Introduction to Compiler Construction

Free Code Camp - Elk Grove
Friday, July 19

The plan.

1. My background
2. Compilers 101
3. Overview of Compiler Construction
4. Live Demo

1 My background

Hi, I’m Jeff

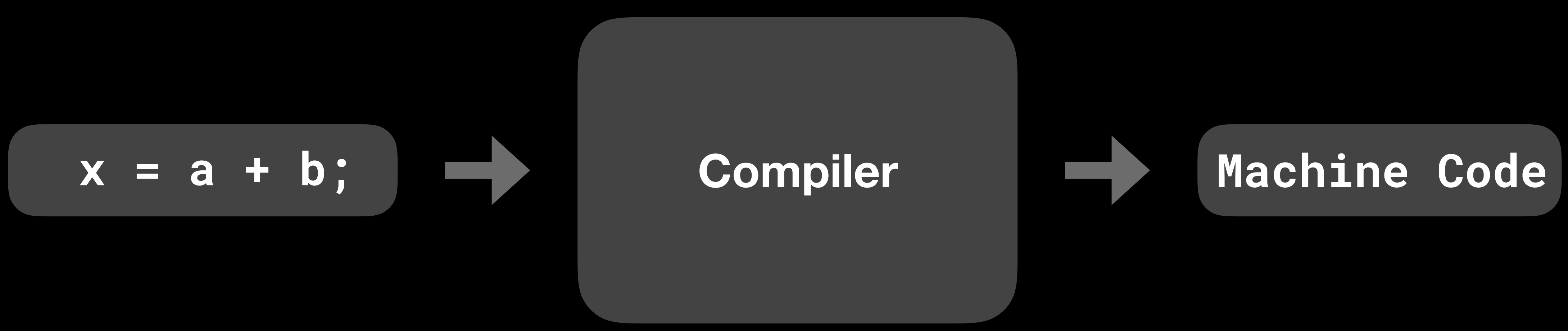
Areas of Interest:
Operating Systems, Compilers, UX/UI Design

To keep things short and simple, I enjoy photography, graphic design, programming, technology, and exploring new possibilities.

2 Compilers 101

What is a **compiler**?

- A compiler is a software program that turns your high-level programming language into machine readable code.
- There are many types of compilers.



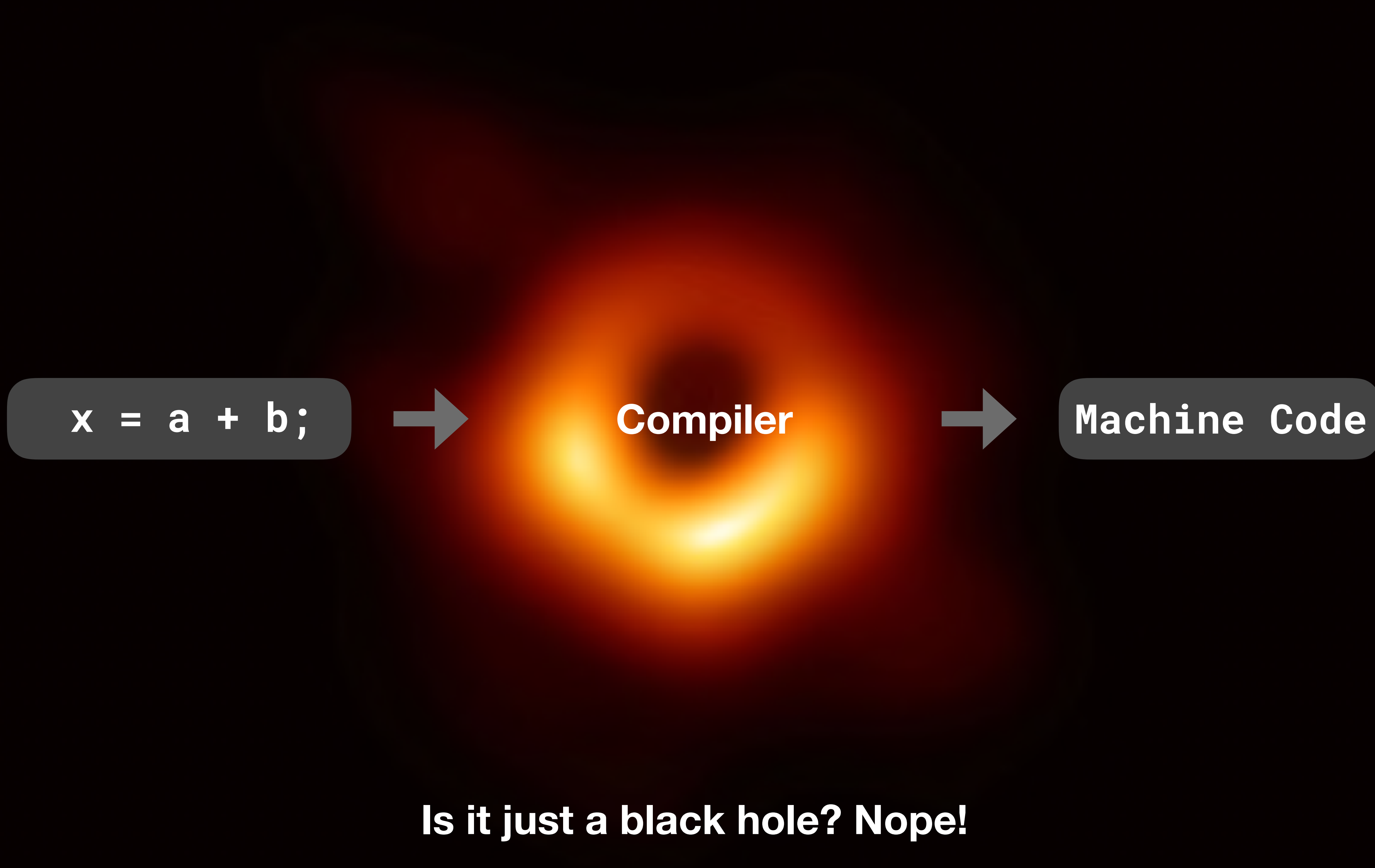
There are different types of compilers

- **Cross-compilers:** A compiler that is capable of creating executable code for a platform other than the one which the compiler is running.

- **Translation Compiler:** A compiler that translates a high-level programming language to another high-level language.

- **Decompiler:** Translates low-level languages into a higher level language.

How is a compiler constructed?



Is it just a black hole? Nope!

A compiler's construction

- A compiler is usually built with three major components:

Inside each component

Front End

Scanner
(Lexical Analysis)

Parser
(Syntactic Analysis)

Semantic Analysis

IR

Optimizer
(aka the Middle End)

Optimization 1
Optimization 2
.
.
.
Optimization *N*

Optimized
IR

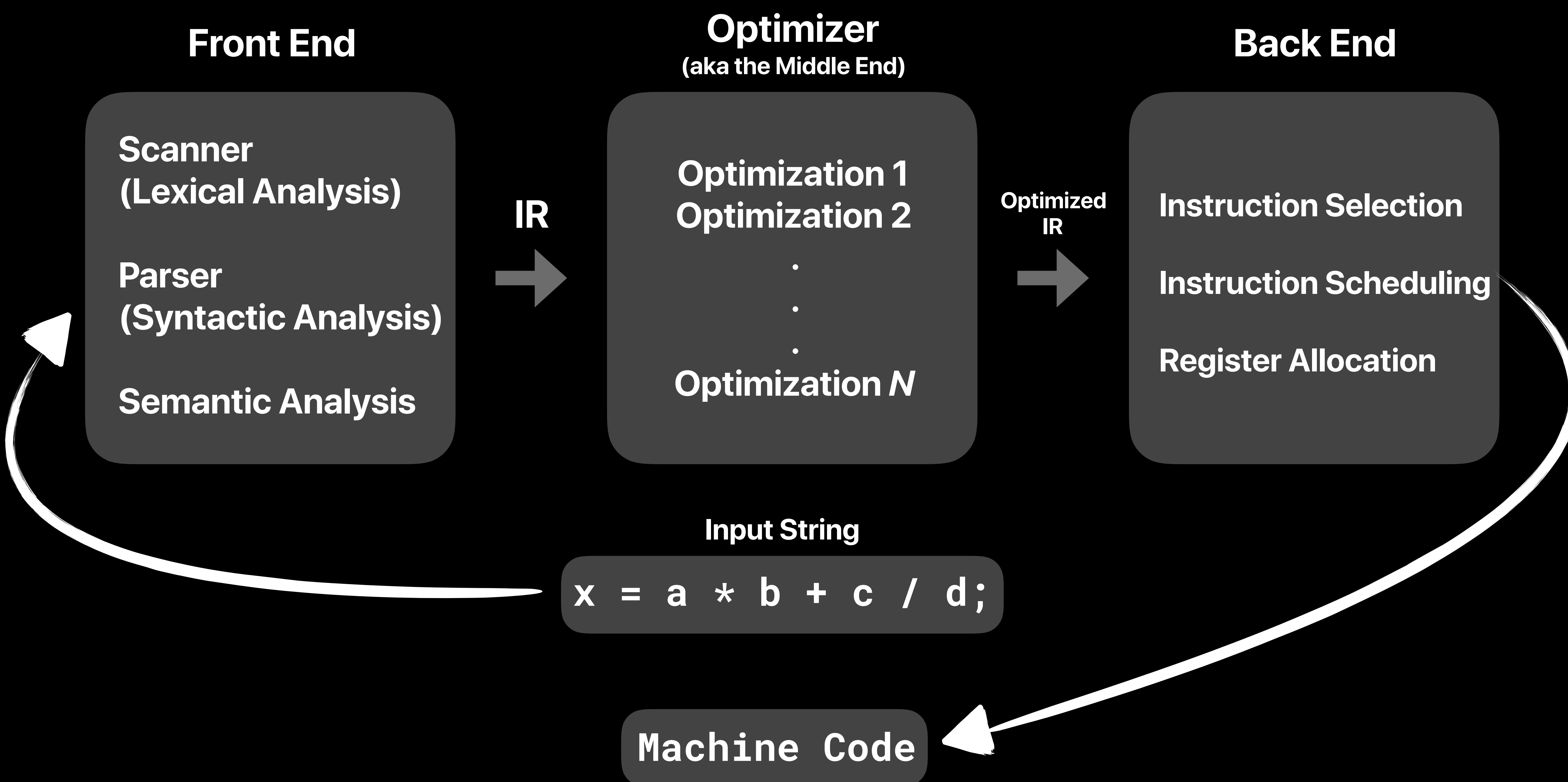
Back End

Instruction Selection

Instruction Scheduling

Register Allocation

A compiler's lifecycle



The *actual* plan.

1. My background
2. Compilers 101
3. The Front End
4. The Optimizer*
5. The Back End*
6. Live Demo

Overview of Compiler Construction

*a brief overview

3 The Front End

3.1 The Scanner

What is a scanner?

- A scanner reads an input string (your code) and breaks it up into “tokens” or lexemes.
- It is also called a lexer.

Human Language		Programming Language
Word	→	Token

Let's trace the scanner.

```
x12 = a3 + b45;
```

Let's trace the **scanner**.

```
x12 = a3 + b45;
```

Tokens

Let's trace the scanner.

x12 = a3 + b45;

Tokens

x12

Let's trace the **scanner**.

x12 = **a3** + b45;

Tokens

x12 =

Let’s trace the **scanner**.

x12 = a3 **+** b45 ;

Tokens

x12 = a3

Let's trace the scanner.

x12 = a3 + $\boxed{\text{b45}}$;

Tokens

x12	=	a3	+

Let's trace the scanner.

x12 = a3 + b45 ;

Tokens

x12	=	a3	+	b45

Let's trace the scanner.

x12 = a3 + b45 ; eof

Tokens

x12	=	a3	+	b45	;

How did the scanner know when to break the code up into tokens?

- The scanner is following a set of rules that are defined by the programmer. These rules are called regular expressions.
- A regular expression (or regex) is a string that defines a certain pattern.
- In the context of compilers, you're creating a pattern for the scanner to follow.
- Regular expressions are constantly being used outside of compilers: HTML, C#, etc.

Some regular expression examples

- Let’s say my programming language have two reserved key words: `class` and `concact`

- Since `class` and `concact` both have the letter ‘c’ in common, we can write the following RE’s:

```
class|concat  OR  c(lass|oncat)
```

RE for `unsigned int`

- An `unsigned int` can be described as *either zero or a nonzero digit followed by zero or more digits*, so the RE would be:

```
0|[1...9][0...9]*
```

RE for multiline comments

- Multi-line comments in C, C++, C#, and Java begins with the delimiter `/*` and ends with `*/`
- Example of a multiline comment:

```
/*
  Hey there, this is a
  multiline comment
 */
```

```
/*
 * Hey there, this is a
 * multiline comment
 */
```

RE: `/* ( ^* | *+ ^/ )* */`

Implementing a scanner
(Part 1)

1. We first need to define our regular expressions for basic constructs like comments, identifiers, etc.

2. Since a scanner must recognize or accept tokens based on our regular expressions, we need to design an algorithm of <u>how</u> the scanner is going to accept/reject these tokens. We can visually design this by using a mathematical model called a finite-state acceptor (aka state machines or <u>finite automaton</u>)

3. Cry because you realized that you have to implement this. :(

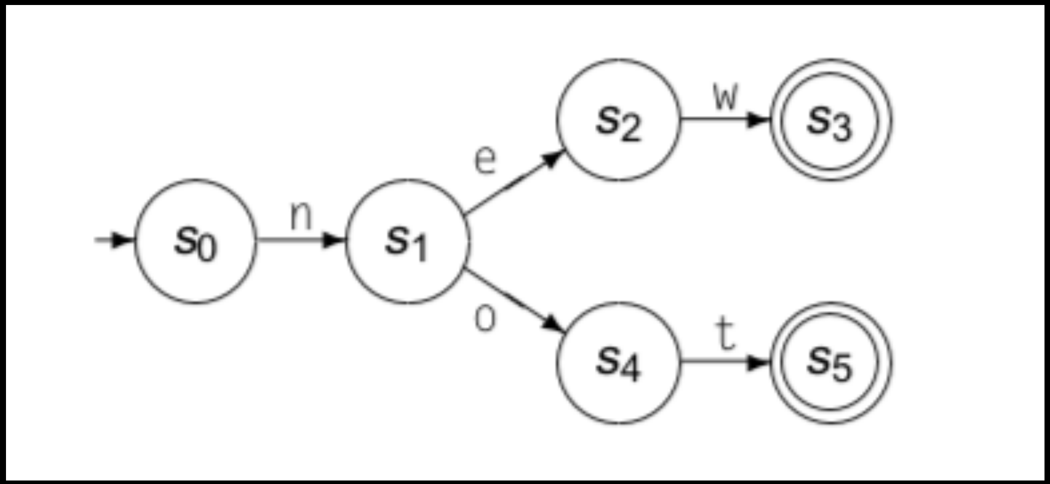
Finite Automaton Example

Let's say my programming language have two reserved key words: `new` and `not`

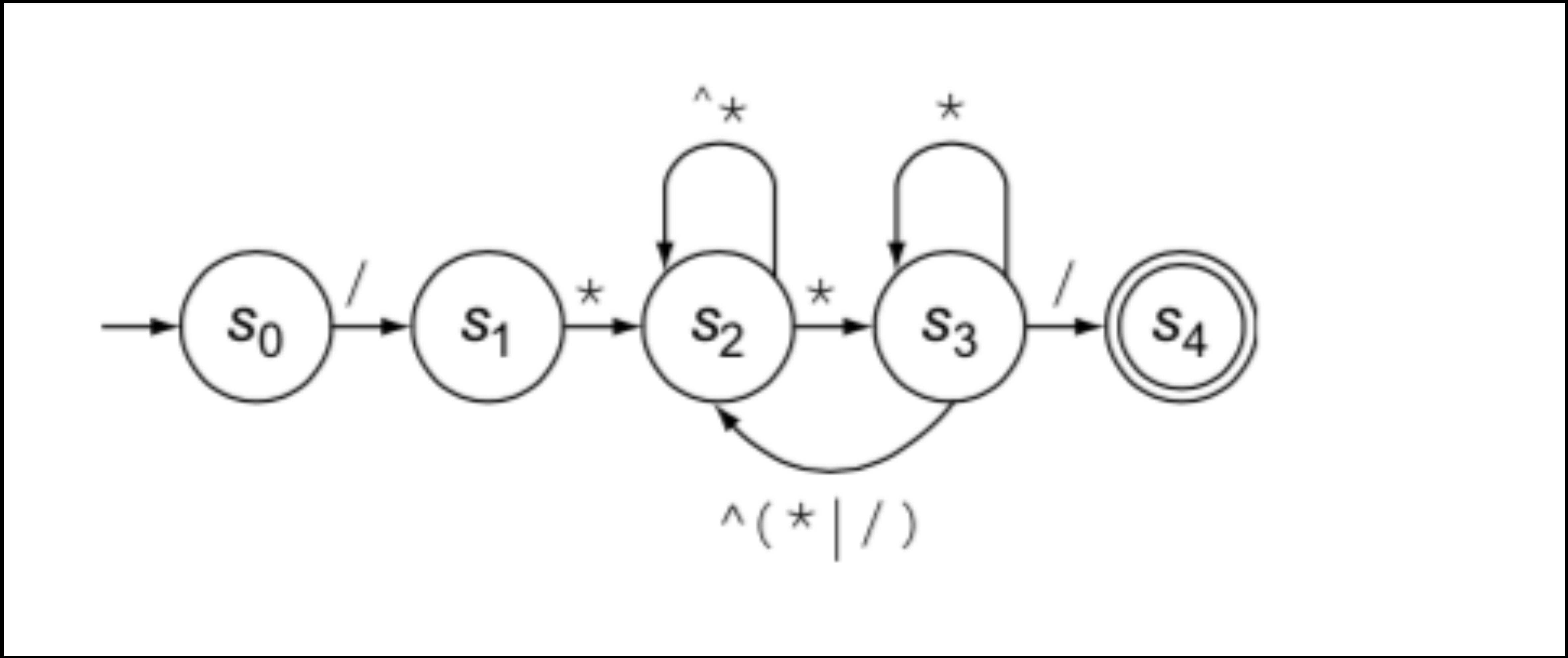
Assume my regular expression is:

`n(ew|ot)`

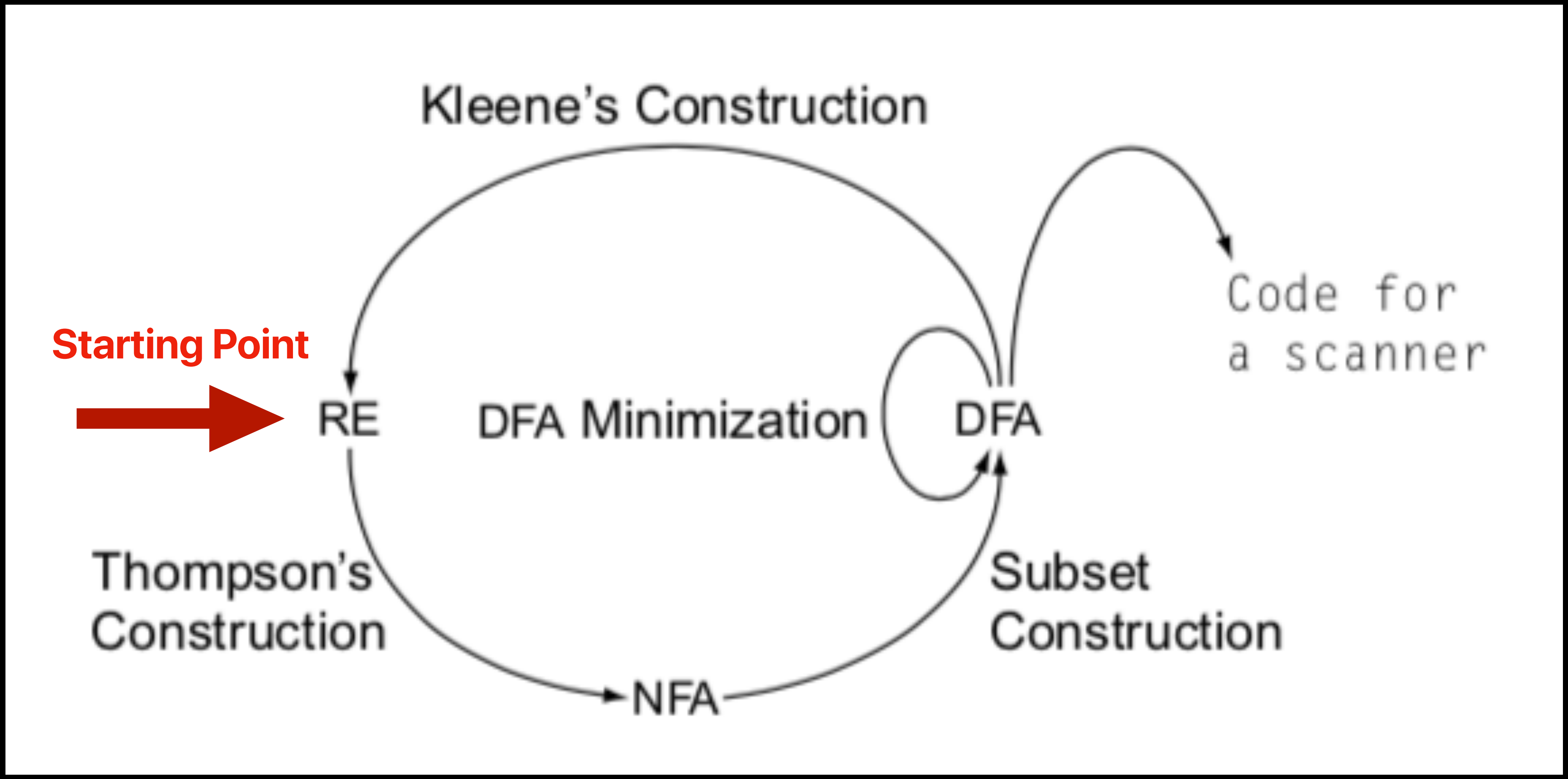
Then my FA is:



FA for $/* (\hat{}* \mid *+ \hat{}/)* */$



Overview of RE to FA construction



Implementing a scanner
(Part 2)

- So far, we looked at how formal theory can help us design the scanner. Now we must convert the DFA into actual code.

- There are three ways to implement scanners: table-driven, direct-coded, and hand-coded scanners.

- We'll only be looking at a table-driven scanner.

Pseudocode for Table Driven Scanners

```
NextWord()
  state ← s0;
  lexeme ← "";
  clear stack;
  push(bad);

  while (state≠se) do
    NextChar(char);
    lexeme ← lexeme + char;

    if state ∈ SA
        then clear stack;

    push(state);

    cat ← CharCat[char];
    state ← δ[state,cat];
  end;

  while(state ∉ SA and
        state≠bad) do
    state ← pop();
    truncate lexeme;
    RollBack();
  end;

  if state ∈ SA
    then return Type[state];
    else return invalid;
```

r	0, 1, 2, ..., 9	EOF	Other
Register	*Digit*	*Other*	*Other*

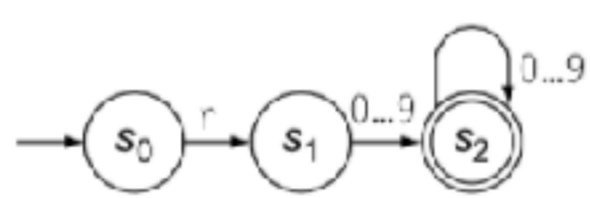
The Classifier Table, CharCat

	Register	*Digit*	*Other*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

The Token Type Table, Type



The Underlying DFA

About table driven scanners

- Has two key components: the skeleton scanner for controlling the scanning process and a set of tables for lookup.
- The most difficult part of implementing this type of scanner are the tables.
- The easiest part is the skeleton scanner - it's basically a bunch of simple loops.
- Pros: Portability because the if the RE's were to change, the tables would change but not the scanning algorithm.
- Biggest issue: Poor performance due to table lookups, numerous memory references, and stays inside the middle loop for a while

“Implementing” a scanner
(Part 3)

But Jeff, I don’t want to implement a scanner from scratch…

There’s a solution! :)

Introducing Flex

- FLEX(fast lexical analyzer generator) is a scanner generator.
- It takes care of tokenization, NFA → DFA conversion, DFA Minimizations, and more.
- It generates a scanner based on the regular expressions you give to Flex.
- I'll show what this looks like in the demo.

3.2 The Parser

What is a parser?

- The main purpose of the parser is to analyze the input (which is from the scanner) and check if it's a valid sentence in the programming language.

- In the real world, what makes a sentence "valid" is the grammar being used in the sentence.

- This same idea applies to a programming language. The parser uses a grammar, that the programmer wrote, to check if the input string is syntactically correct.

Grammar for Classic Expressions

```
Goal   → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term * Factor
       | Term / Factor
Factor → ( Expr )
       |   num
       |   name
```

How does a parser parse an input string?

1. It builds a data structure. Most of the time, parsers use parse trees or ASTs.

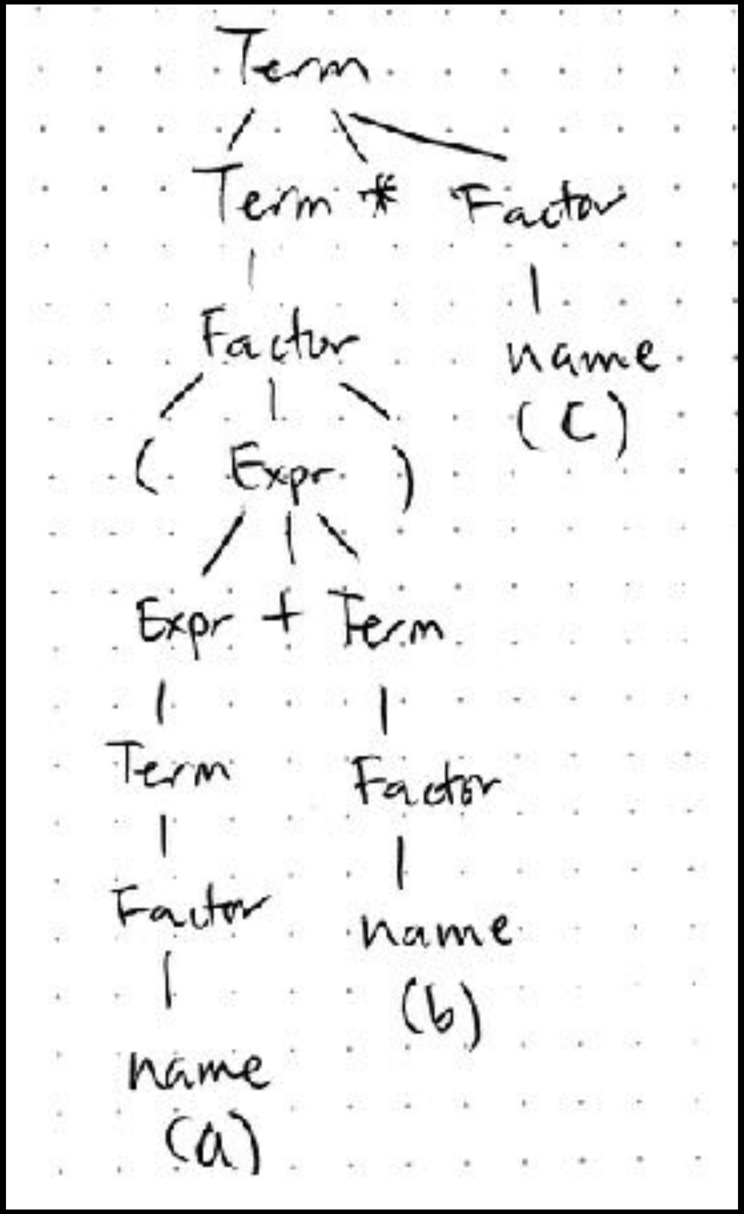
2. Design the data structure where it's based on the grammar you wrote.

   If you make one mistake to your grammar, your parser will fail.

Example of a parse tree

Input String: `(a+b) * c`

```
Goal   → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term * Factor
       | Term / Factor
       | Factor
Factor → ( Expr )
       |   num
       |   name
```



Implementing a parser
(Part 1)

- There are two parsing techniques: top-down parsing and bottom-up parsing.

- Top-down parsing: It constructs a parse tree starting from the root node and gradually moves down to the leaf nodes (bottom).

- Bottom-up parsing: It constructs a parse tree starting from the leaf nodes of the tree and moves upwards to the root node.

Example of a bottom-up parsing technique: `LR(k)` Algorithm

`LR(k)`

Scan left to right | Right most derivation | Number of look aheads

- A LR parser is a non-recursive, shift-reducing, table driven bottom-up parser.
- Since this is table-driven, it contains a skeleton parser code, then uses two tables: `Action` and `Goto` tables.

Other types of LR parsing

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

“Implementing” a **parser**

(Part 2)

Just like scanning, you don’t actually have to implement a parser from scratch!

Introducing Bison

- Bison is a parser generator that accepts a context-free grammar (which you write) and generates a parser based on your grammar.
- Bison uses the LALR(1) technique with options to use it in LR(1), IELR(1) modes.
- Demo will be given later.

3.3

Intermediate Representation

What is an intermediate representation?

- As your code goes through a series of passes through the compiler, it needs to convey the information gathered from one pass to another.
- To pass this information around, we need to have an Intermediate Representation.
- Think of an IR as "pseudo-code", where it doesn't use an actual language to represent its structure.
- There are three ways to convey this information: graphical, linear, and hybrid. In this talk, we'll focus on graphical representations as it's the most common IR.

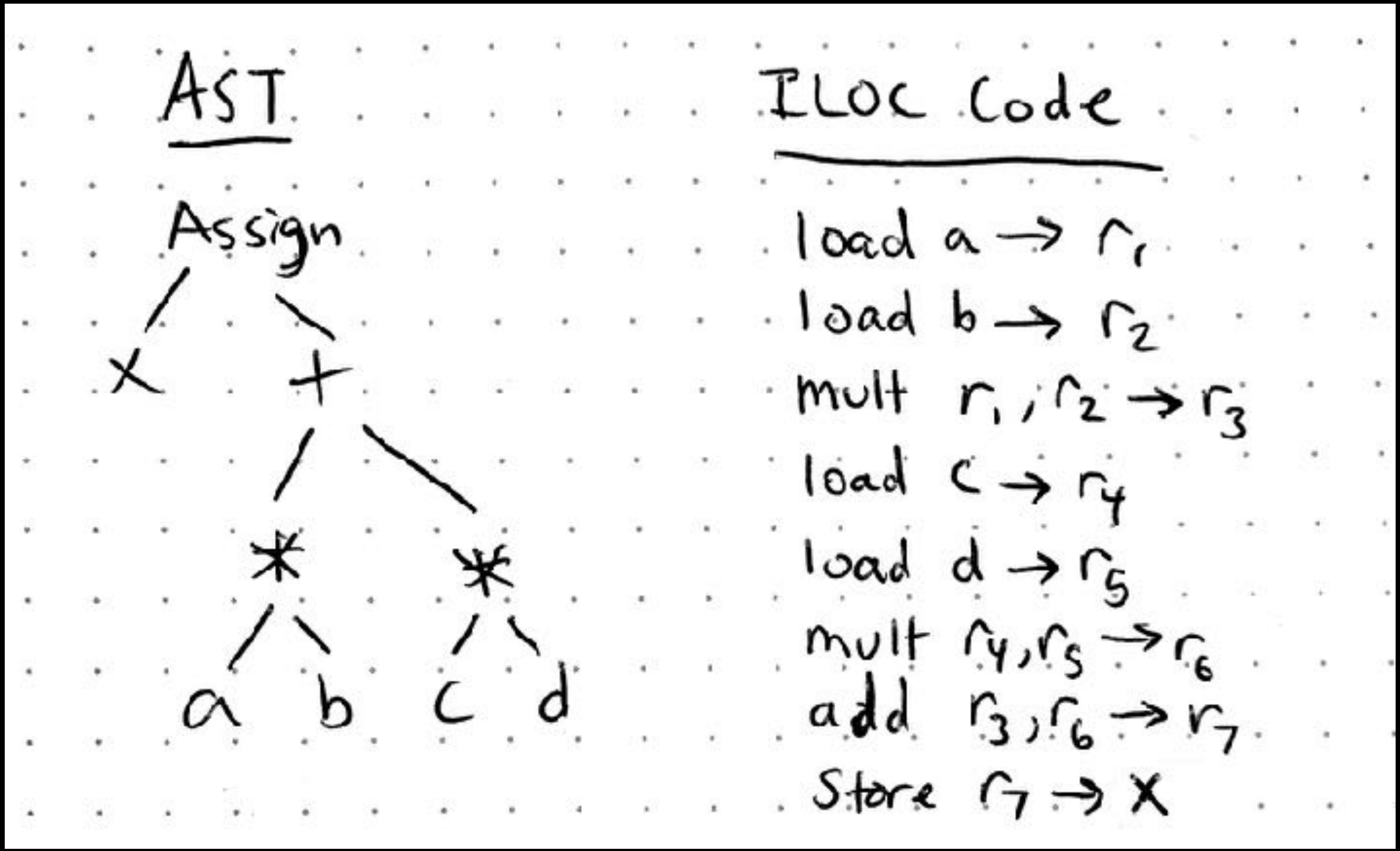
Graphical IR's

- The way how a graphical representation works is that your code will build an AST (high-level IR) based on your input string and then you map the AST into a low-level IR, like ILOC.

- ILOC is an abstract version of the assembly language.

Example: AST → ILOC

Input String: `x = a * b + c * d;`



4 The Optimizer

Why do we to do code optimizations?

- Programmers are lazy. Admit it ;)
- Code optimizations are required to make sure that the intermediate code (or any intermediate representation) uses the least amount of CPU cycles, memory usage, and more.
- These optimizations will transform your intermediate representation into an optimized version of your code.
- After these optimizations, it will output an optimized IR.

Optimization Technique 1:
Common Sub-expression Elimination

Searches for identical expressions and determines (cost-to-benefit analysis) if it's worth replacing it with a single variable. This is a common optimization technique.

```
int j;
j = 7 * arr[3] + (arr[3] + 10)
```

↓

```
int j;
int temp = arr[3];
j = 7 * temp + (temp + 10)
```

Optimization Technique 2: Constant Folding

Expressions with constants can be evaluated at compile time, therefore it improves runtime performance and reduces code size by not evaluating the expression at compile-time.

```
int babyMath (void) {
  return 2 + 2;
}
```

↓

```
int babyMath (void) {
  return 4;
}
```

Optimization Technique 3:
Loop Unrolling

This reduces loop overhead. It reduces the number of iterations and replicating the body of the loop.

```
for (i = 0; i < 100; i++) {
  g();
}
```

↓

```
for (i = 0; i < 100; i += 2) {
  g();
  g(); // Number of iterations
       // were reduced
       // from 100 to 50
}
```

5 The Backend

Overview of the backend

- The backend accepts an IR as the input and generates machine code for the target machine.
- This is why we generate IR's. An IR provides a generic overview of the program and then the backend "simply" translates it into machine code designed for a specific machine.
- This highlights the modular design of a compiler. If you are building it for a different architecture, all you need to change is the backend.

What happens in the backend of a compiler?

It does three major things:

1. Instruction selection
2. Instruction scheduling
3. Register allocation

Overview of **Instruction Selection**

It reads the IR and it determines which instructions to use based on the target machine's architecture.

Example: ILOC → x86 arch

```
load a → r1
```

⬇

```
MOV EAX, a
```

Overview of Instruction Scheduling

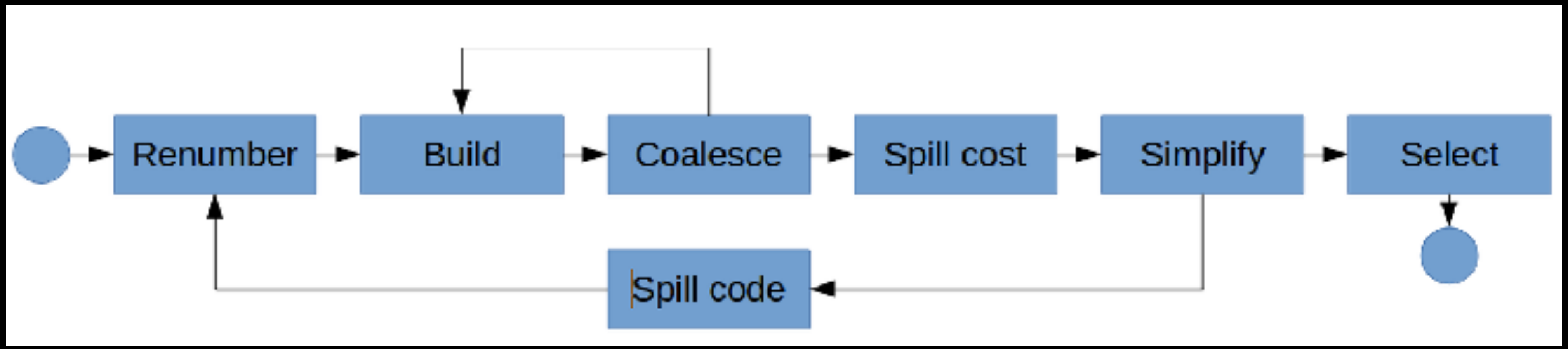
This reads the code generated from the instruction selection phase and does a series of optimizations. It does it in a way where it doesn’t change the meaning of the code.

- Avoids stalling from pipelining by rearranging instructions.
- Avoids data hazards by breaking code into basic blocks and determines whether rearranging the blocks will change the behavior of the code

Overview of Register Allocation

Since not all machines have the same amount of CPU registers, the register allocator must allocate a number of registers that do not exceed the target machine's CPU register count.

The picture below shows some key steps in register allocation:



How do we implement the **Optimizer and Backend**?

LOTS of research

or

use LLVM or gcc

Introducing LLVM

- LLVM (low level virtual machine) is a collection of modular and reusable technologies that can be used to build the optimizer and the backend phases of a compiler.
- You can use it for the front end too but it’s not common.
- Written in C++ and designed to improve comple time, link time, and run time.

6 Live Demo (Finally!)

Q&A

Any Questions?

Free toy compiler

https://github.com/jefflow/baby-c-compiler