Lecture 1: Introduction

Intro. to Computer Language Engineering

Course Administration info.
Course Administration

- Staff
- Optional Text
- Course Outline
- The Project
- Project Groups
- Grading
Staff

• Lecturer
  – Prof. Saman Amarasinghe  saman@mit.edu  253-8879  32-G778
  – Prof. Armando Solar-Lezama  asolar@csail.mit.edu  258-9727  32-G840

• Course Secretary
  – Mary McDavit  mmcdavit@csail.mit.edu  253-9620  32-G735

• Teaching Assistant
Reference Textbooks

- **Modern Compiler Implementation in Java (Tiger book)**
  A.W. Appel
  Cambridge University Press, 1998
  ISBN 0-52158-388-8
  
  A textbook tutorial on compiler implementation, including techniques for many language features.

- **Compilers: Principles, Techniques and Tools (Dragon book)**
  Aho, Lam, Sethi and Ullman
  Addison-Wesley, 2006
  ISBN 0321486811

  The classic compilers textbook, although its front-end emphasis reflects its age. New edition has more optimization material.

- **Advanced Compiler Design and Implementation (Whale book)**
  Steven Muchnick
  Morgan Kaufman Publishers, 1997
  ISBN 1-55860-320-4

  Essentially a recipe book of optimizations; very complete and suited for industrial practitioners and researchers.

- **Engineering a Compiler (Ark book)**
  Keith D. Cooper, Linda Torczon
  Morgan Kaufman Publishers, 2003
  ISBN 1-55860-698-X

  A modern classroom textbook, with increased emphasis on the back-end and implementation techniques.

- **Optimizing Compilers for Modern Architectures**
  Randy Allen and Ken Kennedy
  Morgan Kaufman Publishers, 2001
  ISBN 1-55860-286-0

  A modern textbook that focuses on optimizations including parallelization and memory hierarchy optimization.
The Project: The Five Segments

1. Lexical and Syntax Analysis
2. Semantic Analysis
3. Code Generation
4. Data-flow Analysis
5. 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%
- In-class Quizzes 30% (10% each)
- In-class mini-quizzes 10% (0.5% each)
Grades for the Project

- Scanner/Parser: 5%
- Semantic Checking: 7.5%
- Code Generation: 10%
- Data-flow Analysis: 7.5%
- Optimizations: 30%

Total: 60%
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

- Do any optimizations you choose
  - Including parallelization for multicores

- 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
Mini Quizzes

• You already got one.
• Given at the beginning of the class; Collected at the end
• Collaboration is OK
• This is in lieu of time consuming problem sets
Outline

• Course Administration Information
• Introduction to computer language engineering
  – High-level programming languages
  – The role of the compiler
  – Anatomy of a compiler
  – Program optimization
How to instruct a computer

• In the beginning...
How to instruct a computer

• Programming bit-by-bit doesn’t scale
  – we want to instruct the computer at a higher level of abstraction

• How about natural languages?
  – English??
  - “Open the pod bay doors, Hal.”
  - “I am sorry Dave, I am afraid I cannot do that”
  – We are not there yet!!

• Natural Languages:
  – Powerful, but...
  – Ambiguous
    • Same expression describes many possible action

  Ex: “At last, a computer that understands you like your mother”
  -- 1985 McDonnell-Douglas Ad
Programming Languages

• We want a high level of abstraction

• We also want
  – precision (avoid ambiguity)
  – conciseness
  – expressiveness
  – modularity

• The goal is to achieve:
  – Programmer Productivity
  – Also Efficiency and Performance
Implementing Programming Languages

• Static languages are implemented by compilers

• That’s what this course is all about!
The role of the compiler

- Translate a program
  - from high-level description to low-level implementation

High-level of Abstraction

Compiler

Low-level of Abstraction

Program written in a Programming Languages

Assembly Language Translation
High-level Abstract Description to Low-level Implementation Details

President

My poll ratings are low, let's invade a small nation

General

Cross the river and take defensive positions

Sergeant

Forward march, turn left
Stop!, Shoot

Foot Soldier
The role of the compiler

Program written in a Programming Languages ➔ Compiler ➔ Assembly Language Translation

- Translation involves:
  - 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
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
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
Example (input program)

```c
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;
}
```
Example (Output assembly code)

```
sumcalc:
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -4(%rbp)
    movl %esi, -8(%rbp)
    movl %edx, -12(%rbp)
    movl $0, -20(%rbp)
    movl $0, -24(%rbp)
    movl $0, -16(%rbp)
    .L2:
        movl -16(%rbp), %eax
        cmpl -12(%rbp), %eax
        jg .L3
        movl -4(%rbp), %eax
        leal 0(%rax,4), %edx
        leaq -8(%rbp), %rax
        movq %rax, -40(%rbp)
        movq %edx, %eax
        movl -40(%rbp), %rcx
        cltd
        idivl (%rcx)
        movl %eax, -28(%rbp)
        movl -28(%rbp), %edx
        imull -16(%rbp), %edx
        movl -16(%rbp), %eax
        incl %eax
        imull %eax, %eax
        addl %eax, %edx
        leaq -20(%rbp), %rax
        addl %edx, (%rax)
        movl -8(%rbp), %eax
        movl %eax, %edx
        imull -24(%rbp), %edx
        leaq -20(%rbp), %rax
        addl %edx, (%rax)
        leaq -16(%rbp), %rax
        incl (%rax)
        jmp .L2
    .L3:
        movl -20(%rbp), %eax
    leave
    ret
```
Compiler 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, parallelism
- Security
  - Detection of and Protection against vulnerabilities
- Software Engineering
  - Software development environments, debugging
- Artificial Intelligence
  - Heuristic based search
Outline

• Course Administration Information
• Introduction to computer language engineering
  – High-level programming languages
  – The role of the compiler
  – Anatomy of a compiler
  – Program optimization
Anatomy of a Compiler

- Program written in a Programming Languages
- Compiler
- Assembly Language Translation
Anatomy of a Compiler

- Program (character stream)
- Lexical Analyzer (Scanner)
- Token Stream
Lexical Analyzer (Scanner)

2 3 4 * ( 1 1 + - 2 2 )

Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op
Lexical Analyzer (Scanner)

2 3 4 * ( 1 1 + - 2 2 )

Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op

18..23 + val#ue

Variable names cannot have ‘#’ character

Not a number
Anatomy of a Compiler

Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree
Syntax Analyzer (Parser)

```
num ‘*’ (‘ num ‘+’ num ‘)’
```

```
<expr>
  <expr>
  <op>
  num
  *
  ( <expr> )
  <expr>
  <op>
  <expr>
  num
  +
  num
```
Syntax Analyzer (Parser)

```c
int * foo(i, j, k)
{
    int i;
    int j;
    for(i=0; i < j) {
        fi(i>j)
        return j;
    }
}
```

- Extra parentheses
- Missing increment
- Not an expression
- Not a keyword
Anatomy of a Compiler

Program (character stream)
Lexical Analyzer (Scanner)
Token Stream
Syntax Analyzer (Parser)
Parse Tree
Semantic Analyzer
Intermediate Representation
int * foo(i, j, k)
    int i;
    int j;
    {
        int x;
        x = x + j + N;
        return j;
    }
Anatomy of a Compiler

Program (character stream) →
Lexical Analyzer (Scanner) →
Token Stream →
Syntax Analyzer (Parser) →
Parse Tree →
Semantic Analyzer →
Intermediate Representation →
Code Optimizer →
Optimized Intermediate Representation
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;
}
Anatomy of a Computer

- Program (character stream)
- Lexical Analyzer (Scanner)
- Token Stream
- Syntax Analyzer (Parser)
- Parse Tree
- Semantic Analyzer
- Intermediate Representation
- Code Optimizer
- Optimized Intermediate Representation
- Code Generator
- Assembly code
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;
}
Outline

- Course Administration Information
- Introduction to computer language engineering
  - High-level programming languages
  - The role of the compiler
  - Anatomy of a compiler
  - Program optimization
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)
Efficient Execution

Cross the river and take defensive positions

★ ★ ★ ★★ General

★ ★ ☆ Sergeant

Foot Soldier
Efficient Execution

General

Cross the river and take defensive positions

Sergeant

Where to cross the river? Use the bridge upstream or surprise the enemy by crossing downstream? How do I minimize the casualties??

Foot Soldier
My poll ratings are low, let's invade a small nation

Russia or Bermuda? Or just stall for his poll numbers to go up?
Efficient Execution

• Mapping from High to Low
  – Simple mapping of a program to assembly language produces inefficient execution
  – Higher the level of abstraction $\Rightarrow$ more inefficiency

• If not efficient
  – High-level abstractions are useless

• Need to:
  – provide a high level abstraction
  – with performance of giving low-level instructions
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 ....
The Multicore Dilemma

- Superscalars
- Multicores

High Level Language

Simple von Neumann Machine

Hardware

Compiler

High Level Language

Multiple exposed cores

Hardware

Compiler??
The Multicore Dilemma

- Superscalars
- Multicores

High Level Language

Simple von Neumann Machine

Parallel Language

Multiple exposed cores

Hardware

Compiler

Compiler
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;
}
Optimization Example

```assembly
pushq  %rbp
movq  %rsp, %rbp
movl  %edi, -4(%rbp)

movl  %esi, -8(%rbp)
movl  %edx, -12(%rbp)
movl  $0, -20(%rbp)
movl  $0, -24(%rbp)
movl  $0, -16(%rbp)

movl -16(%rbp), %eax
cmpl -12(%rbp), %eax
jg .L3

.L2:
    movl -16(%rbp), %eax
cmpl -12(%rbp), %eax
    leal 0(,%rax,4), %edx
    leal -8(%rbp), %rax
    movq %rax, -40(%rbp)
    movl %edx, %eax
    movq -40(%rbp), %rcx
cld
    idivl (%rcx)
    movl %eax, -28(%rbp)
    movl -28(%rbp), %edx
    imull -16(%rbp), %edx
    movl -16(%rbp), %eax
    incl %eax
    imull %eax, %eax
    addl %eax, %edx
    leaq -20(%rbp), %rax
    addl %edx, (%rax)
    movl -8(%rbp), %eax
    movl %eax, %edx
    imull -24(%rbp), %edx
    leaq -20(%rbp), %rax
    addl %edx, (%rax)
    leaq -16(%rbp), %rax
    incl (%rax)
jmp (%rax)

.L3:
    movl -20(%rbp), %eax
leave
ret
```

```c
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, 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 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

```c
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

```c
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 + b*0;
}
return x;
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;
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

```c
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

```c
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;
```
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;
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;
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

```c
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

```c
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

```c
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;
```
Loop Invariant Removal

```c
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;
```
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;
Loop Invariant Removal

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

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}

return x;
Strength Reduction

```c
int i, x, t, u;
x = 0;
u = (4*a/b);

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}
return x;
```
Strength Reduction

```c
int i, 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;
```
Register Allocation

Local variable $x$
Local variable $y$
Local variable $I$
fp
Register Allocation

\[
\begin{array}{c}
\text{Local variable } X \\
\text{Local variable } Y \\
\text{Local variable } I \\
\end{array}
\]

\[
\begin{align*}
$r8d &= X \\
$r9d &= t \\
$r10d &= u \\
$ebx &= v \\
$ecx &= i
\end{align*}
\]
Optimized Example

```c
int sumcalc(int a, int b, int N) {
    int i, 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;
}
```
The Effect of Optimization

Unoptimized Code

```assembly
pushq  %rbp
movq  %rsp, %rbp
movl  %edi, -4(%rbp)
movl  %esi, -8(%rbp)
movl  %edx, -12(%rbp)
movl  80, -20(%rbp)
movl  80, -24(%rbp)
movl  80, -16(%rbp)
.L2:
    movl  -16(%rbp), %eax
cmpl  -12(%rbp), %eax
    jg    .L3
movl  -4(%rbp), %eax
lea   0(%rax,4), %edx
lea   -8(%rbp), %rax
movq  %rax, -40(%rbp)
movl  %edx, %eax
movq  -40(%rbp), %rcx
cld
idivl  (%rcx)
movl  -16(%rbp), %edx
imull  %edx, %eax
inc1  %eax
imull  %eax, %eax
addl  %eax, %edx
lea   -20(%rbp), %rax
addl  %edx, (%rax)
movl  -8(%rbp), %eax
movl  %eax, %edx
imull  -24(%rbp), %edx
lea   -20(%rbp), %rax
addl  %edx, (%rax)
lea   -16(%rbp), %rax
inc1  (%rax)
jmp   .L2
.L3:
    movl  -20(%rbp), %eax
leave
ret
```

Optimized Code

```assembly
xorl  %r8d, %r8d
xorl  %ecx, %ecx
movl  %edx, %r9d
cmpl  %edx, %r8d
jg    .L7
sall  $2, %edi

.L5:  movl  %edi, %eax
cld
idivl  %esi
leal  1(%rcx), %edx
movl  %eax, %r10d
imull  %ecx, %r10d
movl  %edx, %ecx
imull  %edx, %ecx
leal  (%r10,%rcx), %eax
movl  %edx, %ecx
addl  %eax, %edx
lea   (%r10,%rcx), %eax
movl  %edx, %ecx
addl  %eax, %edx
lea   (%r9d,%edx)
jle   .L5

.L7:  movl  %r8d, %eax
ret
```

Inner Loop:
4*mov + 2*lea + 1*add/inc + 3*div/mul + 2*cmp/br/jmp = 12 instructions

Inner Loop:
10*mov + 5*lea + 5*add/inc + 4*div/mul + 5*cmp/br/jmp = 29 instructions
Compilers Optimize Programs for...

• Performance/Speed
• Code Size
• Power Consumption
• Fast/Efficient Compilation
• Security/Reliability
• Debugging