

EECS 483 – Compiler Construction

Fall 2006, University of Michigan
Scott Mahlke (mall-key)

September 6, 2006

GSI and Office Hours

❖ Contacting Scott

- » mahlke@umich.edu
- » Office – 4633 CSE
- » Office hours – Right after class 12:30 – 1:00pm
 - Drop by – I will talk to you if I am free
 - Or make appointment

❖ GSI – Simon Chen

- » chenxu@umich.edu
- » Office hrs – Tue/Thu 1:30 – 3:30pm
- » Office hrs will be held in the GSI room - 4th floor CSE
- » **Primary contact for project/coding questions**

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
 - » 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
 - » *Compilers: Principles, Techniques, and Tools*, Aho, Sethi, Ullman (aka red dragon book)
- ❖ 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*, Andrew Appel
- ❖ Course webpage + course newsgroup
 - » Will be set up next week
 - » **Newsgroup on phorum** – <http://phorum.eecs.umich.edu/>
 - You will need an eecs account
 - GSI will regularly check this and answer questions
 - Read regularly, read before asking new questions

What the Class Will be Like?

- ❖ Class meeting time – 10:40 – 12:30, MW
 - » LONG – **Bring caffeine!!**
 - » We'll take a short break around 11:30
- ❖ 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
- » Lots of 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
 - » Exams (2) – 40% (20% each)
 - » Projects (4) – 55% (10%, 12.5%, 12.5%, 20%)
 - » Homeworks – Practice problems only
 - » Class participation and involvement – 5%

Exams

- ❖ 2 exams, tentatively scheduled for:
 - » Wednes, Oct 25, in class
 - » Monday, Dec 11, in class
- ❖ Format
 - » Open book/notes – but that doesn't mean studying is unnecessary
 - » 2 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 4 parts
 1. Preprocessing, parsing, syntax check
 2. Building syntax trees/symbol table
 3. Creating assembly code
 4. Analysis and optimization of assembly code
- ❖ Notes
 - » P1-3: Spec of what you need to do
 - » P4: You have to define and decide what to do
 - Goal is to create the fastest code possible
 - » P1 done individually
 - » P2-4: groups of 2 people

Project Grading

- ❖ Different from what you may be used to
 - » Each group will set up an appointment
 - » They will give a 5-10 min demo of their code and explain their implementation
 - » Show program running on given inputs, run on unseen inputs
 - » Each group member will be expected to understand the implementation and be able to answer questions about it
 - » **Slackers beware!**

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

Course Schedule (subject to change)

Week	Subject	Reading	Projects	Exams
0: 9/6	Introduction	Ch 1		
1: 9/11, 9/13	Lexical analysis	Ch 2-3	P1 out, 9/13	
2: 9/18, 9/20	Syntax analysis	Ch 4-5		
3: 9/25, 9/27	Syntax analysis	Ch 4-5	P1 due, 9/27	
4: 10/2, 10/4	Syntax, Semantic analysis	Ch 4-5 Ch 6	P2 out, 10/2	
5: 10/9, 10/11	Semantic analysis	Ch 6		
6: 10/16, 10/18	Semantic analysis	Ch 6	P2 due, 10/18	
7: 10/23, 10/25	Exam review, Exam 1			E1, 10/25
8: 10/30, 11/1	Intermediate code	Ch 7-8	P3 out, 10/30	
9: 11/6, 11/8	Control flow	Ch 9-10		
10: 11/13, 11/15	Dataflow and Opti	Ch 9-10	P3 due, 11/15	
11: 11/20, 11/22	Dataflow and Opti	Ch 9-10	P4 out, 11/20	
12: 11/27, 11/29	Code generation	Ch 9-10		
13: 12/4, 12/6	Code gen, Exam review	Ch 9-10		
14: 12/11, 12/13	Exam2, Advanced topics			E2, 12/11
15: 12/18	P4 demos		P4 due, 12/18	

Why Compilers?

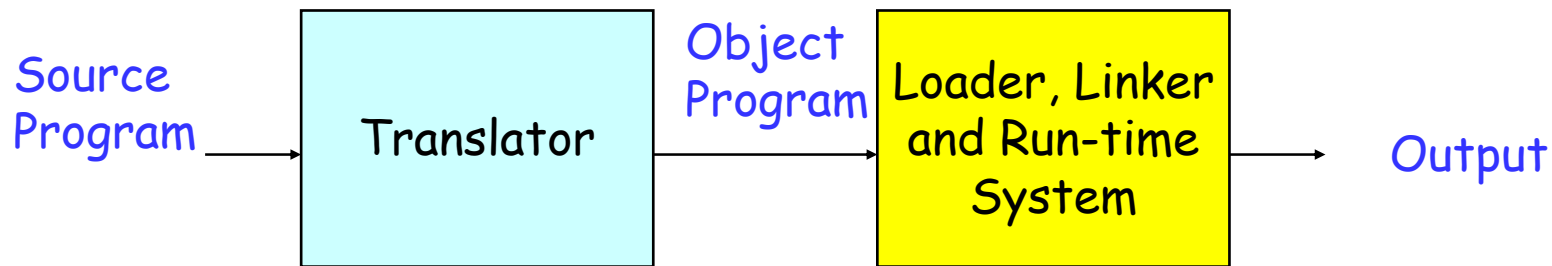
❖ Compiler

- » 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

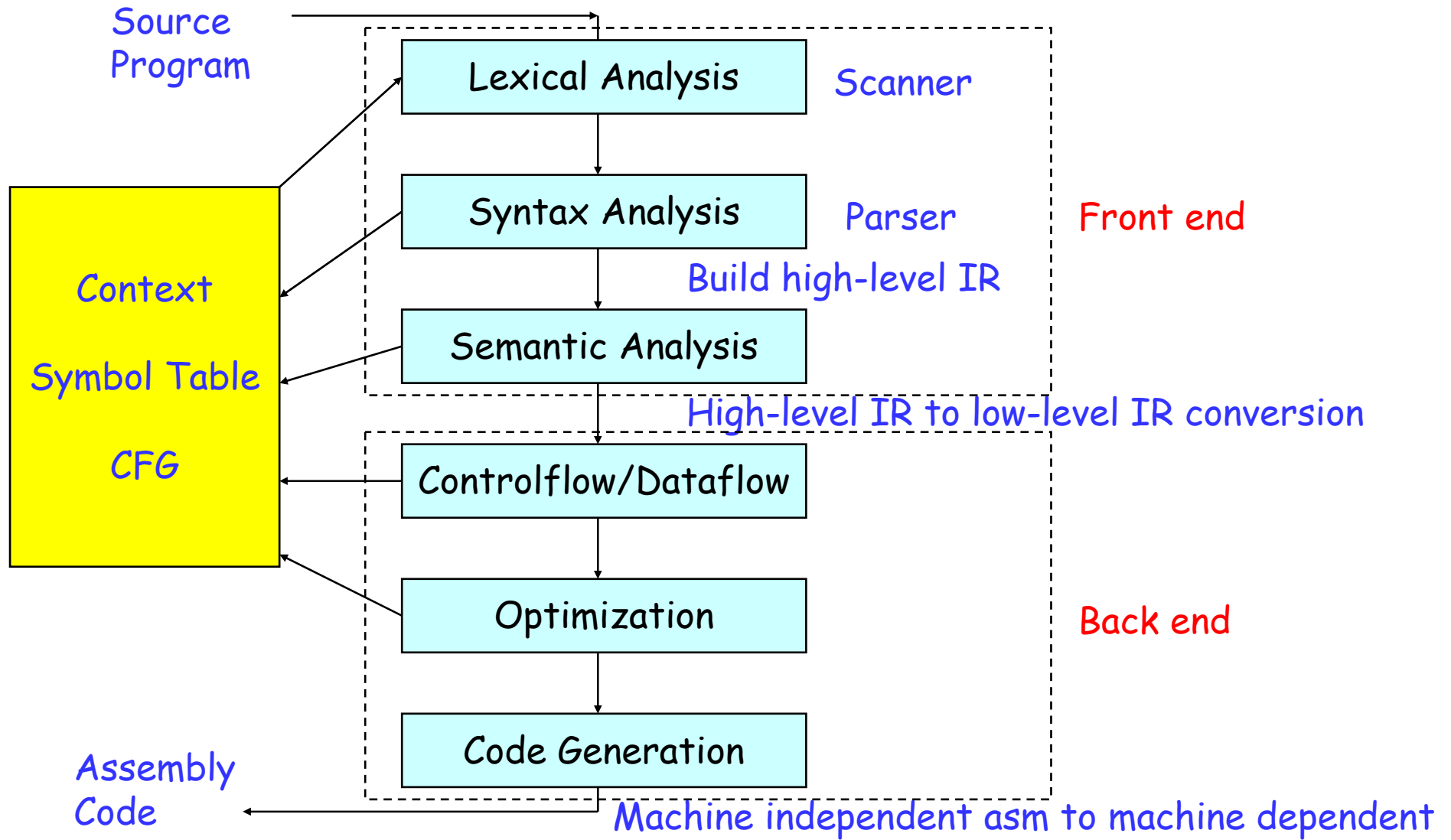
- » Ugly – writing code, debugging
- » Then came textual assembly – still used on DSPs
- » High-level languages – Fortran, Pascal, C, C++
- » Machine structures became too complex and software management too difficult to continue with low-level languages

Compiler Structure



- ❖ Source language
 - » Fortran, Pascal, C, C++
 - » Verilog, VHDL, Tex, Html
- ❖ Target language
 - » Machine code, assembly
 - » High-level languages, simply actions
- ❖ 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



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