CSE 401 – Compilers

Course Introduction

Michael Ringenburg

Winter **2013** (Happy New Year!)

Credits

- Giving credit where it is due:
  – This course borrows heavily from previous versions here at UW (particularly Hal Perkins)
  – Also from UW CSE PMP 501 (Perkins)
  – And my undergraduate compilers course, Dartmouth CS 48 (Cormen), from 1999.
    - I still remember the project, vividly ☺

Agenda

- **Introductions**
- Administrivia
- What's a compiler, and how does it work (at a high level)?
- Why study compilers?
- A brief history of compilers

Who Am I?

- Michael (or Mike) Ringenburg
  - “Final”-year PhD student, working with Dan Grossman and Luis Ceze.
  - Spent 8 years writing compilers for Cray supercomputers, 5 of those as the technical lead for the XMT auto-parallelizing compiler.
  - Consult with Cray/YarcData about compilers and the URIKA graph database every Thursday.
  - Office: CSE 212, there most days except Thursday
  - “Official” office hours: TBD (Vote on the Doodle!)
  - Email: miker[at]cs.washington.edu

Special Note

- I may need to miss a class or two in March, with little notice … We are expecting our second child in mid-March.
- I will try to finish all the material needed for the project before then.
- We will try to find someone to cover any lecture(s) I need to miss – but be sure to check email for any last minute cancellations or changes.

TAs

- Zachary Stein
  - Email: steinz[at]cs.washington.edu
  - Office Hours: TBD (Vote on the Doodle!)
  - …
- Laure Thompson
  - Email: laurejt[at]cs.washington.edu
  - Office Hours: TBD (Vote on the Doodle!)
  - …

Agenda

- Introductions
- **Administrivia**
- What's a compiler, and how does it work (at a high level)?
- Why study compilers?
- A brief history of compilers

Administrivia (noun): the tiresome but essential details that must be taken care of and tasks that must be performed in running an organization. (www.freedictionary.com)

Prerequisites

- A little confusing due to the new core curriculum
- Official prereq: (326 & 378) | (332 & 351)
  - E.g., data structures and machine organization
- I assume most of you have taken 332 and 351, since the older prereqs have not been offered recently.
  - Let me know if this is not correct.
  - For this course, the main difference is the amount of exposure to x86-64 assembly language.
  - We'll review what you need to know for the project.

Overloads

- The class is full. But - if you're interested and haven't been able to register, I have a course overload form with me.
  - Come see me after class to sign the form.
  - I hope to be able to take a few more (but it depends on whether we can fit in this room!)
  - The prerequisites are important – otherwise you may slow the class (and your project team) down.
    - But a mix of the old and new prereqs is probably okay (e.g., 326 and 351, or 332 and 378) .

Course Meetings

- Lectures
  - MWF 12:30-1:20 here (EEB 045 – right here!)
- Sections Thursdays
  - AA: 12:30 (SAV 131), TA: TBD
  - AB: 1:30 (EEB 037), TA: TBD
  - Some sections will deliver important, project-related material, so please attend.
  - Locations may change – we are working on getting them in the same building
  - No sections this week – not far enough along yet

Communications

- Course web site (http://www.cs.washington.edu/education/courses/cse401/13wi/)
- Discussion board
  - For anything related to the course
  - Join in! Help each other out. Staff will monitor the board, but helping each other is a great way to learn.
- Mailing list: automatically subscribed if you are enrolled.
- Staff list: cse401-staff[at]cs.washington.edu
  - We prefer you send questions here, rather than to individual TAs or instructor.

Requirements & Grading

- Primary goal of this course is to write your own compiler (in teams of two). Grading reflects this.
- Roughly
  - 55% project
  - 15% individual written homeworks (probably 3, so 5% each)
  - 10% midterm exam (February 15?)
  - 15% final exam (Thursday, March 21, 8:30am ☹, here)
  - 5% other (be a good course citizen)

We reserve the right to adjust as needed

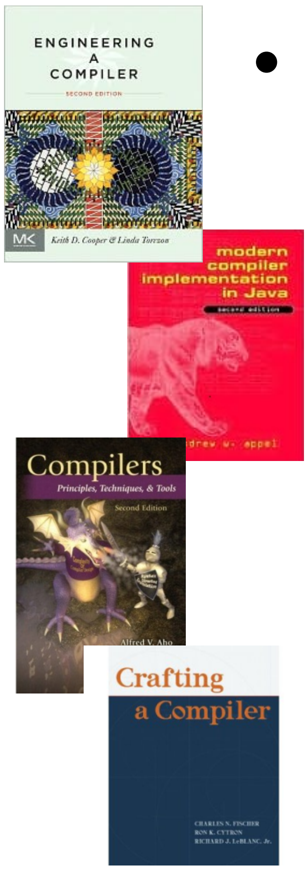
CSE 401 Course Project

- Best way to learn about compilers is to build one yourself!
  - You'll also learn how to use Jflex/Cup, which are useful even if you never write a real compiler
- Course project
  - Mini Java compiler: classes, objects, etc.
    - Basically, Java cut down to essentials
    - From the Appel textbook (but you don't need that text)
  - Generate executable x86-64 code
  - Completed in steps through the quarter
    - Where wind up at the end is the biggest part, but the intermediate steps will count towards your grade as well.

Project Groups

- You should work in teams of two
  - Pair programming encouraged
- Project should run on lab linux machines (or attu) when built with ant.
  - Project page has link to CSE Virtual Machine info (virtually run CSE linux environment at home)
- Pick partners soon (by the end of this week would be best)
  - Send course staff an email letting us know who your partner is.

Books



- Four good books, all on Eng. Lib. Reserve:
  – Cooper & Torczon, *Engineering a Compiler.* "Official text" New edition last year, but first is still good (and very similar).
  – Appel, *Modern Compiler Implementation in Java*, 2nd ed. MiniJava adapted from here.
  – Aho, Lam, Sethi, Ullman, "Dragon Book", 2nd ed (but 1st ed is also fine)
  – Fischer, Cytron, LeBlanc, *Crafting a Compiler*

Academic Integrity

- We want a cooperative group working together to do great stuff!
- But: you must never misrepresent work done by someone else as your own, without proper credit
- Know the rules – ask if in doubt or if tempted

Agenda

- Introductions
- Administrivia
- **What's a compiler, and how does it work (at a high level)?**
- Why study compilers?
- A brief history of compilers

What do compilers do?

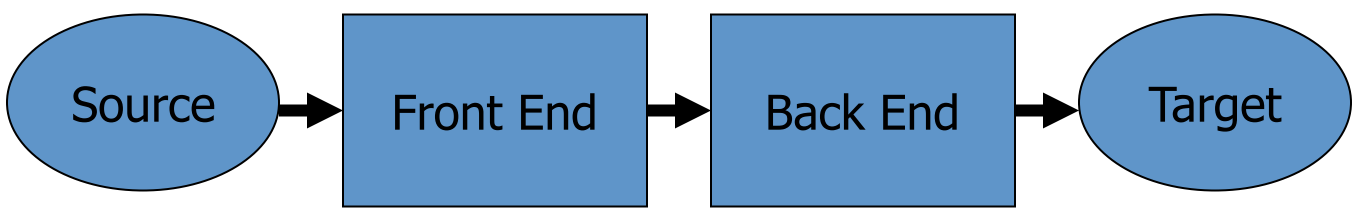
- How do turn this into something the computer can execute?

```
int nPos = 0;
int k = 0;
while (k < length) {
  if (a[k] > 0) {
    nPos++;
  }
}
```

- The computer only knows 1’s & 0’s
- Using a compiler (and/or an interpreter)
  - We’ll discuss the differences in a few slides

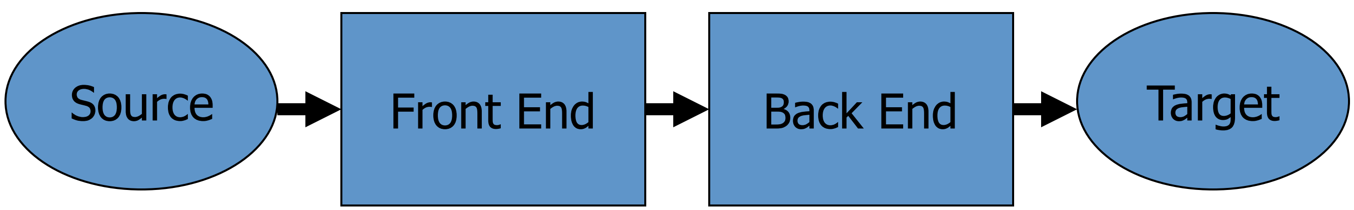
Structure of a Compiler

- At a high level, compilers have two pieces:
  - Front end: read source code
    - Parse the source, understand its structure
  - Back end: write executable
    - Generate equivalent target language program. May optimize (improve) code, but must not change behavior.



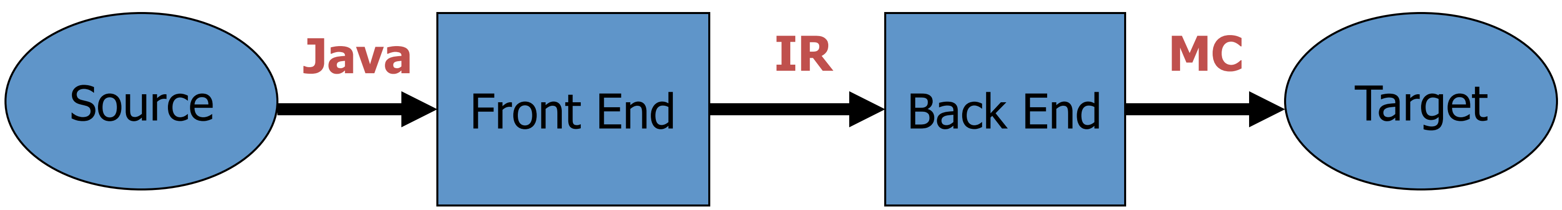
Compiler must…

- recognize legal programs (& complain about illegal ones)
- generate correct code
  - Programmer's favorite pastime is blaming their buggy code on "compiler bugs". ☺
- manage runtime storage of all variables/data
- agree with OS (loader) and linker on target format

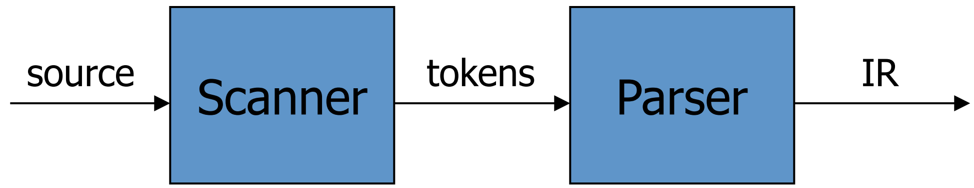


How does this happen?

- Phases communicate via Intermediate Representations, a.k.a., “IR”.
  - Front end maps source into IR
  - Back end maps IR to target machine code
  - Often multiple IRs produced by different phases of front/ back ends – higher level at first, lower level in later phases



Front End



- Usually split into two main parts
  - Scanner: Responsible for converting character stream to token stream: operation, variable, constant, etc.
    - Also: strips out white space, comments
  - Parser: Reads token stream; generates IR
    - (Semantics analysis can happen here, or immediately afterwards)
- Both of these can be generated automatically
  - Use a formal grammar to specify source language (e.g., Java)
  - Tools read the grammar and generate scanner & parser (e.g., lex and yacc for C, or JFlex and CUP for Java)

Scanner Output Example

- Input text

```
// Look, I wrote a comment!  I'm a good programmer!
if (x >= y) y = 42;
```

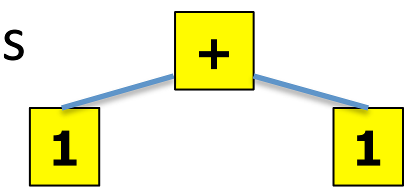
- Token Stream

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

  – Notes: tokens are atomic items, not character strings; comments & whitespace are *not* tokens (in most languages, ahem, FORTRAN)
    - Tokens *may* have associated data, e.g., a value or a variable name.

Parser Output (IR)

- Given token stream from scanner, parser must produce output that conveys meaning of program.
- Most common is an abstract syntax tree (“AST”)
  - Essential meaning of program without syntactic noise
  - Nodes are operations, children are operands
    - E.g., 1 + 1 – Parent: +, Child1: 1, Child2: 1



- Many different forms of IR used in compilers
  - Engineering tradeoffs have changed over time
  - Tradeoffs (and IRs) also can vary between different phases of compilation.

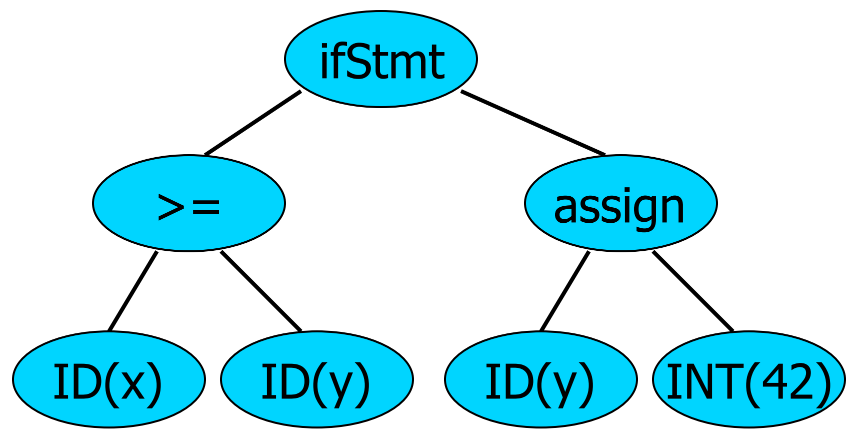
Parser Example

```
// Look, I wrote a comment!  I'm a good programmer!
if (x >= y) y = 42;
```

- Token Stream Input

IF LPAREN ID(x)
GEQ ID(y) RPAREN
ID(y) BECOMES
INT(42) SCOLON

- Abstract Syntax Tree



Static Semantic Analysis

- During and/or after parsing, checks that program is legal, and collects info for back end
  - Type checking
  - Check language requirements like proper declarations/initializations (e.g. Java locals), etc.
  - Collect other information used by back end analysis (e.g., scoping, aliasing restrictions)
- Key data structure: Symbol Table(s)
  - Maps names -> meaning/types/details

Back End

- Responsibilities
  - Translate IR into target machine code
  - Should produce "good" code
    - "good" = fast, compact, low power (pick some)
    - Optimization phases translate code into semantically equivalent but "better" code.
  - Should use machine resources effectively
    - Registers
    - Instructions
    - Memory hierarchy

Back End Structure

- Typically split into two major parts
  - “Optimization” – code improvements, e.g.,
    - Common subexpression elimination:

      `(x+y) * (x+y)` → `t = x + y; t * t`

    - Constant folding: `(1+2) * x` → `3 * x`
    - Optimization phases often interleaved with analysis phases to better understand program meaning/know what transformations preserve that meaning
  - Target Code Generation (machine specific)
    - Instruction selection & scheduling, register allocation

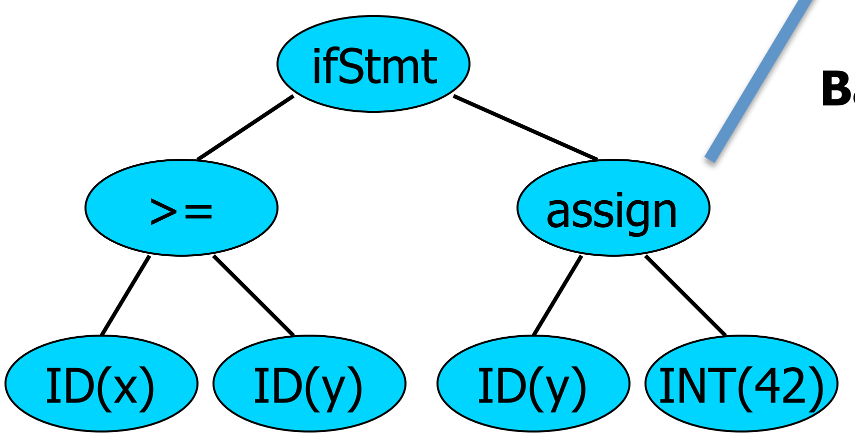
The Result

- Input

```
if (x >= y)
   y = 42;
```

Parser

- AST



Back End

- Output

```
   mov   eax,[ebp+16]
   cmp   eax,[ebp-8]
   jl    L17
   mov   [ebp-8],42
L17:
```

Interpreters & Compilers

- Programs can be compiled or interpreted (or in some cases both)
- Compiler
  - A program that translates a program from one language (the *source*) to another (the *target*)
  - In some cases the source and target can even be the **same**.
- Interpreter
  - A program that reads a source program and produces the results of executing that program on some input

Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and "understand" it: front-end analysis phase

w h i l e (k < l e n g t h) { <nl> <tab> i f (a [k] > 0
) <nl> <tab> <tab>{ n P o s + + ; } <nl> <tab> }

Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier or more efficient to execute
- Offline process
- Tradeoff: compile-time overhead (preprocessing) vs execution performance

Typically implemented with Compilers

- FORTRAN, C, C++, COBOL, other programming languages, (La)TeX, SQL (databases), VHDL (a hardware description language), many others
- Particularly appropriate if significant optimization wanted/needed

Interpreter

- Interpreter
  - Typically implemented with “execution engine” model
  - Program analysis interleaved with execution

```
running = true;
while (running) {
  analyze next statement;
  execute that statement;
}
```

  - Usually requires repeated analysis of individual statements (particularly in loops, functions)
    - But - hybrid approaches can avoid this …
  - But: immediate execution, good debugging/interaction, etc.

Often implemented with interpreters

- Javascript, PERL, Python, Ruby, awk, sed, shells (bash), Scheme/Lisp/ML, postscript/pdf, machine simulators
- Particularly efficient if interpreter overhead is low relative to execution cost of individual statements
  - But even if not (machine simulators), flexibility, immediacy, or portability may be worth it

Hybrid approaches

- Compiler generates byte code intermediate language, e.g., compile Java source to Java Virtual Machine .class files, then
- Interpret byte codes directly, or
- Compile some or all byte codes to native code
  - Variation: Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code
- Also widely use for Javascript, many functional languages (Haskell, ML, Ruby), C# and Microsoft Common Language Runtime, others

Agenda

- Introductions
- Administrivia
- What's a compiler, and how does it work (at a high level)?
- **Why study compilers?**
- A brief history of compilers

Why Study Compilers?

- Become a better programmer(!)
  - Insight into interaction between high-level language source, compilers, and hardware
  - Understanding of implementation techniques, how code maps to hardware
  - Better intuition about what your code does
  - Understanding how compilers optimize code helps you write code that is easier to optimize
    - And not waste time making optimization that the compiler would do as well or better.

Why Study Compilers?

- Compiler techniques are everywhere
  - Parsing (“little” languages, interpreters, XML)
  - Software tools (verifiers, checkers, …)
  - Database engines, query languages
  - Text processing
    - Tex/LaTex -> dvi -> Postscript -> pdf
  - Hardware: VHDL; model-checking tools
  - Mathematics (Mathematica, Matlab)

Why Study Compilers?

- Fascinating blend of theory and engineering
  - Lots of beautiful theory around compilers
  - But also interesting engineering challenges and tradeoffs, particularly in optimization
    - Ordering of optimization phases
    - What's good for some programs may not be good for others
  - Plus some very difficult problems (NP-hard or worse)
    - E.g., register allocation is equivalent to graph-coloring
    - Need to come up with good-enough approximations/ heuristics

Why Study Compilers?

- Draws ideas from many parts of CSE
  - AI: Greedy algorithms, heuristic search
  - Algorithms: graph algorithms, dynamic programming, approximation algorithms
  - Theory: Grammars, DFAs and PDAs, pattern matching, fixed-point algorithms
  - Systems: Interaction with OS, runtimes
  - Architecture: pipelines, instruction set use, memory hierarchy management, locality

Why Study Compilers?

- You might even write a compiler some day!
  - You ***will*** write parsers and interpreters for little languages, if not bigger things
    - Command languages, configuration files, XML, network protocols, …
  - If you like working with compilers, and are good at it, there are many jobs available:
    - Cray, Intel, Microsoft, AMD, and others all have their own compilers that are regularly updated.
    - Processor arms race is effectively a "*perpetual employment act*" for compiler writers.

Agenda

- Introductions
- Administrivia
- What's a compiler, and how does it work (at a high level)?
- Why study compilers?
- **A brief history of compilers**
  - Moved to next lecture (Wednesday)