COMP 520 - Compilers

Lecture 20 – Virtual Methods, Polymorphism, Bootstrapping, and a look into C#

Announcements

- Please do course evaluations!
- Final Exam is 5/9 at 4:00pm
- The exam is written to be taken in 90 minutes, but I'm going to give you the full 180 minutes should you desire it.

Final Exam

- If you are in the situation of “3 exams in 24 hours”, make sure you follow protocol and let the Dean’s office know so that way we can get your exam rescheduled.

- Once again: Final Exam is 5/9 at 4:00pm

Bootstrapping

Bootstrapping is a topic of initialization. How does that relate to Compilers?

Bootstrapping

- Generally related to initializing hardware or software
- In compilers, bootstrapping is similar to what you have been doing with `miniJava`!

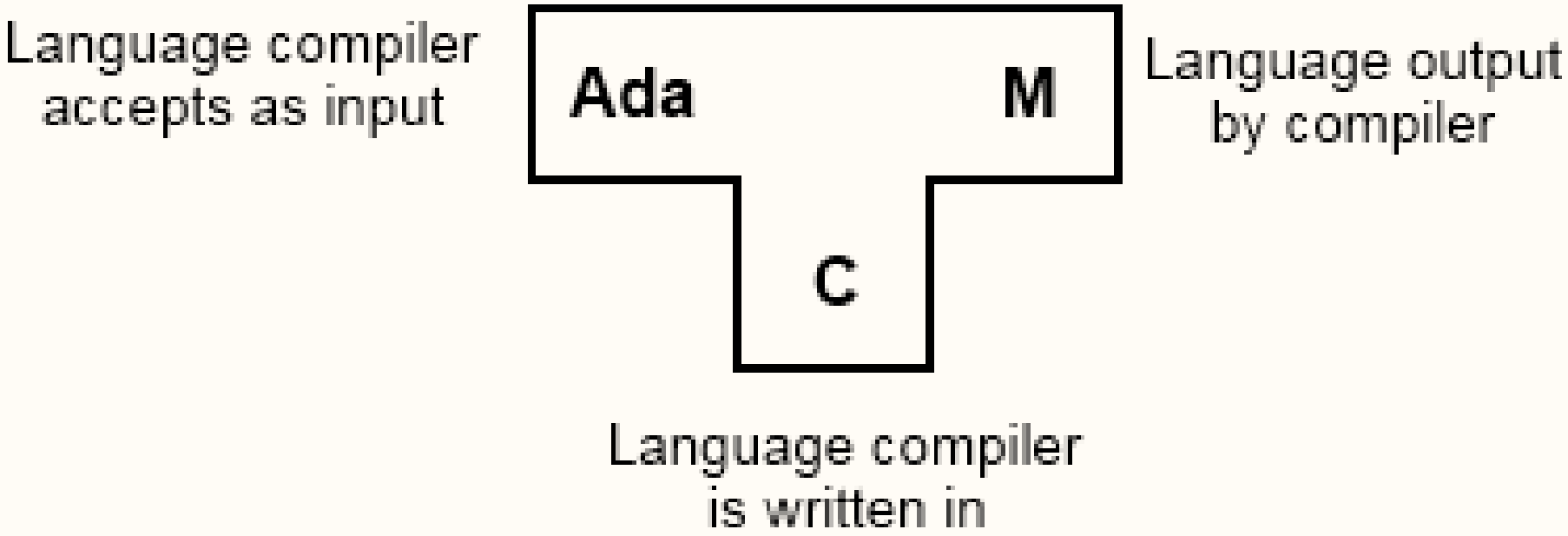
- Consider new hardware with a new ISA "LEG." You are writing the compiler for it. **Full** Programming languages are very complex, so does that mean you need to write all of miniJava *AND MORE* entirely using LEG assembly code?

Bootstrapping (2)

- Idea: Only target a **small part** of your programming language.
- Then, when the compiler is done, *compile a compiler* with the "easier to use" mini-Language.

- Add features that are missing in the mini-Language, using an easier to read language!

Tombstone Diagram (T diagram)
The T-shape is a "Translator"



Let’s create the “first” compiler!

Arguable usage of the word ‘first’

Goal: Create a compiler

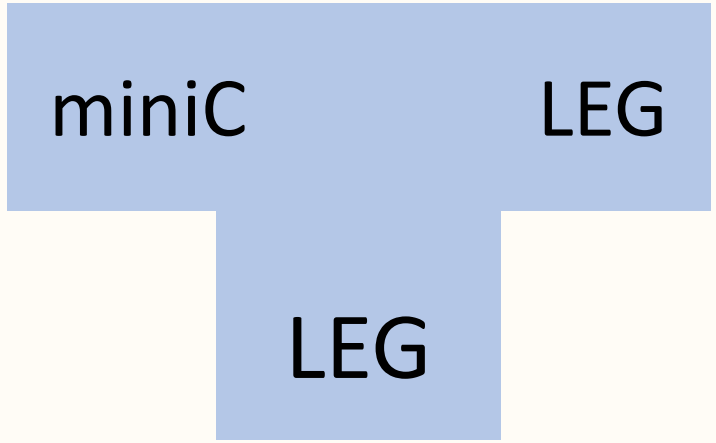
- New architecture just dropped, it's called **LEG**
- A more powerful version of ARM with some tradeoffs in fine-grain instructions

- Our goal: Make a fully functional C compiler for LEG

Making the first compiler for LEG

Language Accepted

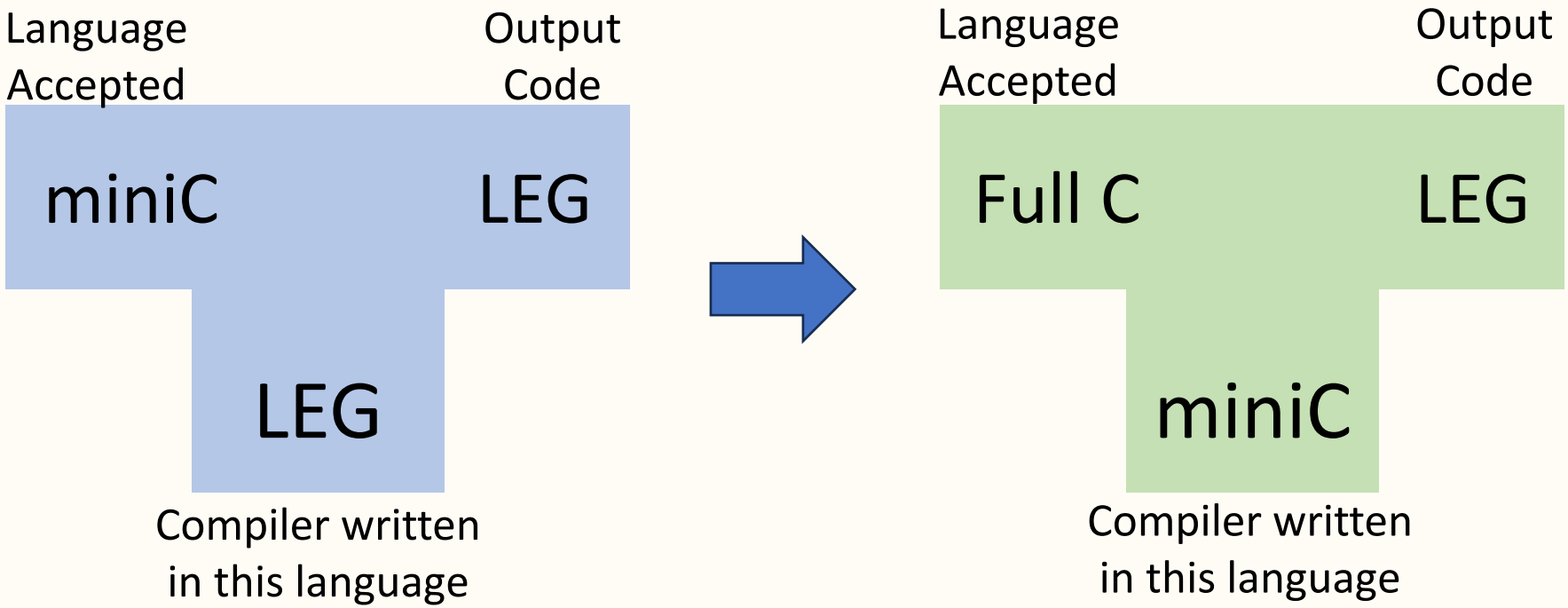
Output Code



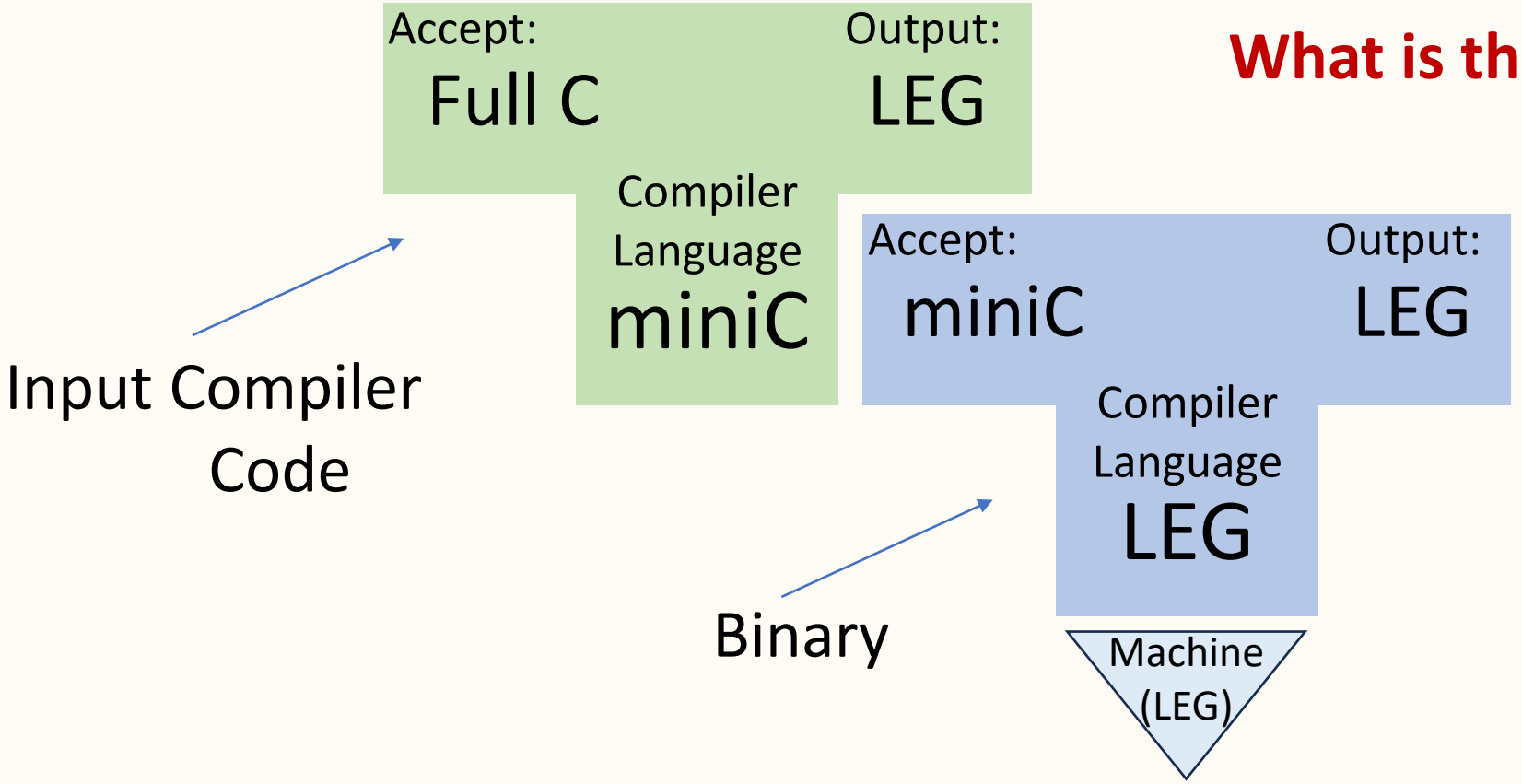
Compiler written in this language

I.e., Write a miniC compiler in x86 for the x86 processor.

Making the second compiler for LEG

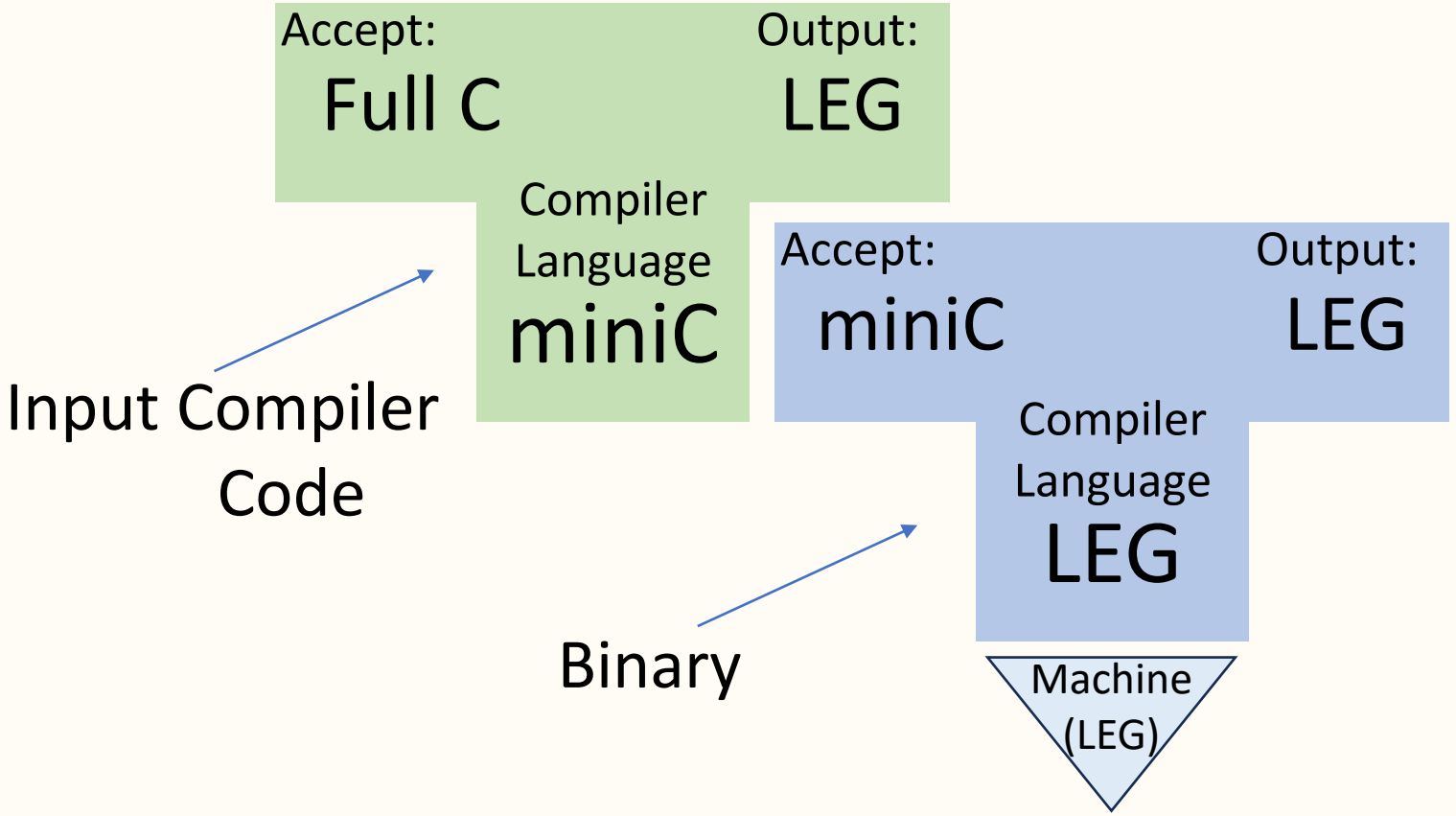


Compiler books really like these diagrams



What is this diagram saying?

Compiler books really like these diagrams

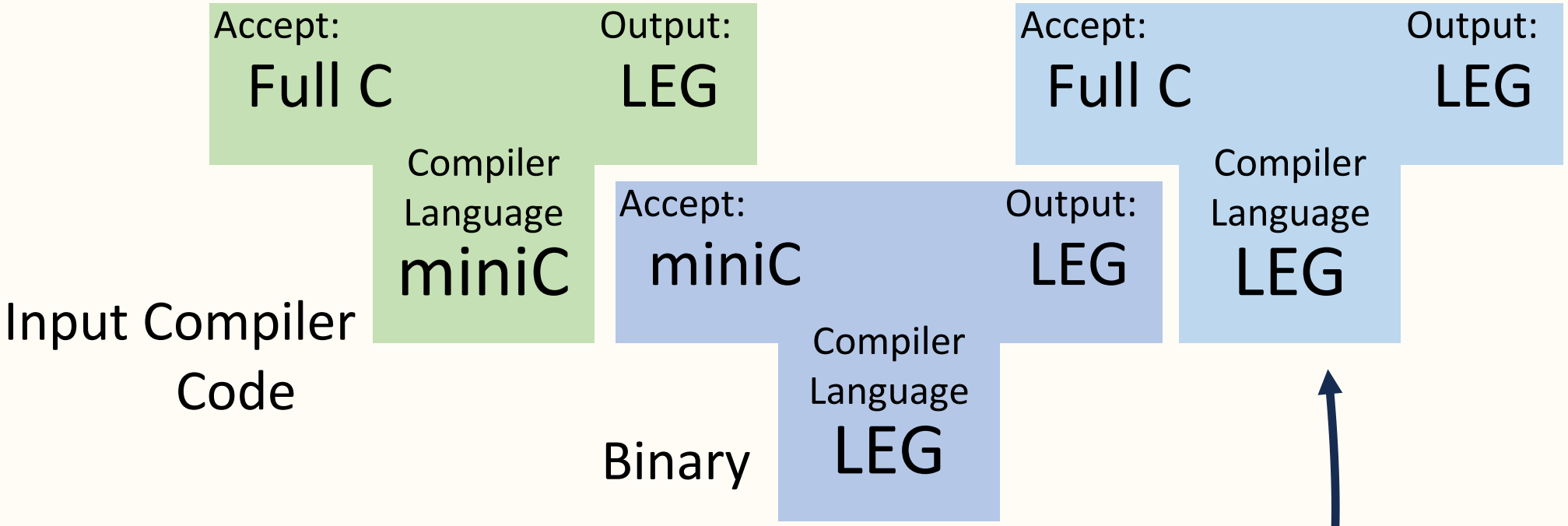


What is this diagram saying?

It says:

I use a compiler painfully written in LEG, which accepts miniC, and use miniC to write a compiler for Full C. Note: It is easier to use miniC than LEG assembly code (first compiler)

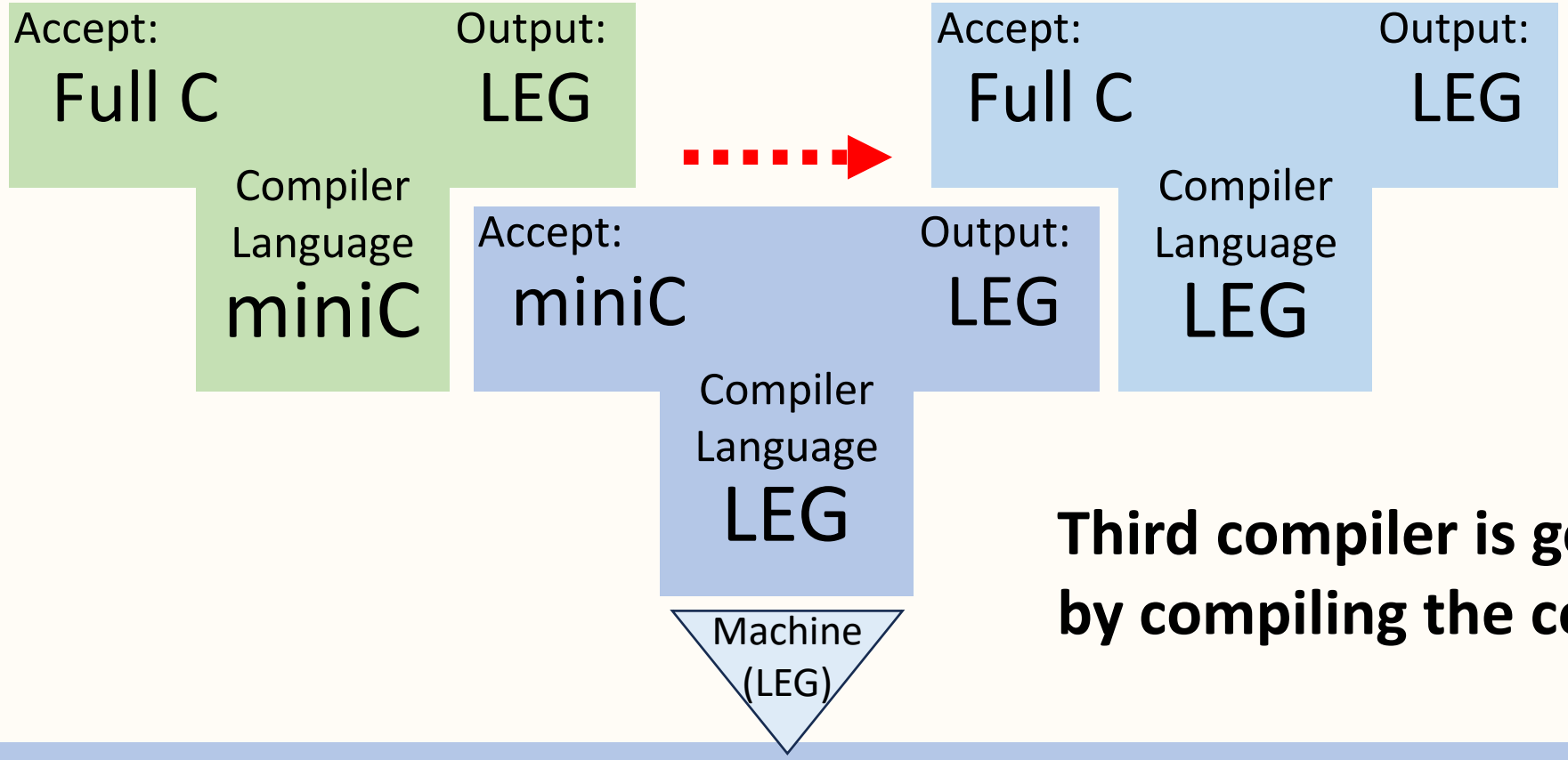
Third Compiler



This is our goal, a compiler that can run on "LEG" machines, but accepts the Full C language.

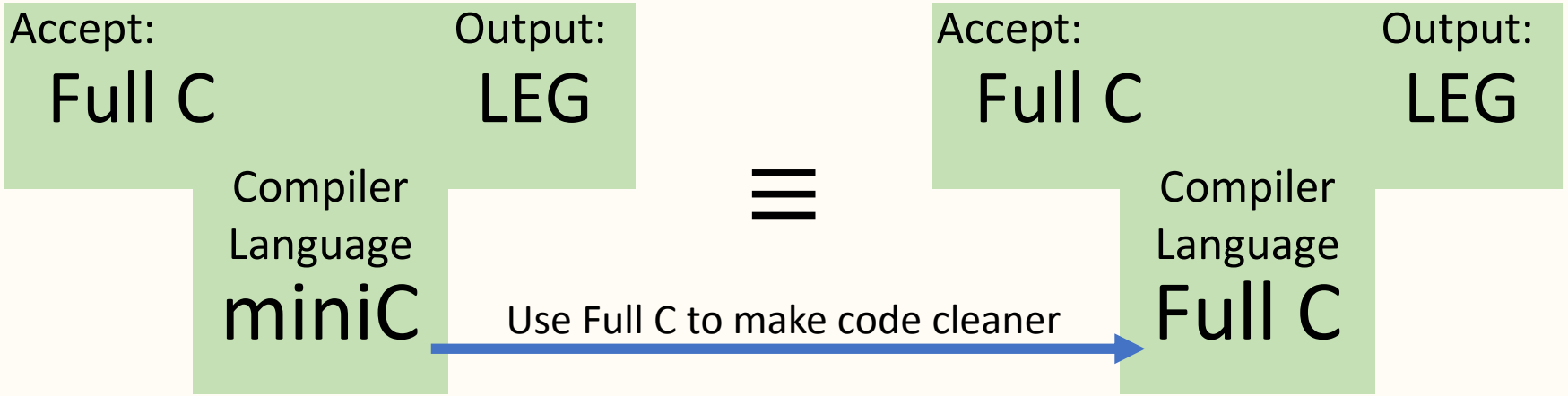
How can we create this?

Completed Diagram: How to go from miniC to Full C



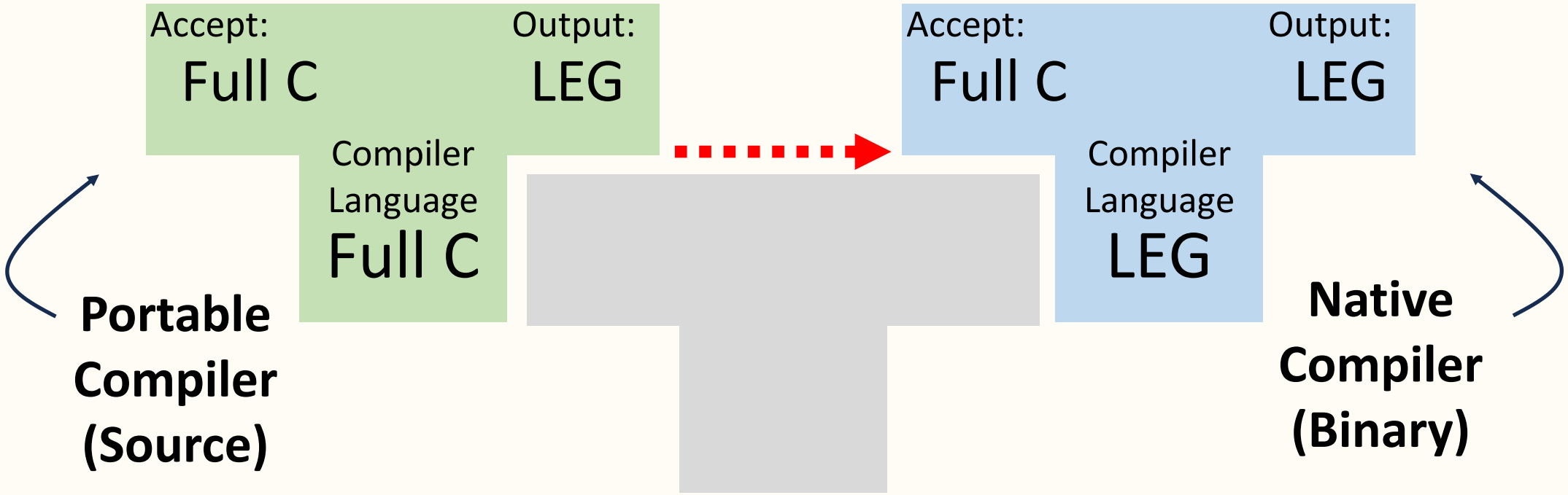
Third compiler is generated by compiling the compiler!

Because miniC ⊂ FullC



- Optionally, can also clean up miniC code to use Full C, but the ***compiler functionality*** remains the same.

Naming Compilers



Next Goal: Retarget Output

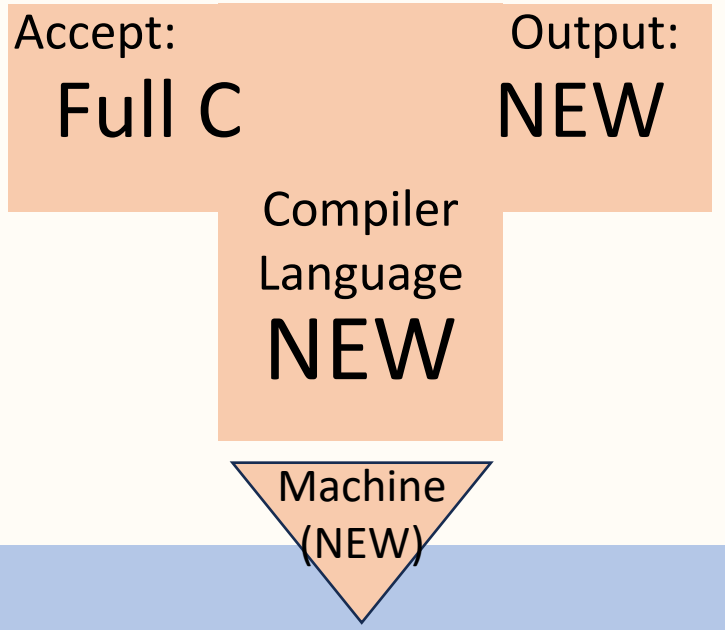
- Consider: A new ISA extends **LEG**, it supports WIDE sizes of operands and more *flexibility*, called KNEE-W, or **NEW** for short.

- Goal: **Retarget** existing compiler for outputting to **NEW** bytecode rather than the legacy **LEG** code.

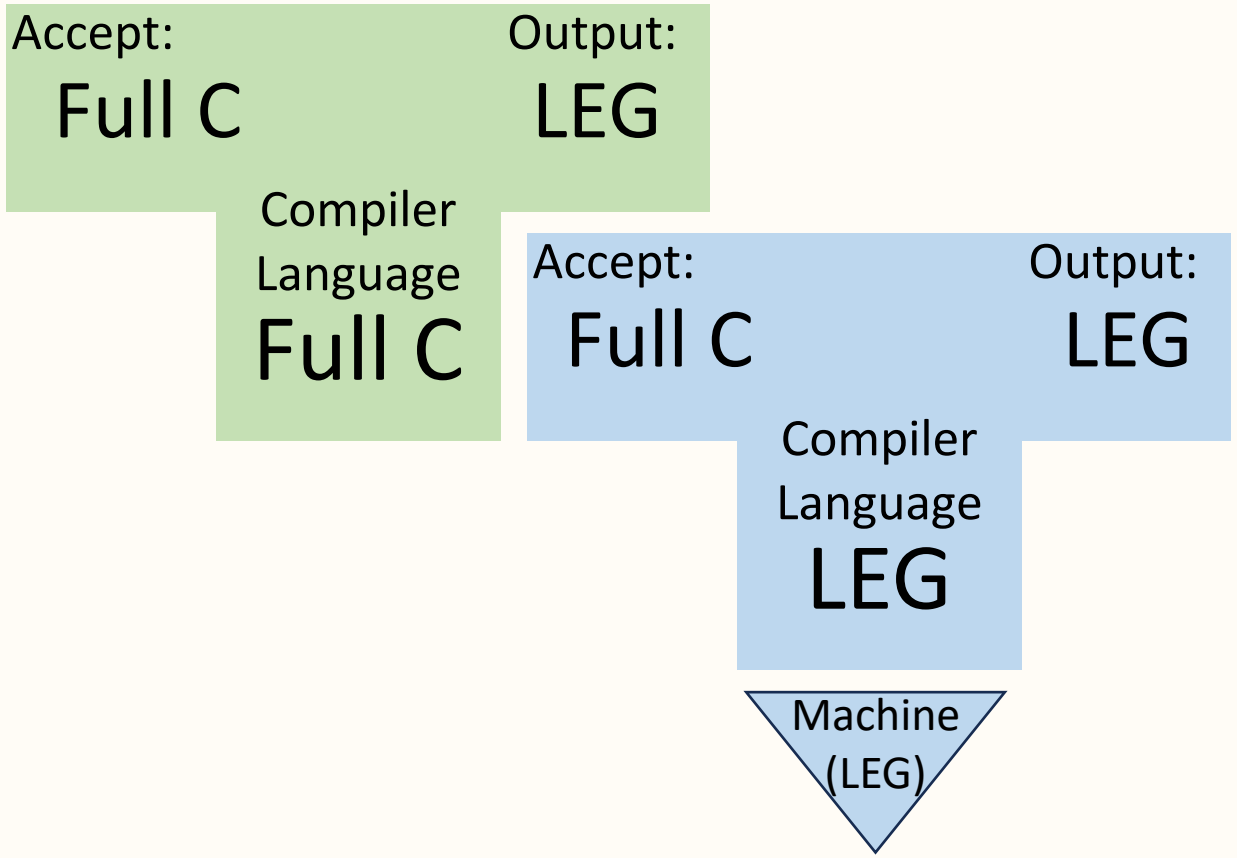
Next Goal: Retarget Output

- Goal: Retarget existing compiler for outputting to NEW bytecode rather than the legacy LEG code.
- We want our compiler to run on NEW processors and output NEW code.

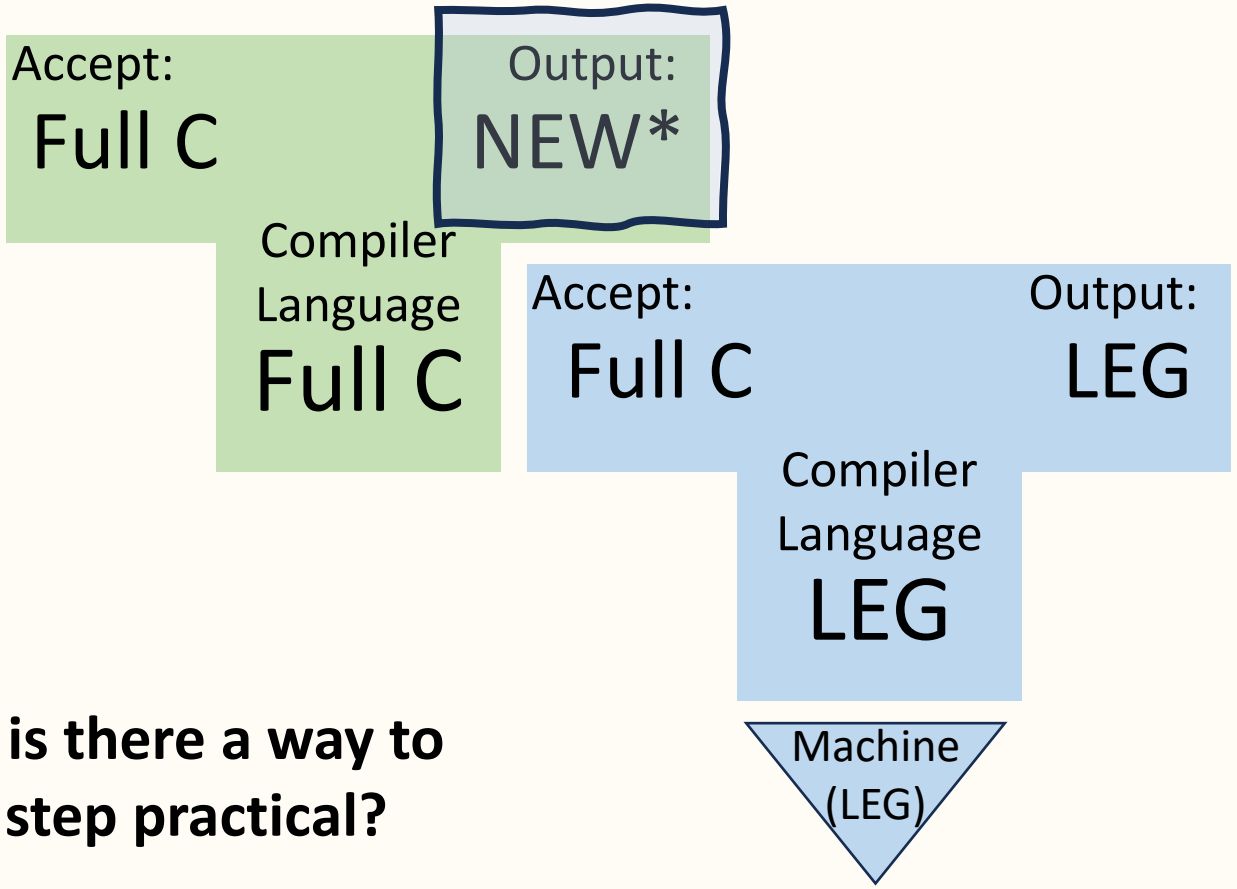
We want this:



Given a Portable and Native Compiler

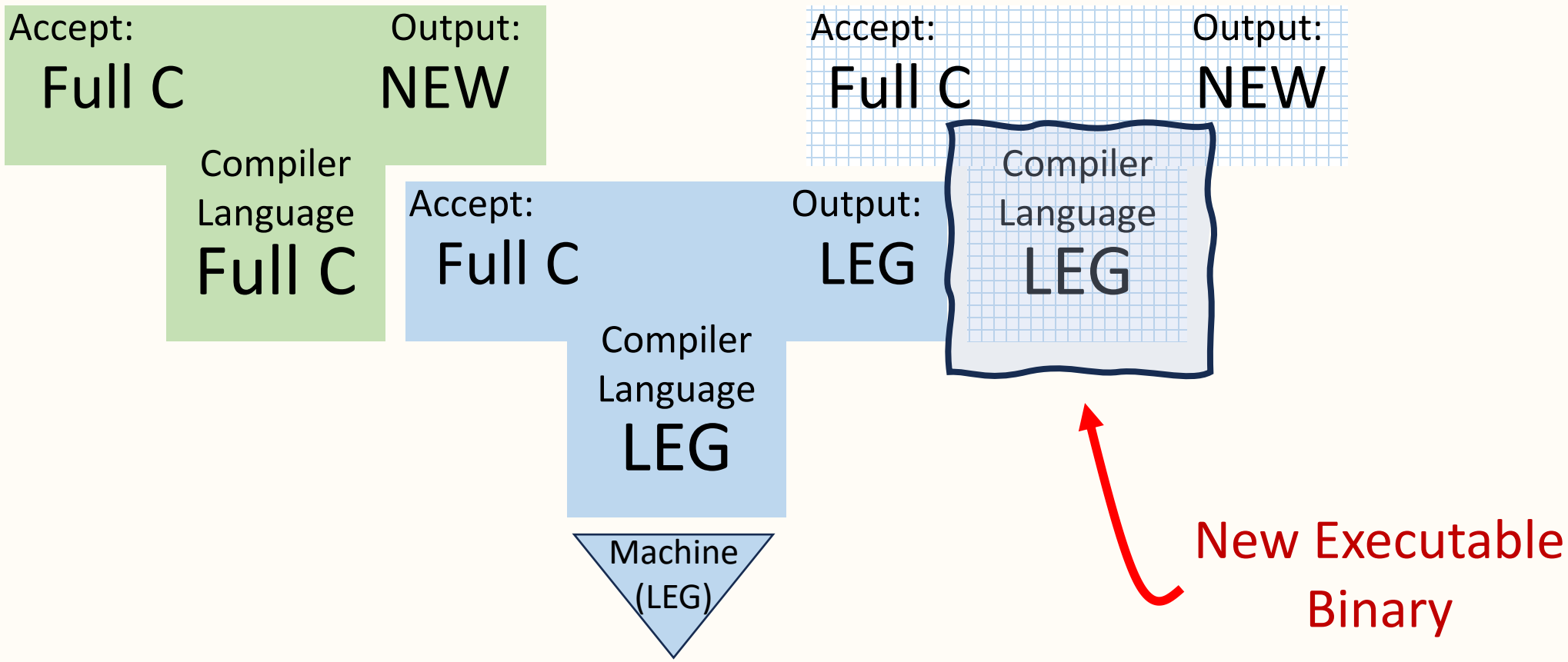


Rewrite your CodeGenerator

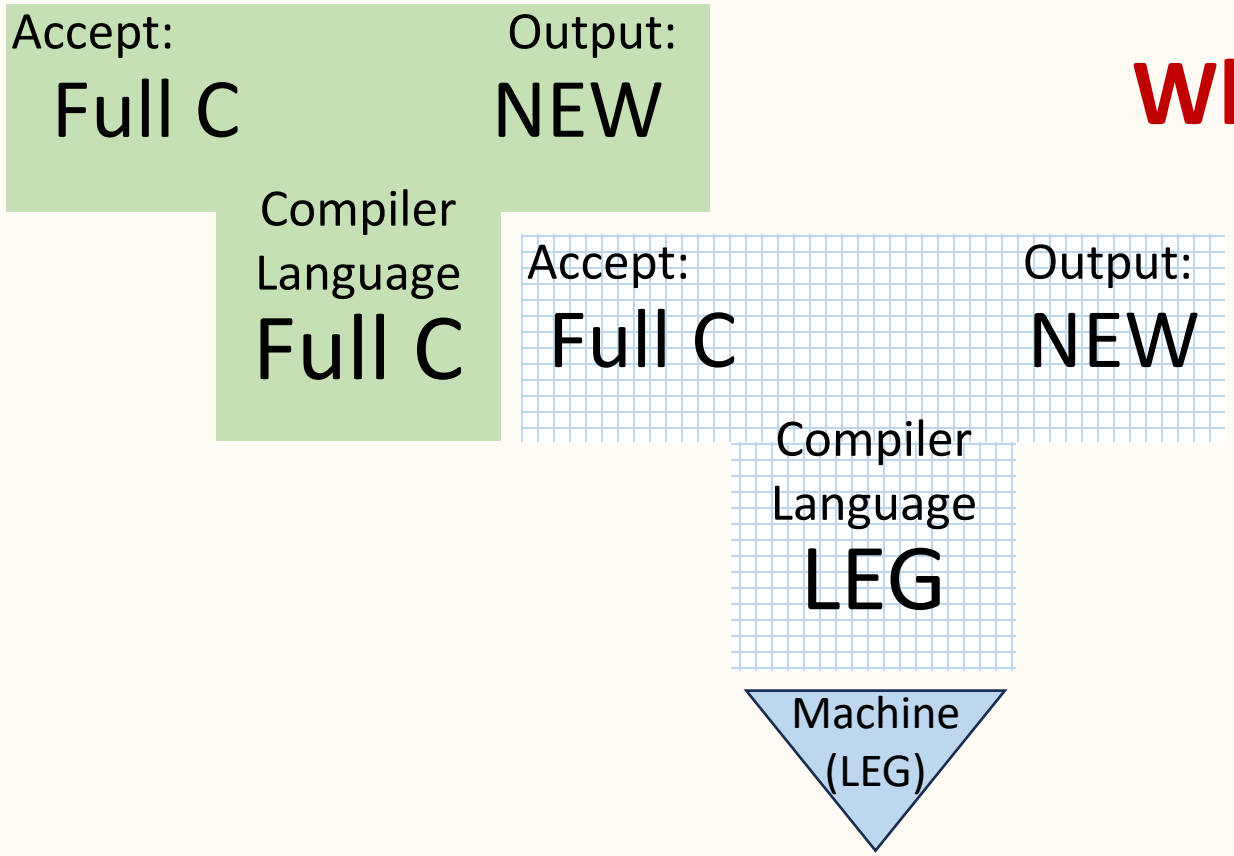


Question: is there a way to make this step practical?

Compile!

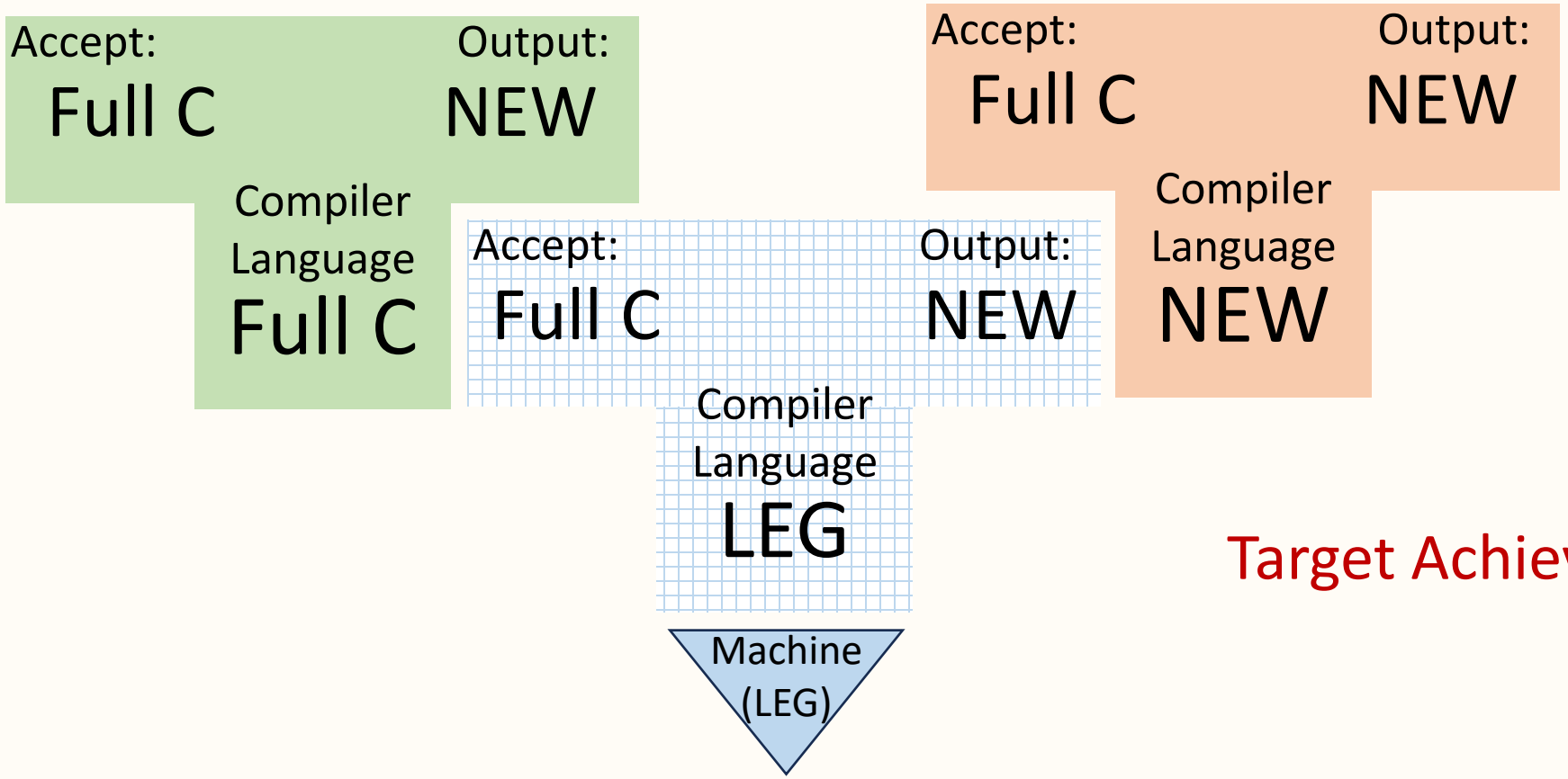

Accept:
Full C
Output:
NEW
Compiler Language
Full C
Accept:
Full C
Output:
LEG
Compiler Language
LEG
Machine (LEG)
Accept:
Full C
Output:
NEW
Compiler Language
LEG
New Executable Binary

Let's use that binary we just made. Same Portable Compiler on the left.



What happens now?

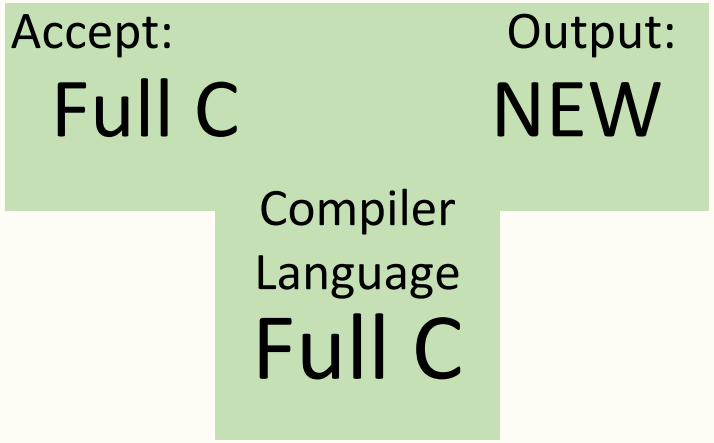
Compile!



Target Achieved!

Portable compiler

- This is why the Green compiler is known as the Portable Compiler.



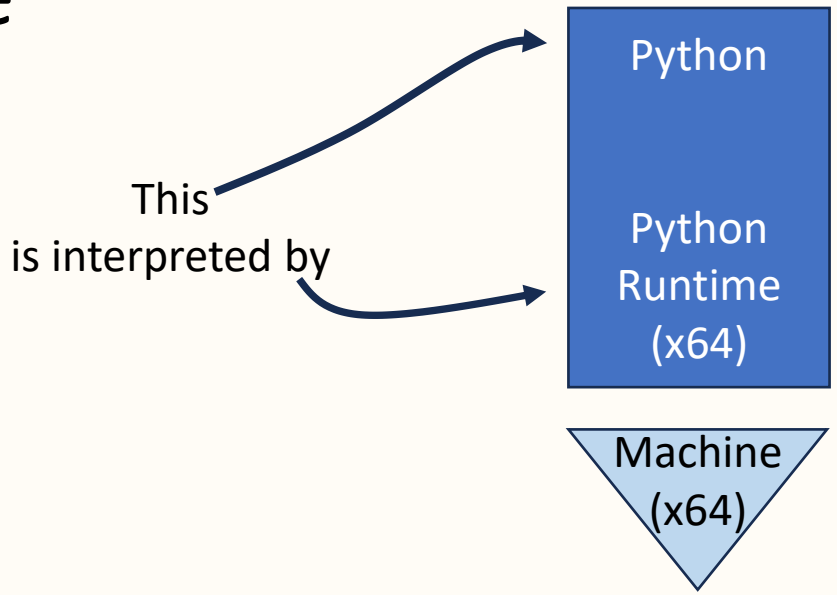
T-Diagram Takeaways

- If forced to write in assembly, make a **mini-language**
- With the *mini-language* compiler, create the *full-language* compiler
- Update the *mini-language* compiler to be written in *full-language* (Portable Compiler)
- Easy to add on new language semantics, and *somewhat* easy to target new binary formats.

Interpreters, Java, and a quick look into C#

Interpreters (Square Rectangle)

- Input is some language (often called a script)
- Input corresponds to operations in interpreter's language

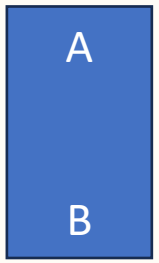


While interpreters can be slow, the input code is easy to modify and debug.

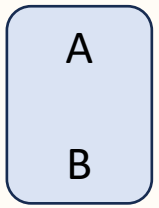
Note the simplicity of the diagram.

Quick to get up and running, and no slow slow compilation step!

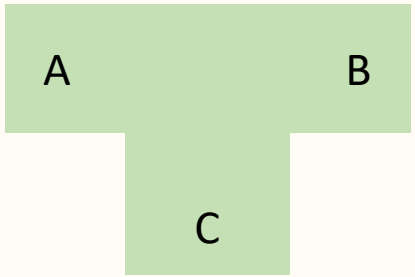
Quick Overview



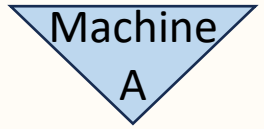
Input A is interpreted by language B



Input A is represented in language B



Input A is translated to language B
The translator is written in language C.



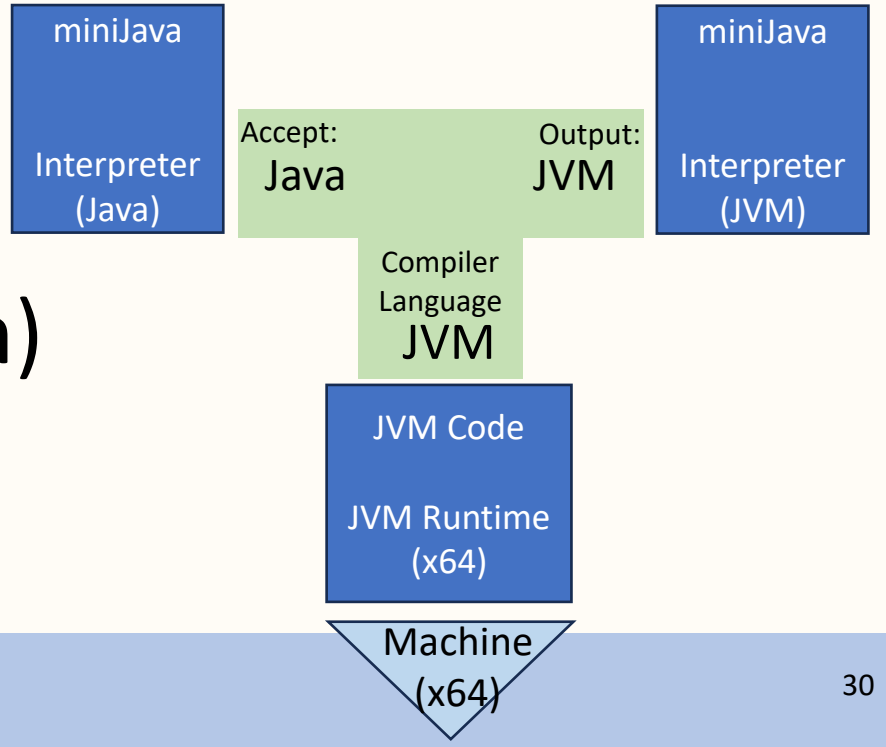
This machine can execute the language A

What if we interpreted miniJava?

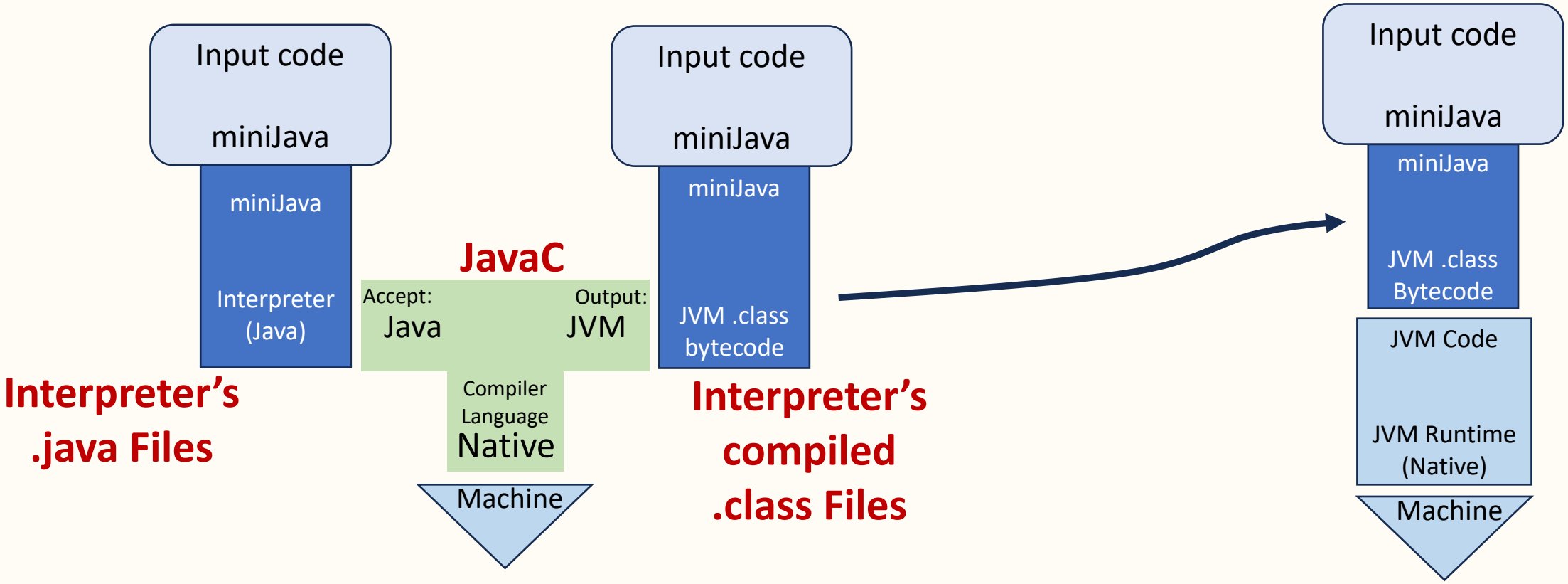
- Instead of outputting bytecode, what if we **interpreted** the input code and executed what it instructs to do in our Java Program?

It would look something like this:

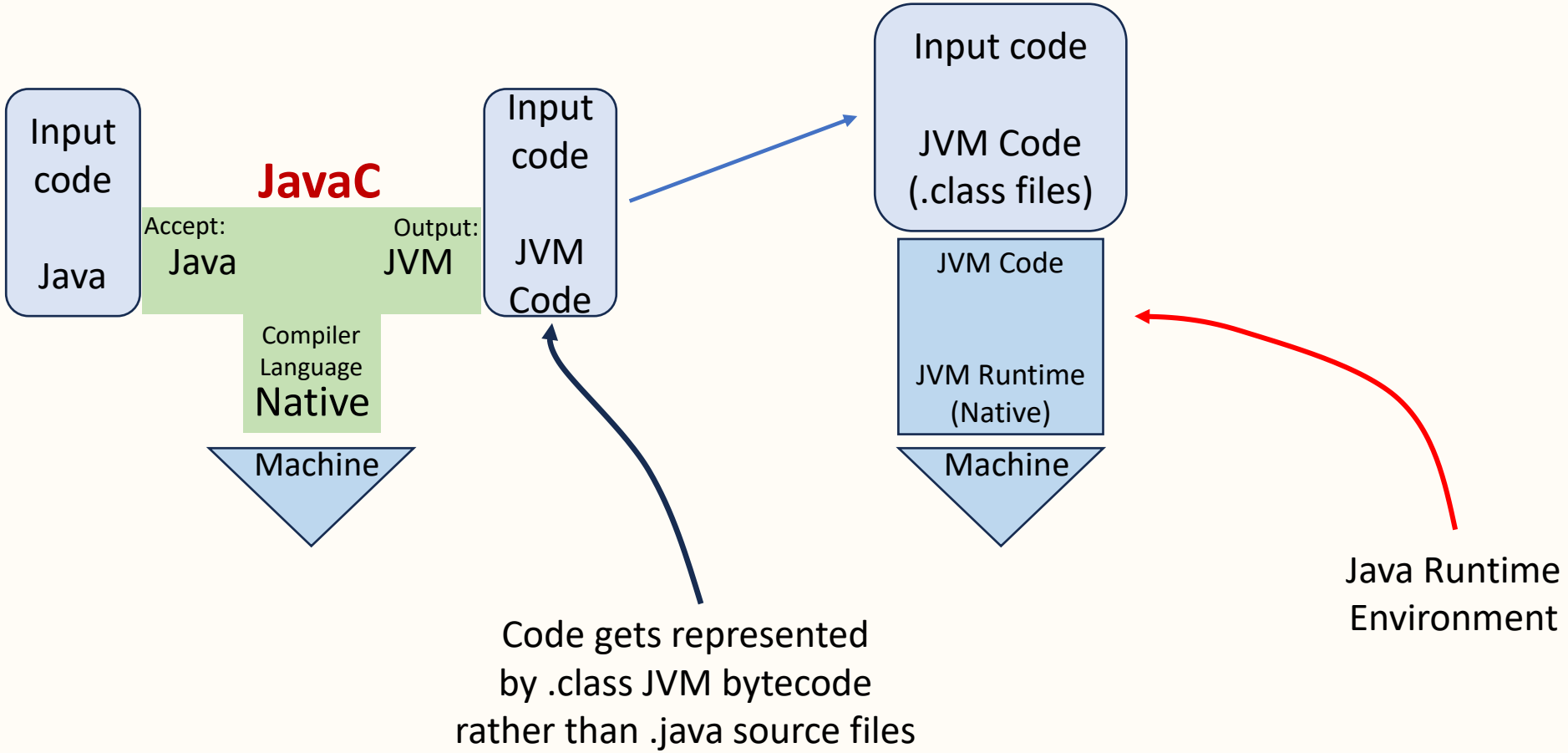
(Next slide details what is going on)



What if we interpreted miniJava? (2)

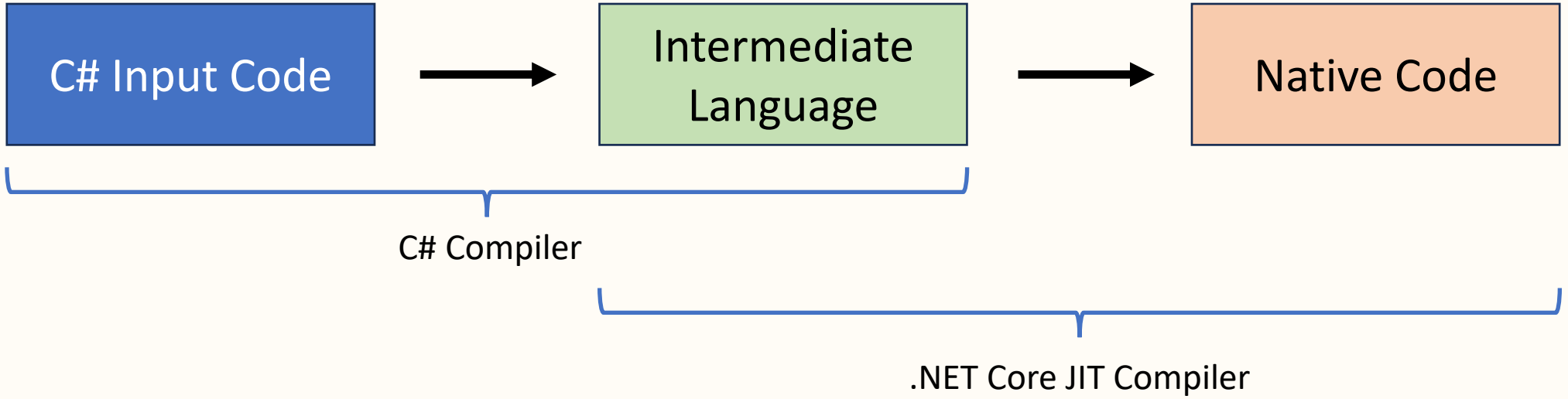


And what is regular Java?

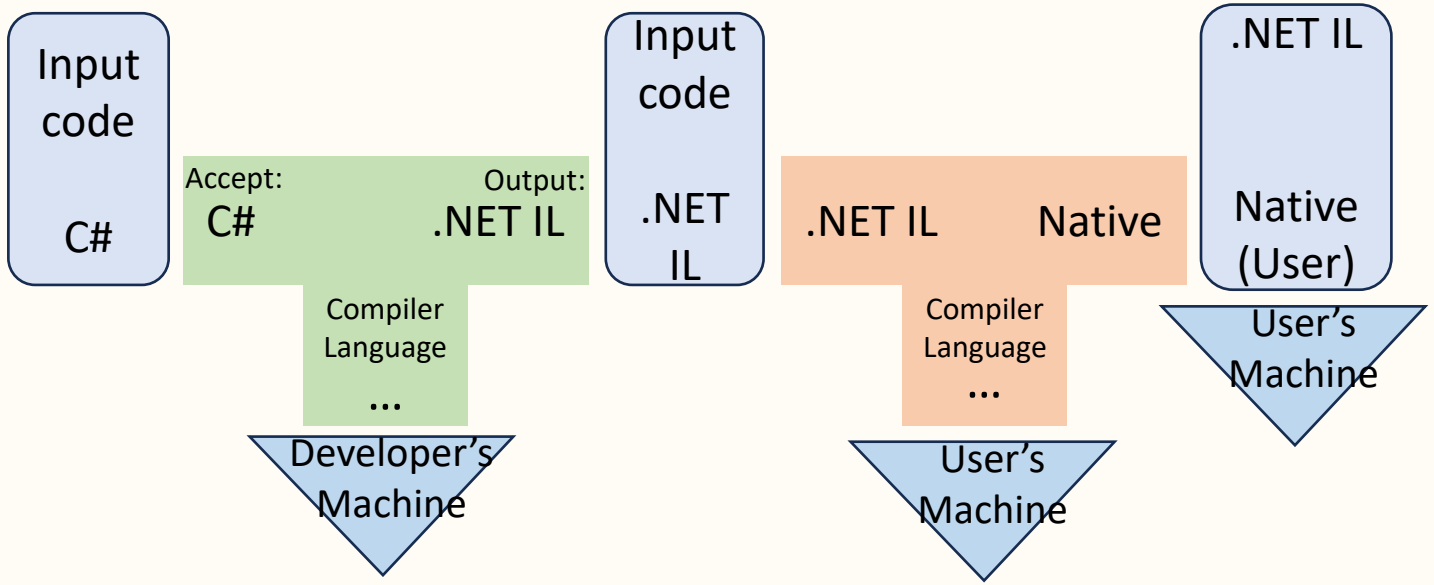


C# (.NET Core Framework)

- Input code compiles to an intermediate language (IL)
- IL then compiles to native code at runtime
- Native code runs on target machine



C# .NET Core T-Diagram



Two-step process but instead of interpreted like JVM, it gets compiled again and turned into native bytecode.

C# Question

- If **compiling** is a lengthy and cumbersome process, then having the user **compile** "Intermediate Language" code into "Native Code" on their machine means programs have a lengthy load time right?

- What are some ways around this?

C# Question (2)

- If compiling is a lengthy and cumbersome process, then having the user recompile "Intermediate Language" code into "Native Code" on their machine means programs have a lengthy load time right?

- Instead, this translator is a "**Just-in-time**" (JIT)
- **JIT Compiler: A compiler where only parts are compiled, and it compiles more and more code when needed.**

JIT Compilers – A middleground

- **JIT Compiler: A compiler where only parts are compiled, and it compiles more and more code when needed.**
- Give up the efficiency of native code
  - Cannot optimize dependent code segments compiled at different times.
- Often faster than an interpreter
- Sometimes slower than an interpreter (loading a new chunk of IL code, then compiling it takes time)
  - **Why is "slow but only for a moment" a problem?**

JIT Compilers

- We will look at JIT compilation techniques on 4/30
- Main idea: compile the code during runtime, and let it run with the efficiency of native machine code

Polymorphism

Brainstorm Time: instanceof

- Recall from PA4: we map memory layouts where fields are stored sequentially in memory.

- So if a class is just some bytes of storage for fields, **how can I tell if a class is an instanceof a class?**

RTTI: Runtime Type Information

- (Sometimes called Runtime Type Identification)
- Special data structure that resolves
  "object type `c1` is an instanceof `c2 c3 c4`"

- Instanceof operation then uses this special table to resolve if LHS instanceof RHS, and returns T/F
- Additionally, each class object now needs to keep track of "I am of type c1"
  - **How?**

Hint: Array.length, String.length

- (Similar PA5 extra credit opportunity)
- What if we **ADD A FIELD** to this object called ".length"?

- String is immutable: meaning calculate the hidden .length whenever the variable is set.
- Alternatively, if your implementation is not immutable, update .length field whenever you AssignStmt the String.

Example: Array

.length	arr[0]	arr[1]	arr[2]	…
objBase+0	objBase+8	objBase+16	objBase+24	…

(Assumption: 8 byte elements)

Idea: when you visit ixExpr for getting array values, just add +1. Additionally, resolve “ArrayType.length” with that object base +0.

Example: String

.length	.maxSize	char[0]	char[1]	char[2]	…
objBase+0	objBase+8	objBase**+16**	objBase**+17**	objBase**+18**	…

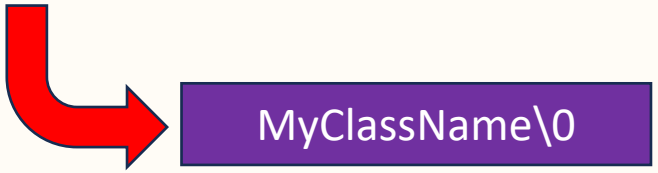
Mutable idea: Allocate new memory if the new .length will be more than the .maxSize (4kb). Otherwise, just write the chars!

New allocated memory could be a multiple of 4kb.

Immutable: no need for a .maxSize hidden field.

Example: Objects

.RTTIName	Field: x	Field: y	Field: z	…
objBase+0	objBase+8	objBase+16	objBase+24	…



(Assumption: 8 byte fields)

Because object names are variable-length, we can just store a pointer to somewhere in the heap for this object's RTTI name.

Polymorphism

- Consider three classes, A, B, C, and ***each*** contain **three ints, x, y, z.**
- Consider a fourth class, D, and it inherits all A, B, C.

```
class D : public A, public B, public C {
public:
    int w;
};
```

Polymorphism – Memory Layout

- The internal format of D is quite simple!

```
class D : public A, public B, public C {
public:
    int w;
};
```

D									
A			B			C			
x	y	z	x	y	z	x	y	z	w
+0	+4	+8	+12	+16	+20	+24	+28	+32	+36

Polymorphism – Member Access

- Unfortunately, difficult to resolve variables with the same identifier.

D									
A			B			C			
x	y	z	**x**	y	z	**x**	y	z	w
+0	+4	+8	+12	+16	+20	+24	+28	+32	+36

Consider:

```
D* d = new D();
d->x; // IDError, which "x" ??
```

Polymorphism – Member Access

- Only way to access variables is by expanding Type-Casting!

D									
A			B			C			
x	y	z	x	y	z	x	y	z	w
+0	+4	+8	+12	+16	+20	+24	+28	+32	+36

Consider:

```
D* d = new D();
B* b = (B*)d;
b->x = 1;// Not an error!
```

Polymorphism typecasting

```
class D : public A, public B, public C {
```

D									
A			B			C			
x	y	z	x	y	z	x	y	z	w
+0	+4	+8	+12	+16	+20	+24	+28	+32	+36

D B

- Operation: `b := (B*)d;`
- Typecast from D to B ≡ **Find where "B" starts in "D"**

Thus: "`b := (B*)d;`" ≡ "`b := (size_t)d + 12;`"

New Typecast Table Entry!

- Table inputs:
  - Original type
  - Destination type
- Table outputs:
  - Destination type (if entry exists, then typecast is allowed)
  - **Memory offset (when dealing with polymorphism)**
- This is why typecasting tables are very effective when compared to earlier simplified examples.

Virtual Methods

Virtual Method

- A virtual method is a method whose location is not known at compile-time.

- Why? Consider AST's `visit` method.
- If I `visit` a generic AST, I do not know what the concrete class is or which method body to invoke.
- The `visit` method is a **virtual method** that is resolved elsewhere and known at runtime.

Consider:

- What is the output?

```
class A {
public:
    virtual void f() { printf("A\n"); }
};

class B : public A {
public:
    virtual void f() { printf("B\n"); }
};

class C : public B {
public:
    virtual void f() { printf("C\n"); }
};

void main() {
    A* a = new A();
    B* b = new B();
    C* c = new C();

    a->f();
    b->f();
    c->f();

    b = (B*)c;
    b->f();

    a = (A*)c;
    a->f();
}
```

Consider:

- What is the output?

- Output is: ABCCC (with line breaks)

```
class A {
public:
    virtual void f() { printf("A\n"); }
};

class B : public A {
public:
    virtual void f() { printf("B\n"); }
};

class C : public B {
public:
    virtual void f() { printf("C\n"); }
};

void main() {
    A* a = new A();
    B* b = new B();
    C* c = new C();

    a->f();
    b->f();
    c->f();

    b = (B*)c;
    b->f();

    a = (A*)c;
    a->f();
}
```

Virtual Methods are Fields

- Memory Layout:

```
class A {
public:
    int w;
    virtual void f() { printf("A\n"); }
    int x;
    int y;
};
```

A			
w	f	x	y
objBase+0	+4	+12	+16

Question: In x64, why is the size of the field “f” 8 bytes long?

Virtual Method Call

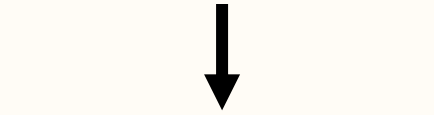
- Load field “f”
- Call whatever memory address it points to

A			
w	f	x	y
objBase+0	+4	+12	+16

- `mov rsi, objectA`
- `mov rax,[rsi+4]`
- `call rax`

Virtual Methods (Visualization)

A			
w	f	x	y
+0	+4	+12	+16



Read this data

CALL [rsi+4]

Data points to an address

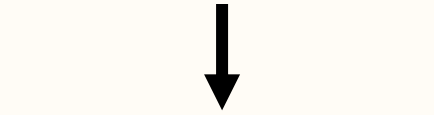
Method A.fn

Method B.fn

E.g. If the object was instantiated as **new B()**

Virtual Methods (Visualization)

A			
w	f	x	y
+0	+4	+12	+16



Read this data

Method A.fn

Method B.fn

E.g. If the object was instantiated as **new A()**

To wrap up virtual methods

- When an object A is allocated, field “f” is filled with the memory address of A.f
- When an object B (extends A) is allocated, field “f” is filled with the memory address of B.f (assuming B defines a method f)

- Combine this with polymorphism and type-casting, and you now can compile OO languages

Have a great weekend!

- Next week Tuesday is the LDOC
  - Modern Compilers use LLVM
  - LLVM, JIT strategies (Emulators are related), and wrap-up

- Please knock out PA5 earlier rather than later.
- Please finish WA5

End