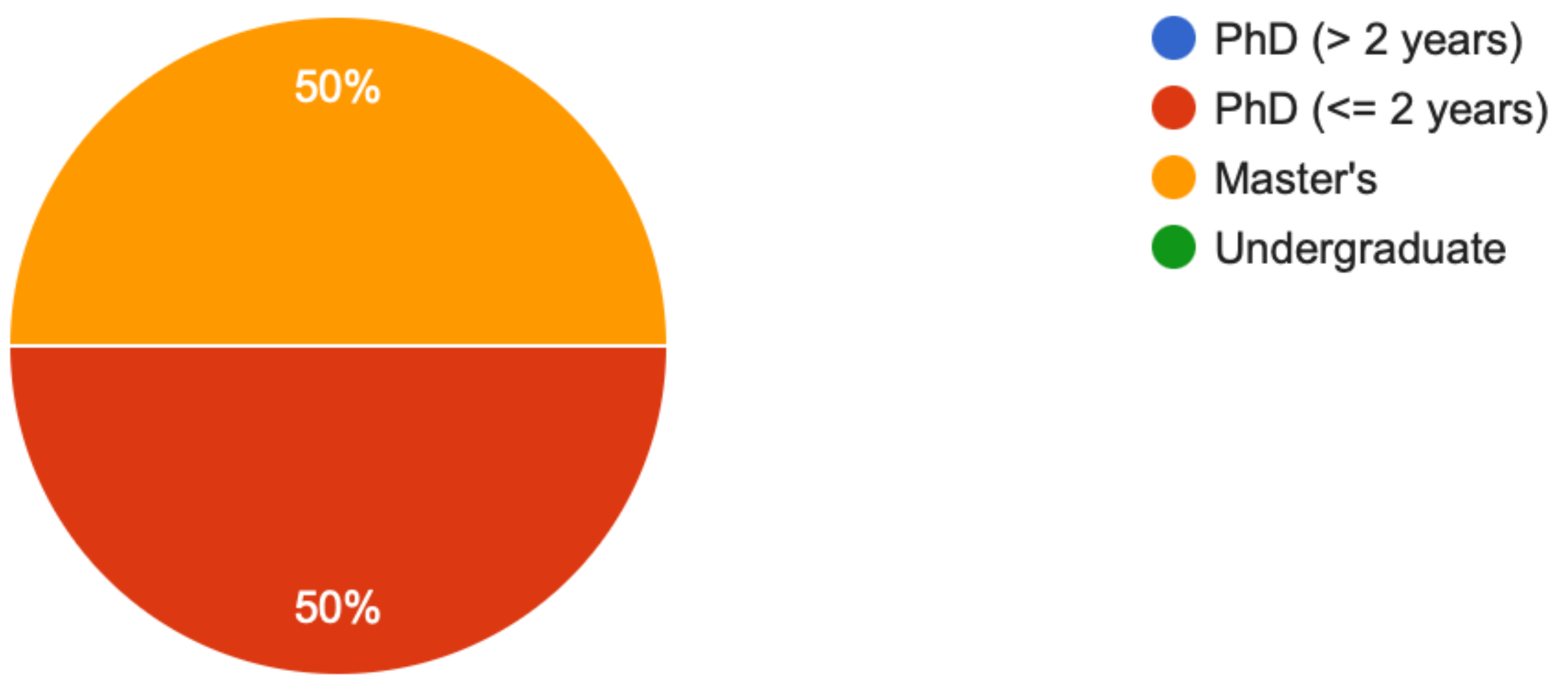
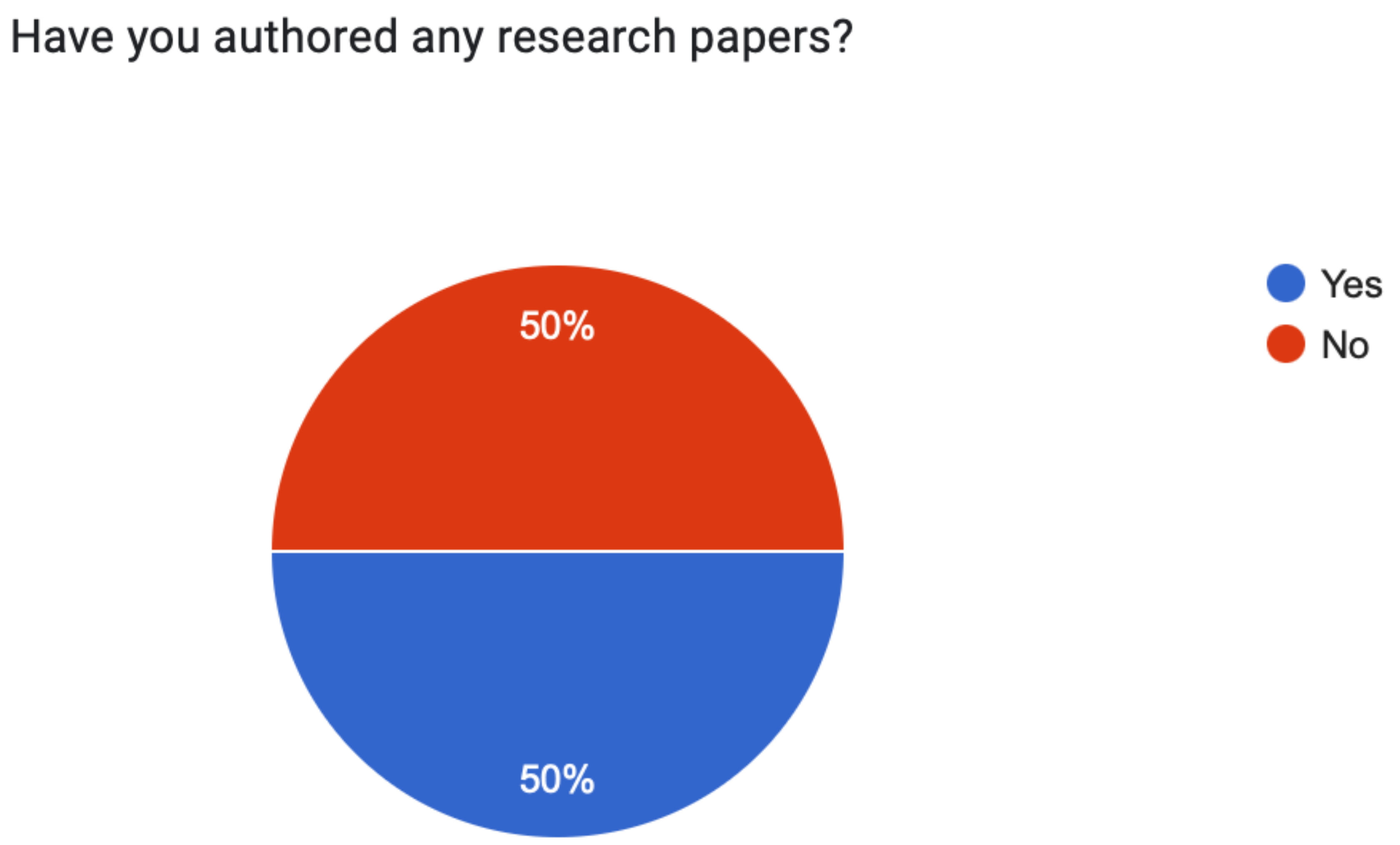
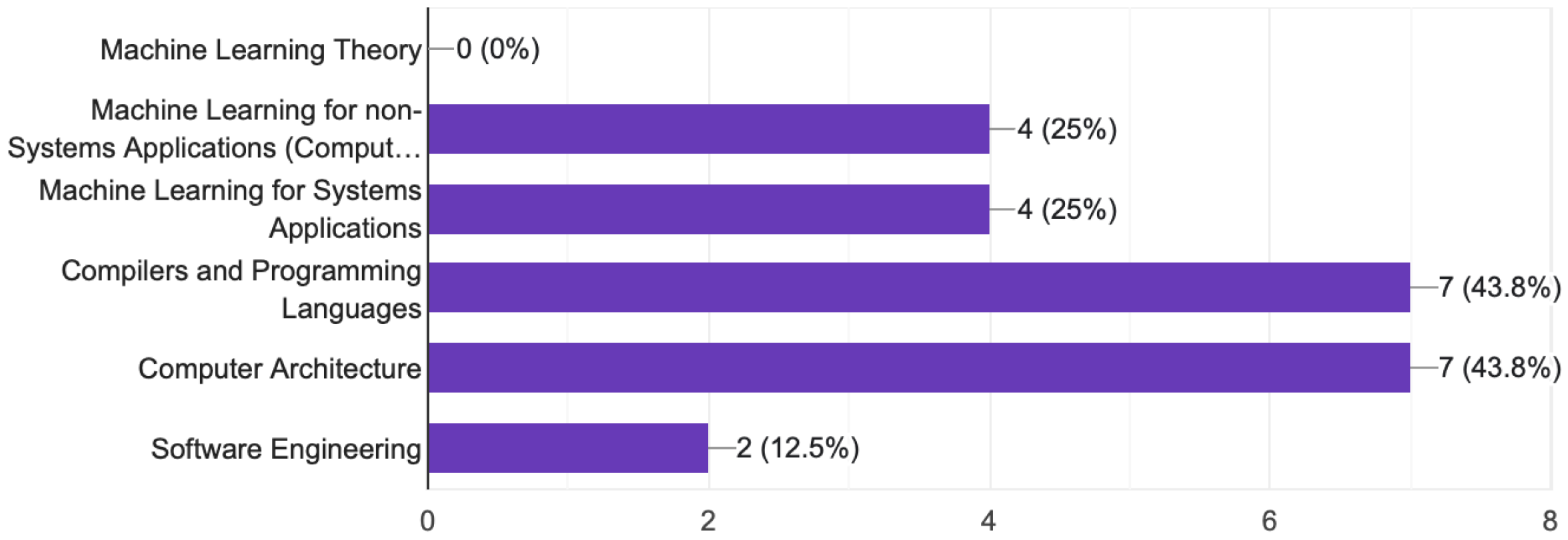
CS 598CM: ML for Compilers and Architecture

Instructor: Charith Mendis

Class Statistics Survey Results







Brief Announcements

- **Pre-requisites:** CS 426, CS 433, CS 421
  - The instructor lectures should be considered as crash courses
  - Willing to learn as you go
- **Reading List:** Up on the website
- **Paper Selections:** Due on August 31st; link on the website.

Lecture 2: Compilers

Crash-course + Optimizations

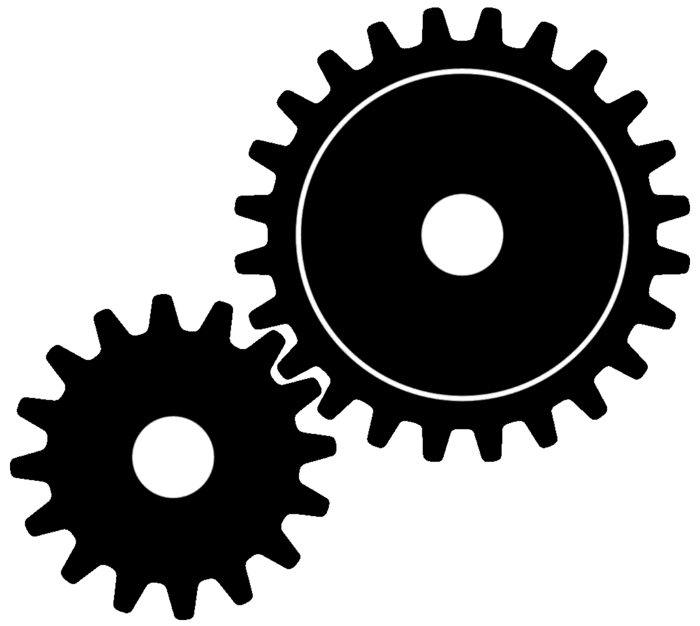
Compilers translate high-level languages to low-level machine code

Program

```
for (i = 0; i < grid_points[0]; i++)
  for (j = 0; j < grid_points[1]; j++)
    for (k = 0; k < grid_points[2]; k++)
      for (m = 0; m < 5; m++)
         add = u[i][j][k][m] − u_exact[m];
         rms[m] = rms[m] + add*add;
```

High-level programming language

Compiler



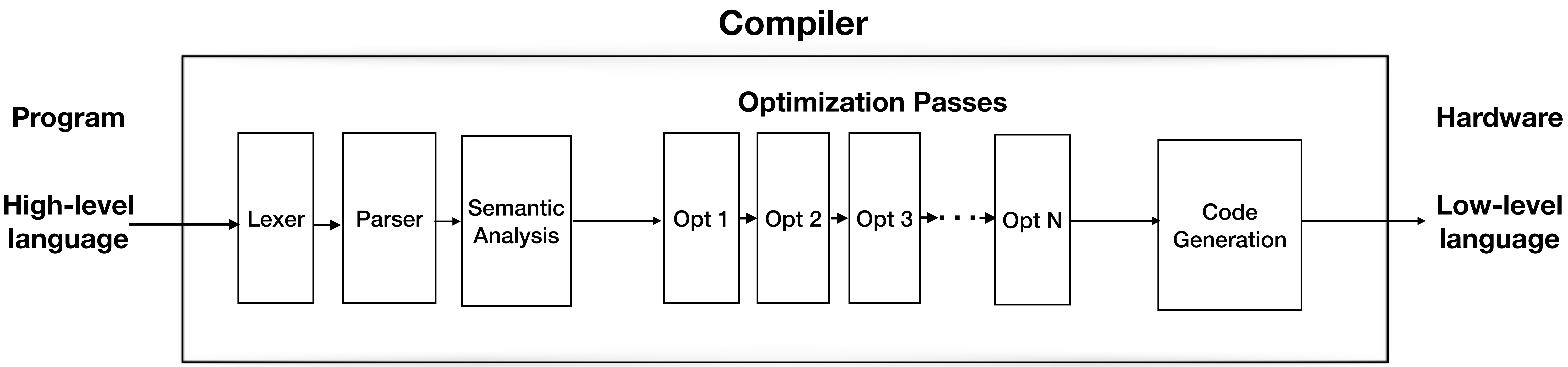
Hardware

```
......
addq    %rcx, %rax
movq    %rax, %rcx
salq    $6, %rcx
addq    %rcx, %rax
imulq   $21125, %rdi, %rcx
addq    %rax, %rcx
movq    %rdx, %rax
salq    $2, %rax
addq    %rsi, %rax
......
```

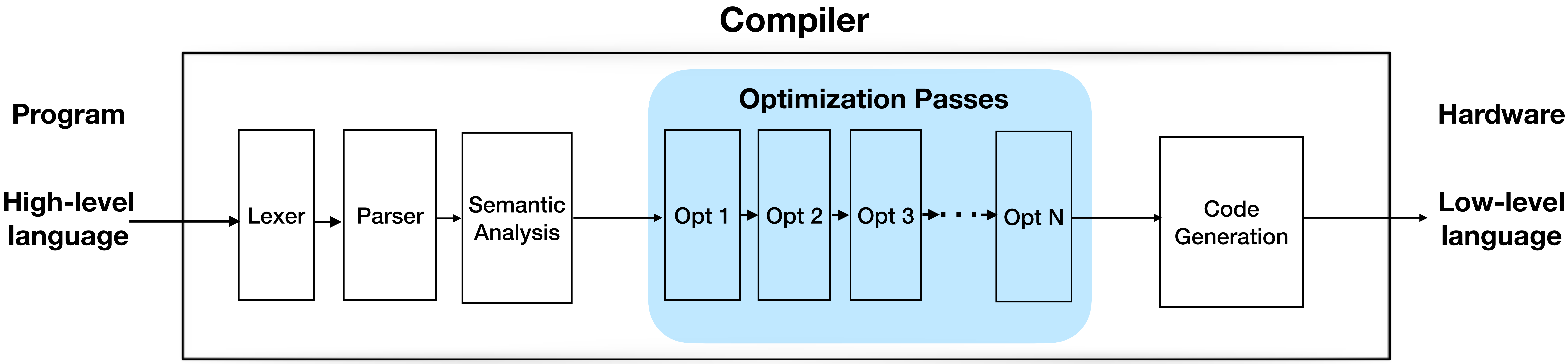
Low-level assembly language

Finding a semantic preserving (correct) translation that generates fast (optimized) code

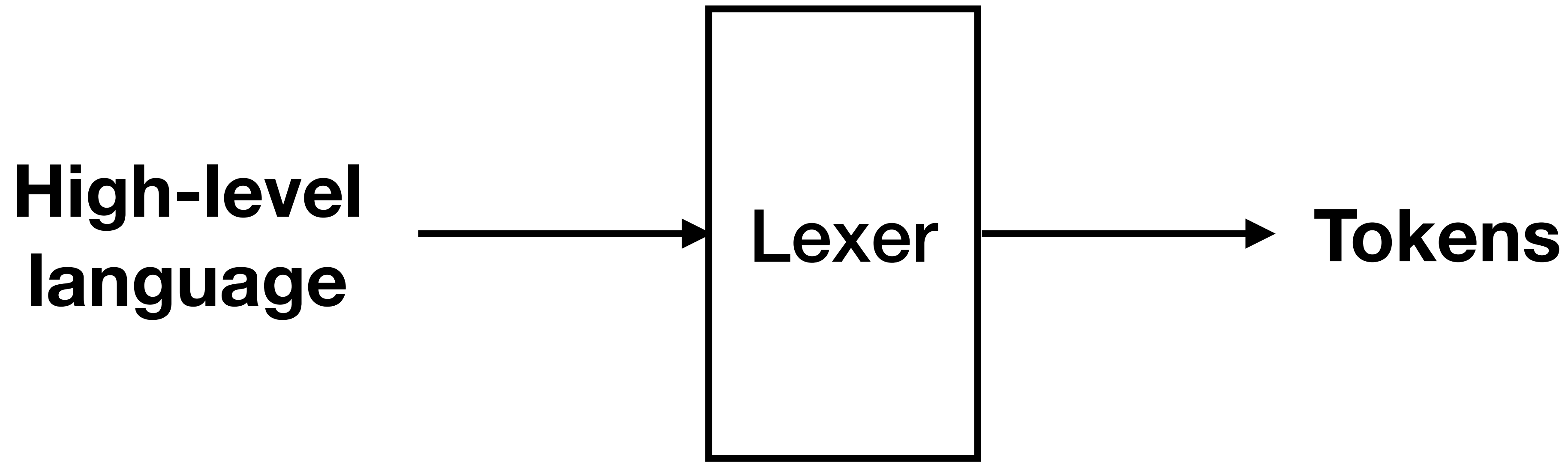
Stages of a Compiler



Stages of a Compiler



Lexer



List or stream of strings with syntactic meaning

```
for (int i= 0; i <100; i++){
   A[i] = A[i+1] + 1;
}
```

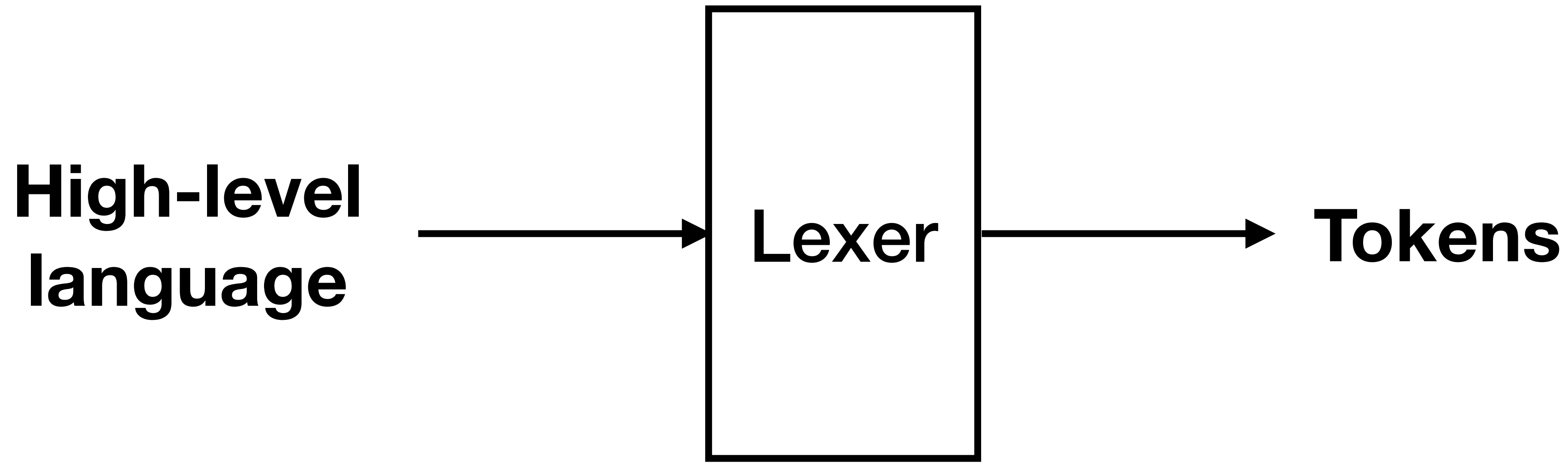
<for> <(> <int> ... <A> <[> <i> <]> ...

Keywords

Separators

Identifiers

Lexer



What errors does lexer catch?
Usually lexer produces tokens from regular languages

```
for (int i= 0; i <100; i++){
   A[i] = A[i+1] + 1;
}
```

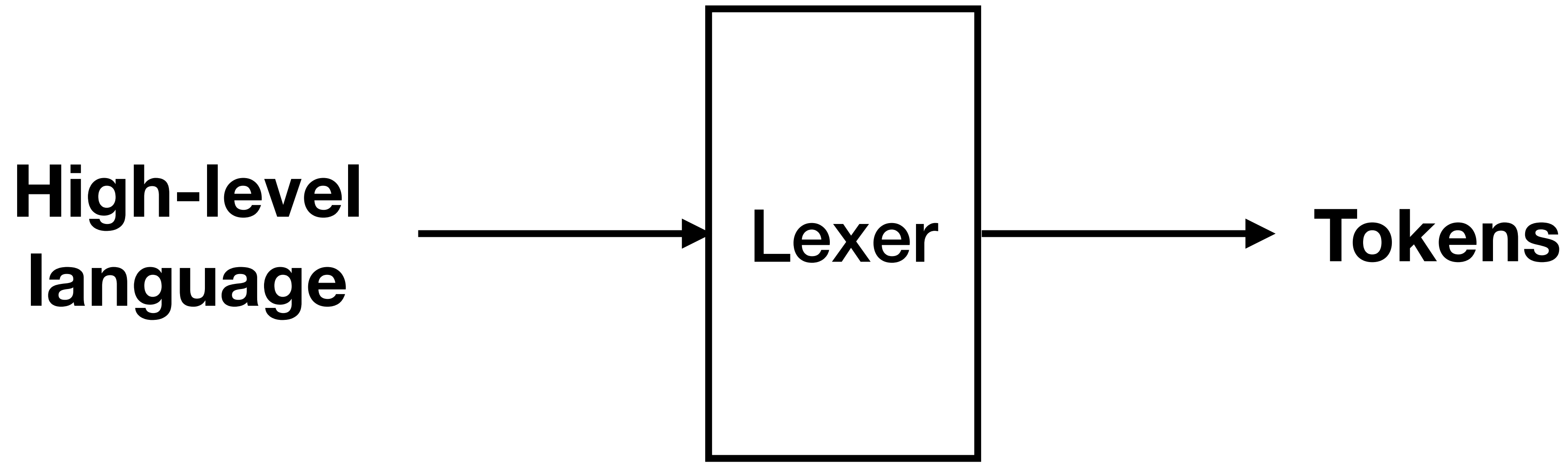
✓

```
for (int i= 0; i <100; i++){
   A[i = A[j+1] + 1;
}
```

```
for (int i= 0; i <100; i++){
   A[i] = A[j+1] + 1;
}
```

```
for (int i= 0; i <100n; i++){
   A[i = A[j+1] + 1;
}
```

Lexer



What errors does lexer catch?
Usually lexer produces tokens
from regular languages

```
for (int i= 0; i <100; i++){
   A[i] = A[i+1] + 1;
}
```

✓

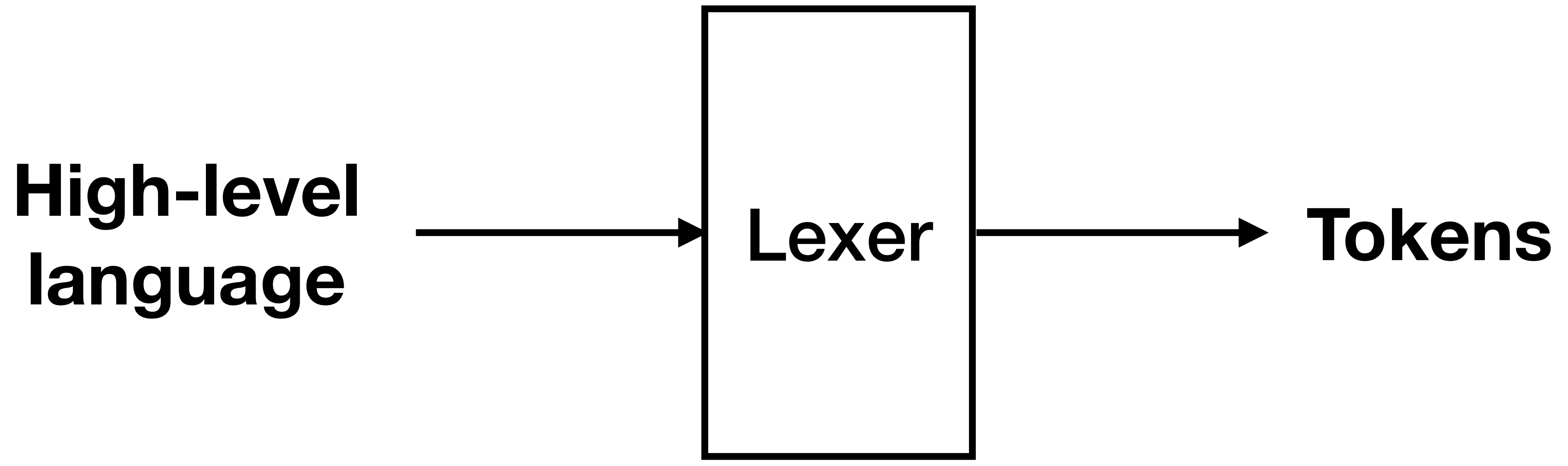
```
for (int i= 0; i <100; i++){
   A[i = A[j+1] + 1;
}
```

```
for (int i= 0; i <100; i++){
   A[i] = A[j+1] + 1;
}
```

✓

```
for (int i= 0; i <100n; i++){
   A[i = A[j+1] + 1;
}
```

Lexer



What errors does lexer catch?
Usually lexer produces tokens from regular languages

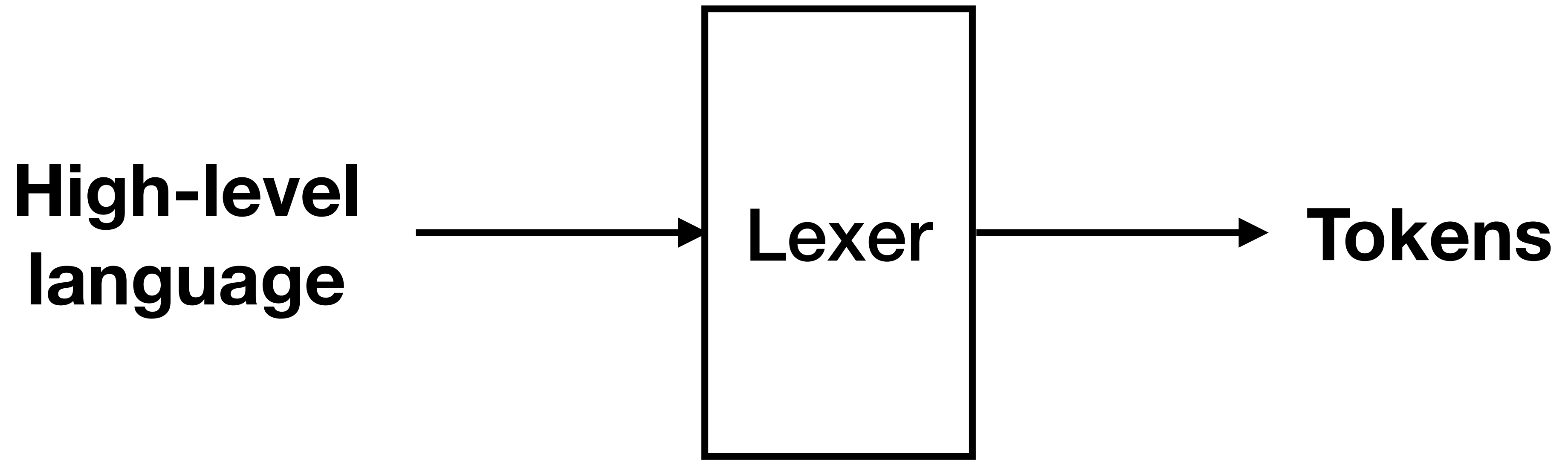
```
for (int i= 0; i <100; i++){
   A[i] = A[i+1] + 1;
}
```

```
for (int i= 0; i <100; i++){
   A[i?= A[j+1] + 1;
}
```

```
for (int i= 0; i <100; i++){
   A[i] = A[j+1] + 1;
}
```

```
for (int i= 0; i <100n; i++){
   A[i = A[j+1] + 1;
}
```

Lexer



What errors does lexer catch?
Usually lexer produces tokens from regular languages

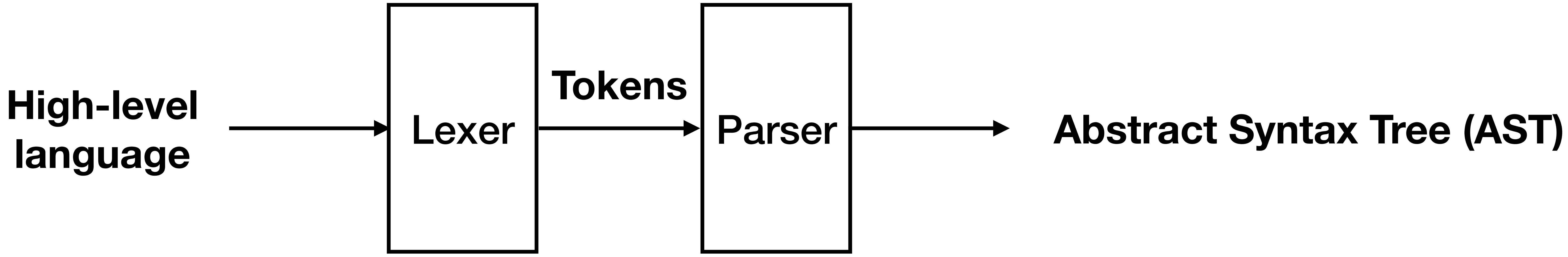
```
for (int i= 0; i <100; i++){
   A[i] = A[i+1] + 1;
}
```
✓

```
for (int i= 0; i <100; i++){
   A[i?= A[j+1] + 1;
}
```
✓

```
for (int i= 0; i <100; i++){
   A[i] = A[j+1] + 1;
}
```
✓

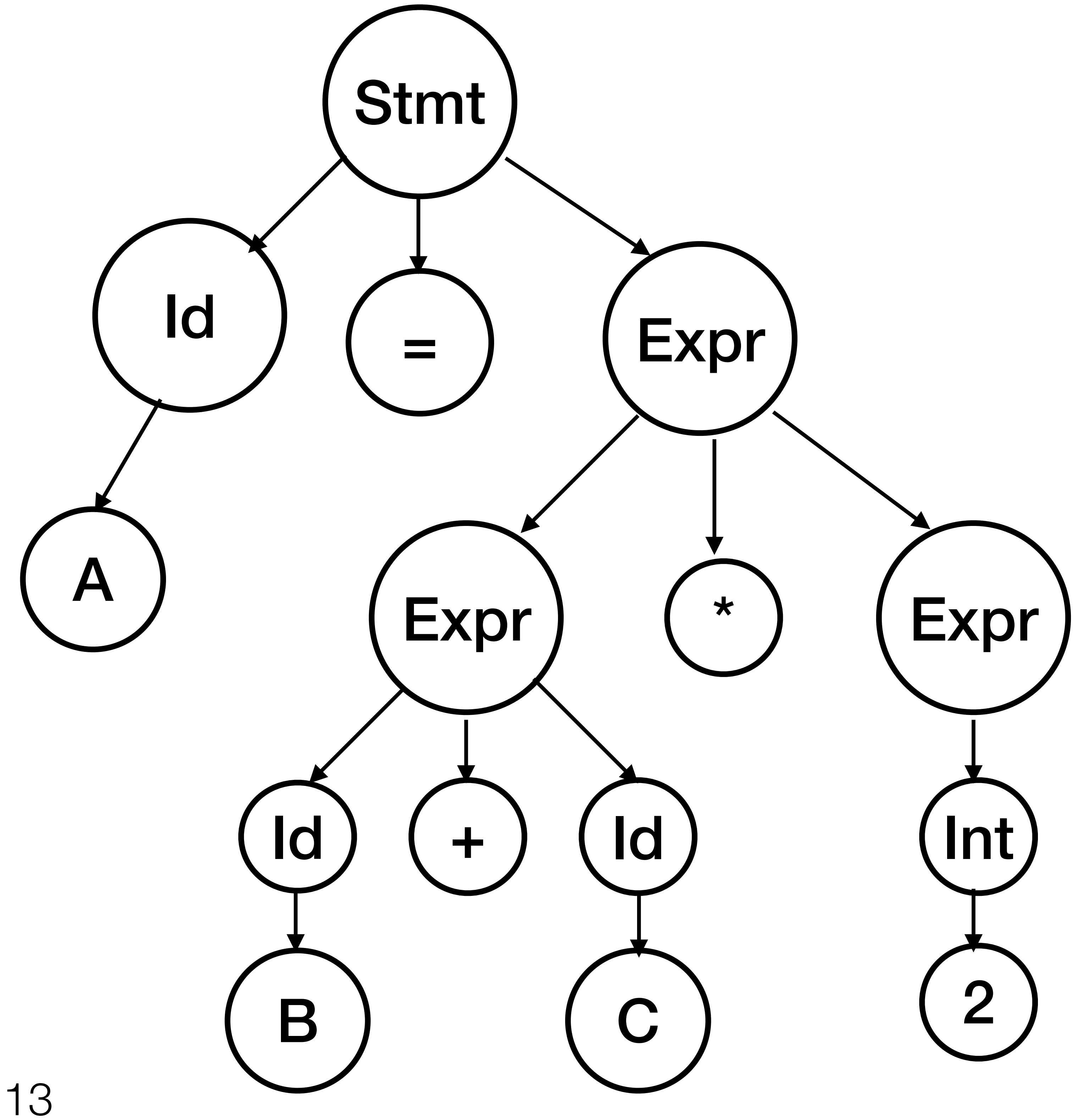
```
for (int i= 0; i <100n; i++){
   A[i = A[j+1] + 1;
}
```
✗

Parser



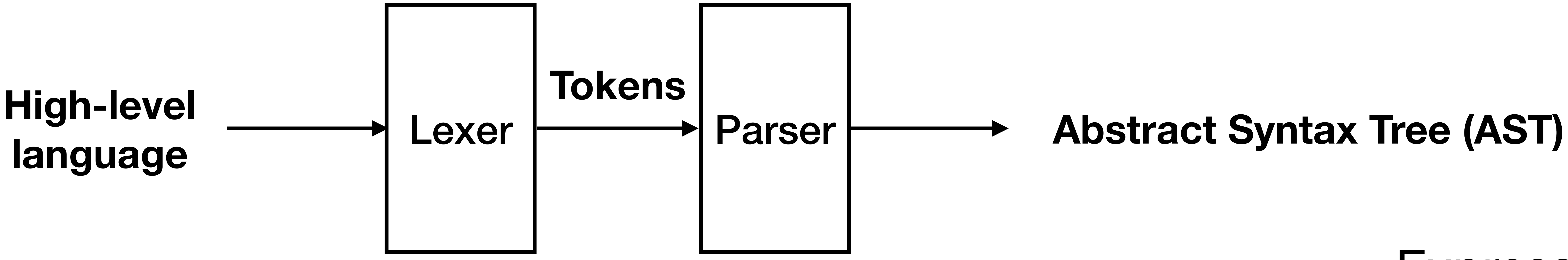
High-level language
Lexer
Tokens
Parser
Abstract Syntax Tree (AST)

A = (B + C) * 2;

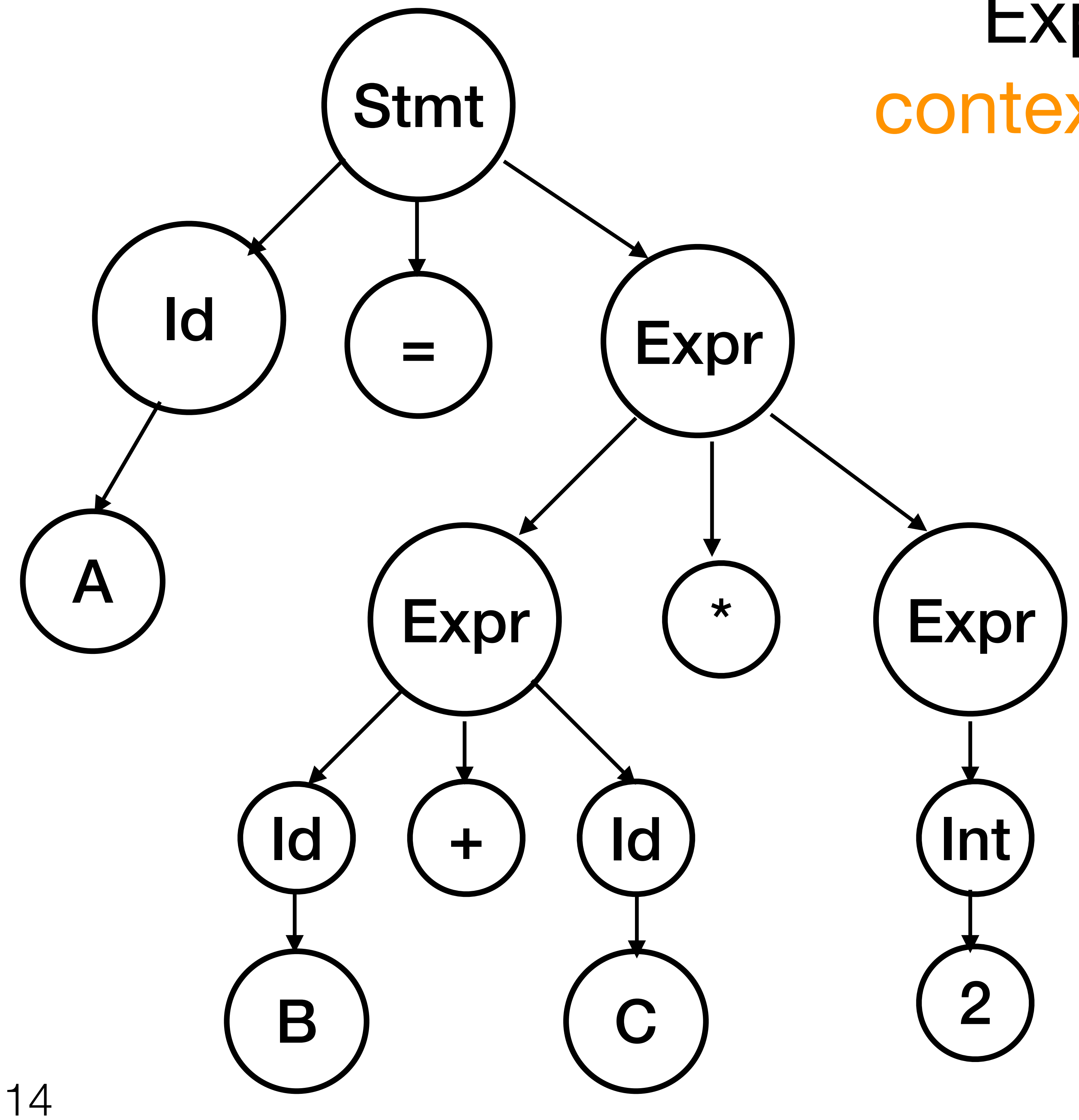


Stmt
Id
=
Expr
A
Expr
*
Expr
Id
+
Id
Int
B
C
2

Parser

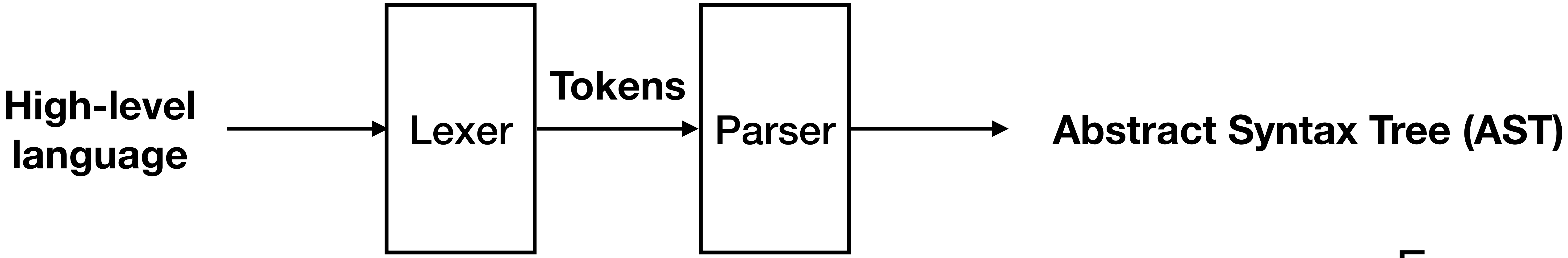


A = (B + C) * 2;

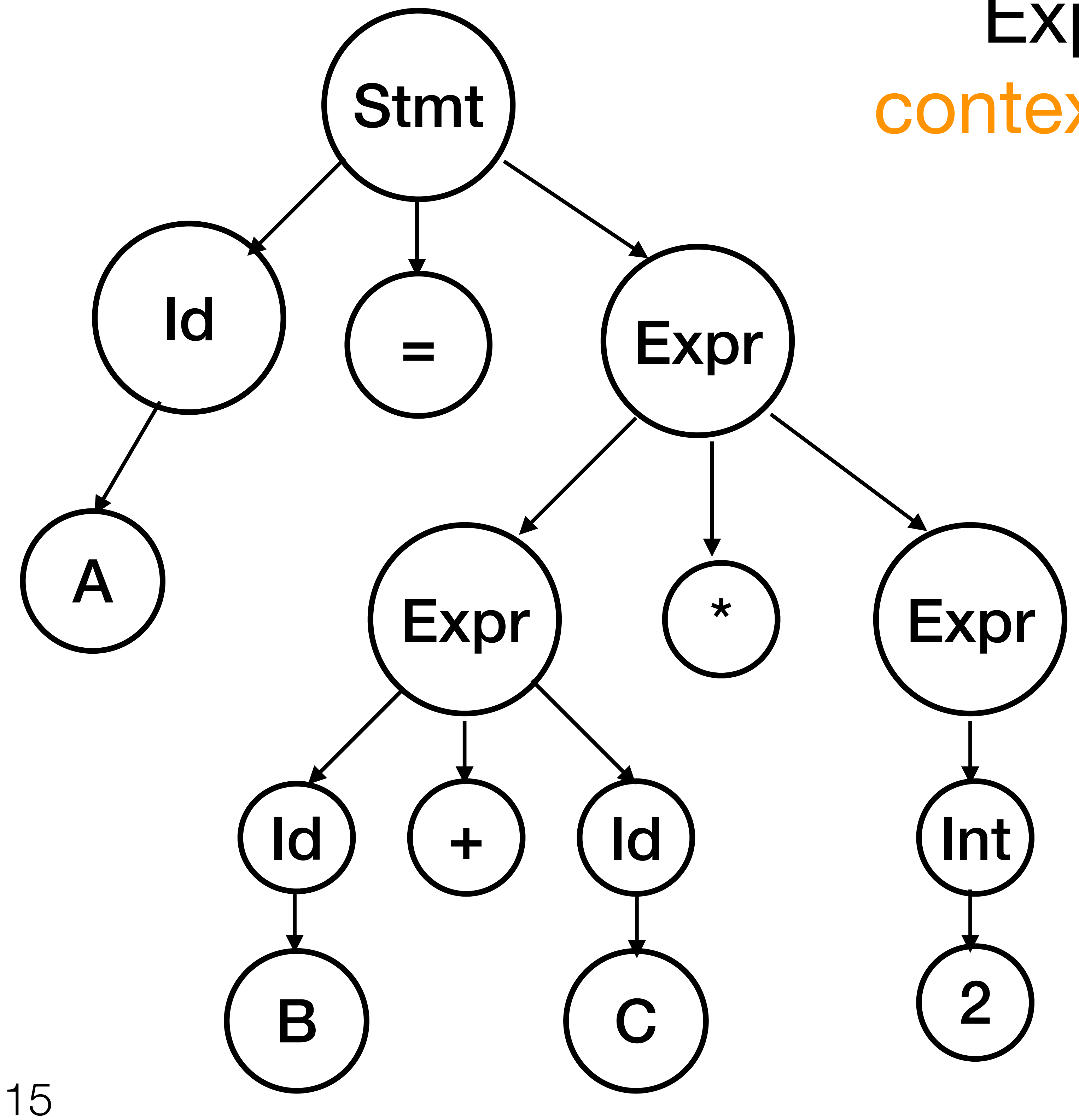


Expressed as a context-free grammar

Parser



Expressed as a context-free grammar

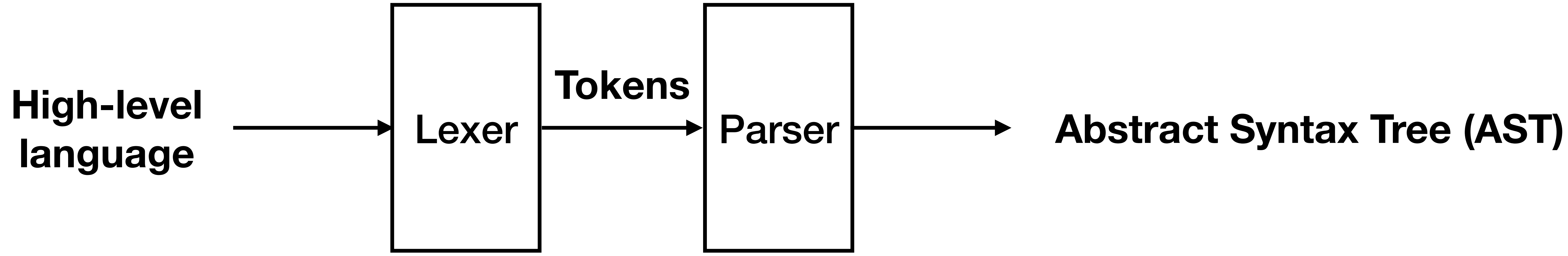


A = (B + C) * 2; ✓

A = (B + C * 2; ✗

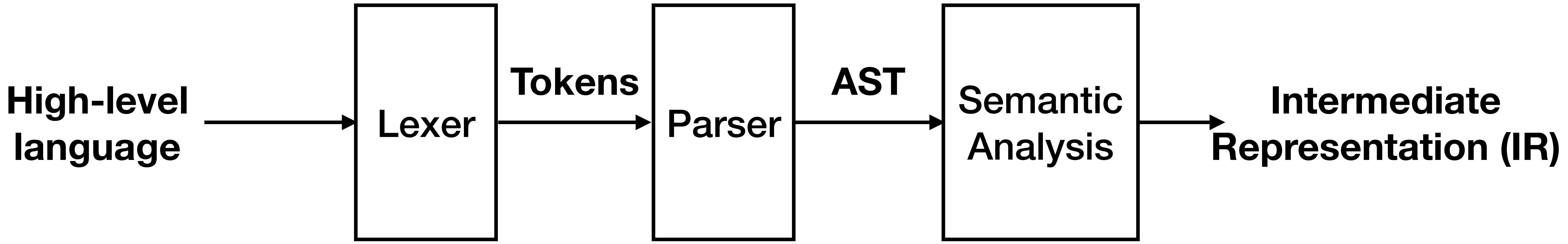
A = (B + C * 2 ✗

Parser



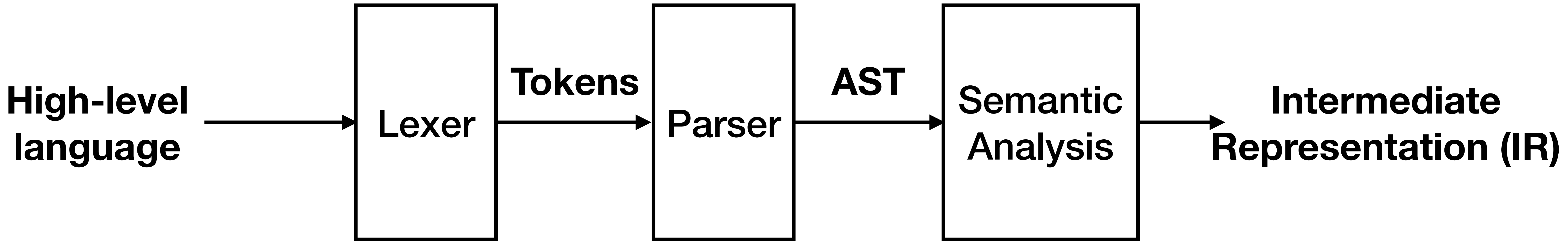
- Does not check if variables are defined
- Does not have scopes; variable bindings not defined
- Control flow or data flow information is not explicit

Semantic Analysis



- Clear variable bindings
- Control flow or data flow information embedded and queryable
- Focuses on the meaning of code (what computation does it perform?)
- Many IRs exist even in a single compiler

Semantic Analysis



- Clear variable bindings
- Control flow or data flow information embedded and queryable
- Focuses on the meaning of code (what computation does it perform?)
- Many IRs exist even in a single compiler

Semantics - we can now optimize!

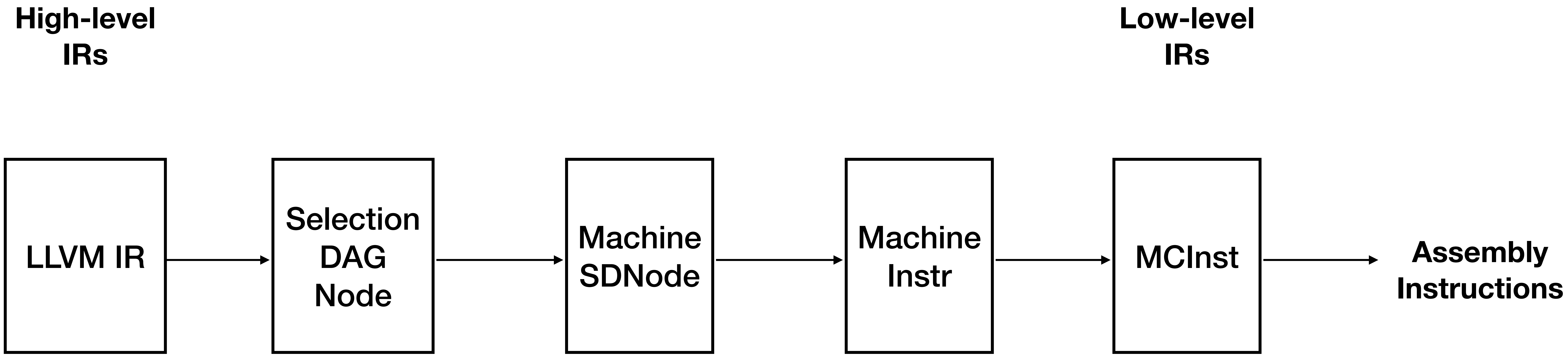
LLVM Intermediate Representation

```
def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}
```

- Each instruction has a clear meaning
- Control flow or data flow information embedded
- Data types encoded

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

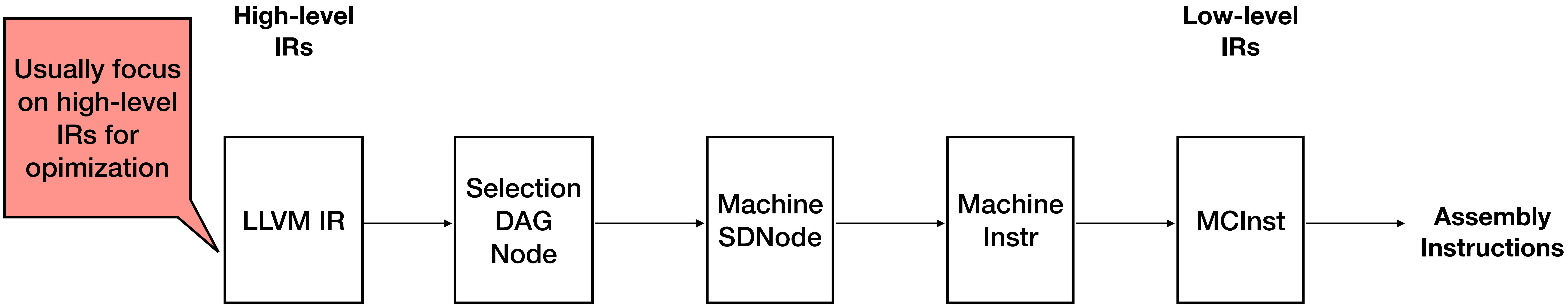
LLVM Intermediate Representation(s)



Compilers typically use many IRs through out code generation lifetime

https://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

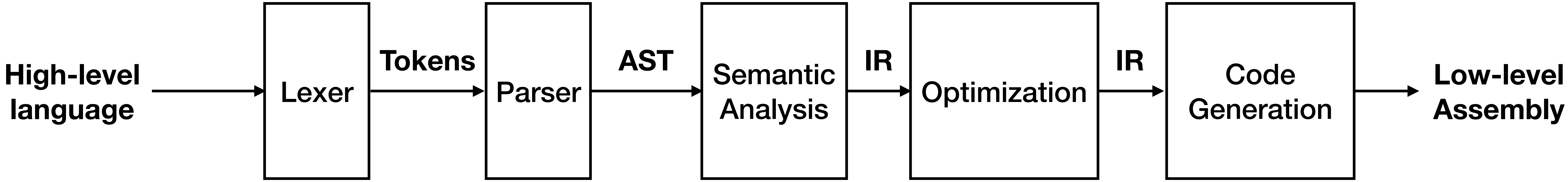
LLVM Intermediate Representation(s)



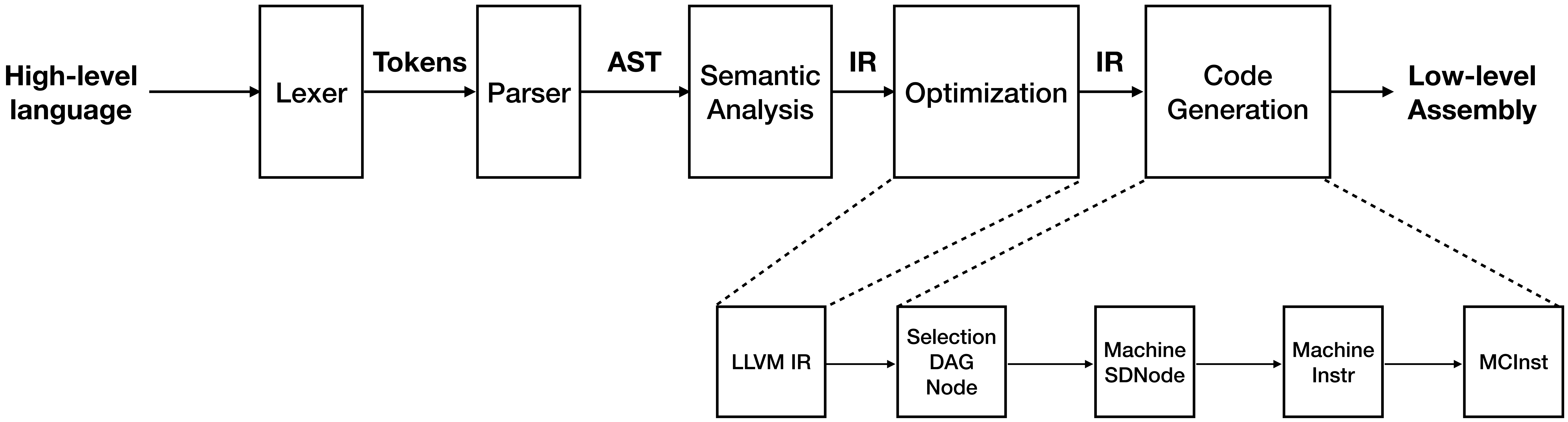
Compilers typically use many IRs through out code generation lifetime

https://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

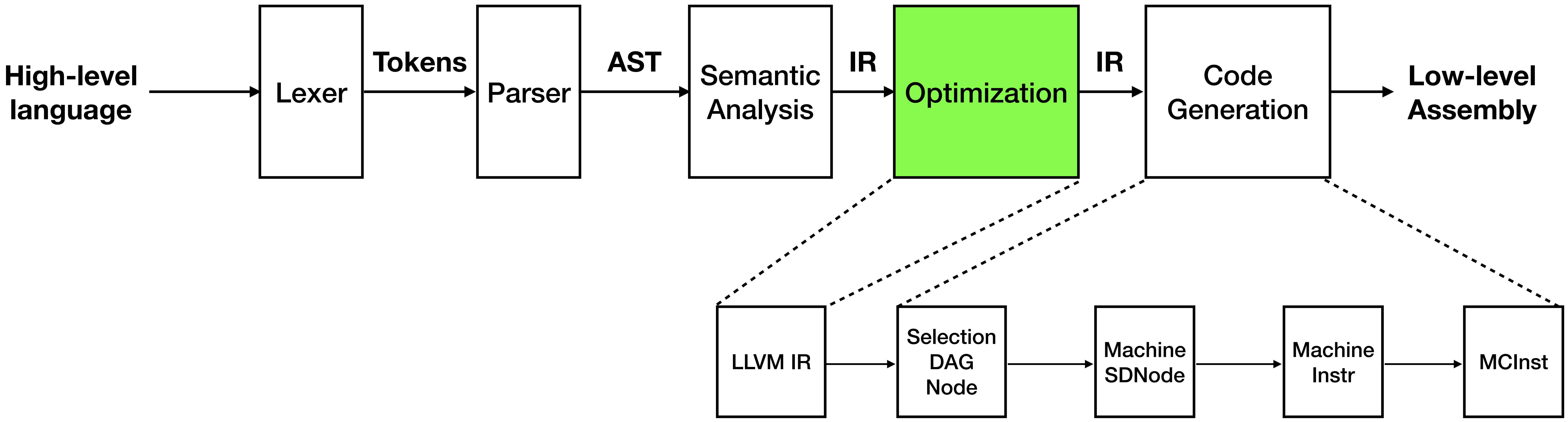
Finishing Up!



Finishing Up!



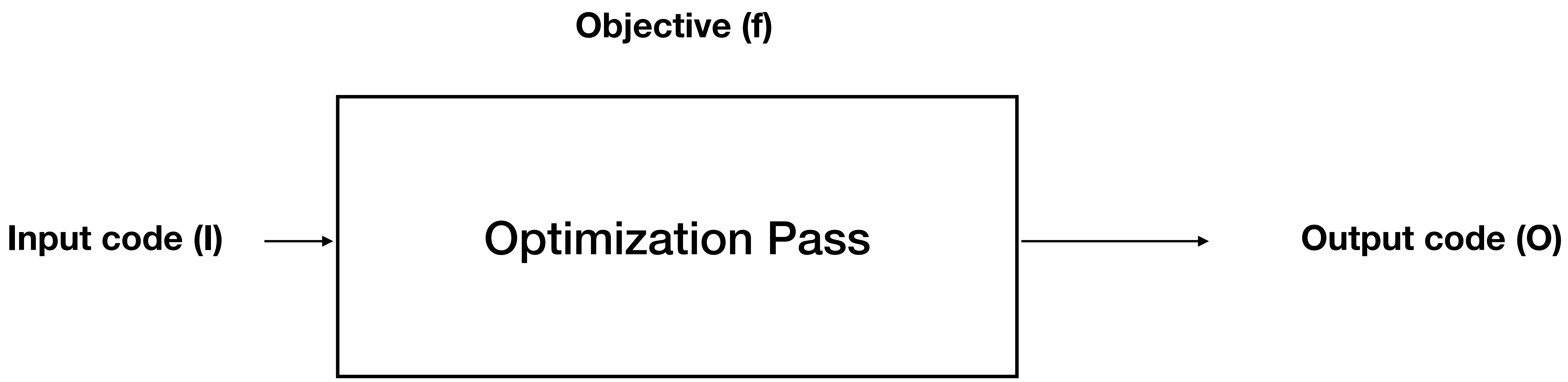
Wait we are just starting!



Code Optimization

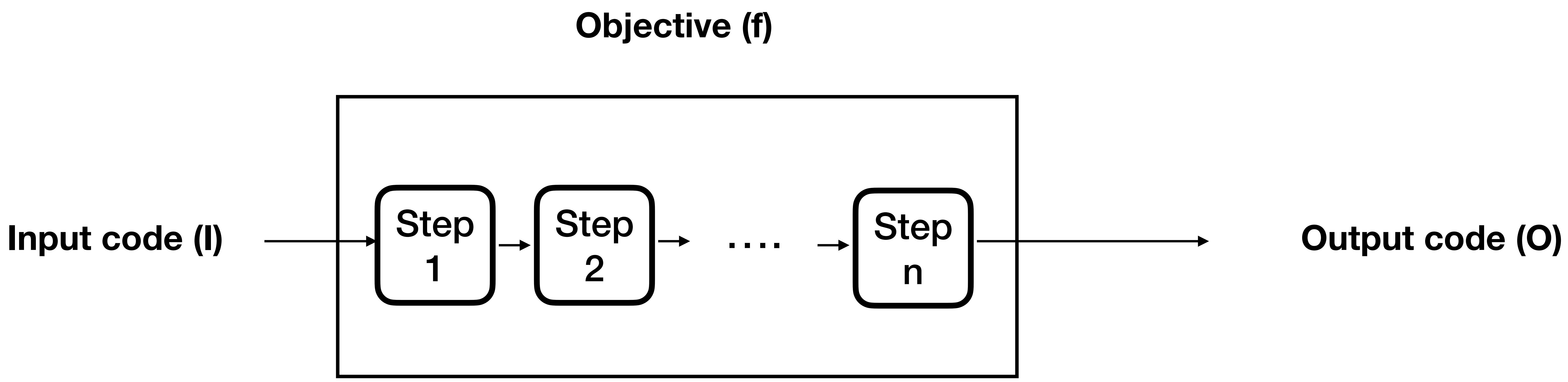
- We are going to spend most time on this in this course
- Usually performed as IR to IR transformations
- Optimizes for an objective or multiple objectives: `f(code)`
  - Runtime
  - Memory footprint
  - Energy consumption
  - Code Size

Two types of Optimizations



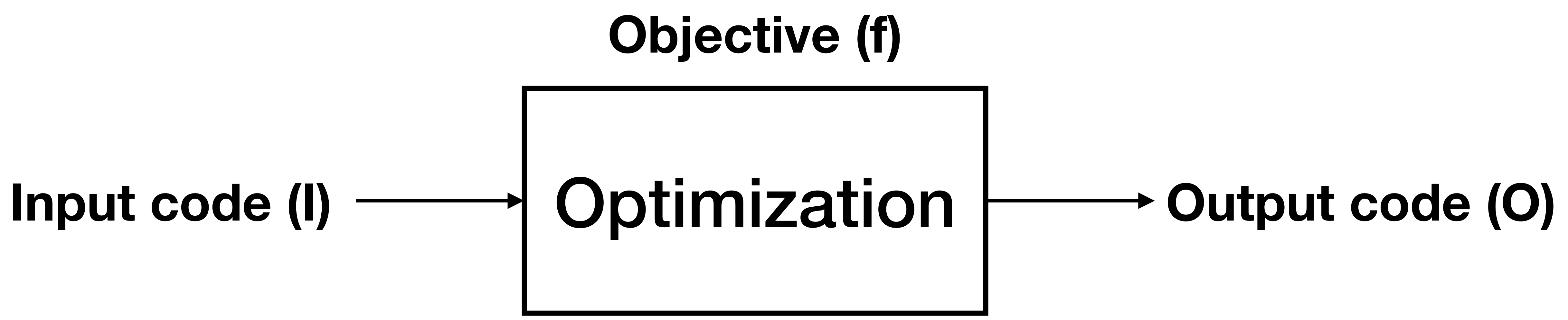
Goal : `f(O) > f(I); where > means better`

Two types of Optimizations



Goal : `f(O) > f(I); where > means better`

Two types of Optimizations



`Type I`

- Steps are always Profitable
  `f(O) > f(I)`
- Mostly independent

Dead Code Elimination, Constant Folding, Peephole Optimizations …….

`Type II`

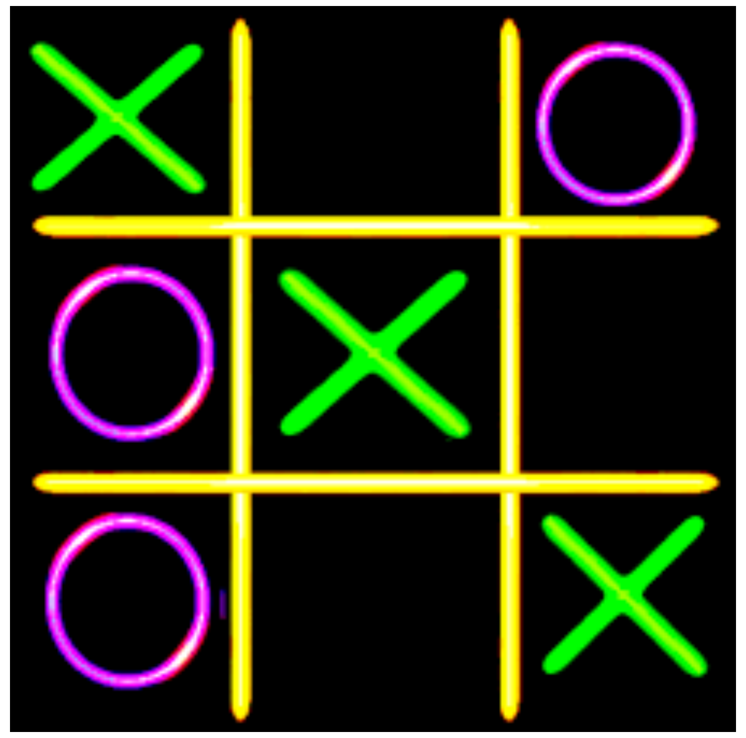
- Steps may not lead to global profitability
  `f(O) > f(I) ??`
- Mostly mutually-exclusive

Loop fusion, fission, unrolling, vectorization, parallelization…….

Gaming Analogy

`Type I`

Known strategy to at least draw
Newell and Simon (1972)



Tic-Tac-Toe

`Type II`

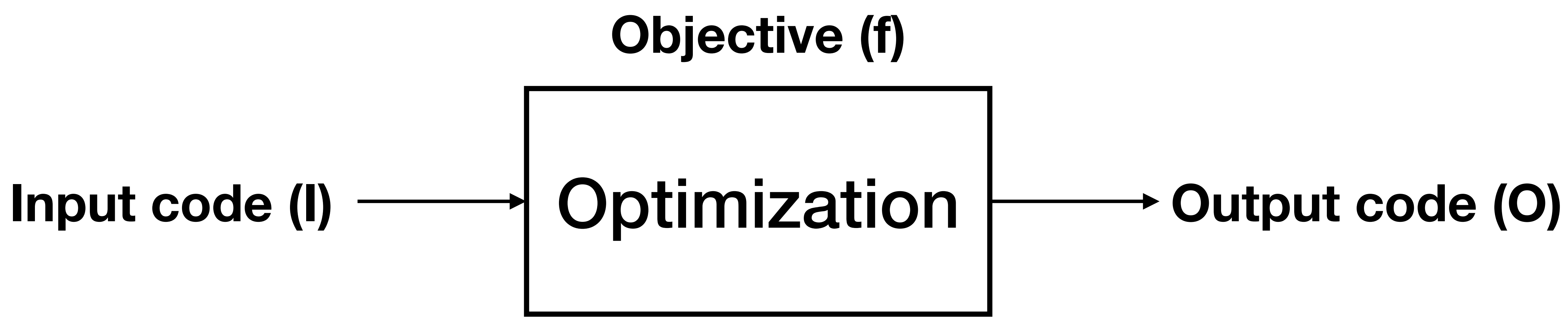
Do not know if a move will be profitable immediately

Chess

That's why it is highly competitive!!

https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1704_3

Two types of Optimizations



`Type I`

- Steps are always Profitable
  `f(O) > f(I)`
- Mostly independent

Dead Code Elimination, Constant Folding, Peephole Optimizations …….

`Type II`

- Steps may not lead to global profitability
  `f(O) > f(I) ??`
- Mostly mutually-exclusive

Loop fusion, fission, unrolling, vectorization, parallelization…….

Dead Code Elimination

```c
int foo(void)
{
  int a = 24;
  int b = 25;
  int c;
  c = a * 4;
  return c;
  b = 24;
  return 0;
}
```

https://en.wikipedia.org/wiki/Dead_code_elimination

Dead Code Elimination

```
int foo(void)
{
  int a = 24;
  int b = 25;
  int c;
  c = a * 4;
  return c;
  b = 24;
  return 0;
}
```

Always a good idea to
get rid of unwanted statements

Always a good idea to
get rid of unreachable code

https://en.wikipedia.org/wiki/Dead_code_elimination

Dead Code Elimination

```
int foo(void)
{
  int a = 24;
  int b = 25;
  int c;
  c = a * 4;
  return c;
  b = 24;
  return 0;
}
```

Always a good idea to get rid of unwanted statements

Always a good idea to get rid of unreachable code

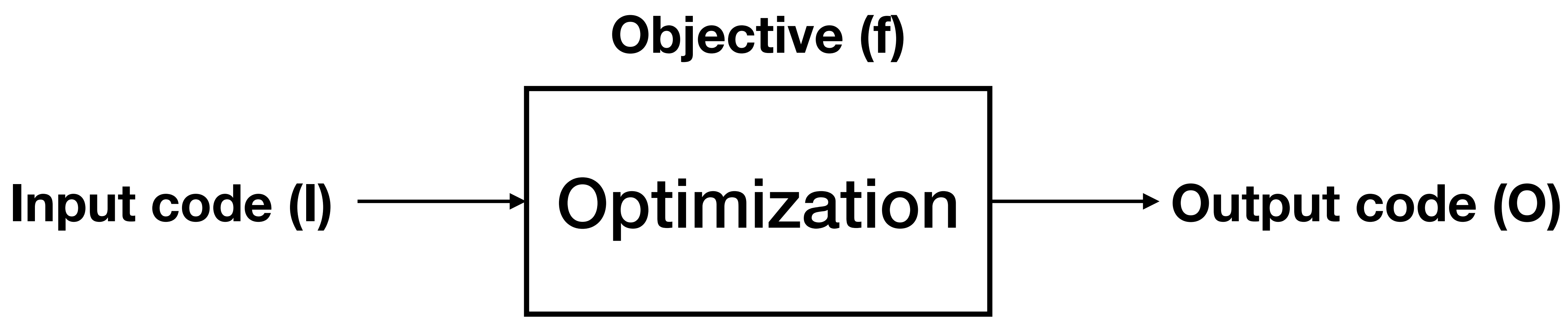
→

```
int foo(void)
{
  int a = 24;
  int c;
  c = a * 4;
  return c;
}
```

No optimization decision making needed!

https://en.wikipedia.org/wiki/Dead_code_elimination

Two types of Optimizations



`Type I`

- Steps are always Profitable
  `f(O) > f(I)`
- Mostly independent

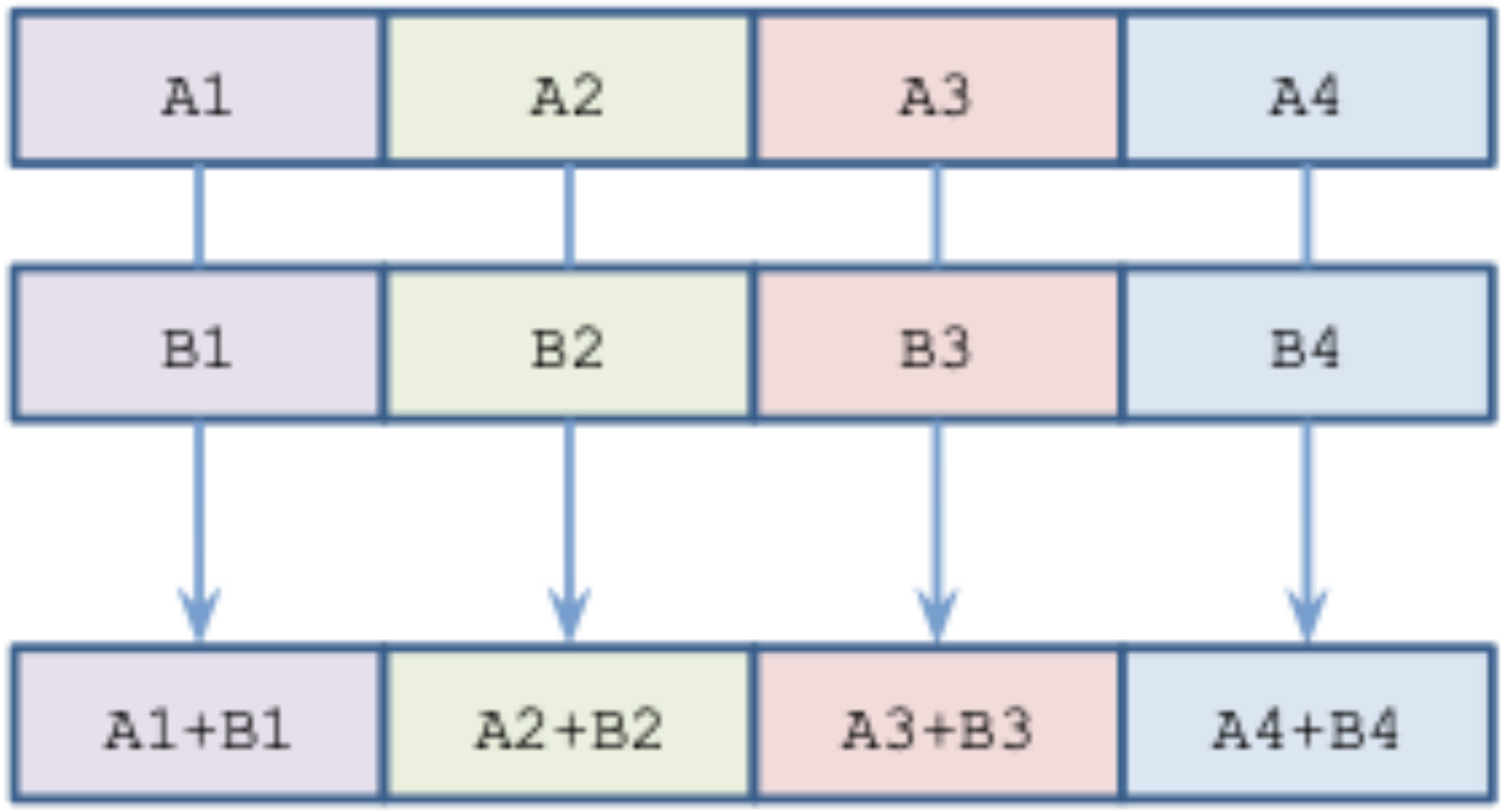
Dead Code Elimination, Constant Folding, Peephole Optimizations …….

`Type II`

- Steps may not lead to global profitability
  `f(O) > f(I) ??`
- Mostly mutually-exclusive

Loop fusion, fission, unrolling, vectorization, parallelization…….

Hardware Vector Units



Single **I**nstruction **M**ultiple **D**ata execution

Intel Vector-ISA Generations

32-bit scalar only	**64**-bit vector (MMX)	**128**-bit vector (SSE2)	**256**-bit vector (AVX2)	**512**-bit vector (AVX512)
	1997	**2000**	**2011**	**2016**

Increase in bit-width

→

Diversity in Instruction Set

Vectorization

Independent and **Similar** statements can be vectorized

Scalar Code

Vector Code
Single **I**nstruction **M**ultiple **D**ata (SIMD)

```
a[0] = b[0] + c[0]
a[1] = b[1] + c[1]
```

Vector Packs

→

```
{a[0],a[1]} = {b[0],b[1]} + {c[0],c[1]}
```

Vectorization

- Are Vectorization opportunities always independent?
- Are Vectorization opportunities always globally profitable?

```
A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]
```

Assume that the vector unit can only execute 2 instructions at a time

What are **all** vectorization possibilities?

Vectorization

- Are Vectorization opportunities always independent?
- Are Vectorization opportunities always globally profitable?

```
A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]
```

Assume that the vector unit can only execute 2 instructions at a time

What are **all** vectorization possibilities?

`{A1,A2}`

Vectorization

- Are Vectorization opportunities always independent?
- Are Vectorization opportunities always globally profitable?

```
A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]
```

Assume that the vector unit can only execute 2 instructions at a time

What are **all** vectorization possibilities?

```
{A1,A2}
{A1,A3}
```

Vectorization

- Are Vectorization opportunities always independent?
- Are Vectorization opportunities always globally profitable?

```
A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]
```

Assume that the vector unit can only execute 2 instructions at a time

What are **all** vectorization possibilities?

```
{A1,A2}
{A1,A3}
{A2,A3}
```

Vectorization

- Are Vectorization opportunities always independent? NO
- Are Vectorization opportunities always globally profitable?

A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]

Assume that the vector unit can only execute 2 instructions at a time

What are **all** vectorization possibilities?

```
{A1,A2}
{A1,A3}
{A2,A3}
```

Vectorization

- Are Vectorization opportunities always independent? NO
- Are Vectorization opportunities always globally profitable? NO

```
A1 = L[0] + L[4]
A2 = L[3] + L[5]
A3 = L[1] + L[5]
```

Assume that the vector unit can only execute 2 instructions at a time

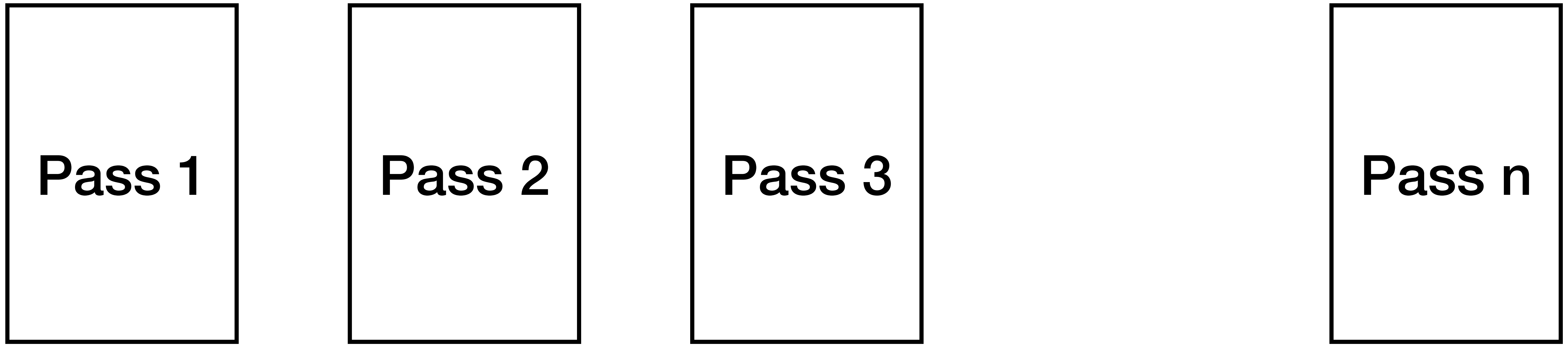
What are **all** vectorization possibilities?

```
{A1,A2}
{A1,A3}
{A2,A3}
```

How to make step decisions?

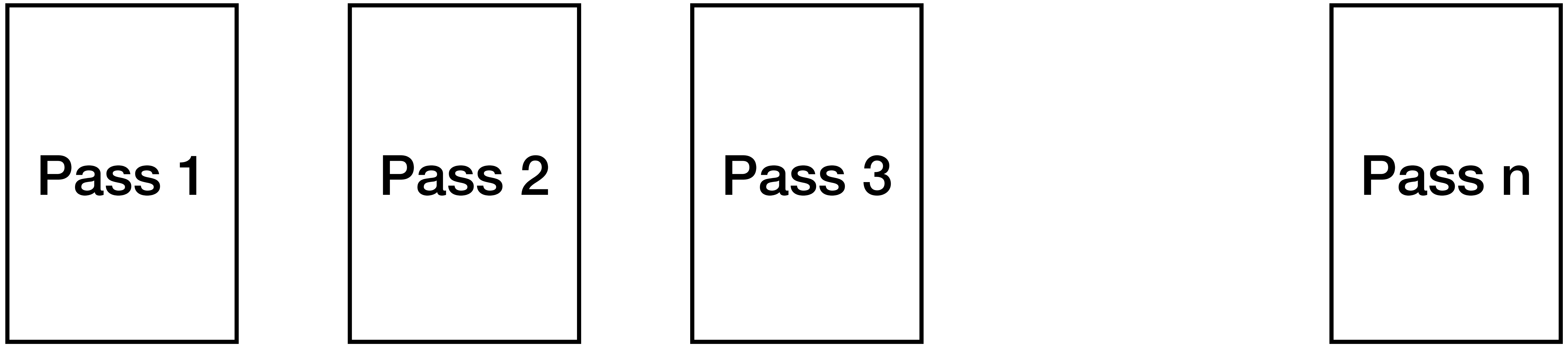
- Enumerate all possible choices and select the most profitable?
- **Intelligent Search**
  - Meta Optimization: improving compiler heuristics with machine learning (PLDI 2003)
- **Learned Optimizations**
  - Compiler Auto-vectorization using Imitation Learning (NeurIPS 2019)
  - NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning (CGO 2020)

Multiple Optimization Passes



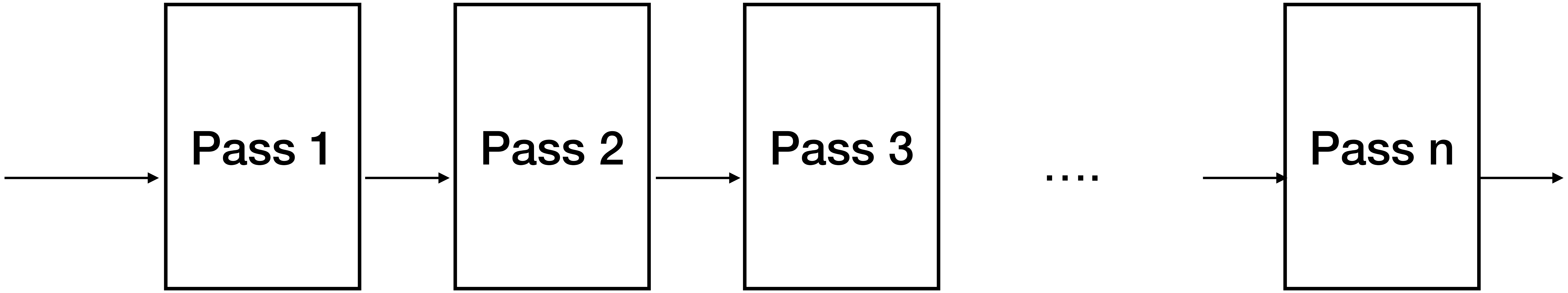
How do we compose these passes?

Multiple Optimization Passes



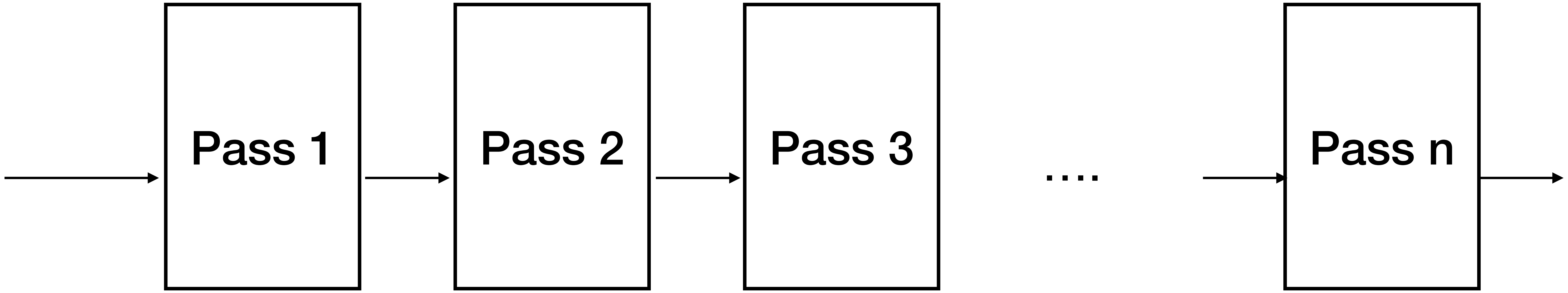
How do we compose these passes?

Multiple Optimization Passes



How do we compose these passes? Run them in sequence

Multiple Optimization Passes

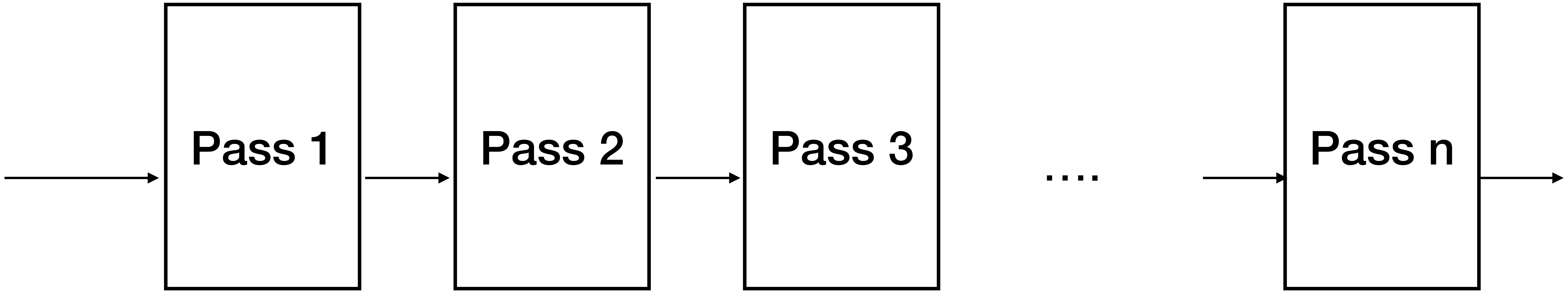


How do we compose these passes? Run them in sequence

Faces the same challenges at Type II Optimizations:
Now passes are the steps

Phase Ordering Problem

Multiple Optimization Passes



How do we compose these passes? Run them in sequence

Faces the same challenges at Type II Optimizations:
Now passes are the steps

Phase Ordering Problem (RL solution in the reading list)

Next Lecture

- Anatomy of a type II compiler optimization pass
- Exposing Tunable parameters
- DSLs and Domain Specific Optimizations
- Examples on Learned Optimization and Cost Models

How to select papers?

- Familiar with the subject area
- Read the contributions and the motivation. Sounds Interesting?
- Not all papers are of equal difficulty to read
  - Difficulty of the paper taken into account during grading
  - Dependency of the paper on related work also taken into account

Any Questions?