6.035

Fall 2000

Lecture 1: Introduction

Intro. to Computer Language Engineering
Lexical Analysis
Course Administration info.

Handouts

- Handout 1 - At a Glance
- Handout 2 - Group Information
- Handout 3 - General Information
- Handout 4 - Project Overview
- Handout 5 - Athena, Tools
- Handout 6 - Decaf Language Definition
- Handout 7 - Espresso Language Definition
- Handout 8 - The Scanner
- Lecture Notes 1 – Introduction
- Voucher to pickup the textbook

Course Administration

- Staff
- Text
- Course Outline
- Project
- Credit, Project Group and Sections
- Grading

Staff

- Lecturers
  - Prof. Saman Amarasinghe
  - Prof. Martin Rinard
- Course Secretary
  - Rachel Allen
- Teaching Assistants
  - Nathan Williams
  - Matt Deeds
  - David Maze
  - Kenneth Lu

Textbook

- **Engineering a Compiler**
  By Keith Cooper and Linda Torczon
- Unpublished Text
  - We are a beta test site
- Pros. and Cons.
  - Pay only for printing ($14 available from 38-501)
  - May/will have bugs
  - Your help is expected in debugging the book

Optional Textbooks

Tiger Book
- Modern Compiler Implementation in Java
  by A.W. Appel
  Cambridge University Press, 1998

Whale Book
- Advanced Compiler Design and Implementation
  by Steven Muchnick
  Morgan Kaufman Publishers, 1997

Dragon Book
- Compilers -- Principles, Techniques and Tools
  by Aho, Sethi and Ullman
  Addison-Wesley, 1988

The Six Segments

❶ Lexical Analysis
❷ Syntax Analysis (Parsing)
❸ Semantic Analysis
❹ Code Generation
❺ Data-flow Optimizations
❻ Instruction Optimizations

Each Segment...

- Segment Start
  - Project Description
  - (Problem set)
- Lectures
  - 2 to 5 lectures
- (In Class Quiz)
- Project Time
- (Project checkpoint)
- Project Time
- Project Due

Comes in two flavors

- Decaf
  - 12 unit version of the course
  - Implements a simple imperative language
- Espresso
  - 18 unit version of the course
    - Sign-up for 6.907 for 6 additional units
  - Implements an object oriented language
    - A fair subset of Java

Project Groups and Sections

- Each project group consists of 3 to 4 students
- All members of a group should take either the decaf version or the espresso version
- All the members of the group should be in the same section
- Grading
  - Scanner project is individually graded
  - All group members get the same grade for the rest

Grades

- For 6.035 (12 units)
  - Compiler project 54%
  - Problem Sets 16% (8% each)
  - In-class Quizzes 30% (10% each)
- For 6.907 (additional 6 units)
  - Compiler project 100%

Grades for the Project

	6.035	6.907
– Scanner	6%	
– Parser	6%	12%
– Semantic Checking	8%	18%
– Code Generation	14%	28%
– Data-flow Opt.	12%	24%
– Instruction Opt.	8%	18%
	54%	**100%**

The Quiz/Problem Set

- Each segment has either a quiz or a problem set
- **In-Class Quiz**
  - 50 Minutes (be on time!)
  - Open book, open notes
- Problem Sets
  - Due during Recitation
  - Solutions given-out at the Recitation
  - Thus, no extensions

What is Computer Language Engineering

1. How to give instructions to a computer
2. How to make the computer carryout the instructions efficiently

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

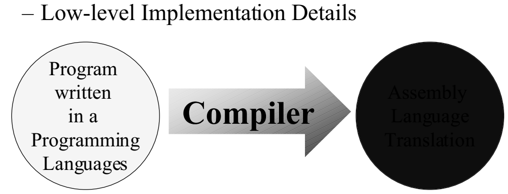
- How about natural languages?
  - English??
  - **"Open the pod bay doors, Hal."**
  - **"I am sorry Dave, I am afraid I cannot do that"**
  - We are not there yet!!
- Natural Languages:
  Same expression describes many possible actions
  - Ambiguous
- Use a programming language
  - Examples: Java, C, C++, Pascal, BASIC, Scheme

Programming Languages

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

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 has to:
  - Read and understand the program
  - Precisely determine what actions it require
  - Figure-out how to carry-out those actions
  - Instruct the computer to carry out those actions

Example (input program)

```
int expr(int n)
{
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```


Example (Output assembly code)

```
    lda $30,-32($30)
    stq $26,0($30)
    stq $15,8($30)
    bis $30,$30,$15
    bis $16,$16,$1
    stl $1,16($15)
    lds $f1,16($15)
    sts $f1,24($15)
    ldl $5,24($15)
    bis $5,$5,$2
    s4addq $2,0,$3
    ldl $4,16($15)
    mull $4,$3,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    stl $2,20($15)
    ldl $0,20($15)
    br $31,$33
$33:
    bis $15,$15,$30
    ldq $26,0($30)
    ldq $15,8($30)
    addq $30,32,$30
    ret $31,($26),1
```


Efficient Execution of the Actions

- 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

Example (Output assembly code)

Unoptimized Code

```
    lda $30,-32($30)
    stq $26,0($30)
    stq $15,8($30)
    bis $30,$30,$15
    bis $16,$16,$1
    stl $1,16($15)
    lds $f1,16($15)
    sts $f1,24($15)
    ldl $5,24($15)
    bis $5,$5,$2
    s4addq $2,0,$3
    ldl $4,16($15)
    mull $4,$3,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    ldl $3,16($15)
    addq $3,1,$4
    mull $2,$4,$2
    stl $2,20($15)
    ldl $0,20($15)
    br $31,$33
$33:
    bis $15,$15,$30
    ldq $26,0($30)
    ldq $15,8($30)
    addq $30,32,$30
    ret $31,($26),1
```

Optimized Code

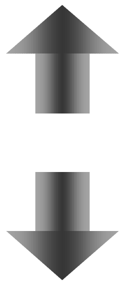
```
s4addq $16,0,$0
mull $16,$0,$0
addq $16,1,$16
mull $0,$16,$0
mull $0,$16,$0
ret $31,($26),1
```


Compilers Optimize Programs for…

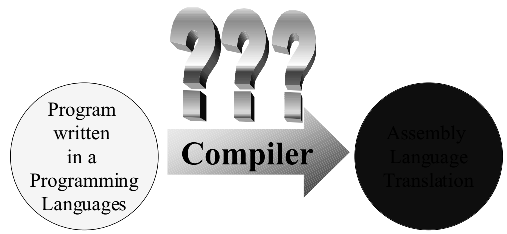
- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging

Compilers 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 ….



Anatomy of a Computer



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

What is a Lexical Analyzer?

Source program text ⇒ Tokens

- What do we do in natural language processing?
- First we tokenize
- Example
  - Howareyoudoingmyfriend?

What is a Lexical Analyzer?

Source program text ⇒ Tokens

- What do we do in natural language processing?
- First we tokenize
- Example
  - Howareyoudoingmyfriend?

  Becomes
  - How are you doing my friend?

What is a Lexical Analyzer?

Source program text ⇒ Tokens

- Example of Tokens
  - Operators = + - > ({ := == <>
  - Keywords if while for int double
  - Numeric literals 43 6.035 -3.6e10 0x13F3A
  - Character literals 'a' '~' '\''
  - String literals "6.891" "Fall 98" "\"\" = empty"
- Example of non-tokens
  - White space space(' ') tab('\t') end-of-line('\n')
  - Comments /*this is not a token*/

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10) ID("var1") leq_op

Lexical Analyzer needs to...

- Partition input program text into subsequence of characters corresponding to tokens
- Attach the corresponding attributes to the tokens
- Eliminate white space and comments

Why is this not trivial?

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10) ID("var1") leq_op

- Precisely separate the text stream into the correct stream of tokens
  - ID("var1") not ID("var") Num(1)
  - ID("var1") leq_op not ID("var1<=")

In the next lecture

- How to precisely describe what substrings become tokens
- How to implement a lexical analyzer

Roadmap

Monday 9/4	Tuesday 9/5	Wednesday 9/6	Thursday 9/7
		Room 3-370 L1: Intro to compilers, course admin. info., lexical analysis **TODAY**	**Room 3-270** **L2: Lexical Analyzer** **Tools tutorial 7:00 – 9:00** **Next Lecture**
Monday 9/11	**Tuesday 9/12**	**Wednesday 9/13**	**Thursday 9/14**
	Room 3-270 L3: Syntax analysis, bottom-up parsing	**Room 3-370** L4: LR(0) Paring Algorithms and Parsing Tables	**Room 3-270** L5: LR(1) & LALR(1) Paring Algorithms

Scanner Segment assigned:

Project due on 9/11

Reading

- Chapter 1
- Chapter 2.1 – 2.4, 2.6