Welcome!

Simone Campanoni
simonec@eecs.northwestern.edu
Who we are

Simone Campanoni
simonec@eecs.northwestern.edu

Enrico Deiana
enricodeiana2020@u.northwestern.edu
Outline

• Structure of the course

• Compilers

• Compiler evolution
CC in a nutshell

• EECS 322: main blocks of modern compilers
• Satisfy the system breadth and depth for CS major
• Satisfy the project requirement too
• When: Monday/Wednesday 11:00am – 12:20pm
• Where: Frances Searle Building 2370

• Simone’s office hours: Friday 3:00pm – 5:00pm in 3512@Mudd
• Enrico’s office hours: Tuesday 4:00pm – 5:00pm in 3532@Mudd

• CC is on Canvas
  • Materials/Assignments/Grades on Canvas
  • You’ll upload your assignments on Canvas
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 will observe during the next 10 weeks
  • as well as your feedbacks we will ask you 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
The CC structure

Today

Week

Monday

Wednesday Homework

Topic & homework

• Needs to be done before next Wed.
Output of your work

Homework after homework

you’ll **build**

your *own* compiler

from **scratch**

Source code (e.g., C)

Homework N

... 

Homework 2

Homework 1

Target code (e.g., x86_64)
Evaluation of your work

For each assignment, you get 1 point iff:
1. You pass all tests using your current and prior work and
2. 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

Source code (e.g., C)

Homework N

... 

Homework 2

Homework 1

Target code (e.g., x86_64)
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

- 4 panelist experiences (4 points)

1. Manager
2. Second manager
3. Secretary

<table>
<thead>
<tr>
<th>Grade</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&gt;= 12</td>
</tr>
<tr>
<td>A -</td>
<td>10 - 11</td>
</tr>
<tr>
<td>B +</td>
<td>8 - 9</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>0 - 4</td>
</tr>
</tbody>
</table>
Rules for homework

• You are encouraged (but not required) to work in pairs
  • Pair programming is *not* team programming
• 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](http://www.northwestern.edu/provost/policies/academic-integrity/how-to-avoid-plagiarism.html)

• If you don’t know, please ask: simonec@eecs.northwestern.edu
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 a few 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
- Welcome/Structure
- Intro to compilers

Wednesday
Generating assembly

F.E. → M.E. → B.E.
Outline

• Structure of the course

• Compilers

• Compiler evolution
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.
Compiler goals

• Goal #1: correctness
• Goal #2: correctness
• Goal #3: maximize performance and/or energy consumptions
• Goal #4: 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)
• Minimize maintainability costs
  • Write DRY code (Don’t Repeat Yourself)
• Exploit code generation
Goals of your compiler

• Goal #1: correctness
• Goal #2: correctness
• Goal #3: maximize performance and/or energy consumptions
• Goal #4: 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 (e.g., RISC V)
  • New PL (e.g., Rust, Swift)
• 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

int main(){
    printf("Hello World!\n");
    return 0;
}
Structure of a compiler

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

```
int main ...
```

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

```
Function signature

Return type

INT

Function name

STRING
```
Structure of a compiler

Syntactic & semantic analysis

AST

IR code generation

Function signature

Return type

Function name

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...
Multiple IRs

- Abstract Syntax Tree

- Register-based representation (three-address code)
  \[ R1 = R2 + R3 \]

- Stack-based representation
  push 5; push 3; add; pop ;

IR needs to be easy
1) to be generated
2) to translate into machine code
3) to transform/optimize
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

L1

Static compiler

IR1

Dynamic compiler FE

IR2

Dynamic compiler BE

Machine code
Multiple IRs used together

Java

Java compiler

Java bytecode

Java VM FE

IR2

Java VM BE

Machine code
IRs are languages

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

Translation 8

... 

Homework 2

L1

Homework 1

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)

• All languages are written/read into/from files
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

```c
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 -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?
Outline

• Structure of the course

• Compilers

• Compiler evolutions
Evolution of compilers (hardware point of view)

• Simple hardware (few resources), simple compilers
Evolution of compilers (hardware point of view)

• Simple hardware (few resources), simple compilers
Evolution of compilers (hardware point of view)

- Simple hardware (few resources), simple compilers

<table>
<thead>
<tr>
<th>Speed</th>
<th>1ns</th>
<th>10ns</th>
<th>100ns</th>
<th>10ms</th>
<th>10sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>B</td>
<td>KB</td>
<td>MB</td>
<td>GB</td>
<td>TB</td>
</tr>
</tbody>
</table>

Size vs. Latency diagram
Evolution of compilers (hardware point of view)

• Simple hardware (few resources), simple compilers
Evolution of compilers (hardware point of view)

- Simple hardware (few resources), simple compilers
- More hardware resources available to compilers
- Opportunities to improve programs
- More complex compilers
- Execution model mismatch between source code and hardware
- Challenges for compilers

Compilers are considered in the processor-design stage!
Evolution of compilers (PL point of view)

- First electronic computers appeared in the ’40s
- They were programmed in machine language

- Low level operations only
  - Move data from one location to another
  - Add the contexts of two registers
  - Compare two values

- **Programming**: slow, tedious, and error prone
Evolution of compilers (PL point of view)

• Low level programming language, simple compilers
  • Not very productive

• More abstraction in programming language, more work for compilers to reduce their performance overhead
  • Macros -> Fortran, Cobol, Lisp -> C, C++, Java, C#, Python, PHP, SQL, ...

• Compilers enable new programming languages
Conclusion

• Compilers translate a source language to a destination language
• 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