A (Very) Brief Survey of Compiler Design

Independent Study

Sydney Newmark

Objective

```
fn main() {

    println!("Hello, world");

}
```

?

```
0101010101010101
```

Materials

Source: Modern Compiler Implementation in Java, by Andrew W. Appel and Jens Palsberg

Frontend

- Lexical Analysis
- Parsing
- Abstract Syntax Trees
- Semantic Analysis
- IR

Backend

- Activation Records
- Blocks
- Traces
- Liveness Analysis
- Register Allocation

Focus

Frontend

- Lexical Analysis
- Parsing
- Abstract Syntax Trees
- Semantic Analysis
- IR

Backend

- Activation Records
- Blocks
- Traces
- Liveness Analysis
- ~~Register Allocation~~

Lexical Analysis

Lexical Tokens

```
fn main() {
    println!("Hello, world");
}
```

123

// comments

fn

main

(

)

{

println

!

(

"Hello, world"

)

;

}

Regular Expressions

Represent sets of strings

Symbol, Alternation, Concatenation, Epsilon, Repetition (Kleene closure)

(0 | 1)* · 0

Kleene closure: For some regular expression, M, its Kleene closure is M*, the concatenation of zero or more strings in M

Even Binary Numbers

```
00000010 (4)
10000000 (128)
```

$$(0 \mid 1)^* \cdot 0$$

Sequence of zero or more "0"s and "1"s in any combination

Concatenate this sequence with a single 0

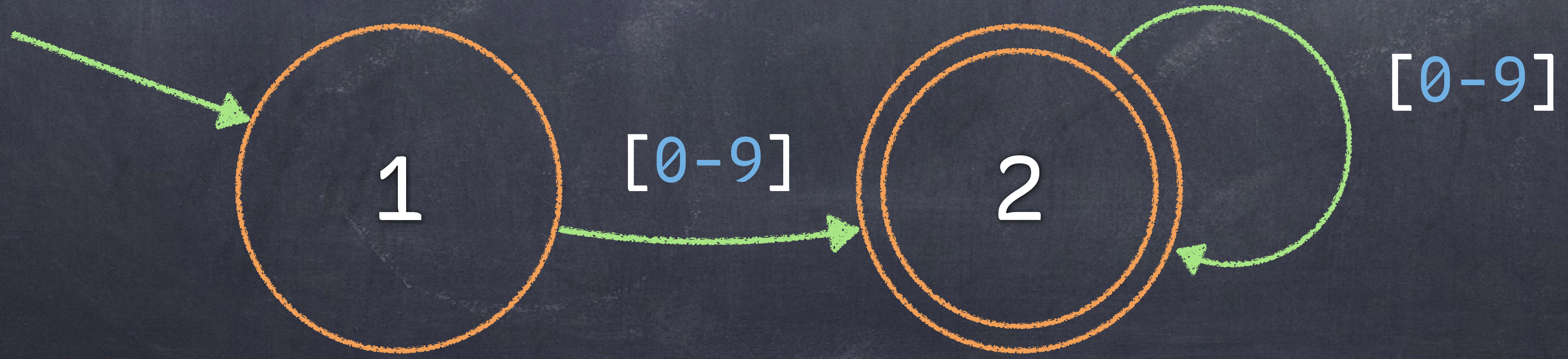
Finite Automata

A way to formalize regular expressions so they can be implemented by a computer

Consist of states, edges, and symbols

[0-9]+

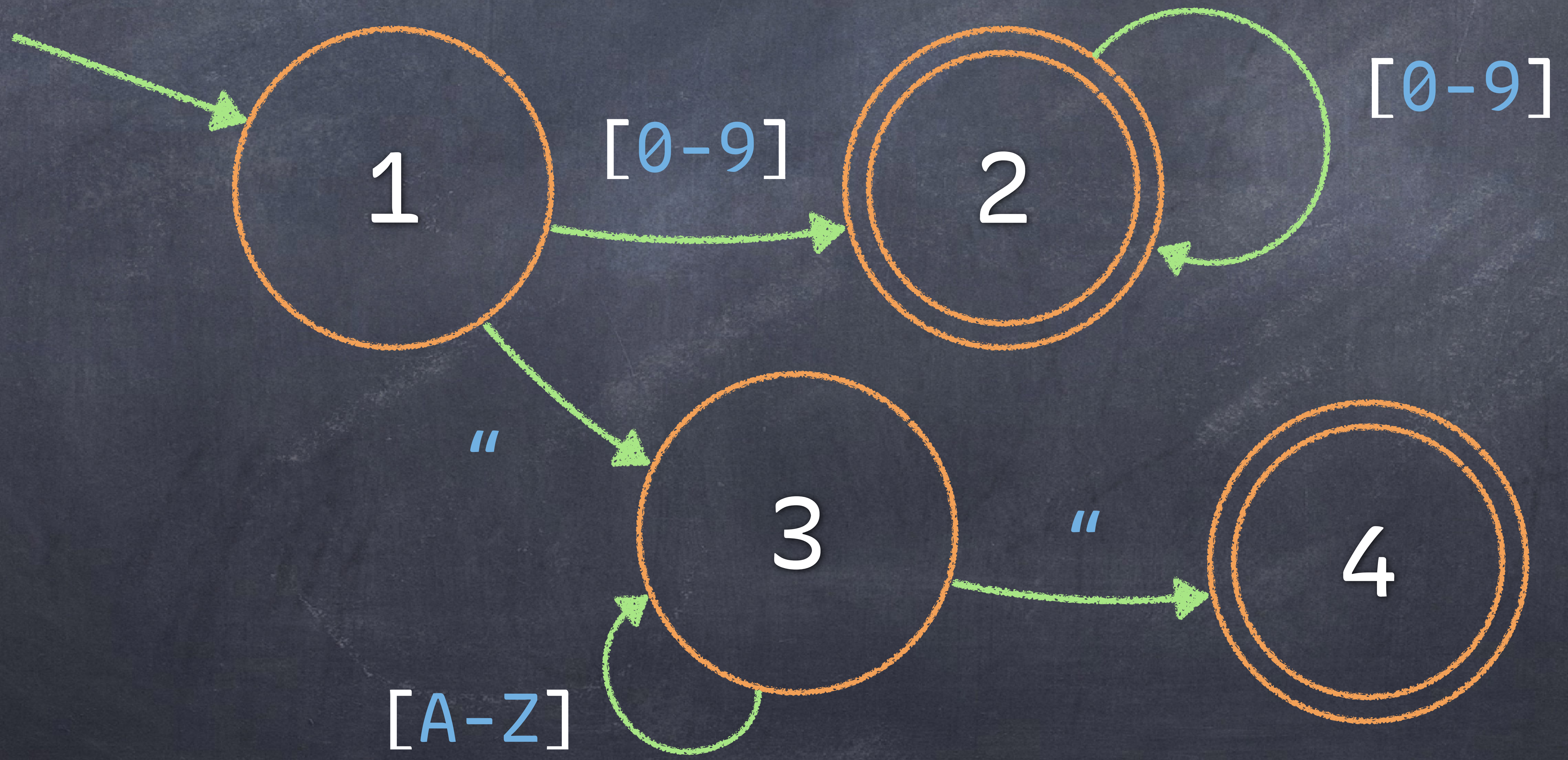
(+ denotes repetition of **one or more** times)



Deterministic Finite Automata

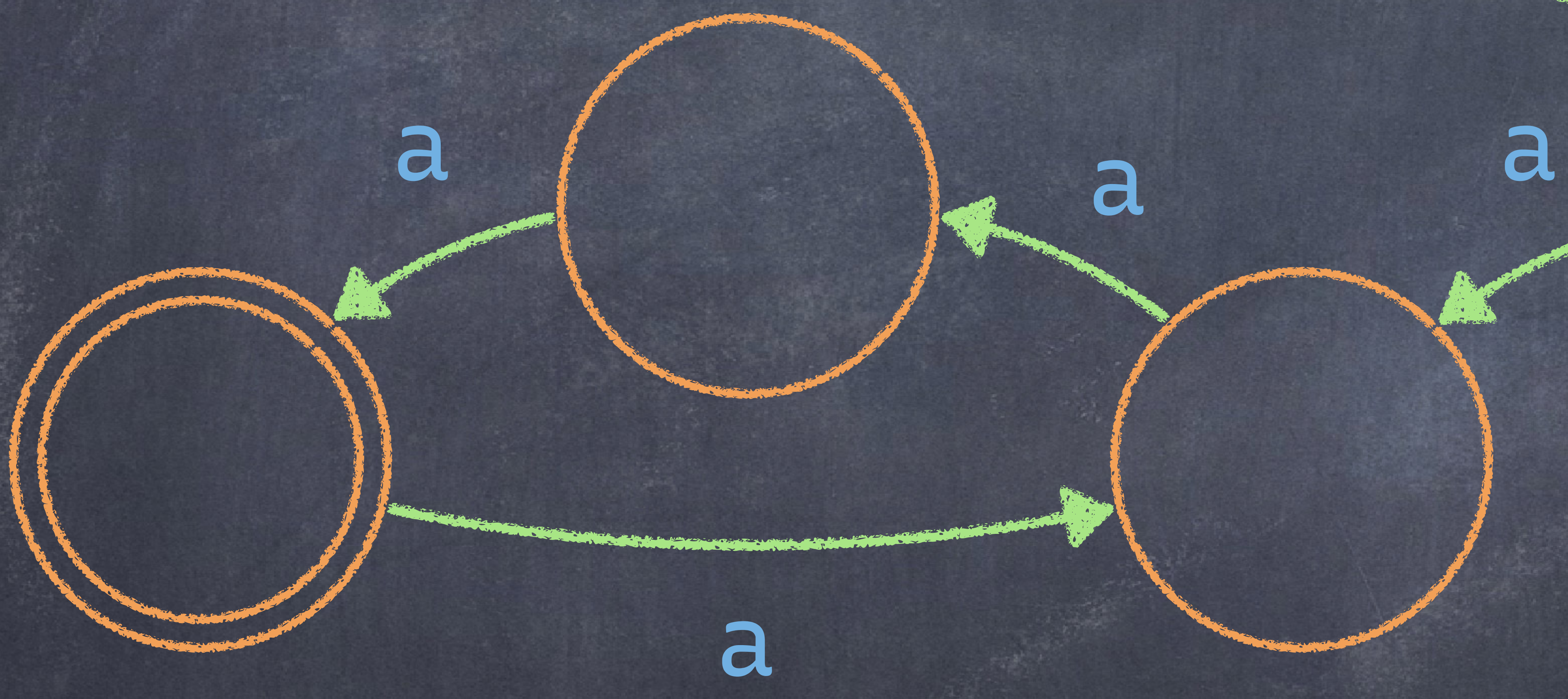
We can combine automata to create a lexical analyzer

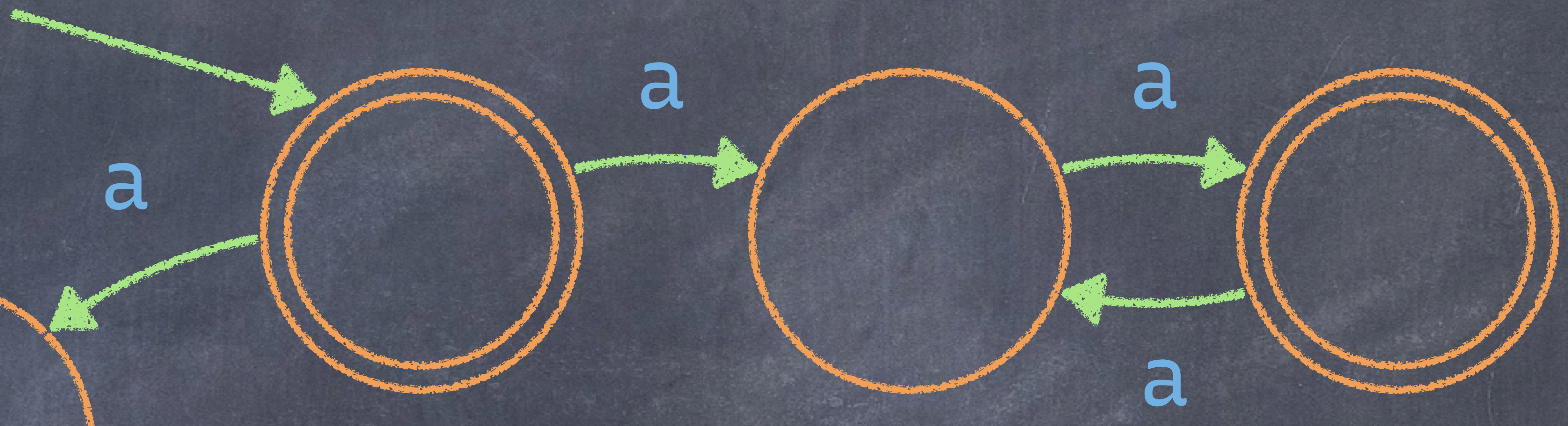
The goal is to find the longest match; i.e. the longest substring of the input that is a valid token



Nondeterministic Finite Automata

An automata which has a choice of edges labeled with the same symbol to flow from the current state





strings of a whose length is a multiple of 3

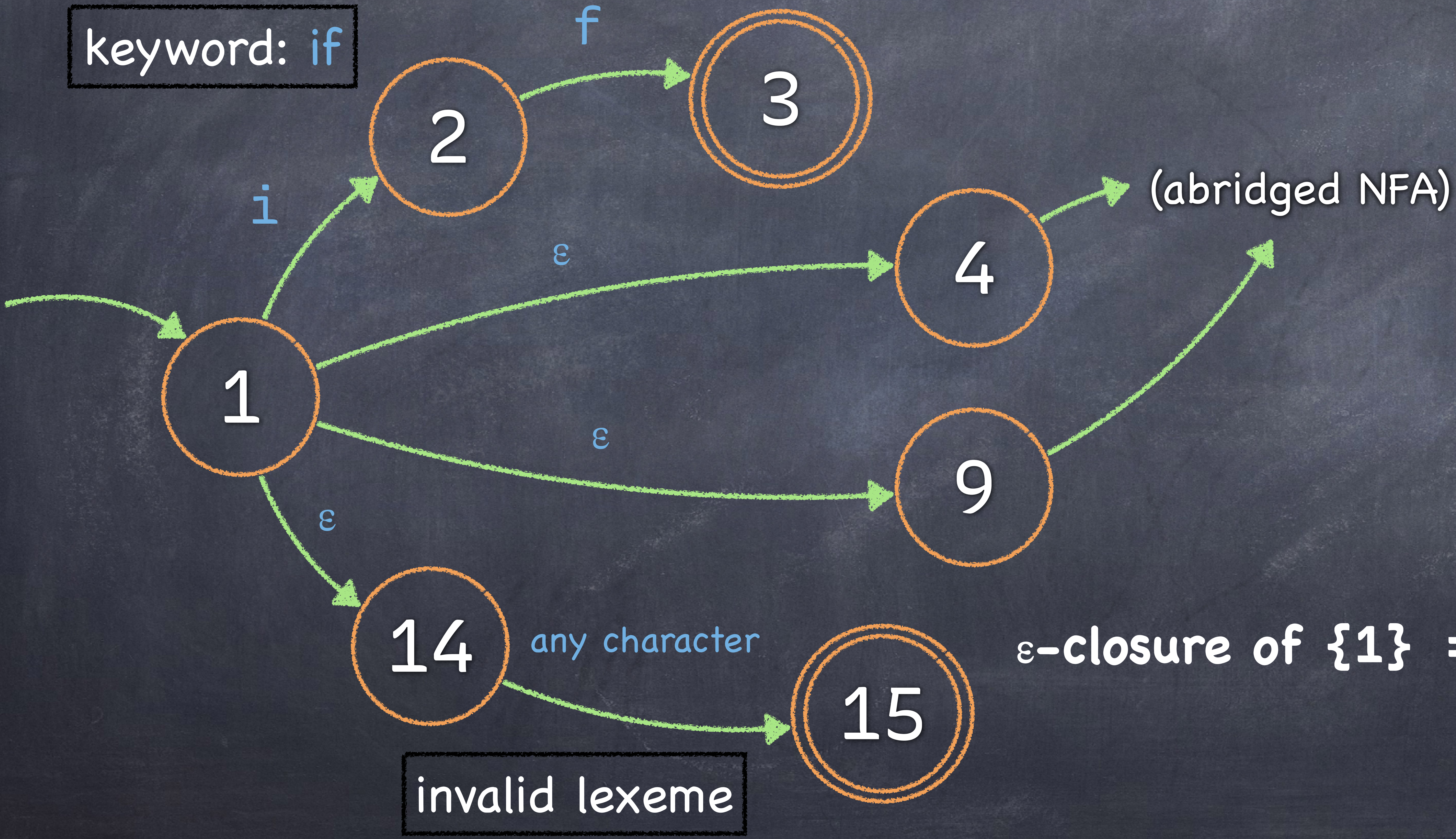
strings of a whose length is even

So what?

We can convert regular expressions to nondeterministic finite automata, and then to deterministic finite automata

- Simultaneously traverse all possibilities at each “step” of the NFA
- Each set of destination states for these possibilities is called the ε-closure
- Every ε-closure is one node in the corresponding DFA

Partial Example



ε-closure of {1} = {1, 4, 9, 14}

ε-closure of {1}

{1, 4, 9, 14}

states reachable without consuming a token of the input

ε-closure always contains itself

Parsing

Context-Free Grammars

Describes a language using sets of **productions**

Each production follows the form:

$$\text{symbol} \rightarrow s_1\ s_2\ s_3\ \dots\ s_n$$

Terminal

“Hello, world”

123

println

Nonterminal

expression

function

An Example Grammar

5 * 6 / (3 + 3)

```
expression → term;

term → factor (("+" | "-") factor)*;

factor → primary (("*" | "/") primary)*;

primary → [0-9]+ | "(" expression ")";
```

```
expression → term;

term → factor (("+" | "-") factor)*;

factor → primary (("*" | "/") primary)*;

primary → [0-9]+ | "(" expression ")";
```

Predictive Parsing

Recursive Descent

5 * 6 / (3 + 3)

primary * 6 / (3 + 3)

primary * primary / (3 + 3)

primary * primary / (primary + primary)

primary * primary / (term)

primary * primary / primary

factor

expression

Predictive Parsing

Limitations

Cannot parse grammars that use **left recursion** or when two productions for the **same nonterminal** start with the same symbols

```
E → E "+" T;

E → T;


S → "if" E "then" S "else" S;

S → "if" E "then" S;
```

Eliminating Left Recursion

```
E → E "+" num;

E → num;
```

Rewrite using right recursion

```
E → num E';

E' → "+" num E';

E' → ;
```

Left Factoring

```
S → "if" E "then" S "else" S;

S → "if" E "then" S;
```

Create another nonterminal, X

```
S → "if" E "then" S X;

X → "else" S;

X → ;
```

LR Parsing

Left-to-right parse with a **rightmost**-derivation

Shift and **reduce** are the primary actions

Has a stack and an input

Shift: Advance the input one token and push that token onto the stack

Reduce: Pop the stack as many times as the number of symbols on the right-hand side of a production rule

LR Parsing

A deterministic finite automata is applied to the stack to determine when to shift and when to reduce

The edges of the DFA are labeled by the symbols that can appear on the stack

The transitions between edges can be modeled by a simple table

(1) S → S ";"

(2) S → id ":=" E;

..

(5) E → num

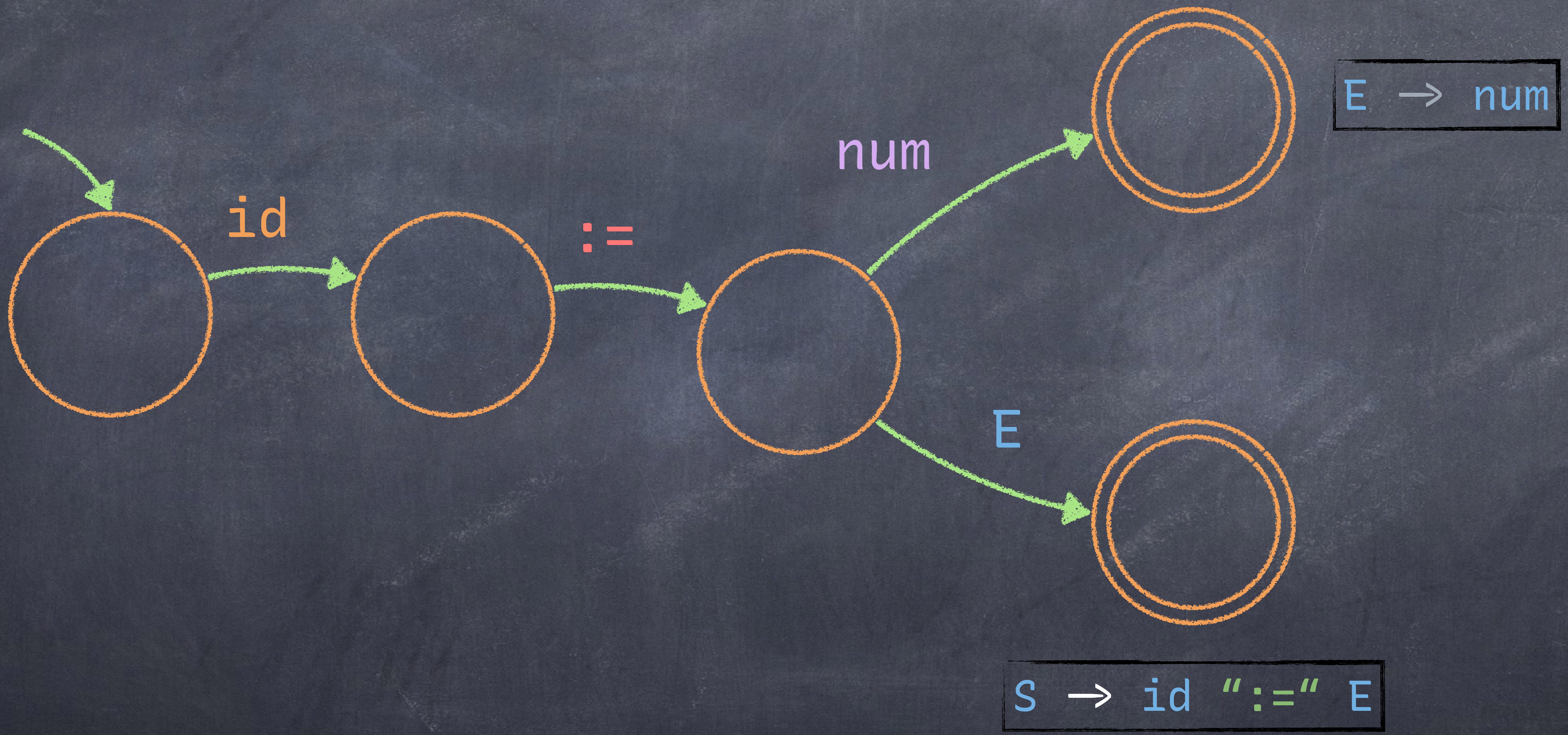
Parse Table

	id	num	;	:=		S	E
1	s4					g2	
4				s6			
6	s20	s10					g11
10			r5				
11			r2				

Based on DFA: numbers represent DFA state numbers

Stack	Input	Action
	a := 7;	shift
id	:= 7;	shift
id :=	7;	shift
id := num	;	reduce E → num
id := E	;	reduce S → id ":=" E

DFA: Stack



Interlude

Now what?

During parsing, we convert the lexemes to an **Abstract Syntax Tree (AST)**

Each nonterminal of a grammar corresponds to a node type

Using this tree we can more easily perform:

- Type checking
- Translation to an intermediate representation

A (re)Introduction to Visitors

Incredibly useful for many semantic analysis phases, including type checking

Each AST node type implements two methods:

- accept()
- visit()

i.e. type checking:

```
left.accept(this).type == right.accept(this).type
```

Or just use a language with good enums like Rust

Intermediate Representation (IR)

A machine-independent abstraction for assembly

Simplifies the process for converting the recursive AST into a more machine-friendly format

Examples of IR nodes:

```
MOVE(src, dest)   JUMP(label)   CJUMP(trueLabel, falseLabel)
BINARY(op, left, right)   SEQ(left, right)   TEMP(name)
ESEQ(stmt, exp)
```

ESEQ?

ESEQ(stmt, exp)

Execute stmt, then evaluate and return exp

```
def id(x):
    return x
y = id(5 + 5)
```

→

```
MOVE(t2, ESEQ(MOVE(t1, CALL(BINARY(Add,
5 + 5)))), t1))
```

Wait, this violates assembly's linearity!

Simplifies translation to IR, can clean it up later

Eliminating ESEQ

Recursively pull all ESEQ nodes out of the tree

Replace ESEQ node with its expression

```
MOVE(t2, ESEQ(MOVE(t1,CALL(BINARY(Add,
5 + 5)))), t1))
```

↓

```
MOVE(t1,CALL(BINARY(Add, 5 + 5))))

MOVE(t2, t1)
```

Instruction Selection

For each IR node, we recursively generate assembly instructions, starting with the most deeply nested

```
MOVE(t1,CALL(BINARY(Add, 5 + 5))))

MOVE(t2, t1)
```

```
mov t1, 5

mov t2, 5

add t1,t2

mov t1, %rdi

call

mov %rax, t1

mov t1, t2
```

[1]

[1] extra move instructions can be cleaned up by register allocator

Blocks and Traces

What about control flow?

We convert the linear list of statements into a set of **blocks**, and then arrange those **blocks** into **traces**

Blocks are always **entered** at the beginning and **exited** at the end

Traces are used to optimize control flow (more details on this later)

Blocks

A **block** is a sequence of statements where:

- The first statement is a LABEL
- The last statement is either a JUMP or CJUMP
- Each of these must occur **at most one time**

Generating Blocks

Starting with the first statement, when a LABEL is found, start a new block

Whenever a JUMP or CJUMP is found, the current block is ended

If any block does not end with a JUMP or CJUMP, add a JUMP to the next block's label

If any block has no LABEL, create a new one for it

Example

Color-coded blocks

```python
def main():
    t1 = 5
    t2 = 6
    if t1 < t2:
        t1 += t2
    else:
        t1 -= t2
    print(t1)
```

```
.main:

mov t1,5

mov t2,6

jmp .L1 if t1 < t2 else .L2
```

```
.L1

add t1,t2

jmp .L3
```

```
.L2

sub t1,t2

jmp .L3
```

```
.L3

call printf t1

jmp main.epilogue
```

How do we order these?

Generating Traces

Start with some initial block, and follow a possible execution path (the rest of the trace)

Repeat until all blocks are associated with one trace

For any blocks which end with an unconditional jump and are followed by the target label, remove the jump

```
Add all blocks to a list, Q

while Q is not empty:

    Start a new trace, T

    Remove the head element b from Q

    while b is not marked:

        Mark b

        Append b to the end of T

        for successor of b:

            if successor is not marked:

                b = c
   End T
```

Example

Traces

```
.main:

mov t1,5

mov t2,6

jmp .L1 if t1 < t2 else .L2
```

Trace #1

```
.L2

sub t1,t2

jmp .L3
```

```
.L3

call printf t1

jmp main.epilogue
```

Trace #2

```
.L1

add t1,t2

jmp .L3
```

```
def main():
    t1 = 5
    t2 = 6
    if t1 < t2:
        t1 += t2
    else:
        t1 -= t2
    print(t1)
```

Liveness Analysis

What is liveness?

For a given variable, it is live if its current value will be used in the future

We can calculate the **live range** of a particular variable by starting at a **use** of a variable and searching backwards until that variable is **defined**

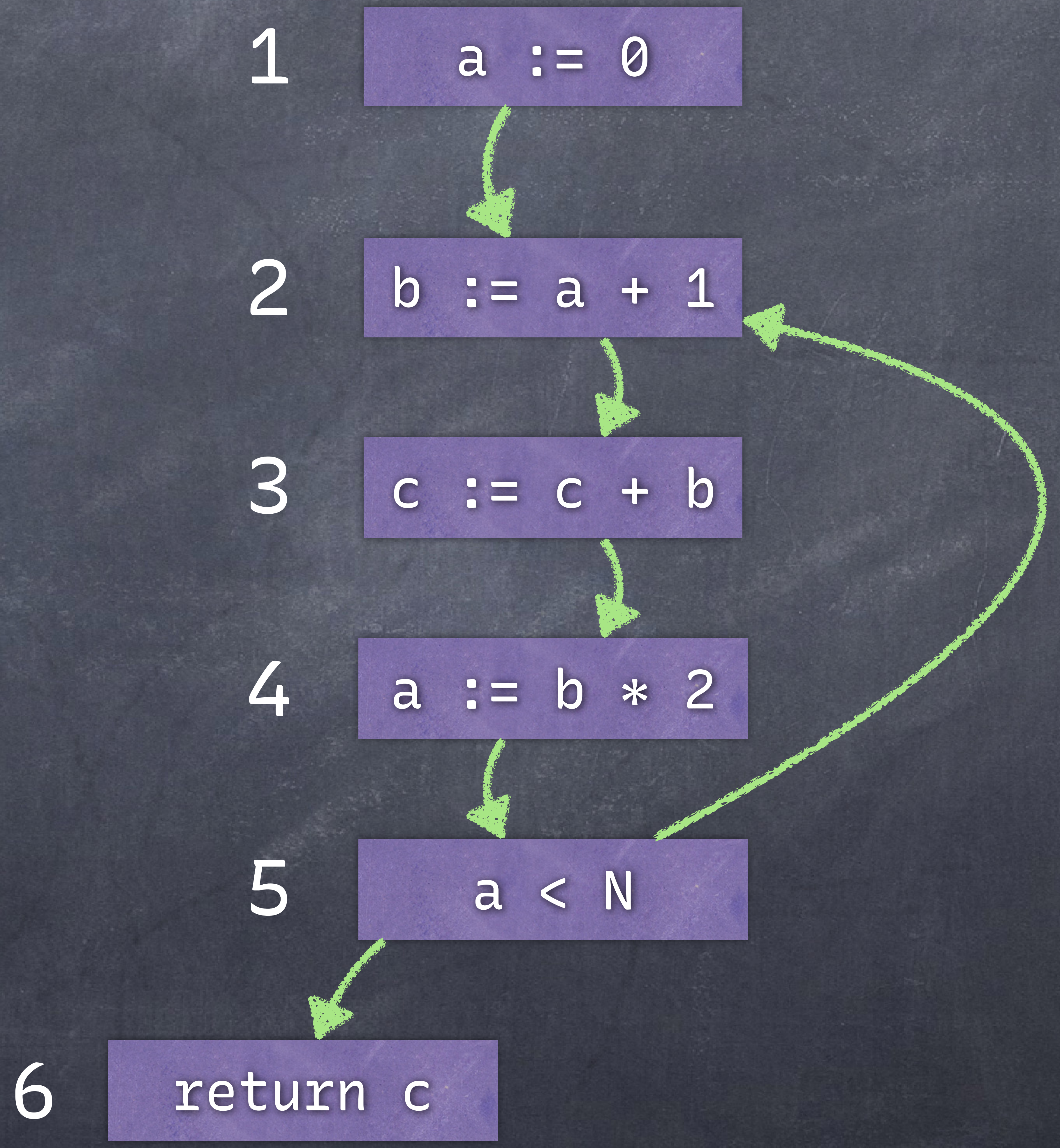
This can be done either **one-variable-at-a-time** or using set equations

Example

a = {1 → 2, 4 → 5, 5 → 2}

b = {2 → 3, 3 → 4}

c = entire range



n → m is (n, m]

Interference Graphs

	a	b	c
a			x
b			x
c	x	x	

How are control flow graphs created?

Based on breadth-first search

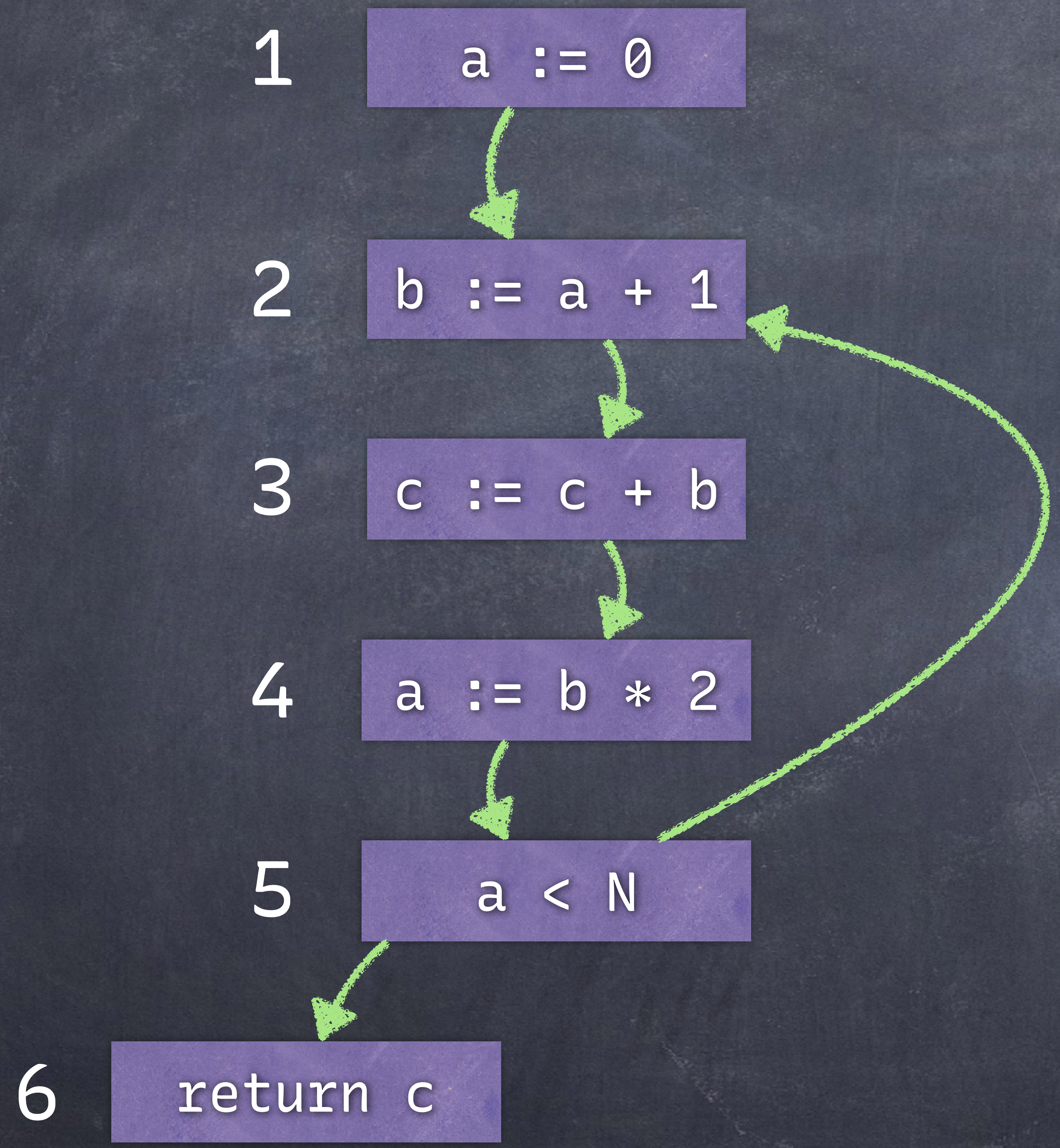
Start with a queue containing the index of the first statement

For each statement, its “neighbors” are any successor nodes

Add an edge between statement and each neighbor

Add neighbors to queue

Example Revisited



(5) is the only interesting node

Has two successors: 6 and 2

Project Challenges

Tooling Pitfalls

There are various ways to automate lexical analysis and parsing for both LL(k) and LR(k) classes of grammars

Examples

JavaCC

SableCC (Java)

pest (Rust)

Type-Checking

```
foo.bar.baz = 5;
```

Figure out whether bar exists on foo and what its type is

Now that we know what bar's type is, see if baz exists, etc.

Straightforward concept, several small errors and many cases to handle

What if bar is a function?

Repetitive Analysis

```
foo.bar.baz = 5;
```

It's useful to know what these types are in future phases

We don't want to recompute every time

Initially was not saving results of type-checking phase which complicated future phases

How do we eliminate abstractions?

High-level concepts difficult to translate to something compilable

Classes or structs may have nested fields, but stack frames are linear

Solution: flatten structs

ESEQ and its myriad issues

Unresolved Questions:

Scope of Program Input

What is the class of valid programs?

Where are the edge cases?

How do I adequately test algorithms on **many** different inputs?

Thanks for listening