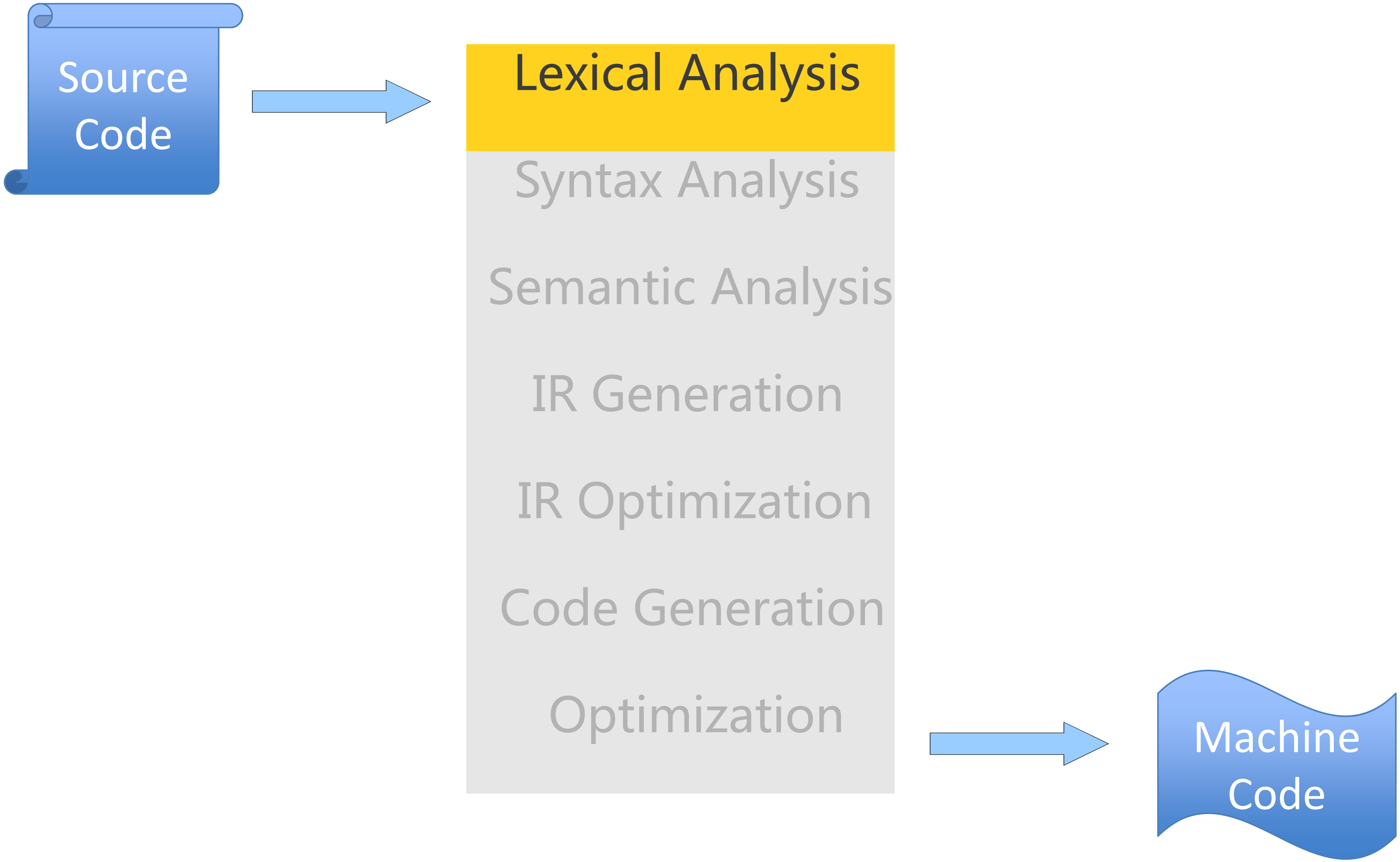
Compilers and Interpreters

Lexical Analysis

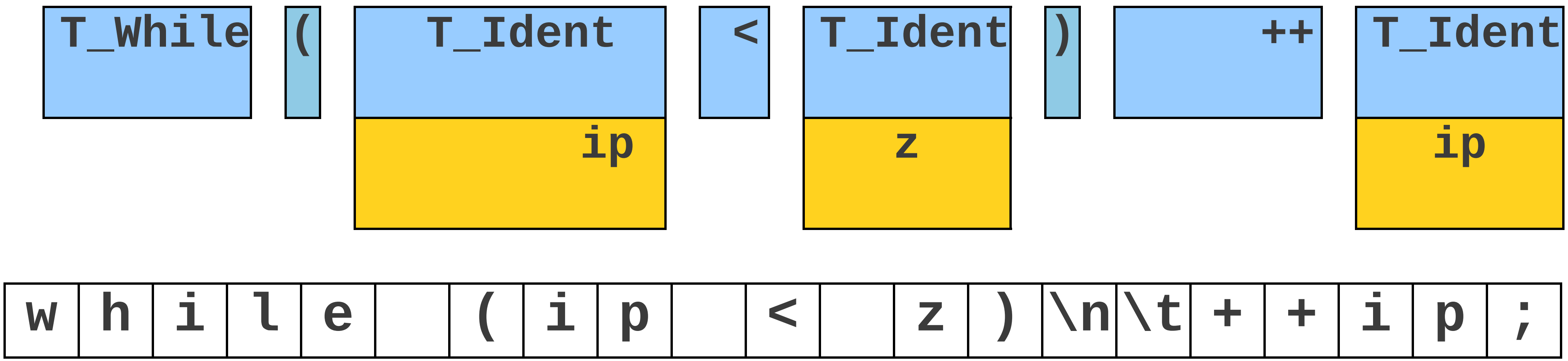
Where We Are


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

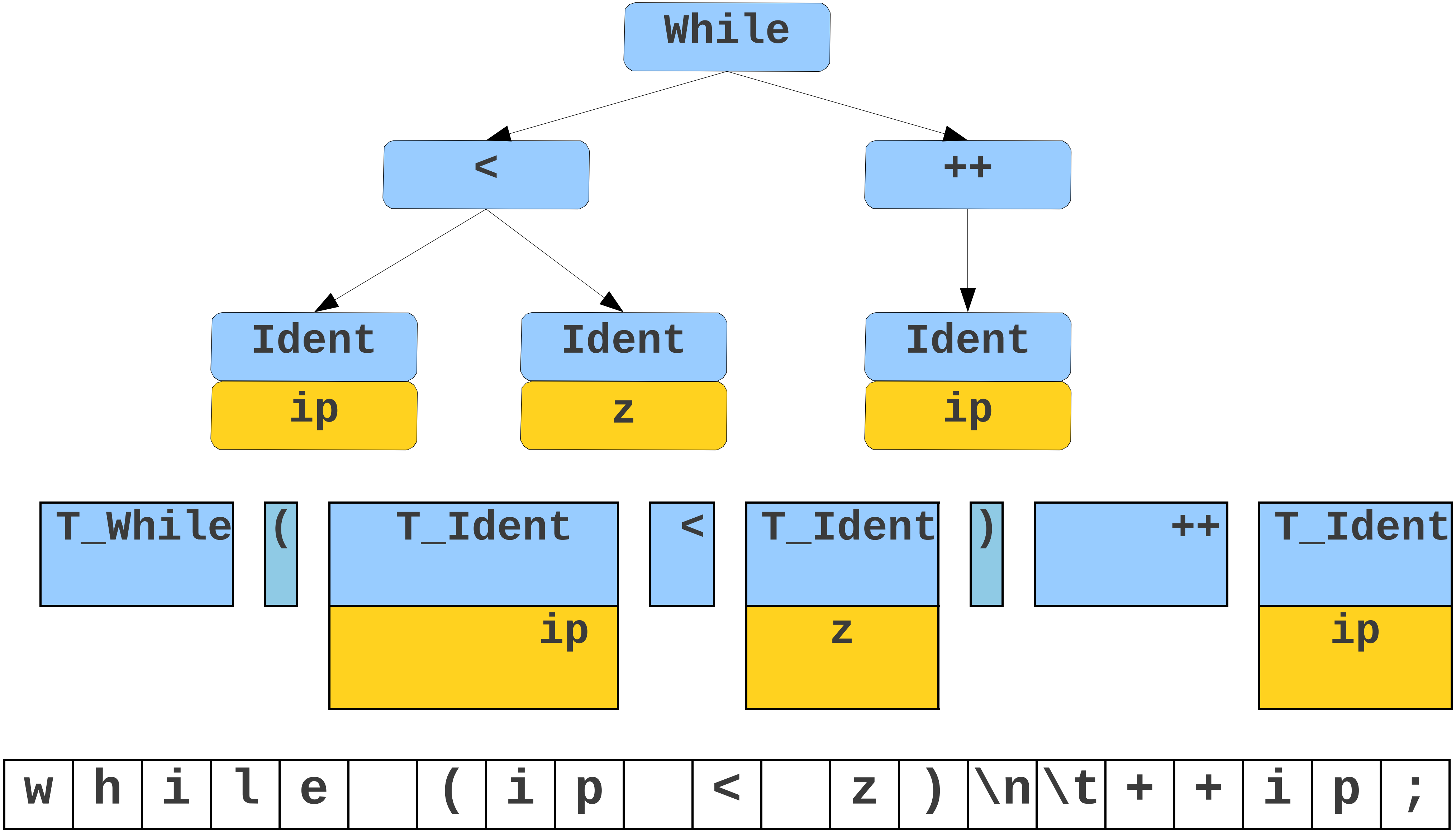
```
while (ip < z)
    ++ip;
```

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;
```

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	;

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	;

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	;

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**. You can 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

Scanning a Source File

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

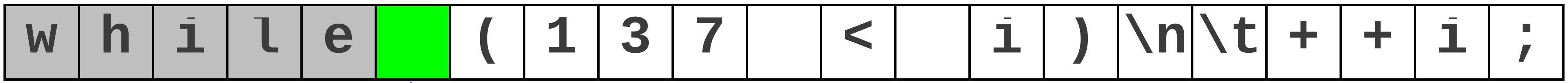
T_While

Scanning a Source File

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

T_While

Scanning a Source File



`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

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	;

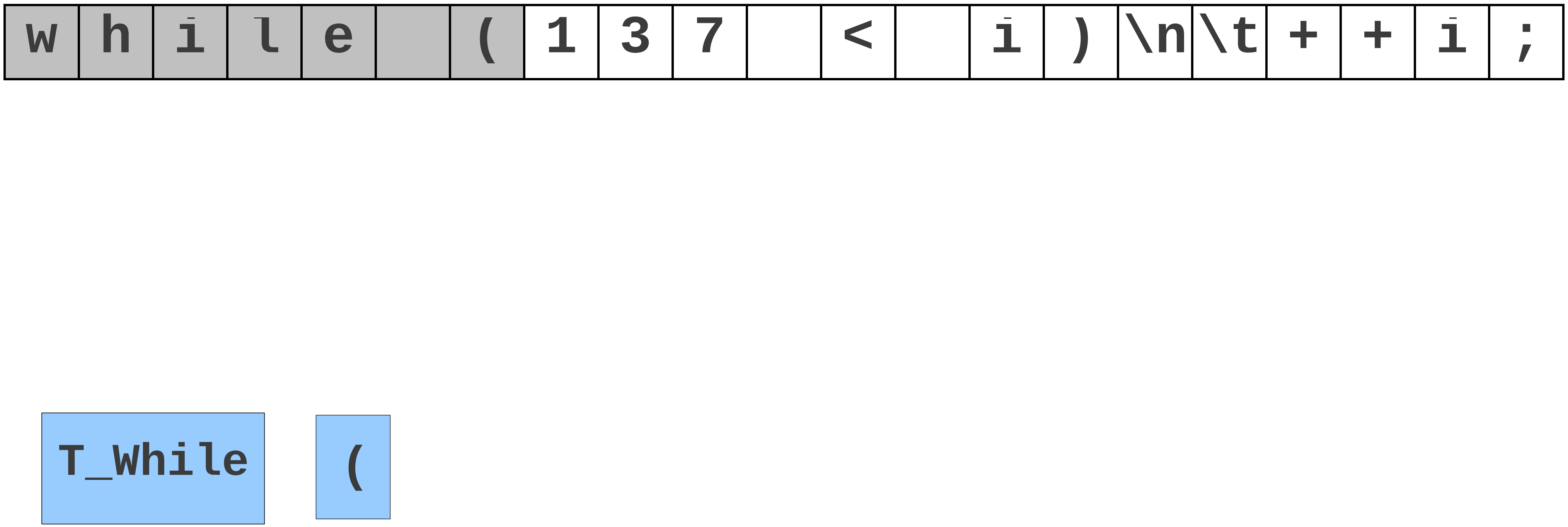
T_While

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

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	;

T_While

(

Scanning a Source File

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

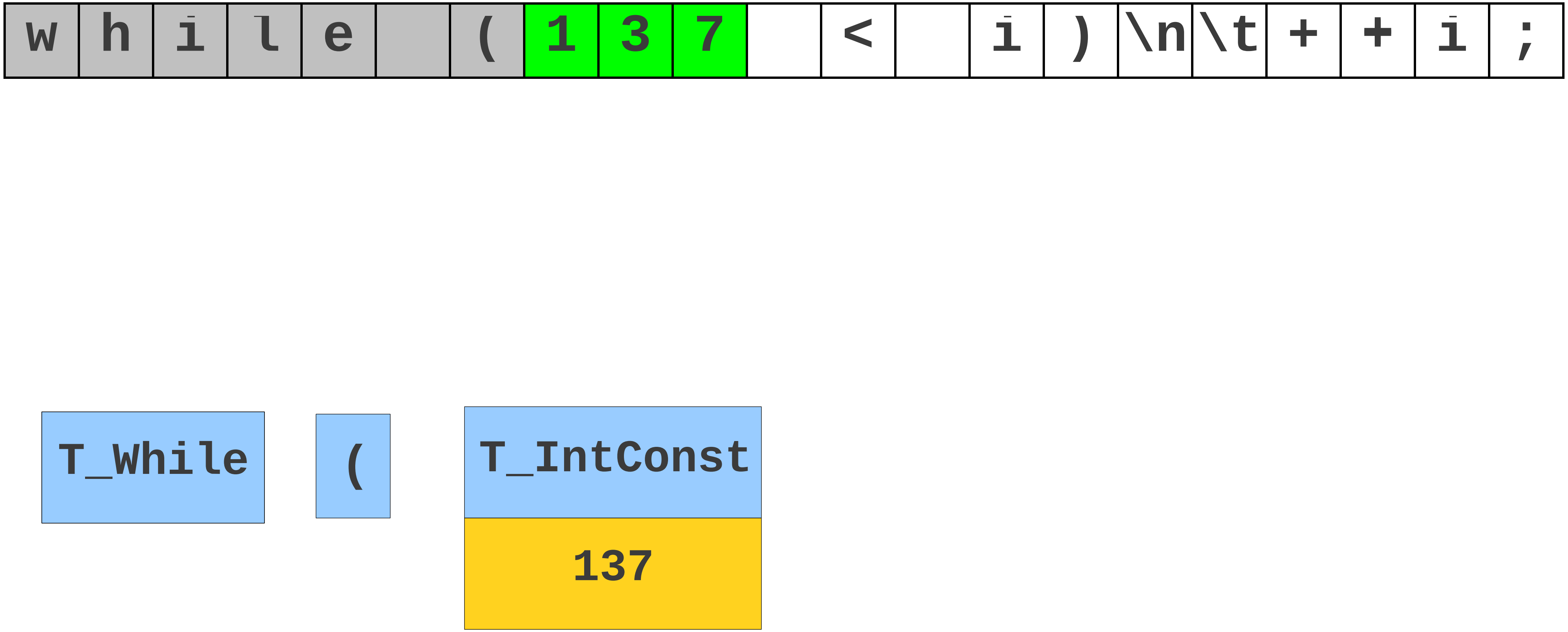
T_While	(

Scanning a Source File

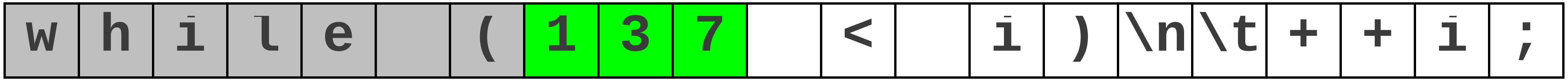
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

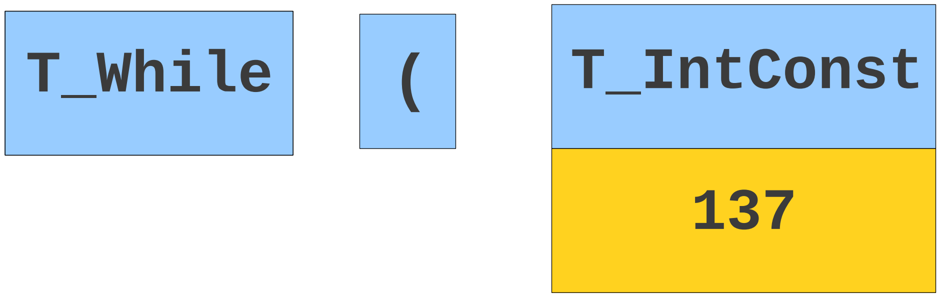
`T_While` `(`

Scanning a Source File



Scanning a Source File





Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: "137," "int," etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

Choosing Tokens

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
cout     << k << endl;
}
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k) {
        cout << k << endl;
}
```

for	{
int	}
<<	;
=	<
(	[
)	]
++	

Identifier
IntegerConstant

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
DO 5 I = 1.25
```

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
DO 5 I = 1.25
```

Can be difficult to tell when to partition input.

Scanning is Hard

- C++: Nested template declarations

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

Scanning is Hard

- C++: Nested template declarations

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

Scanning is Hard

- C++: Nested template declarations

```
( vector < ( vector < ( int >> myVector ) )
```

Can be difficult to determine where to split.

Scanning is Hard

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

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

Can be difficult to determine how to label lexeme.

Challenges in Scanning

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

Challenges in Scanning

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme

- Left-to-right scan
  - Lookahead sometimes required

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`** probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes
  - All variable names, all possible numbers, all possible strings, etc.

Sets of Lexemes

- Idea: Associate a set of lexemes with each token.
- We might associate the "number" token with the set { **0**, **1**, **2**, …, **10**, **11**, **12**, … }
- We might associate the "string" token with the set { **""**, **"a"**, **"b"**, **"c"**, … }
- We might associate the token for the keyword **while** with the set { **while** }.

How to 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.
- Over the course of this class, 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.
- Used as the basis for numerous software systems, e.g. **`flex, antlr.`**

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

1. If R_1 and R_2 are regular expressions, **R_1R_2** is a regular expression represents the **concatenation** of the languages of R_1 and R_2.
2. If R_1 and R_2 are regular expressions, **$R_1 \mid R_2$** is a regular expression representing the **union** of $R_1$and R_2 .
3. If R is a regular expression, **R*** is a regular expression for the **Kleene closure** of R.
4. If R is a regular expression, **(R)** is a regular expression with the same meaning as R.

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R^*$$

$$R_1R_2$$

$$R_1 | R_2$$

- So **ab*c|d** is parsed as **((a(b*))c)|d**

Algebraic Laws for Regular Expression

<table>
<tr><th>LAW</th><th>DESCRIPTION</th></tr>
<tr><td>r|s = s|r</td><td>| is commutative</td></tr>
<tr><td>r|(s|t) = (r|s)|t</td><td>| is associate</td></tr>
<tr><td>r(st) = (rs)t</td><td>Concatenation is associate</td></tr>
<tr><td>r(s|t) = rs|rt; (s|t)r = sr|tr</td><td>Concatenation distributes over |</td></tr>
<tr><td>εr = rε = r</td><td>ε is the identity for concatenation</td></tr>
<tr><td>r* = (r|ε)*</td><td>ε is guaranteed in a closure</td></tr>
<tr><td>r** = r*</td><td>ε is idempotent</td></tr>
</table>

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
11111011110011111

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings containing `00` as a substring:

(0 | 1)*00(0 | 1)*

11011100101
0000
11111011110011111

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are **`0`** and **`1`**.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1)(0|1)(0|1)(0|1)

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

- Suppose the only characters are **0** and **1**.
- Here is a regular expression for strings that contain at most one zero:

1*(0 | ε)1*   **1*0?1***

11110111
111111
0111
0

Applied Regular Expressions

- Suppose our alphabet is **a**, **@**, and **.**, where **a** represents "some letter."
- A regular expression for email addresses is

aa* (.aa*)* @aa*.aa*(.aa*)*

$$a+\ (.\ a^{+})^{*}\ @\ a^{+}\ .a^{+}\ (.a^{+})^{*}$$

$$a+\ (.\ a^{+})^{*}\ @\ a^{+}\ .\ (.a^{+})^{*}$$

thecompiler@126.com
lli@whu.edu.cn

Applied Regular Expressions

- Suppose that our alphabet is all ASCII characters.
- A regular expression for even numbers is

(+|-)?(0|1|2|3|4|5|6|7|8|9)*(0|2|4|6|8)

(+|-)?[0123456789]*[02468]

(+|-)?[0-9]*[02468]

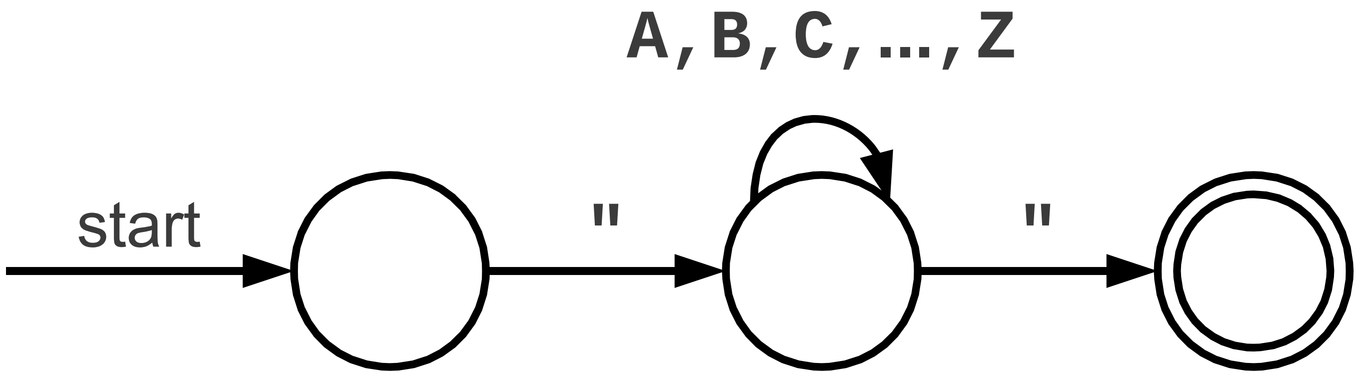
42
+1370
-3248
-9999912

Matching Regular Expressions

Implementing Regular Expressions

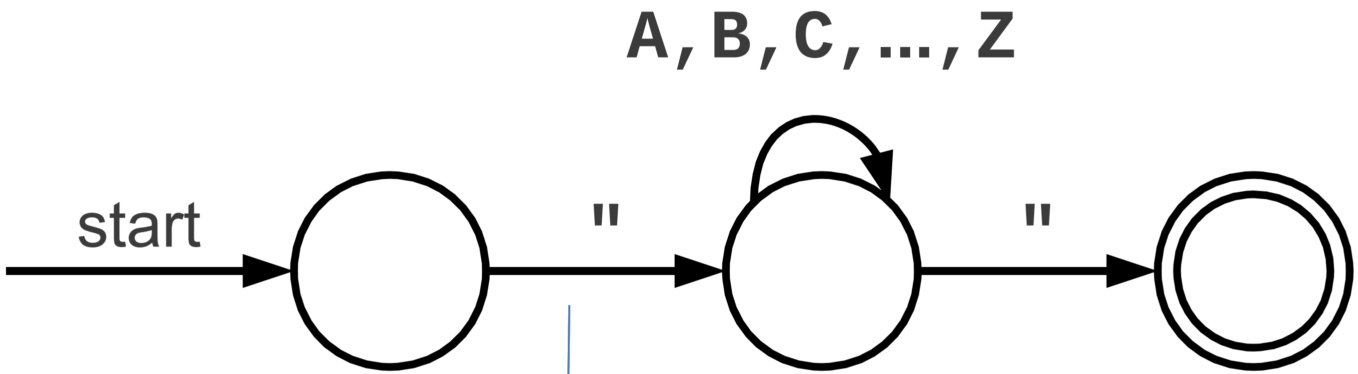
- Regular expressions can be implemented using **finite automata**.
- There are two main kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata), which we'll see in a second, and
  - **DFA**s (**deterministic** finite automata), which we'll see later.

A Simple Automaton



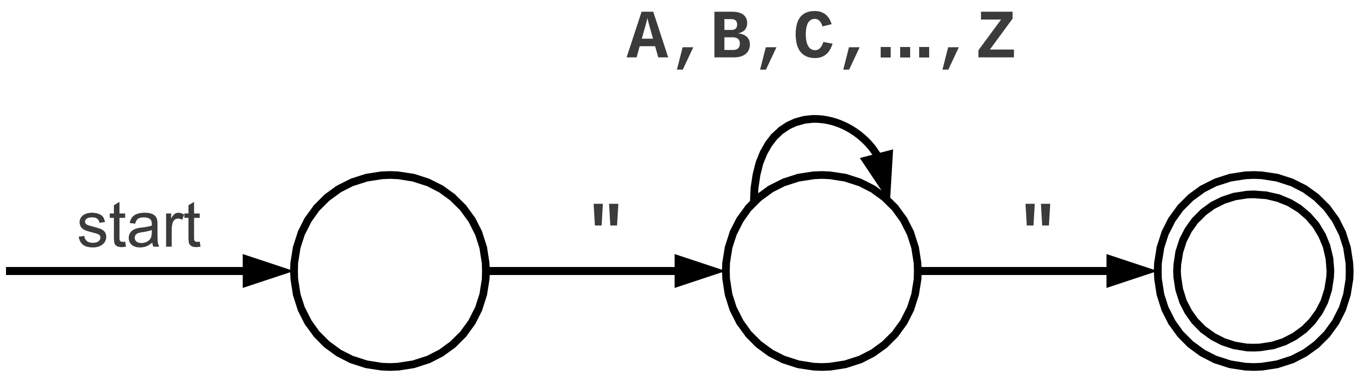
Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

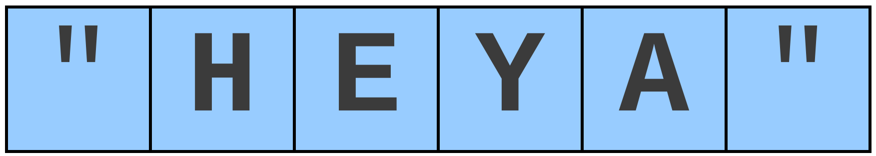
A Simple Automaton



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

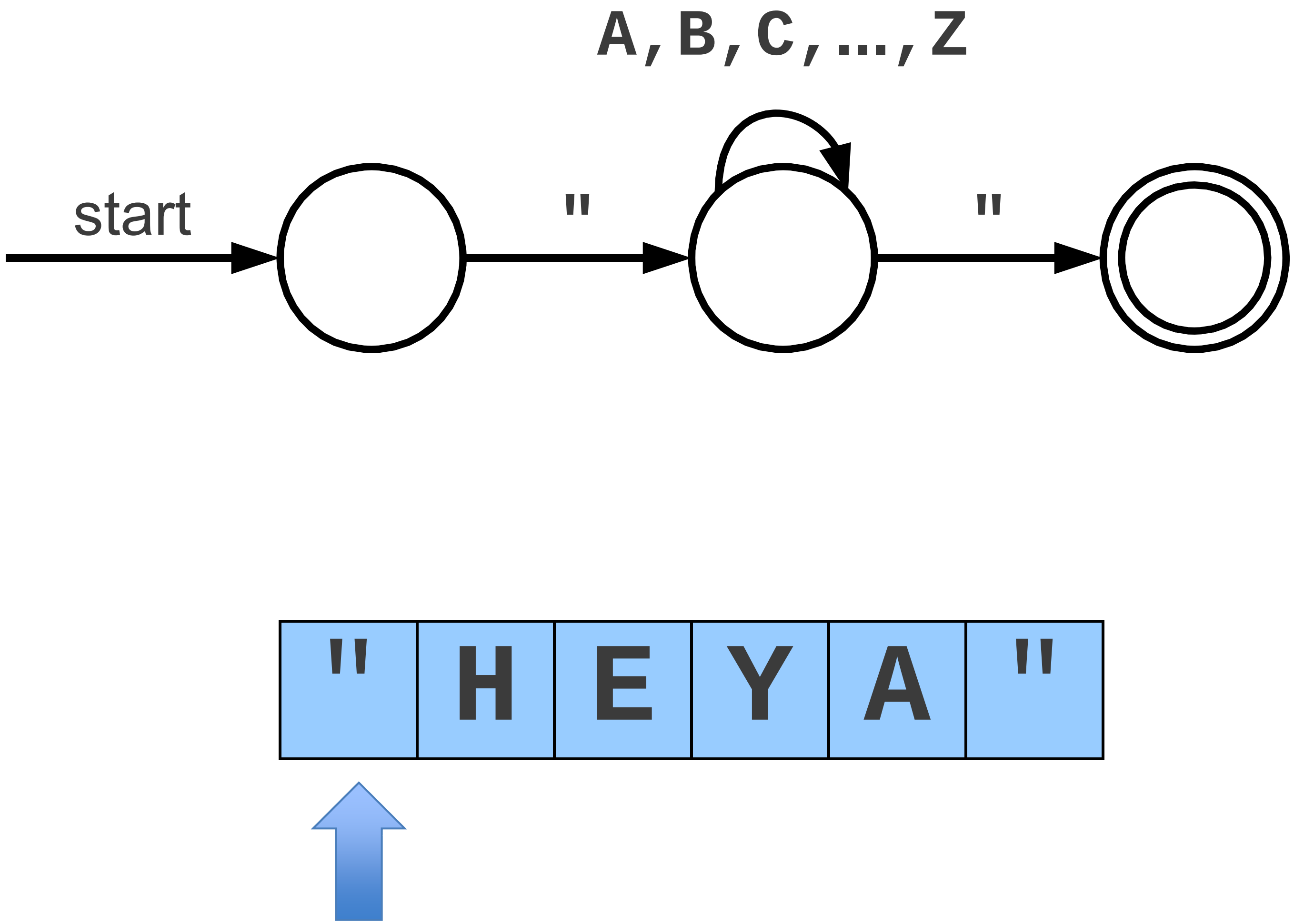
A Simple Automaton



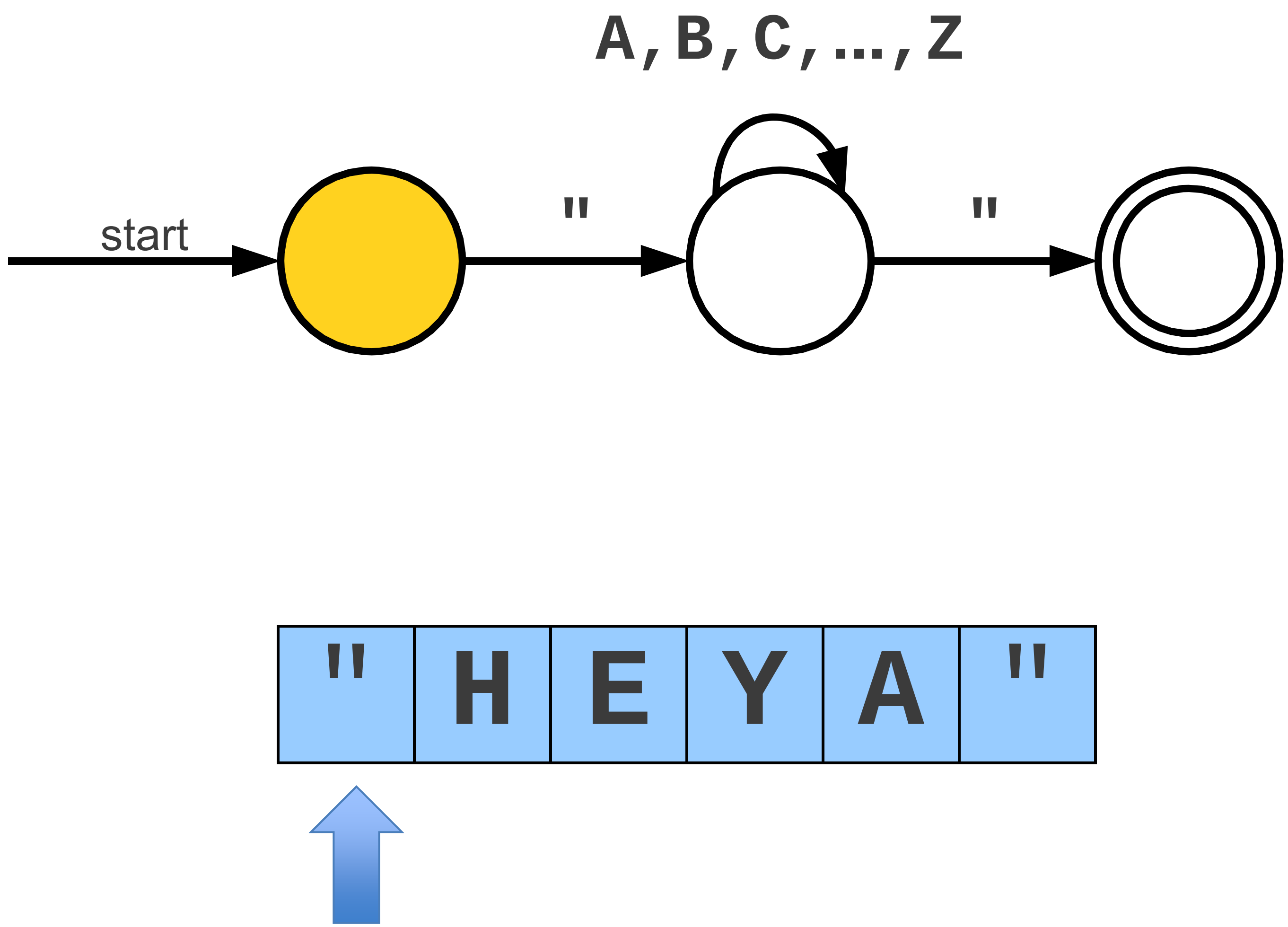


The automaton takes a string as input and decide whether to accept or reject the string.

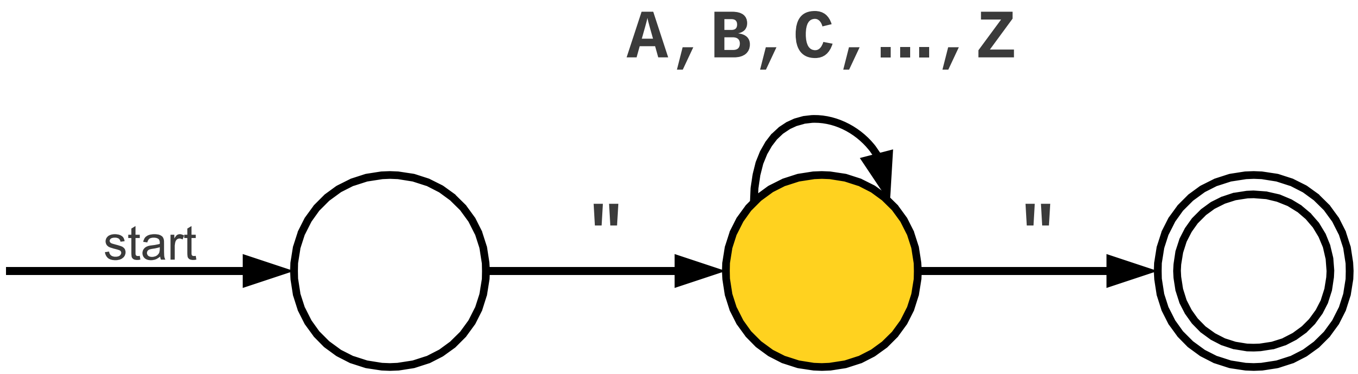
A Simple Automaton

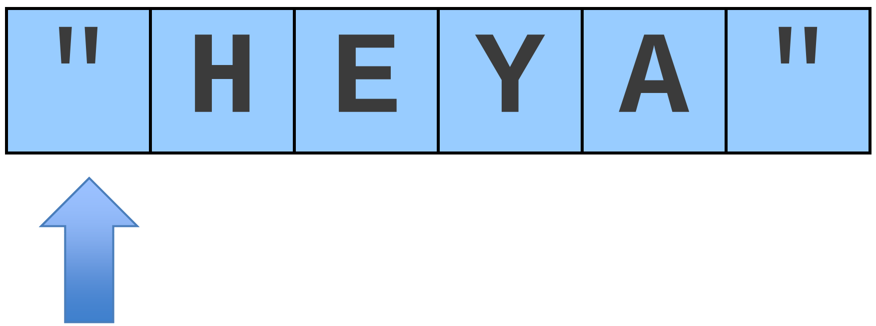


A Simple Automaton

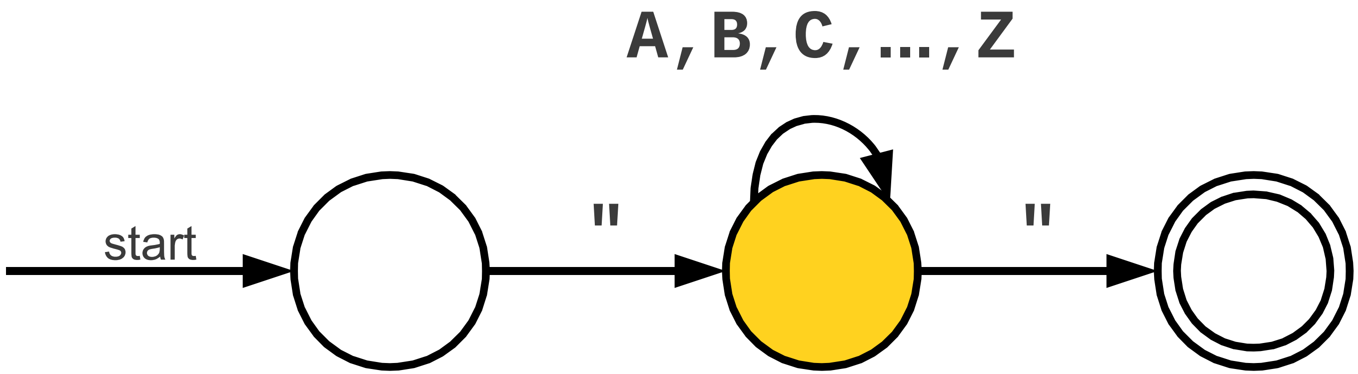


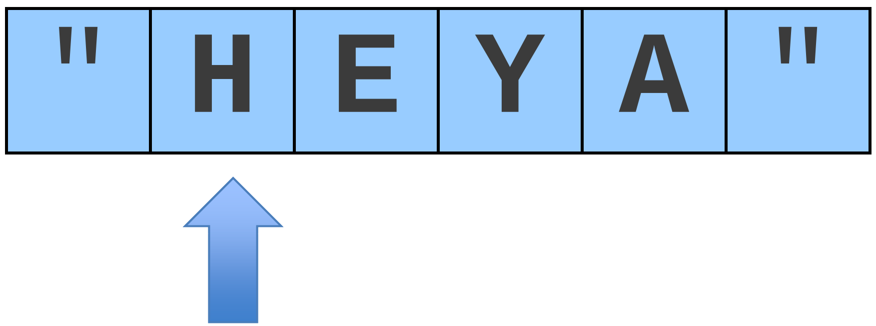
A Simple Automaton



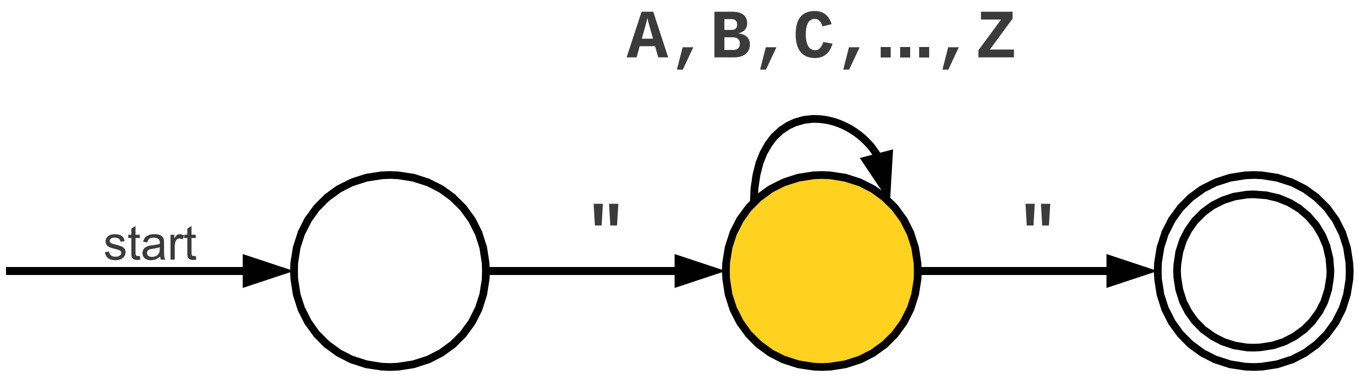


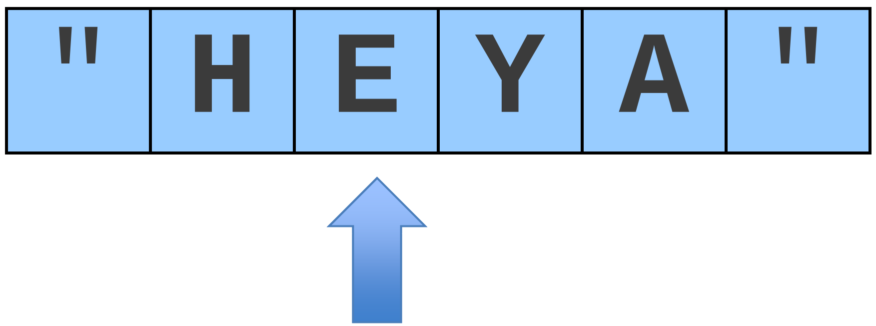
A Simple Automaton



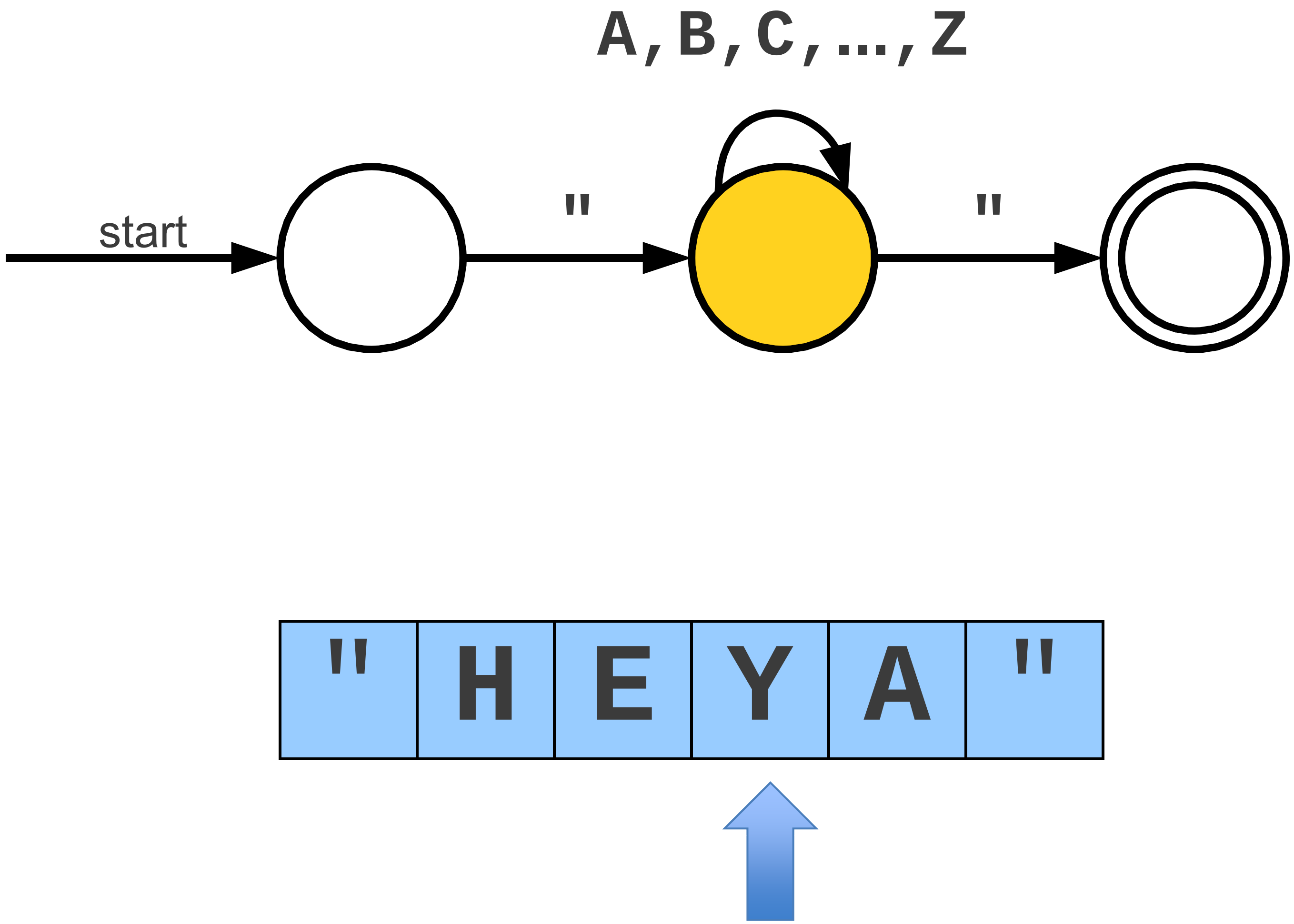


A Simple Automaton

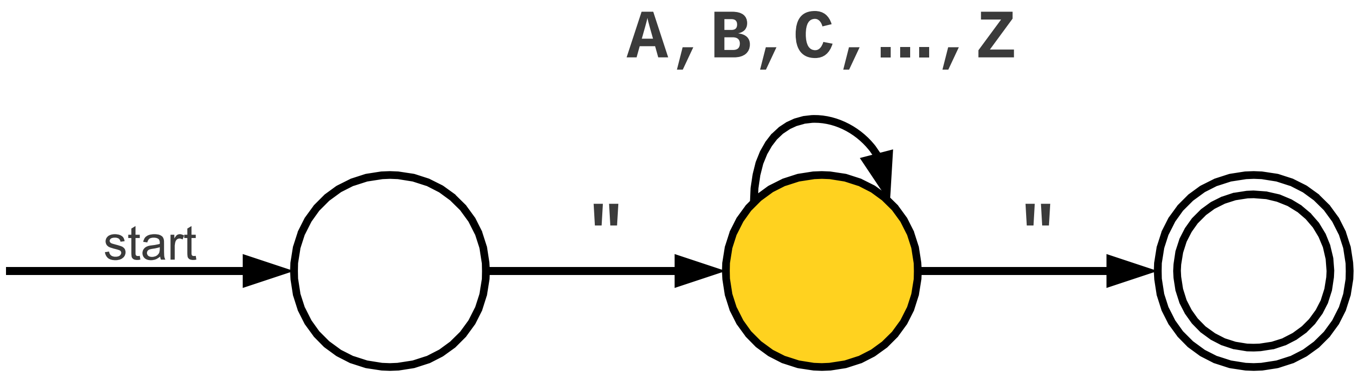


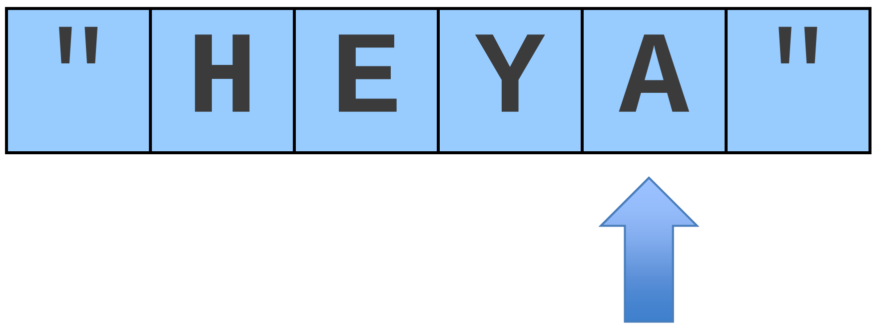


A Simple Automaton

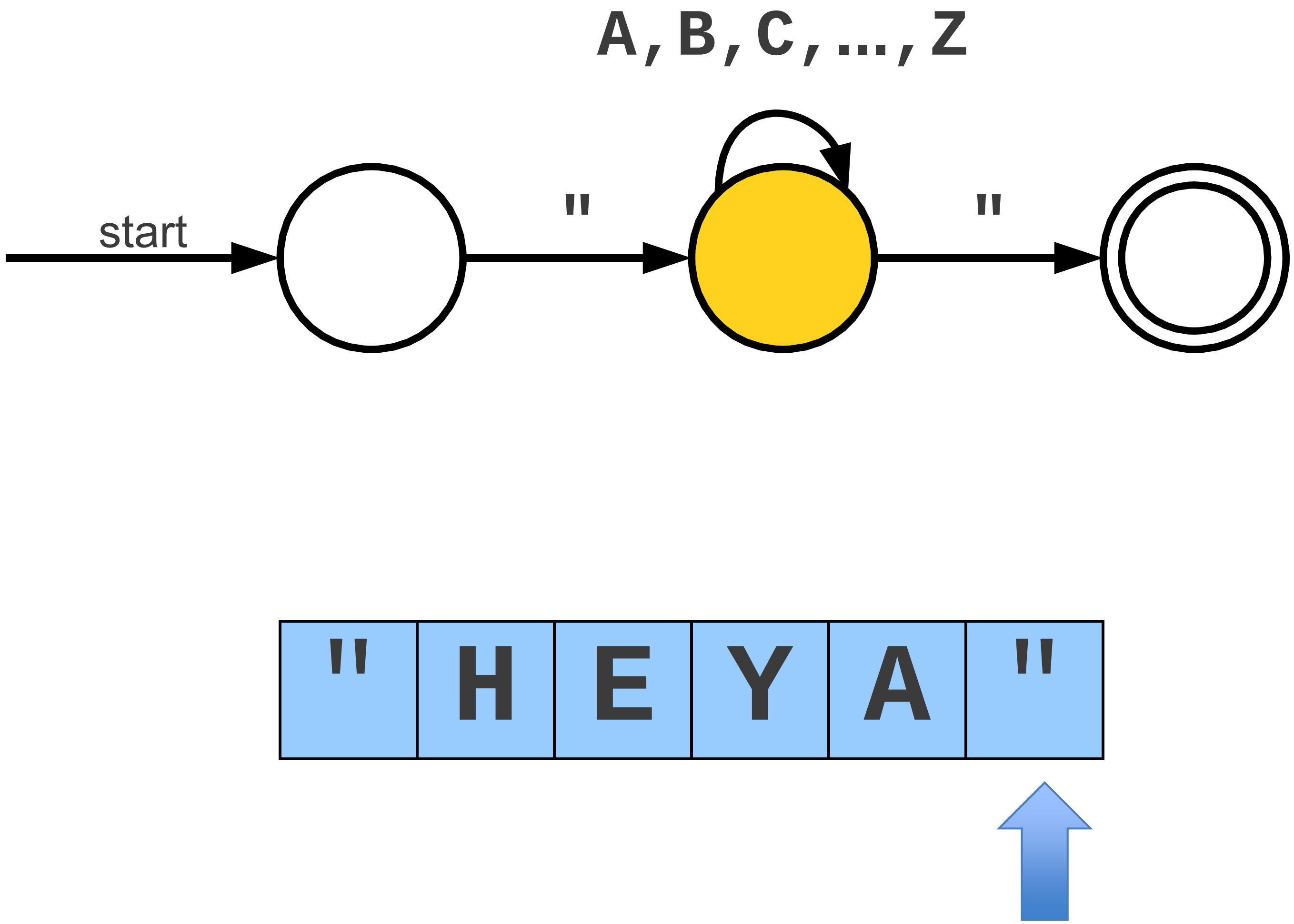


A Simple Automaton

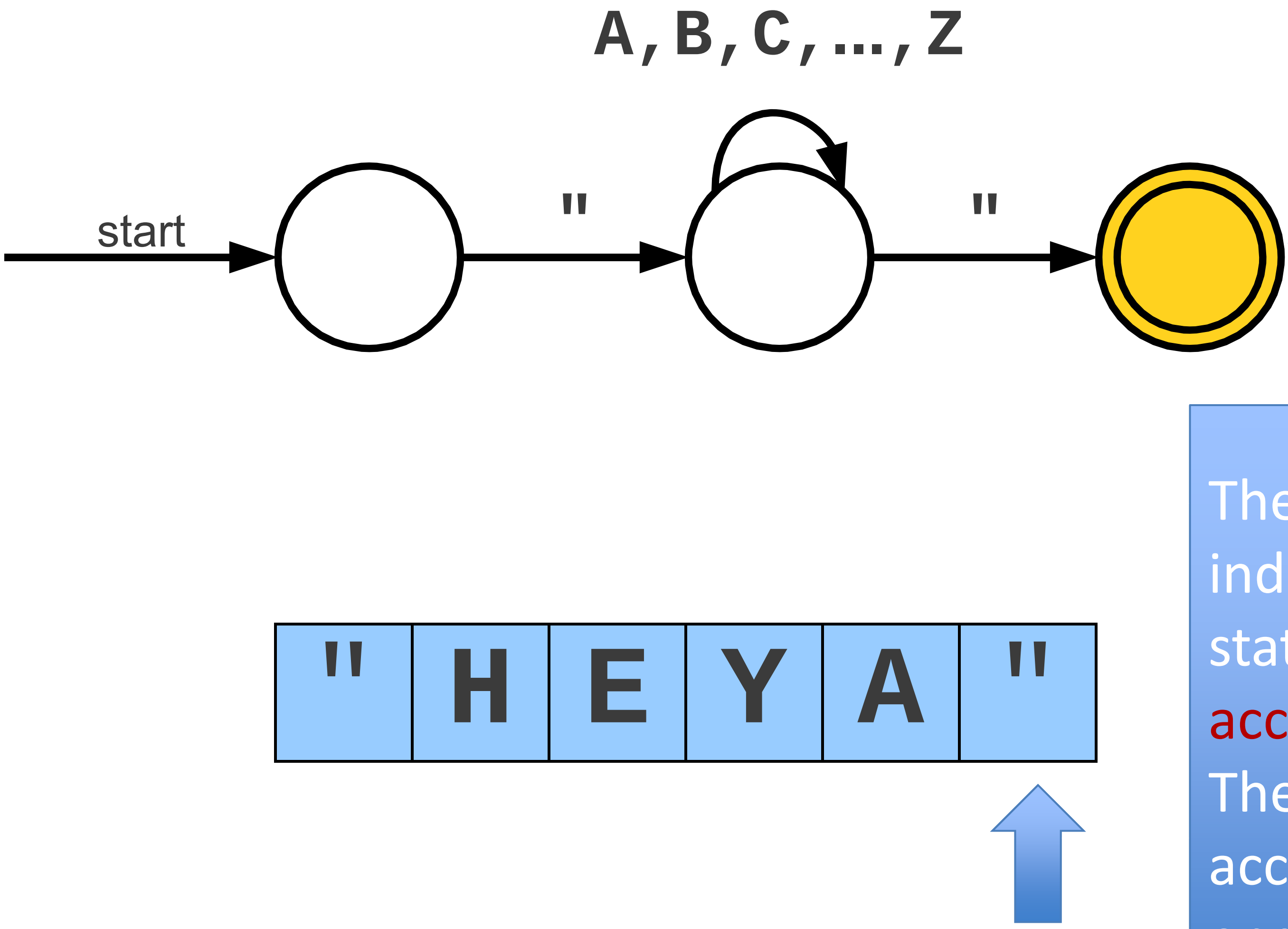




A Simple Automaton

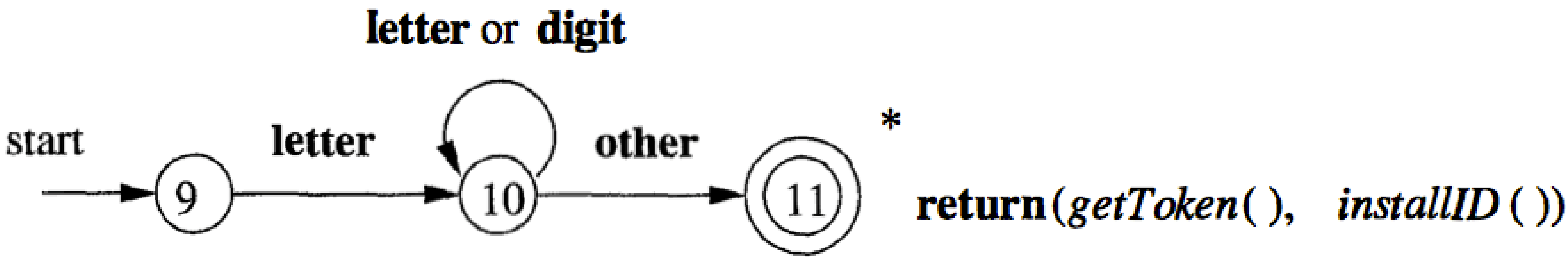


A Simple Automaton



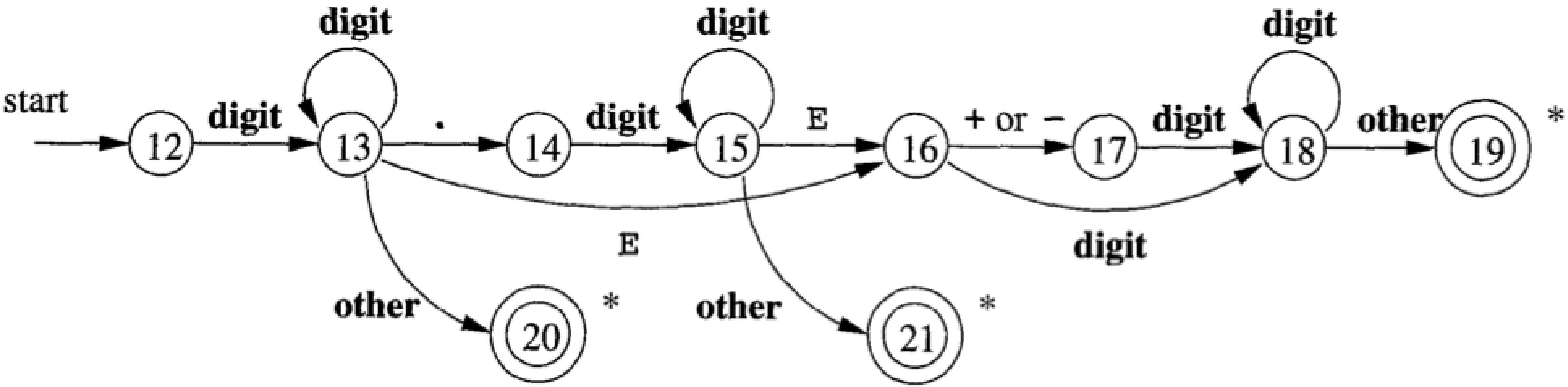
The double circle indicates that this state is an accepting state. The automaton accepts string if it ends in an accepting state.

A More Complex Automaton



h	i	1	2	3

A More Complex Automaton



1	2	.	3	7	5

Finite Automata

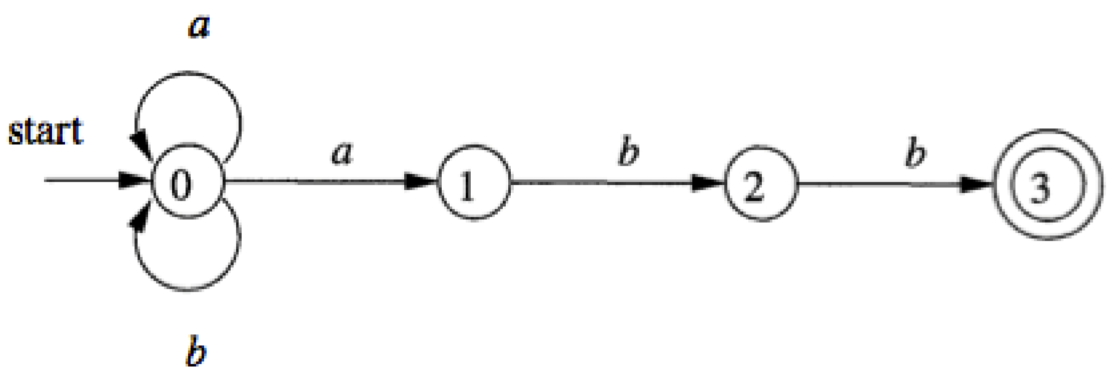
- Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
- Finite automata come in two flavors:
  - Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and E, the empty string, is a possible label.
  - Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

NFA

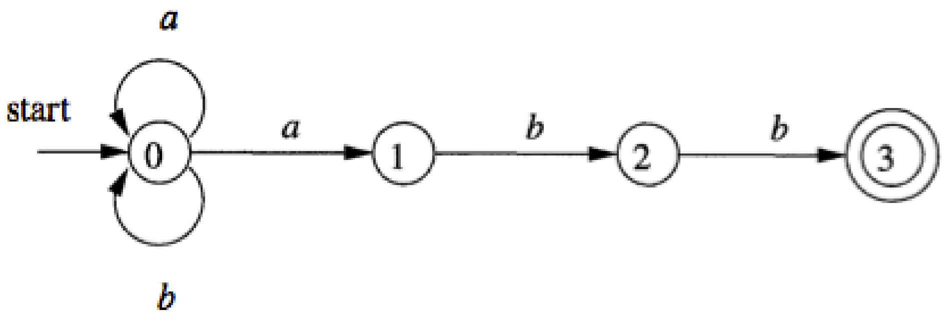
- A nondeterministic finite automaton (NFA) consists of:
  1. A finite set of states S.
  2. A set of input symbols Σ, the input alphabet. We assume that ε, which stands for the empty string, is never a member of Σ .
  3. A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\varepsilon\}$ a set of next states.
  4. A state s_0 from S that is distinguished as the start state (or initial state) .
  5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

NFA (cont.)

- The same symbol can label edges from one state to several different states, and
- An edge may be labeled by ε, the empty string, instead of, or in addition to, symbols from the input alphabet.



NFA (cont.)



Transition Table

STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Acceptance of input strings

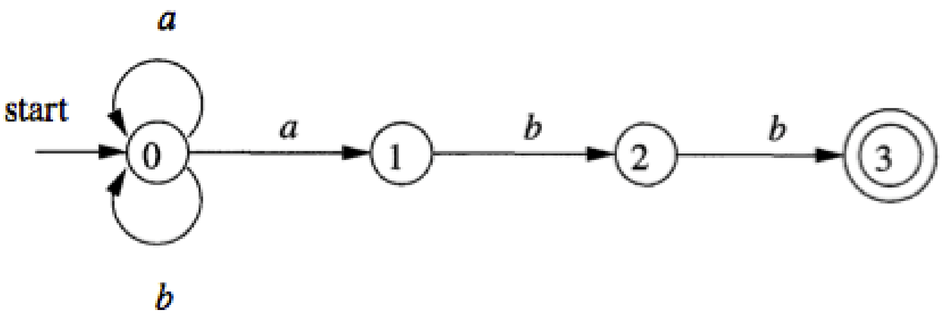
$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

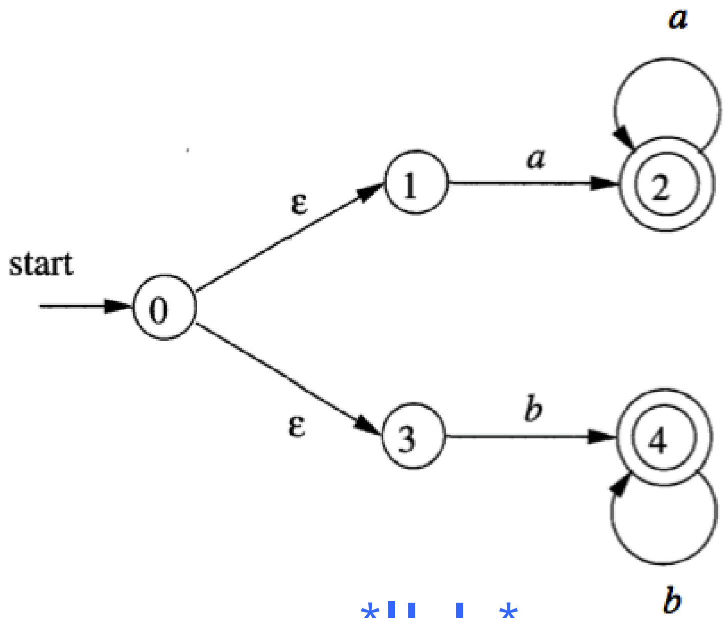
Complexity: O(mn^2) for strings of length m and automata with n states.

NFA (cont.)

- The language defined (or accepted) by an NFA is the set of strings labeling some path from the start to an accepting state.



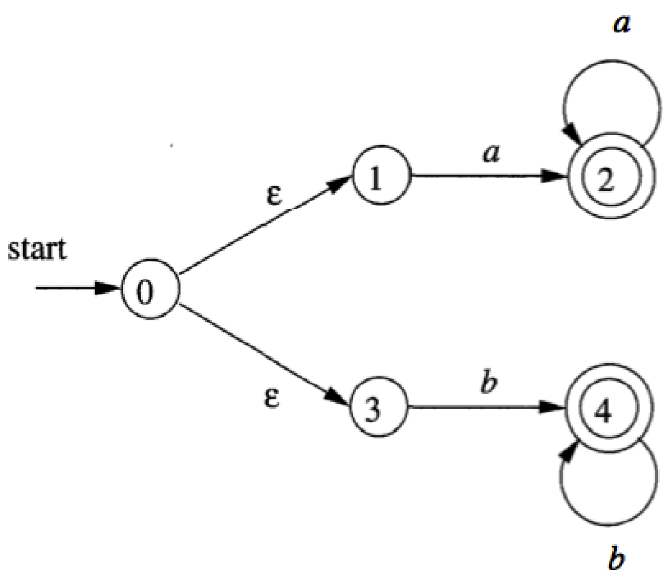
(a|b)*abb

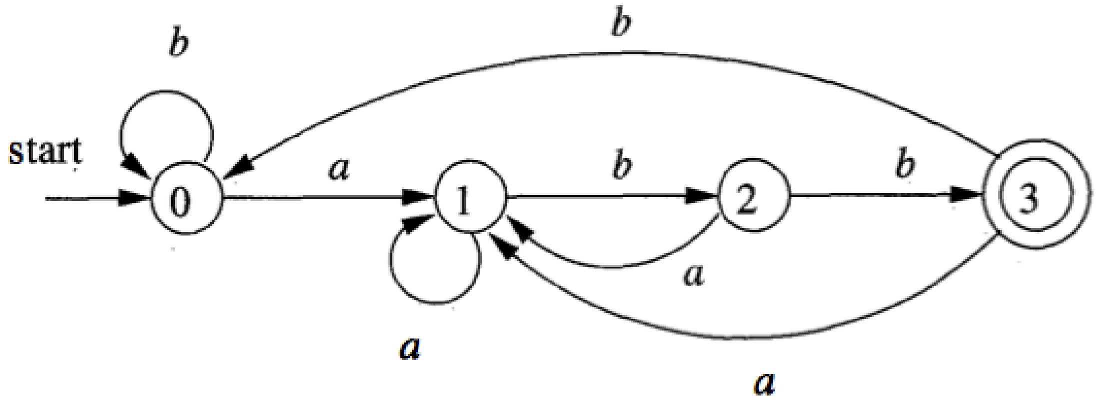


aa*|bb*

DFA

- A deterministic finite automaton (DFA) is a special case of an NFA where:
  1. There are no moves on input ε, and
  2. For each state *s* and input symbol *a*, there is **exactly one** edge out of *s* labeled *a*.



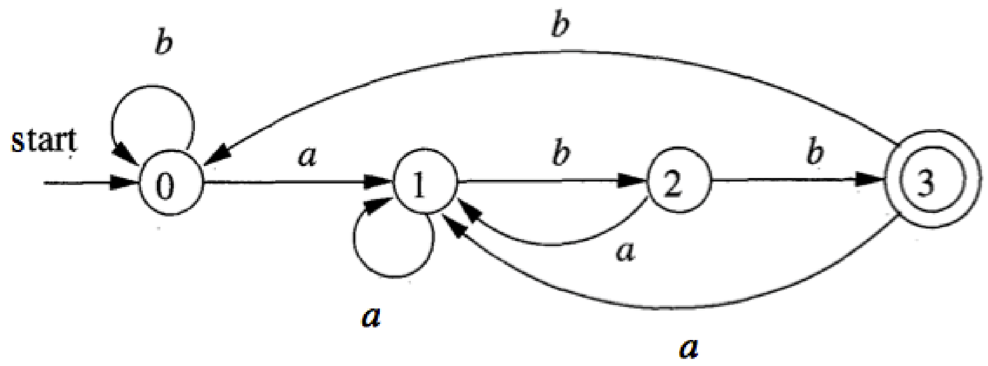


aa*|bb*

Simulating DFA

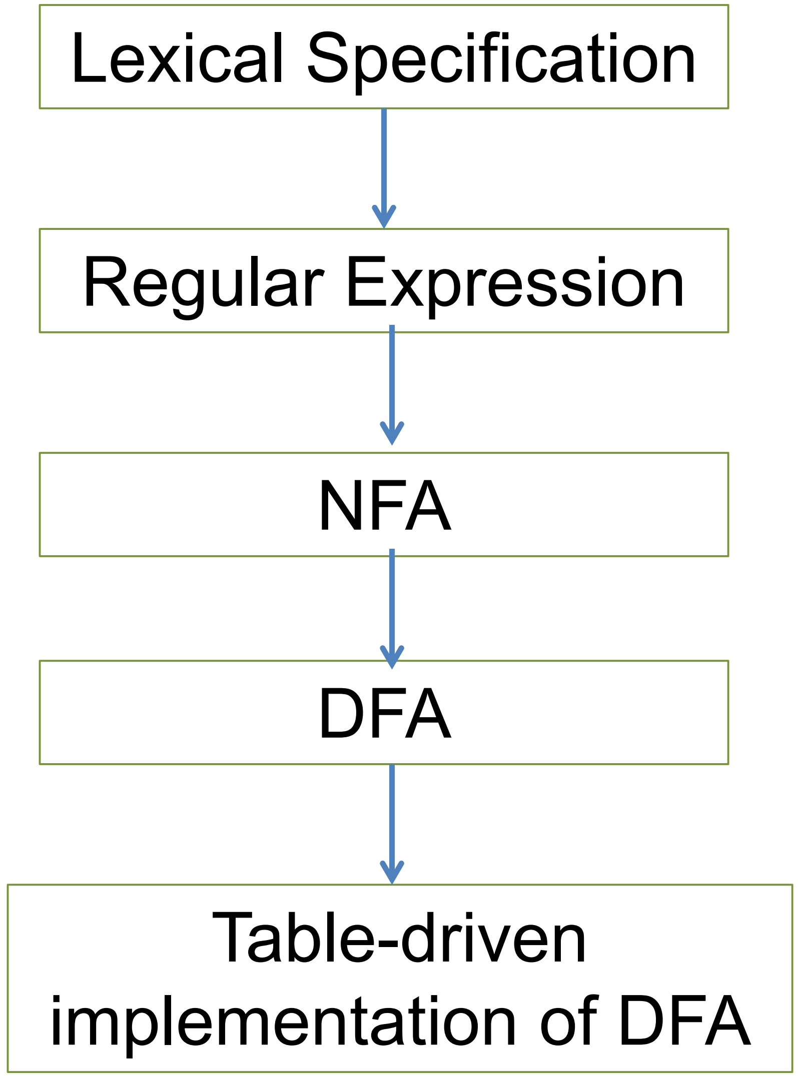
- **INPUT**: An input string *x* terminated by an end-of-file character eof. A DFA D with start state s_0, accepting states F, and transition function move.
- **OUTPUT**: Answer "yes" if D accepts *x*, "no" otherwise.

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```



aa*|bb*

Get a lexical analyzer



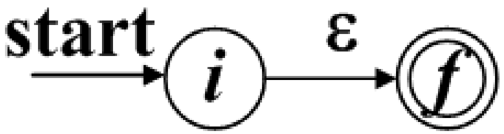
From Regular Expressions to NFA

- The **McNaughton-Yamada-Thompson** algorithm converts a regular expression to an NFA.
  - **INPUT**: A regular expression *r* over alphabet Σ.
  - **OUTPUT**: An NFA *N* accepting L(*r*).

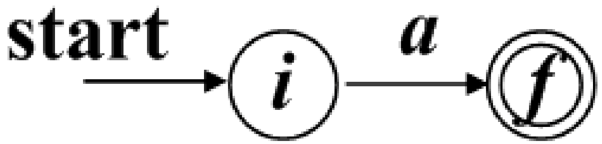
- Begin by parsing *r* into its constituent subexpressions. The rules for constructing an NFA consist of basis rules for handling subexpressiolls with no operators, and inductive rules for constructing larger NFA's from the NFA's
  - for the immediate subexpressions of a given expression.

From Regular Expressions to NFA(cont.)

- **Basic Rules**:



$r = \varepsilon$

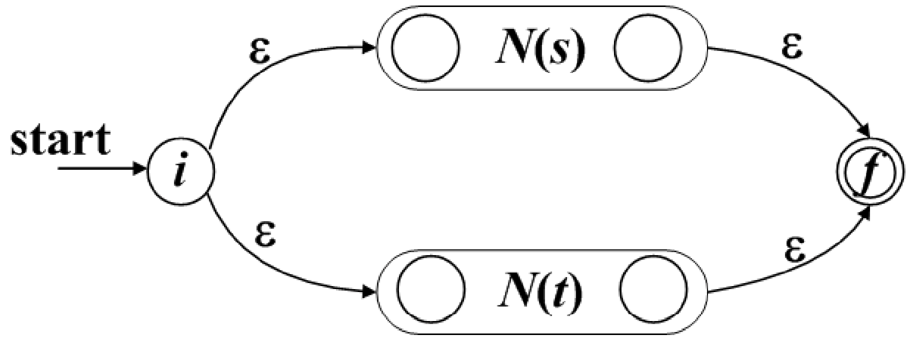


$r = a,\ a \in \Sigma$

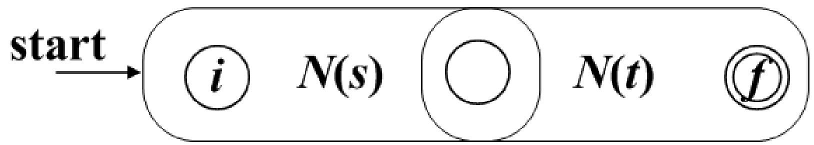
From Regular Expressions to NFA(cont.)

- **Inductive Rules**: Suppose N(s) and N(t) are NFA's for regular expressions *s* and *t*, respectively

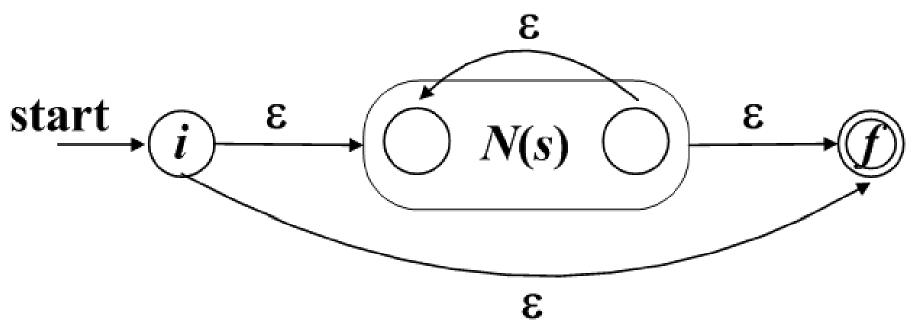
$r = s|t$ $\quad N(r)$:



$r = st$ $\quad N(r)$:



$r = s^*$ $\quad N(r)$:



From Regular Expressions to NFA(cont.)

- Any regular expression r of length $|r|$ can be converted into an NFA with $O(|r|)$ states.
- Can determine whether a string x of length $|x|$ matches a regular expression r of length $|r|$ in time $O(|r| \times |x|)$.

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_]*
```

Lexing Ambiguities

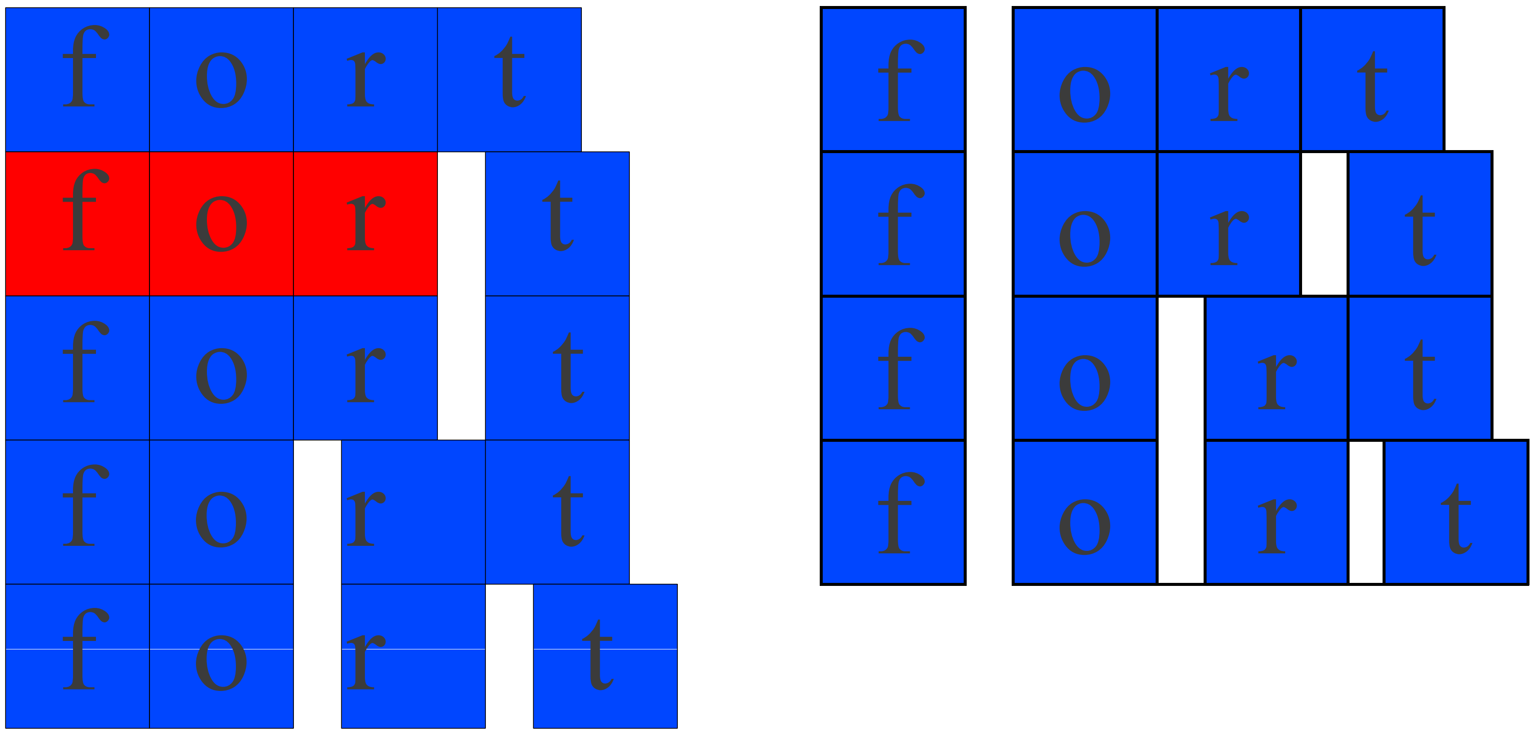
```
T_For           for
T_Identifier    [A-Za-z_][A-Za-z0-9_]*
```

f o r t

Lexing Ambiguities

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

f o r t



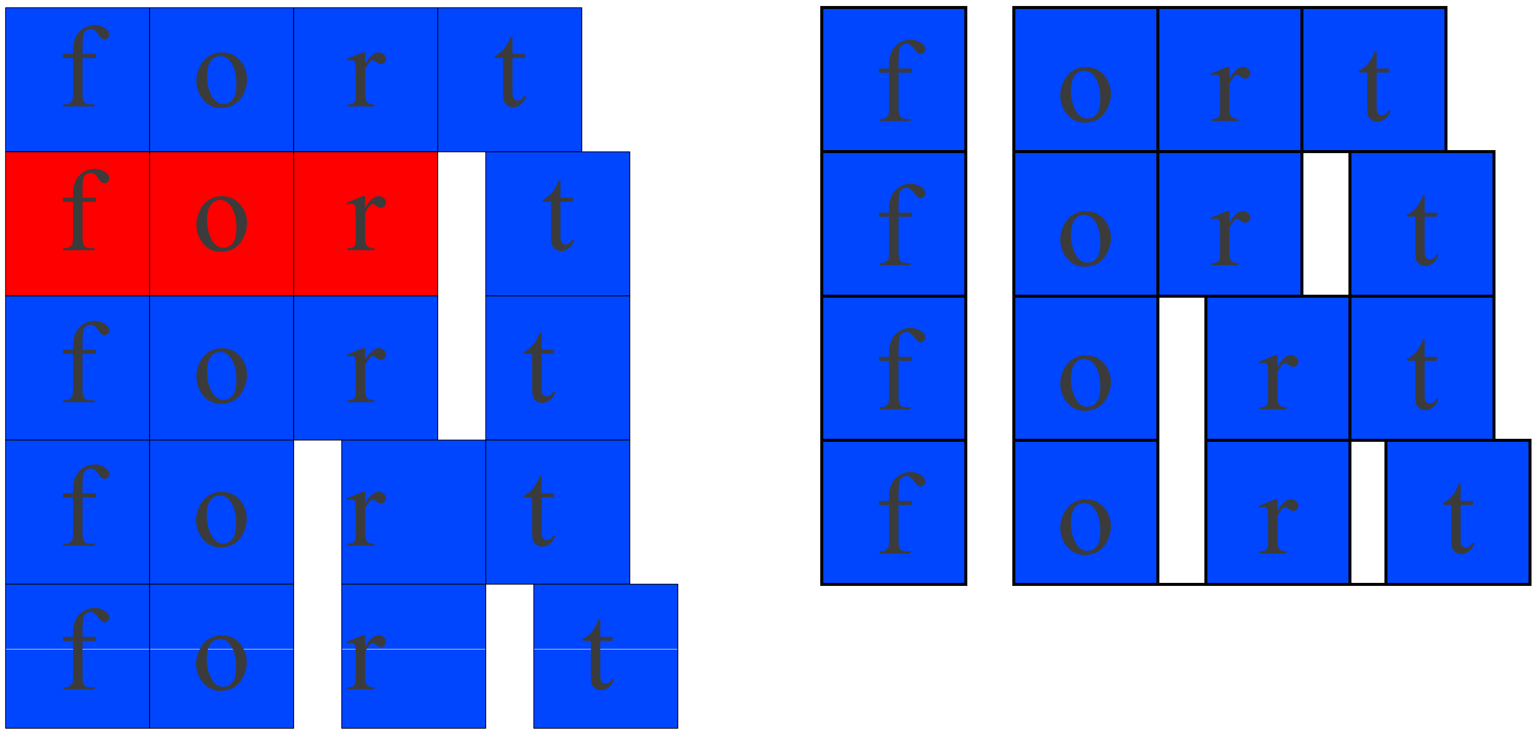
Conflict Resolution

- 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_]*
```

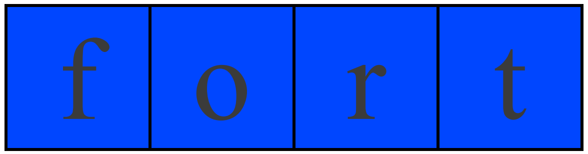
f o r t



Lexing Ambiguities

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

f o r t



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?

Implementing Maximal Munch

- Given a set of regular expressions, how to use them to implement maximum munch?
- Idea:
  - 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.

Other Conflicts

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

d o u b l e

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
d	o	u	b	l	e

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

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

Summary of Conflict Resolution

- Construct an automaton for each regular expression.
- Merge them into one automaton by adding a new start state.
- Scan the input, keeping track of the last known match.
- Break ties by choosing higher- precedence matches.
- Have a catch-all rule to handle errors.

One Last Detail...

- We know what to do if *multiple* rules match
- What if *nothing* matches?
- Trick: Add a “catch-all” rule that matches any character and reports an error.

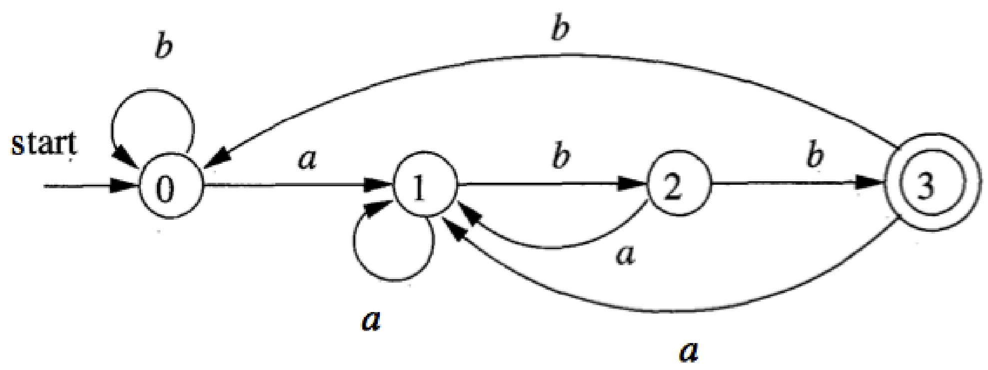
DFAs

- 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.

Simulating DFA

- **INPUT**: An input string *x* terminated by an end-of-file character eof. A DFA D with start state s_0, accepting states F, and transition function move.
- **OUTPUT**: Answer "yes" if D accepts *x*, "no" otherwise.

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```



$aa^*|bb^*$

DFA runs in time O(|*x*|) on an input string *x*.

Speeding up Matching

- In the worst-case, an NFA with n states takes time $O(|r||x|)$ to match a string x.
- DFAs, on the other hand, take only $O(|x|)$.
- There is another (beautiful!) algorithm to convert NFAs to DFAs.

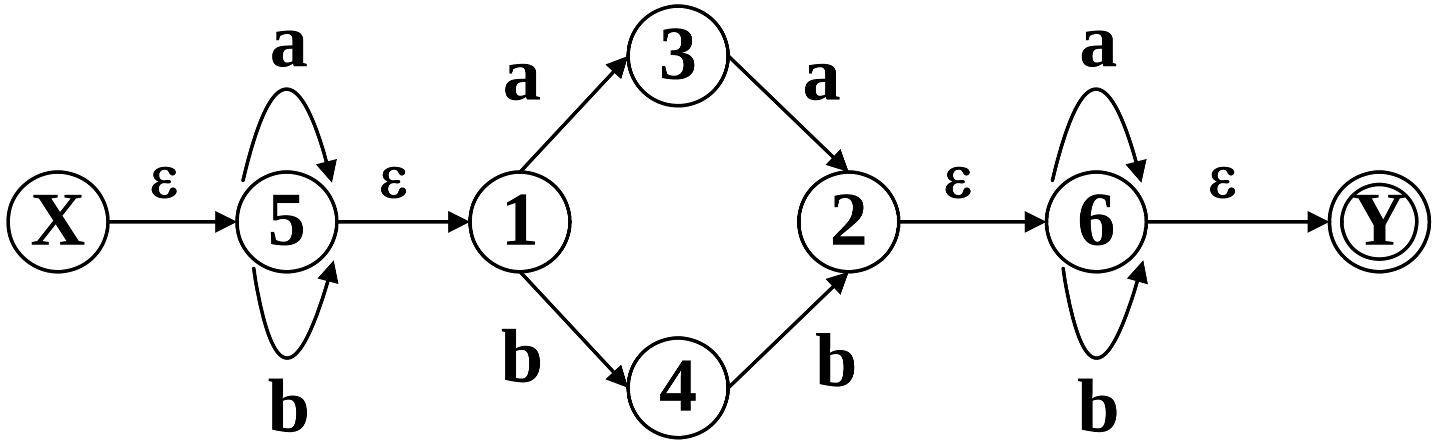
Subset Construction

- 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.

Conversion of an NFA to a DFA

- **INPUT**: An NFA N.
- **OUTPUT**: A DFA D accepting the same language as N.
- **METHOD**: The algorithm constructs a transition table Dtran for D. Each state of D is a set of NFA states, and we construct Dtran so D will simulate "in parallel" all possible moves N can make on a given input string.

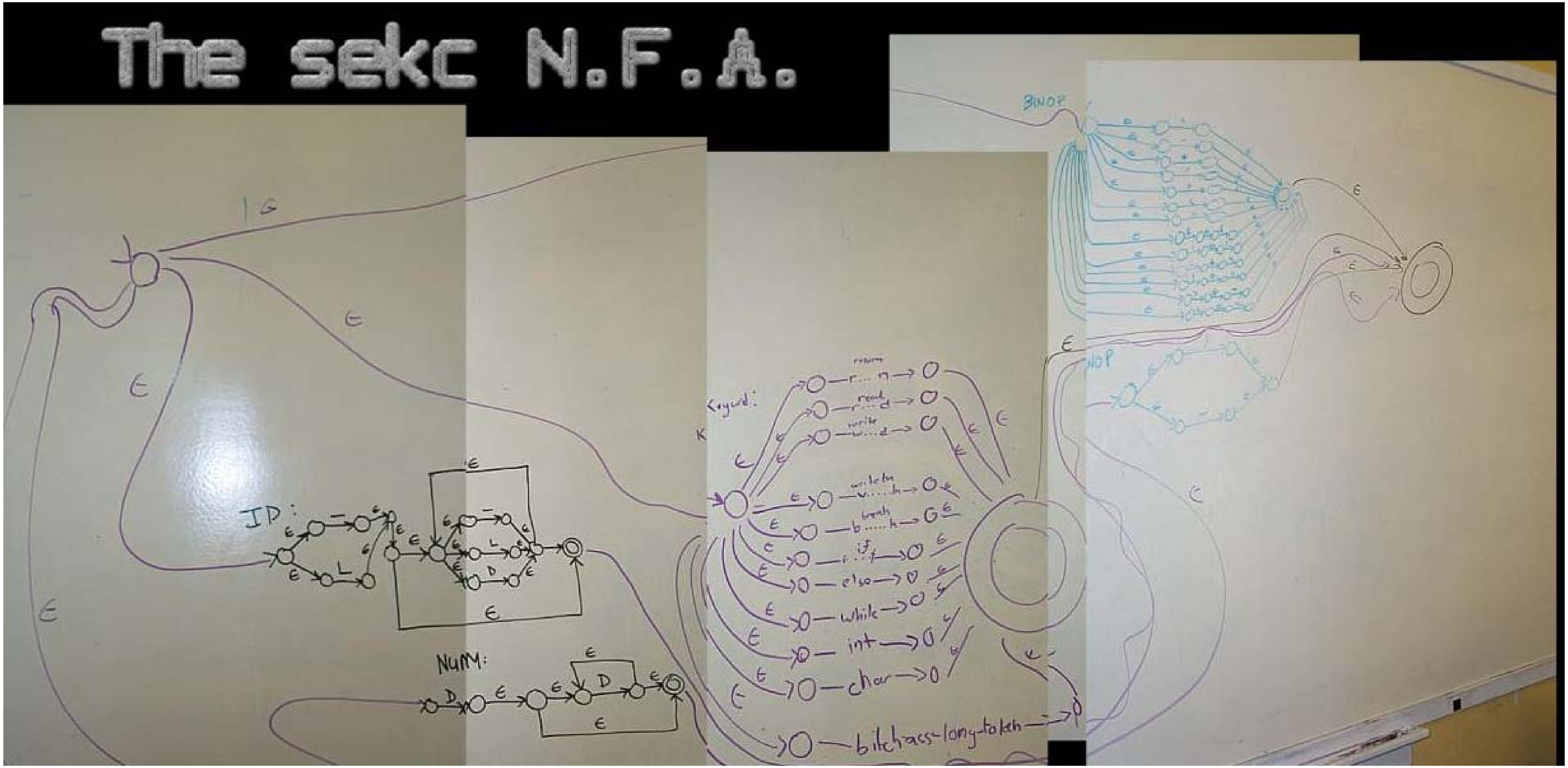
OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

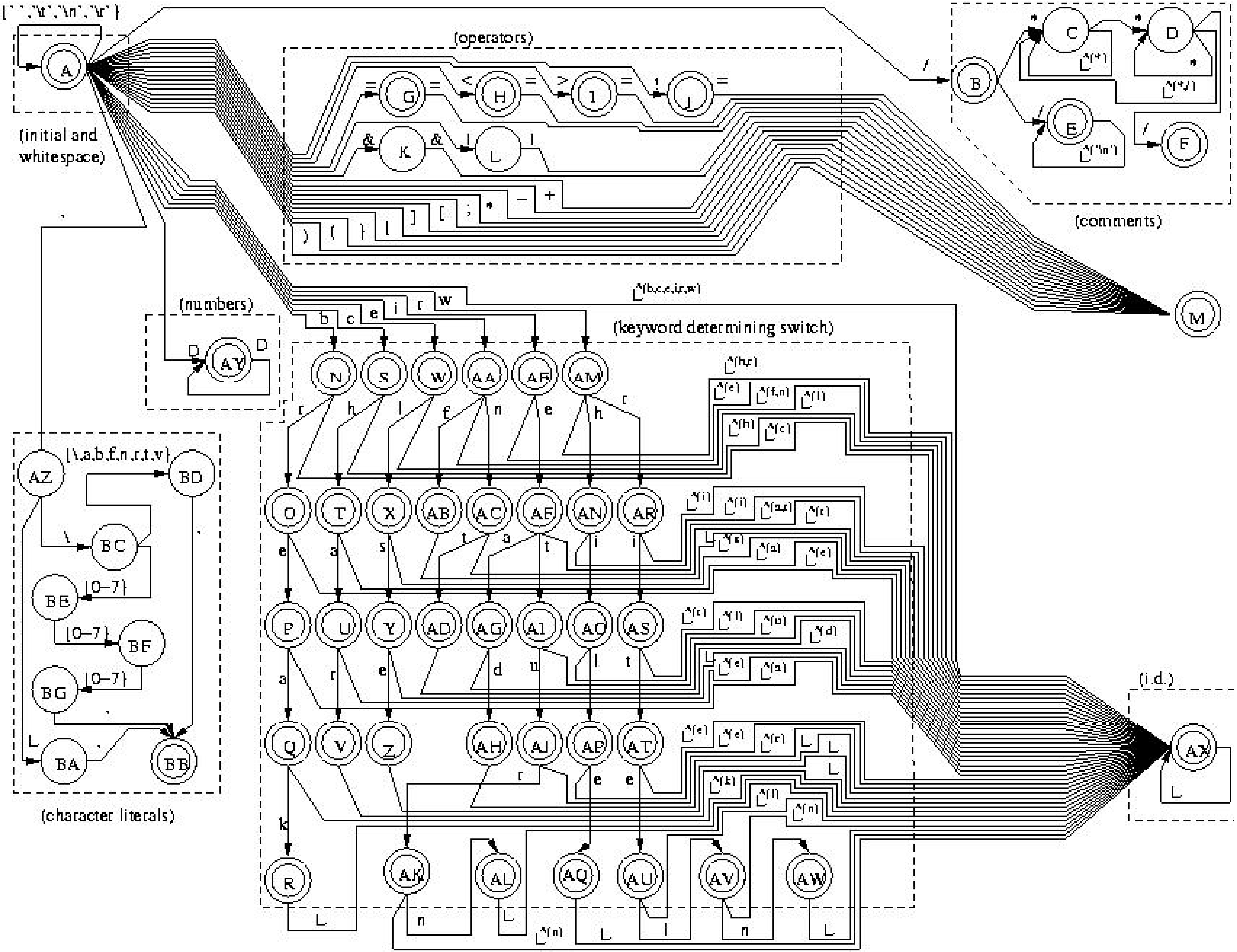


I	I_a	I_b
{X, 5, 1}	{5, 3, 1}	{5, 4, 1}
{5, 3, 1}	{5, 3, 1, 2, 6, Y}	{5, 4, 1}
{5, 4, 1}	{5, 3, 1}	{5, 4, 1, 2, 6, Y}
{5, 3, 1, 2, 6, Y}	{5, 3, 1, 2, 6, Y}	{5, 4, 1, 6, Y}
{5, 4, 1, 6, Y}	{5, 3, 1, 6, Y}	{5, 4, 1, 2, 6, Y}
{5, 3, 1, 2, 6, Y}	{5, 3, 1, 6, Y}	{5, 4, 1, 2, 6, Y}
{5, 3, 1, 6, Y}	{5, 3, 1, 2, 6, Y}	{5, 4, 1, 6, Y}

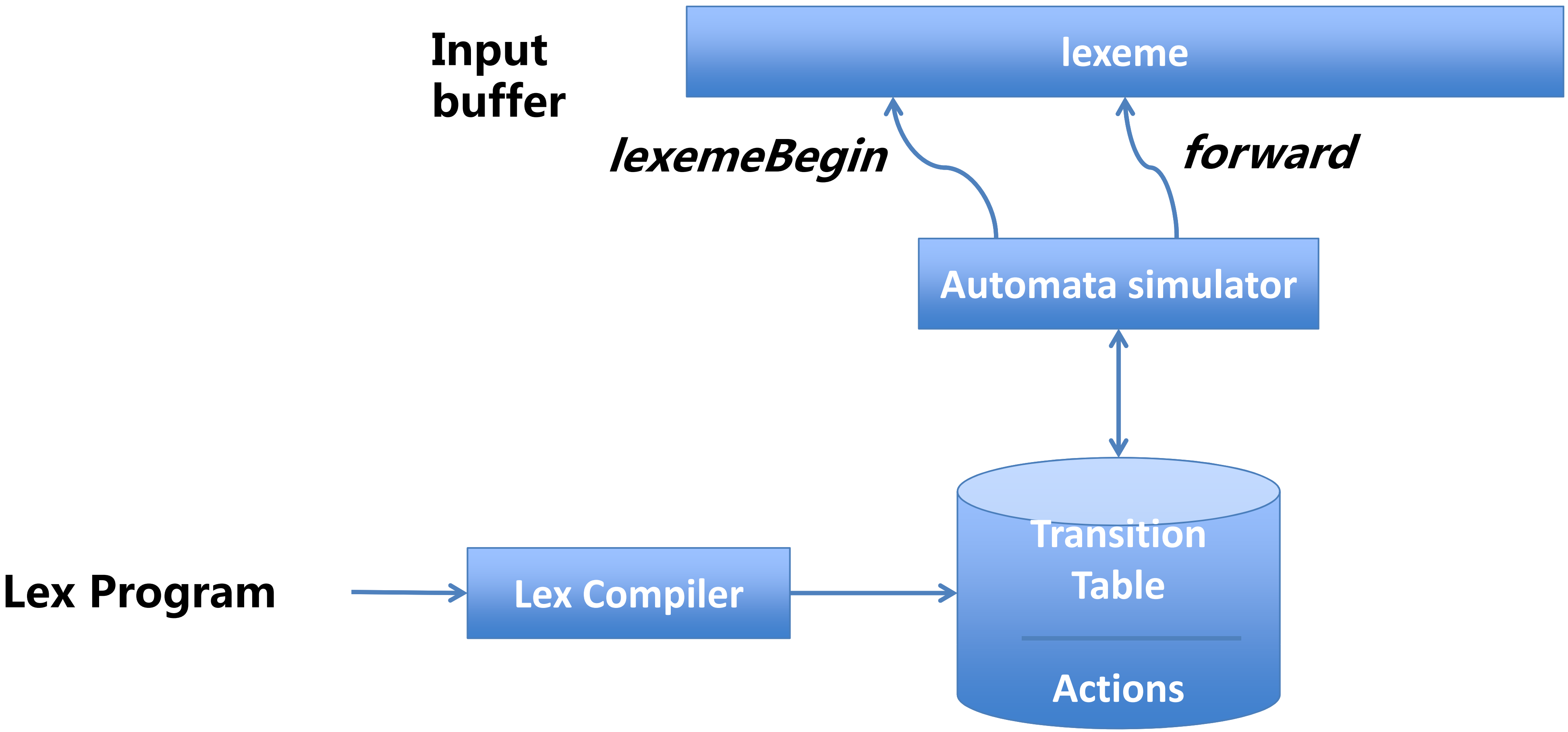
Minimizing the number of states of a DFA

- **INPUT**: An DFA *D*
- **OUTPUT**: A DFA *D’* accepting the same language as *D* and having as few states as possible.
- **METHOD**:
  1. Construct initial partition Π of S with two groups: accepting/ non-accepting.
  2. (Construct Π_{new})For each group *G* of Π do begin
     a) Partition *G* into subgroups such that two states s,t of *G* are in the same subgroup if for all symbols a states s,t have transitions on a to states of the same group of Π.
     b) Replace *G* in Π_{new} by the set of all these subgroups.
  3. Compare Π_{new} and Π. If equal, Π_{final}:= Π then proceed to 4, otherwise, set $\Pi := \Pi_{new}$ and goto 2.
  4. Choose one state in each group of Πfinal as the representative for that group.


The sekc N.F.A.
ID:
NUM:
BINOP
while
else
int
char
bitch-ass-long-token


(operators)
(initial and whitespace)
(comments)
(numbers)
(keyword determining switch)
(character literals)
(i.d.)

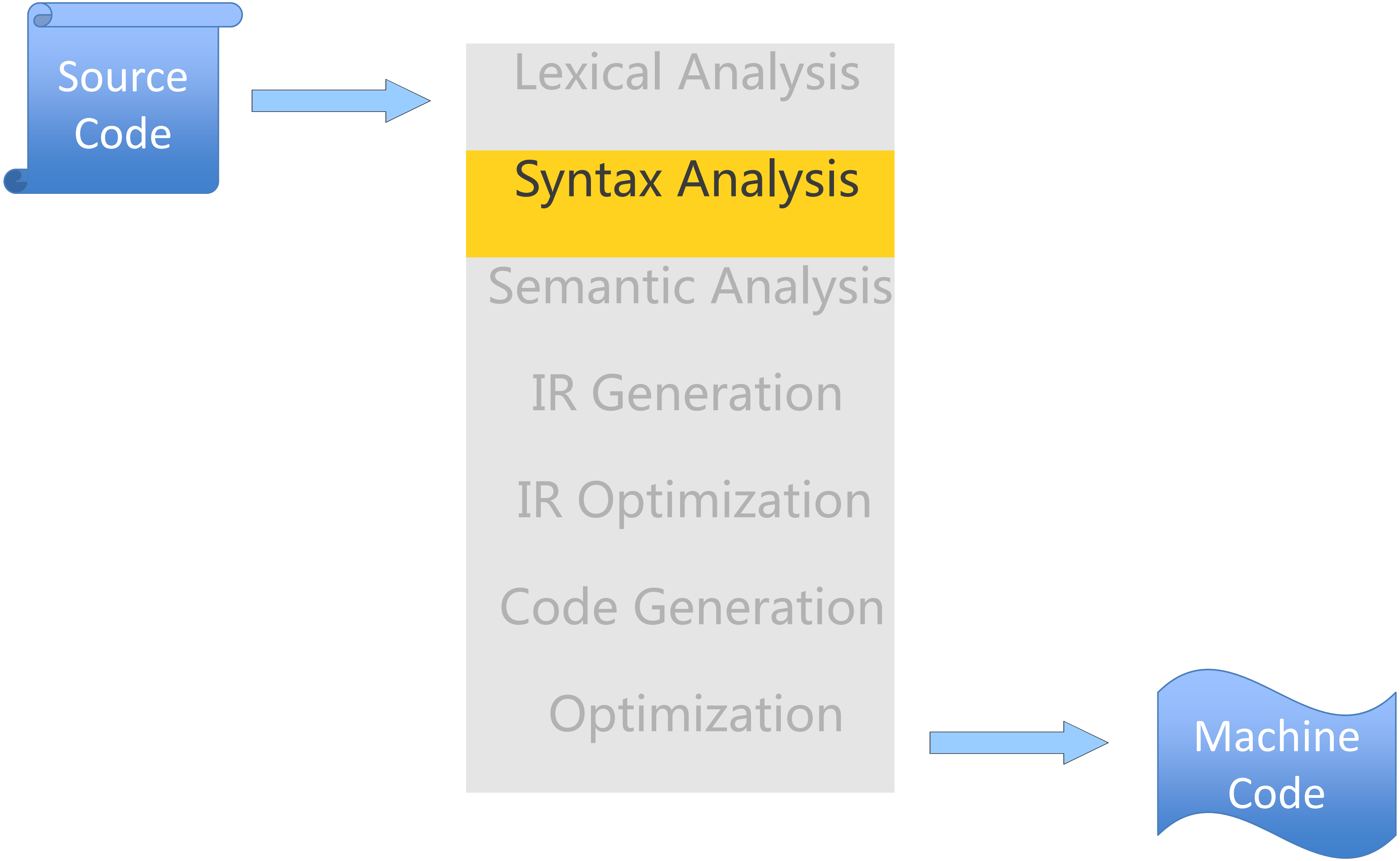
A Lexical Analyzer Generated by Lex



Lexical Analysis Summary

- Problems and challenges in Lexical Analysis
  - 3.1, 3.2
- Describe problems and Lexical rules
  - 3.3
  - Lexical Specification
  - Regular Expressions
- Recognition of Tokens
  - 3.4
  - NFA (3.6.1, 3.6.2, 3.6.3)
  - DFA (3.6.4)
  - From RE to automata (3.7)
- Lexical-Analyzer Generator
  - 3.5, 3.8

Next Time


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

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu