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 IRs
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: Tuesday/Thursday 5pm - 6:20pm
• Where: here 😊

• Simone’s office hours: Friday 5:00pm – 7:00pm in 3512@Mudd
• Enrico’s office hours: Monday 8:00pm – 9:00pm in 3536@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 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
The CC structure

Topic & homework

Today

Week

Tuesday

Thursday

Homework

• Needs to be done before next Thursday
Output of your work

Homework after homework

you’ll **build**

your **own** compiler

from **scratch**

Source code (C like)

Homework N

... 

Homework 2

Homework 1

Target code (x86_64)
Assignments

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

2. A compiler that translates the source PL to the destination PL
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
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

• 9 assignments (9 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. Two manager supports
3. Secretary

<table>
<thead>
<tr>
<th>Grade</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&gt;= 13</td>
</tr>
<tr>
<td>A -</td>
<td>10 - 12</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
  • Declare your pair by the next lecture (send email to TA)

• 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: 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
• Structure
• Intro to compilers
• L1
  → F.E. → M.E. → B.E.

Thursday
• Compiler structure
• Parsing
• From L1 to x86_64
Outline

• Structure of the course

• Compilers

• Compiler IRs
Math

Arch

Practice

Compilers

PL
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: 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

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

Character stream (Source code)

Lexical analysis

Tokens

Syntactic & semantic analysis

AST

\[
\text{int main ...}
\]

\[
\text{INT SPACE STRING SPACE ...}
\]

Function signature

Return type

Function name

\[
\text{INT}
\]

\[
\text{STRING}
\]
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)

\[ \text{int main} \ldots \]

Front-end

\[ \text{EECS 322: Compiler Construction} \]

\[ \text{myVarX} = 40 \]

\[ \text{myVarY} = \text{myVarX} + 2 \]

Middle-end

\[ \text{EECS 323: Code analysis and transformation} \]

\[ \text{myVarY} = 42 \]

Back-end

\[ \text{EECS 322: Compiler Construction} \]

Machine code

010101110101010101
Outline

• Structure of the course

• Compilers

• Compiler IRs
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

Programming language

Translation

IR1

Translation

IR2

Translation

Machine code
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

- 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

```assembly
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

```assembly
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