Compiler Construction

EECS 483 – Winter 2011

Lecture 1

Satish Narayanasamy

Today ..

- Administrative information
- Course structure
- Course content overview

Administrivia

- CTools – will contain most of the course information
  - http://ctools.eecs.umich.edu/
    - Post your questions in the forum here. GSI and I will try to reply to your questions, but you can also reply to others questions.
- Website -- outline of the course
  - http://www.eecs.umich.edu/courses/eecs483/
- Instructor: Satish Narayanasamy
  - nsatish@umich.edu (mention EECS 483 in the subject)
  - Office: 4721 CSE
  - Lecture: MW: 1:30 – 3:00pm (2166 DOW)
  - Office hours: MW: 3:00 – 4:00pm (right after class)
- GSI: Ashutosh Parkhi
  - aparkhi@umich.edu (put EECS 483 in the subject)
  - Discussion: Fri: 1:30 – 2:30pm (2166 DOW)
  - Office hrs – Tue/Thu 3:00pm – 4:00pm (Room: CSE Learning Center)
  - Primary contact for project/coding questions

Textbook

- Class textbook
  - *Compilers: Principles, Techniques, and Tools*, Aho, Sethi, Ullman (aka dragon book), 2nd edition

- Other useful books, but not required
  - *Lex & Yacc*, Levine, Mason and Brown
  - ***Engineering a Compiler* by Keith Cooper, Linda Torczon**

Background you should have

- Programming (essential)
  - Good C/C++ programmer (essential)
  - Linux, gcc, gdb, emacs/vi, stl
- Basics of computer organization (helpful)
  - EECS 370
  - RISC architecture, registers, assembly code
- Basics of theory of computation (helpful)
  - EECS 376
  - Finite automata, regular expressions, context-free grammars

Course Structure

- Components
  - Projects (5) – 50%
  - Exams (2) – 40% (20% each)
  - Homeworks – 5%
  - Class and forum participation – 5%
- Grading
  - No preset distribution of grades, scale, etc.
  - Most (hopefully all) will get As and Bs
  - Projects are extremely important

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 forum
  - Feedback to me about the class
- Not a large part of your grade
  - But it will have an affect, particularly in the borderline cases

Homeworks

- About 5-6 homeworks
  - Each homework accounts for only 1% of your grade
  - You can discuss, but should be written individually
  - Due at the start of the class, unless otherwise noted
- Goals
  - Learn important concepts and be able to keep up to speed with the lectures
  - Practice questions for the exams

Exams

- 2 exams, tentative schedule:
  - Midterm Exam - 20% of your grade
    - Feb 23rd, 1:30-3:00p (DOW 2166)
  - Final Exam - 20% of your grade
    - April 21st, 4-6p (Room: TBD)

- Format
  - Open book/notes – but that doesn't mean studying is unnecessary
  - Tested on lecture materials and projects

Projects

- Building a compiler for the Decaf language (simpler form of C++/Java)
- Divided into 5 parts
  - Scanner (understand words)
  - Parser (make sentences)
  - Semantic Checker (check grammar)
  - Generate assembly code
  - Optimize assembly code for performance (open ended)
    - We may have a competition between groups
- Final project report describing your optimizations
- Groups of 3 (2 is also ok)
  - First project will be assigned next Mon, 10th Jan

Tasks

- Pick your partners ASAP
  - Choose wisely
  - Use CTools forum to find partners
- Select a weekly meeting time for your team to meet
- Team members along with your meeting time is due 12th Jan
  - Mail Ashutosh

Project Administration

- Project startup files will be give out through ctools
- Project submission will also be through automated submission system
- Grading is partly automated
  - Grade will be based on the number of test cases that your code handles correctly
  - Don’t cheat! We use plagiarism detection software.
- Don’t slack. Projects are very time consuming. Start early!
  - Each group has **three late days**
  - If you have exhausted all three, 25% penalty will be imposed for each additional calendar day.
- Caveats
  - Underlying infrastructure for projects continues to be revised

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

What the Class Will be Like?

- 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
    - 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
  - Lots of examples to work out in class

Overview

Role of Compilers

- In the beginning, there was machine language
  - Ugly – writing code, debugging

- Then came textual assembly
  - Better, but still inadequate for building large software
  - Still used in embedded system programming (DSPs)

Role of Compilers

- Today: Program in high-level languages (Fortran, C, Java), and automatically translate it to assembly
  - John Backus
  - An assembler translates assembly to machine language

- Tomorrow: Program by sketching?

What is a Compiler

- Compiler
  - A program that translates from one language to another
  - It must *preserve semantics* of the source
  - It should create an *efficient* version of the target language

Compiler Vs Interpreter

- Batch compilation systems dominate
  - gcc
- Some languages are primarily interpreted
  - javascript
- Some environments (e.g. Lisp) provide a choice
  - Interpreter for development
  - Compiler for production

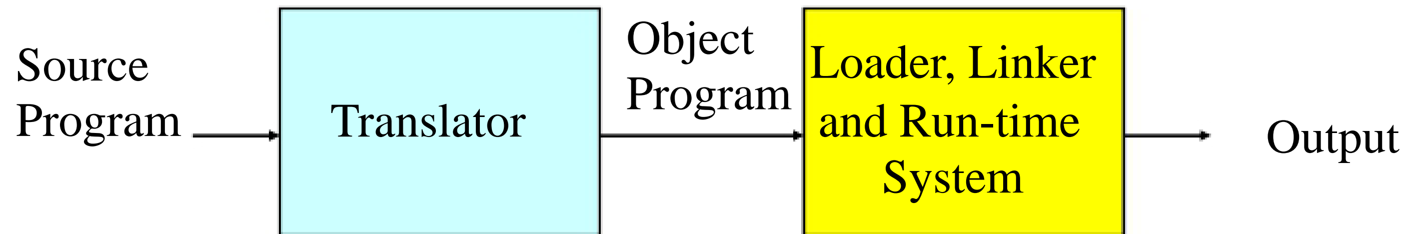
Why are Compilers Important?

- 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 does a compiler map my code to the hardware?

Areas where you can apply techniques that you learn in 483

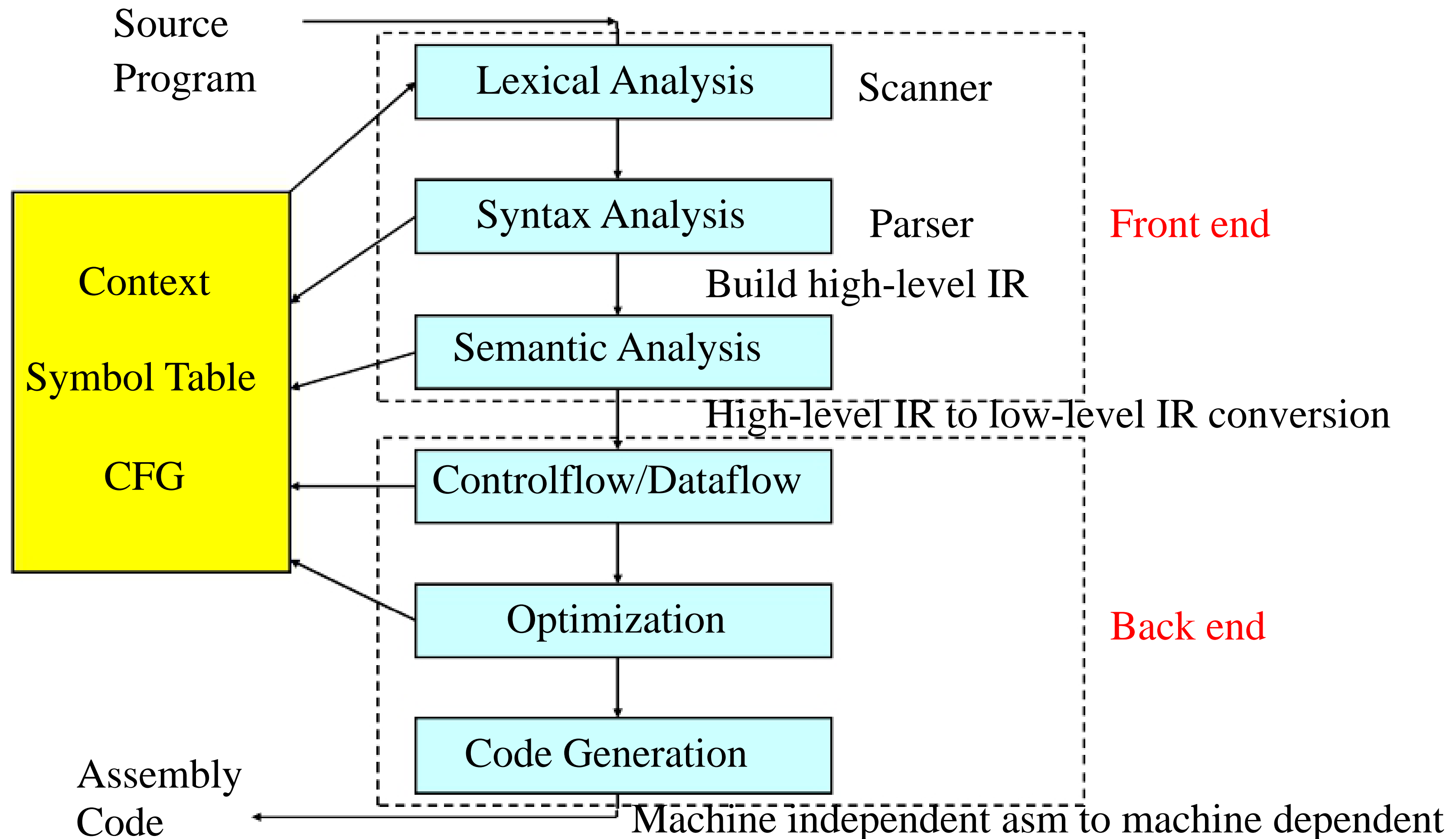
- Web 2.0 applications (javascript)
- Dynamic compilation (Java Virtual Machine)
- Web search (parsing)
- Hardware description languages (e.g. Verilog)
- Automatic debuggers (detect memory leaks)
- Security (Bounds checking)
- Automatic parallel compilers
- Compilers for domain specific languages
- Binary translation (x86 to some internal RISC)

Compiler Structure



- Source language
  - Fortran, Pascal, C, C++
  - Verilog, VHDL, Tex, Html
- Target language
  - Machine code, assembly
  - High-level languages, simply actions
- Compile time vs run time
  - Compile time or static – Positioning of variables
  - Run time or dynamic –heap allocation

General Structure of a Modern Compiler



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, generates your scanner

Parser

- Similar to understanding a sentence structure in English
- Check input stream for syntactic correctness
  - Framework for subsequent semantic processing
- Yacc (yet another compiler compiler)/bison
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Semantic Analysis

- Similar to understanding the meaning of a sentence, and identifying grammatical errors

- 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)

Error Checking

- Tom said Tom came late to class

```
{
        int Tom = 100;
        {
                int Tom = 1000;
                cout << Tom;
        }
}
```

Code Generation

- Translate IR to the assembly language of 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

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

– Strength reduction

- Most of the compiler research today is centered around error checking, and runtime management/optimization