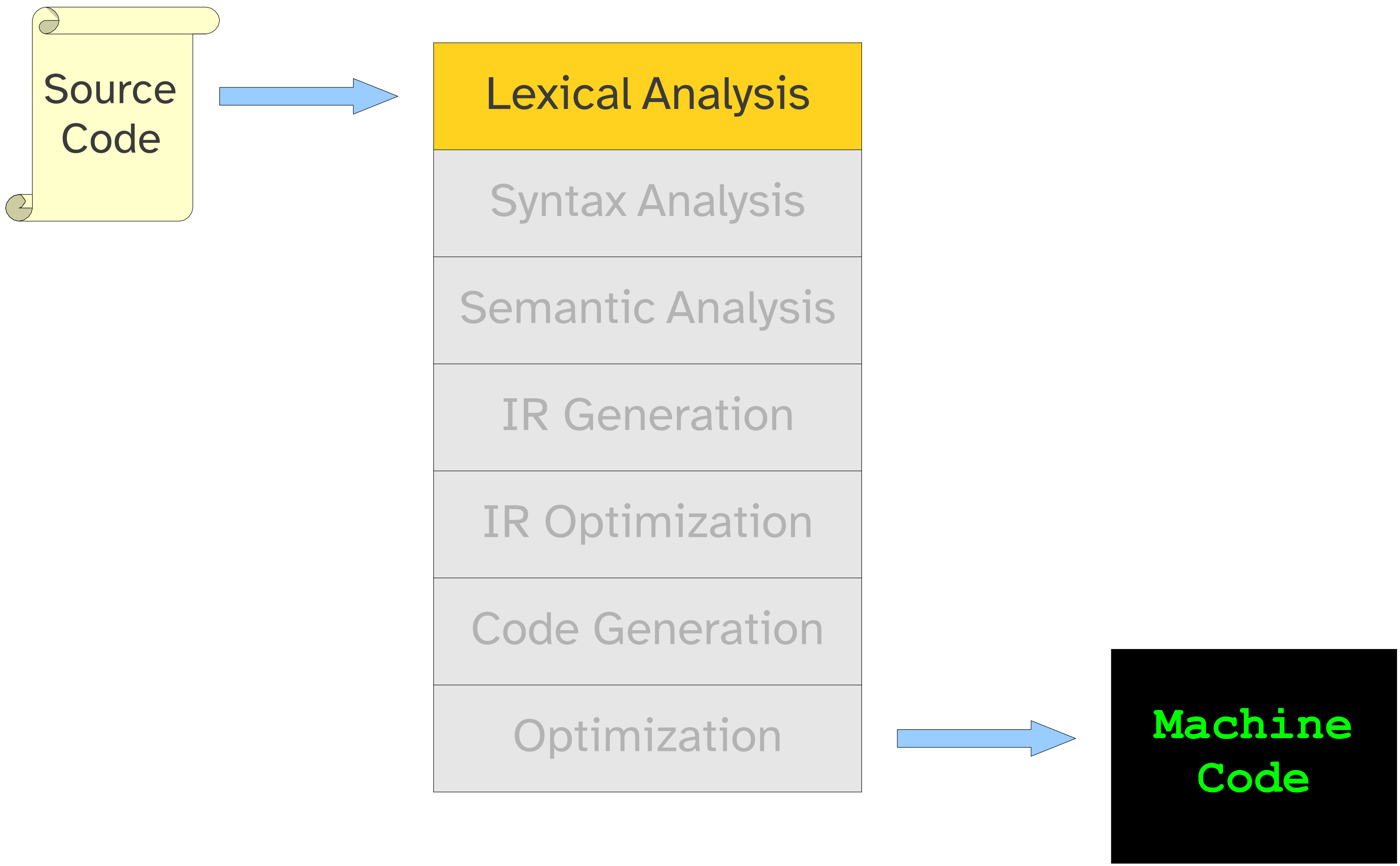
Lexical Analysis

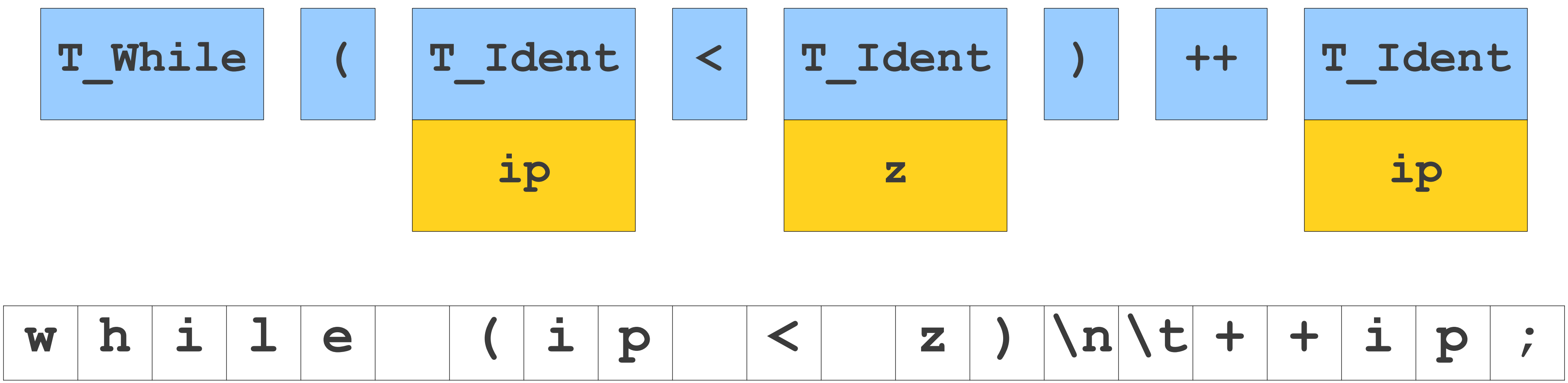
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

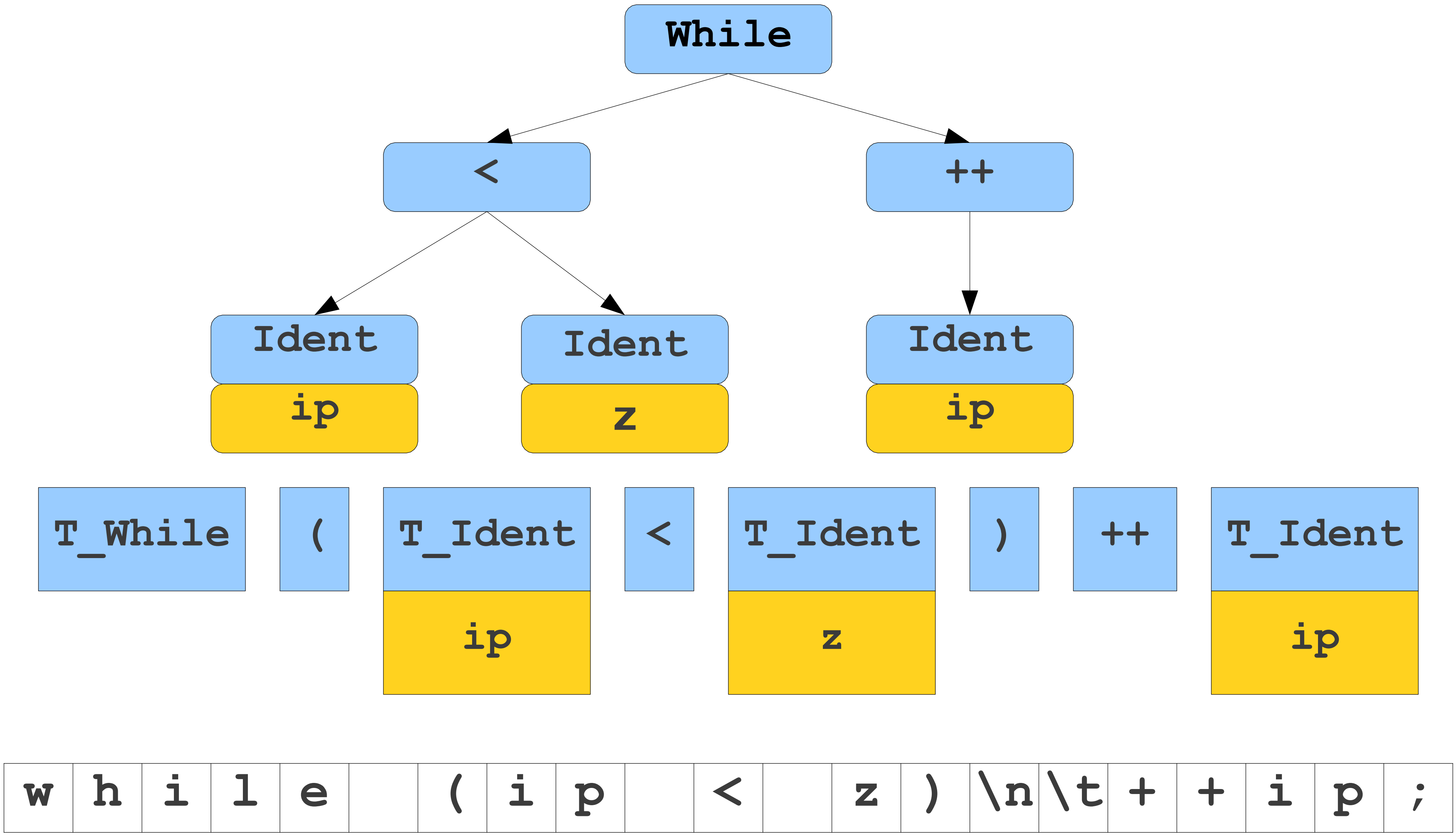
```
while (ip < z)
    ++ip;
```

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```




```
while (ip < z)
    ++ip;
```


While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
  ++ip;
```

```
do[for] = new 0;
```

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```

T_Do	[	T_For	]	=	T_New	T_IntConst
						0

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```

T_Do	[	T_For	]	=	T_New	T_IntConst
						0

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Scanning a Source File

`w`	`h`	`i`	`l`	`e`		`(`	`1`	`3`	`7`		`<`		`i`	`)`	`\n`	`\t`	`+`	`+`	`i`	`;`

`T_While`

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

The piece of the original program from which we made the token is called a **lexeme**.

`T_While`

This is called a **token**. Think of it as an enumerated type representing what logical entity we read out of the source code.

Scanning a Source File

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

`T_While`

Sometimes we will discard a lexeme rather than storing it for later use.

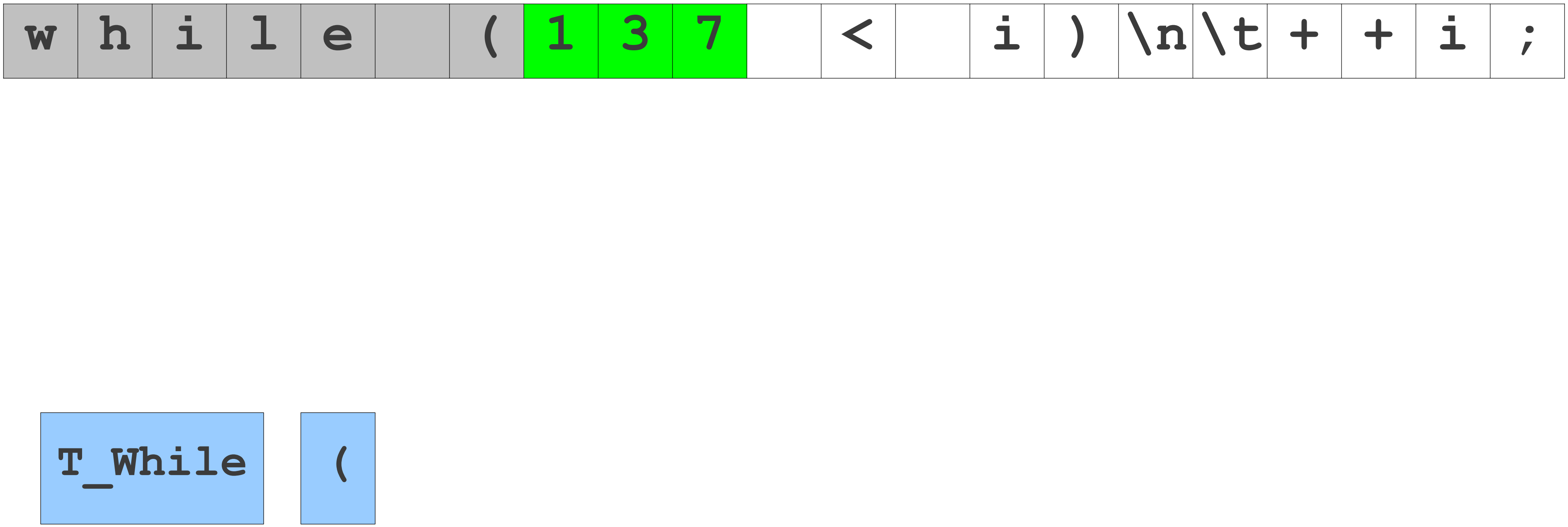
Here, we ignore whitespace, since it has no bearing on the meaning of the program.

Scanning a Source File

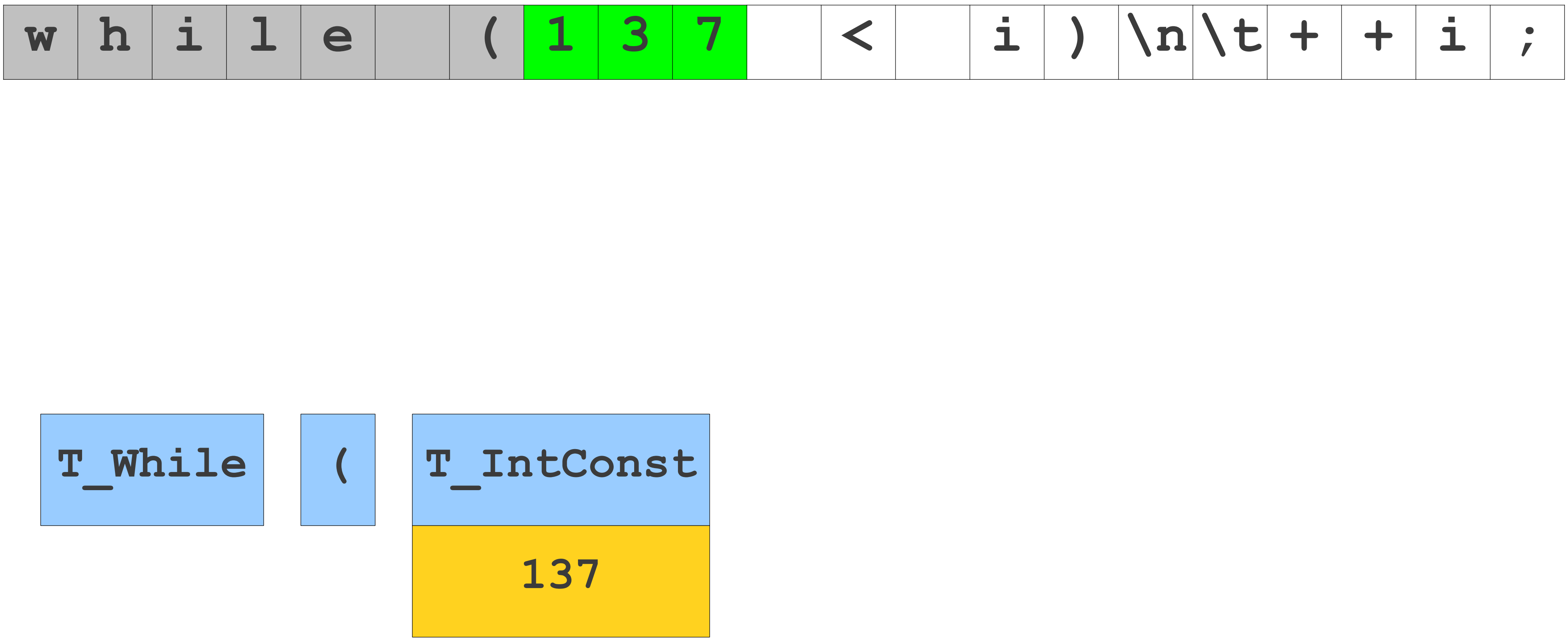
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While

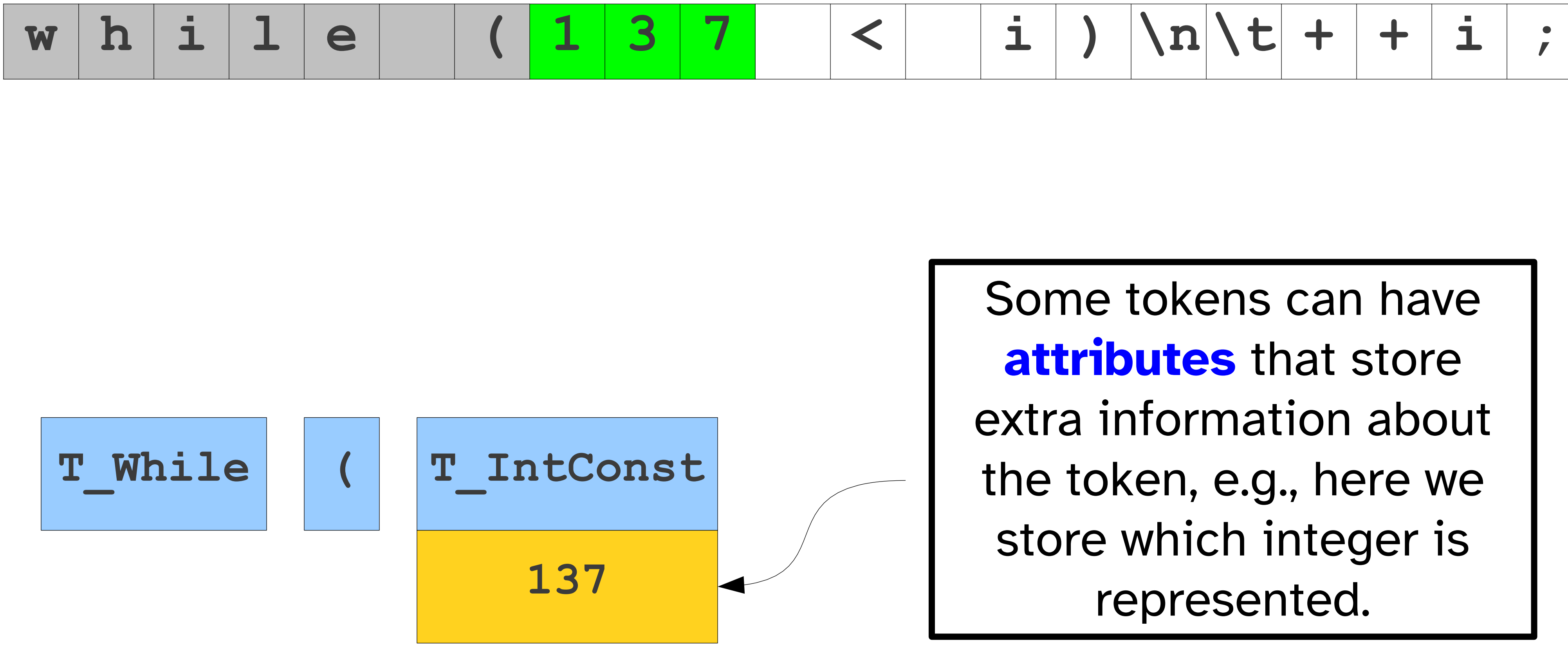
Scanning a Source File



Scanning a Source File



Scanning a Source File



Some tokens can have **attributes** that store extra information about the token, e.g., here we store which integer is represented.

Recap: Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.

- Each token is associated with a **lexeme**.

- Each token may have optional **attributes**.

Choosing Tokens

Scanning is Hard

- C++: Nested template declarations

```
vector<vector<int>> myVector
```

Scanning is Hard

- PL/1: Keywords can be used as identifiers.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF;
```

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF;
```

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like `if` and `while`
- Some tokens might be associated with lots of different lexemes:
  - All variable names, all possible numbers, all possible strings, etc.

How do we describe which (potentially infinite) set of lexemes is associated with each token type?

Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have finite descriptions:
  - Define the language using an automaton.
  - Define the language using a grammar.
  - Define the language using a regular expression.
- We can use these compact descriptions of the language to define sets of strings.
  - We will use all of these approaches

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (*the regular languages*).
- Often provide a compact and human-readable description of the language.

Atomic Regular Expressions

- The regular expressions we will use in this course begin with two simple building blocks.
- The symbol **ε** is a regular expression matches the empty string.
- For any symbol **`a`**, the symbol **`a`** is a regular expression that just matches **`a`**.

Compound Regular Expressions

- If R_1 and R_2 are regular expressions
  - **R_1R_2** is a regular expression representing the **concatenation** of R_1 and R_2.
  - **$R_1 | R_2$** is a regular expression representing the **union** of R_1 and R_2.
- If R is a regular expression
  - **R*** is a regular expression for the **Kleene closure** of R.
    - **R^+** is shorthand for **R R***
  - **(R)** is a regular expression with the same meaning as R. (aka scope)

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R^*$$

$$R_1R_2$$

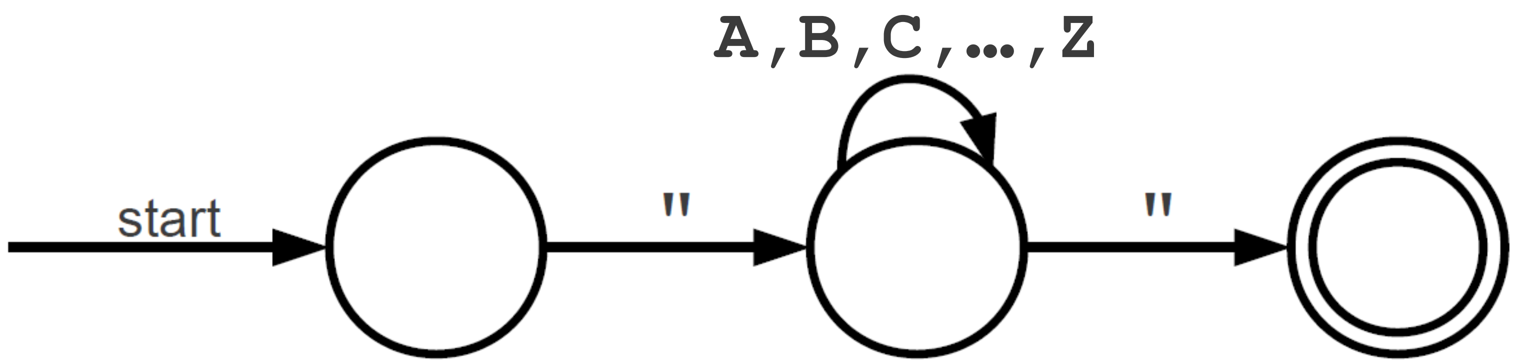
$$R_1 \mid R_2$$

Matching Regular Expressions

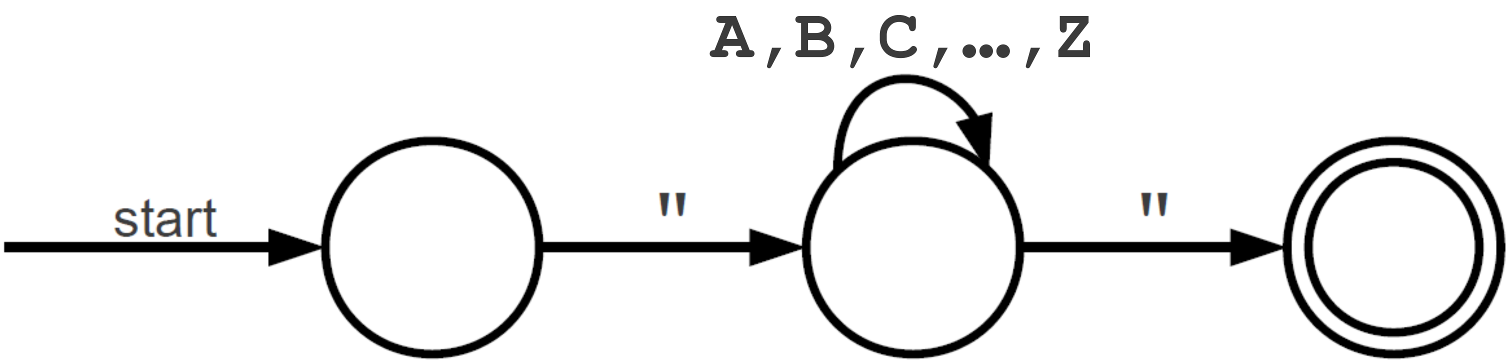
Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
- There are two main kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata),
  - **DFA**s (**deterministic** finite automata)
- Automata are best explained by example...

A Simple Automaton

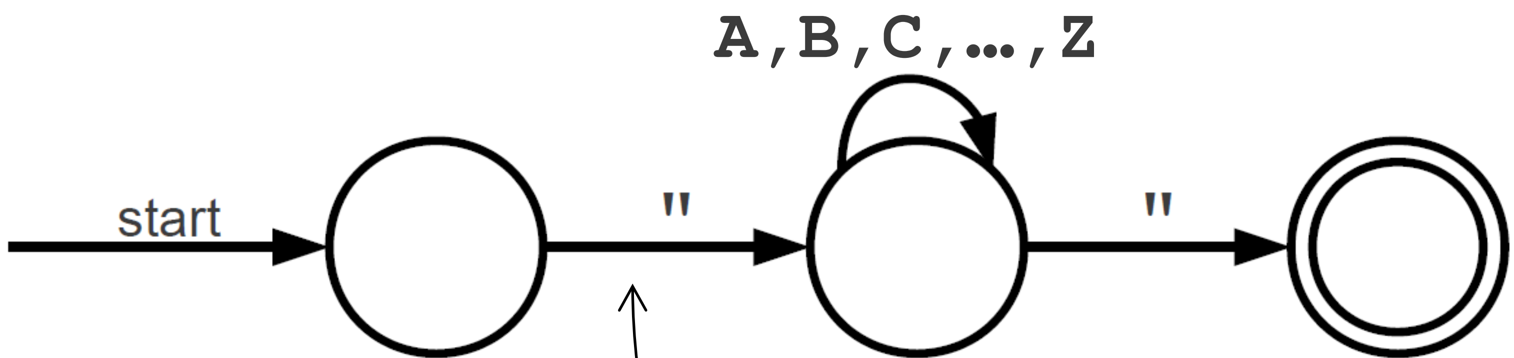


A Simple Automaton



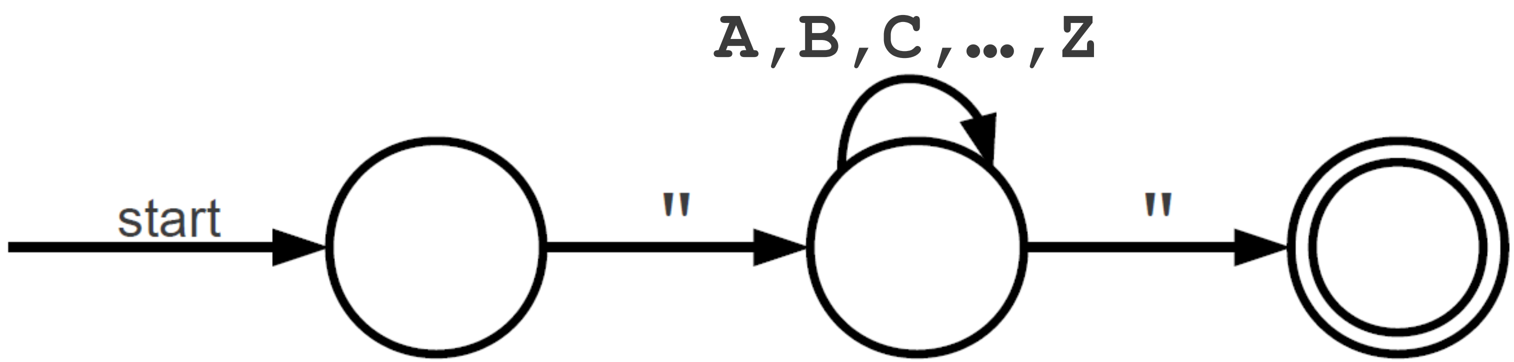
Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

A Simple Automaton



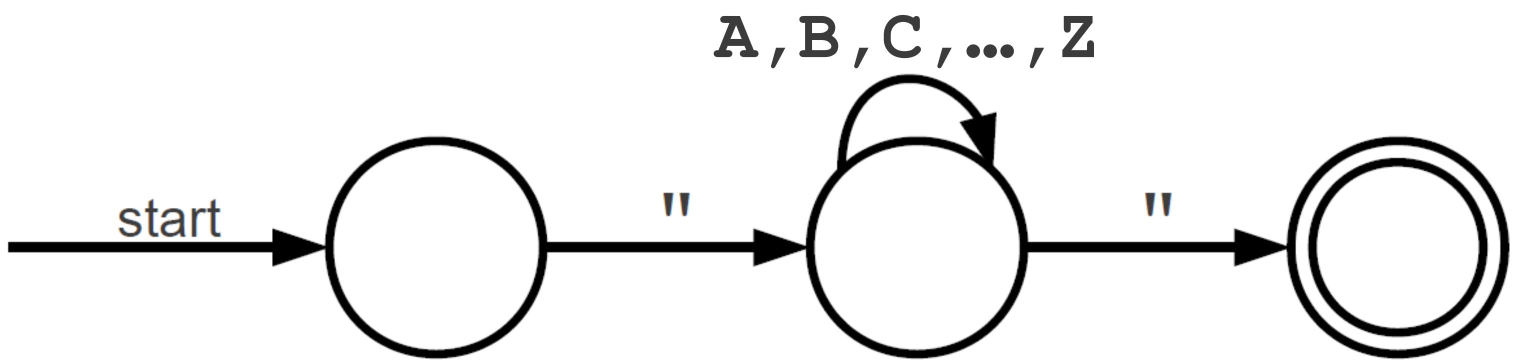
These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

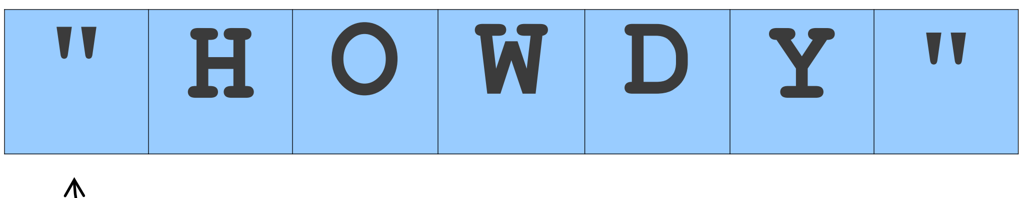
A Simple Automaton



The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.

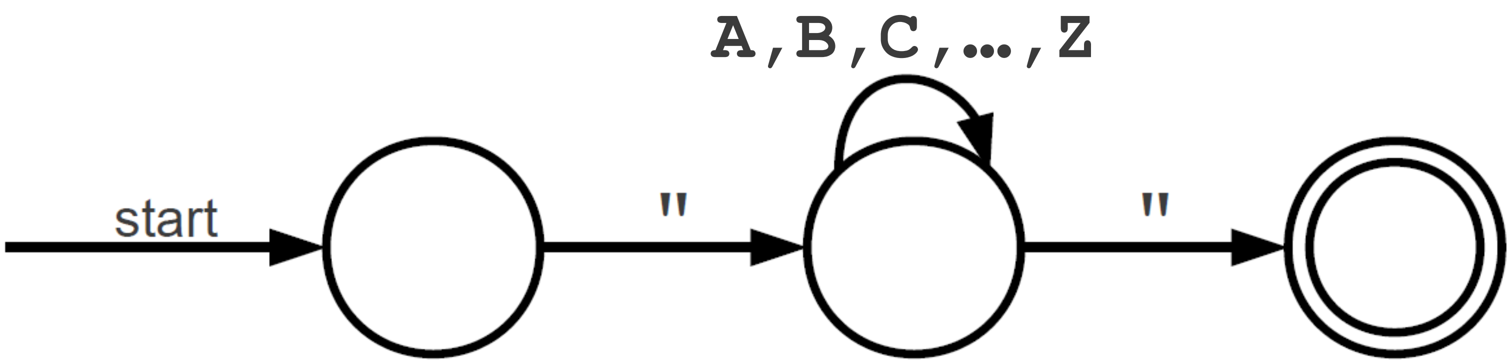
A Simple Automaton



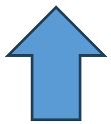


The automaton takes a string as input and decides whether to accept or reject the string.

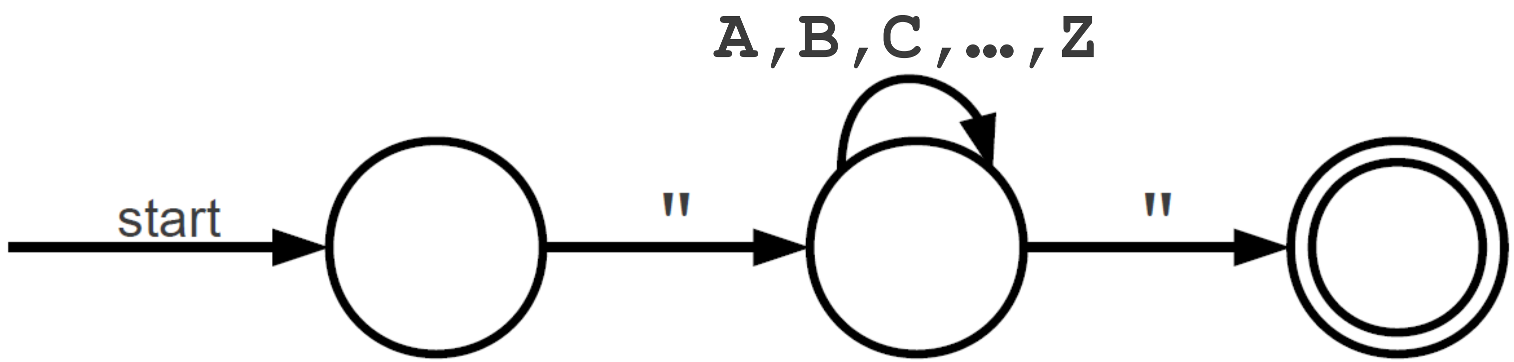
A Simple Automaton

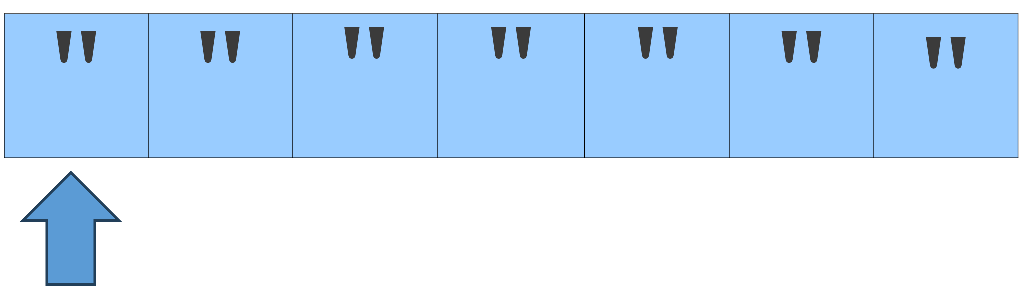


"	H	O	W	D	Y	"

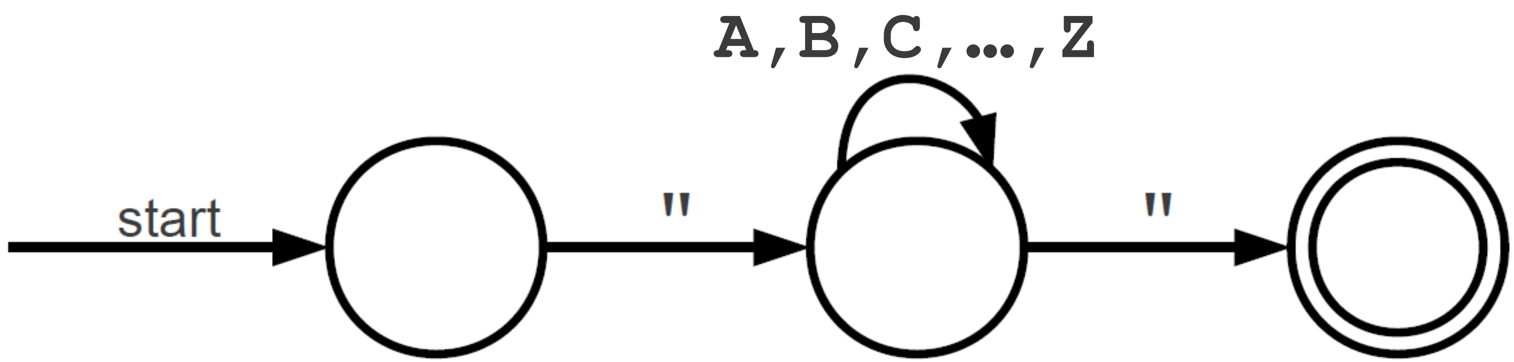


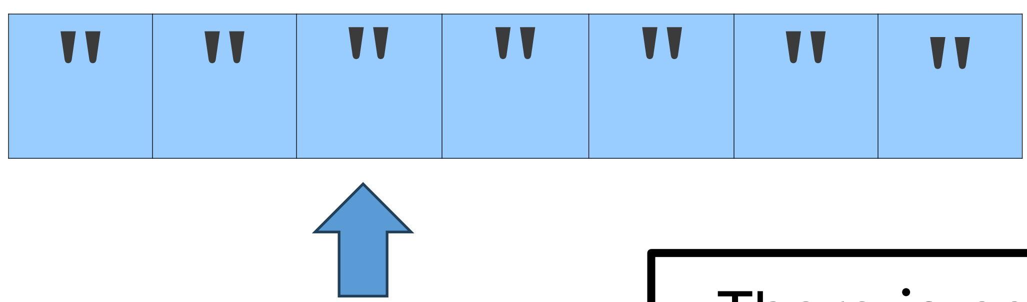
A Simple Automaton





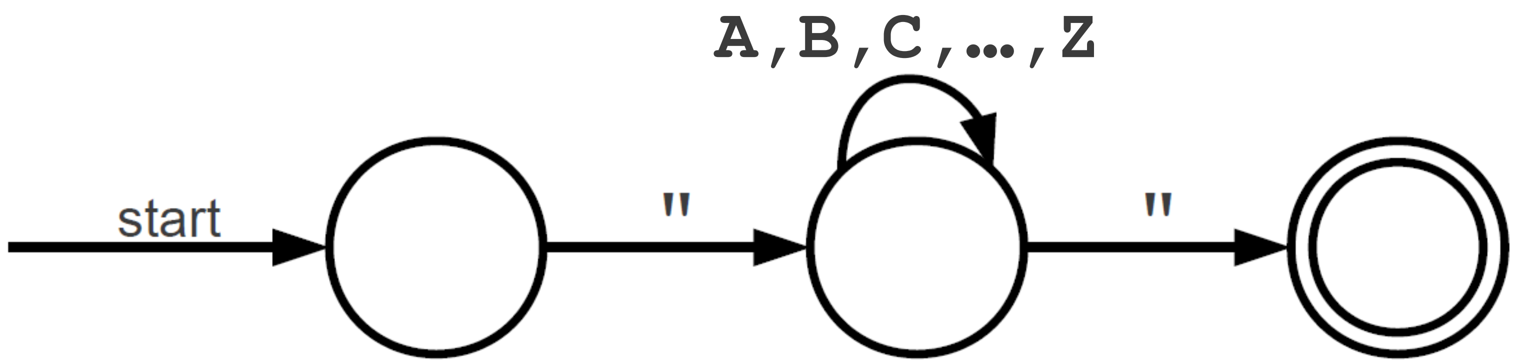
A Simple Automaton

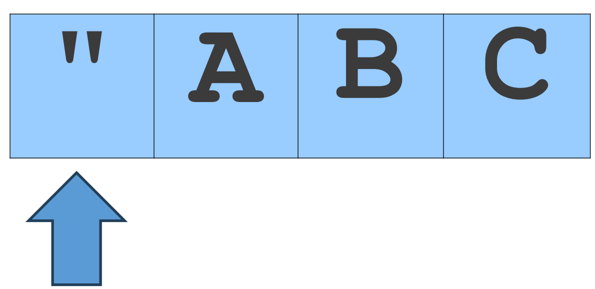




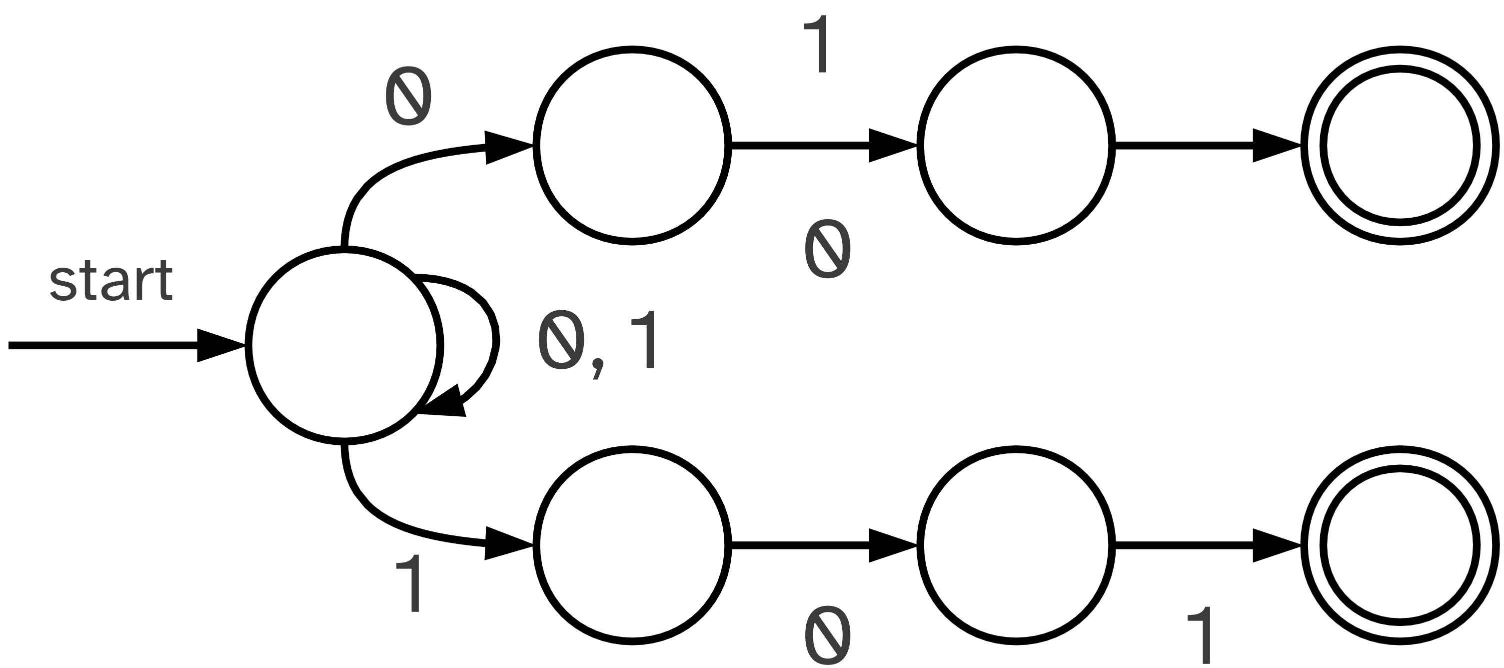
There is no transition on " here, so the automaton **dies** and rejects.

A Simple Automaton

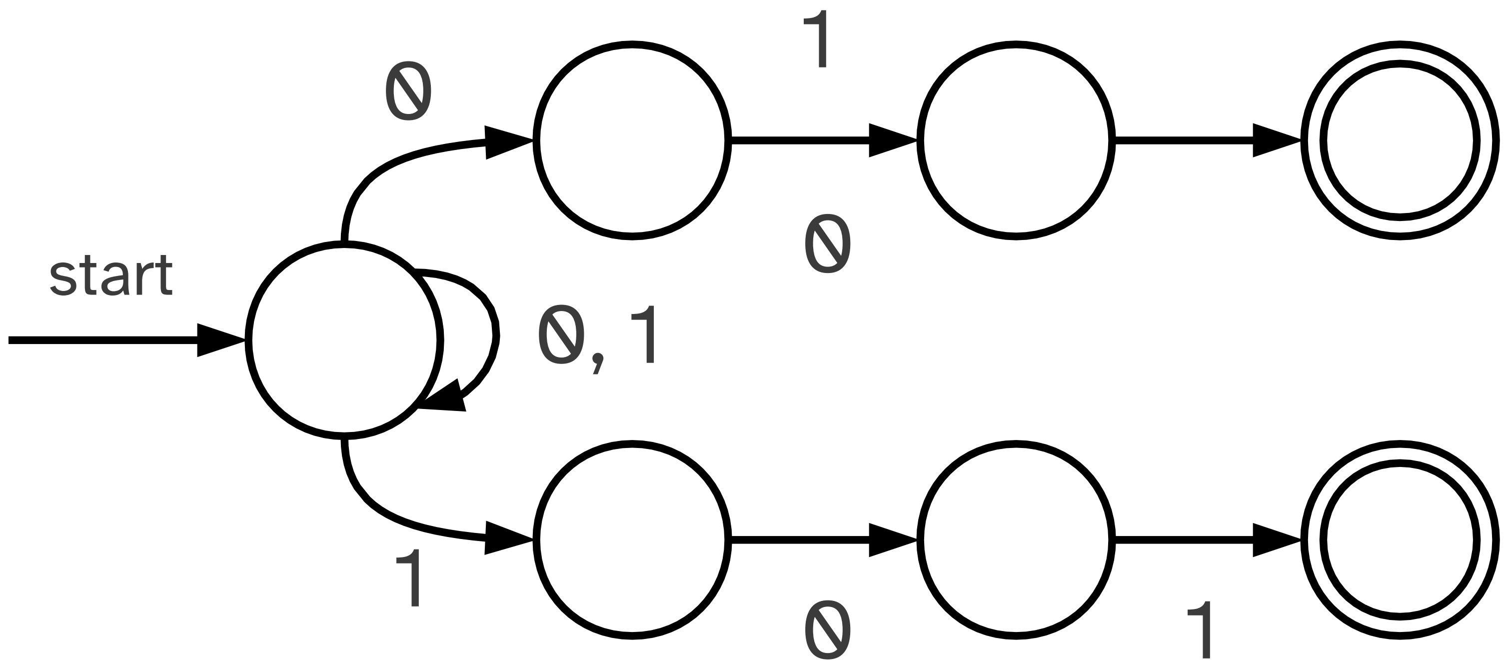


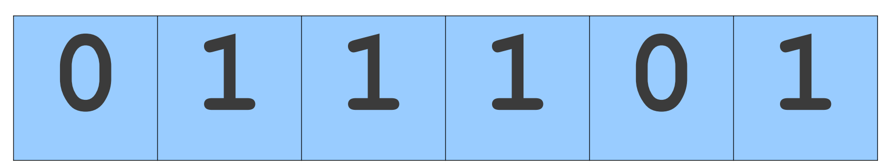


A More Complex Automaton

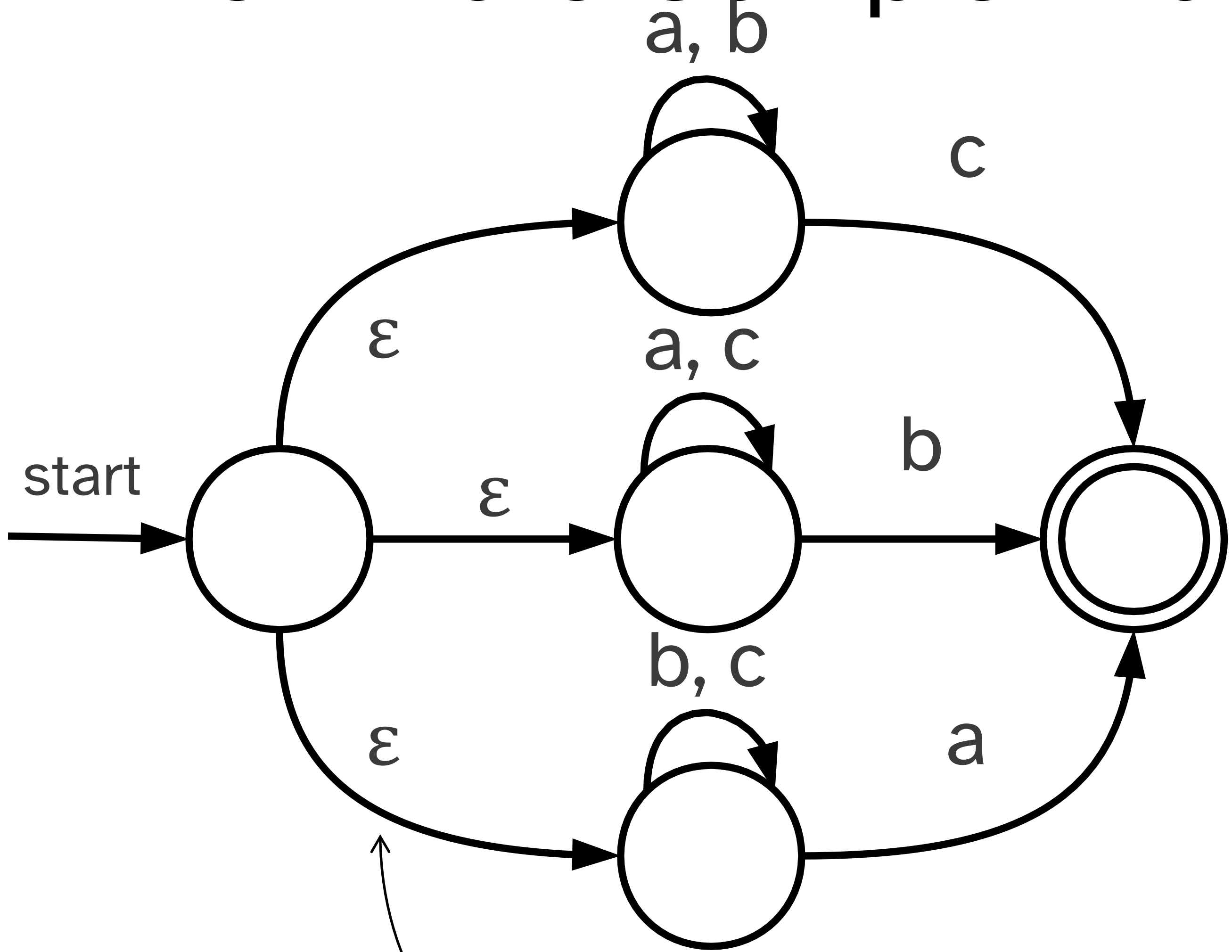


A More Complex Automaton



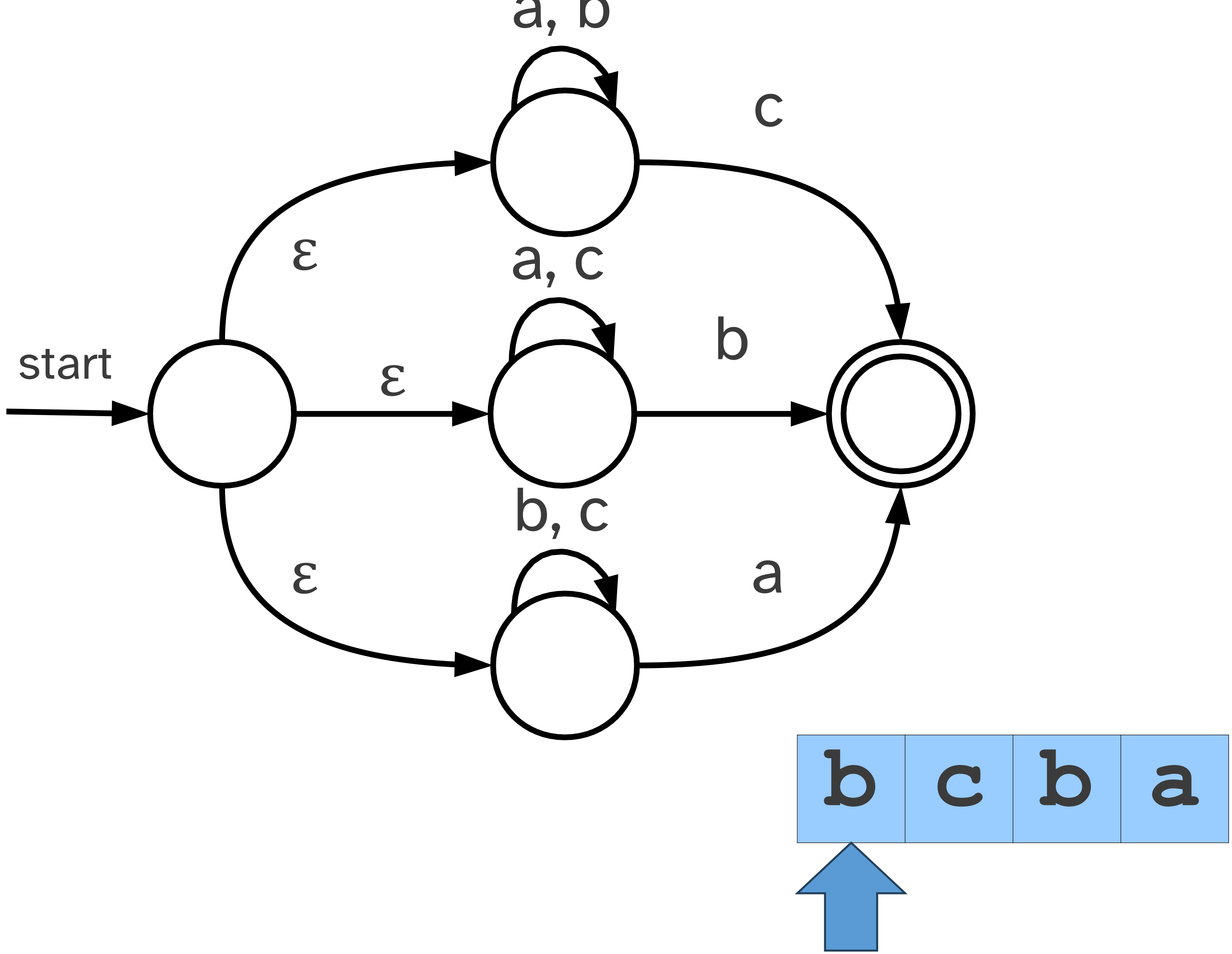


An Even More Complex Automaton



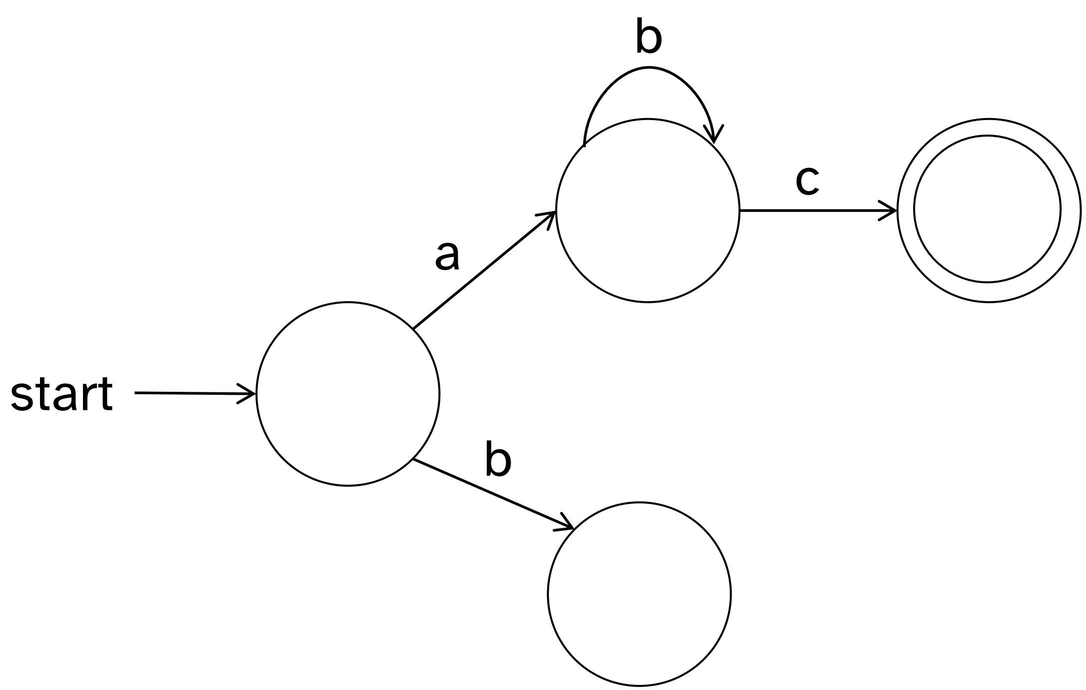
These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

An Even More Complex Automaton



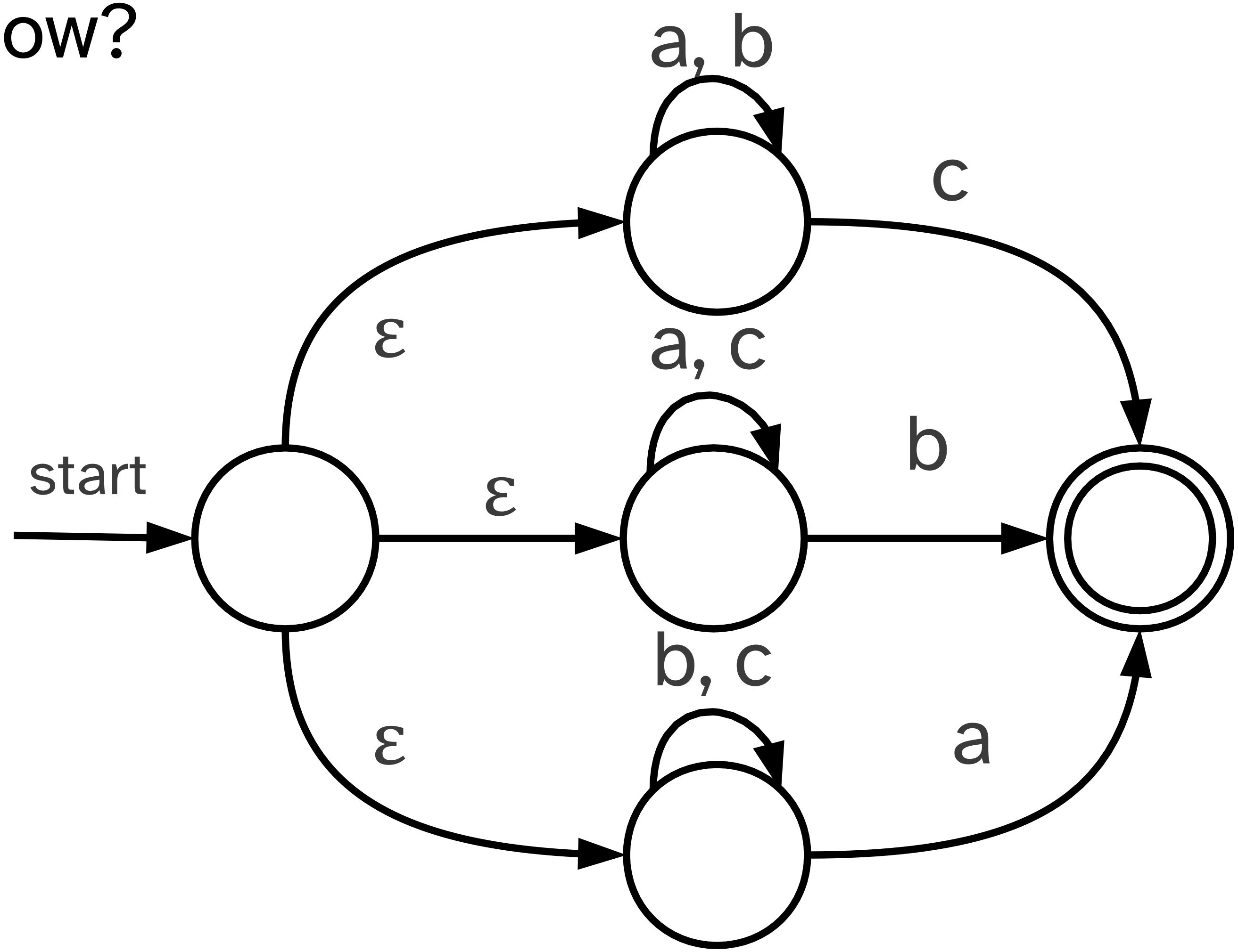
Exercise

- What is the language defined by the automaton below?



Exercise

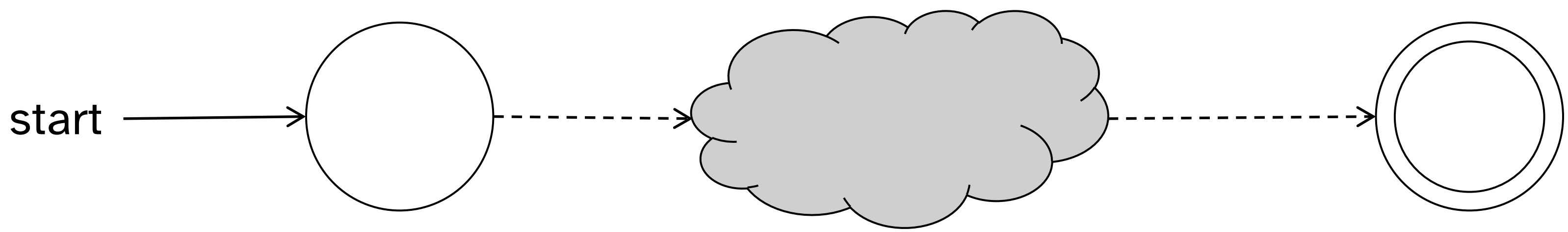
- What is the language defined by the automaton below?



RegEx vs. automata

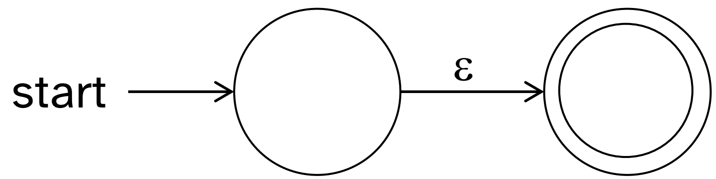
- Regular expressions are **declarative**
  - Offer compact way to define a regular language *by humans*
  - Don’t offer direct way to check whether a given word is in the language
- Automata are **operative**
  - Define an *algorithm* for deciding whether a given word is in a regular language
  - Not a natural notation for humans

From Regular Expressions to NFAs

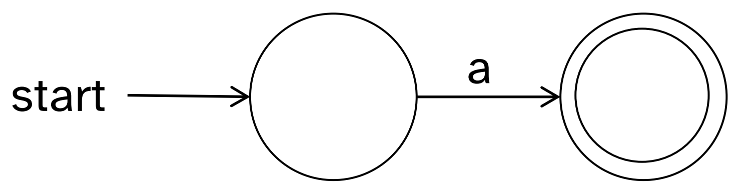


- By induction on the structure of the RegEx:
  - For each sub-expression R we build an automaton with exactly one start state and one accepting state
  - Start state has no incoming transitions
  - Accepting state has no outgoing transitions

Base Cases

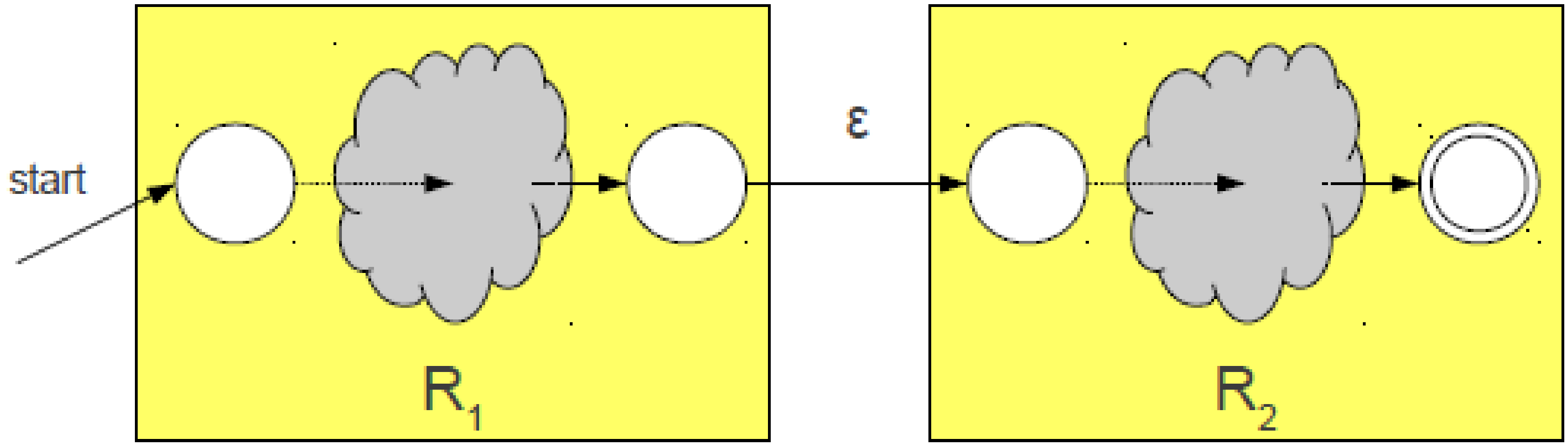


Automaton for ε

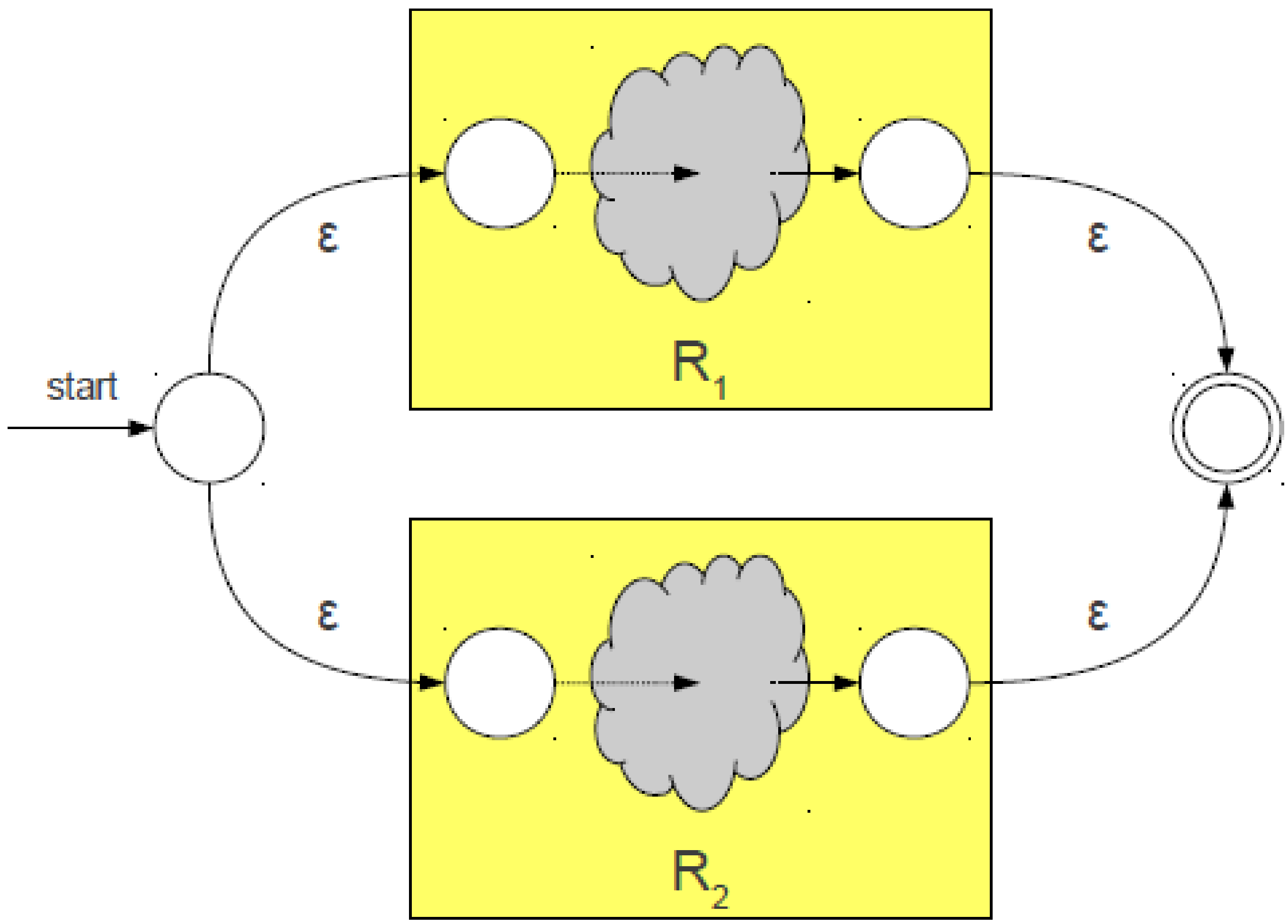


Automaton for single character **a**

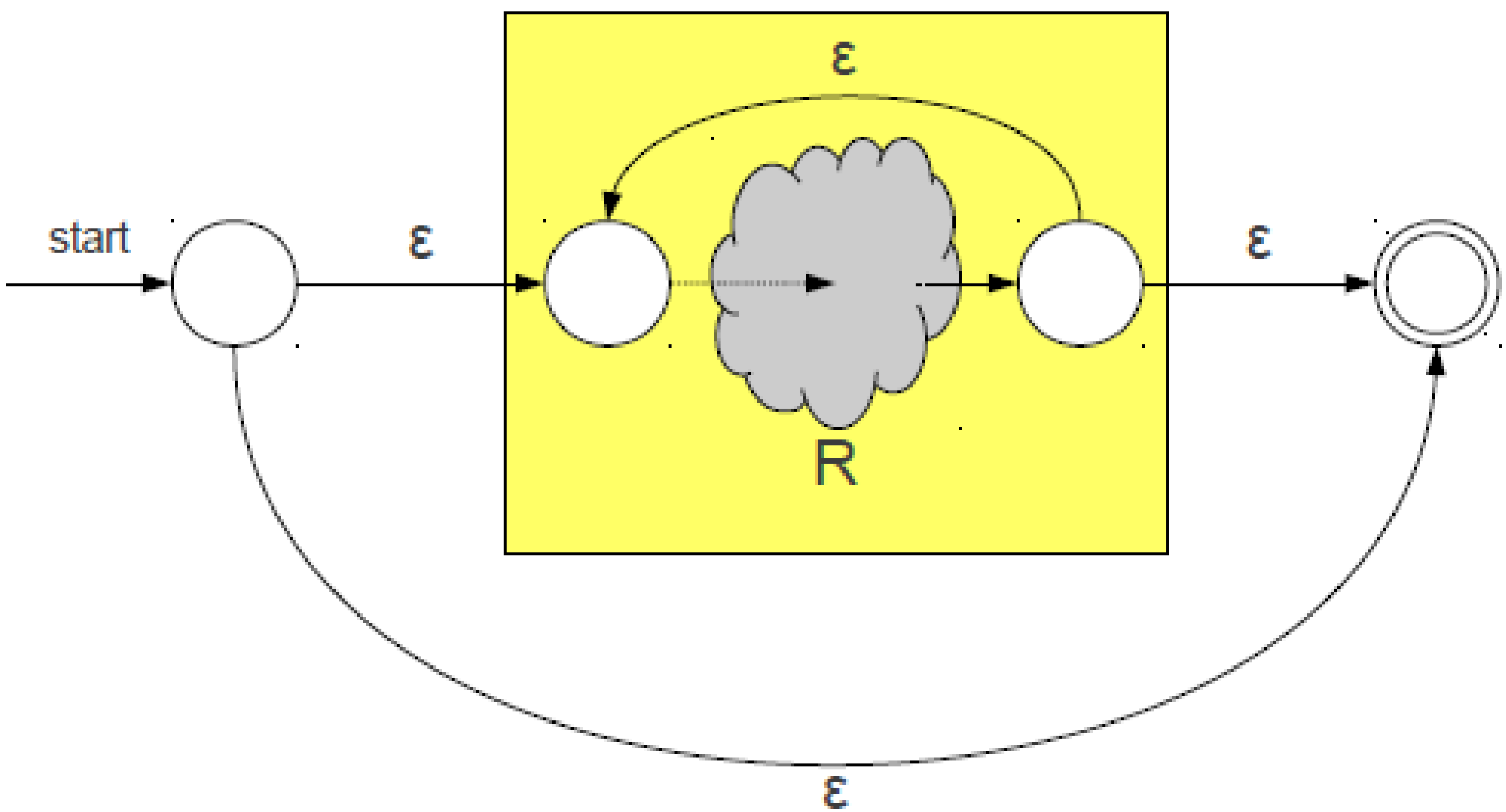
Construction for R_1R_2



Construction for $R_1 \mid R_2$



Construction for R*



Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

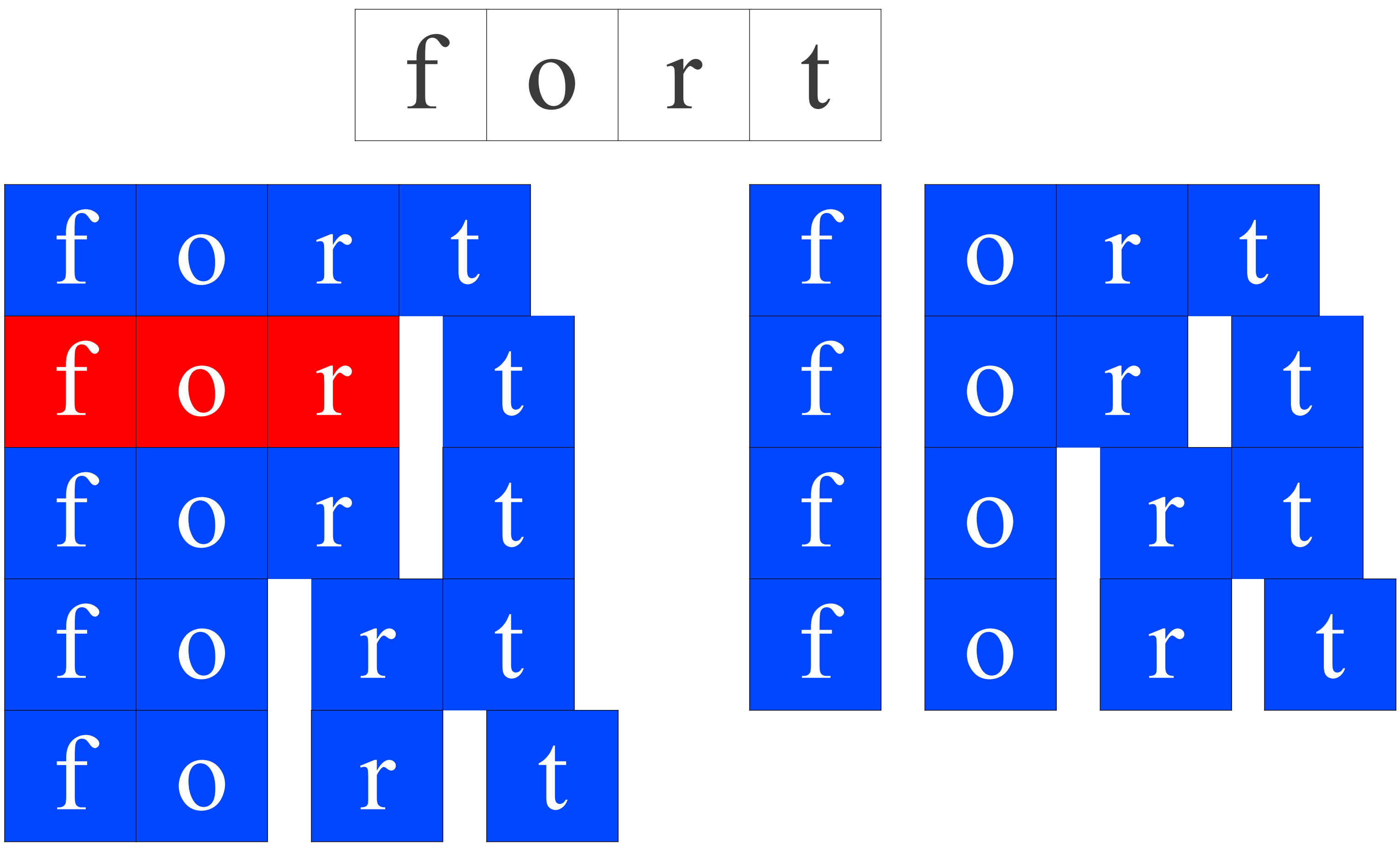
Lexing Ambiguities

```
T_For          for
T_Identifier   [A-Za-z_][A-Za-z0-9_]*
```

f	o	r	t

Lexing Ambiguities

T_For	for
T_Identifier	[A-Za-z_][A-Za-z0-9_]*



Conflict Resolution

- Assume all tokens are specified as regular expressions.
- Algorithm: **Left-to-right scan**.
- Tiebreaking rule one: **Maximal munch**.
    - Always match the longest possible prefix of the remaining text.

Lexing Ambiguities

`T_For`	`for`
`T_Identifier`	`[A-Za-z_][A-Za-z0-9_]*`

f	o	r	t

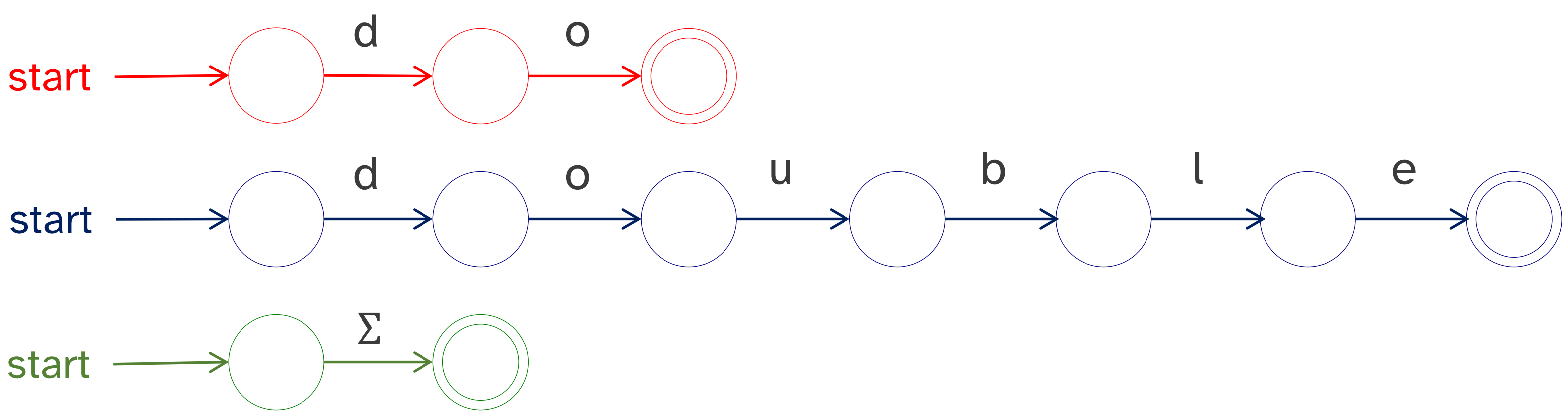
f	o	r	t

Implementing Maximal Munch

- Given a set of regular expressions, how can we use them to implement maximum munch?
- Algorithm:
  - Convert expressions to NFAs.
  - Run all NFAs in parallel, keeping track of the last match.
  - When all automata get stuck, report the last match and restart the search at that point.

Maximal Munch

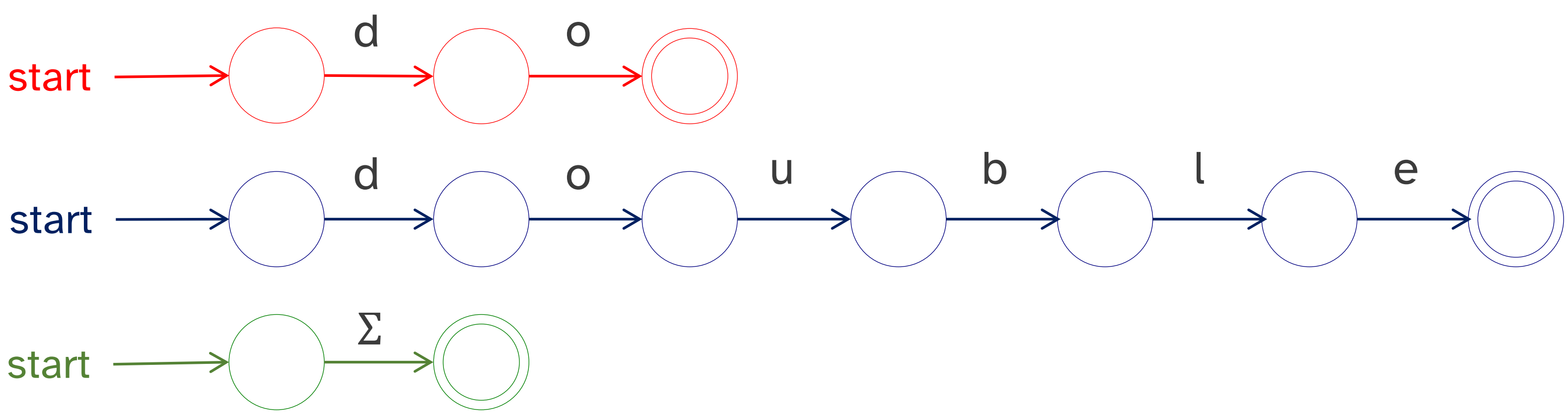
T_Do do
T_Double double
T_Mystery [A-Za-z]

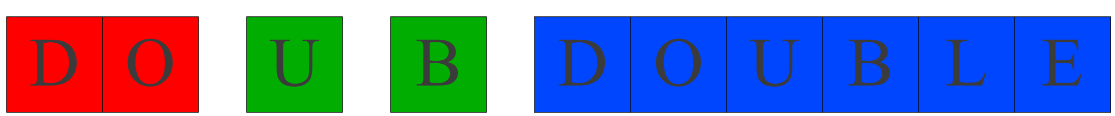


D	O	U	B	D	O	U	B	L	E

Maximal Munch

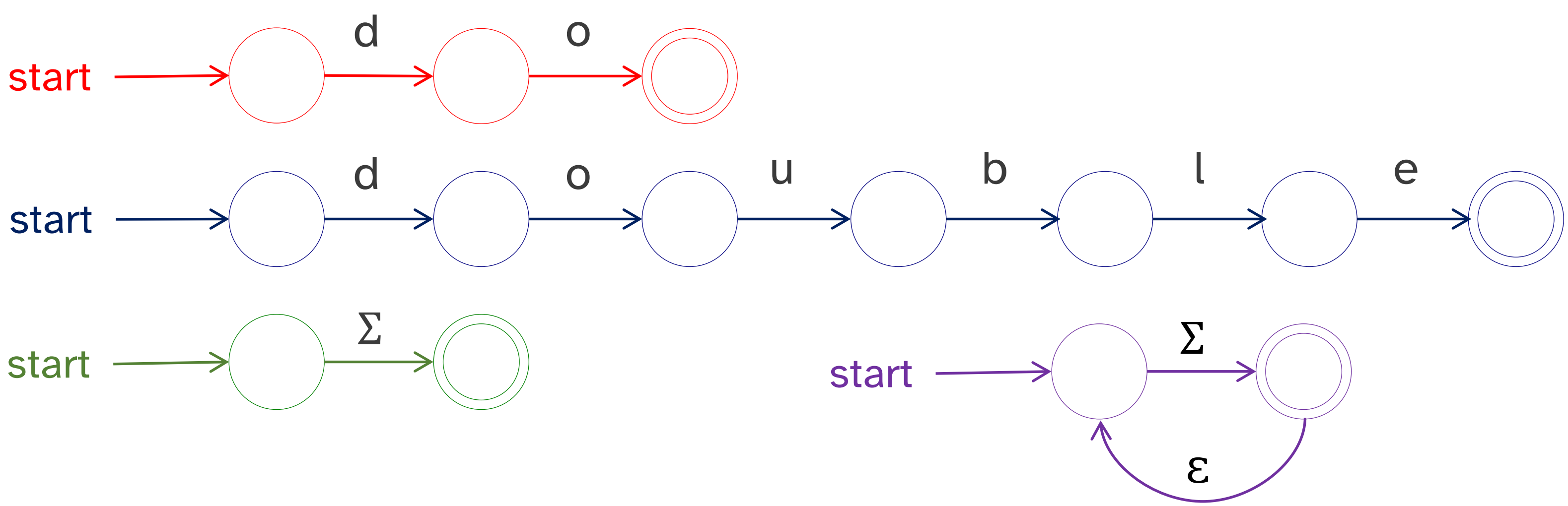
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





Repeat: Maximal Munch

T_Do	do
T_Double	double
T_Mystery	[A-Za-z]
T_Identifier	[A-Za-z][A-Za-z]*



D	O	U	B	D	O	U	B	L	E

Simplified Maximal Munch

- General idea: Consume the **largest** possible token that makes sense. Produce the token then proceed.
  - Maximal Munch:
    - Consume characters until you no longer have a valid transition. If you have characters left to consume, back track to the last valid accepting state and resume.
  - Simplified Maximal Munch:
    - Consume characters until you no longer have a valid transition. If you are currently in an accepting state, produce the token and proceed. Otherwise go to an error state.

What is the difference?

Σ = {a,b,c}

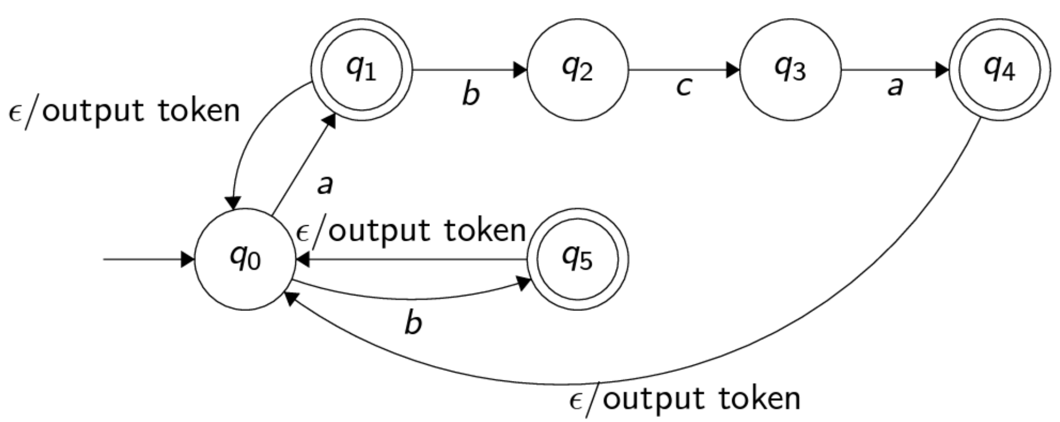
Tokens = {a, b, abca}

Input = **ababca**

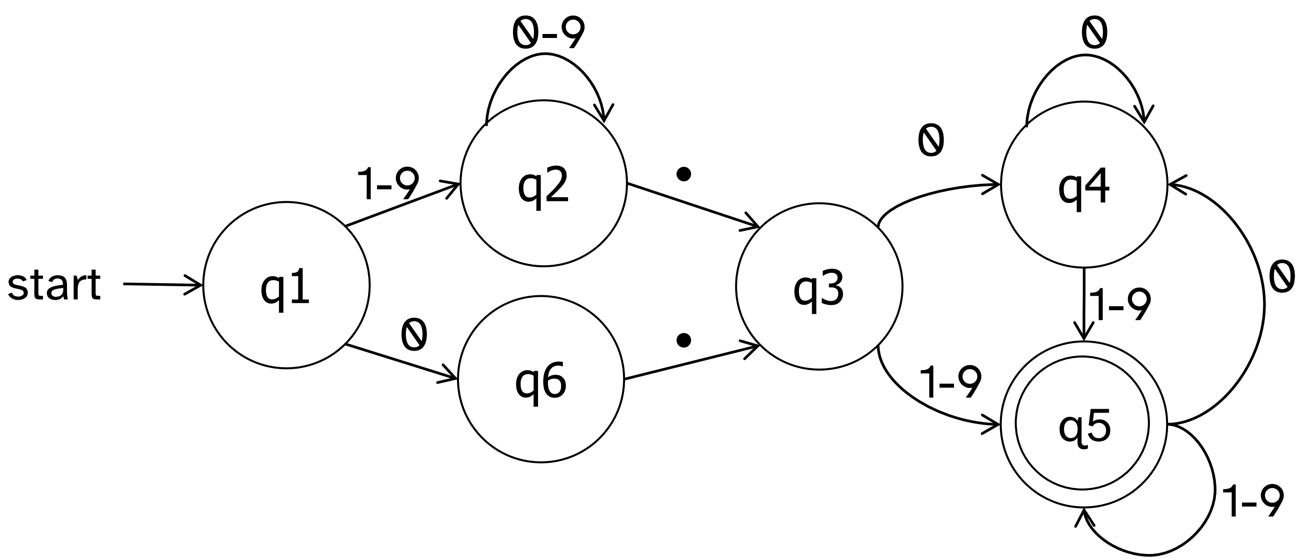
Not a typo:
~~flags this state as accepting~~

- ## Maximal Munch
- Consumes **a** and flags this state as accepting, then **b** then tries to consume **a** but ends up in error.
- Backtracks to first **a** (last accepting state) . Output token a.
- Resumes consuming **b** and flags this state as accepting, then error.
- Backtracks to output token b.
- Resumes … finally output token abca.
- **Accept the string**

- ## Simplified Max. Munch
- Consumes **a**, then **b**, then tries to consume **a** but ends up in error.
- Checks to see if **ab** is accepting, it is not. **Rejects the input**
- Usually good enough
- Is used in practice



Exercise: Run on input 1.230.2



Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

d	o	u	b	l	e
d	o	u	b	l	e

More Tiebreaking

- When two regular expressions apply, choose the one with the greater "priority."
- Simple priority system: **pick the rule that was defined first.**

Other Conflicts

```
T_Do         do
T_Double     double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

d	o	u	b	l	e

d	o	u	b	l	e

One Last Detail...

- We know what to do if *multiple* rules match.
- What if *nothing* matches?

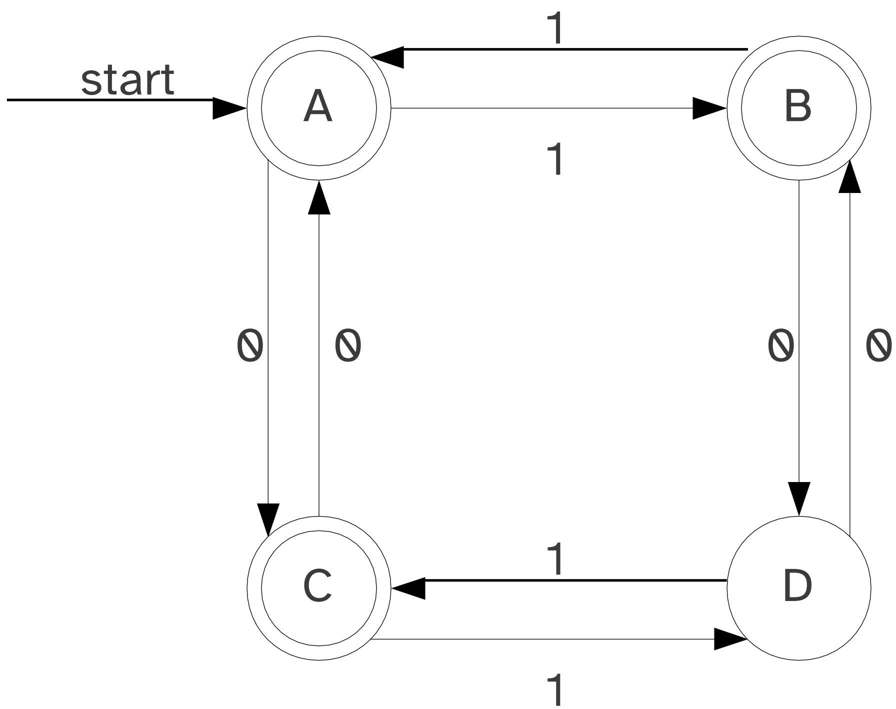
Challenges in Scanning

- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

DFAs

- The automata we've seen so far have all been NFAs.
- A **DFA** is like an NFA, but with tighter restrictions:
  - Every state must have **exactly one** transition defined for every letter.
  - ε-moves are not allowed.

A Sample DFA



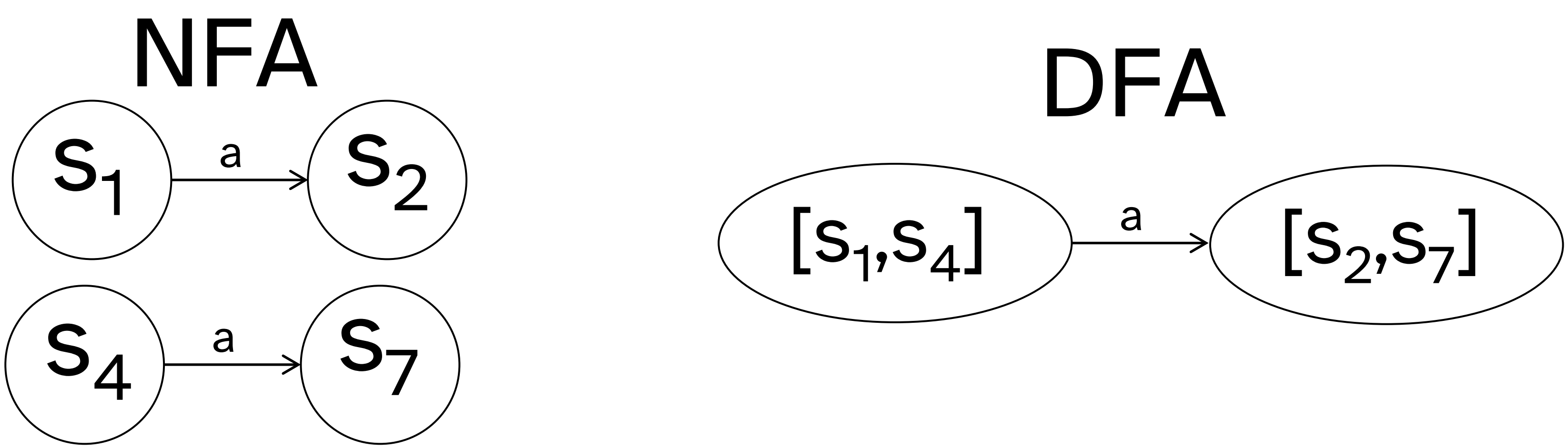
	0	1
A.	C	B
B.	D	A
C.	A	D
D.	B	C

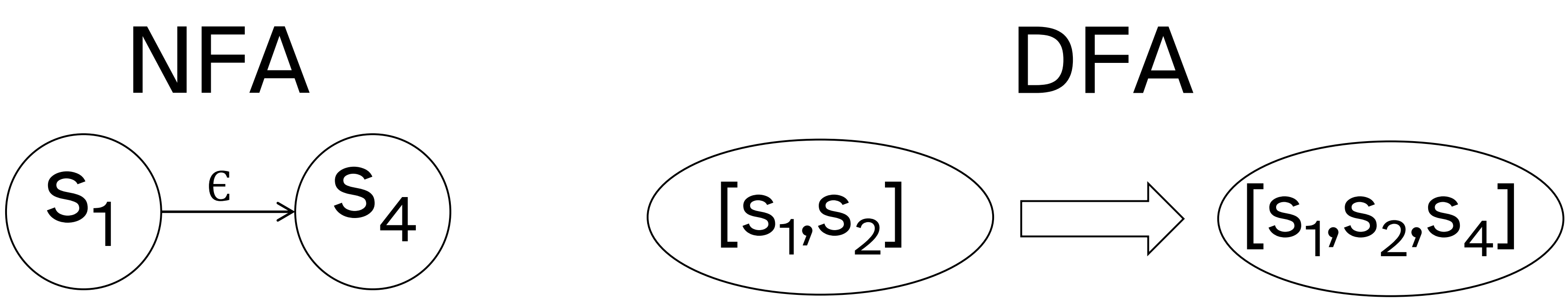
Subset Construction

- NFAs can be in many states at once, while DFAs can only be in a single state at a time.
- Key idea: **Make the DFA simulate the NFA**.
- Have the states of the DFA correspond to the *sets of states* of the NFA.
- Transitions between states of DFA correspond to transitions between *sets of states* in the NFA.

From NFA to DFA

- NFA without ε
- NFA with ε





Performance Concerns

- The NFA-to-DFA construction can introduce *exponentially* many states.
- Time/memory tradeoff:
  - Low-memory NFA has higher scan time.
  - High-memory DFA has lower scan time.

Real-World Scanning: **Python**

Python Blocks

- Scoping handled by whitespace:

```
if w == z:
    a = b
    c = d
  else:
    e = f
g = h
```

- What does that mean for the scanner?

Scanning Python

```
if w == z:
      a = b
      c = d
   else:
      e = f
g = h
```

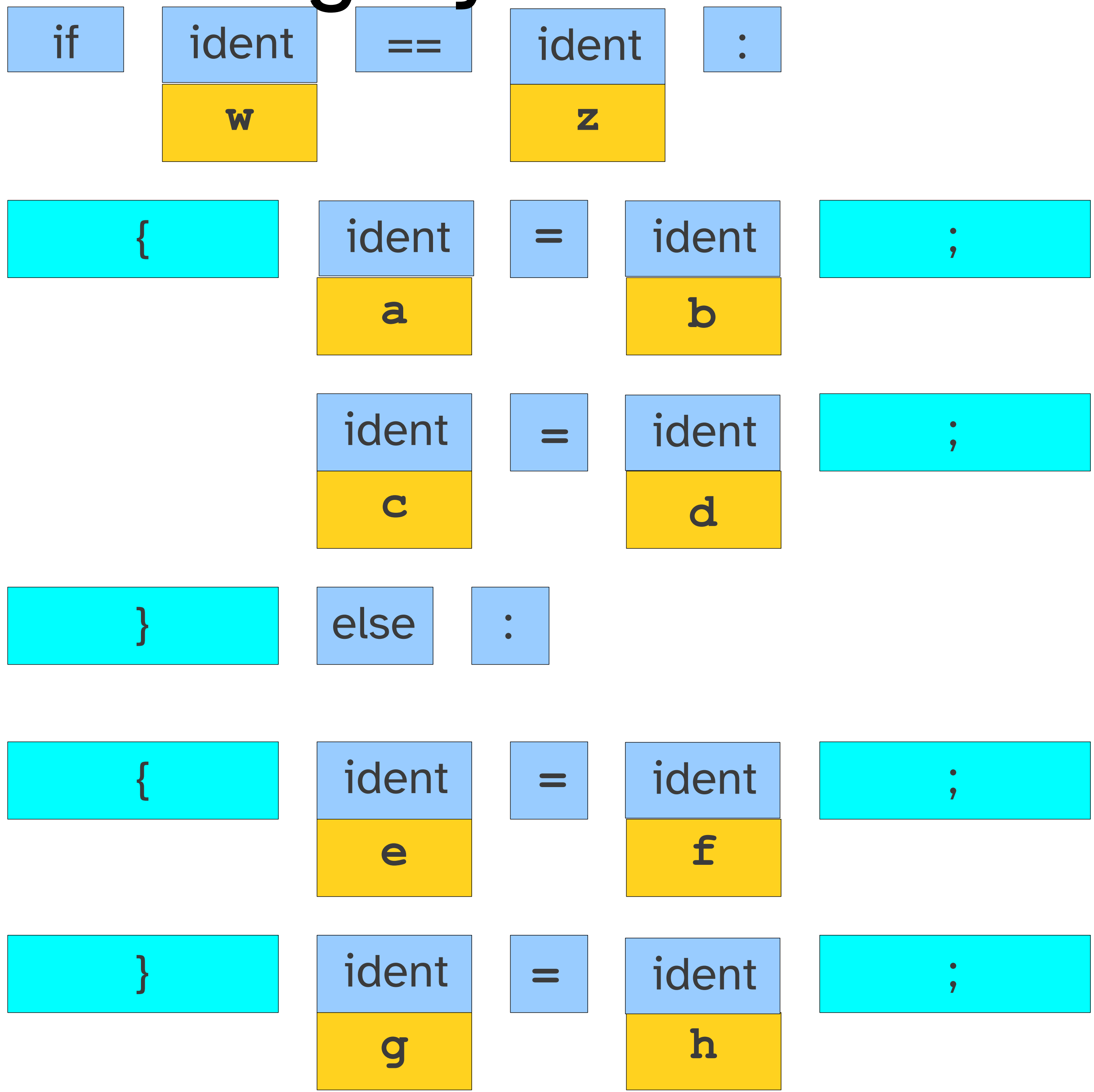
Scanning Python

```
if w == z:
    a = b
    c = d
else:
    e = f
g = h
```

if	ident (w)	==	ident (z)	:	NEWLINE	
INDENT	ident (a)	=	ident (b)	NEWLINE		
	ident (c)	=	ident (d)	NEWLINE		
DEDENT	else	:	NEWLINE			
INDENT	ident (e)	=	ident (f)	NEWLINE		
DEDENT	ident (g)	=	ident (h)	NEWLINE		

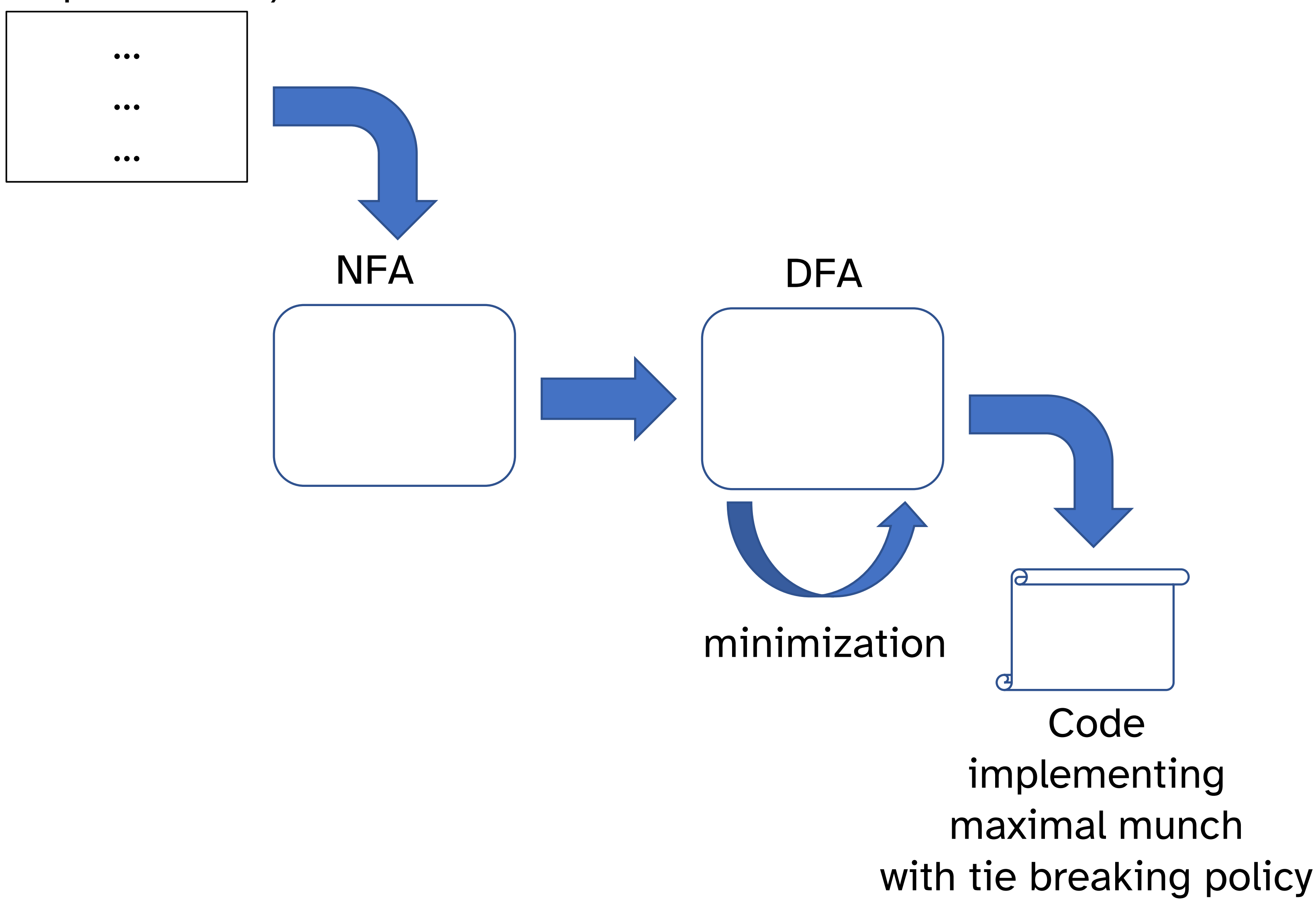
Scanning Python

```
if w == z:{
        a = b;
        c = d;
}  else: {
        e = f;
} g = h;
```



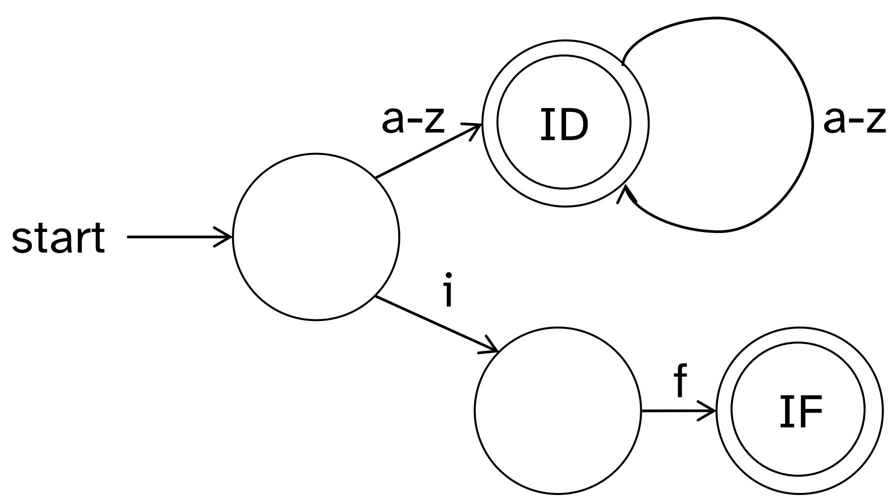
Putting all together to build a Scanner

List of
regular expressions
(one per lexeme)



Example

- T_ID = (a|b|…|z) (a|b|…|z)*
- T_IF = if

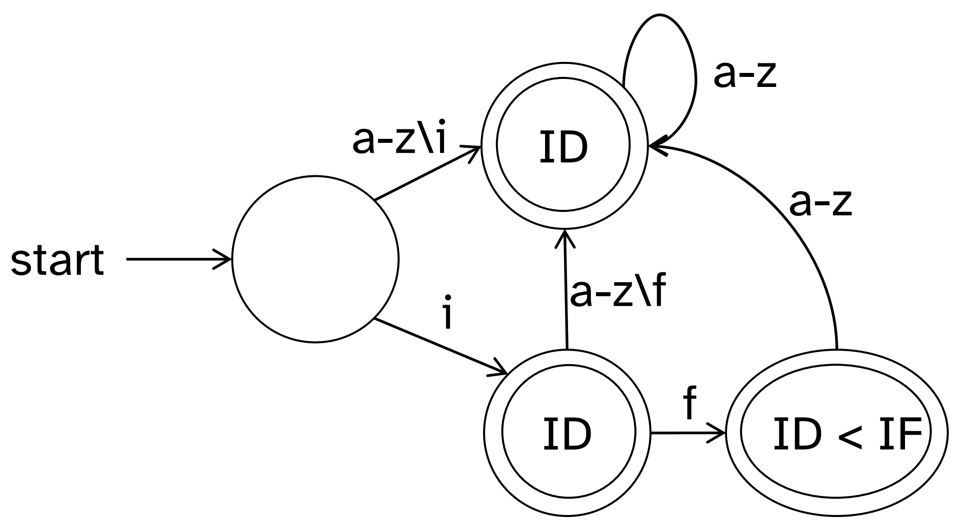


NFA

- Matches both tokens -> Solution: break tie using order of definitions
  - Output: ID(if)

Example

- T_IF = if
- T_ID = (a|b|…|z) (a|b|…|z)*



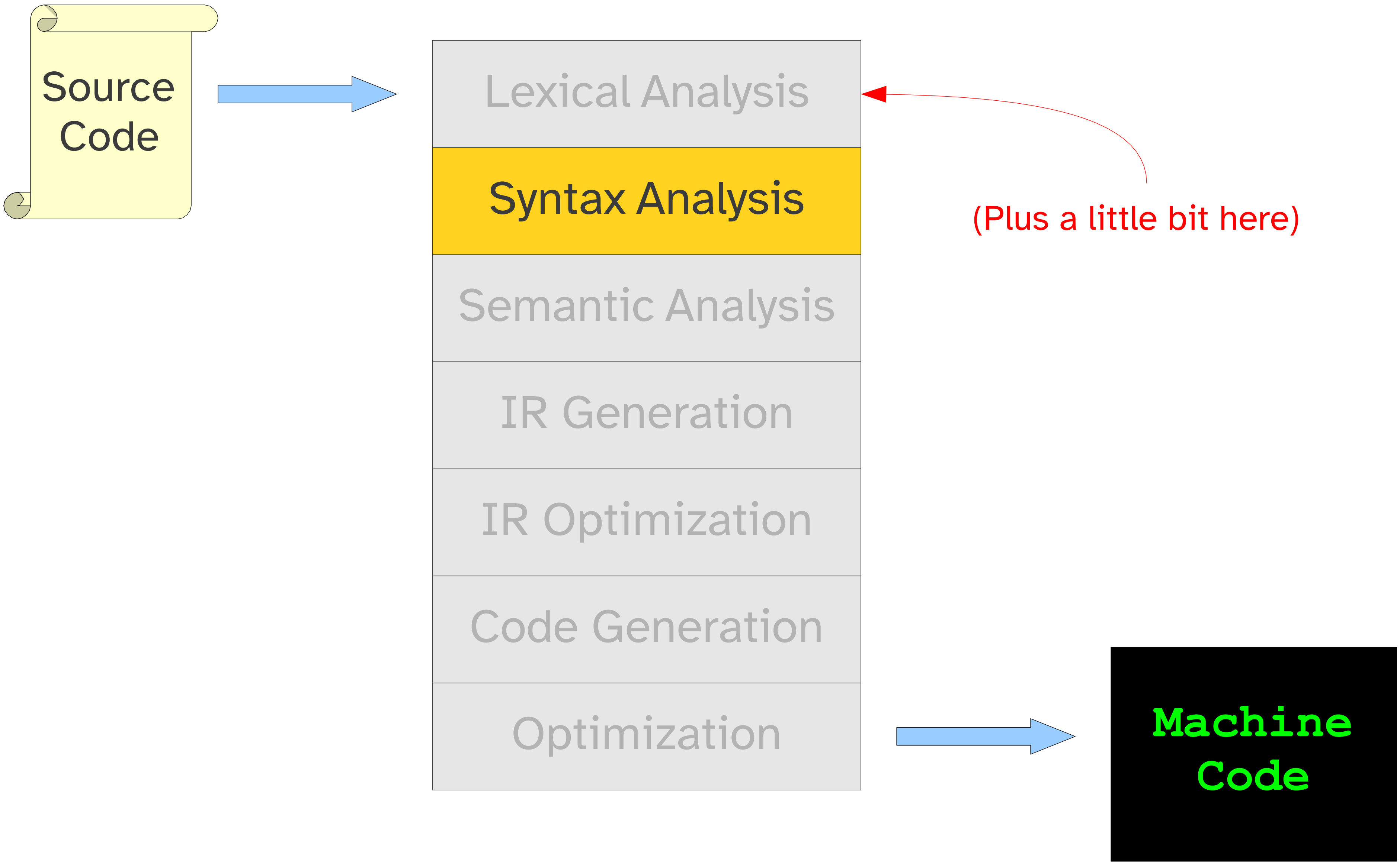
DFA

Output: IF

Summary

- Lexical analysis splits input text into **tokens** holding a **lexeme** and an **attribute**.
- Lexemes are sets of strings often defined with **regular expressions**.
- Regular expressions can be converted to **NFAs** and from there to **DFAs**.
- **Maximal-munch** using an automaton allows for fast scanning.
- Not all tokens come directly from the source code.

Next Time


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
(Plus a little bit here)
Machine Code