



## *Overview of the Course*

# Critical Facts

---



Welcome to CISC 672 — *Advanced Compiler Construction*

*Topics in the design of programming language translators,  
including parsing, semantic analysis, error recovery, code  
generation, and optimization*

- Instructor: Dr. John Cavazos (cavazos@cis.udel.edu)
- Office Hours: Tues/Thurs 3PM to 4PM, Smith Hall 412
- Text: Engineering a Compiler  
by Keith Cooper and Linda Torzcan
- Web Site: <http://www.cis.udel.edu/~cavazos/CISC672>
  - Lab handouts, homework, slides, practice exams, ...
  - I will not have handouts in class; get them from the web

Lab data is on  
the web site



## Basis for Grading

• Exams			
→ Midterm	20%	}	This only adds up to 95%. Where is the other 5%?
→ Final	20%		
• Quizzes	10%		
• Projects			
→ Scanner	7%		
→ Parser	8%	}	Class participation!
→ Semantic Analyzer	15%		
→ Code Generation	15%		

Notice: Any student with a disability requiring accommodations in this class is encouraged to contact me after class or during office hours, and to contact UDel's Coordinator for Disabled Student Services.



## Basis for Grading

<ul style="list-style-type: none"><li>• Exams<ul style="list-style-type: none"><li>→ Midterm</li><li>→ Final</li></ul></li></ul>	<ul style="list-style-type: none"><li>♦ Closed-notes, closed-book</li><li>♦ Old exam on web site as an example</li></ul>
<ul style="list-style-type: none"><li>• Quizzes</li></ul>	<ul style="list-style-type: none"><li>♦ Reinforce concepts</li><li>♦ Number of quizzes <i>t.b.d.</i></li></ul>
<ul style="list-style-type: none"><li>• Projects<ul style="list-style-type: none"><li>→ Parser (&amp; scanner)</li><li>→ Semantic Analyzer</li><li>→ Code Generation</li></ul></li></ul>	<ul style="list-style-type: none"><li>♦ Parser lab might be a team lab</li><li>♦ High ratio of thought to programming</li><li>♦ Will build a compiler for a language called COOL (Java)</li></ul>

# Rough Syllabus

---



- Overview § 1
- Scanning § 2
- Parsing § 3
- Context Sensitive Analysis § 4
- Inner Workings of Compiled Code § 6, 7
- Introduction to Optimization § 8
- Instruction Selection § 11
- Instruction Scheduling § 12
- Register Allocation § 13
- More Optimization (*time permitting*)

# Class-taking technique for CISC 672

---



- I will use projected material extensively
  - I will moderate my speed, *you* sometimes need to say "STOP"
- You should read the book
  - Not all material will be covered in class
  - Book complements the lectures
- You are responsible for material from class
  - The tests will cover both lecture and reading
  - I will probably hint at good test questions in class
- CISC 672 is not a programming course
  - Projects are graded on functionality, documentation, and lab reports more than style (*results matter*)
- It will take me time to learn your names (*please remind me*)

# Compilers

---

- What is a **compiler**?



# Compilers

---



- What is a **compiler**?
  - A program that translates a program in one language into a program in another language
  - The compiler should improve the program, *in some way*
- What is an **interpreter**?



# Compilers

---



- What is a **compiler**?
  - A program that translates a program in one language into a program in another language
  - The compiler should improve the program, *in some way*
- What is an **interpreter**?
  - A program that reads a program and produces the results of executing that program

# Compilers

---



- What is a **compiler**?
  - A program that translates a program in one language into a program in another language
  - The compiler should improve the program, *in some way*
- What is an **interpreter**?
  - A program that reads a program and produces the results of executing that program
- C is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
  - which can then interpreted
  - Or a hybrid strategy is used
    - Just-in-time compilation

# Taking a Broader View

---



- Compiler Technology
  - Offline
    - Typically C, C++, Fortran
  - Online
    - Typically Java, C##
  - **Goals:** improved performance and language usability
    - Making it practical to use the full power of the language
  - **Trade-off:** preprocessing time versus execution time (or space)
  - **Rule:** performance of both compiler and application must be acceptable to the end user

# Why Study Compilation?

---



- Compilers are important system software components
  - They are intimately interconnected with architecture, systems, programming methodology, and language design
- Compilers include many applications of theory to practice
  - Scanning, parsing, static analysis, instruction selection
- Many practical applications have embedded languages
  - Commands, macros, formatting tags ...
- Many applications have input formats that look like languages,
  - Matlab, Mathematica
- Writing a compiler exposes practical algorithmic & engineering issues
  - Approximating hard problems; efficiency & scalability

# Intrinsic interest



- Compiler construction involves ideas from many different parts of computer science

<i>Artificial intelligence</i>	Greedy algorithms Heuristic search techniques
<i>Algorithms</i>	Graph algorithms, Dynamic programming
<i>Theory</i>	DFAs & PDAs, pattern matching Fixed-point algorithms
<i>Systems</i>	Allocation & naming, Synchronization, locality
<i>Architecture</i>	Pipeline & hierarchy management Instruction set use

# Intrinsic merit

---



- Compiler construction poses challenging and interesting problems:
  - Compilers must do a lot but also **run fast**
  - Compilers have responsibility for **run-time performance**
  - Compilers are responsible for making it acceptable to use the **full power** of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more **complex machines**
  - Compilers must hide that complexity from the programmer
  - Success requires mastery of complex interactions

Aren't compilers a solved problem?

---



"Optimization for scalar machines is a problem that was solved ten years ago."

David Kuck, Fall 1990

## Aren't compilers a solved problem?

---



"Optimization for scalar machines is a problem that was solved ten years ago."

David Kuck, Fall 1990

- Architectures keep changing
- Languages keep changing
- Applications keep changing
- When to compile keeps changing



# About the instructor

---



- My own research
  - Applying machine learning to solve hard systems problems
  - Compiling for advanced microprocessor systems
  - Interplay between static and dynamic compilation
  - Optimization for embedded systems (*space, power, speed*)
  - Interprocedural analysis and optimization
  - Nitty-gritty things that happen in compiler back ends
  - Distributing compiled code in a heterogeneous environment
  - Rethinking the fundamental structure of optimizing compilers
- Thus, my interests lie in
  - Building "Intelligent" Compilers
  - Quality of generated code(smaller, more efficient, faster)
  - Interplay between compiler and architecture
  - Static analysis to discern program behavior
  - Run-time performance analysis

## Next class

---



- The view from 35,000 feet
  - How a compiler works
  - What I think is important
  - What is hard and what is easy