C⚙mpiler
C⚙nstruction

Welcome!

Simone Campanoni
simone.campanoni@northwestern.edu

NORTHWESTERN UNIVERSITY
1851


Namaste
مرحبا
Willkommen
Bem Vindo
Selamat Datang
Namaste
Croeso
Bienvenidos
Bienvenue
Bienvenue
Welcome
Bienvenidos
أهلا وسهلا
Welcome
Benvenuti
Welkom
Bem Vindo
Bienvenue
مرحبا
مرحبا
Welcome
Croeso
Welkom
Bienvenidos
Namaste
Welkom
أهلا وسهلا
مرحبا
أهلا وسهلا
Selamat Datang
Bienvenue
Welcome
Bem Vindo
Willkommen
Willkommen
Selamat Datang
Croeso
добре дошъл
Benvenuti
Willkommen
Καλώς ήλθατε
Benvenuti

Who we are

Simone Campanoni

Brian Homerding

BrianHomerding2026@u.northwestern.edu

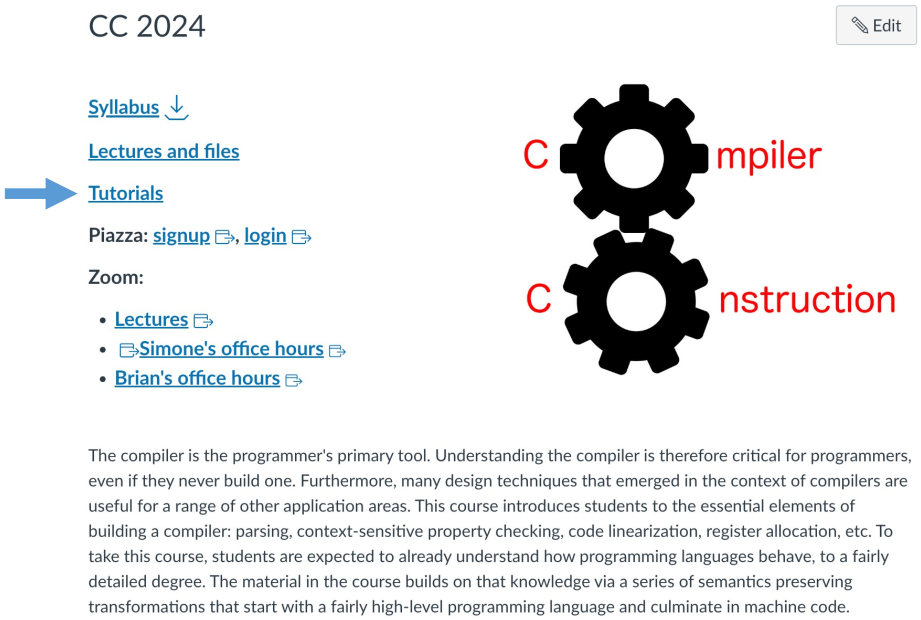
Outline

- 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**:
    - Brian: Friday 4:30pm – 6:30pm via Zoom (link on Canvas)
    - Simone: Monday 4:30pm – 5:30pm via Zoom (link on Canvas)
- CC is on Canvas
    - Materials/Assignments/Grades on Canvas

CC in a nutshell



CC 2024

[Syllabus](#)

[Lectures and files](#)

[Tutorials](#)

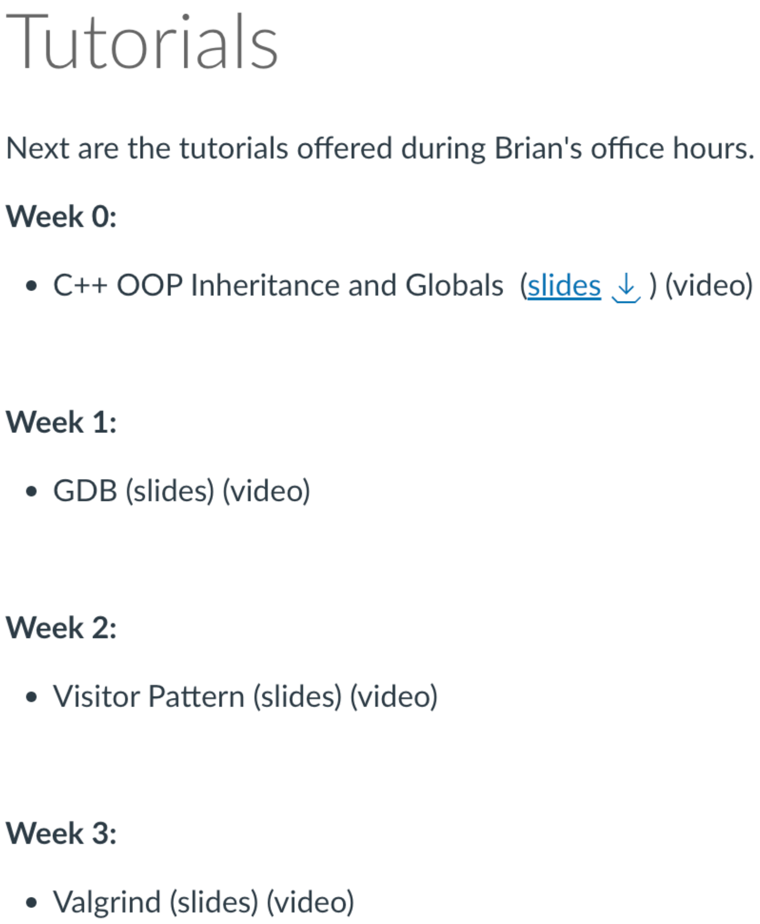
Piazza: [signup](#), [login](#)

Zoom:

- [Lectures](#)
- [Simone's office hours](#)
- [Brian'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.

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



Tutorials

Next are the tutorials offered during Brian's office hours.

Week 0:

- C++ OOP Inheritance and Globals ([slides](#)) (video)

Week 1:

- GDB (slides) (video)

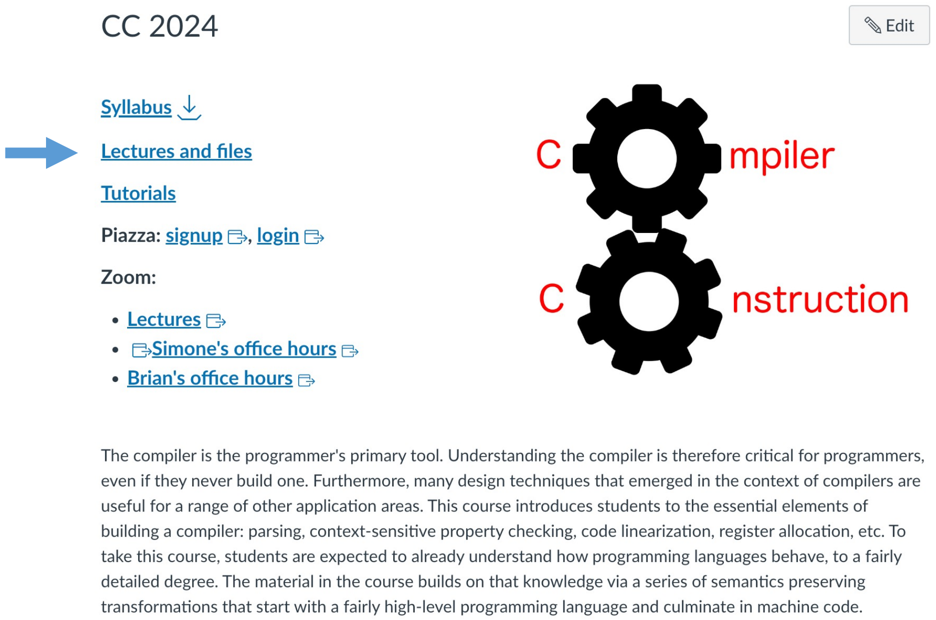
Week 2:

- Visitor Pattern (slides) (video)

Week 3:

- Valgrind (slides) (video)

CC in a nutshell



CC 2024

Syllabus

Lectures and files

Tutorials

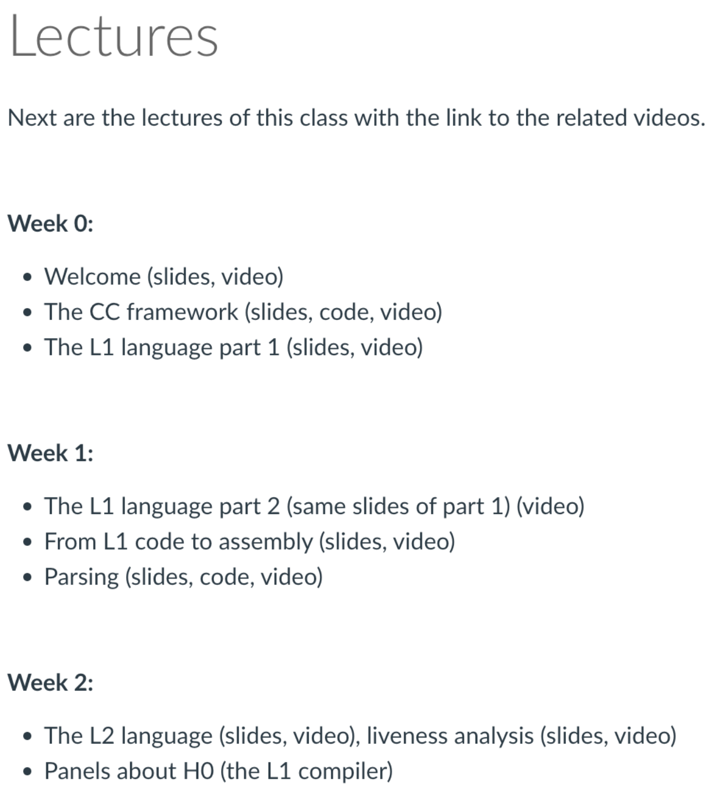
Piazza: signup, login

Zoom:

- Lectures
- Simone's office hours
- Brian'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.

- 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, video)
- The CC framework (slides, code, video)
- The L1 language part 1 (slides, video)

Week 1:

- The L1 language part 2 (same slides of part 1) (video)
- From L1 code to assembly (slides, video)
- Parsing (slides, code, video)

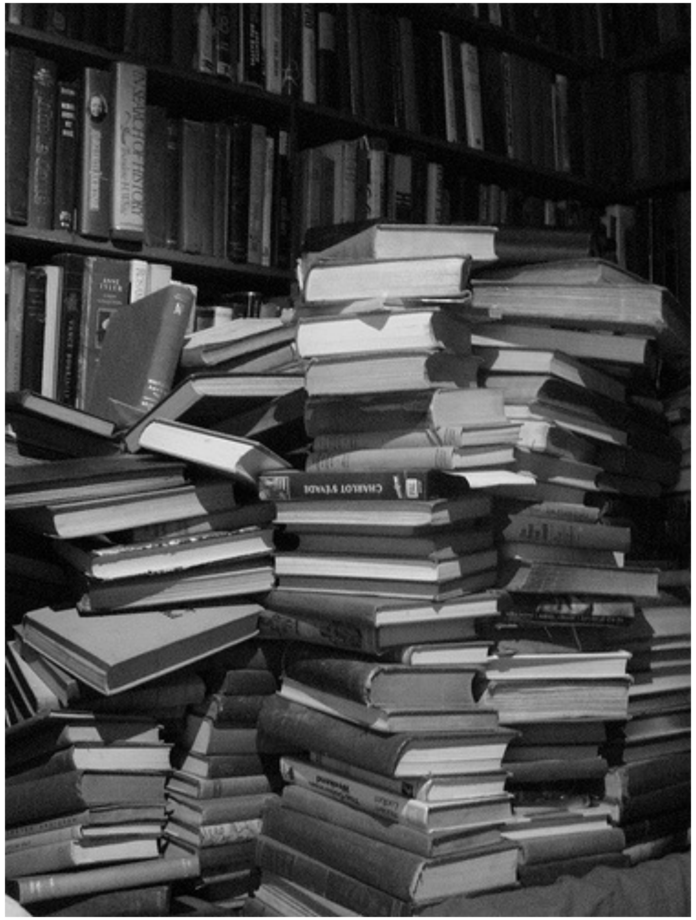
Week 2:

- The L2 language (slides, video), liveness analysis (slides, video)
- Panels about H0 (the L1 compiler)

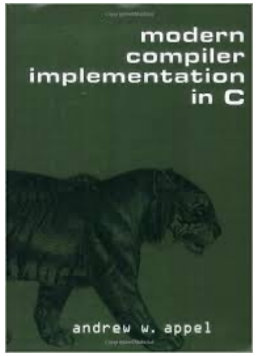
CC in a nutshell

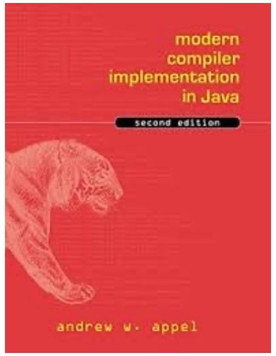
- **CS 322**: main blocks of modern compilers
  - Satisfy the system breadth for CS major
- **When**: Tuesday/Thursday 5pm - 6:20pm
- **Where**: here ☺
- **Office hours**:
  - Tommy: Friday 12:30pm – 2:30pm via Zoom (link on Canvas)
  - Simone: Tuesday Noon – 1:00pm via Zoom (link on Canvas)
- CC is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

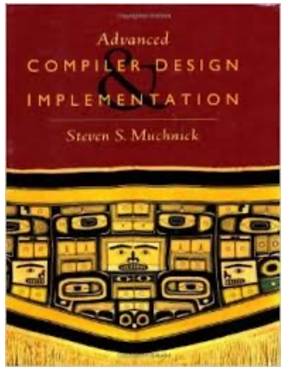
CC materials

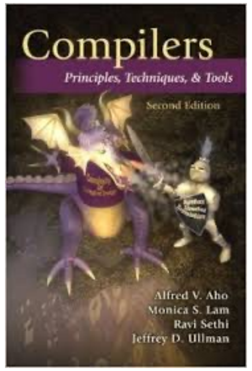


- Slides
- Books





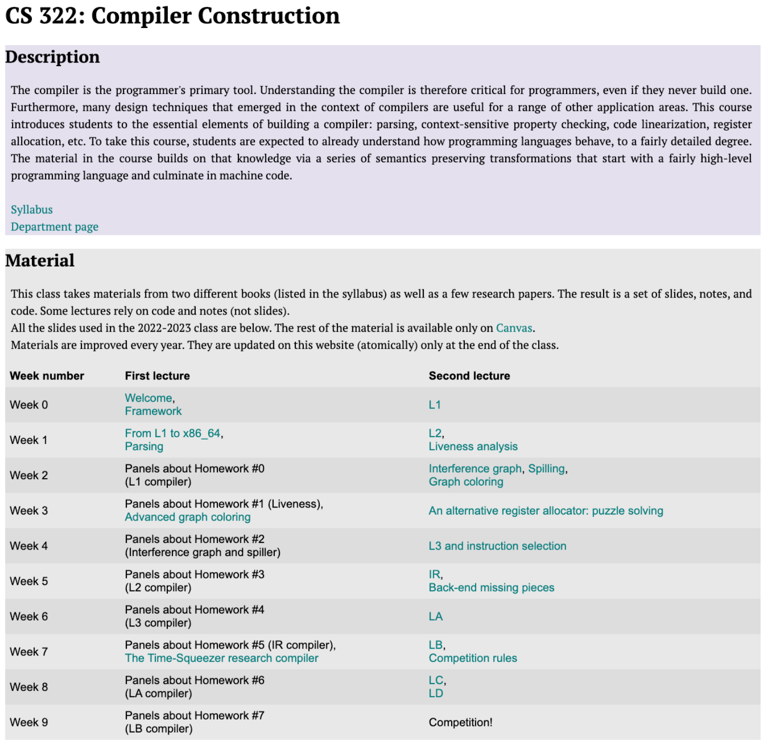




- Papers and library documentation for further information

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 before each class



CS 322: Compiler Construction

Description

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.

Syllabus
Department page

Material

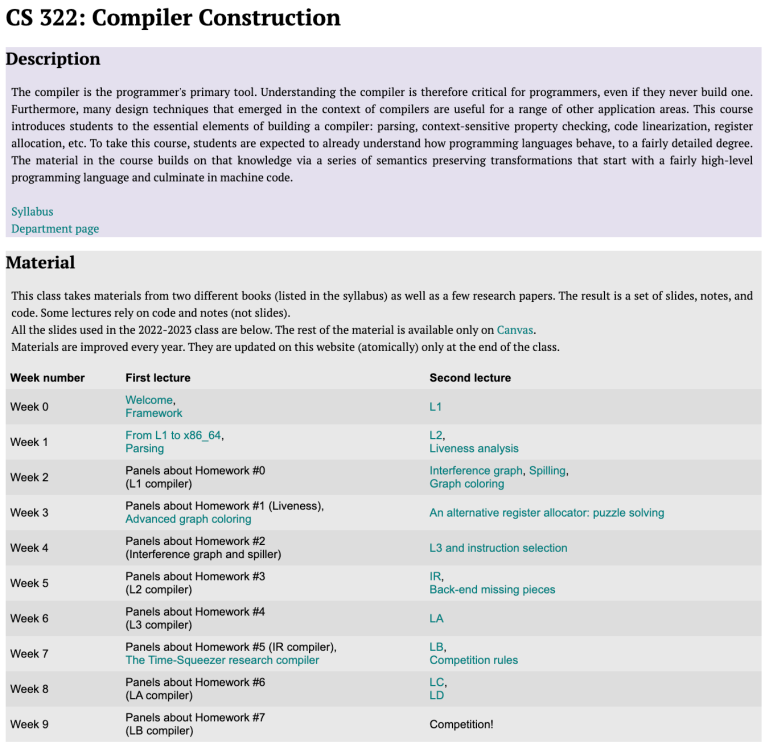
This class takes materials from two different books (listed in the syllabus) as well as a few research papers. The result is a set of slides, notes, and code. Some lectures rely on code and notes (not slides).
All the slides used in the 2022-2023 class are below. The rest of the material is available only on Canvas.
Materials are improved every year. They are updated on this website (atomically) only at the end of the class.

Week number	First lecture	Second lecture
Week 0	Welcome, Framework	L1
Week 1	From L1 to x86_64, Parsing	L2, Liveness analysis
Week 2	Panels about Homework #0 (L1 compiler)	Interference graph, Spilling, Graph coloring
Week 3	Panels about Homework #1 (Liveness), Advanced graph coloring	An alternative register allocator: puzzle solving
Week 4	Panels about Homework #2 (Interference graph and spiller)	L3 and instruction selection
Week 5	Panels about Homework #3 (L2 compiler)	IR, Back-end missing pieces
Week 6	Panels about Homework #4 (L3 compiler)	LA
Week 7	Panels about Homework #5 (IR compiler), The Time-Squeezer research compiler	LB, Competition rules
Week 8	Panels about Homework #6 (LA compiler)	LC, LD
Week 9	Panels about Homework #7 (LB compiler)	Competition!

CC slides

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

- We will cover one topic per week



CS 322: Compiler Construction

Description

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.

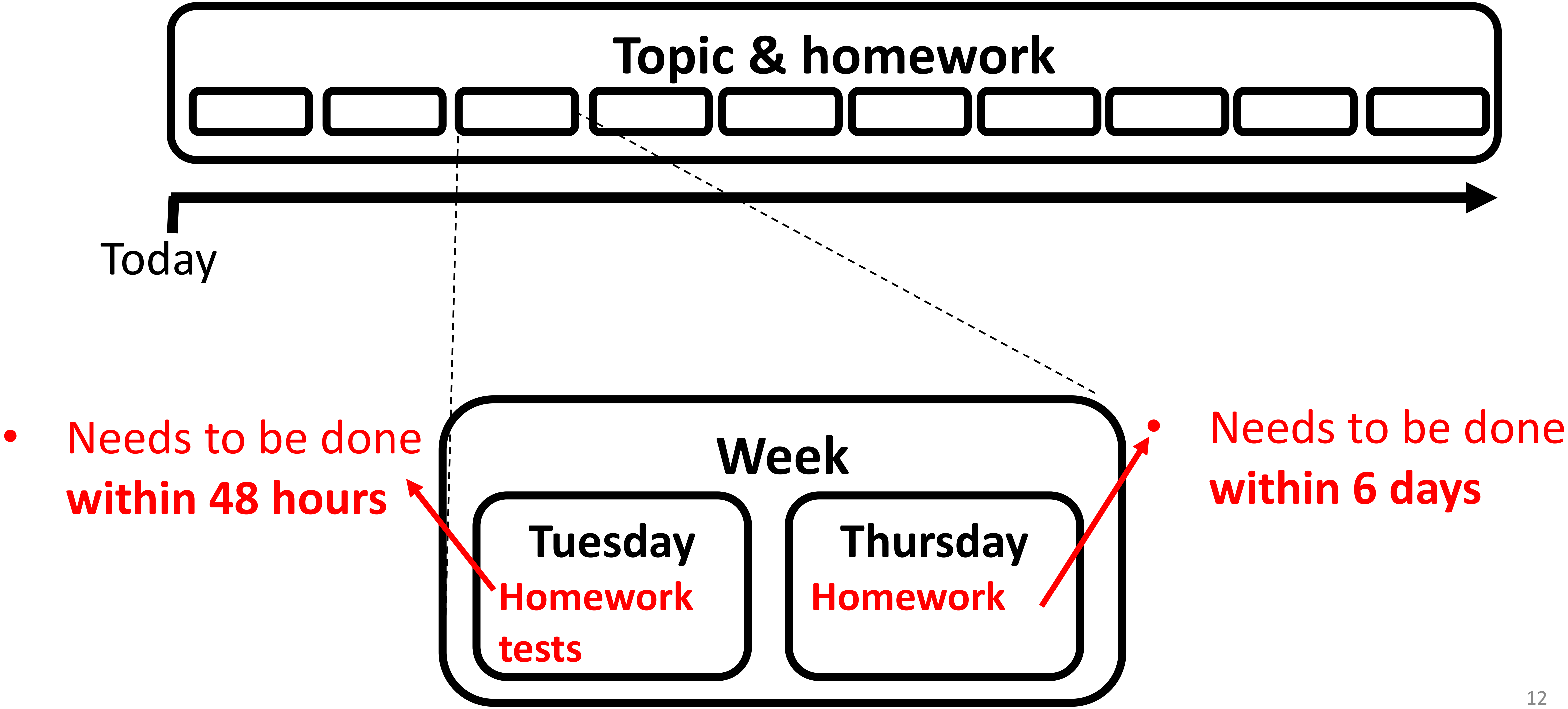
Syllabus
Department page

Material

This class takes materials from two different books (listed in the syllabus) as well as a few research papers. The result is a set of slides, notes, and code. Some lectures rely on code and notes (not slides).
All the slides used in the 2022-2023 class are below. The rest of the material is available only on Canvas.
Materials are improved every year. They are updated on this website (atomically) only at the end of the class.

Week number	First lecture	Second lecture
Week 0	Welcome, Framework	L1
Week 1	From L1 to x86_64, Parsing	L2, Liveness analysis
Week 2	Panels about Homework #0 (L1 compiler)	Interference graph, Spilling, Graph coloring
Week 3	Panels about Homework #1 (Liveness), Advanced graph coloring	An alternative register allocator: puzzle solving
Week 4	Panels about Homework #2 (Interference graph and spiller)	L3 and instruction selection
Week 5	Panels about Homework #3 (L2 compiler)	IR, Back-end missing pieces
Week 6	Panels about Homework #4 (L3 compiler)	LA
Week 7	Panels about Homework #5 (IR compiler), The Time-Squeezer research compiler	LB, Competition rules
Week 8	Panels about Homework #6 (LA compiler)	LC, LD
Week 9	Panels about Homework #7 (LB compiler)	Competition!

The CC structure



Topic & homework

Today

Week

Tuesday

Homework

tests

Thursday

Homework

- 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**

Source code (C like)

Homework N

…

Homework 2

Homework 1

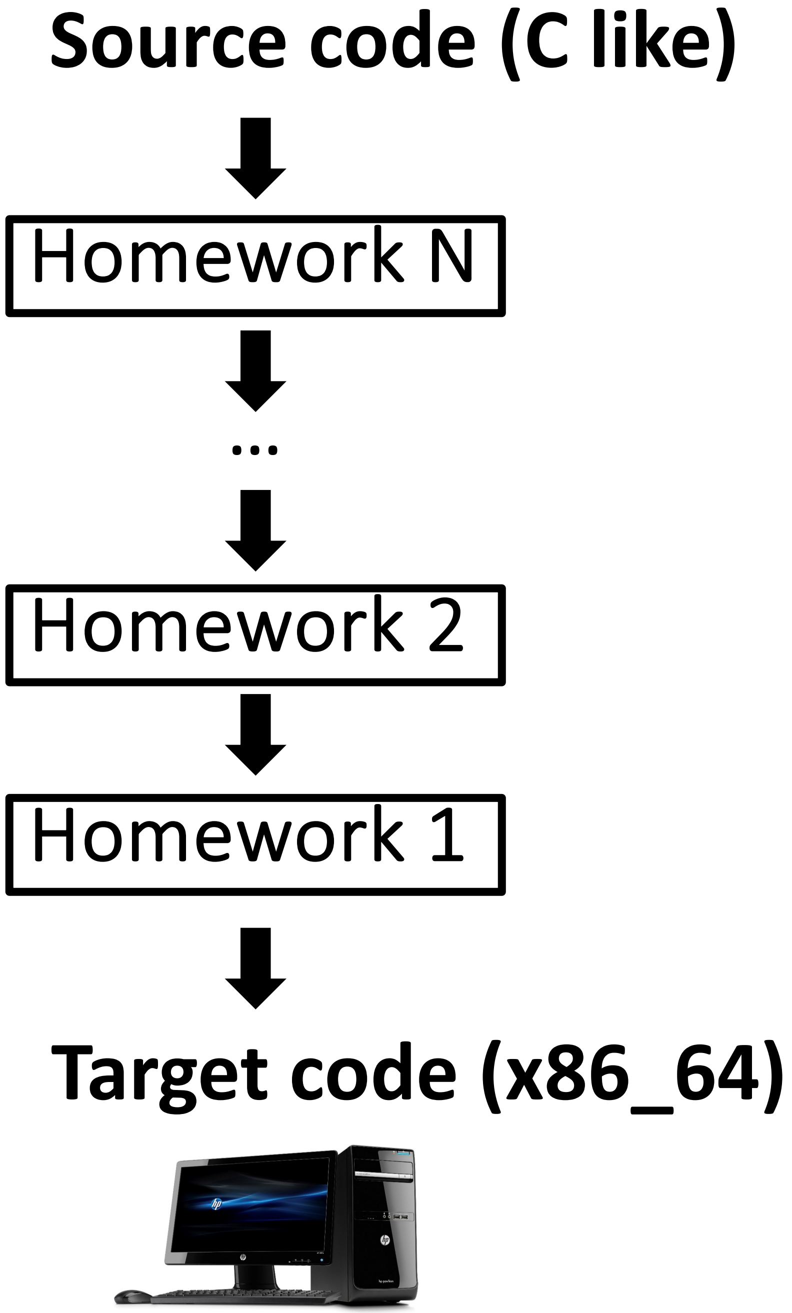
Target code (x86_64)

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)

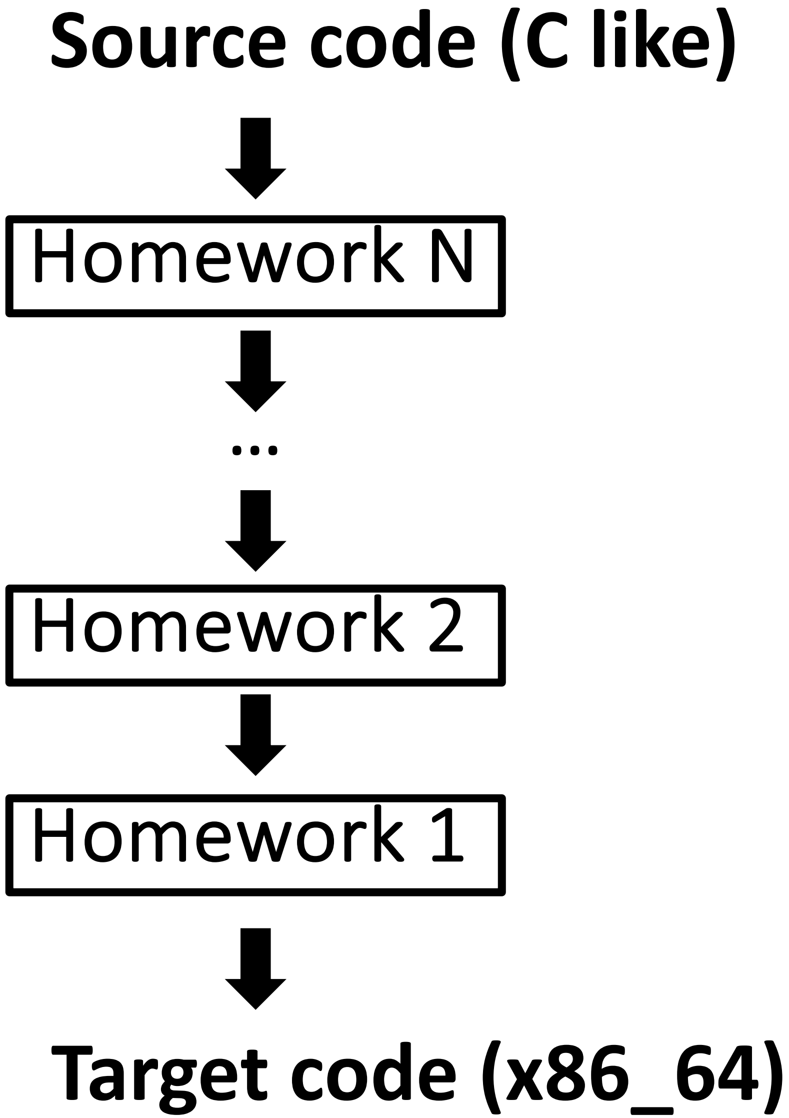
Evaluation of your work

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)*



Source code (C like)

Homework N

…

Homework 2

Homework 1

Target code (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

Hall of Fame

Students design and build a complete compiler able to translate an almost-C language to Intel x86-64 machine code. At the end of the class, the resulting compilers compete and the names of the students that designed and built the best compilers are reporeted below.

Year	Name	Picture
2018	Matt C. Cheung	
2017	Zhiping Xiu	

The CC grading **No final exam**

- 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

Grade	Passed
A	>= 11
A -	10
B +	9
B	8
B-	7
C+	6
C	5
C-	4
D	3
F	0 – 2

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

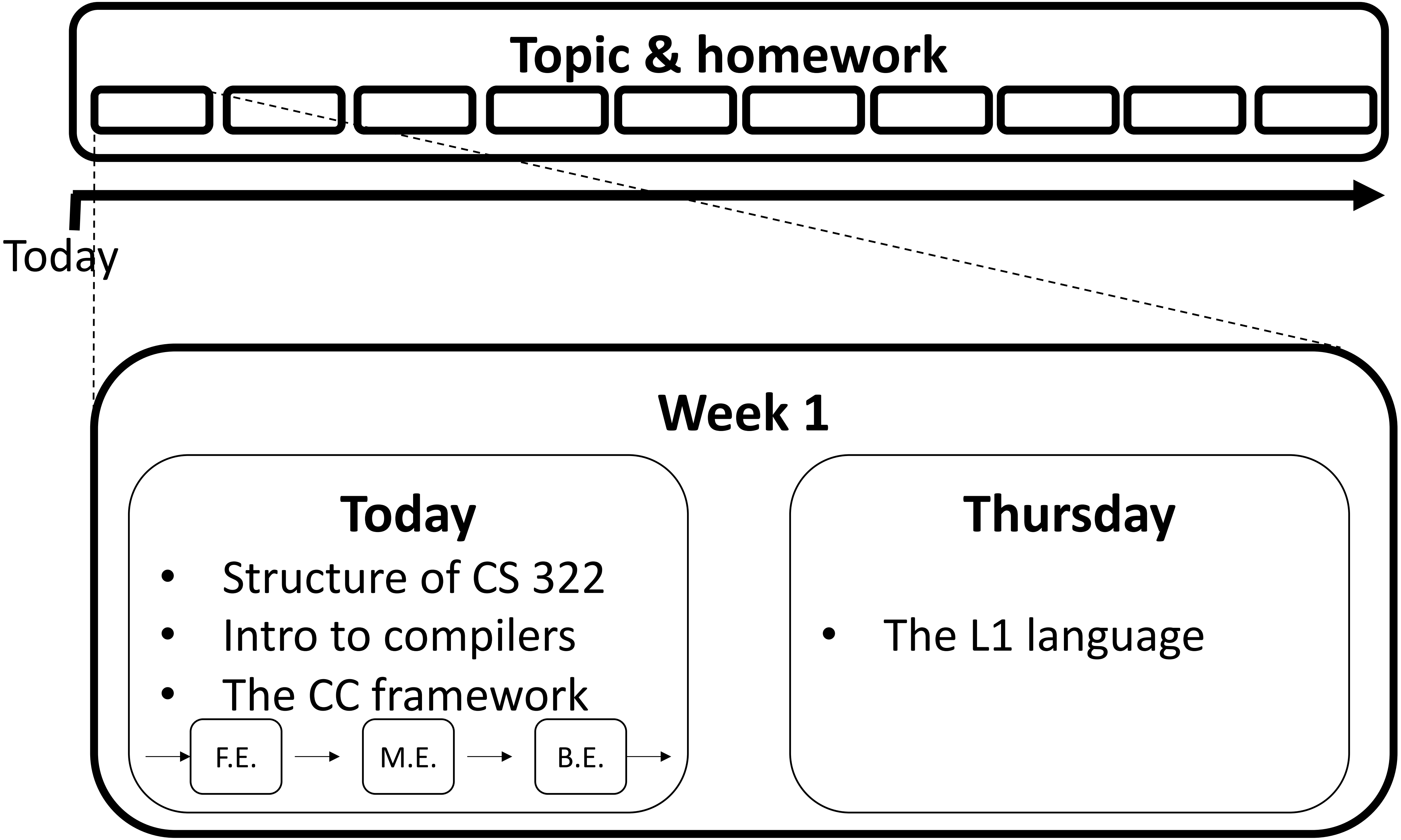
- Your duties
  - Learn all compiler blocks presented in class
  - Implement some of them (the most important ones)
    - Write code in C++
    - Test your code
    - Then, think **much harder** about how to **actually** test your code
    - Be ready for being in a panel when asked (the day before)

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



Topic & homework

Today

Week 1

Today

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

Thursday

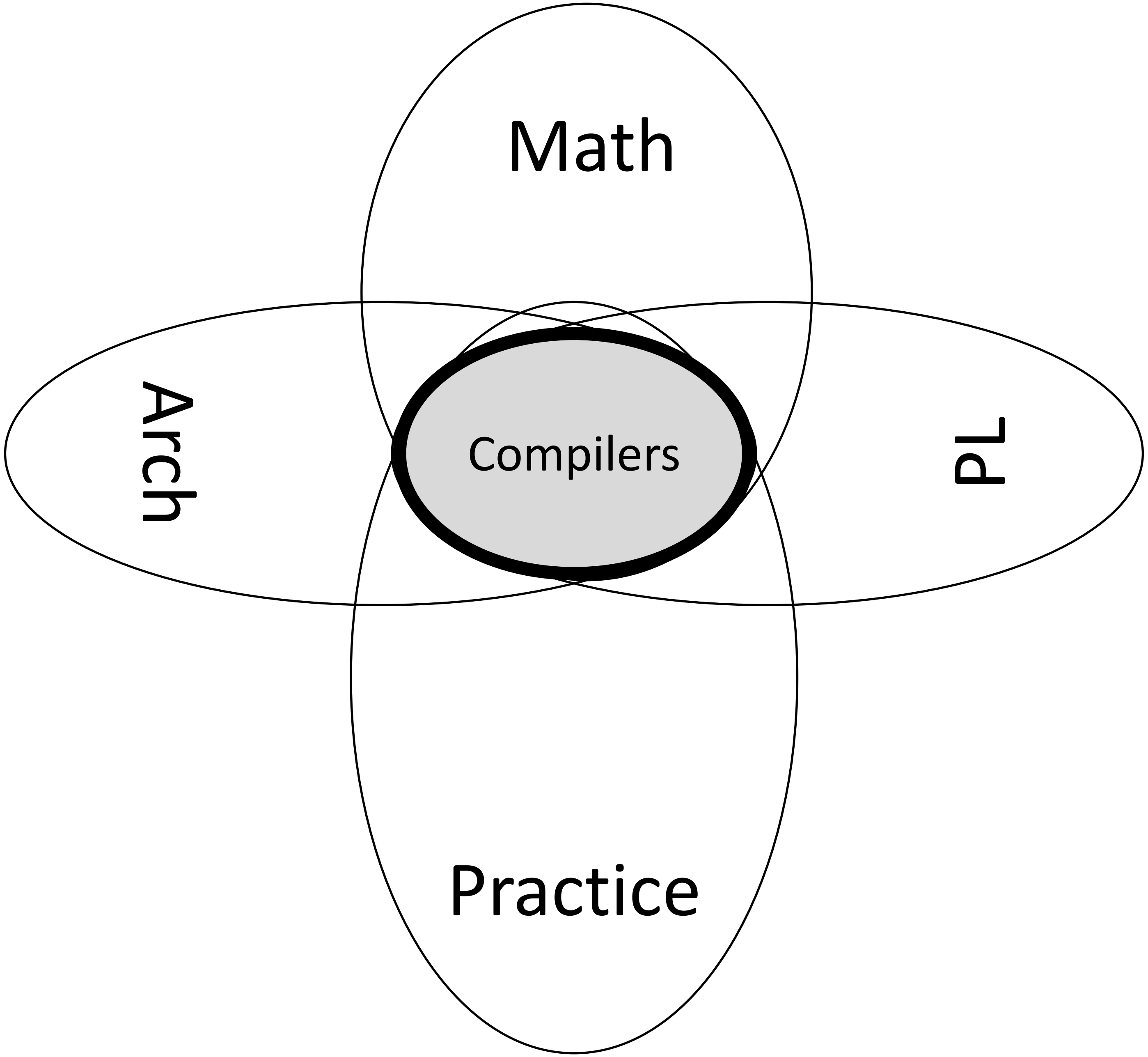
- The L1 language

Outline

- Structure of the course

- Compilers

- Compiler IRs


Math
Arch
Compilers
PL
Practice

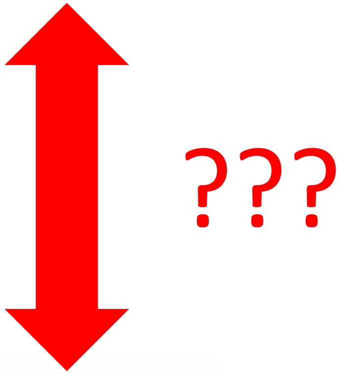
The role of compilers

00101010111001010101001010101011010

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;
}
```



00101010111001010101001010101011010

```
While (s = nextStatement()){
  switch (s.type){
    case IF: …
    case ADDITION: …
    case RETURN: …
  }
}
```

```
→ 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

00101010111001010101001010101011010

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

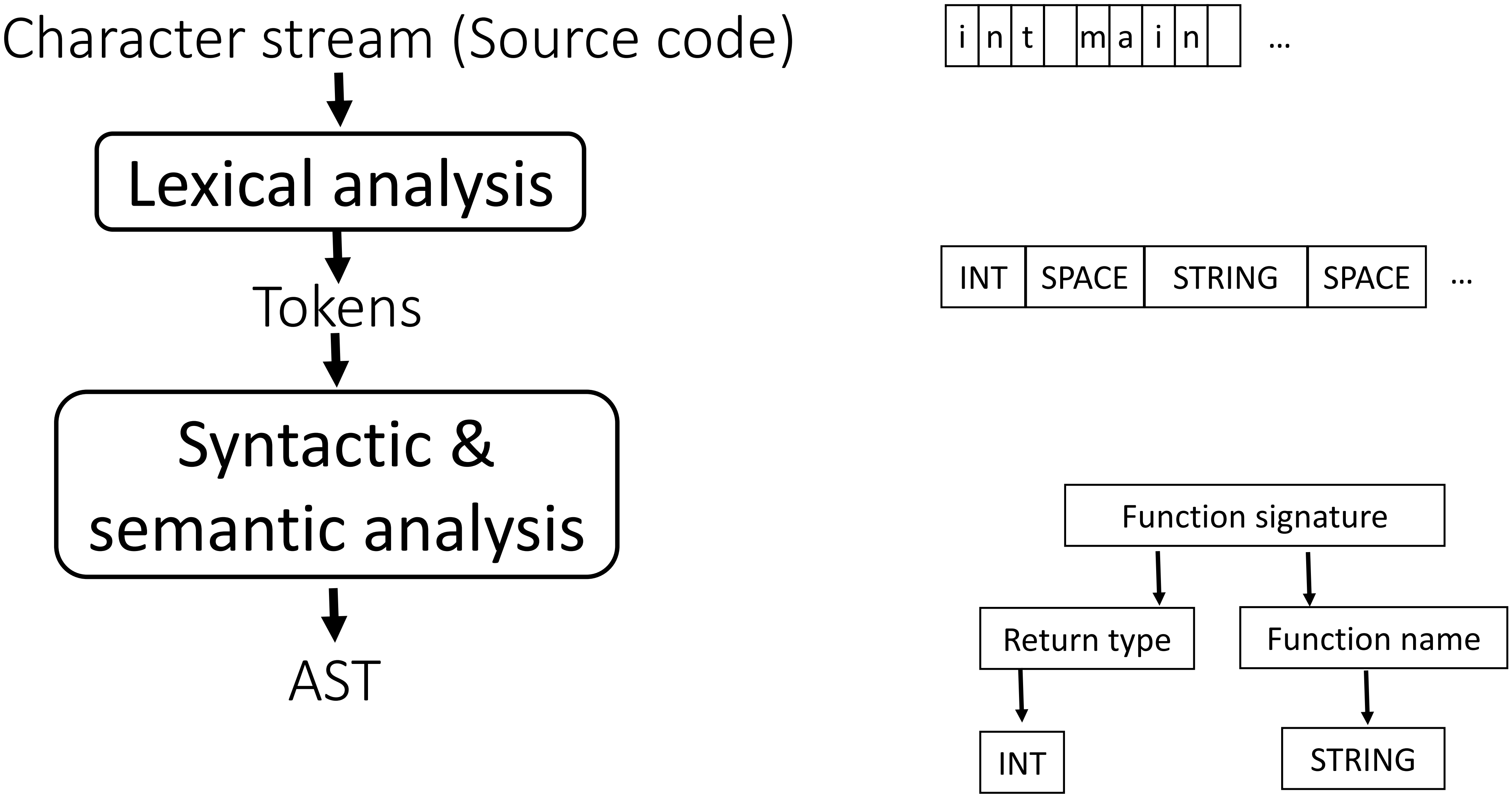
AST

i	n	t		m	a	i	n		…

INT	SPACE	STRING	SPACE	…

```
int main (){
  printf(“Hello World!\n”);
  return 0;
}
```

Structure of a compiler



Character stream (Source code)

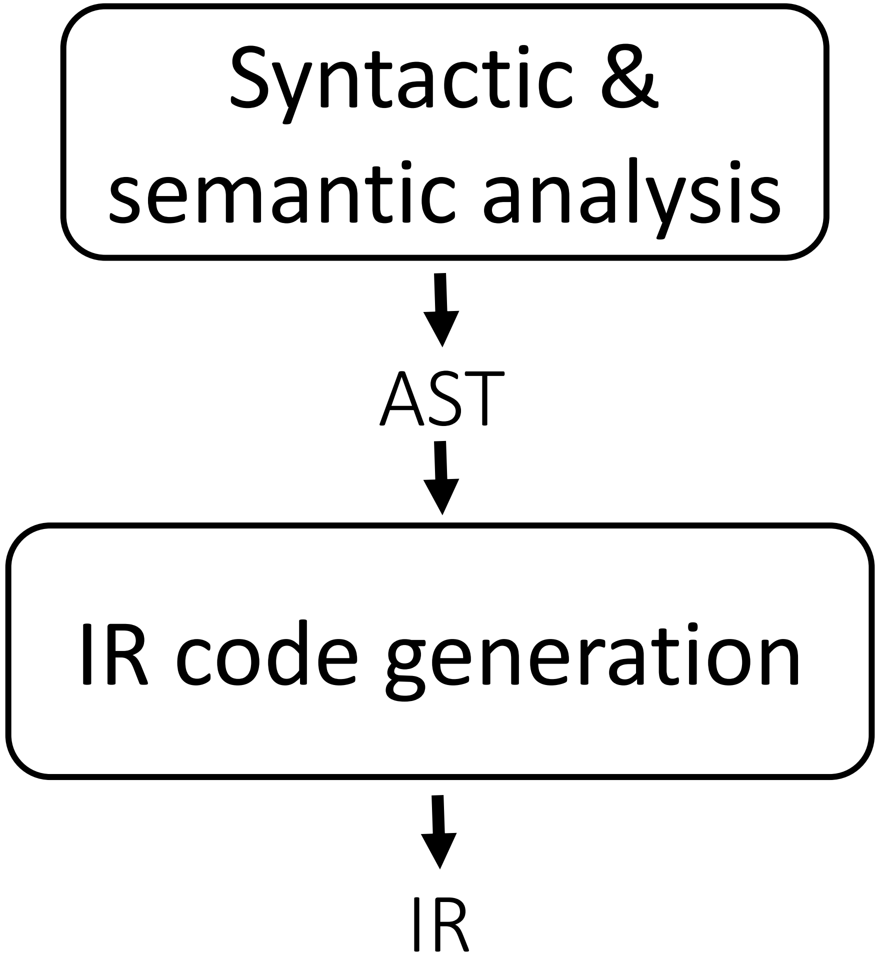
Lexical analysis

Tokens

Syntactic & semantic analysis

AST

Structure of a compiler

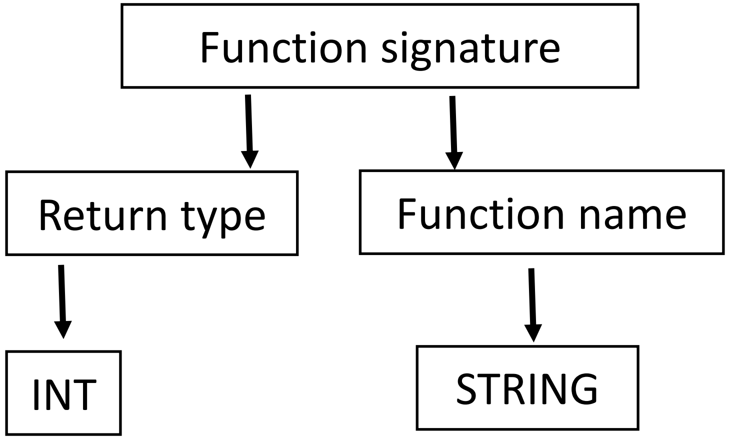


Syntactic & semantic analysis

AST

IR code generation

IR



Function signature

Return type

Function name

INT

STRING

myVarX = 40
myVarY = myVarX + 2

Structure of a compiler

Character stream (Source code)

i	n	t		m	a	i	n	

…

↓

Front-end

CS 322: Compiler Construction

↓

IR

↓

myVarX = 40
myVarY = myVarX + 2

Middle-end

CS 323: Code analysis and transformation

↓

IR

↓

myVarY = 42

Back-end

CS 322: Compiler Construction

↓

Machine code

010101110101010101

Outline

- Structure of the course
- Compilers
- Compiler IRs

Example of LLVM IR

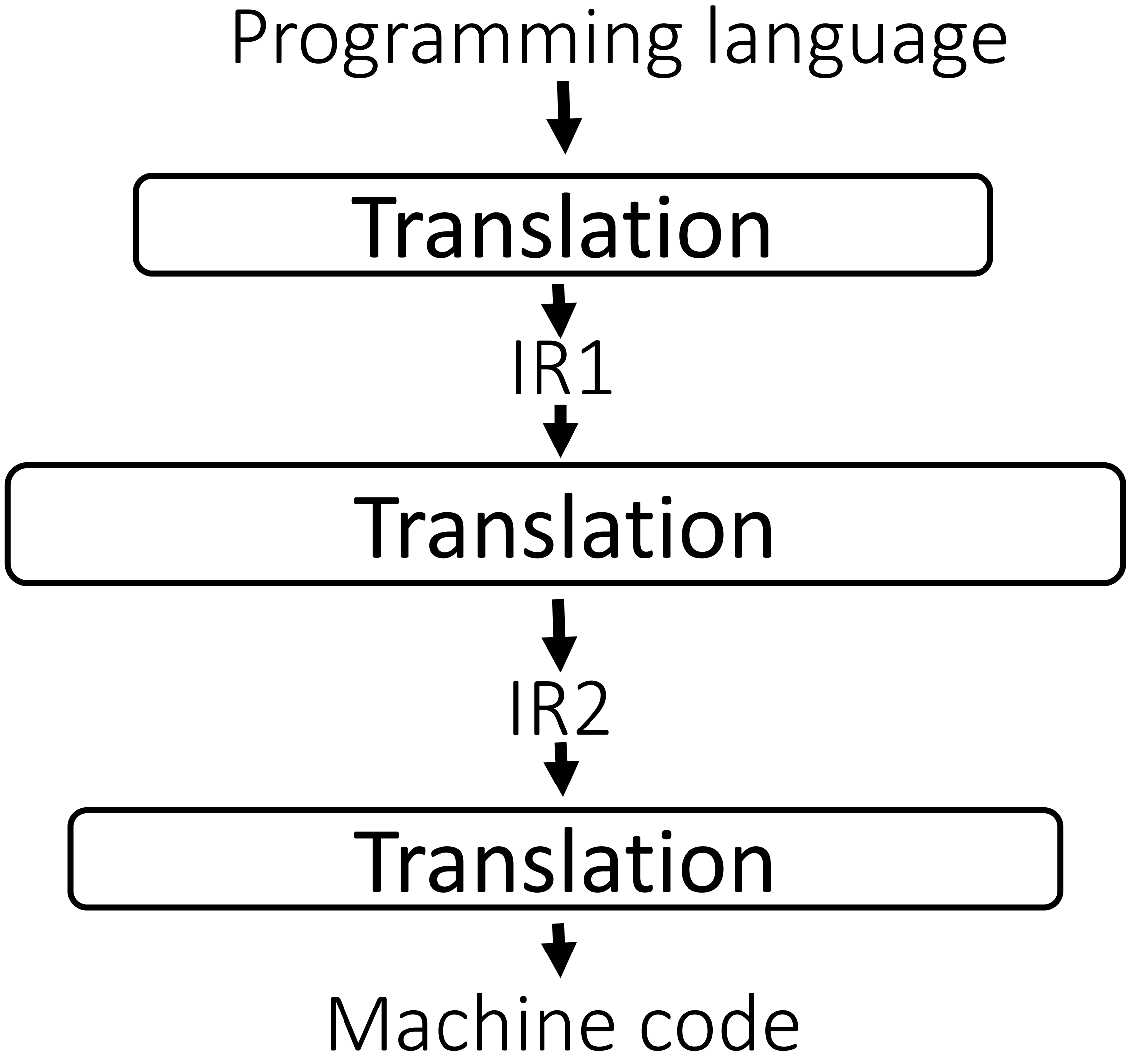
```
define i64 @f (i64 %p0) {
  entry:
    %myVar1 = add i64 %p0, 1
    ret i64 %myVar1
}
```

Another example of IR

```
define int64 @f (int64 %p0) {
  :entry
    int64 %myVar1
    %myVar1 <- %p0 + 1
    return %myVar1
}
```

C⚙mpiler
C⚙nstruction

Multiple IRs used together



Programming language

Translation

IR1

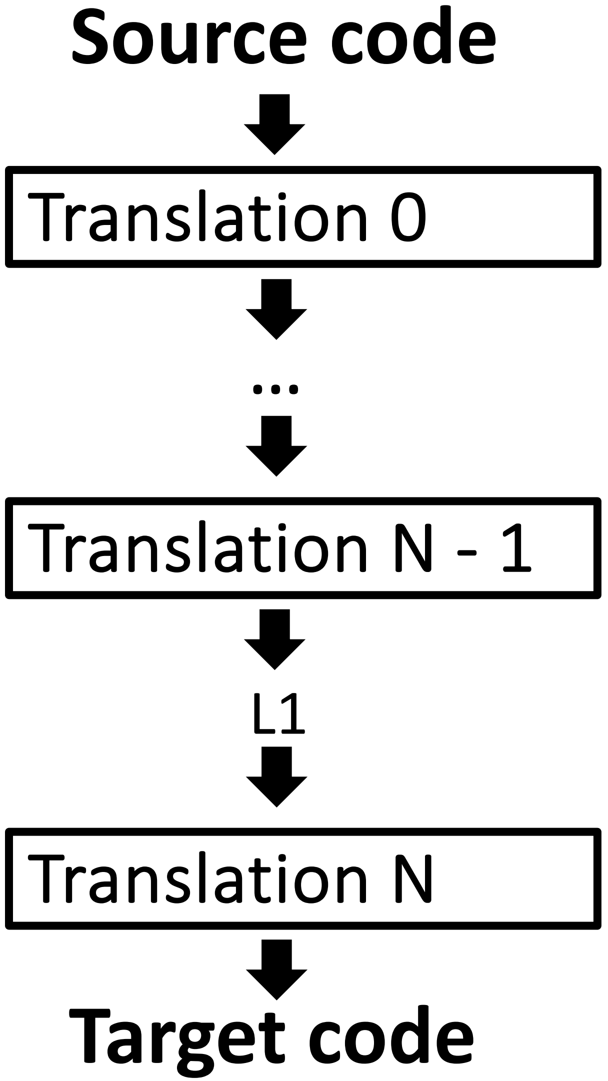
Translation

IR2

Translation

Machine code

IRs are languages



Source code

Translation 0

…

Translation N - 1

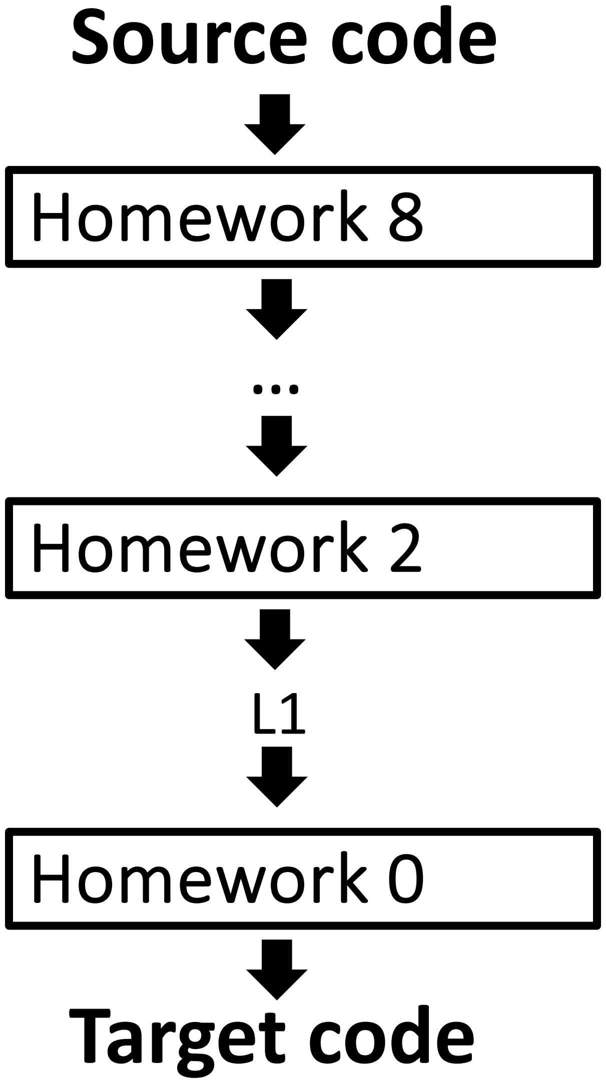
L1

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



Source code

Homework 8

…

Homework 2

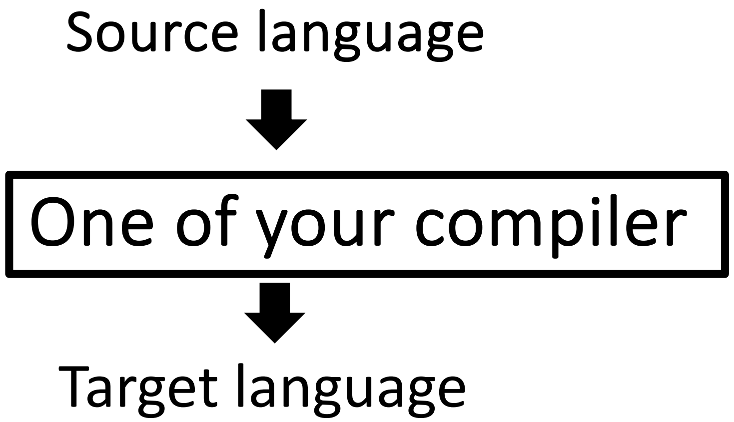
L1

Homework 0

Target 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



Source language

One of your compiler

Target language

- 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)

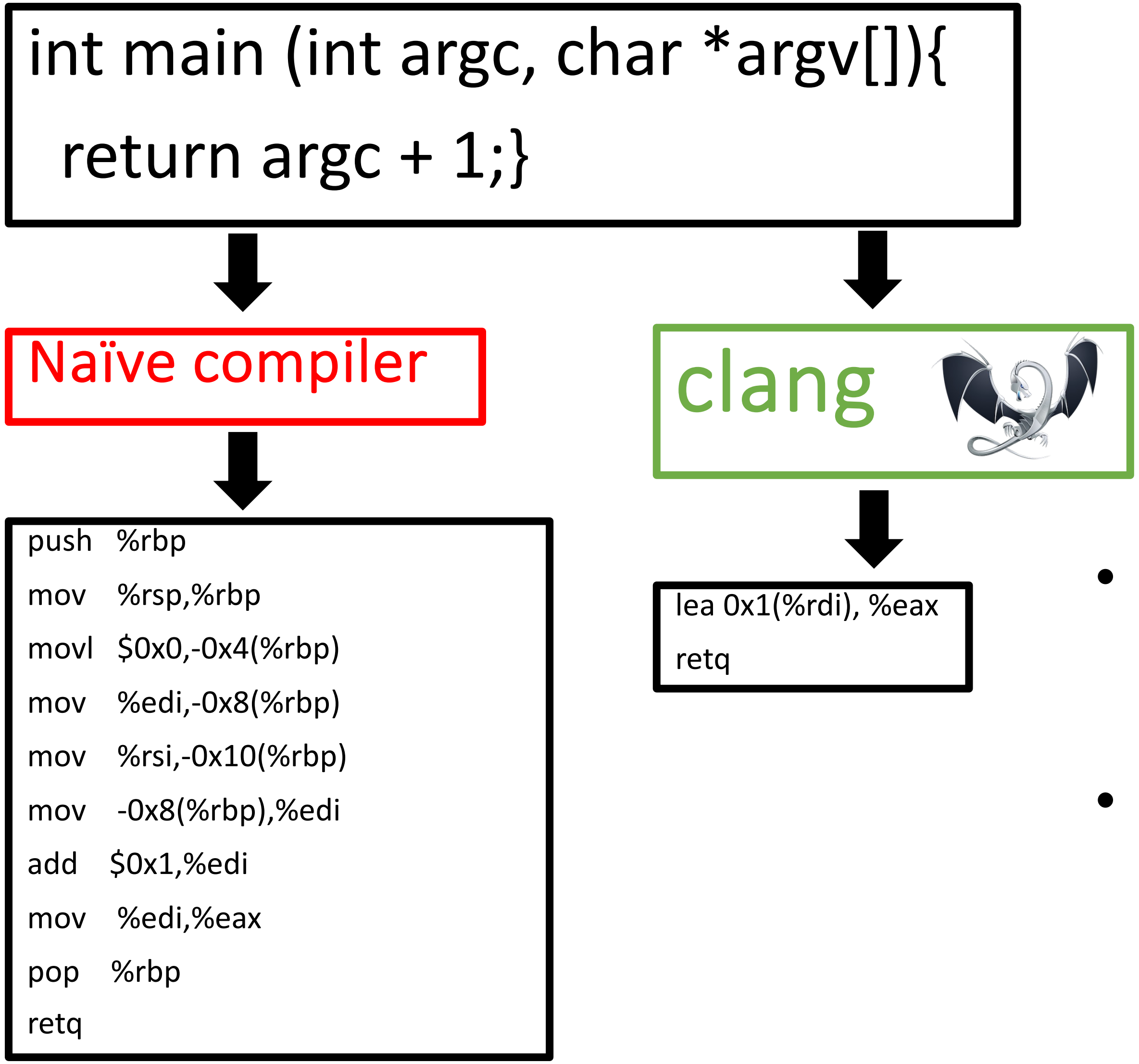
Let’s build our first compiler

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;}
```

```
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
```

```
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!