COMP 520 - Compilers

Lecture 21 – LLVM, JIT Compilers

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.

COMP-750

- I recommend this class, but keep in mind, COMP-750 assumes you are taking at most one other class.
- Very difficult class even if you are only taking one other.

Interesting ID Errors

- Can access only the top-most class's variables when multiple inherited fields use the same variable.
- Assume A, B, C all contain public: int x;

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

void main() {
    D* d = new D();
    d->x = 1;
}
```

Virtual Address Table vs VAT Pointer

- Either works, but the primary difference is two memory deference operations vs one
- This memory dereference is already very slow. Turns out the reason is because the method address is a location in the code section, but the VAT will be in the heap, so you go to the heap to eventually load code in two possibly very different locations.
- With a VAT pointer, you might have two different cache lines for the VAT pointer and the VAT itself.
- Can cause cache interference galore!

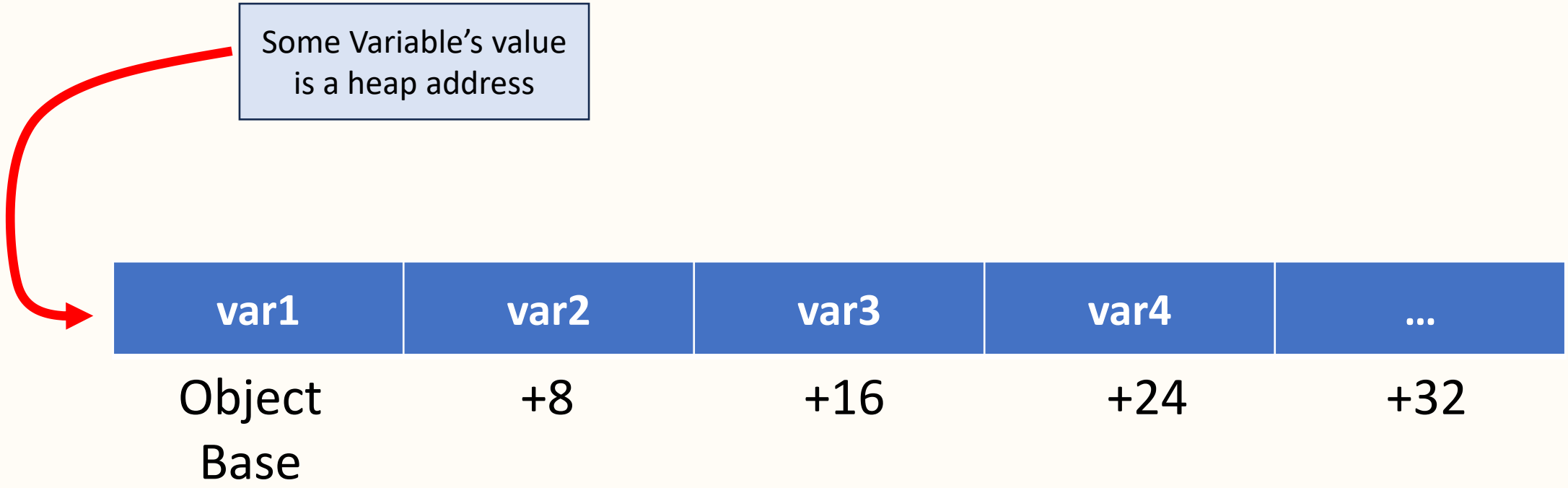
PA4 Overview

LocalDecl- ParameterDecl & VarDecl

ParameterDecl ← → VarDecl

x	[return addr]	*Old RBP*	Local var1	Local var2
RBP+16	RBP+8	RBP	RBP-8	RBP-16
RSP+32	**RSP+24**	**RSP+16**	**RSP+8**	**RSP**

FieldDecl- non-static



MethodDecl

- Start off with a stack frame
- End with removing the stack frame
- If it is the main method, then…
  - Consider static vars on the stack
  - End with a sys_exit

```
push rbp
mov rbp,rsp
```

```
mov rsp,rbp
pop rbp
```

Variable Access

- If the variable has a LocalDecl (parameter or VarDecl)
  - Access the variable from [rbp+VD.offset]

- If the variable has a FieldDecl:
  - If it is static, then access however you access static variables
  - Otherwise, the FieldDecl has an object offset, and access it from some context point (base address in the heap)

QRef

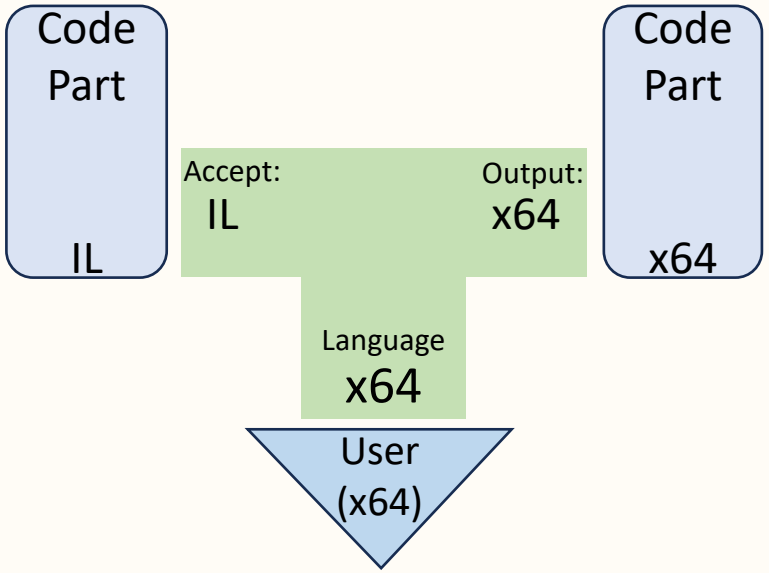
- If it is static, access static variable, otherwise…
- Visit the LHS to get the heap address, and once again, the RHS is a FieldDecl and has an objOffset, so we can access the variable as normal

Just-in-time Compilers

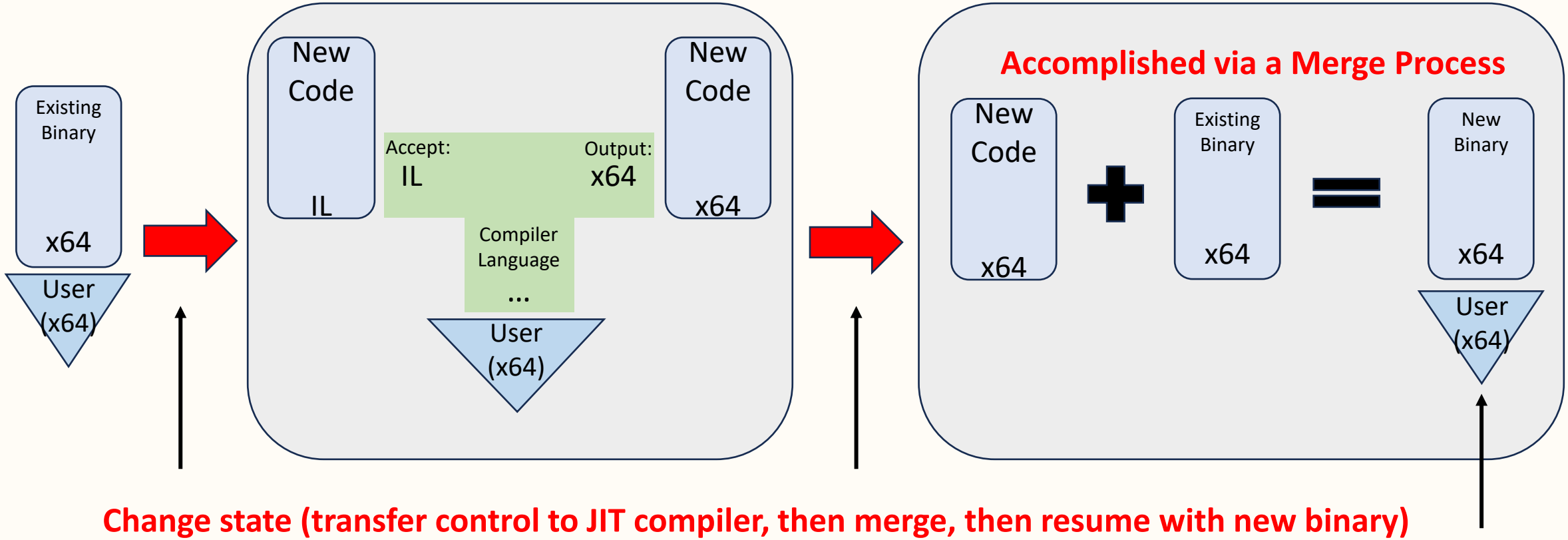
JIT Compilation

- Idea:
  - Partially compile parts of a program
  - Compile more of the program as needed
  - A mix of runtime states:
    - Can be running the program normally
    - Program may return to a “higher-level” runtime where it returns control to the JIT compiler

Step 1: Compile a part of the code



Step 2: On-demand (JIT) compile new code



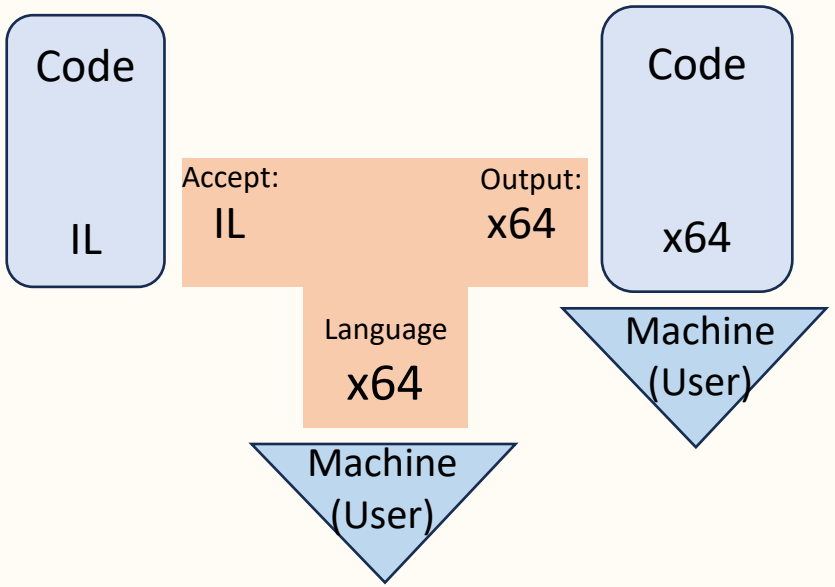
Change state (transfer control to JIT compiler, then merge, then resume with new binary)

Problem Statements

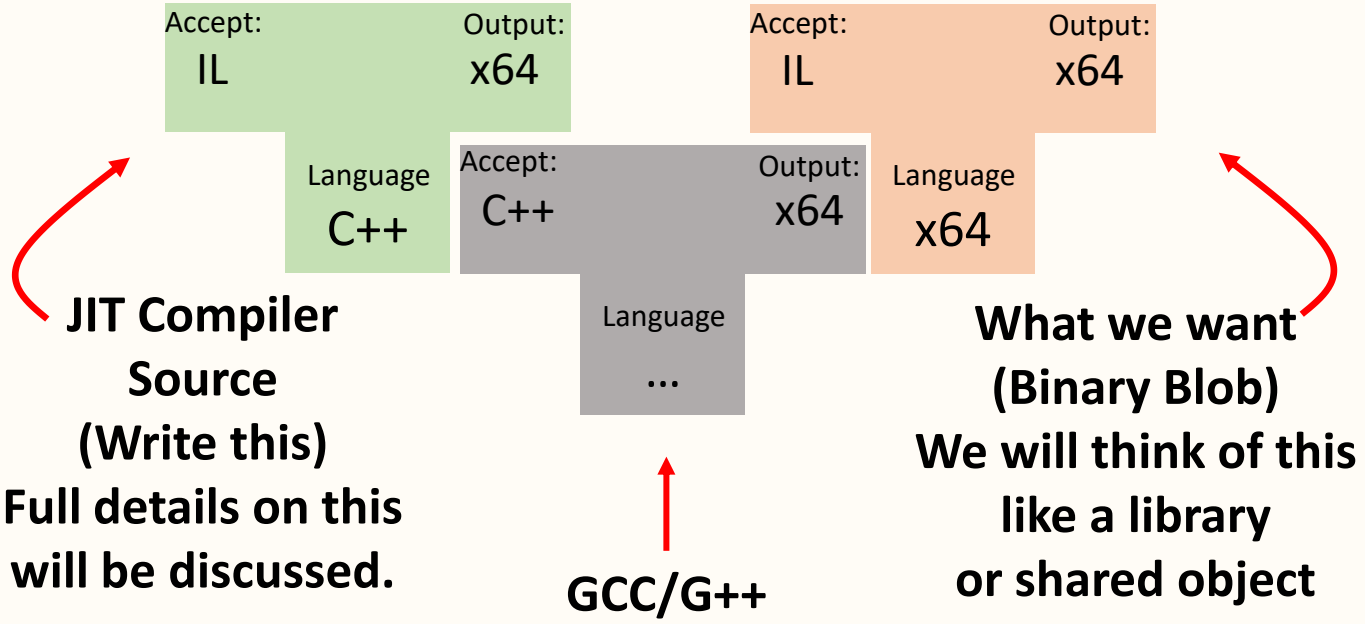
1. How much initial/subsequent code to JIT compile?
2. How/When do I invoke the on-demand JIT compiler?
3. How do I merge the two binaries together?
4. What does the entire process look like?
   (Note, the final binary contains the JIT compiler in it)

First, let's compile the JIT compiler

GOAL

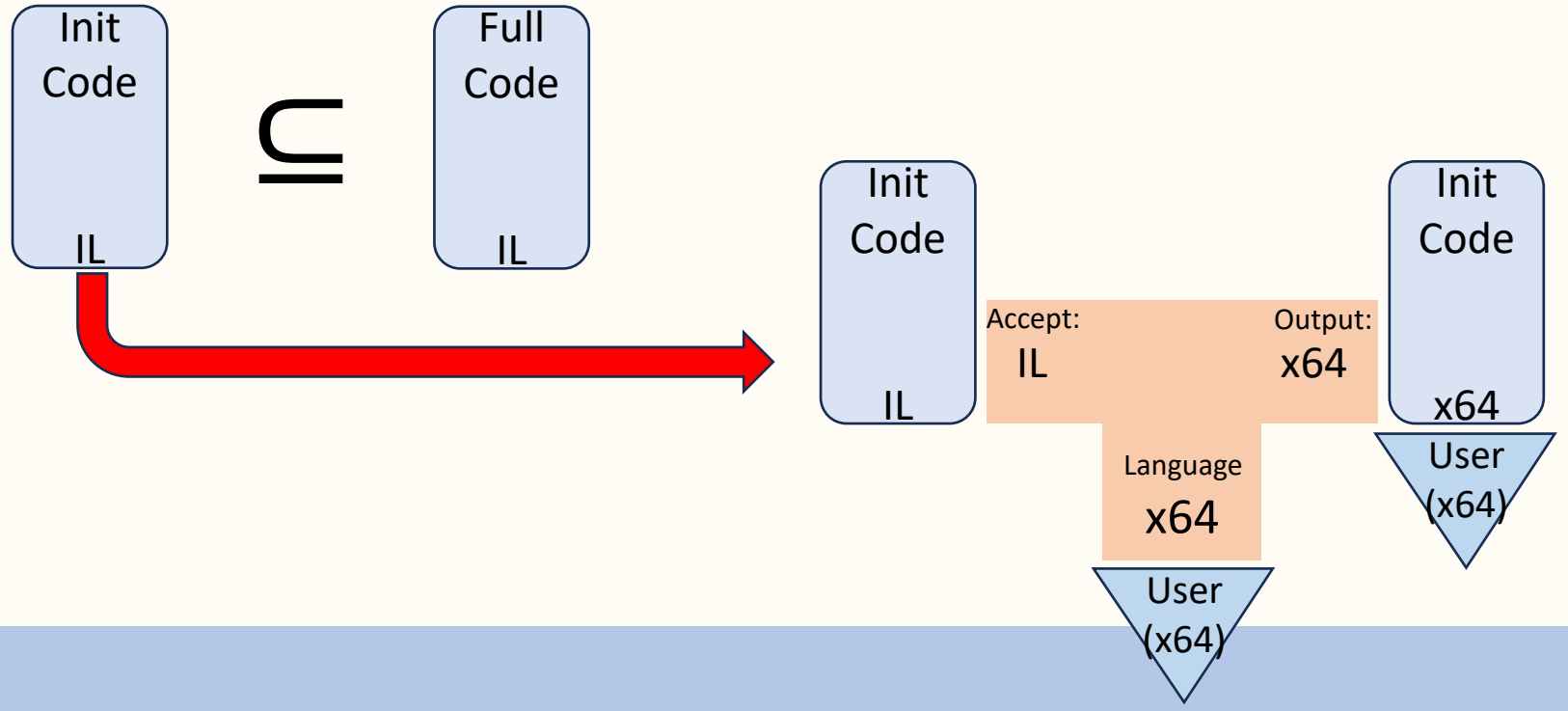


Need to Compile the Compiler



How much initial code to compile?

- On runtime, we have our input file in IL
- How much do we compile to start the program?



Initial Code – One method at a time

- Entrypoint in the IL makes sense. Consider:

```
void main(char* argv[], int argn) {
    bool debugMode = argn > 1;
    LoadData(debugMode);
    MainProgram::Instance()->Run();
    Cleanup();
}
```

Method Compilation

```
bool debugMode = argn > 1;
```

```
cmp [rbp+24],1
setg byte ptr[rbp-8]
```

Method Compilation (2)

```
bool debugMode = argn > 1;
LoadData(debugMode);
```

```
cmp [rbp+24],1
setg byte ptr[rbp-8]
call ???
```

Unknown JIT Entity

```
cmp [rbp+16],1
setg byte ptr[rbp-8]
call LoadData???
```

When compiling this instruction, we don't actually have `LoadData` compiled in native x64 code. (Infact, we're compiling our main function, nothing else is compiled yet!)

Unknown JIT Entity (2)

```
cmp [rbp+16],1
setg byte ptr[rbp-8]
call JIT(LoadData)
```

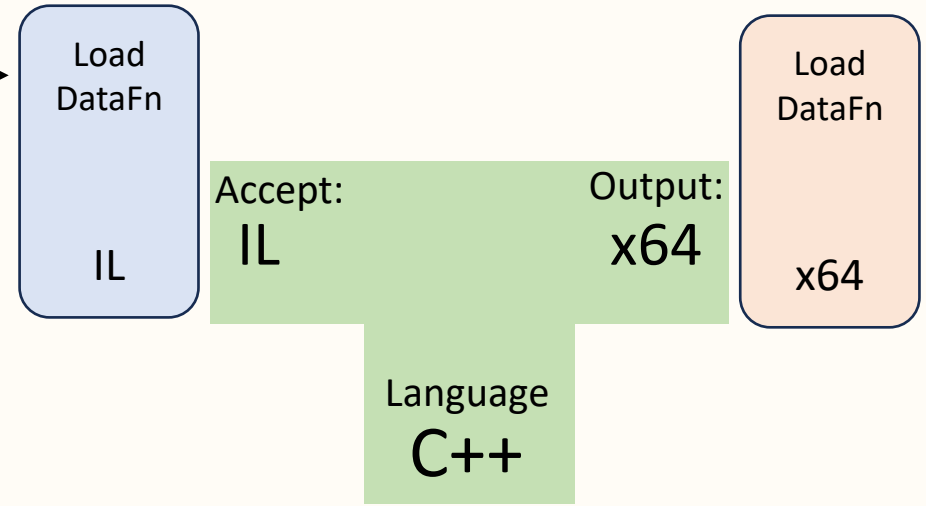
Generate a call to a method in the JIT compiler. Additionally, the "LoadData" parameter is associated with:

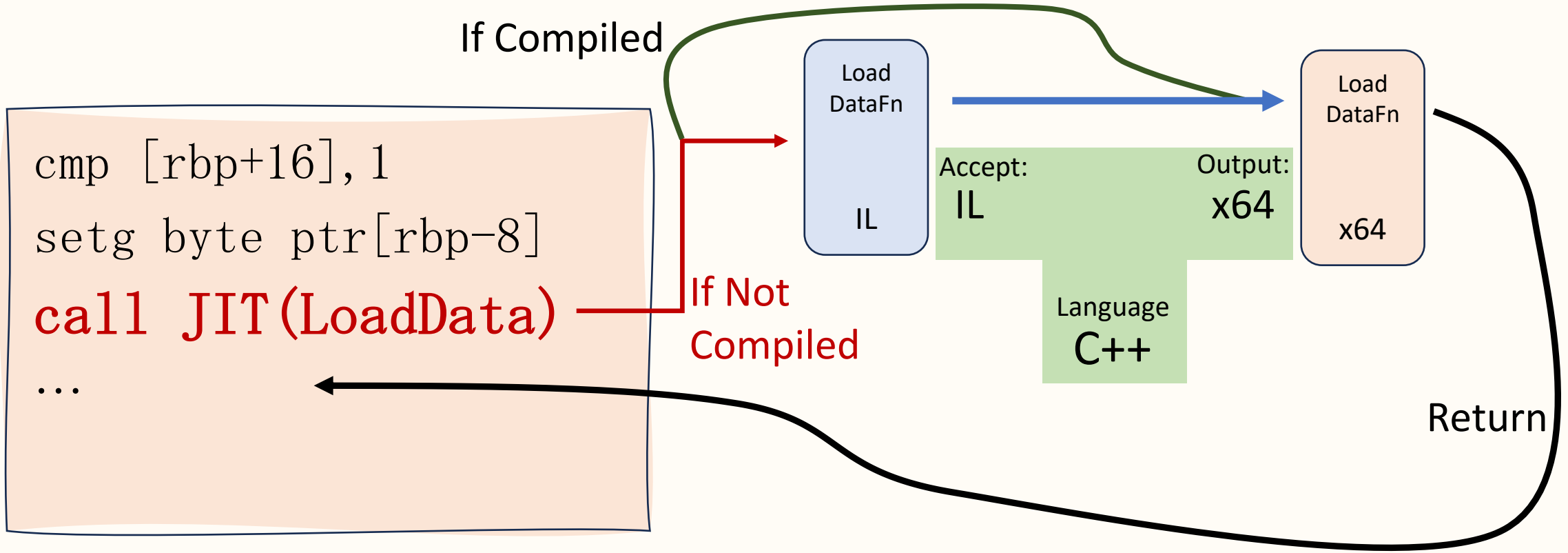
In the JIT compiler:

```
pushad
if( LoadData already compiled ) {
    popad, call LoadData
} else ???
```

Unknown JIT Entity (3)

```
cmp [rbp+16],1
setg byte ptr[rbp-8]
call JIT(LoadData)
```




If Compiled
cmp [rbp+16],1
setg byte ptr[rbp-8]
call JIT(LoadData)
…
If Not Compiled
Load DataFn
IL
Accept:
IL
Output:
x64
Language
C++
Load DataFn
x64
Return

Gritty Details

- How would you minimally define a program's state?

Gritty Details (2)

- How would you minimally define a program's state?
- Let's assume very simple hardware:
  - The register file
    - Including the instruction pointer (RIP)
  - The program's memory space (stack and heap)
  - Misc items (handles, pipes, file descriptors, control page, etc.)

Gritty Details (3)

- The register file: how to go from code to compiler?
  - If we use any registers, the program code might mess up.
  - Consider calling your JIT method when a variable is live in a register, might accidentally write over the live variable.

  - To solve this problem: use the instruction **pushad** (pushes all registers on the stack)
  - Before calling the method, use **popad** to restore the register state

Gritty Details (4)

- Memory space (stack, heap)
  - Because we created the JIT compiler, we know how the stack space is used.
  - For example, in our code, we have a local variable in **[rbp-8]** but we never moved **rsp** forward.
  - Determine the maximum amount of "unclaimed" stack space that contains live data, and move rsp so that it always points to ACTUALLY unused stack space.
  - E.g., sub rsp, 0x100, then after the popad, add rsp,0x100

Gritty Details (5)

- Misc Items (file descriptors, etc.)
  - Just don’t touch these in the JIT compiler, and they will remain in the same state.

Let's continue

```
bool debugMode = argn > 1;
LoadData(debugMode);
```

```
cmp [rbp+24],1
setg byte ptr[rbp-8]
invoke JIT(LoadFn)
```

Let's use a shorthand for "sub rsp,0x100, pushad, call, popad, add rsp,0x100" and call it invoke. Invoke will also push parameters on the stack. So far so good…

Problem: Virtual Method Call

```
bool debugMode = argn > 1;
LoadData(debugMode);
mp = MainProgram::Instance();
mp->run(); // virtual method
```

```
cmp [rbp+24],1
setg byte ptr[rbp-8]
invoke JIT(LoadFn)
invoke JIT(MP::Instance)
mov [rbp-16], rax
call [rax+8]
```

Virtual method call is a problem,
What do I pass to my JIT method??

Virtual Methods in JIT

- Need to be clever and solution will be specific to the hardware.

- Original code: call [rax+8]
- Consider:

```
sub rsp, 0x100
pushad
push [rax+8]
push SpecialIdentifier
call JIT
```

JIT(-1, MethodAddr)
…
Inside the JIT method:
"if MethodAddr is a known address, then call it, otherwise… **PROBLEM!!"**

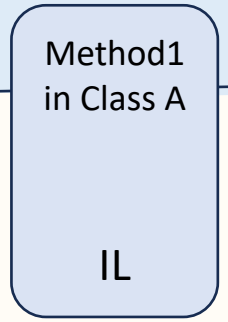
Virtual Methods in JIT (2)

- Need to be clever and solution will be specific to the hardware.

- Original code: call [rax+8]
- Consider:

```
sub rsp, 0x100
pushad
push rax
push 8
push SpecialIdentifier
call JIT
```

JIT(-1, 8, object)
…
If the object is sent along with the VAT index, then we know what the method should be, and can find the associated IL code:



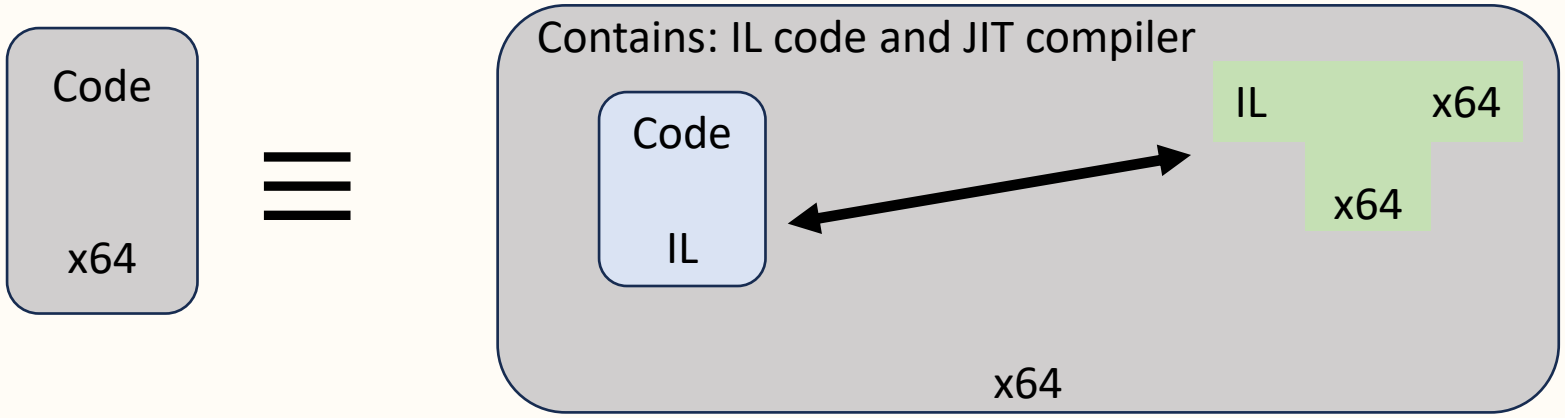
Some details omitted

- Need to store object type (RTTI) within objects, and that way the JIT compiler will know how to find the method in the IL code.
- There are some solutions WITHOUT RTTI that involve object allocation occur in the JIT compiler instead of regular runtime, and the VAT entries are all JIT methods.

- Takeaway: tons of ways to be clever here.

Binary Merging

- In our example, handled by having the JIT compiler as a loaded library inside our binary.
- Often, the JIT compiler is inside of the initial binary, and the IL code is too. (Not always, but it's faster)

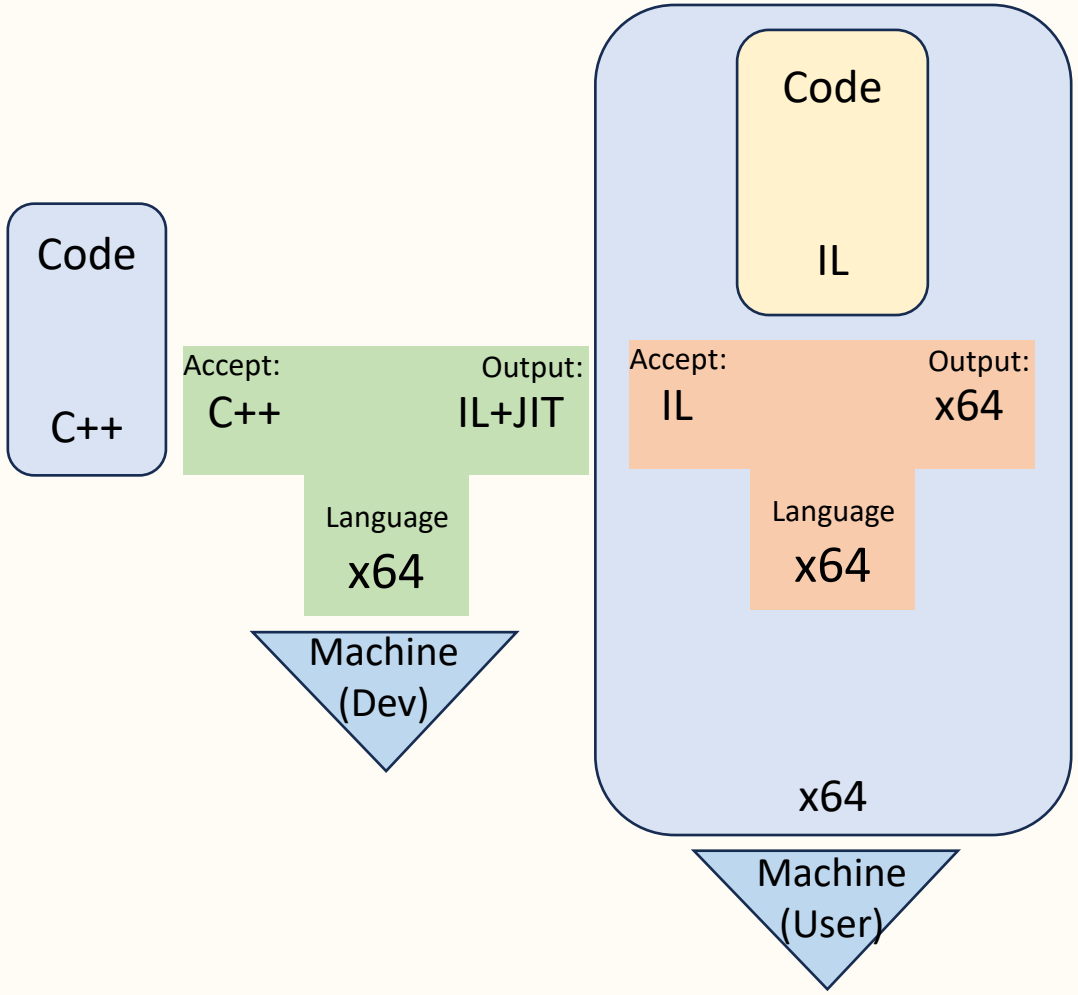


Final Process

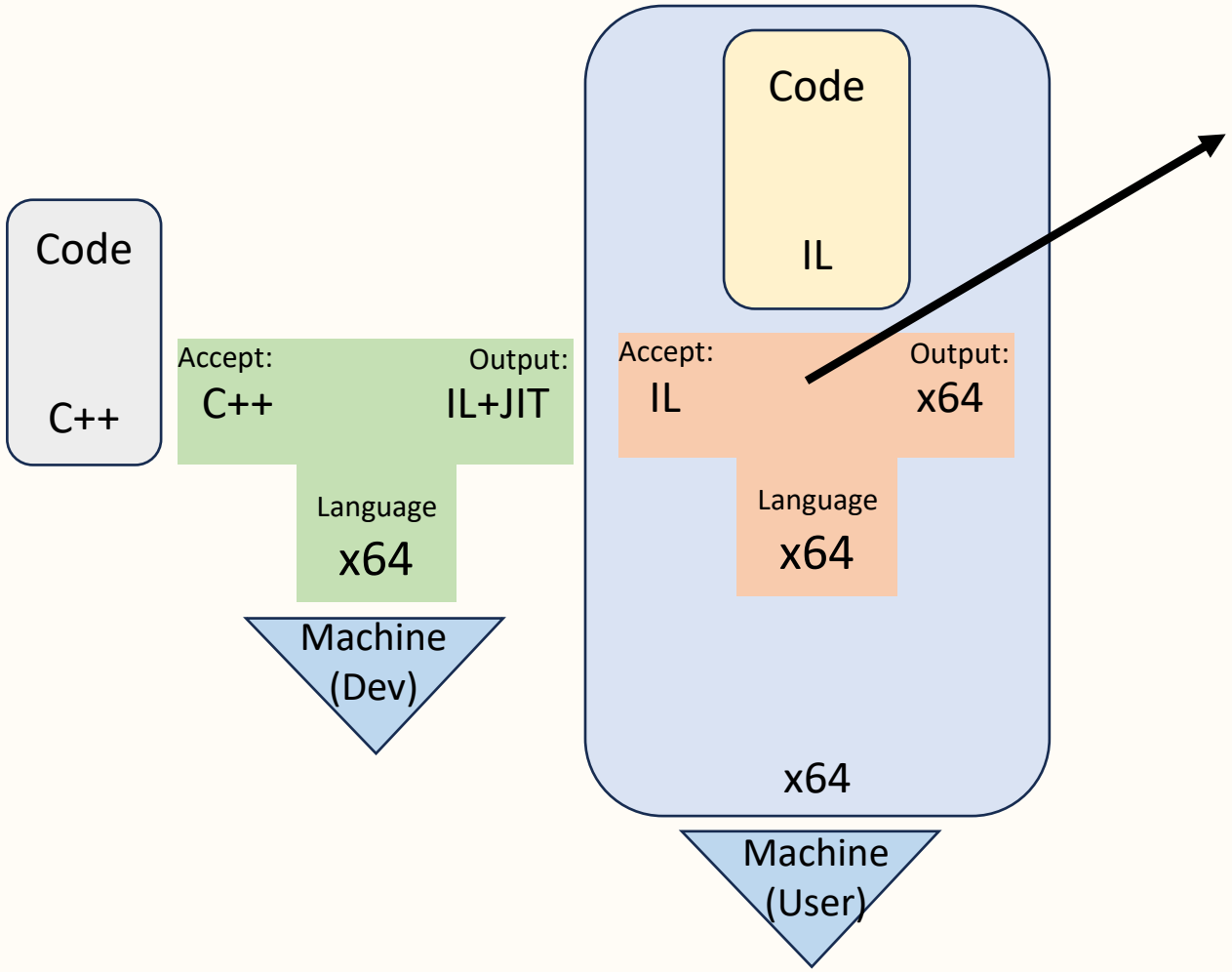
- You actually need to write a compiler that writes a JIT compiler specific to your input code, and that input code also needs to be converted to IL code that calls the compiled JIT compiler.

- Okay, that's a lot of words. But why?

Compiler creates IL and JIT



Compiler creates IL and JIT



Contains mappings from method names to parts in the IL code.

When JIT is invoked, find the associated IL code, check if it was already compiled, and if not, compile IL code, and patch the jump through.

Thus: need a table based upon the original code.

LLVM – Let’s look at modern compilers

Kaleidoscope: An LLVM Tutorial

- https://llvm.org/docs/tutorial/
- Let's look at how we can create a modern compiler
- We can compare and contrast with what we had to do

Step 1: The Lexer/Scanner

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html#the-lexer

- Idea: accumulate single letters at a time and store them in “Tokens”
- After accumulating a string, determine the token type. For example: tok_identifier, tok_number, etc.

Step 1: The Lexer/Scanner

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html#the-lexer

```
if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}
```

This is all pretty straightforward code for processing input. When reading a numeric value from input, we use the C `strtod` function to convert it to a numeric value that we store in `NumVal`. Note that this isn't doing sufficient error checking: it will incorrectly read "1.23.45.67" and handle it as if you typed in "1.23". Feel free to extend it! Next we handle comments:

```
if (LastChar == '#') {
  // Comment until end of line.
  do
    LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}
```

We handle comments by skipping to the end of the line and then return the next token. Finally, if the input doesn't match one of the above cases, it is either an operator character like '+' or the end of the file. These are handled with this code:

Step 1: The Lexer/Scanner - miniJava

- In comparison, what you learned in this class:

- Techniques:
  - Make **everything** a Token or…
  - **Reduce** the types of Tokens
- **ε-closure** and conversion from an **NFA** to a **DFA**
- Can pass on/reduce the burden of context to later stages (*e.g.* parsing factorial vs negation, or reducing parsing by making fancy decisions like scanning "[]" as a single Token)

Step 2 – Implementing a Parser and AST

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html

- An AST contains the constructs of the input language.
- Should closely model the language

Step 2 – Implementing a Parser and AST

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html

```
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;

public:
  VariableExprAST(const std::string &Name) : Name(Name) {}
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  std::unique_ptr<ExprAST> LHS, RHS;

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
    : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<std::unique_ptr<ExprAST>> Args;

public:
  CallExprAST(const std::string &Callee,
              std::vector<std::unique_ptr<ExprAST>> Args)
    : Callee(Callee), Args(std::move(Args)) {}
};
```

This is all (intentionally) rather straight-forward: variables capture the variable name, binary operators capture their opcode (e.g. '+'), and calls capture a function name as well as a list of any argument expressions. One thing that is nice about our AST is that it captures the language features without talking about the syntax of the language. Note that there is no discussion about precedence of binary operators, lexical structure, etc.

Precedence in LLVM

- Need to only specify precedence:
- Assign each Operator token the relevant precedence value.
- Loop through and find highest precedence operations in an Expression, and resolve those first.

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.
  ...
}
```

```
// If this is a binop, find its precedence.
while (true) {
  int TokPrec = GetTokPrecedence();

  // If this is a binop that binds at least as tightly as the current binop,
  // consume it, otherwise we are done.
  if (TokPrec < ExprPrec)
    return LHS;
```

Step 2 – Parsing and ASTs in miniJava

- We learned/discussed Parsing:
  - **Recursive descent** (implemented)
  - **Shift-Reduce parsing** (discussed)
  - **Push-down automata** (you learned this in 455, discussed)
- We discussed ASTs:
  - Selection of AST grammars should be to achieve a **separation of concerns**
  - **Stratified grammars** can create ASTs with proper precedence constraints easily (never worry about precedence again)

LLVM combines PA3 and PA4

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

- In their tutorial for LLVM, only one IDTable (called NamedValues), a very simple language that doesn't need SI or objects or fields.
- In comparison: you learned object oriented contextual analysis, which is significantly harder.
- CodeGen is actually done per-AST. Each concrete AST defines a codegen method and generates code that way.

LLVM Traversal: "You should try visitors"

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() = default;
  virtual Value *codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}
  Value *codegen() override;
};
...
```

The codegen() method says to emit IR for that AST node along with all the things it depends on, and they all return an LLVM Value object. "Value" is the class used to represent a "Static Single Assignment (SSA) register" or "SSA value" in LLVM. The most distinct aspect of SSA values is that their value is computed as the related instruction executes, and it does not get a new value until (and if) the instruction re-executes. In other words, there is no way to "change" an SSA value. For more information, please read up on Static Single Assignment – the concepts are really quite natural once you grok them.

Note that instead of adding virtual methods to the ExprAST class hierarchy, it could also make sense to use a visitor pattern or some other way to model this. Again, this tutorial won't dwell on good software engineering practices: for our purposes, adding a virtual method is simplest.

LLVM- Front End vs Back End

- Front-end deals with parsing (think PA1-PA3)
- Back-end deals with code generation
- LLVM has prebuild back-end drivers that will generate things like x64 or arm

- If you have a custom architecture, will have to do code generation just like you did with PA4.

LLVM – Backend programming

- Must specify a register set (number of registers)
- Must specify register overlap (think RAX/EAX/AX/AL)
- Unsurprisingly, would have to do file headers yourself (can't quite automate that process)

- Unlike what we did in class, LLVM requires you to go from the IL to your destination code (instead of from ASTs).
- This is easier as IL is similar to assembly.

Backend / PA4

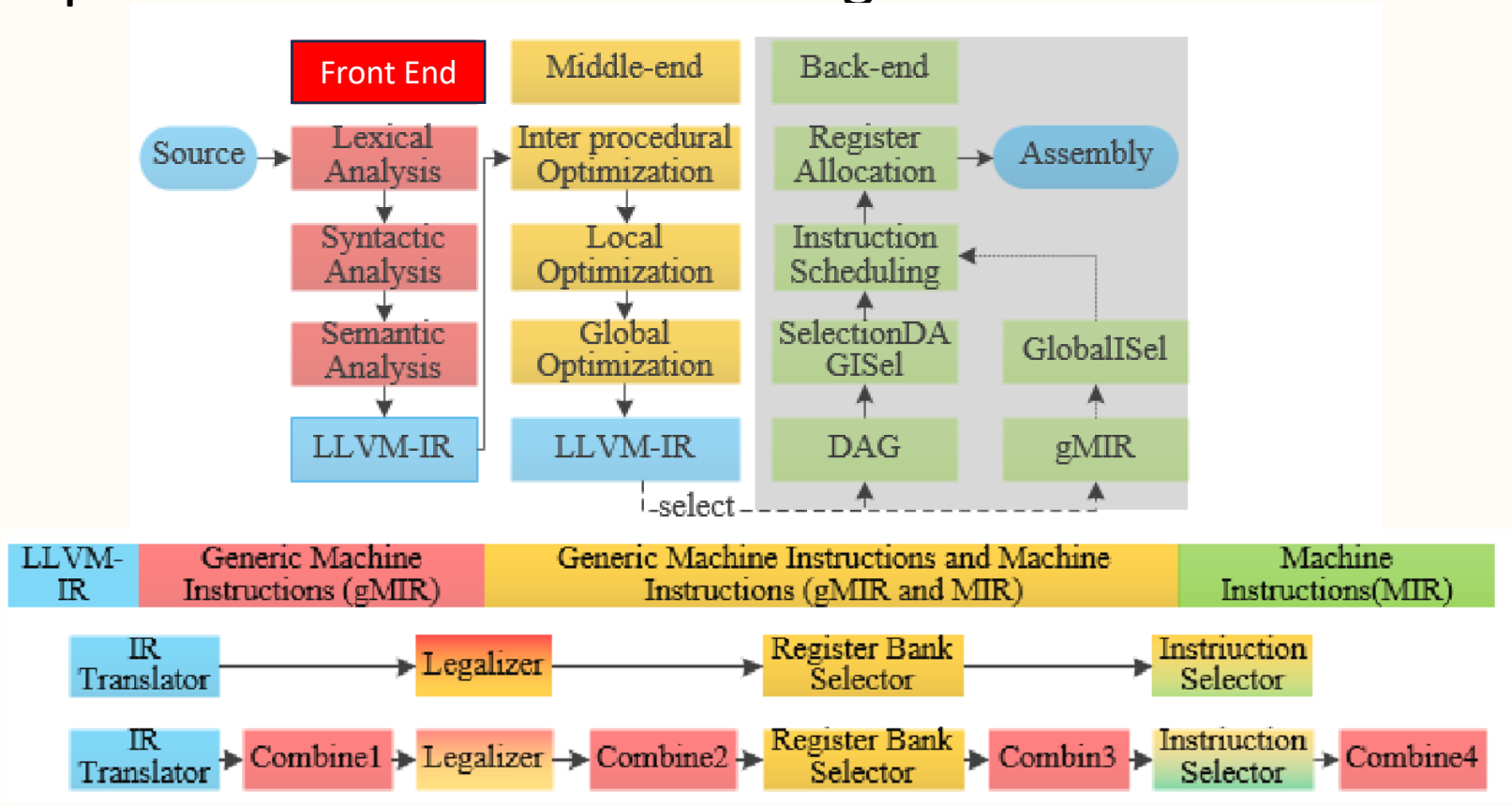
- What you did in comparison:
  - Did the more difficult output to assembly from ASTs
- This means you are well equipped to output to an IL then convert from your own IL to real assembly whenever you need to retarget.

Backend / PA4 (2)

- LLVM backend tutorial: https://llvm.org/docs/WritingAnLLVMBackend.html
- Idea: register sets, instruction sets, instruction scheduling, relation mapping, and branch construction.
- After that, you create a directed acyclic graph (DAG) as your instruction selector

Backend / PA4 (3)

Source: Optimization based on LLVM global instruction selection



Code Generation

- Idea: several passes to reach usable code
- Similar idea to generating tuple code, then “legalizing” the concept of unlimited registers by reducing register usage, generating spillover code, and changing “extra” registers to memory operations.

- Multiple passes needed to achieve usable code.

To conclude COMP-520

- "Reflections on Trusting Trust" – Ken Thompson
  - Idea: how to compile bugs/backdoors in a compiler that can't be discovered unless you analyze the assembly code
- You learned quite a bit more than "how to build a basic compiler that isn't practical"
- Use techniques and theory from this class to build modern and competitive compilers. (Recall MSVS example)
- Ideally, you also learned the importance of time management when you have a month to work on a programming project.

See you at the final!

5/9 at 4pm

Have a great summer!

End