Welcome!

Simone Campanoni
simone.campanoni@northwestern.edu
Who we are

Simone Campanoni

Ben Sevi Geduld
BenGeduld2023@u.northwestern.edu
Outline

• Structure of the course

• Compilers

• Compiler IRs
CC in a nutshell

- **CS 322**: main blocks of modern compilers
  - Satisfy the system breadth for CS major
- **When**: Tuesday/Thursday 5pm - 6:20pm
- **Where**: here 😊
- **Office hours**:
  - Ben: Friday 12:30pm – 2:30pm via Zoom (link on Canvas)
  - Simone: Tuesday Noon – 1:00pm via Zoom (link on Canvas)
- CC is on Canvas
  - Materials/Assignments/Grades on Canvas
CC in a nutshell

CC 2023

Important links:
- Syllabus
- Lectures and files
- Tutorials
- Plazza: semester login
- Zoom:
- Lectures
- Simone’s office hours
- Ben’s office hours

The compiler is the programmer's primary tool. Understanding the compiler is therefore critical for programmers, even if they never build one. Furthermore, many design techniques that emerged in the context of compilers are useful for a range of other application areas. This course introduces students to the essential elements of building a compiler: parsing, context-sensitive property checking, code linearization, register allocation, etc. To take this course, students are expected to already understand how programming languages behave, to a fairly detailed degree. The material in the course builds on that knowledge via a series of semantics-preserving transformations that start with a fairly high-level programming language and culminate in machine code.

Previous years Hall of Fame

• CC is on Canvas
  • Materials/Assignments/Grades on Canvas

Tutorials

Next are the tutorials offered during Ben’s office hours.

Week 0: C++ and Makefile

Week 1: Debugging with gdb

Week 2: Visitor pattern
CC in a nutshell

CC 2023

Important links:
- Syllabus
- Lectures and files
- Tutorials
- Plazza: simone <link>, ben <link>
- Zoom:
  - Lectures <link>
  - Simone’s office hours <link>
  - Ben’s office hours <link>

The compiler is the programmer’s primary tool. Understanding the compiler is therefore critical for programmers, even if they never build one. Furthermore, many design techniques that emerged in the context of compilers are useful for a range of other application areas. This course introduces students to the essential elements of building a compiler: parsing, context-sensitive property checking, code linearization, register allocation, etc. To take this course, students are expected to already understand how programming languages behave, to a fairly detailed degree. The material in the course builds on that knowledge via a series of semantics-preserving transformations that start with a fairly high-level programming language and culminate in machine code.

Previous years Hall of Fame

• CC is on Canvas
  • Materials/Assignments/Grades on Canvas

Lectures

Next are the lectures of this class with the link to the related videos.

Week 0:
- Welcome (slides), the CC framework (slides, code) (video)
- The L1 language (slides) (video)

Week 1:
- Translating L1 code to assembly (slides), parsing (slides, code) (video)
- The L2 language (slides), liveness analysis (slides) (video)

Week 2:
- Second part of the liveness analysis (same slides, code, video), Panels about H0 (the L1 compiler)
- Interference graph (slides), Spilling (slides), Graph coloring (slides) (video)
CC in a nutshell

• **CS 322**: main blocks of modern compilers
  - Satisfy the system breadth for CS major
• **When**: Tuesday/Thursday 5pm - 6:20pm
• **Where**: here 😊
• **Office hours**:
  - Tommy: Friday 12:30pm – 2:30pm via Zoom (link on Canvas)
  - Simone: Tuesday Noon – 1:00pm via Zoom (link on Canvas)
• CC is on Canvas
  - Materials/Assignments/Grades on Canvas
  - You’ll upload your assignments on Canvas
• CC is part of the sequence of compiler classes at Northwestern University
  - Other compiler-heavy classes: CS 323 and CS 397/497
  - My teaching philosophy (e.g., learn by building): [link](#)
CC materials

- Slides
- Books
- Papers and library documentation for further information
CC slides

• You can find last year slides from the class website

• We improve slides every year
  • Based on problems we observe the year before
  • So: we will ask your feedbacks at the end
  • Our goal: maximize how much you learn in 10 weeks

• We will upload to Canvas the new version of the slides after each class
CC slides

- Organized in topics that follow the compilation steps of modern compilers

- We will cover one topic per week
The CC structure

- Needs to be done within 48 hours
- Needs to be done within 6 days
Output of your work

Homework after homework

you’ll **build**

your **own** compiler

from **scratch**
Homework

Each assignment is composed by:
1. A set of programs written in the source programming language (PL) considered
   (program assignment)

2. A compiler that translates the source PL to the destination PL
   (compiler assignment)

Source code (C like)

Homework N

... ...

Homework 2

Homework 1

Target code (x86_64)
Homework

• Program assignment (when I’ll mention in the class)
  • You need to write Y programs
    in the source language of that assignment
  Deadline: 2 days

• Compiler assignment
  • Day X: you have the assignment
  • Deadline: 6 days after
  • Your compiler has to pass all tests included in the framework

• Late submission: you cannot be selected as a panelist (see later)
Evaluation of your work

For each assignment, you get 1 point iff:
1. Your tests are correct
2. You pass all tests using your current and prior work and
3. I will not find a bug in your implementation (I will manually inspect your code)

Some assignments can be passed either:
- **Properly**: by implementing the algorithm discussed in class
- **Naively**: you will not get the point, but you can access the next assignment (*do not submit naïve solutions*)
The CC competition

• At the end, there will be a competition between your compilers

• The team that designed the best compiler
  • Get an A automatically
    (no matter how many points they have)
  • Their names go to the “hall of fame” of this class
The CC grading

• 8 assignments (8 points)
  • If not submitted on time, you cannot be selected for being a panelist

• +1 point if you submit the last assignment on time for the final competition

• 3 panelist experiences (3 points)
  1. Manager
  2. Manager supports
  3. Secretary

<table>
<thead>
<tr>
<th>Grade</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&gt;= 11</td>
</tr>
<tr>
<td>A -</td>
<td>10</td>
</tr>
<tr>
<td>B +</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
</tr>
<tr>
<td>B-</td>
<td>7</td>
</tr>
<tr>
<td>C+</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>C-</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>0 – 2</td>
</tr>
</tbody>
</table>
Rules for homework

• You are encouraged (but not required) to work in pairs
  • Pair programming is not team programming
  • Declare your pair by the next lecture (send message via email to TA)
    • After this deadline, you can only split (no new/merging pairs is allowed)
    • If you don’t declare your pair, then you’ll work alone

• No copying of code is allowed between pairs

• Tool, infrastructure help is allowed between pairs
  • First try it on your own
    (google and tool documentation are your friends)

• Avoid plagiarism
  www.northwestern.edu/provost/policies/academic-integrity/how-to-avoid-plagiarism.html

• If you don’t know, please ask
Summary

• My duties
  • Teach you the blocks of a compiler
  • And how to implement them

• Your duties
  • Learn all compiler blocks presented in class
  • Implement some of them (the most important ones)
    • Write code in C++
    • Test your code
    • Then, think much harder about how to actually test your code
    • Be ready for being in a panel when asked (the day before)
Structure & flexibility

• CC is structured w/ topics

• Best way to learn is to be excited about a topic

• Interested in something? **Speak**
  I’ll do my best to include your topic on the fly
Week 1

Today
- Structure of CS 322
- Intro to compilers
- The CC framework

Thursday
- The L1 language
Outline

• Structure of the course

• Compilers

• Compiler IRs
The role of compilers
The role of compilers

If there is no coffee, if I still have work to do, I’ll keep working, I’ll go to the coffee shop

If there is no coffee{
  if I still have work to do{
    I’ll keep working;
  }
  I’ll go to the coffee shop;
}
If there is no coffee{
  if I still have work to do{
    I’ll keep working;
  }
  I’ll go to the coffee shop;
}

Interpreter

👍 Relatively simple to build and maintain
👍 Great for prototyping
👎 It quickly becomes too slow
👎 It often consumes a lot of memory
The role of compilers

If there is no coffee{
if I still have work to do{
   I’ll keep working;
}
I’ll go to the coffee shop;
}

Compilers

👍 Great performance of the generated binaries that get optimized
👍 Important energy savings are unlocked for the generated binaries
👎 Compilation/optimization time can increase significantly when large programs are compiled
👎 Compilers are large and complex codebases

001010101111001010101010010010101010101001011010
Compiler goals

• Goal #1: correctness
• Goal #2: maximize performance and/or energy consumptions
• Goal #3: easy to be extended to
  • New architecture features (e.g., x86_64, +AVX, +TSX)
  • Evolutions of the targeted PL (e.g., C++99, C++11, C++14, C++17)
  • New architecture / ISA (e.g., RISC V)
  • New PL (e.g., Rust, Swift)
• Goal #4: Minimize maintainability costs
  • Write DRY code (Don’t Repeat Yourself)
  • Exploit code generation
Goals of your compilers in this class

• Goal #1: correctness
• Goal #2: maximize performance and/or energy consumptions
• Goal #3: easy to be extended to
  • New architecture features (e.g., x86_64, +AVX, +TSX)
  • Evolutions of the targeted PL (e.g., C++99, C++11, C++14, C++17)
  • New architecture / ISA (e.g., RISC V)
  • New PL (e.g., Rust, Swift)
• Goal #4: Minimize maintainability costs
  • Write DRY code (Don’t Repeat Yourself)
  • Exploit code generation
Structure of a compiler

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

```c
int main (){
    printf(“Hello World!
”);
    return 0;
}
```
Structure of a compiler

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

Example tokens:

```
int main ...
```

Example tokens:

```
INT SPACE STRING SPACE ...
```

Example tokens:

```
Function signature
Return type
Function name

INT
STRING
```
Structure of a compiler

Syntactic & semantic analysis

- AST

IR code generation

- IR

Function signature

- Return type
  - INT
- Function name
  - STRING

myVarX = 40
myVarY = myVarX + 2
Structure of a compiler

Character stream (Source code)

Front-end

Middle-end

Back-end

Machine code

**EECS 322: Compiler Construction**

myVarX = 40
myVarY = myVarX + 2

**EECS 323: Code analysis and transformation**

myVarY = 42

**EECS 322: Compiler Construction**

010101110101010101
Outline

• Structure of the course

• Compilers

• Compiler IRs
Example of LLVM IR

define i64 @f (i64 %p0) {
    entry:
      %myVar1 = add i64 %p0, 1
      ret i64 %myVar1
}
Another example of IR

define int64 @f (int64 %p0) {
    :entry
    int64 %myVar1
    %myVar1 <- %p0 + 1
    return %myVar1
}
Multiple IRs used together

Programming language

Translation

IR1

Translation

IR2

Translation

Machine code

38
IRs are languages

Source code

Translation 0

... 

Translation N - 1

L1

Translation N

Target code

• A compiler is a sequence of passes

• Each pass translates from a source language to a target language

• Source and target languages can be the same (transformations in the middle end)

• Some languages have the support to be written/read into/from files
In this class

Source code

• The final compiler is built as a sequence of internal compilers

• Each internal compiler translates from a source language to a target language

• Source and target languages are always different

• All languages are written/read into/from files

• Each homework is a standalone compiler
In this class

• All compilers you will build can assume the program given as input is correct
  • No need to check program’s correctness
  • Production compilers first check program’s correctness, then they do the translations/optimizations/code generations

• When you write a program in a given language, it is your job to guarantee the correctness of the program you have written

• When a compiler generates the code in its target language, it is the compiler responsibility to generate correct code in the target language (while assuming the correctness of the code written in the source language given as input)
Let’s build our first compiler
The recipe of a disaster

1. Let’s translate independently a statement of the source program to a sequence of IR instructions

2. Let’s translate independently an IR instruction to a sequence of machine code instructions
The **good** and the **bad** compiler

```
int main (int argc, char *argv[]){
    return argc + 1;}
```

**Naïve compiler**

```
push  %rbp
mov   %rsp,%rbp
movl  $0x0,-0x4(%rbp)
mov   %edi,-0x8(%rbp)
mov   %rsi,-0x10(%rbp)
mov   -0x8(%rbp),%edi
add   $0x1,%edi
mov   %edi,%eax
pop   %rbp
retq
```

**clang**

```
lea  0x1(%rdi), %eax
retq
```

- Would you use a new PL if the resulting code is 100x slower compared to a C++ version?
- Would you use a CPU if your code is 100x slower compared to running it on an Intel CPU?
Conclusion

• Compilers translate a source language to a destination language
  • Front-end -> IR -> Middle-end -> IR -> back-end

• They help developers to be productive (enabling new PLs and abstractions)

• They help systems to run faster (enabling new resources of new CPUs)

• Correctness, efficiency (generated code and compiler itself), maintainability, extensibility are all aspects to consider when designing a compiler
Always have faith in your ability

Success will come your way eventually

Best of luck!