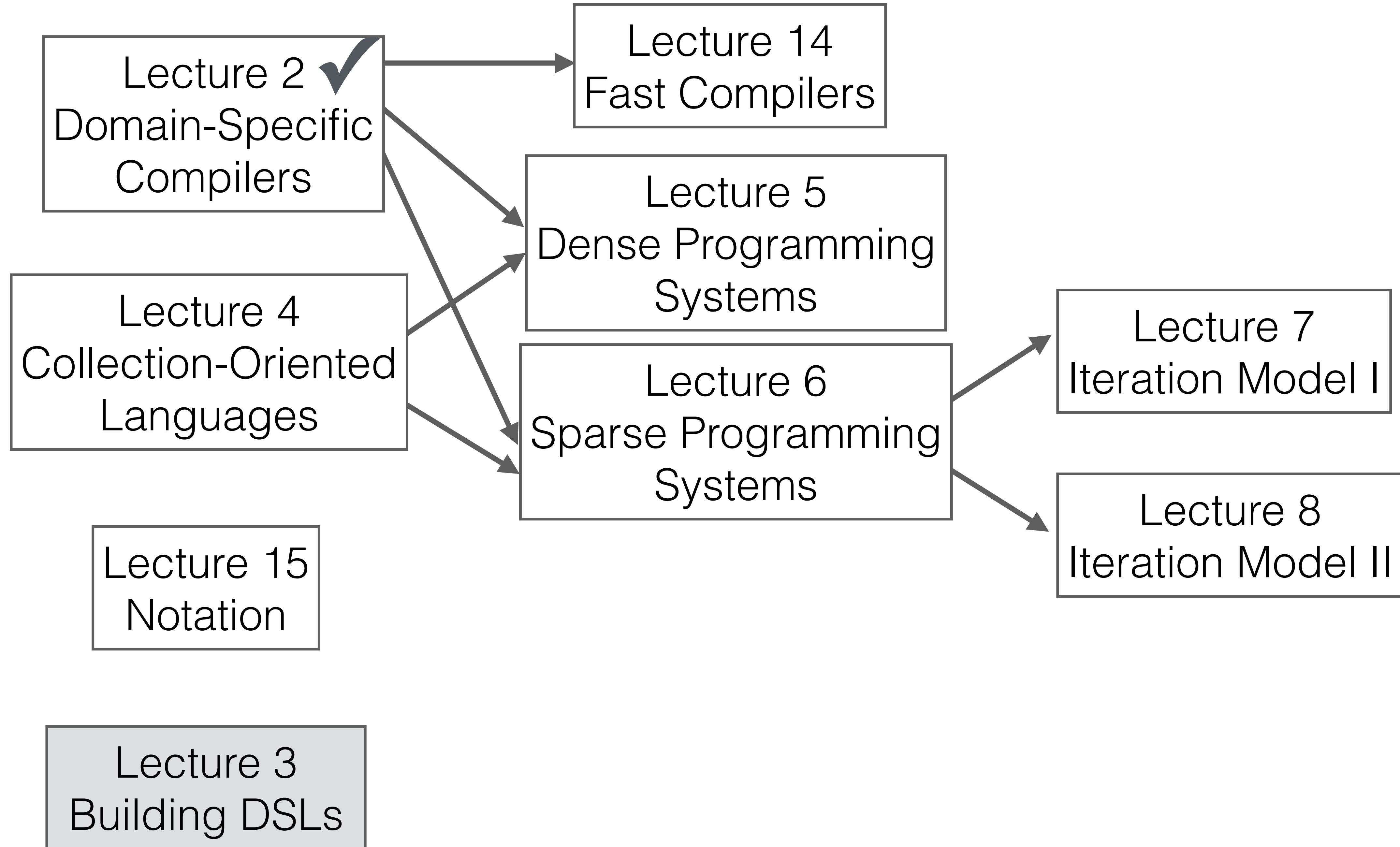


# Lecture 3 — Building DSLs

Stanford CS343D (Winter 2023)

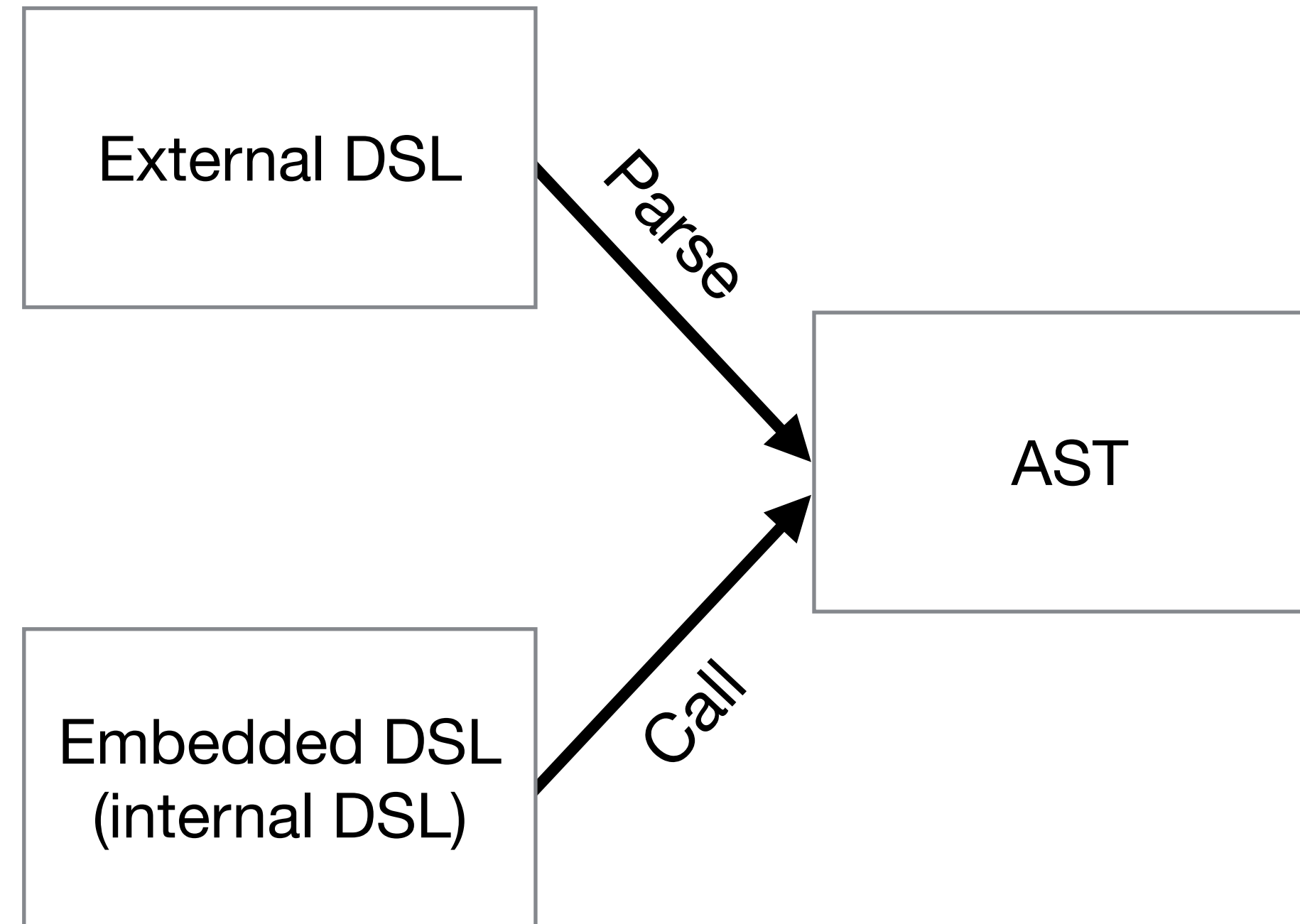
Fred Kjolstad

# Lecture Overview



# Types of DSLs – languages or libraries?

Implemented as standalone language

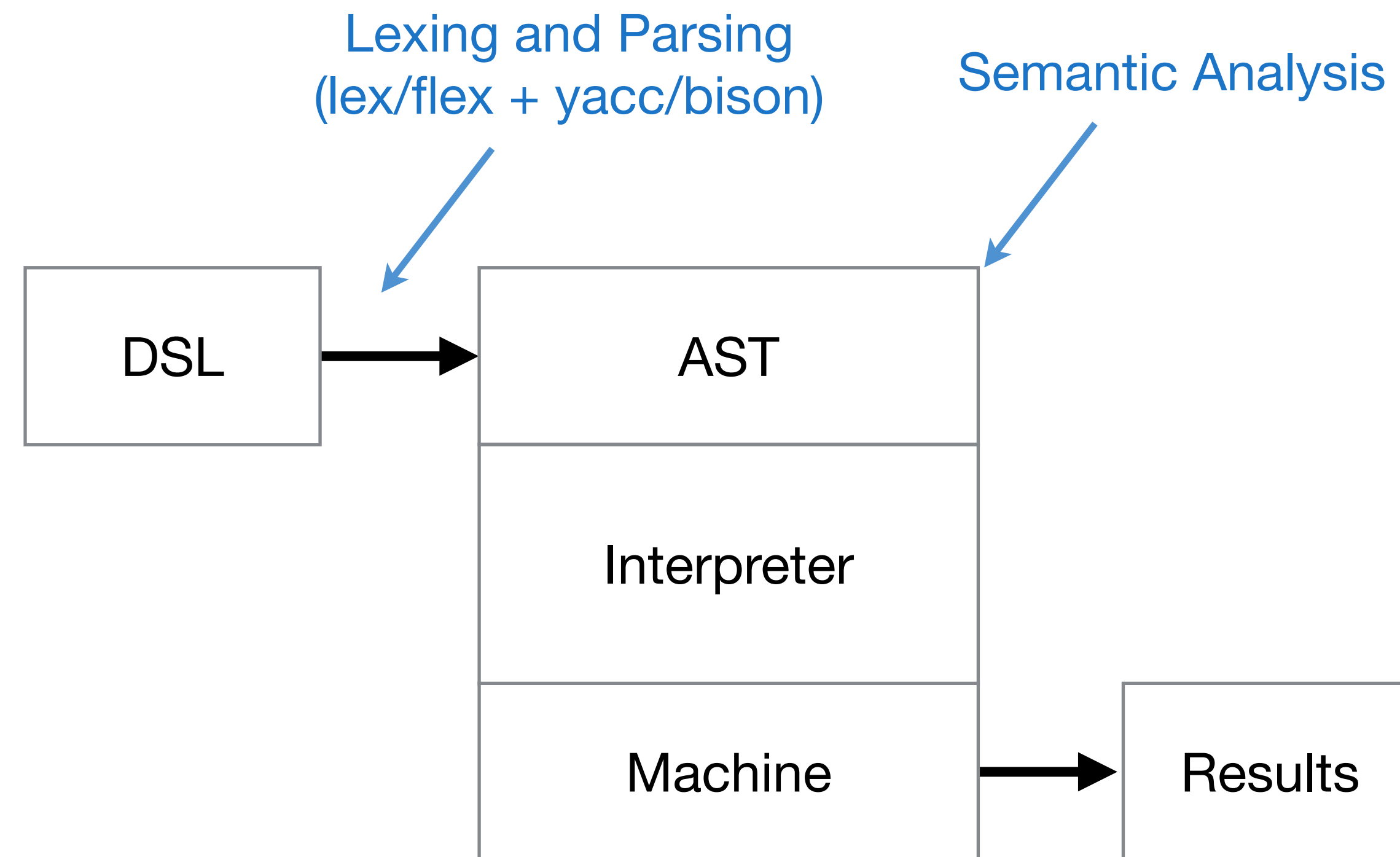


Embedded inside another language.  
Ideally the host language has features to  
make it easy to embed DSLs.

# External DSLs

That is, DSLs as textual languages

# External DSLs – Implementation



# External DSLs — Demo

calc1.py

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

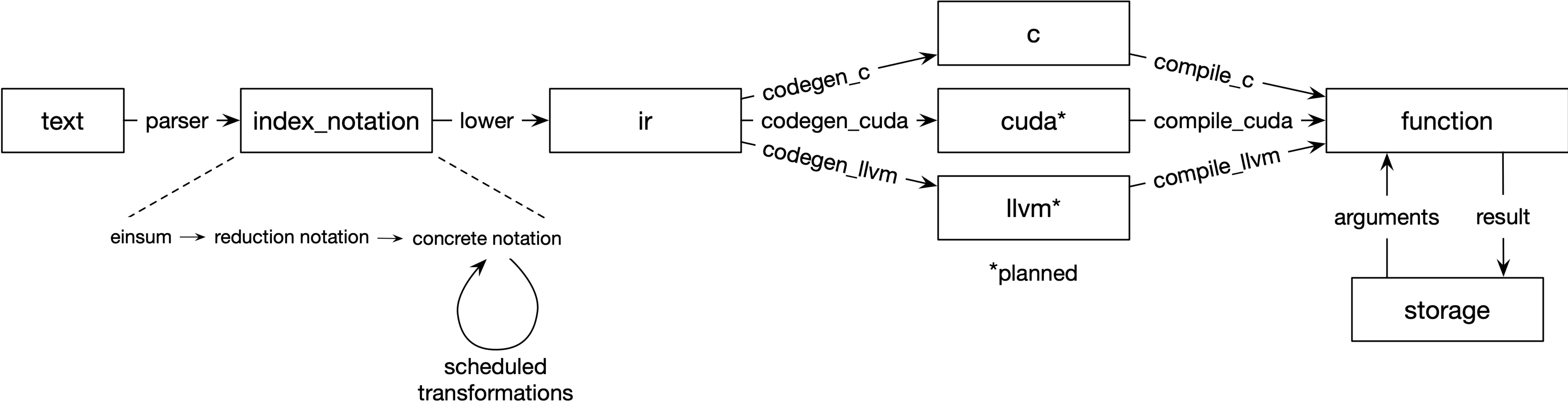
[https://www.d3-graph-gallery.com/graph/density\\_basic.html](https://www.d3-graph-gallery.com/graph/density_basic.html)

<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;
```

# taco — many languages



# 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

# C# language designed for libraries and DSLs

**linq**

```
int count =  
    (from character in Characters  
     where character.Episodes > 120  
     select character).Count();
```

# 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



# DSL Construction Features

Type system: algebraic types or classes with inheritance

Polymorphism (multiple interpretation of the same AST)

Higher-order functions and lamdas (insert code)

Flexible syntax (e.g., operator overloading)

# 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

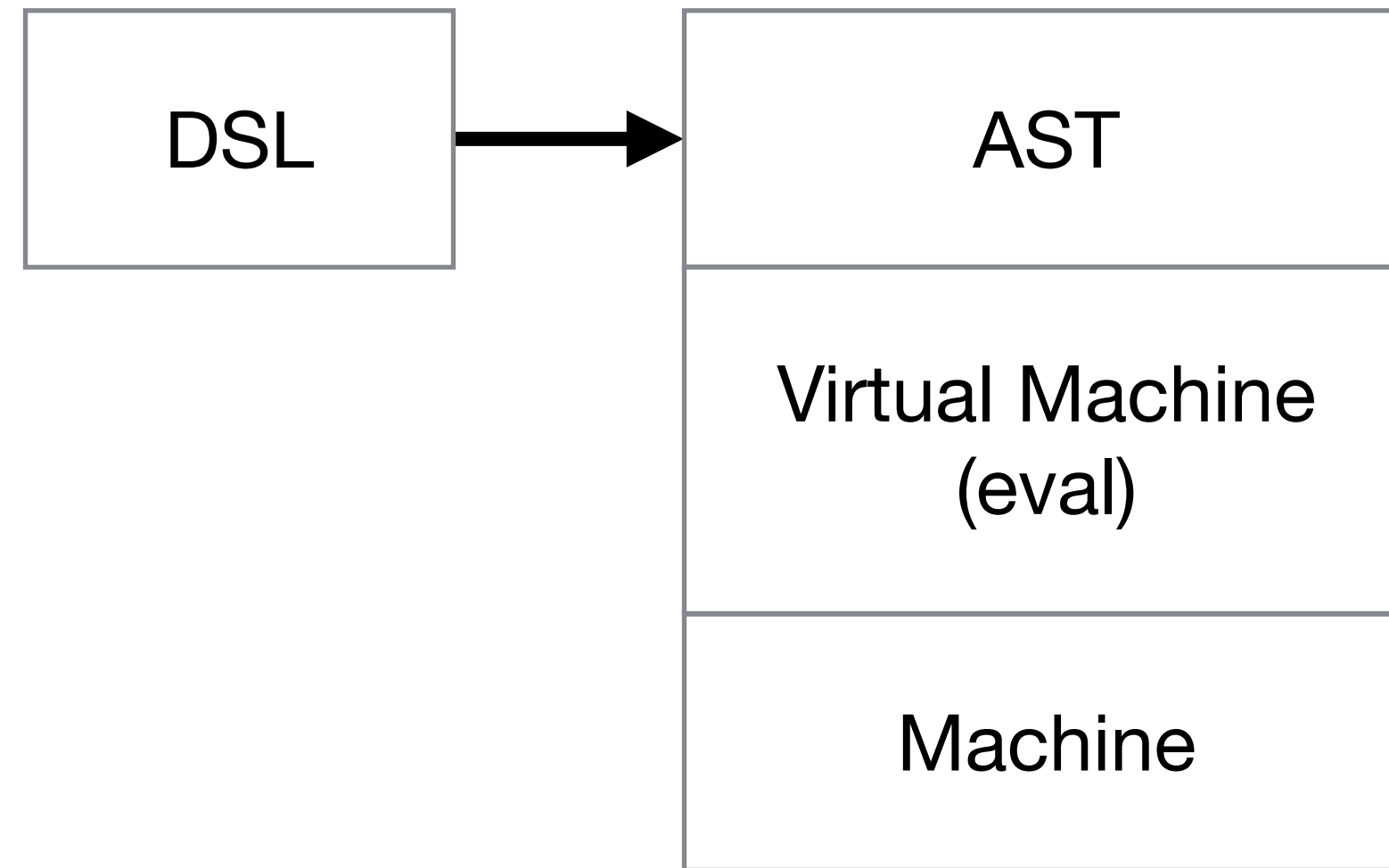
Not all “operations” can be intercepted

- Arithmetic operators
- Iteration operators
- Function definition?
- Type/class definition?
- Equality?
- Assignment?

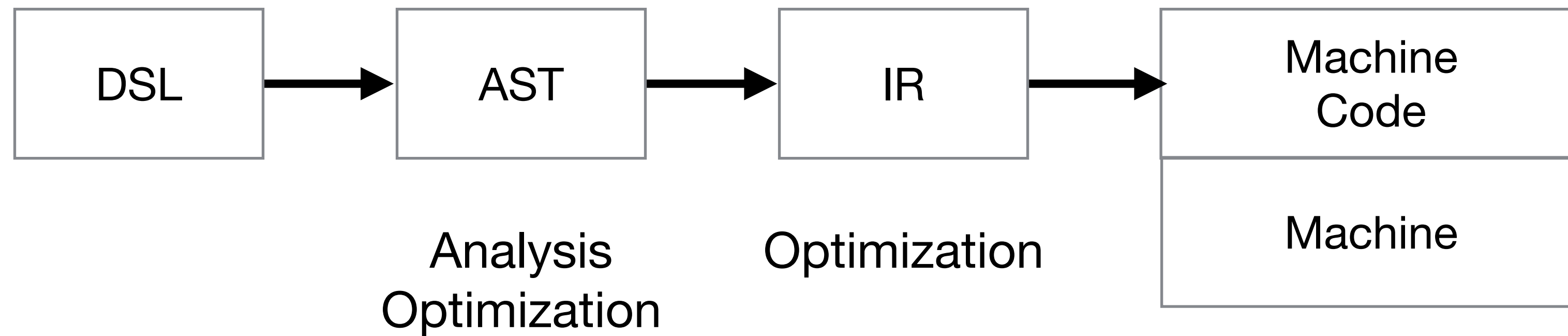
“Monkey patching” like this can be dangerous

# Interpretation vs. Compilation

**Interpreter**



**Compiler**



# 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 next Friday (Jan. 27)