **Software developers**
  - How do I create a compiler?
  - How does it map my code to the hardware