CS412/413

Introduction to
Compilers and Translators
Spring '01

Lecture 1: Overview

Outline

- About this course
- Introduction to compilers
  - What are compilers?
  - Why should we learn about them?
  - Anatomy of a compiler
- Introduction to lexical analysis
  - Text stream to tokens

Course Information

- MWF 10:10 - 11:00AM in Phillips 203
- Faculty: Andrew Myers
- Teaching Assistants: Michael Clarkson, Sunny Gleason, Lantian Zheng
- E-mail: cs412@cs.cornell.edu
- Web page: http://www.cs.cornell.edu/courses/cs412
- Newsgroup: cornell.class.cs412

CS 413 is required!

Textbooks

- Required text
  - Modern Compiler Implementation in Java. Andrew Appel.
- Optional texts
  - Compilers -- Principles, Techniques and Tools. Aho, Sethi and Ullman (The Dragon Book)
  - Advanced Compiler Design and Implementation. Steve Muchnick.
- Java reference
  - Java Language Specification. James Gosling, Bill Joy, and Guy Steele.
- All are on reserve in Engineering Library

Work

- Homeworks: 4, 20% total
  - 5/5/5/5
- Programming Assignments: 6, 50%
  - 5/7/8/10/10/10
- Exams: 2 prelims, 30%
  - 15/15
  - No final exam

Homeworks

- Three assignments in first half of course; one homework in second half
- *Not* done in groups—you may discuss with others but do your own work
  - Write down who you discussed problems with

Projects

- Six programming assignments
- Groups of 3-4 students
  - same grade for all
- Group information due Friday
  - we will respect consistent preferences
- Java will be implementation language

Assignments

- Due at beginning of class
- Late homeworks, programming assignments increasingly penalized
  - 1 day: 5%, 2 days: 15%, 3 days: 30%, 4 days: 50%
  - weekend = 1 day
  - Extensions often granted, but must be approved <u>2</u> days in advance
- Project files turned in to CSUGLAB directory

Why take this course?

- CS412 is an elective course
- Expect to learn:
  - practical applications of theory
  - parsing
  - deeper understanding of code
  - manipulation of complex data structures
  - how high-level languages are implemented in machine language
  - a little programming language semantics
  - Intel x86 architecture, Java
  - how to be a better programmer (esp. in groups)

What are Compilers?

- Translators from one representation of a program to another
- Typically: high-level source code to machine language (object code)
- Not always
  - Java compiler: Java to interpretable bytecodes
  - Java JIT: bytecode to executable image

Source Code

- Source code: optimized for human readability
  - expressive: matches human notions of grammar
  - redundant to help avoid programming errors
  - computation possibly not fully determined by code

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

Machine code

- Optimized for hardware
  - Redundancy, ambiguity reduced
  - Information about intent lost
  - Assembly code ≈ machine 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
```

How to translate?

- Source code and machine code mismatch
- Some languages farther from machine code than others ("higher-level")
- Goal:
  - source-level expressiveness for task
  - best performance for concrete computation
  - reasonable translation efficiency (< O(n^3))
  - maintainable code

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
```

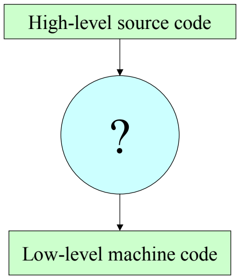
Optimized Code

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

Correctness

- Programming languages describe computation precisely
- Therefore: translation can be precisely described (a compiler can be correct)
- Correctness is very important!
  - hard to debug programs with broken compiler…
  - non-trivial: programming languages are expressive
  - implications for development cost, security
  - this course: techniques for building correct compilers

How to translate effectively?


High-level source code

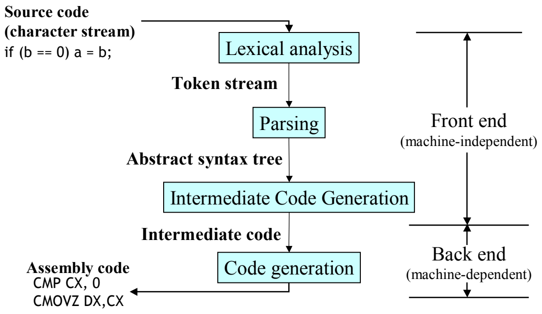
?

Low-level machine code

Idea: Translate in Steps

- Series of program representations
- Intermediate representations optimized for program manipulations of various kinds (checking, optimization)
- More machine-specific, less language-specific as translation proceeds

Simplified Compiler Structure



Source code
(character stream)
if (b == 0) a = b;

Lexical analysis

Token stream

Parsing

Abstract syntax tree

Intermediate Code Generation

Intermediate code

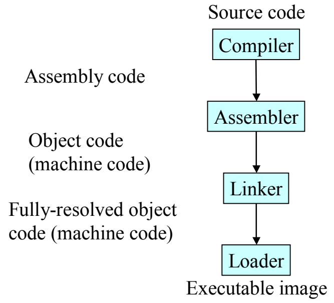
Code generation

Assembly code
```
CMP CX, 0
CMOVZ DX,CX
```

Front end
(machine-independent)

Back end
(machine-dependent)

Big picture



Source code

Compiler

Assembly code

Assembler

Object code
(machine code)

Linker

Fully-resolved object
code (machine code)

Loader

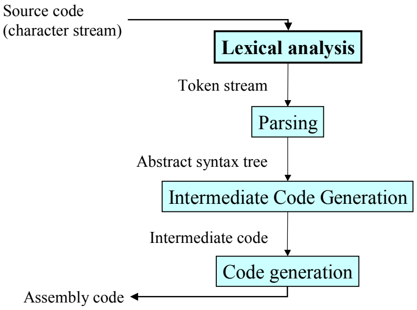
Executable image

Schedule

- Detailed schedule on web page, with links

Lexical analysis and parsing:	6
Semantic analysis:	5
Intermediate code:	4
Prelim #1	
Code generation:	3
Separate compilation and objects:	4
Optimization:	8
Prelim #2	
Run-time, link-time support:	2
Advanced topics:	7

First step: Lexical Analysis



Source code
(character stream)

Lexical analysis

Token stream

Parsing

Abstract syntax tree

Intermediate Code Generation

Intermediate code

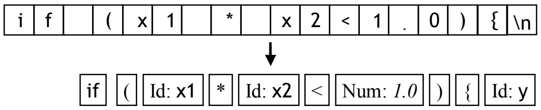
Code generation

Assembly code

What is Lexical Analysis?

- Converts character stream to token stream of pairs ⟨*token type, attribute*⟩

```
if  (x1 * x2<1.0) {
   y = x1;
}
```



i	f		(	x	1		*		x	2	<	1	.	0	)	{	\n

↓

if	(	Id: x1	*	Id: x2	<	Num: *1.0*	)	{	Id: y

Token stream

- Gets rid of whitespace, comments
- Only ⟨Token type, attribute⟩:
  ⟨Id, “x”⟩, ⟨Float, *1.0e0*⟩
- Token location preserved for debugging, error messages (source file, line number)
  ⟨Id, “x”, “Main.java”, 542⟩
- Issues:
  - how to specify tokens?
  - how to implement tokenizer/lexer