Lecture 1: Introduction

Intro. to Computer Language Engineering
Course Administration info.
Outline

• Course Administration Information
• Introduction to computer language engineering
  – Why do we need a compiler?
  – What are compilers?
  – Anatomy of a compiler
Course Administration

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

- *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.

- *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.

- *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
  - (Design Document)
  - (Project Checkpoint)
- Project Due
Project Groups

- 1\textsuperscript{st} project is an individual project
- Projects 2 to 5 are group projects consists of 3 to 4 students
- Grading
  - All group members (mostly) get the same grade
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
  - What are compilers?
  - Why should we learn about them?
  - Anatomy of a compiler
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, parallelism
- Security
  - Detection of and Protection against vulnerabilities
- Software Engineering
  - Software development environments, debugging
- Artificial Intelligence
  - Heuristic based search for best optimizations
Power of a Language

• Can use to describe any action
  – Not tied to a “context”

• Many ways to describe the same action
  – Flexible
How to instruct a computer

• 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 actions
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)
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

Figure by MIT OpenCourseWare.

Saman Amarasinghe
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
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
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)

```assembly
sumcalc:
pushq %rbp
movq %rsp, %rbp
movl %edi, -4(%rbp)
movl %esi, -8(%rbp)
movl $0, -20(%rbp)
movl $0, -24(%rbp)
movl $0, -16(%rbp)
.L2:
  movl -16(%rbp), %eax
cmpl -12(%rbp), %eax
  jg .L3
leal 0(,%rax,4), %edx
leaq -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 .L2
.L3:
movl -20(%rbp), %eax
leave
ret
```

.size sumcalc, .-sumcalc
.section Lframe1:
.long .LECIE1-.LSCIE1
.LSCIE1:.long 0x0
.byte 0x1
.string ""
.uleb128 0x1
.sleb128 -8
.byte 0x10
.byte 0xc
.uleb128 0x7
.uleb128 0x8
.byte 0x90
.uleb128 0x1
.align 8
.LECIE1:.long .LEFDE1-.LASFDE1
.long .LASFDE1-.Lframe1
.quad .LFB2
.quad .LFE2-.LFB2
.byte 0x4
.byte 0xe
.uleb128 0x10
.byte 0x86
.uleb128 0x2
.byte 0x4
.uleb128 0x6
.align 8
.long .LCFI1-.LCFI0
.byte 0xd
.uleb128 0x6
.align 8
Mapping Time Continuum
Compilation to Interpretation

• Compile time
  - Ex: C compiler

• Link time
  - Ex: Binary layout optimizer

• Load time
  - Ex: JIT compiler

• Run time
  - Ex: Java Interpreter
Anatomy of a Computer

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

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

Saman Amarasinghe

©MIT Fall 1998
Lexical Analyzer (Scanner)

Num(234) * (11 + -22)

Saman Amarasinghe

6.035 ©MIT Fall 1998
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 Computer

Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree
Syntax Analyzer (Parser)

\[ \text{num} \times (\text{num} + \text{num}) \]

\[
\begin{array}{c}
\text{<expr>} \\
\text{<expr>} \text{<op>} \text{<expr>}
\end{array}
\]

\[
\begin{array}{c}
\text{num} \\
\text{*} \\
\text{<expr>}
\end{array}
\]

\[
\begin{array}{c}
\text{num} \\
\text{+} \\
\text{num}
\end{array}
\]
Syntax Analyzer (Parser)

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

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

1. Program (character stream)
2. Lexical Analyzer (Scanner)
3. Token Stream
4. Syntax Analyzer (Parser)
5. Parse Tree
6. Semantic Analyzer
7. Intermediate Representation
int * foo(i, j, k)
{
    int i;
    int j;
    int x;
    x = x + j + N;
    return j;
}
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
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

1. Program (character stream)
2. Lexical Analyzer (Scanner)
3. Token Stream
4. Syntax Analyzer (Parser)
5. Parse Tree
6. Semantic Analyzer
7. Intermediate Representation
8. Code Optimizer
9. Optimized Intermediate Representation
10. Code Generator
11. 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;
}
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

Figure by MIT OpenCourseWare.

Saman Amarasinghe
Cross the river and take defensive positions

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

Efficient Execution

General

Sergeant

Foot Soldier
Efficient Execution

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

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

Figure by MIT OpenCourseWare.
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
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

Compiler

Hardware

Multiple exposed cores

Compiler??

Hardware

Saman Amarasinghe  ©MIT Fall 1998
The Multicore Dilemma

- Superscalars
- Multicores

High Level Language

Simple von Neumann Machine

Parallel Language

Multiple exposed cores

Compiler

Hardware

Compiler

Hardware

Saman Amarasinghe
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;
}
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)
movl  %edx, %eax
movq  -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
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;
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*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 + 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 + 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;
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;
```
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;
```
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;
```
```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

int i, x, t, u;
x = 0;
   
   /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, 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

fp

Local variable X
Local variable Y
Local variable I

$r8d = X$
$r9d = t$
$r10d = u$
$ebx = v$
$ecx = i$
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;
}
Unoptimized Code

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:
cmpl -16(%rbp), %eax
movj .L3
movl -4(%rbp), %eax
lea 0, (%rax, 4), %edx
leaq -8(%rbp), %rax
movq %rax, -40(%rbp)
movl %edx, %eax
movq -40(%rbp), %rcx
cltd
cltd (%rcx)
movl %eax, -28(%rbp)
movl -20(%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:
leave
movl %eax

Optimized Code

xorl %rd, %rd
xorl %ecx, %ecx
movl %edx, %rd
jmp .L7
sall $2, %edi
.L5:
cld
movl %esi
leal 1(%rcx), %edx
imull %ecx, %rd
movl %edx, %ecx
imull %edx, %ecx
lea -10, (%rcx), %eax
movl %edx, %ecx
addl %eax, %rd
cmpl %eax, %rd
jle .L5
movl
ret

Inner Loop:

10*mov + 5*lea + 5*add/inc + 4*div/mul + 5*cmp/br/jmp = 29 instructions
Execution time = 43 sec

4*mov + 2*lea + 1*add/inc + 3*div/mul + 2*cmp/br/jmp = 12 instructions
Execution time = 17 sec
Compilers Optimize Programs for...

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.