

6.035

Fall 2006

Lecture 1: Introduction

- Intro. to Computer Language Engineering
- Course Administration info.

Staff

- Lecturer
 - Prof. Saman Amarasinghe saman@mit.edu 32-G778 253-8879
- Course Secretary
 - Mary McDavid mmcdavit@csail.mit.edu 32-G764 253-9620
- Teaching Assistant
 - Michael Gordon mgordon@mit.edu

Reference Textbooks

- Modern Compiler Implementation in Java
by A.W. Appel
Cambridge University Press, 1998
(available online from MIT Libraries)
- Advanced Compiler Design and Implementation
by Steven Muchnick
Morgan Kaufman Publishers, 1997
- Compilers -- Principles, Techniques and Tools
by Aho, Sethi and Ullman
Addison-Wesley, 1988
- Engineering a Compiler
by Keith Cooper and Linda Torczon
Morgan Kaufmann, 2004
- Optimizing Compilers for Modern Architectures
by Randy Allen and Ken Kennedy
Morgan Kaufmann, 2002
(available online from MIT Libraries)

The Project: The Five Segments

- Lexical and Syntax Analysis
- Semantic Analysis
- Code Generation
- Data-flow Analysis
- Optimizations

Each Segment...

- Segment Start
 - Project Description
- Lectures
 - 2 to 5 lectures
- Project Time
- (Project Checkpoint & Design Document)
- Project Time
- Project Due

Project Groups

- Each project group consists of 3 to 4 students
- Grading
 - All group members (mostly) get the same grade

Weekly Group Meeting with the TA

- TA will schedule a weekly 30 minute slot
- The group will get to:
 - Ask questions about the project
 - Review feedback on design documents
 - Discuss design decisions
 - Describe what each person is doing
 - Answer questions the TA has about the project
- TA will use this to measure individual contribution to the project → be there!

Grades

- Compiler project 70%
 - Scanner/Parser 5%
 - Semantic Checking 10%
 - Code Generation 15%
 - Data-flow Analysis 10%
 - Optimizations 30%
- In-class Quizzes 30% (10% each)

Optimization Segment

- Making programs run fast
 - We provide a test set of applications
 - Figure-out what will make them run fast
 - Prioritize and implement the optimizations
 - Compiler derby at the end
 - A "similar" application to the test set is provided the day before
 - The compiler that produced the fastest code is the winner
- Grade is divided into:
 - Documentation 6%
 - Justify your optimizations and the selection process
 - Optimization Implementation 12%
 - Producing correct code
 - Derby performance 12%

30%

The Quiz

- Three Quizzes
- In-Class Quiz
 - 50 Minutes (be on time!)
 - Open book, open notes

Why Study Compilers?

- Compilers enable programming at a high level language instead of machine instructions.
 - Malleability, Portability, Modularity, Simplicity, Programmer Productivity
 - Also Efficiency and Performance

Compilers Construction touches many topics in Computer Science

- Theory
 - Finite State Automata, Grammars and Parsing, data-flow
- Algorithms
 - Graph manipulation, dynamic programming
- Data structures
 - Symbol tables, abstract syntax trees
- Systems
 - Allocation and naming, multi-pass systems, compiler construction
- Computer Architecture
 - Memory hierarchy, instruction selection, interlocks and latencies
- Security
 - Detection of and Protection against vulnerabilities
- Software Engineering
 - Software development environments, debugging
- Artificial Intelligence
 - Heuristic based search

Power of a Language

- Can use to describe any action
 - Not tied to a “context”
- Many ways to describe the same action
 - Flexible

Suman Amarasinghe

13

6.035 ©MIT Fall 1998

Programming Languages

- Properties
 - need to be precise
 - need to be concise
 - need to be expressive
 - need to be at a high-level (lot of abstractions)

Suman Amarasinghe

14

6.035 ©MIT Fall 1998

1. How to instruct the computer

- Write a program using a programming language
 - High-level Abstract Description
- Microprocessors talk in assembly language
 - Low-level Implementation Details



Suman Amarasinghe

15

6.035 ©MIT Fall 1998

1. How to instruct the computer

- Input: High-level programming language
- Output: Low-level assembly instructions
- Compiler does the translation:
 - Read and understand the program
 - Precisely determine what actions it require
 - Figure-out how to faithfully carry-out those actions
 - Instruct the computer to carry out those actions

Suman Amarasinghe

16

6.035 ©MIT Fall 1998

Input to the Compiler

- Standard imperative language (Java, C, C++)
 - State
 - Variables,
 - Structures,
 - Arrays
 - Computation
 - Expressions (arithmetic, logical, etc.)
 - Assignment statements
 - Control flow (conditionals, loops)
 - Procedures

Suman Amarasinghe

17

6.035 ©MIT Fall 1998

Output of the Compiler

- State
 - Registers
 - Memory with Flat Address Space
- Machine code – load/store architecture
 - Load, store instructions
 - Arithmetic, logical operations on registers
 - Branch instructions

Suman Amarasinghe

18

6.035 ©MIT Fall 1998

Semantic Analyzer

```

int * foo(i, j, k)
{
  int i;
  int j;
  {
    int x;
    x = x + j + N;
    return j;
  }
}

```

Type not declared
 Mismatched return type
 Uninitialized variable used
 Undeclared variable

Suman Amarasinghe 25 6.035 ©MIT Fall 1998

Optimizer

```

int sumcalc(int a, int b, int N)
{
  int i;
  int x, y;
  x = 0;
  y = 0;
  for(i = 0; i <= N; i++) {
    x = x+4*a/b*i+(i+1)*(i+1);
    x = x + b*y;
  }
  return x;
}

```

```

int sumcalc(int a, int b, int N)
{
  int i;
  int x, t, u, v;
  x = 0;
  u = ((a<<2)/b);
  v = 0;
  for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + v + t*t;
    v = v + u;
  }
  return x;
}

```

Suman Amarasinghe 26 6.035 ©MIT Fall 1998

Code Generator

```

int sumcalc(int a, int b, int N)
{
  int i;
  int x, t, u, v;
  x = 0;
  u = ((a<<2)/b);
  v = 0;
  for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + v + t*t;
    v = v + u;
  }
  return x;
}

```

```

sumcalc:
xorl   %r8d, %r8d
xorl   %ecx, %ecx
movl   %edx, %r9d
cmpl   %edx, %r9d
jg     .L7
shll   %2, %edi
.L5:   movl   %edi, %eax
cld
idivl  %eax
leal   1(%ecx), %edx
movl   %eax, %r10d
imull  %ecx, %r10d
movl   %edx, %ecx
imull  %edx, %ecx
leal   (%r10,%ecx), %eax
movl   %edx, %ecx
addl   %eax, %r8d
cmpl   %r9d, %edx
jle    .L5
.L7:   movl   %r8d, %eax
ret

```

Suman Amarasinghe 27 6.035 ©MIT Fall 1998

Program Translation

- Correct
 - The actions requested by the program has to be faithfully executed
- Efficient
 - Intelligently and efficiently use the available resources to carry out the requests
 - (the word optimization is used loosely in the compiler community – Optimizing compilers are never optimal)

Suman Amarasinghe 28 6.035 ©MIT Fall 1998



Efficient Execution

- Mapping from High to Low
 - Simple mapping of a program to assembly language produces inefficient execution
 - Higher the level of abstraction ⇒ more inefficiency
- If not efficient
 - High-level abstractions are useless
- Need to:
 - provide a high level abstraction
 - with performance of giving low-level instructions

Suman Amarasinghe 29 6.035 ©MIT Fall 1998

Efficient Execution help increase the level of abstraction

- Programming languages
 - From C to OO-languages with garbage collection
 - Even more abstract definitions
- Microprocessor
 - From simple CISC to RISC to VLIW to

Suman Amarasinghe 30 6.035 ©MIT Fall 1998

Lets Optimize...

```
int sumcalc(int a, int b, int N)
{
  int i, x, y;
  x = 0;
  y = 0;
  for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
  }
  return x;
}
```

Constant Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x + b*0;
}
return x;
```

Algebraic Simplification

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x;
}
return x;
```

Copy Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```

Common Subexpression Elimination

```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  t = i+1;
  x = x + (4*a/b)*i + t*t;
}
return x;
```

Dead Code Elimination

```
int i, x, t;
x = 0;
for(i = 0; i <= N; i++) {
  t = i+1;
  x = x + (4*a/b)*i + t*t;
}
return x;
```

