15-411 Compilers
Who are we?

- Andre Platzer
  - Out of town the first week
  - GHC 9103

- TAs
  - Alex Crichton, senior in CS and ECE
  - Ian Gillis, senior in CS
Logistics

- symbolaris.com/course/compiler12.html
  - symbolaris.com -> Teaching -> Compiler 12
- autolab.cs.cmu.edu/15411-f12
- Lectures
  - Tues/Thurs 1:30-2:50pm
  - GHC 4211
- Recitations - none!
- Office Hours
  - More coming soon...
Contact us

- 15411@symbolaris.com
  - Course staff
- Individually
  - Andre - aplatzer@cs.cmu.edu
  - Alex - acrichto@andrew.cmu.edu
  - Ian - igillis@andrew.cmu.edu
- Office Hours
Waitlisted?

● Long waitlist
  ○ Room may become available!
● Beware of partnering
  ○ If admitted but no singles left, you must solo
● Talk to me after lecture
Course Overview

- No exams
  - Not even a final!
- 5 homeworks
- 6 Labs
  - Required tests for each lab
- Paper at the end
Textbook(s)

- Modern Compiler Implementations in ML
  - Andrew W. Appel
  - Optional

- Compiler Construction
  - William M. Waite and Gerhard Goos
  - Optional

- Supplement lecture
  - Do not replace it
Homeworks

● One before each lab is due
  ○ About a week to work on each one
● Submitted through autolab individually
● Must be your own work
● 30% of the final grade (300 points total)
  ○ Each homework is 6% of your grade
● Due at the beginning of lecture
  ○ Can turn two homeworks in late
  ○ Only up to the next lecture
  ○ Excludes Thanksgiving
Labs - Overview

- Also submitted through autolab
- May be done in pairs (same pair for all labs)
  - Must be entirely team's work
  - Acknowledge outside sources in readme
- 70% of final grade (700 points total)
- 6 labs
  - First 5 are 100 points each
  - Last is 200
• Cumulatively build a compiler for C0
  ○ Expressions
  ○ Control flow
  ○ Functions
  ○ Structs and arrays
  ○ Memory safety and optimizations
  ○ Choose your own adventure

• Each lab is a subset of C0
  ○ Also superset of previous lab
Labs - Language

● Can write compiler in language of choice
● Starter code (initial parser/layout)
  ○ SML
  ○ Haskell
  ○ Scala
  ○ Java
● Grading process
  ○ make
  ○ ./bin/l{1,2,3,4,5,6}c
Labs - Layout

- Each lab has two parts
- Part 1: submit 10 tests
  - 20% of the lab grade
  - Based on number of tests submitted
  - Can be as creative as you like
- Part 2: submit a compiler
  - 80% of the lab grade
  - Based on number of tests passed
  - Tested against everyone's tests
    - And previous labs
    - And last year's
    - And the year before that
Labs - Tests

- Very good way to test compilers
  - Aren't comprehensive, however
  - Purpose is to find individual bugs
- You are graded on everyone's tests
- `assert(1 + 1 == 2)`
Labs - Submission

- SVN repositories set up
- Work is submitted through SVN into autolab
  - Only most recent submission is relevant
- We publish updates to tests and runtime
  - You just run 'svn update'
- Only one autolab submission is necessary per team for labs
  - We don't grade SVN, so submit updates to autolab!
Labs - Timing

- Two weeks for each lab
  - Tests due at end of first week (11:59)
  - Compiler due at end of second (11:59)
- No late days for tests
- 6 late days for compiler
  - At most two per lab
Labs - Partners

• Can do labs alone
• Can also do with a partner
  ○ Should remain the same for all labs
• Email 15411@symbolaris.com with partner
  ○ We will then assign you a team name
Labs - Partners

- If partnering, choose wisely
  - Must work as a team to be effective
  - Cannot let the other "do all the work"

- Trouble arises
  - Email 15411@symbolaris.com before too late
  - Day before lab is due is too late
  - Beginning of second lab is not too late
Labs - Warnings

● Labs are **hard** and take time
● Don't start the compiler only after submitting tests
● Errors in one lab carry over to the next
  ○ Each lab still runs previous tests
● Do not take labs lightly, plan accordingly
  ○ This class will consume much time
● 15-411 is by no means easy
  ○ Compilers take **a lot** of work
Labs - Suggestions

- **Start early**
  - Fixing tests takes a long time
- **If submitted compiler has errors, fix quickly**
  - Errors for lab 1 must be fixed for lab 2!
- **Schedule with partner**
  - Specifically set aside time for 15-411
- **Talk to us!**
  - Talk about design plans
  - Especially if soloing
  - Office hours or email
- **Remember that this is exciting!**
Labs - My suggestions

- Do not cram entire compiler into one week
- Compiler passes own tests when tests due
- Get to know the driver well
  - You will be running this many many times
  - Ask us if you want it do have feature X
- Write difficult tests
  - Forces you to think
- Submit early to autolab
  - Avoid the rush
After 6th lab, a paper is required
- Technical paper demonstrating what you learned
  - What design decisions did you make?
  - What design decisions were good?
  - Which ones ended badly?
  - Were certain tests good or tricky?
- More details when time comes
Questions?

- Waitlist
- Course outline
- Homework
- Labs
  - Partners
- Paper
Writing a Compiler
Course Goals

● Understand how compilers work
  ○ General structure of compilers
  ○ Influence of target/source language on design
  ○ Restrictions of hardware

● Gain experience with a complex project
  ○ Both maintain it and work with others

● Develop in a modular fashion
  ○ Each lab builds on the next
What is a compiler?

- Translator from one language to another
  - Might have a few changes in the middle
- Adheres to 5 principles
  - Correctness
  - Efficiency
  - Interoperability
  - Usability
  - Retargetability
Correctness
- How useful is an incorrect compiler?
- What if it were extremely fast?

How do you know?
- Language specification
- Formal proof
- Tests, lots of tests
• **What to test for correctness?**
  - 1 + 1 == 2
  - 1 + 1 != 1
  - *a == 3
  - *NULL is a segv
  - while (1); loops forever

• **Language design**
  - Can make correctness a lot easier
  - Or harder
  - C0 is much better specified than C
Compiler Principles - 2

- **Efficiency**
  - Generated code is fast
  - Compiling process is also fast
- **Cannot forsake correctness**
  - "But I got the wrong answer really fast!"
Compiler Principles - 3

- Interoperability
  - Most binaries are not static
  - Run with code from other compilers

- Interface, or an ABI
  - C0 uses the C ABI
  - x86 is different than x86-64
  - arm is very different
Compiler Principles - 4

- Usability
- Error messages
  - Error.
  - Error in file foo.c
  - Error at foo.c:3
  - Error at foo.c:3:5
  - Type Error at foo.c:3:5
  - Type Error at foo.c:3:5, did you mean ...?
- Not formally tested in this class
  - You're still writing code!
Retargetability
  ○ Multiple sources?
  ○ Multiple targets?

We will not emphasize this
  ○ Does not mean you should disregard it
Designing a Compiler

- Correctness
- Efficiency
- Interoperability
- Usability
- Retargetability
Designing a Compiler

Source

Compiler

Executable
Designing a Compiler

Source  →  Executable

C to x86
Designing a Compiler

Source → C to x86-64 → Executable

C to x86-64

C to x86
Designing a Compiler

Source -> Need common language -> Executable
Designing a Compiler

Source → Intermediate Representation → Executable
Designing a Compiler

Source  \(\rightarrow\) Intermediate Representation  \(\rightarrow\) Executable

Java  \(\rightarrow\) Intermediate Representation  \(\rightarrow\) x86

C  \(\rightarrow\) Intermediate Representation  \(\rightarrow\) x86-64
Designing a Compiler

Source → Intermediate Representation → Executable

C0 → Intermediate Representation → x86-64
Designing a Compiler

Source → Intermediate Representation → Executable

C0 → Intermediate Representation → x86-64

What is this line?
Designing a Compiler

C0 Source → Lex → Intermediate Representation → x86-64 → Executable
Designing a Compiler

C0 Source

Lex

tokens

Intermediate Representation

Executable

x86-64
Designing a Compiler

C0 Source

Lex

Parse

tokens

Intermediate Representation

Executable

x86-64
Designing a Compiler

C0 Source → Lex → Parse → Intermediate Representation → x86-64 → Executable
Designing a Compiler

C0 Source

Lex

Parse

Semantic Analysis

Intermediate Representation

Executable

x86-64
Designing a Compiler

- C0 Source
- Executable

Flow:
- Lex
  - tokens
- Parse
  - AST
- Semantic Analysis
  - AST attributed
- Intermediate Representation
  - x86-64
Designing a Compiler

C0 Source

Lex
 tokens

Parse

AST

Semantic Analysis

Intermediate Representation

Translate

AST attributed

x86-64

Executable
Designing a Compiler

C0 Source → Lex → Parse → Semantic Analysis → Intermediate Representation → Translate → x86-64 → Executable

- Lex: tokens
- Parse: AST
- Semantic Analysis: AST attributed
- Translate: Intermediate Representation → x86-64

How about this?
Designing a Compiler

C0 Source

Lex

Parse

Semantic Analysis

Intermediate Representation

Translate

Optimize

tokens

AST

AST attributed

Executable
Designing a Compiler

C0 Source → Lex → Parse → Semantic Analysis → Intermediate Representation

Intermediate Representation → Translate → AST attributed → IR → Optimize → IR

Executable
Designing a Compiler

C0 Source

Lex

Parse

tokens

AST

Semantic Analysis

Intermediate Representation

Translate

Optimize

AST attributed

IR

Reg. Alloc & Codegen

Executable
Designing a Compiler

C0 Source

Lex

Parse

Semantic Analysis

tokens

AST

Intermediate Representation

Translate

Optimize

AST attributed

IR

Reg. Alloc & Codegen

Executable

ASM
Designing a Compiler

- **C0 Source**
- **Lex**
  - tokens
- **Parse**
  - AST
- **Semantic Analysis**
- **Intermediate Representation**
- **Translate**
- **Optimize**
  - IR
  - AST attributed
- **ASM+Link**
- **Reg. Alloc & Codegen**
- **Executable**

- **ASM**
The Compiler 'W'

C0 Source

Lex

Parse

Semantic Analysis

Intermediate Representation

Translate

Optimize

ASM+Link

Reg. Alloc & Codegen

Executable

tokens

AST

AST attributed

IR
The Compiler 'W'

- Easy to re-target all source languages
  - Just add a new back end from the IR
- Easy to optimize all sources
  - Just add a pass to the IR
- Easy to add a new source language
  - Just add a new front end into the IR
The Compiler 'W'

- Variants
  - Split register allocation and code generation
  - Another optimize pass in codegen
  - Reorder passes in backend
What to compile?

- **Simple**
  - Goal is to learn how compilers work, not feature X

- **Safe**
  - Semantics should be well defined
  - Enables many optimizations
What to compile?

● What should happen here?

```c
int foo(int a, int b, int *c) {
    if (a / b == 1 || *c == 3)
        return 3;
    return 4;
}
```
What to compile?

- C
  - Simple
  - Unsafe
- Java
  - Not simple
  - Safe(er)
- C0?
What to compile?

- C0 is a safe variant of C
  - Developed at CMU by Frank Pfenning and others
- All C0 programs are deterministic given same input
- Differences
  - No pointer arithmetic
  - No casting
  - No stack allocated structs
  - Hard(er) to shoot yourself in the foot
  - Can enable memory safety
## What to target?

<table>
<thead>
<tr>
<th></th>
<th>ISA</th>
<th>Runnable?</th>
<th>Oddities?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>CISC</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>x86-64</td>
<td>CISC</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>arm, mips</td>
<td>RISC</td>
<td>simulators</td>
<td>✓</td>
</tr>
</tbody>
</table>
What to target?

- We have chosen x86-64
  - You generate assembly, gcc links it
- Lots of fun caveats to deal with still
Questions?

- Compiler Principles
- The compiler 'W'
  - Lexing/Parsing
  - Semantic analysis
  - IR/optimizations
  - Codegen/register allocation
- C0
  - Well-defined semantics
  - "safer C"
Remember...

- symbolaris.com
- Choose a partner
  - Email 15411@symbolaris.com
- Labs are cumulative
  - Don't fall behind
- Think about language you'll write in