Lecture 3 — Building DSLs

Stanford CS343D (Winter 2026)
Fred Kjolstad
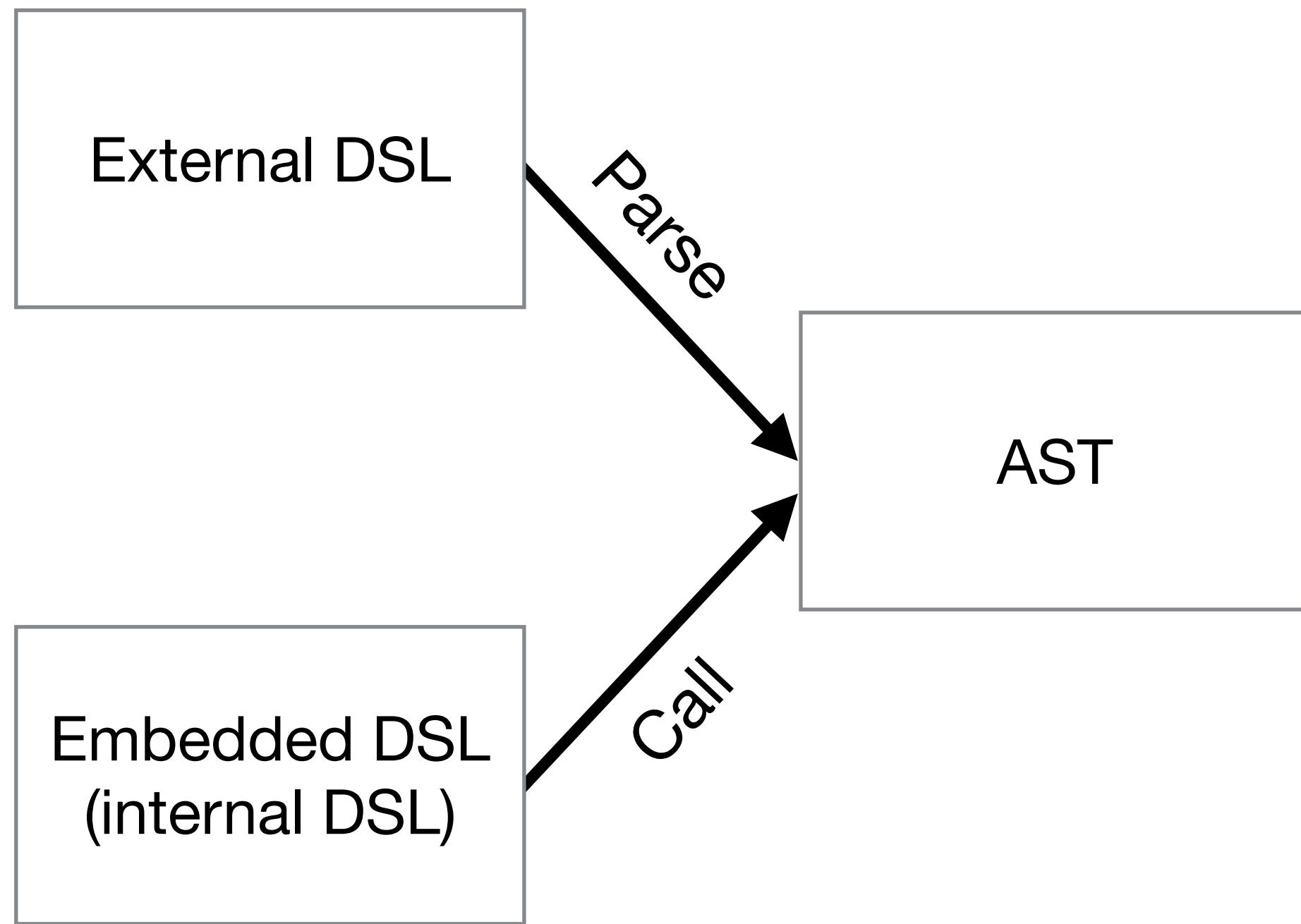
Types of DSLs — languages or libraries?

Implemented as standalone language

External DSL

Parse

AST

Types of DSLs — languages or libraries?

Implemented as standalone language

External DSL → Parse → AST

Embedded DSL
(internal DSL) → Call → AST
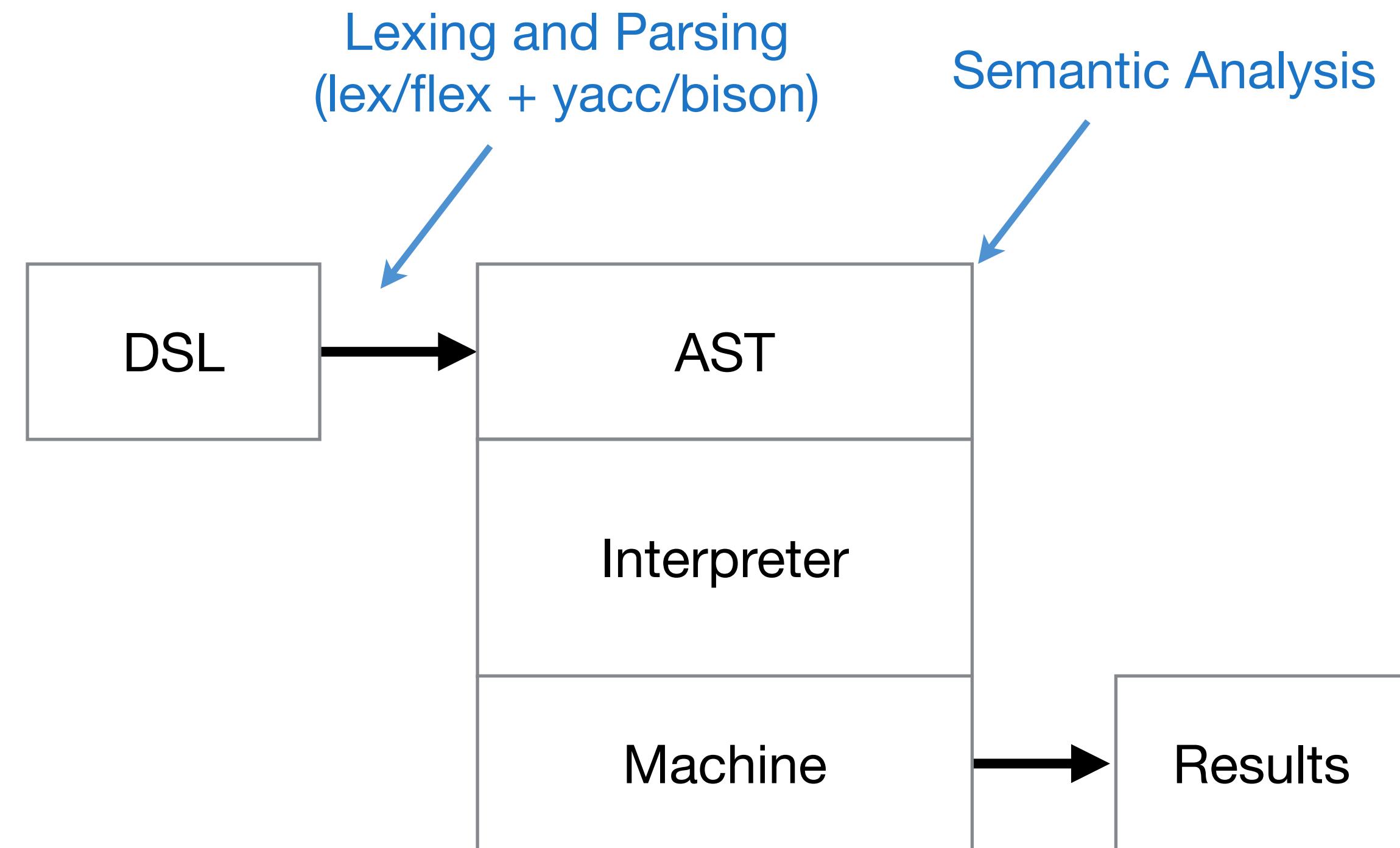
Embedded inside another language.
Ideally the host language has features to
make it easy to embed DSLs.

External DSLs

That is, DSLs as textual languages

External DSLs — Implementation

External DSLs — Demo

calc1.py
calc2.py

lexical analysis
syntactic analysis
interpretation
ASTs

External DSLs — Advantages and Disadvantages

Advantages

+ Flexibility (syntax and semantics)

+ Easy to make a small textual language

Disadvantages

- Yet another programming language

- Syntactic cacophony

- Slippery slope towards generality

- Hard to interoperate with other languages

- No tool chain: IDE, debuggers, profilers

Embedded DSLs

That is, DSLs as a library

Embedded DSL — Language implemented as a library

OpenGL

```
glMatrixMode(GL_PROJECTION);
glPerspective(45.0);

for(;;) {
    glBegin(TRIANGLES);
        glVertex(…);
        glVertex(…);

        …
    glEnd();
}

glSwapBuffers();
```

Fluent Interfaces — Composable API calls with method chaining

html

```
<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```

jquery

```
// turn first element green
$("li:first").css("color", "green");
```

Sophisticated data rendering with embedded DSL

https://www.d3-graph-gallery.com/graph/density_basic.html

http://d3js.org/

Sparse Tensor Algebra DSL in C++ (taco)

```
Format dv({dense});
Format csr({dense,compressed});

Tensor<double> a({m},   dv);
Tensor<double> c({n},   dv);
Tensor<double> B({m,n}, csr);

// Load data

IndexVar i,j,i1,i2;
a(i) = sum(j, B(i,j) * c(j));

a.split(i, i1, i2, Down, 32);
 .parallelize(i1, CPUThread, NoRaces);

std::cout << a << std::endl;
```

C-like DSL (Pochi) embedded in C++ for online code generation

```
1  Function* regexfn = codegen("ab.d*e");
2  using Regexs = int(*)(vector<string>*);
3  auto [regexs, inputs] = newFunction<Regexs>("regexs");
4  auto result = regexs.newVariable<int>();
5  auto it = regexs.newVariable<vector<string>::iterator>();
6  regexs.setBody(
7    Declare(result, 0),
8    For(Declare(it, inputs->begin()),
9        it != inputs->end(),
10       it++
11   ).Do(
12     result += StaticCast<int>(
13       Call<RegexFn>(regexfn, it->c_str()))
14   ),
15   Return(result)
16 );
17
18 vector<string> input {"abcde", "abcdde",  // good input
19                       "abde" , "abcdef"}; // bad input
20 buildModule();
21 Regexs match = getFunction<Regexs>("regexs");
22 assert(match(&input) == 2);
```

Pochi loop iterates over
a C++ STL iterator

```
1  using RegexFn = bool(*)(char* /*input*/);
2  Function* codegen(const char* regex) {
3    auto [regexfn, input] = newFunction<RegexFn>();
4    if (regex[0] == '\0') {
5      regexfn.setBody(
6        Return(*input == '\0')
7      );
8    } else if (regex[1] == '*') {
9      regexfn.setBody(
10       While(*input == regex[0]).Do(
11         input++,
12         If (Call<RegexFn>(codegen(regex+2), input)).Then(
13           Return(true)
14         )
15       ),
16       Return(false)
17     );
18   } else if (regex[0] == '.') {
19     regexfn.setBody(
20       Return(*input != '\0' &&
21              Call<RegexFn>(codegen(regex+1), input+1))
22     );
23   } else {
24     regexfn.setBody(
25       Return(*input == *regex &&
26              Call<RegexFn>(codegen(regex+1), input+1))
27     );
28   }
29   return regexfn;
30 }
```

Pochi test on
runtime regex

13

Embedded DSLs — Advantages and Disadvantages

Advantages

+ Familiar host language syntax

+ Can combine DSL code with host language features

+ Can interoperate with other libraries

+ Complete host language toolchain

Disadvantages

- Host language syntax can be rigid and verbose

- Hard to debug DSL with host language tools

- Hard to restrict features in DSL

- Still hard to develop

Shallow Embedding

A shallow embedding is when the expressions are interpreted in the semantics of the base language

calc1.py: direct interpretation of arithmetic

Deep Embedding

A deep embedding first builds an abstract syntax tree (AST). The abstract syntax tree is typically an algebraic data type. The AST is then evaluated with an interpreter.

calc2.py: AST represented as lists of lists

Operator Overloading
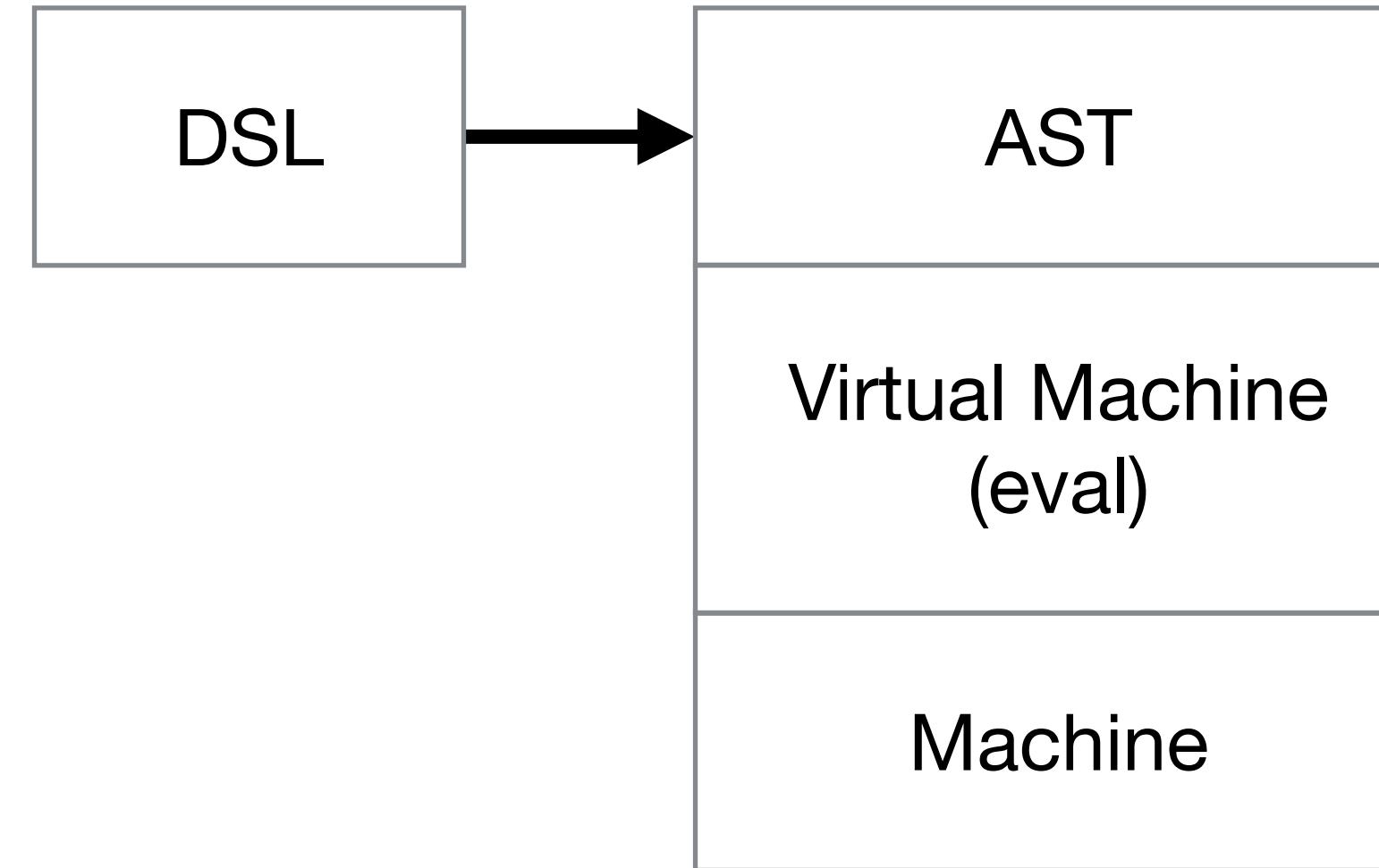
Not all "operations" can be intercepted

- Arithmetic operators

- Iteration operators

- Function definition?

- Type/class definition?

- Equality?

- Assignment?
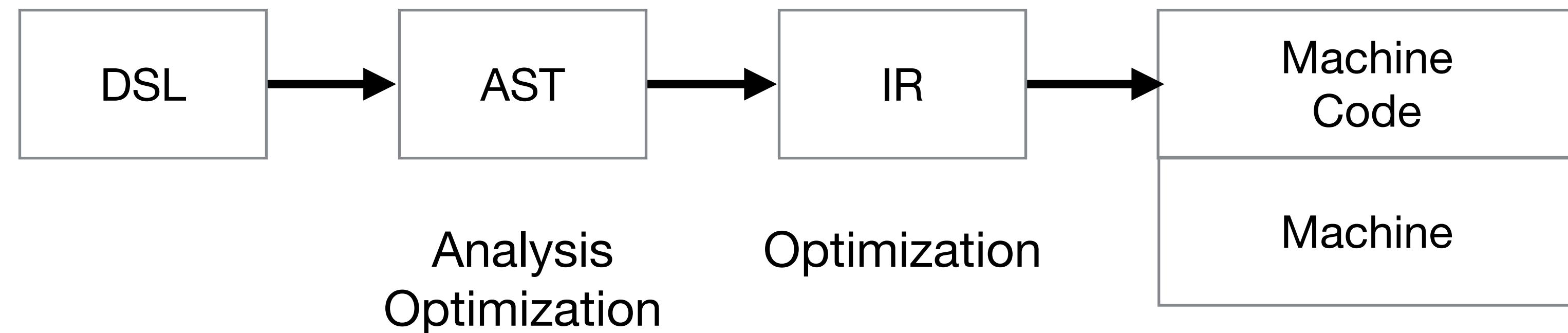
"Monkey patching" like this can be dangerous

Interpretation vs. Compilation

Interpreter

DSL	→	AST
		Virtual Machine (eval)
		Machine

Compiler

| DSL | → | AST | → | IR | → | Machine Code |
| | | Analysis Optimization | | Optimization | | Machine |

Mini-APL Assignment

- Implement simple array processing language in C++

- We provide recursive descent parser that builds an AST

- Lower the AST to LLVM; use LLVM to generate machine code!

- The LLVM Kaleidoscope tutorial contains most of what you need to know: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

- Assignment released today and due January 30th

LLVM Tutorial

Christophe Gyurgyik

The O.G. Paper

**LLVM: A Compilation Framework for
Lifelong Program Analysis & Transformation**

Chris Lattner   Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu/

"This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs."

Relevance

|	2014 —————————————————————————— 2024
# Contributors	336 2,138
Lines of Code	5M 36M
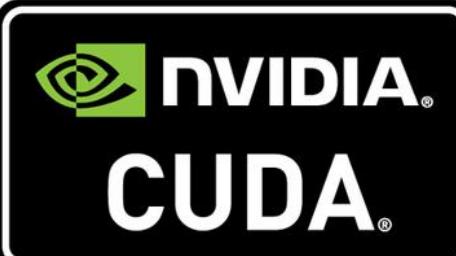
"In 2024 there were 37,486 commits to the LLVM repository...
Roughly inline with the 37.4~37.5k commits seen in 2022 and
2023. Those 37.4k commits added 9,339,334 lines of code
while removing 5,591,115 lines."

System Overview
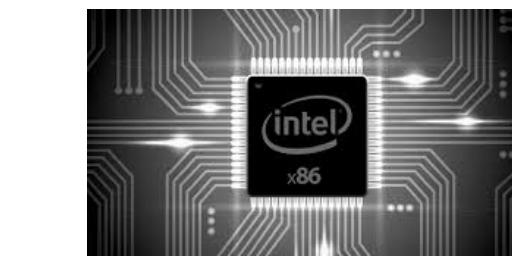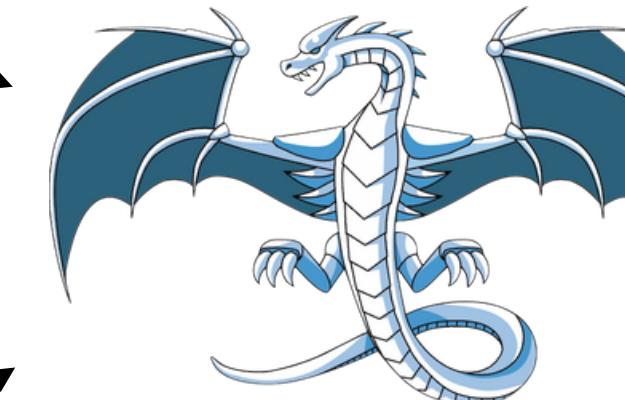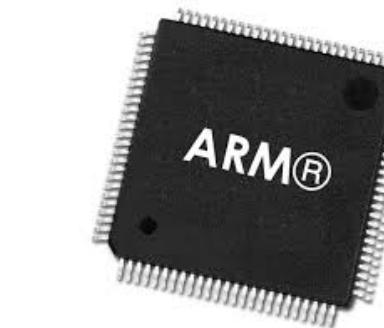
System Overview

System Overview

C or C++ → Front End → IR → Pass → IR → Pass → IR → Pass → IR → Back End → machine code

Clang

LLVM proper
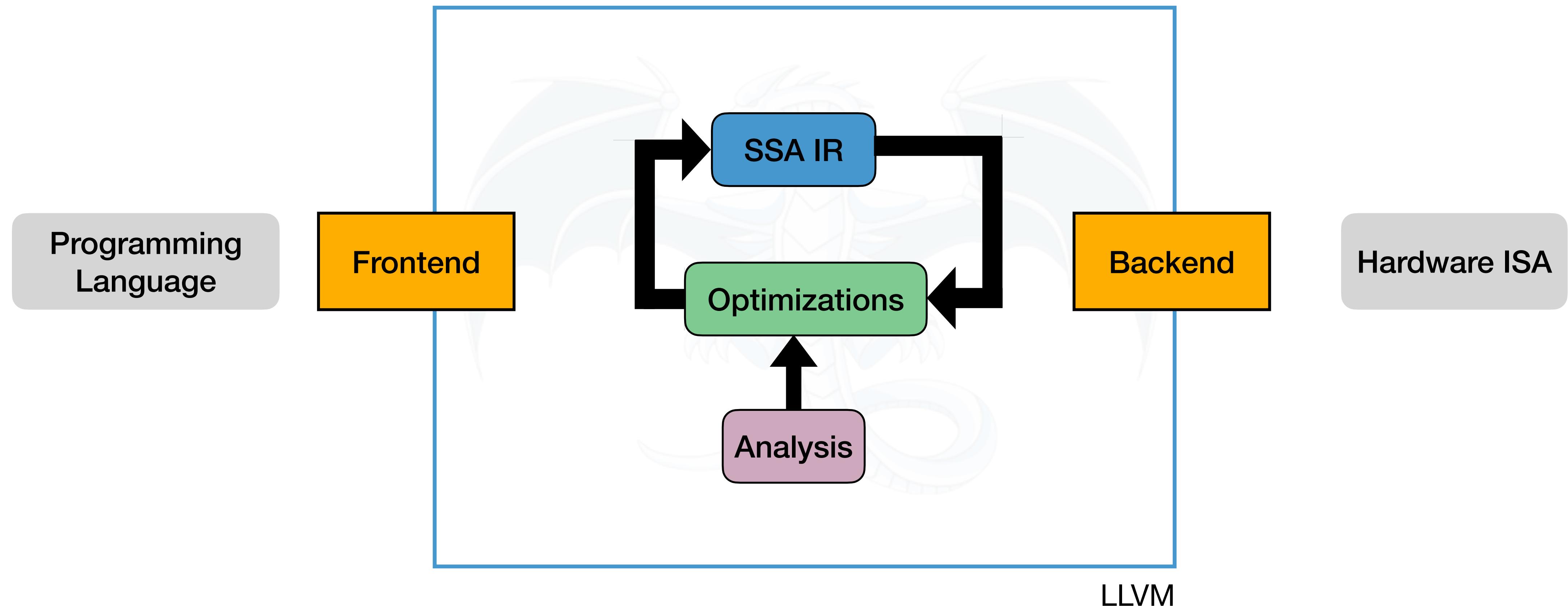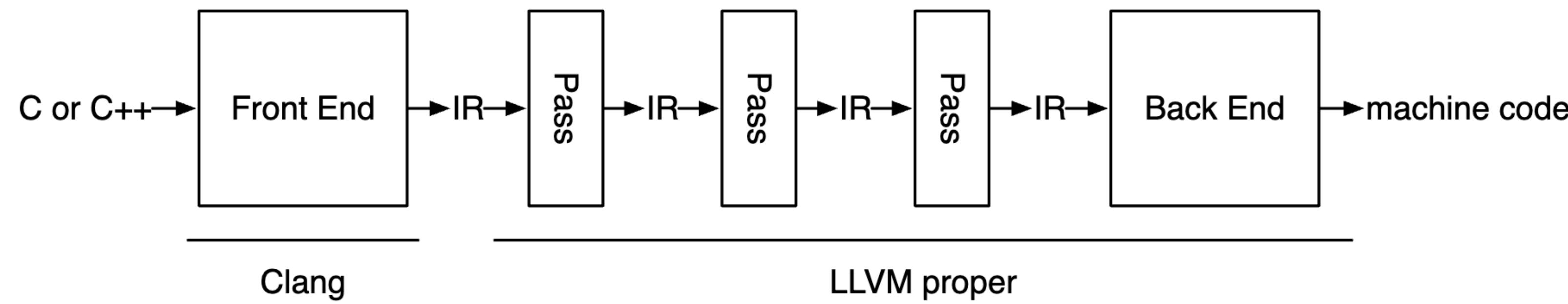
Static Single Assignment (SSA)

- Type of intermediate representation (IR)

  - Typically used for imperative languages.

  - (the other popular IR for general purpose languages is *Continuation-Passing Style* (CPS), in functional languages.

- **Key ingredient**: Every variable is assigned exactly once.

  - Implication: "for every use there is *one* definition."

Examples (whiteboard-ed in lecture)

```
%1 = constant i32 4
%2 = muli i32 %1, %1
%3 = subi i32 %2, %1
%4 = addi i32 %1, %2
ret i32 %4
```

```
define i32 @max(i32 %a, i32 %b) {
entry:
    %0 = icmp sgt i32 %a, %b
    br i1 %0, label %true, label %false
true:
    br label %exit
false:
    br label %exit
exit:
    %retval = phi i32 [%a, %true], [%b, %false]
    ret i32 %retval
}
```

LLVM IR Structure

LLVM IR Structure

Module

Top-level container.
Stores a list of functions, libraries, global variables, etc.

```
ModuleID = 'module'
```

```
auto context = std::make_unique<LLVMContext>();
auto module = std::make_unique<Module>("module", *context);
```

LLVM IR   LLVM Library

LLVM IR Structure

Function

Function type, linkage, and list of basic blocks

```
ModuleID = 'module'
define i32 @foo() {
}
```

```
auto context = std::make_unique<LLVMContext>();
auto module = std::make_unique<Module>("module", *context);

Type* intType = Type::getInt32Ty(*context);
FunctionType* functionType =
  FunctionType::get(intType, /*isVarArg=*/false);

Function* fooFunction = Function::Create(
  /*FunctionType=*/functionType,
  /*LinkageTypes=*/GlobalValue::InternalLinkage,
  /*N=*/"foo",
  /*M=*/module
);
```

LLVM IR   LLVM Library

LLVM IR Structure

Basic Block

A list of instructions and a terminator

```
ModuleID = 'module'
define i32 @foo() {
entry:
}
```

LLVM IR

```
auto context = std::make_unique<LLVMContext>();
auto module = std::make_unique<Module>("module", *context);

Function* fooFunction = Function::Create(…)

BasicBlock* entryBlock = BasicBlock::Create(
  *context,
  /*Name=*/"entry",
  /*Parent=*/fooFunction,
  /*InsertBefore=*/nullptr
);
```

LLVM Library

LLVM IR Structure

Instruction

The smallest unit: an abstraction for machine code

```
ModuleID = 'module'
define i32 @foo() {
entry:
  ...
  %0 = fmul double %lhs, %rhs
  %1 = call i32 @bar(%0, 42.0e+00)
  ret i32 %1
}
```

LLVM IR

```
...
BasicBlock* entryBlock = BasicBlock::Create(...);

IRBuilder<> b(*context);
b.SetInsertPoint(entryBlock);
...
Value* r0 = b.CreateFMul(lhs, rhs);
CallInst* r1 = b.CreateCall(barFunction, {r0, f42});
ReturnInst* r2 = b.CreateRet(r1);
```

LLVM Library

Can think of **%0** and and **%1** as typed registers. LLVM IR has *virtually* unlimited registers

More resources

- **[IMPORTANT]** **LLVM doxygen**
- **CS6120: Advanced Compilers:** compiler course by Adrian Sampson, uses a simpler version of LLVM IR (named *BRIL*)
  - aka "if you're going to work on industry compilers then you should know this"
- **Godbolt** (aka Compiler Explorer)
  - Formidable tool for exploring compilation of *many* different languages.
  - LLVM example**: max**
- **LLVM for Grad Students**: blogpost on LLVM passes (again by Adrian Sampson).
- **LLVM Weekly**
  - LLVM is always changing! Backwards compatibility is *not* a priority.