code analysis *and* transformation

Welcome!

Simone Campanoni
simone.campanoni@northwestern.edu

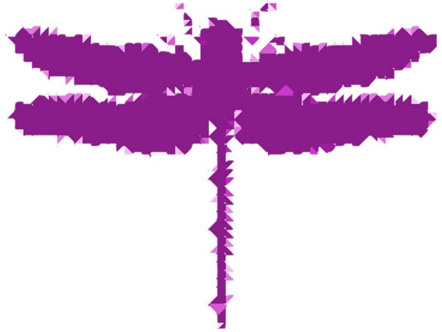
Namaste
مرحبا
Willkommen
Bem Vindo
Selamat Datang
Namaste
Croeso
Bienvenidos
Bienvenue
Bienvenue
Welcome
Bienvenidos
أهلا وسهلا
Benvenuti
Welcome
Welkom
Bem Vindo
Bienvenue
مرحبا
مرحبا
Welcome
Croeso
Bienvenidos
Welkom
Welkom
أهلا وسهلا
Namaste
Selamat Datang
مرحبا
أهلا وسهلا
Welcome
Bienvenue
Bem Vindo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

The CAT team

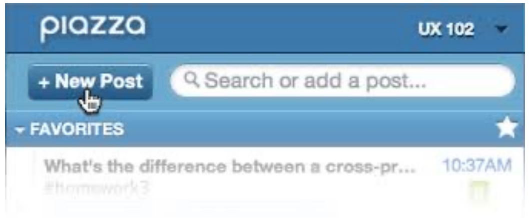
All of us have office hours to answer your questions throughout the quarter

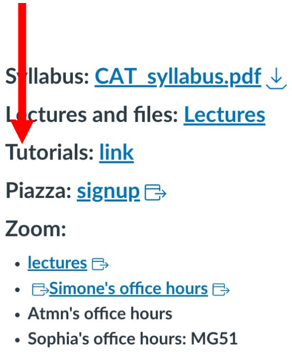
Simone Campanoni

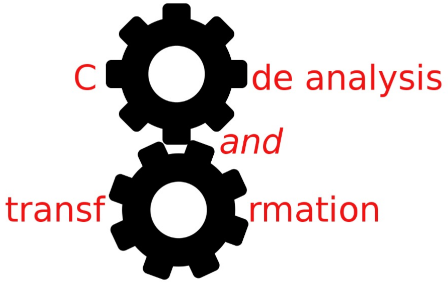
Atmn Patel (TA)



Riley Sophia Boksenbaum (PM)

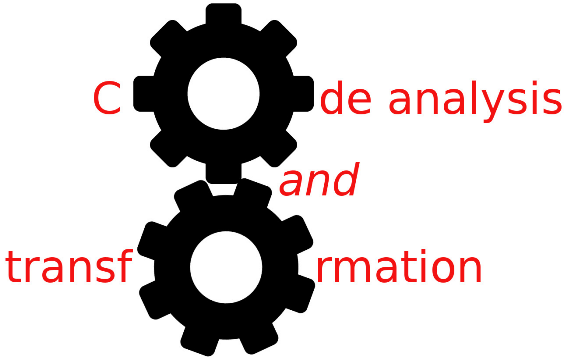






What we are going to do

- Teach you **code analysis and transformation**



- What they do
- What they could do

- What they can’t do

CAT

Who you are

- An engineer

- A C++ developer
  (you don’t have to be an incredible coder)

- An enthusiastic learner

Compiler expert is not mentioned ;)

Software knowledge assumed

- You know how to write C++ code in Linux platforms
  (e.g., class, inheritance, method overloading, containers like a set)
  C++ tutorial: http://www.cplusplus.com/doc/tutorial/

- You know Makefile
  Makefile tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

- You know how to debug C++ code
  gdb tutorial: https://www.tutorialspoint.com/gnu_debugger/index.htm

Machines to use for this class

You have access to the following machines,
which are used to test your homework

- Wilkinson lab
  gotham.ece.northwestern.edu, batman.ece.northwestern.edu, robin.ece.northwestern.edu, alfred.ece.northwestern.edu ,gordon.ece.northwestern.edu ,madhatter.ece.northwestern.edu ,joker.ece.northwestern.edu ,cobblepott.ece.northwestern.edu ,bane.ece.northwestern.edu ,nightwing.ece.northwestern.edu ,selina.ece.northwestern.edu ,ras.ece.northwestern.edu ,poisonivy.ece.northwestern.edu ,freeze.ece.northwestern.edu ,scarecrow.ece.northwestern.edu ,clayface.ece.northwestern.edu ,harley.ece.northwestern.edu ,killercroc.ece.northwestern.edu ,huntress.ece.northwestern.edu ,batgirl.ece.northwestern.edu ,riddler.ece.northwestern.edu ,hush.ece.northwestern.edu

- WOT systems
  murphy.wot.ece.northwestern.edu, finagle.wot.ece.northwestern.edu, hanlon.wot.ece.northwestern.edu, moore.wot.ece.northwestern.edu

Outline of today's CAT

- Structure of the course
- CAT and compilers
- CAT and computer architecture
- CAT and programming language

CS 323 CAT in a nutshell

- About: understanding and transforming code automatically
- Tuesday/Thursday 5pm – 6:20pm

- Atmn's office hours. : Wednesday 1pm – 3pm via Zoom *Starting next week*
- Sophia's office hours: Thursday 2pm – 4pm in MG51 *Starting this week*
- Simone's office hours: Monday 5pm – 6pm via Zoom *Starting next week*

- CAT is on Canvas
  - Materials/Assignments/Grades on Canvas
  - You'll upload your assignments on Canvas

Syllabus: CAT_syllabus.pdf

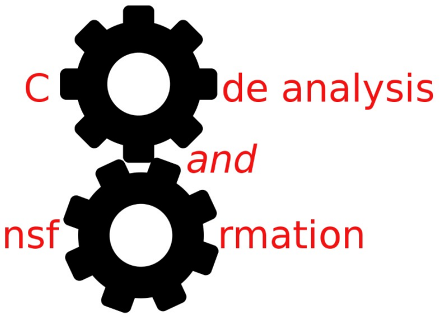
Lectures and files: Lectures

Tutorials: link

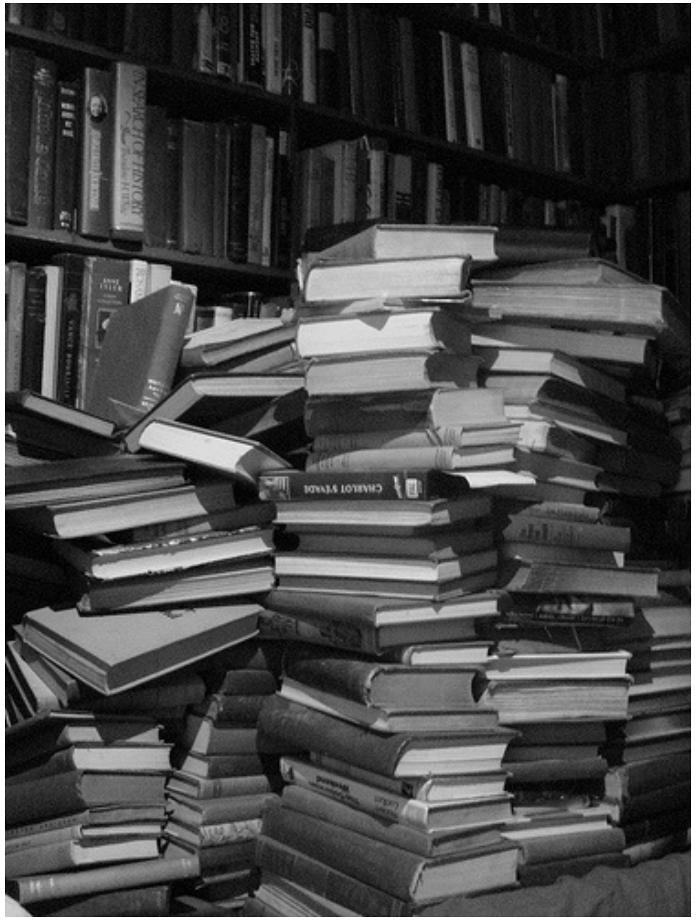
Piazza: signup

Zoom:

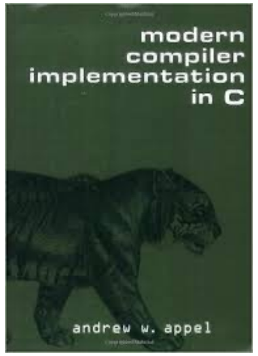
- lectures
- Simone's office hours
- Atmn's office hours
- Sophia's office hours: MG51

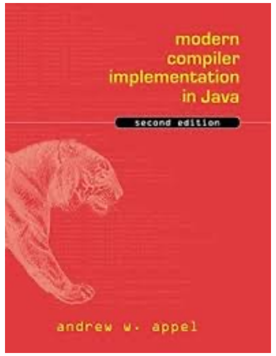


CAT materials



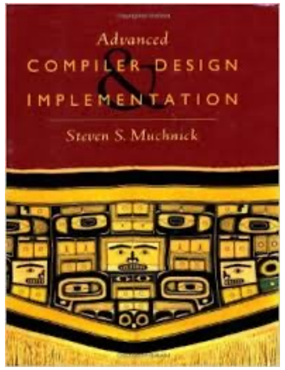
- Modern compiler implementation

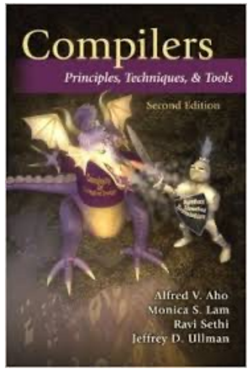




- Slides and assigned papers
- LLVM documentation

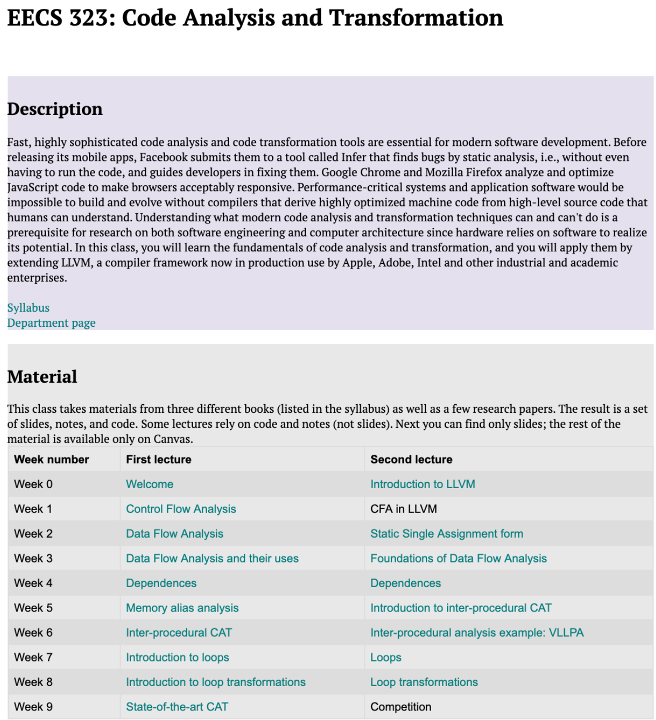
http://llvm.org





CAT 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 just before each class
- Slides support my teaching philosophy



EECS 323: Code Analysis and Transformation

Description

Fast, highly sophisticated code analysis and code transformation tools are essential for modern software development. Before releasing its mobile apps, Facebook submits them to a tool called Infer that finds bugs by static analysis, i.e., without even having to run the code, and guides developers in fixing them. Google Chrome and Mozilla Firefox analyze and optimize JavaScript code to make browsers acceptably responsive. Performance-critical systems and application software would be impossible to build and evolve without compilers that derive highly optimized machine code from high-level source code that humans can understand. Understanding what modern code analysis and transformation techniques can and can't do is a prerequisite for research on both software engineering and computer architecture since hardware relies on software to realize its potential. In this class, you will learn the fundamentals of code analysis and transformation, and you will apply them by extending LLVM, a compiler framework now in production use by Apple, Adobe, Intel and other industrial and academic enterprises.

Syllabus
Department page

Material

This class takes materials from three 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). Next you can find only slides; the rest of the material is available only on Canvas.

Week number	First lecture	Second lecture
Week 0	Welcome	Introduction to LLVM
Week 1	Control Flow Analysis	CFA in LLVM
Week 2	Data Flow Analysis	Static Single Assignment form
Week 3	Data Flow Analysis and their uses	Foundations of Data Flow Analysis
Week 4	Dependences	Dependences
Week 5	Memory alias analysis	Introduction to inter-procedural CAT
Week 6	Inter-procedural CAT	Inter-procedural analysis example: VLLPA
Week 7	Introduction to loops	Loops
Week 8	Introduction to loop transformations	Loop transformations
Week 9	State-of-the-art CAT	Competition

The spirit of my lectures
a.k.a. my teaching philosophy

- I’ll describe problems/opportunities
- I’ll describe concepts required to solve these problems (take advantage of these opportunities)
- I’ll describe their solutions that are based on these concepts

Problems/opportunities/concepts are structured in weeks

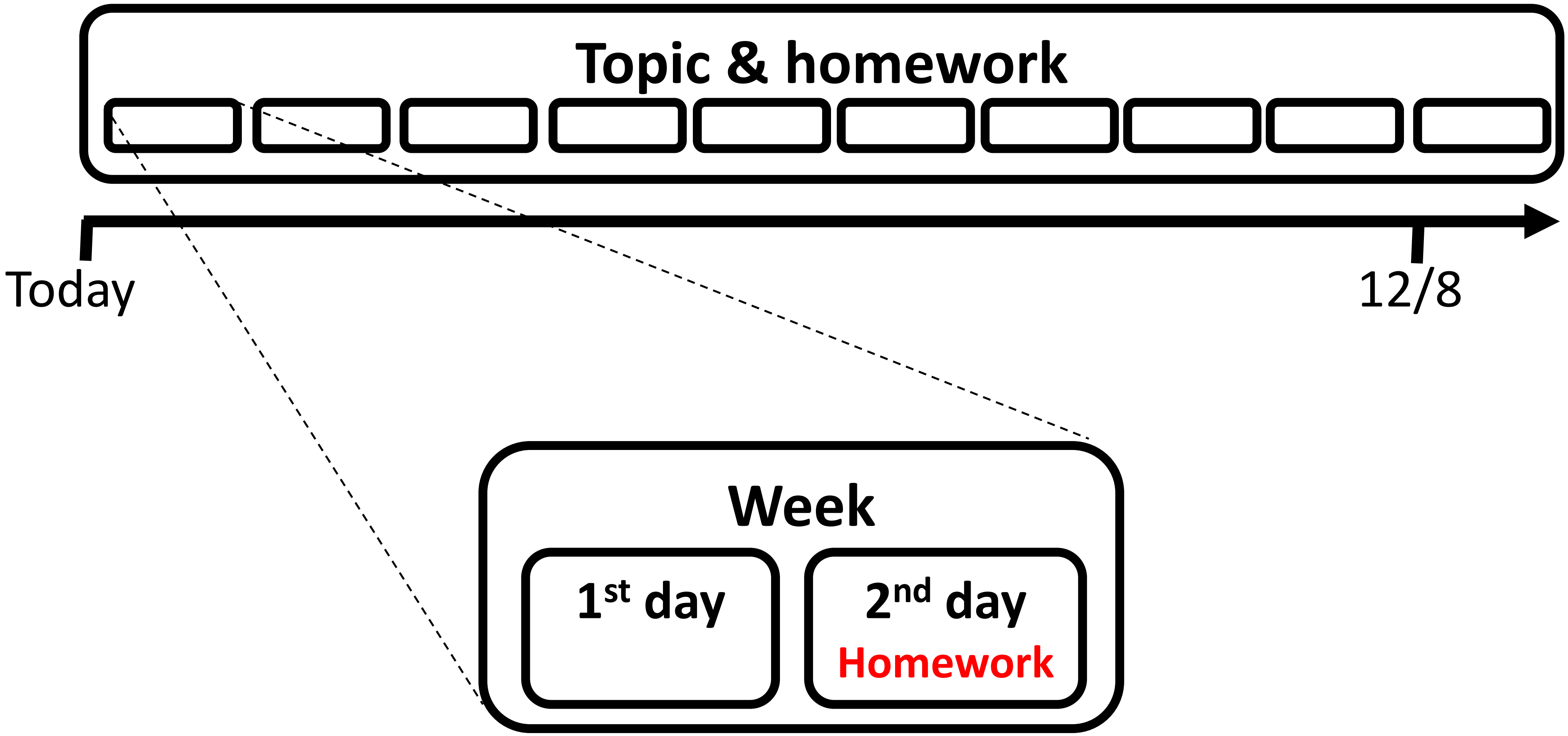
- I’ll describe new problems/opportunities

My output

Your output

- You’ll apply concepts/solutions learned during my lectures to solve the new problems/opportunities
  - Required to pass the homework

The CAT structure



The CAT grading

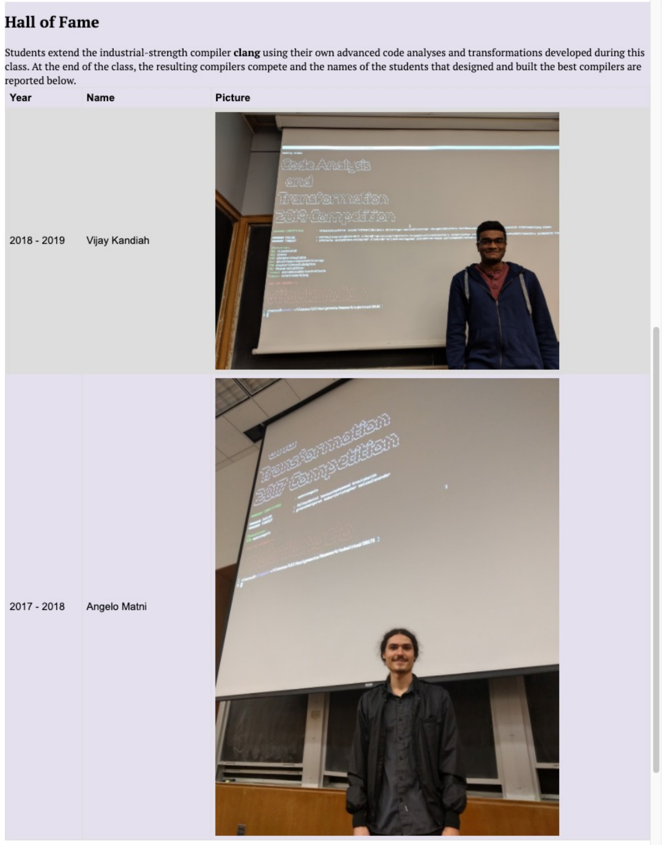
- Homework: 100 points
  - 10 points per assignment
  - The first assignment is easy
- Extra points
  - Extra homework
  - Answering (correctly) special questions (I will emphasize them) during lectures
  - Best student so far: **114 points!**

A	95 - 100+
A-	90 - 94
B+	83 - 89
B	74 - 82
B-	67 - 73
C+	60 - 66
C	55 - 59
C-	50 - 54
D	40 - 49
F	0 - 39

The CAT competition

- At the end, there will be a competition between your CATs

- The team that designed the best CATs
  - 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 extend the industrial-strength compiler **clang** using their own advanced code analyses and transformations developed during this class. At the end of the class, the resulting compilers compete and the names of the students that designed and built the best compilers are reported below.

Year	Name	Picture
2018 - 2019	Vijay Kandiah	
2017 - 2018	Angelo Matni	

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 (via email to TA)**
    After a pair is formed, you can only split
    (no new pairs will be allowed; also, pairs cannot merge)
- No copying of code is allowed between pairs
- Tool, infrastructure help is allowed
  - 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: simone.campanoni@northwestern.edu

Summary

- My duties
  - Teach you code analysis and transformation
  - And how to implement them in a production compiler (LLVM)
- Your duties
  - Learn code analysis and transformation
  - Implement a few of them in LLVM
    - Write code
    - Test your code
    - Then, think **much harder** about how to **actually** test your code
    - (Sometimes) Answer my questions about your code

No final exam

Structure & flexibility

- CAT 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 → 12/8

Week 1

Today

- Welcome/Structure
- Compiler/CAT

→ F.E. → **M.E.** → B.E. →

Next lecture

LLVM

Outline of today's CAT

- Structure of the course
- CAT and compilers
- CAT and computer architecture
- CAT and programming language

The role of compilers

If there is no coffee, if I still have w
I'll keep working, I'll go to the coffe

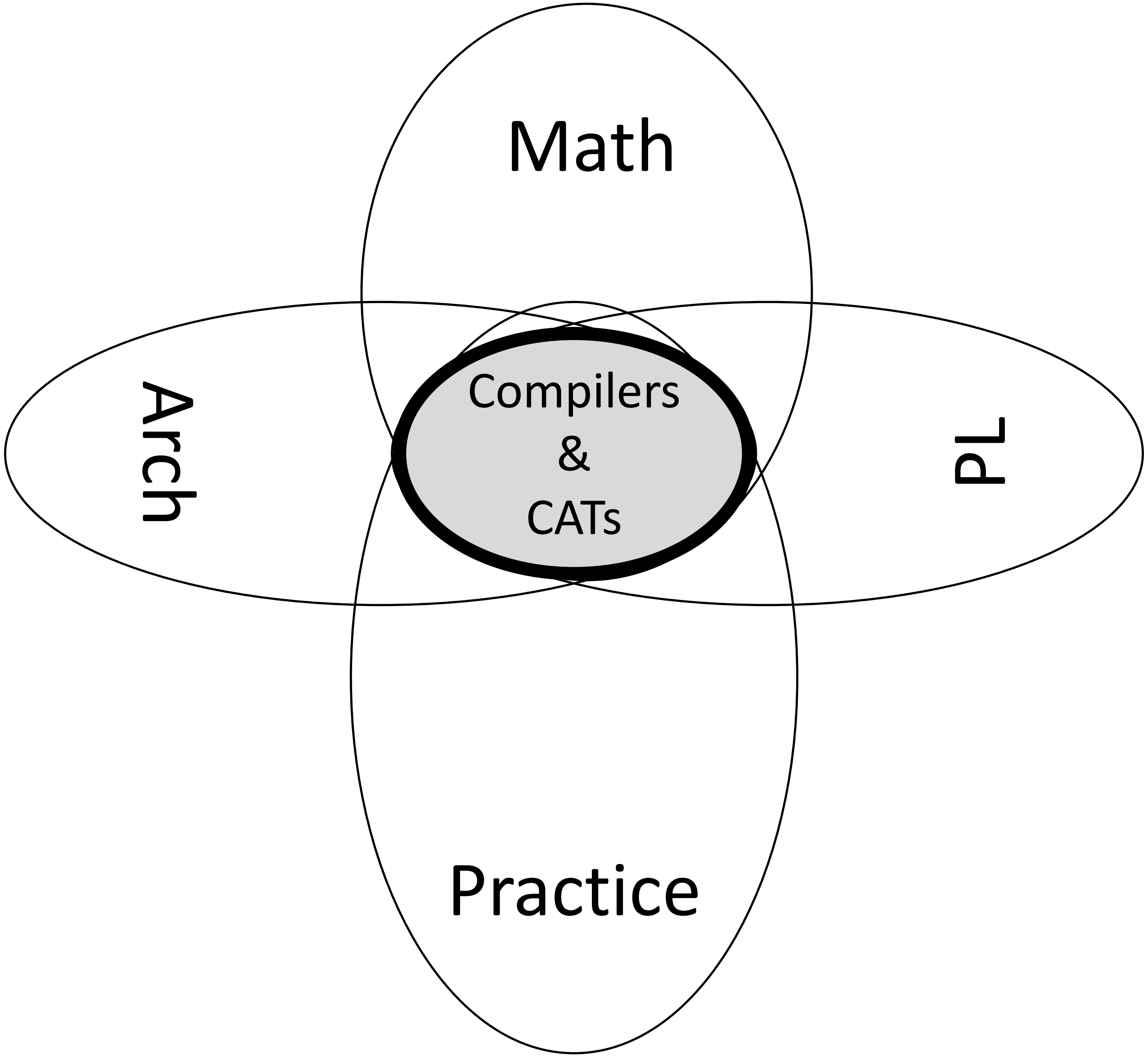
Will I go to the coffee shop when I have coffee?

e{
to do{
;
ee shop

Code analysis and transformation

???

00101010111001010101001010101011010


Math
Arch
Compilers
&
CATs
PL
Practice

Example of CAT

varX = 5

…

…

…

…

~~print varX~~

…

What will it print?

Example of CAT

varX = 5

…

…

…

…

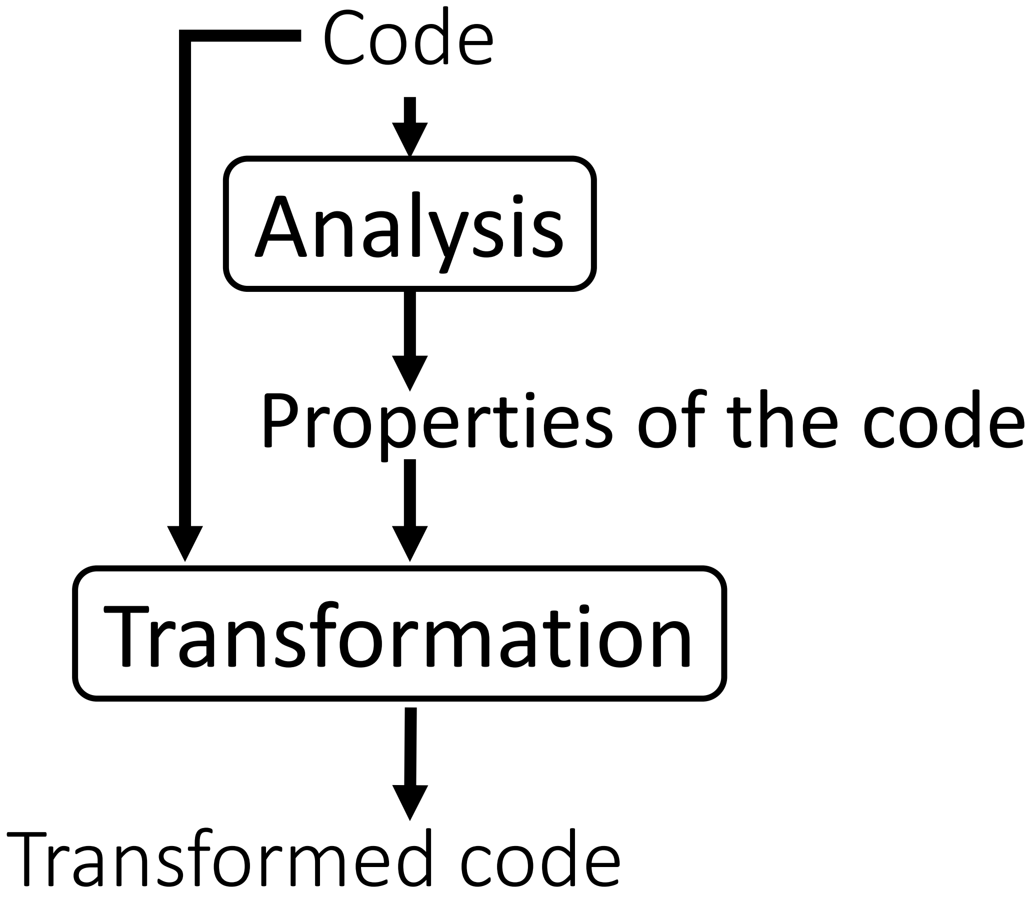
What will it print? print 5 ~~print varX~~

…

Example of CAT

```
varX = 5
    …
    …
    …
    …
print 5
    …
```

```
varX = 5
    …
    …
    …
    …
print varX
    …
```



Designing CATs

- Choose a goal
  - Performance, energy, identifying bugs, discovering code properties, …
- Design automatic code analyses to obtain the required information
- Occasionally design code transformations

Use of CATs

- Compilers
  - Increase performance
  - Decrease energy consumption
  - Decrease code size
  - Drive the code translation

- Developing tools (e.g., VIM, EMACS)
  - Understanding code (e.g., scopes, variables)
  - Generate suggestions

- Computer architecture

Structure of a compiler

Character stream (Source code)

↓

Lexical analysis

↓

Tokens

↓

Syntactic & semantic analysis

↓

AST

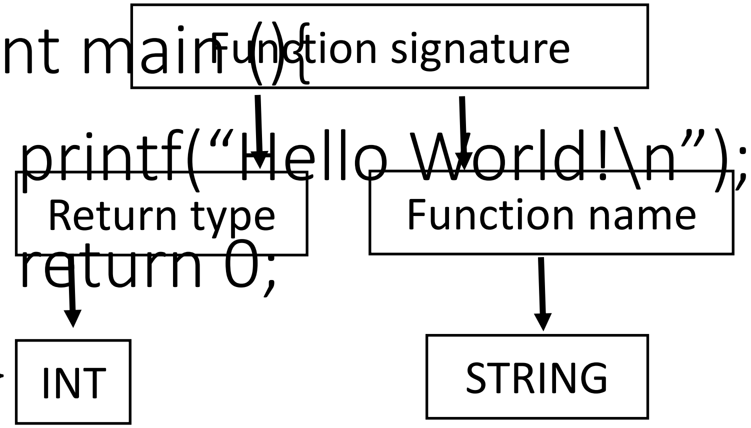
i	n	t		m	a	i	n	

…

INT	SPACE	STRING	SPACE

…

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



Structure of a compiler

Character stream (Source code)

↓

Lexical analysis

↓

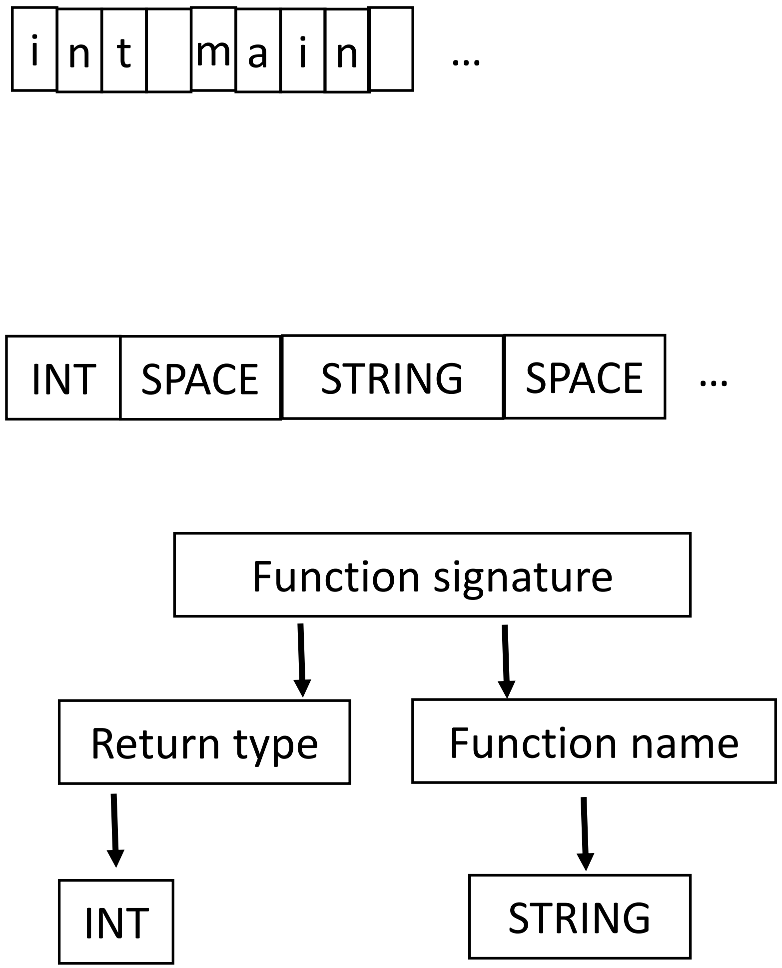
Tokens

↓

Syntactic & semantic analysis

↓

AST



Structure of a compiler

Syntactic & semantic analysis

↓

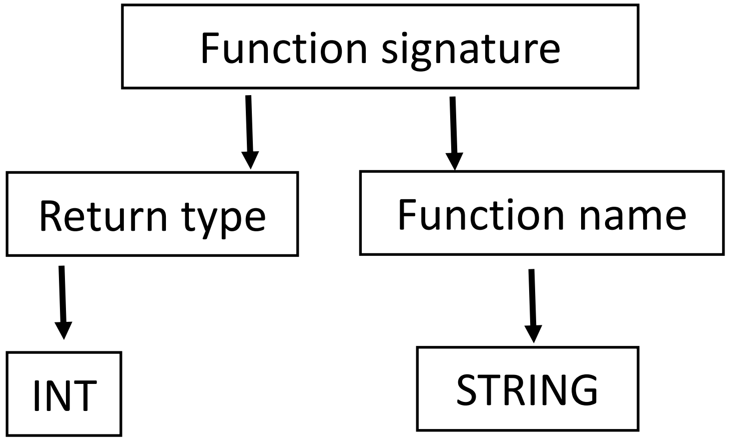
AST

↓

IR code generation

↓

IR




```
; Function Attrs: nounwind uwtable
define int @main() {
```

Structure of a compiler

Character stream (Source code)

i	n	t		m	a	i	n		…

↓

Front-end — ***CS 322: Compiler Construction***

↓ IR ↓

```
; Function Attrs: nounwind uwtable
define int @main() {
```

Middle-end — ***Code analysis and transformation***

↓ IR ↓

```
; Function Attrs: nounwind uwtable
define int @main() {
```

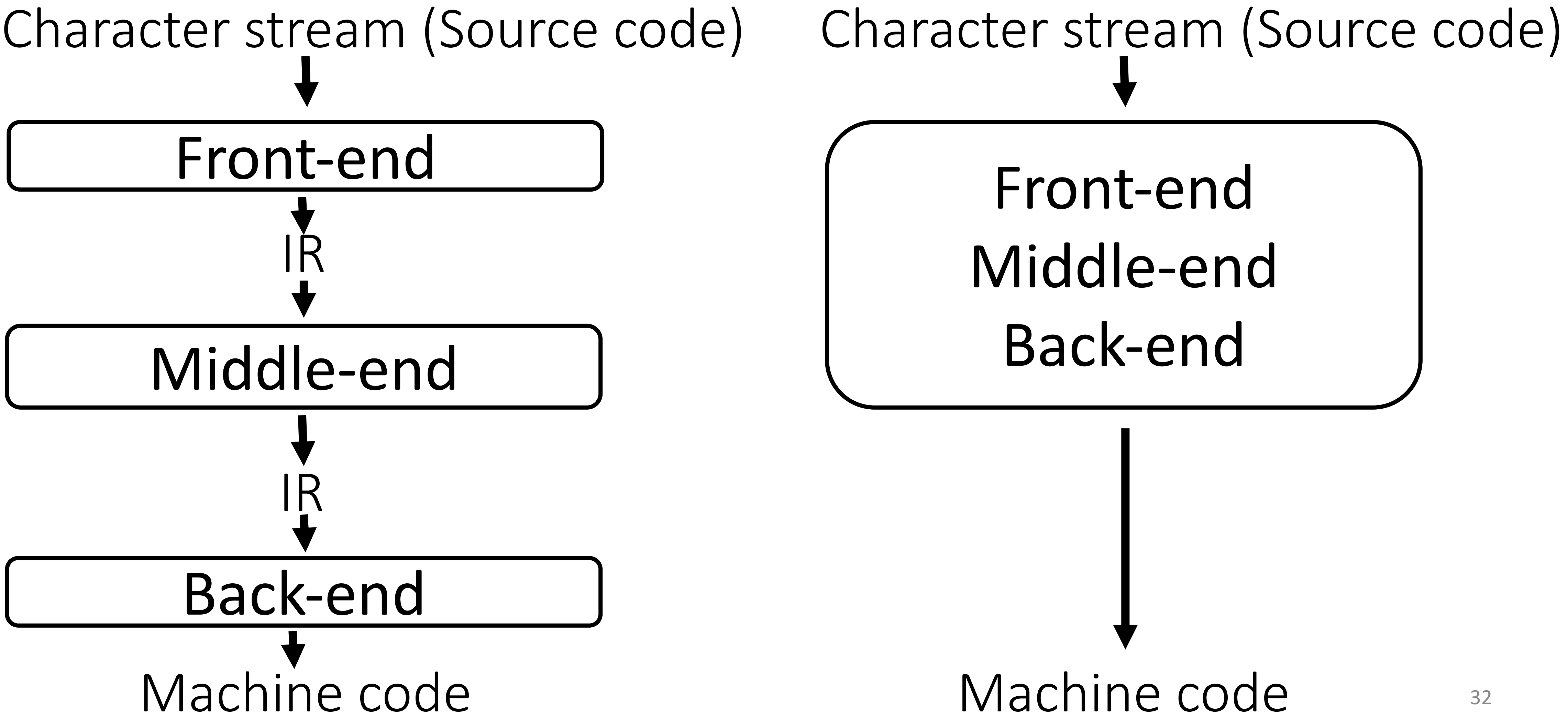
Back-end — ***CS 322: Compiler Construction***

↓

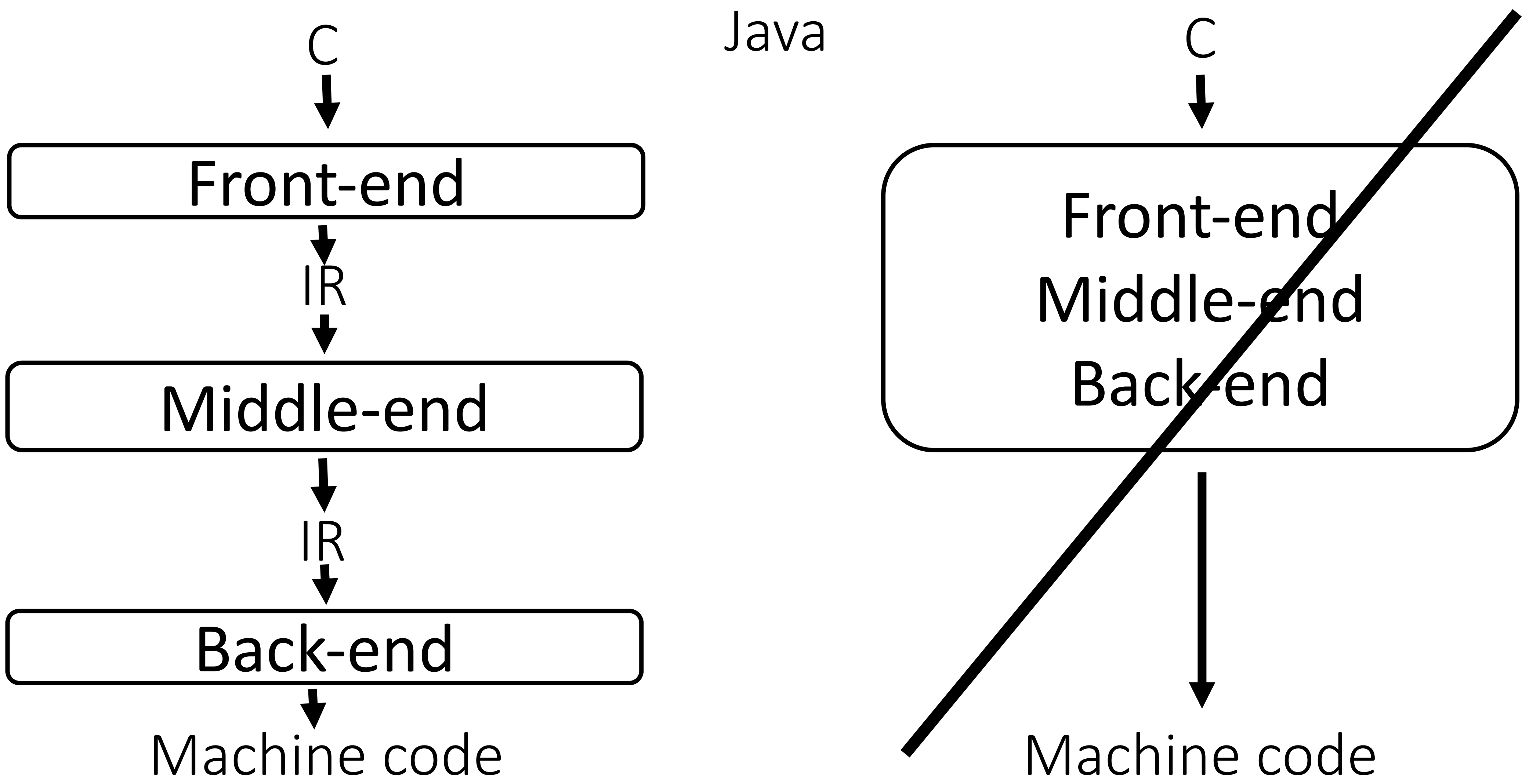
Machine code

01010111010101010101

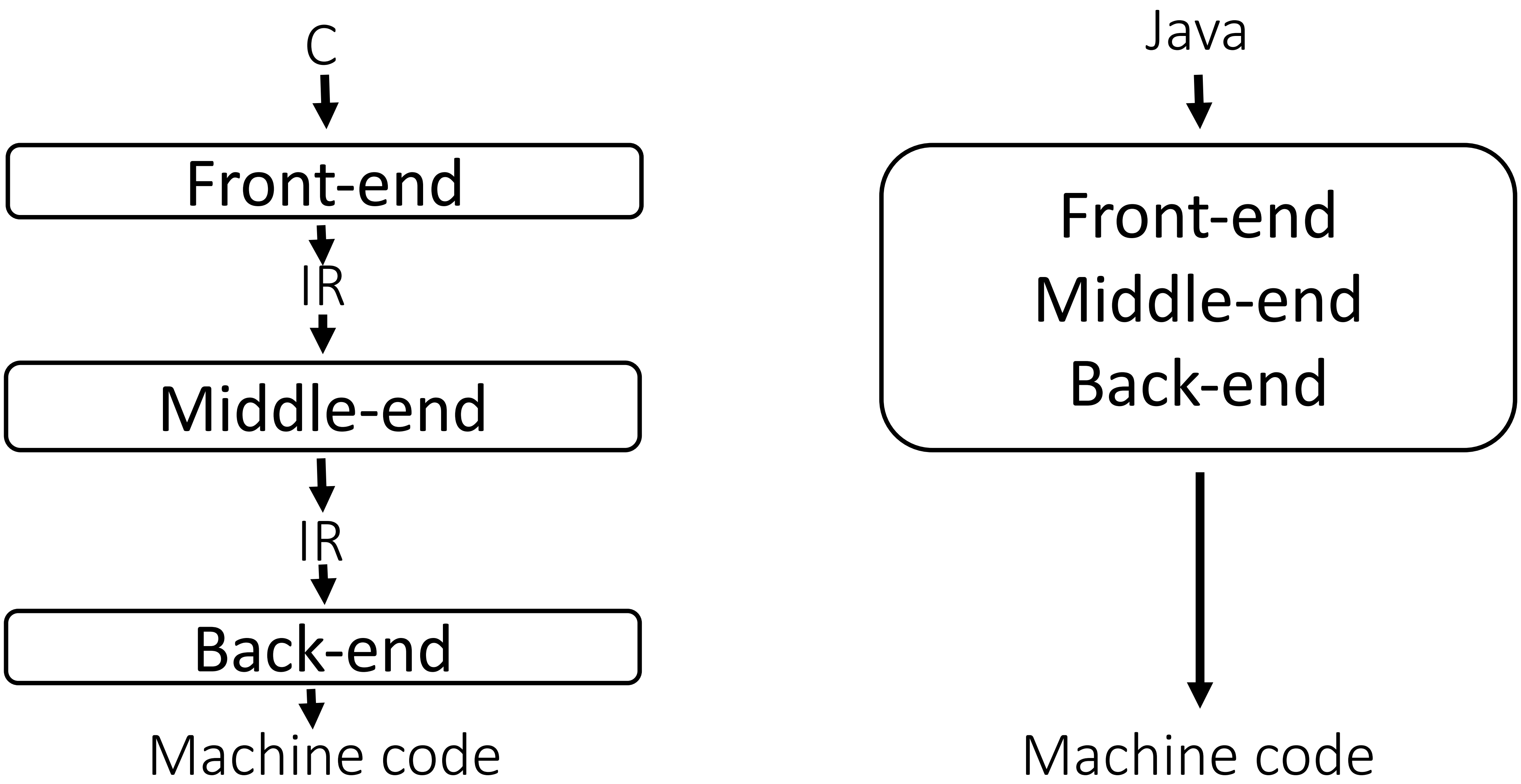
Structure of a compiler



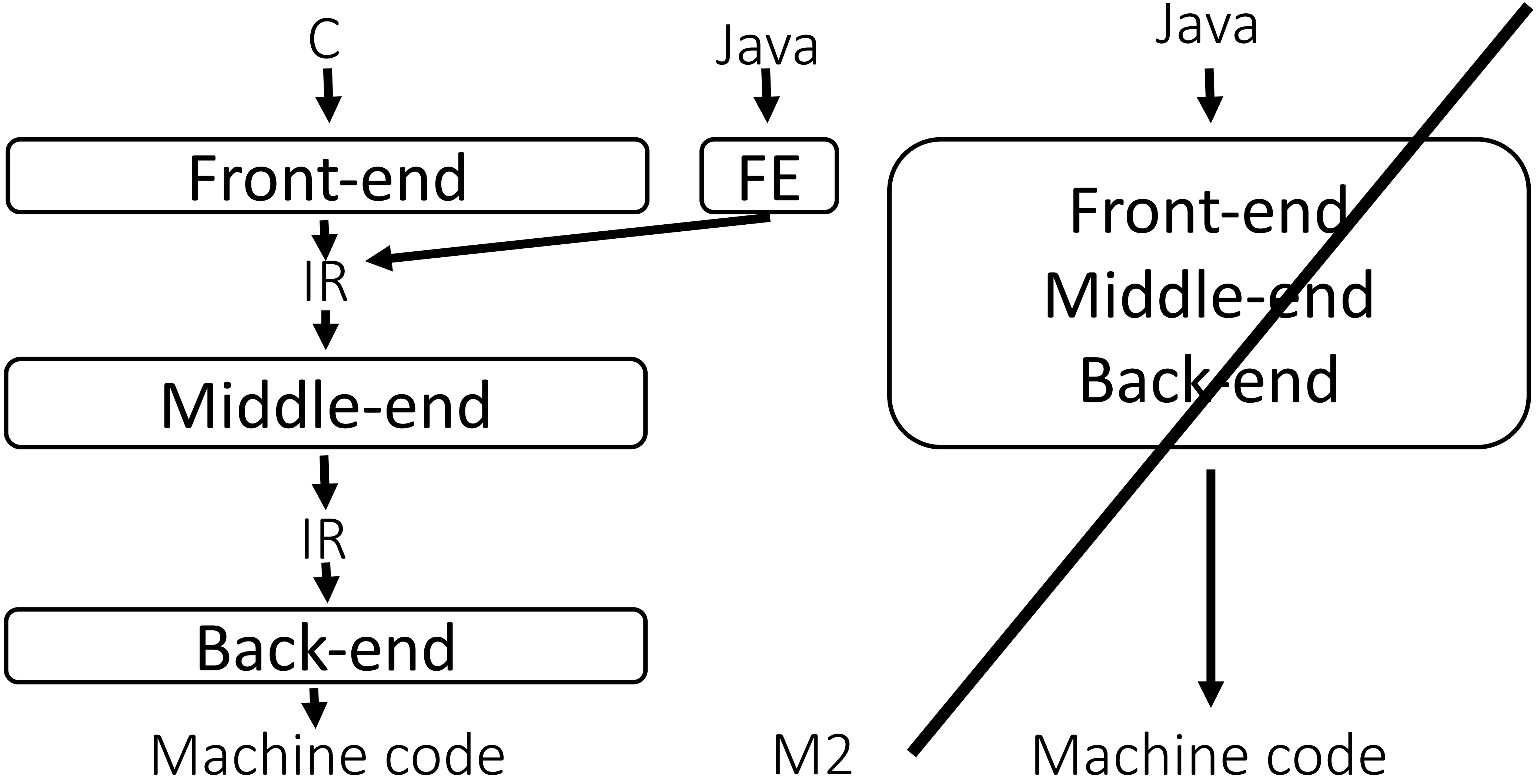
Structure of a compiler



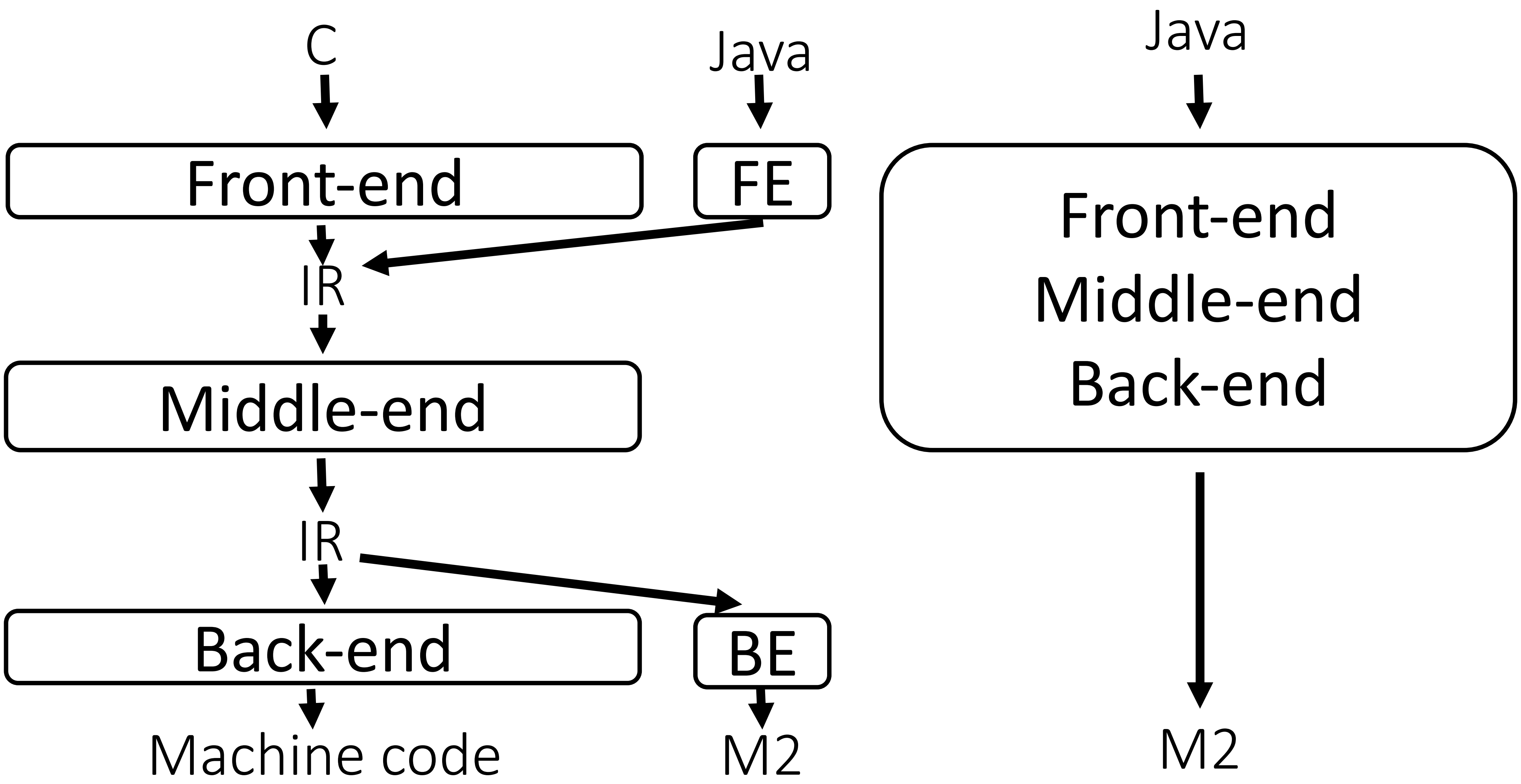
Structure of a compiler



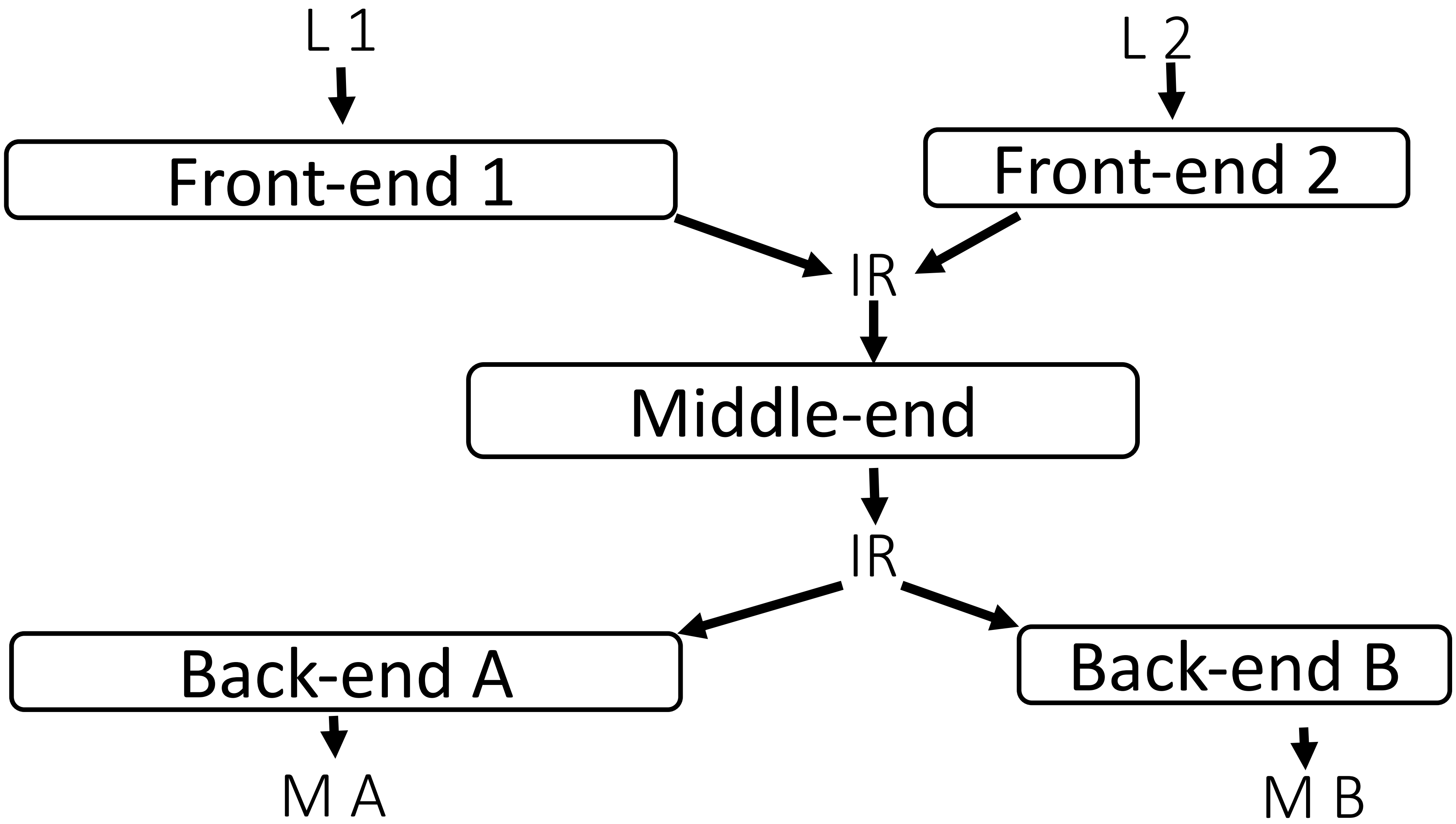
Structure of a compiler



Structure of a compiler



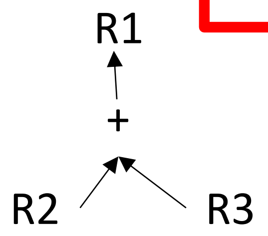
Structure of a compiler



Multiple IRs

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

- Abstract Syntax Tree



- Register-based representation (three-address code)

R1 = R2 add R3

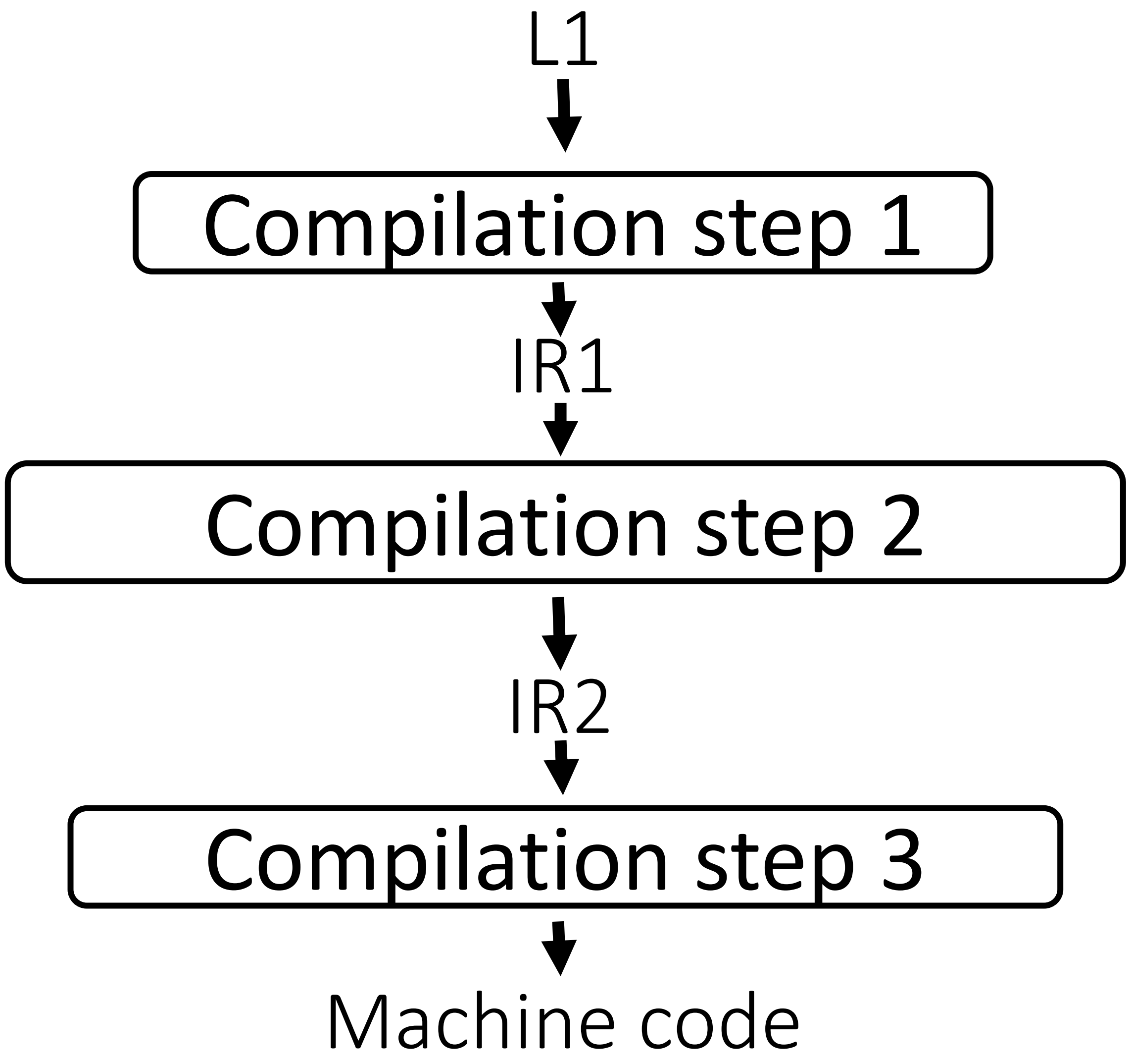
- Stack-based representation

push 5; push 3; add; pop ;

Example of LLVM IR

```
define i32 @main(i32 %argc, i8** %argv) {
entry:
  %add = add i32 %argc, 1
  ret i32 %add
}
```

Multiple IRs used together



Multiple IRs used together

Rust

↓

Compilation step 1

↓

MIR

↓

Compilation step 2

↓

LLVM IR

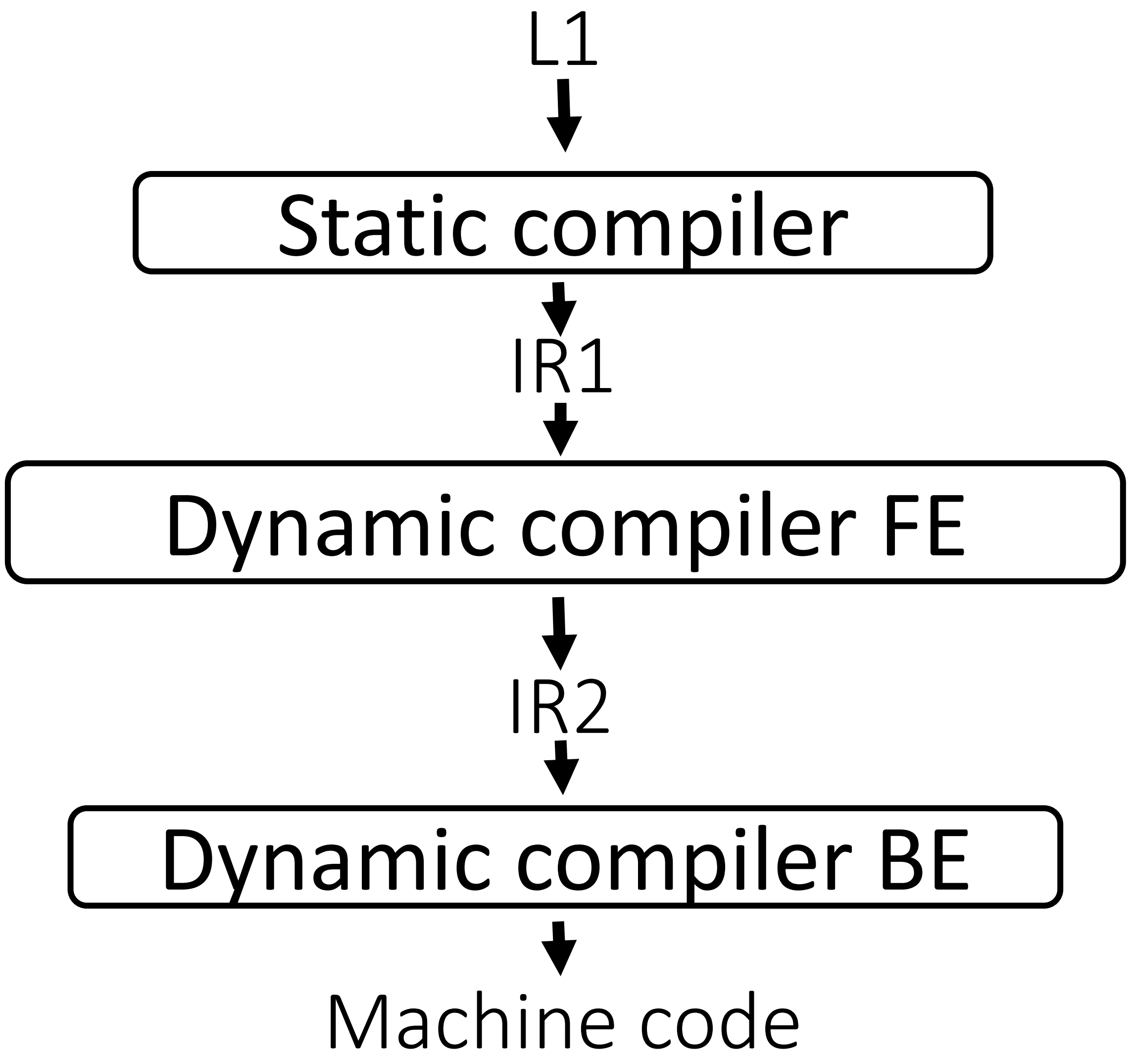
↓

Compilation step 3

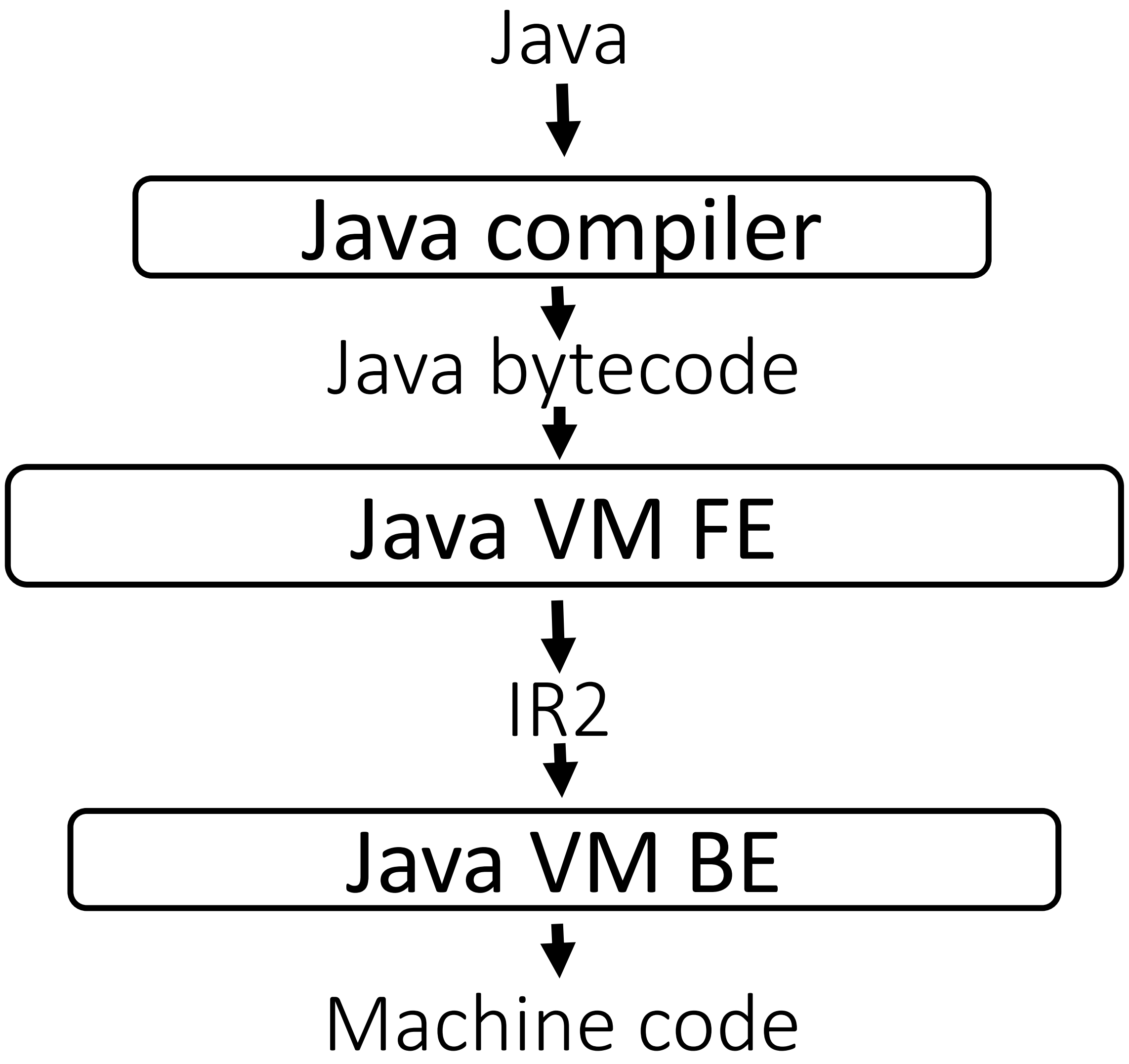
↓

Machine code

Multiple IRs used together



Multiple IRs used together



CATs that we'll focus on

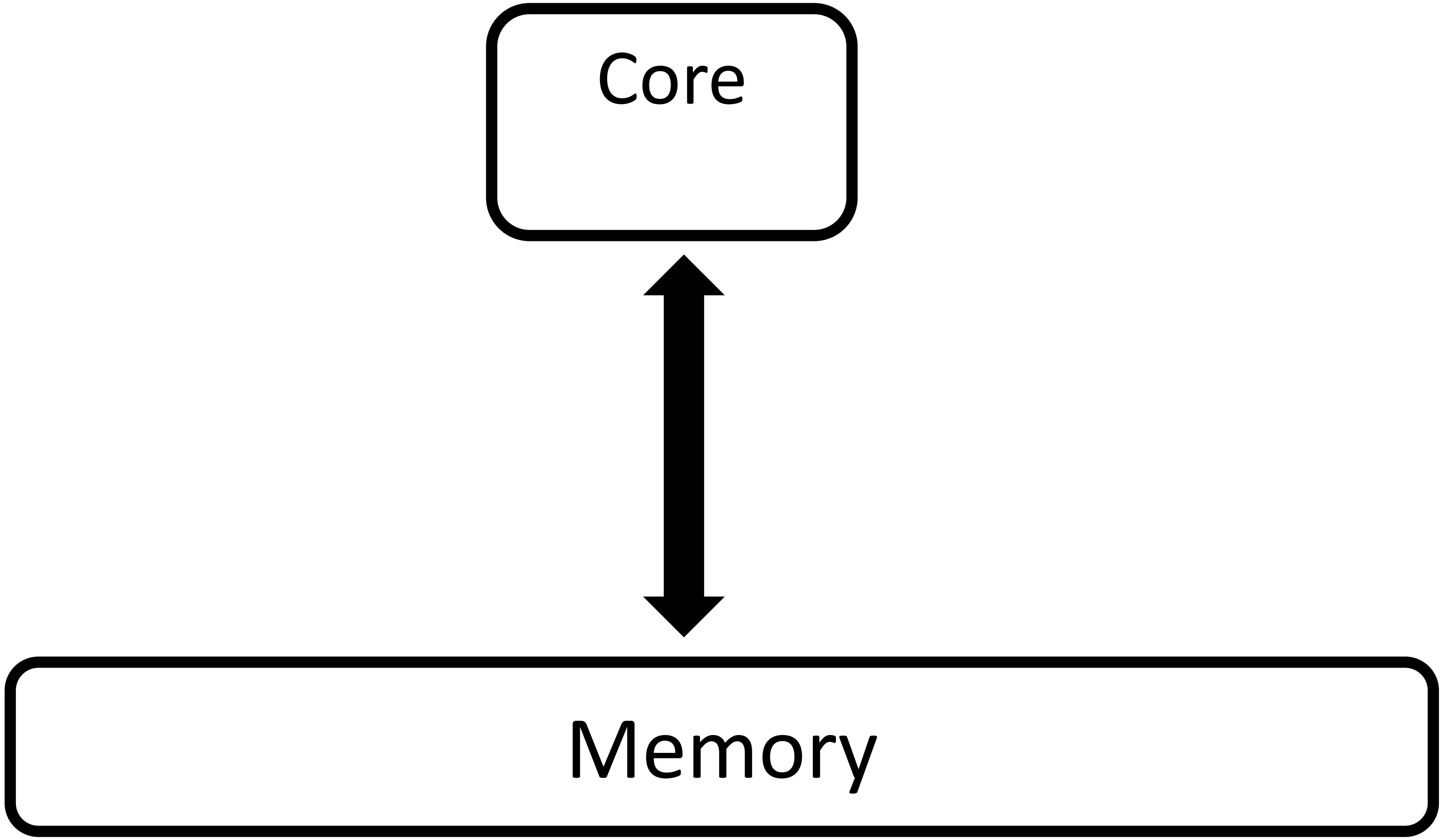
- Semantics-preserving transformations
  - Correctness guaranteed
- Goal: performance
- Automatic
- Efficient

Outline of today's CAT

- Structure of the course
- CAT and compilers
- CAT and computer architecture
- CAT and programming language

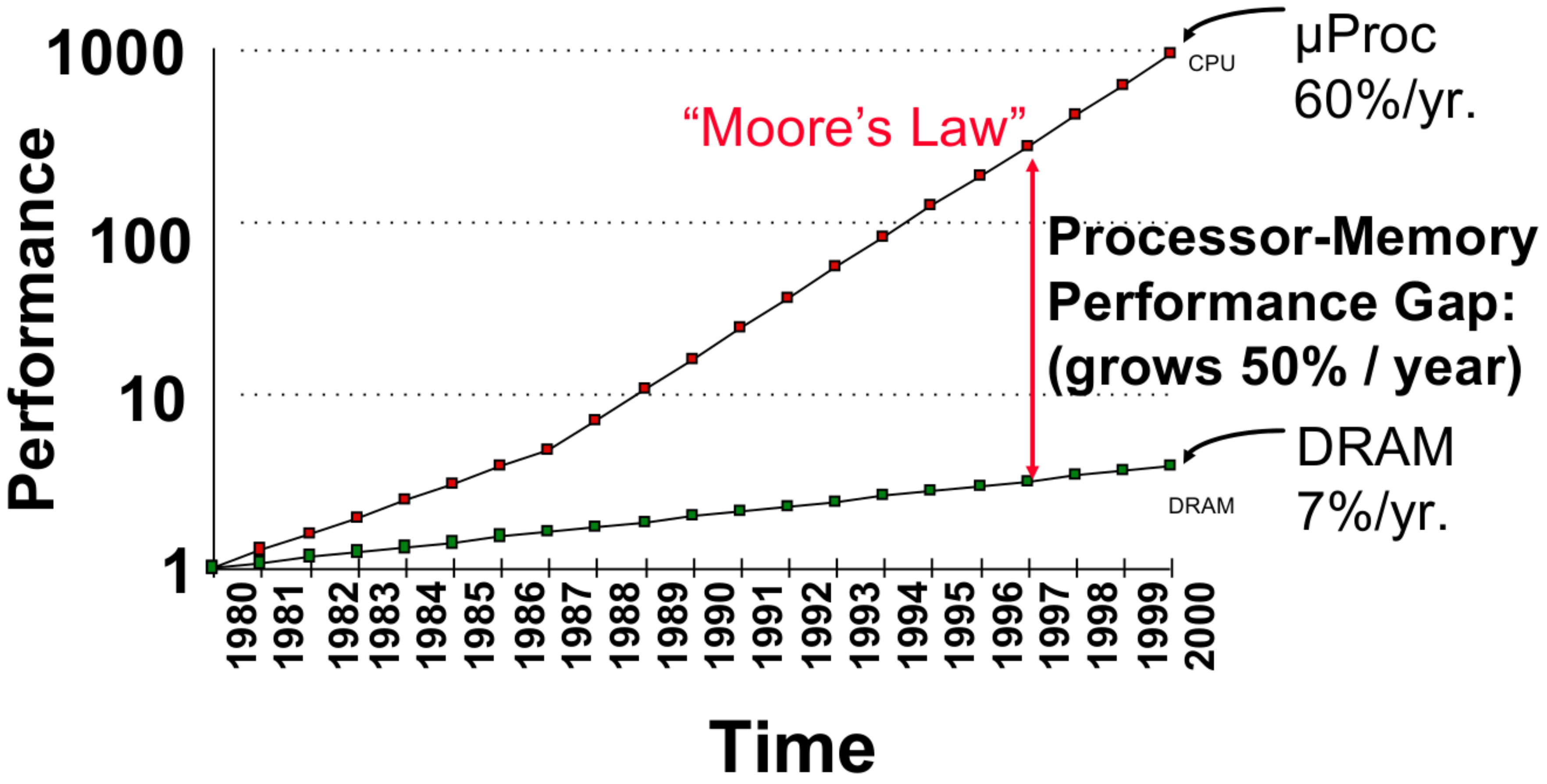
Evolution of CATs (hardware point of view)

- Simple hardware (few resources), simple CATs



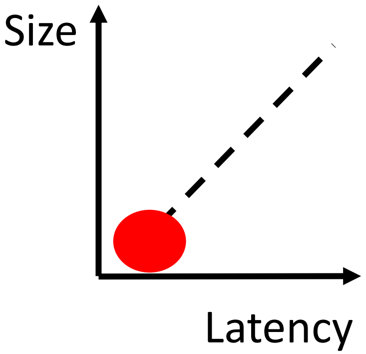
Evolution of CATs (hardware point of view)

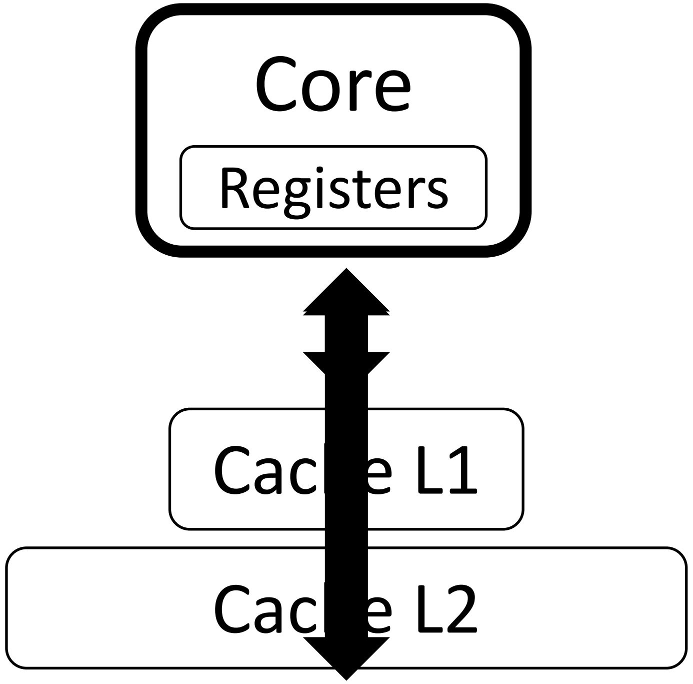
- Simple hardware (few resources), simple CATs

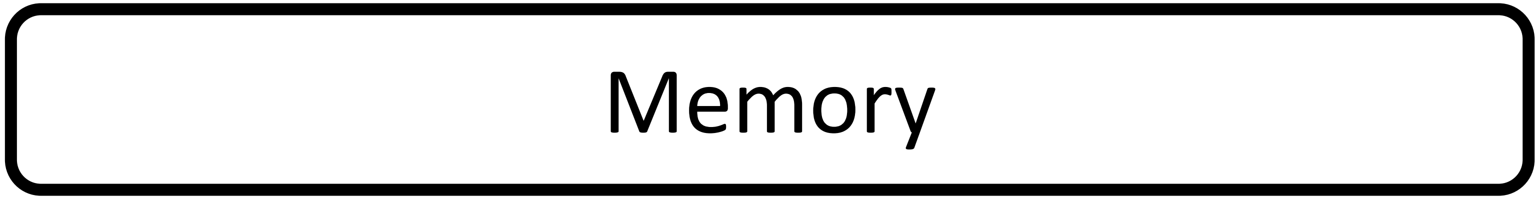


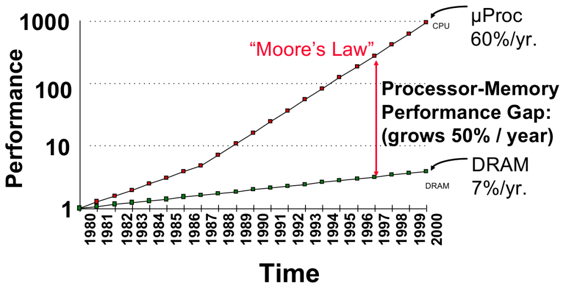
Evolution of CATs (hardware point of view)

- Simple hardware (few resources), simple CATs









Evolution of CATs (hardware point of view)

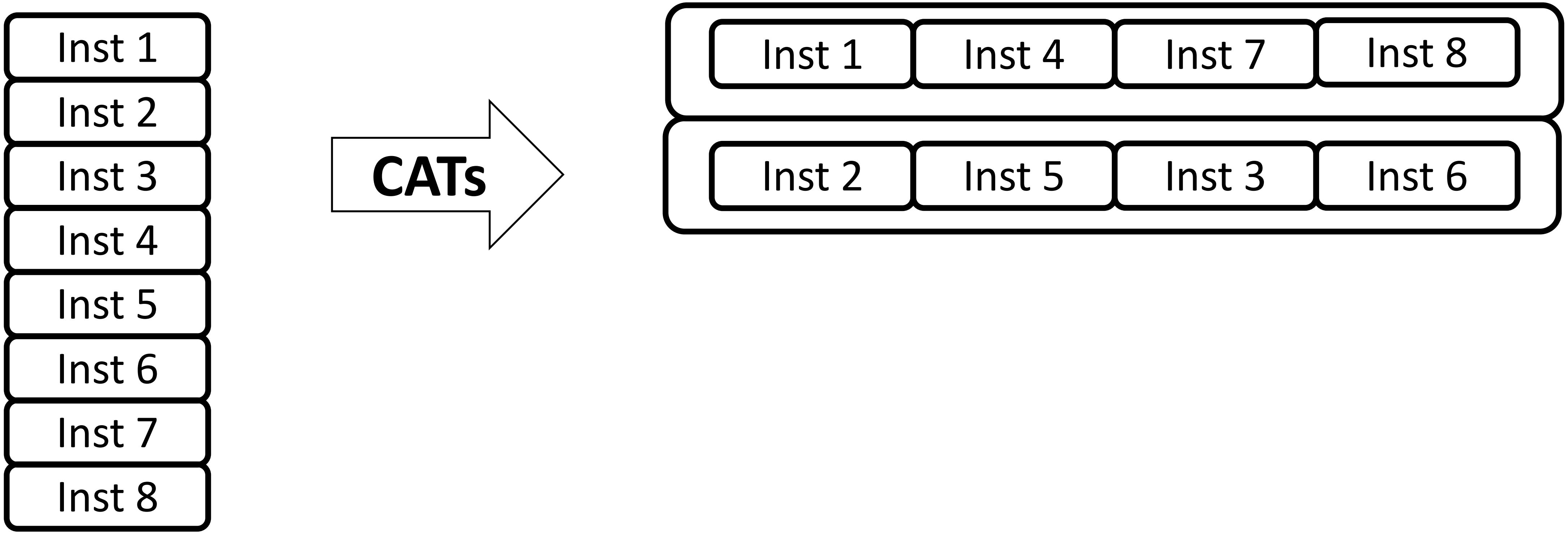
- Si
- M
- E
  sc
  - Challenging CATs

Compilers/CATs
are considered
in the processor-design stage!

Evolution of CATs (hardware point of view) (3)

Superscalar

Very long instruction word (VLIW)



Outline of today’s CAT

- Structure of the course
- CAT and compilers
- CAT and computer architecture
- CAT and programming language

Evolution of CATs (PL point of view)

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

00101010111001010101001010101011010

- 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 CATs (PL point of view)

- Low level programming language, simple CATs
  - Not very productive
- More abstraction in programming language,
  more work for CATs to reduce their performance overhead
  - Macros -> Fortran, Cobol, Lisp -> C, C++, Java, C#, Python, PHP, SQL, …
- **CATs enable new programming languages**

Evolution of CATs (PL point of view)

- Abstractions are great for productivity
- CATs remove their overhead
- But abstractions must be carefully evaluated considering CATs
- **A simple abstraction in PL can generate challenges for CATs**
  - **CATs need to be understood**

Evolution of CATs (PL point of view)(2)

PL without procedures

```
void main (){
  Int v1,v2;
  v1 = 1;
  v2 = 2;
  …
}
```

Evolution of CATs (PL point of view)(3)

Let's add procedures to our PL

```
void myProc (int *a, int *b){…}
myProc(&myVar1, &myVar2);
```

Evolution of CATs (PL point of view)(2)

```
void myProc (int *v1, int *v2){
  (*v1) = 1;
  (*v2) = 2;
}
```

What's the problem for CATs? … if v1 and v2 alias …

Understanding if pointers alias: pointer alias analysis

This is one of the most challenging problem in CATs

Conclusion

- CATs used for multiple goals
  - Enable PLs
  - Enable hardware features

- CATs are effected by
  - Their input language
  - The target hardware

- When you design a PL or a new hardware platform, you need to understand what CATs **can** and **can’t** do
  - Often: a **can’t** becomes **can** thanks to research on CATs

Ideal CATs

- Proved to be correct
- Improve performance of many important programs
- Minor compilation time
- Negligible implementation efforts

Code transformations

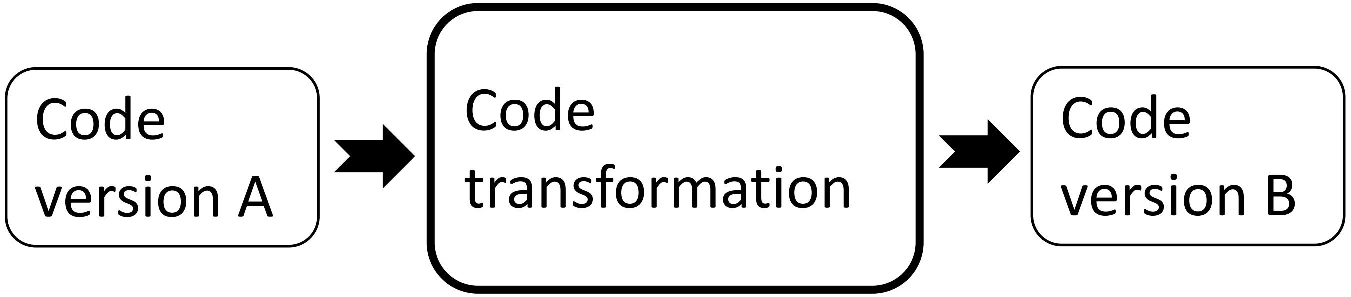
- Conventional transformations:
  they preserve the original program semantics
  - These are the transformations that are included in commodity compilers (e.g., gcc, clang, icc)

- In this class, we only consider this type of code transformations

Code transformation

Code transformation:

An algorithm that
takes code as input and it generates new code as output



Semantically-preserving code transformation:

A code transformation that **always** generates code that is **guaranteed** to have the **same semantics** of the code given as input.

What is the program semantics?

Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input

```
int main (
      int argc, char *argv[]
      ){
int x = argc;
int y = x + 1;
y++;
printf("%d", x + y);
return 0;
}
```

=

```
int main (
      int argc, char *argv[]
      ){
int y = argc + 2;
printf("%d", argc + y);
return 0;
}
```

≠

```
int main (
      int argc, char *argv[]
      ){
int y = argc + 2;
printf("%d", 2*argc + 3);
return 0;
}
```

Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input

```
int main (
      int argc, char *argv[]
      ){
int y = argc + 2;
printf("%d", 2*argc + 2);
return 1;
}
```

```
$ ./myprog 2
6
$ echo $?
```

```
int main (
      int argc, char *argv[]
      ){
int y = argc + 2;
printf("%d", 2*argc + 2);
return 0;
}
```

Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input

```
int main (
    int argc, char *argv[]
    ){
int y = 42;
return y;
}
```

→ Our new code transformation →

```
int main (
    int argc, char *argv[]
    ){
int y = 42;
return 42;
}
```

We have preserved the semantics of the original code!

Program semantic

Program semantic: Input -> Output

Two programs, p1 and p2, are semantically equivalent if for a given input, p1 and p2 generate the same output for every possible input

Our transformation needs to understand how the execution flows through the instructions to preserve the semantics!

```
int main (
      int argc, char *argv[]
      ){
int y = 42;
int x = y;
if (argc > 20)
   y = 81;
return x + y;
}
```

Our new code transformation

```
int main (
      int argc, char *argv[]
      ){
int y = 42;
int x = 42;    <- This is ok!
if (argc > 20)
   y = 81;     <- When this is executed
return x + 42;
}
```

This is ok!

When this is executed

We haven't preserved the semantics of the original code

As Linus Torvalds says …

Talk is cheap. Show me the code.

Demo time

Always have faith in your ability

Success will come your way eventually

Best of luck!