Lecture 1

Introduction

I. Why Study Compilers?

II. Course Syllabus

Chapters 1.1-1.5, 8.4, 8.5, 9.1

I. Why Study Compilers?

Reasons for Studying Compilers

- **Compilers are important**
  - An essential programming tool
    - Improves software productivity by hiding low-level details
  - A tool for designing and evaluating computer architectures
    - Inspired RISC, VLIW machines
    - Machines' performance measured on compiled code
  - Techniques for developing other programming tools
    - Examples: error detection tools
  - Little languages and program translations can be used to solve other problems
- **Compilers have impact: affect all programs**

Compiler Study Trains Good Developers

Excellent software engineering case study

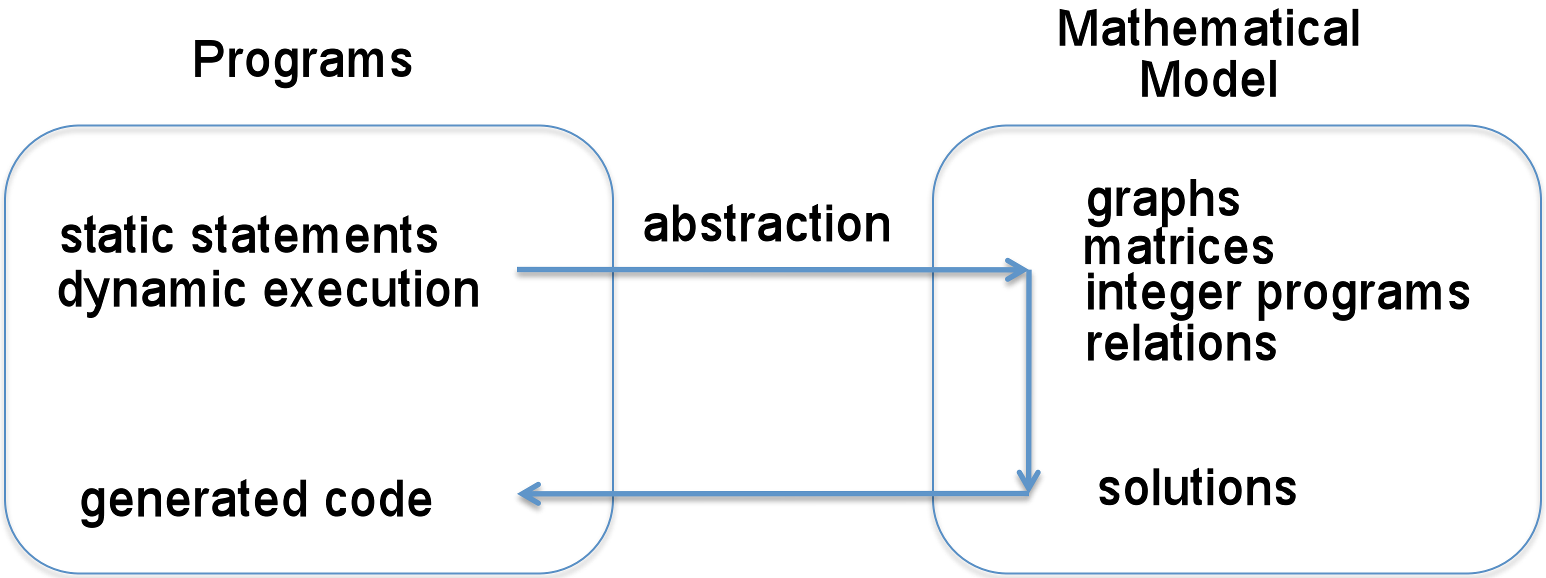
- **Optimizing compilers are hard to build**
  - Input: all programs
  - Objectives:
- **Methodology for solving complex real-life problem s**
  - Key to success: Formulate the right approximation!
    - Desired solutions are often NP-complete / undecidable
  - Where theory meets practice
    - Can't be solved by just pure hacking
      - theory aids generality and correctness
    - Can't be solved by just theory
      - experimentation validates and provides feedback to problem formulation
- **Reasoning about programs, reliability & security makes you a better programmer**

There are programmers, and there are tool builders ...

Example

- **Tools for web application security vulnerabilities**
- **PQL: a general language for describing information flow of interest**
- **Static techniques to locate errors automatically**
- **Illustrates:**
  - Exciting research area!
  - Importance of programming tools
  - Sophistication of static analysis techniques
  - What static analysis looks like
  - Use of little languages
  - Combination of theory and hacking

Use of Mathematical Abstraction



- **Design of mathematical model & algorithm**
  - Generality, power, simplicity and efficiency

Course Syllabus

1. Basic compiler optimizations

Goal	Eliminates redundancy in high-level language programs Allocates registers Schedules instructions (for instruction-level parallelism)
Scope	Simple scalar variables, intraprocedural, flow-sensitive
Theory	Data-flow analysis (graphs & solving fix-point equations)

2. Pointer alias analysis

Goal	Used in program understanding, concrete type inference in OO programs (resolve target of method invocation, inline, and optimize)
Scope	Pointers, interprocedural, flow-insensitive
Theory	Relations, Binary decision diagrams (BDD)

3. Parallelization and memory hierarchy optimization

Goal	Parallelizes sequential programs (for multiprocessors) Optimizes for the memory hierarchy
Scope	Arrays, loops
Theory	Linear algebra

4. Garbage collection (run-time system)

Tentative Course Schedule

1	Course introduction	
2	Basic compiler	Data-flow analysis: introduction
3		Data-flow analysis: theoretic foundation
4		(joeq)
5		Optimization: constant propagation
6		Optimization: redundancy elimination
7		Register allocation
8		Scheduling: non-numerical code
9		Scheduling: software pipelining
10		Dynamic compilation
11	Pointer alias analysis	Formulation
12		BDDs in pointer analysis
13	Parallelism/Locality	Introduction
14		Affine partitioning
15	Garbage Collection	Basic concepts
16		Optimizations

Course Emphasis

- **Methodology: apply the methodology to other real life problems**
  - Problem statement
    - Which problem to solve?
  - Theory and Algorithm
    - Theoretical frameworks
    - Algorithms
  - Experimentation: Hands-on experience
- **Compiler knowledge:**
  - Non-goal: how to build a complete optimizing compiler
  - Important algorithms
  - Exposure to new ideas
  - Background to learn existing techniques

Assignment by next class (no need to hand in)

- **Think about how to build a compiler that converts the code on page 11 to page 12**
  - (Read Chapter 9.1 for introduction of the optimizations)
- **Example:**
  Bubblesort program that sorts array *A* allocated in static storage

```
for (i = n-2; i >= 0; i--) {
   for (j = 0; j <= i; j++) {
      if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
      }
   }
}
```

Code Generated by the Front End

```
     i := n-2
S5:  if i<0 goto s1
     j := 0
s4:  if j>i goto s2
     t1 = 4*j
     t2 = &A
     t3 = t2+t1
     t4 = *t3        ;A[j]
     t5 = j+1
     t6 = 4*t5
     t7 = &A
     t8 = t7+t6
     t9 = *t8         ;A[j+1]
     if t4 <= t9 goto s3
     t10 = 4*j
     t11 = &A
     t12 = t11+t10
     temp = *t12   ;temp=A[j]
```

```
     t13 = j+1
     t14 = 4*t13
     t15 = &A
     t16 = t15+t14
     t17 = *t16        ;A[j+1]
     t18 = 4*j
     t19 = &A
     t20 = t19+t18   ;&A[j]
    *t20 = t17         ;A[j]=A[j+1]
     t21 = j+1
     t22 = 4*t21
     t23 = &A
     t24 = t23+t22
    *t24 = temp        ;A[j+1]=temp
s3:  j = j+1
     goto S4
S2:  i = i-1
     goto s5
s1:
```

*(t4=*t3 means read memory at address in t3 and write to t4:*
**t20=t17: store value of t17 into memory at address in t20)*

After Optimization

Result of applying:

- global common subexpression
- loop invariant code motion
- induction variable elimination
- dead-code elimination

to all the scalar and temp. variables

These traditional optimizations can make a big difference!

```
     i = n-2
     t27 = 4*i
     t28 = &A
     t29 = t27+t28
     t30 = t28+4
S5:  if t29 < t28 goto s1
     t25 = t28
     t26 = t30
s4:  if t25 > t29 goto s2
     t4 = *t25              ;A[j]
     t9 = *t26            ;A[j+1]
     if t4 <= t9 goto s3
     temp = *t25            ;temp=A[j]
     t17 = *t26             ;A[j+1]
     *t25 = t17             ;A[j]=A[j+1]
     *t26 = temp            ;A[j+1]=temp
s3:  t25 = t25+4
     t26 = t26+4
     goto S4
S2:  t29 = t29-4
     goto s5
s1:
```