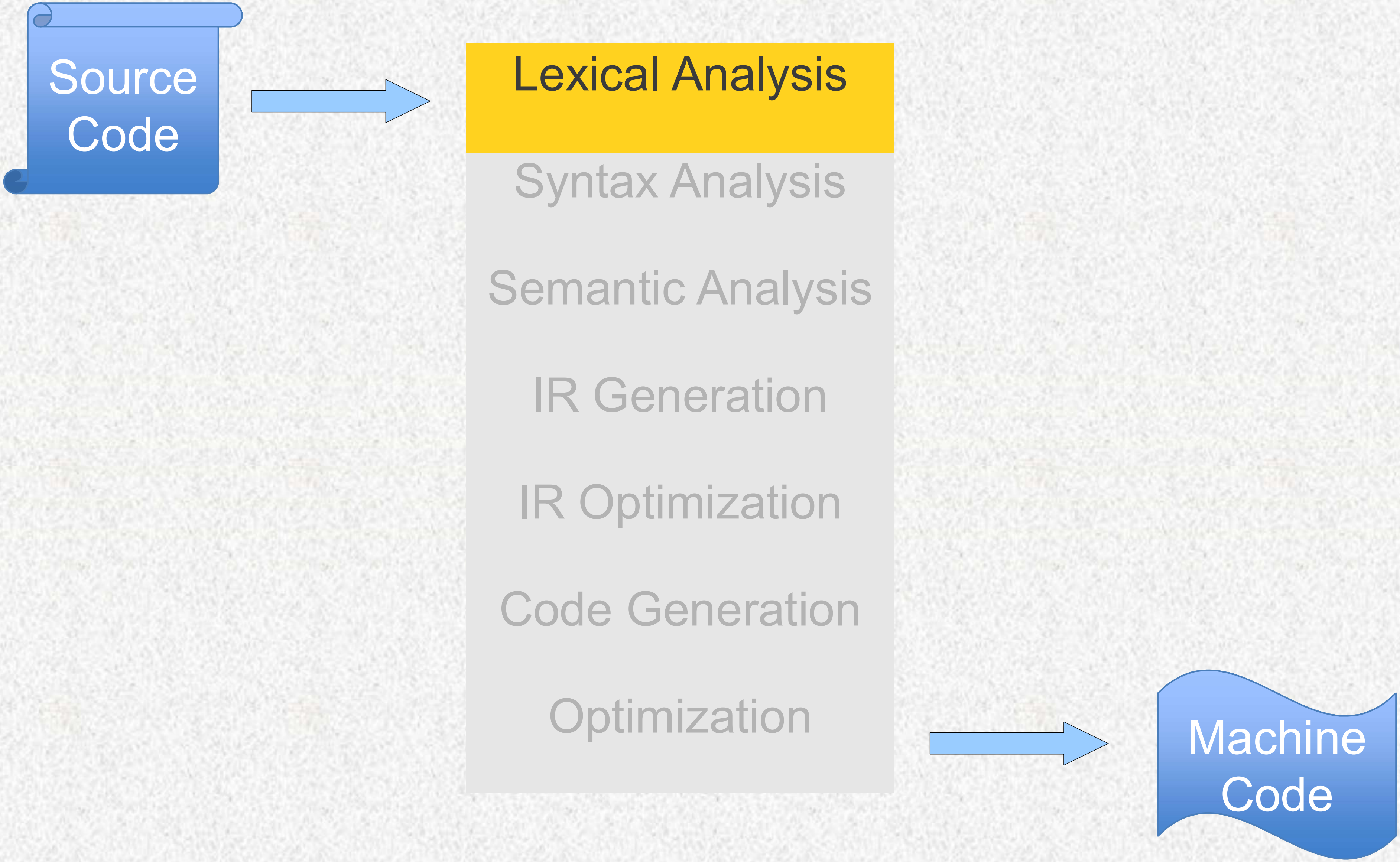
Lecture 2: Lexical Analysis

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
School of Computer Science E301

Where We Are


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

A motivation example

What do we want to do?

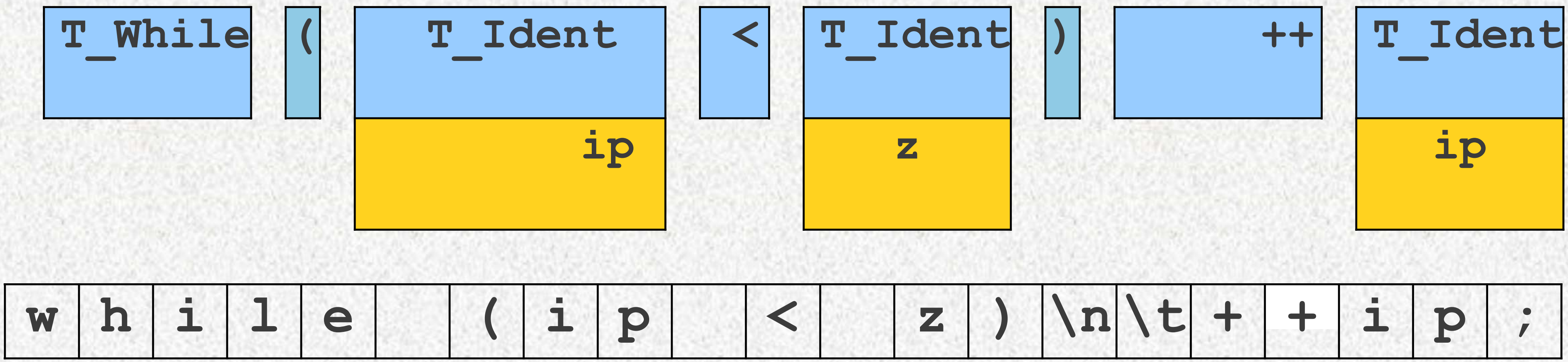
```
while (ip < z)
    ++ip;
```

What do we want to do?

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

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

What do we want to do?




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

What do we want to do?

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

do[for]   = new 0;

What do we want to do?

T_Do	[	T_For	]	=	T_New	T_IntConst
						0

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

do[for]   = new 0;

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

This is straightforward

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.

How to decide the type?

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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.

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

T_While

Scan and partition input string into substrings (i.e. tokens)

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

T_While

(

Scan and partition input string into substrings (i.e. tokens)

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

T_While

(

Scan and partition input string into substrings (i.e. tokens)

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

T_While   (

Scan and partition input string into substrings (i.e. tokens)

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

T_While

(

Scan and partition input string into substrings (i.e. tokens)

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

T_While

(

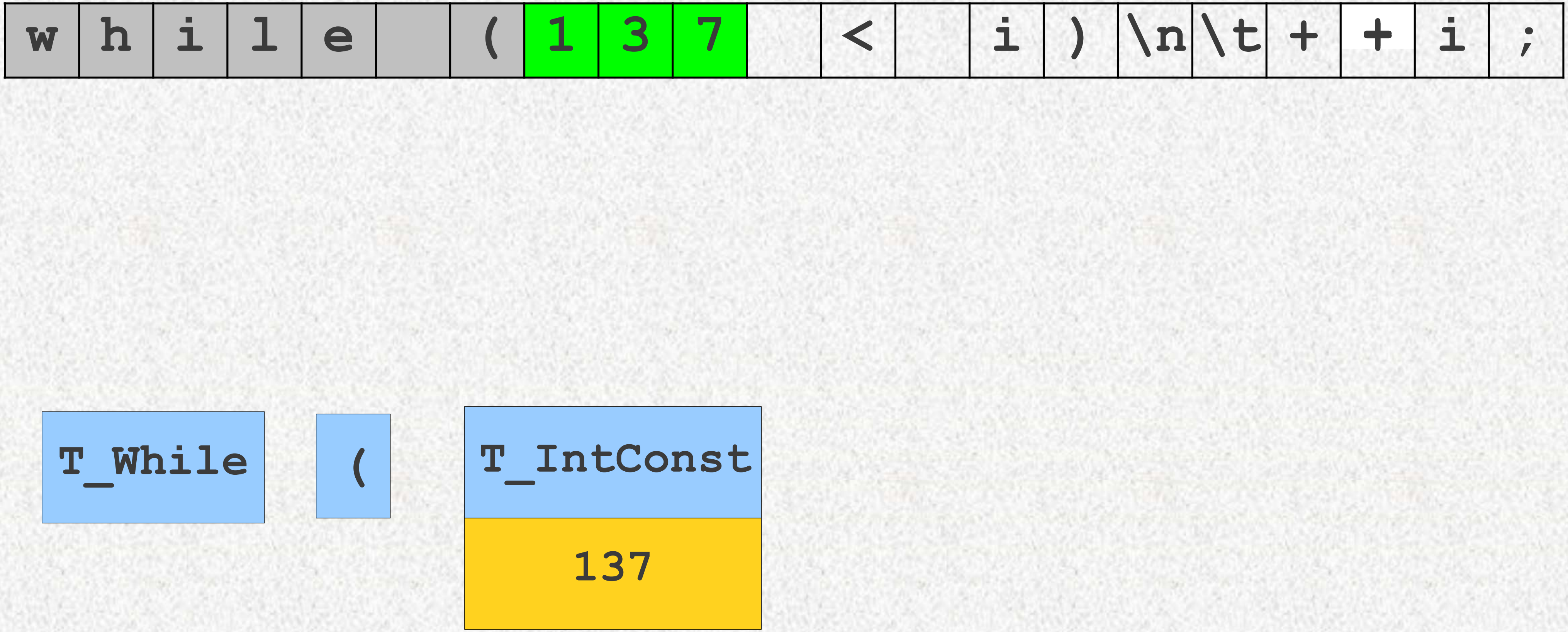
Scan and partition input string into substrings (i.e. tokens)

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

T_While

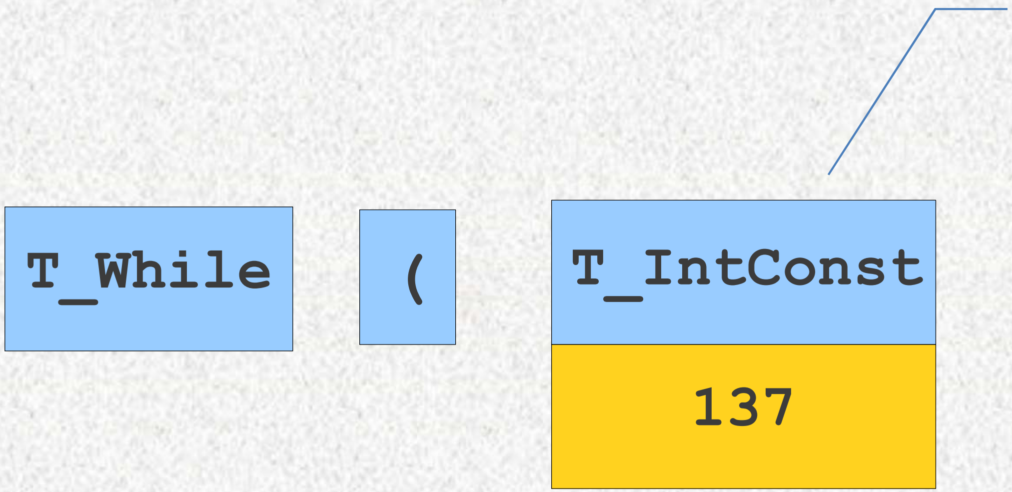
(

Scan and partition input string into substrings (i.e. tokens)



Scan and partition input string into substrings (i.e. tokens)

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

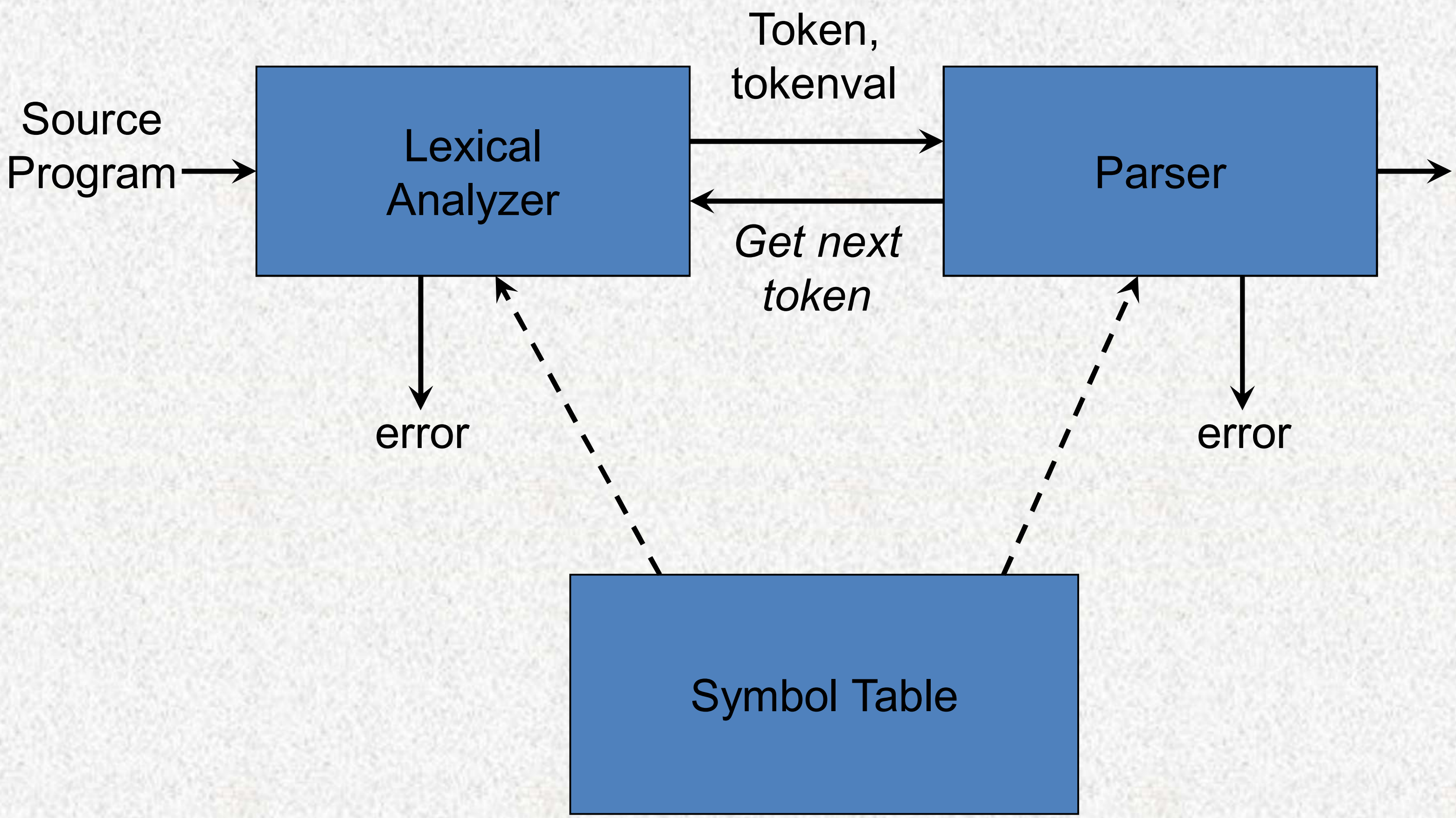


Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

Goals of Lexical Analysis

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

Interaction of the Lexical Analyzer with the Parser

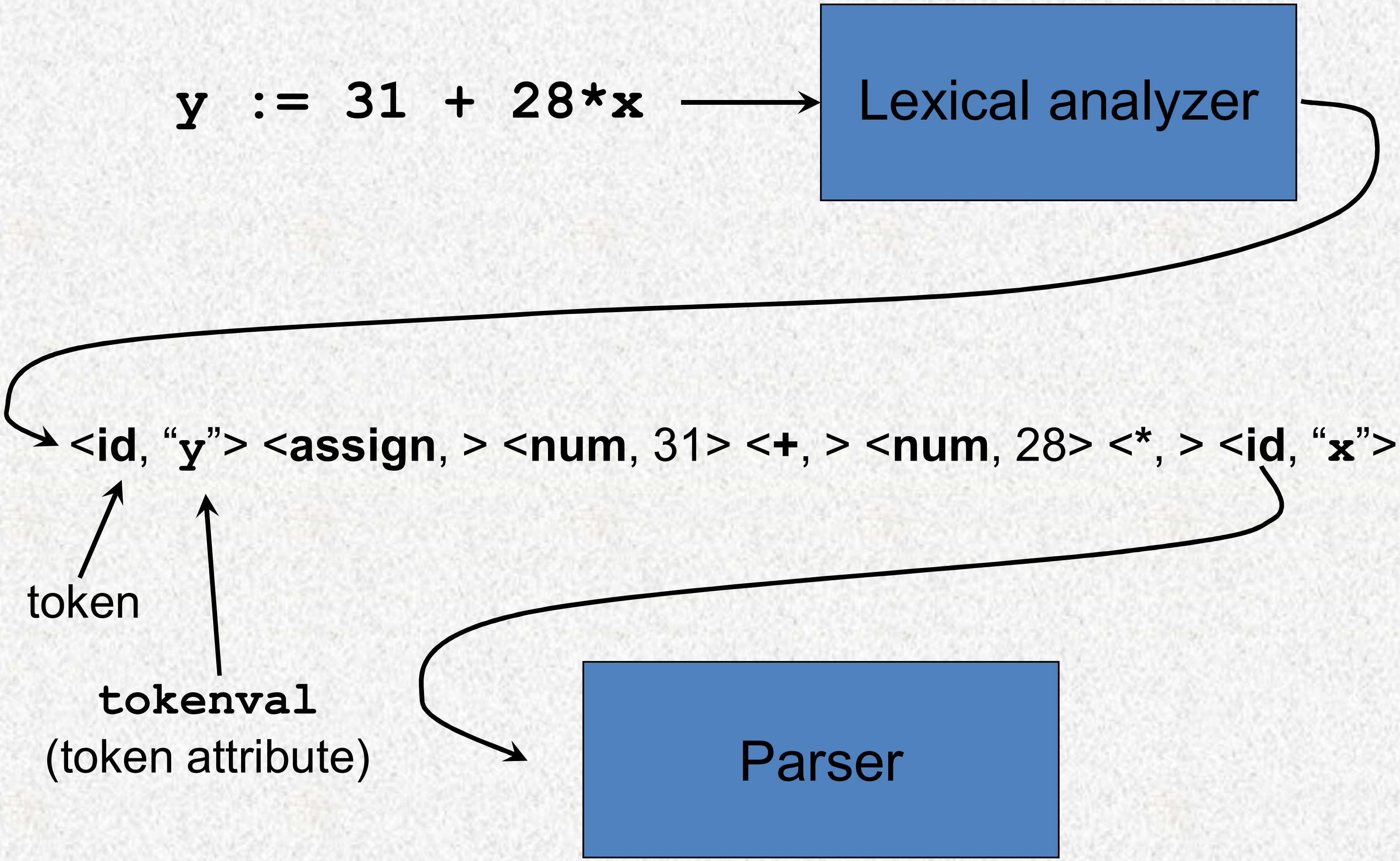


What is a token

What is a token?

- A token should indicate a syntactic category of a lexeme
  - In English: noun, verb, adjective, …
  - In a programming language: identifier, Integer, Keyword, Whitespace, …

Attributes of tokens



What is a token?

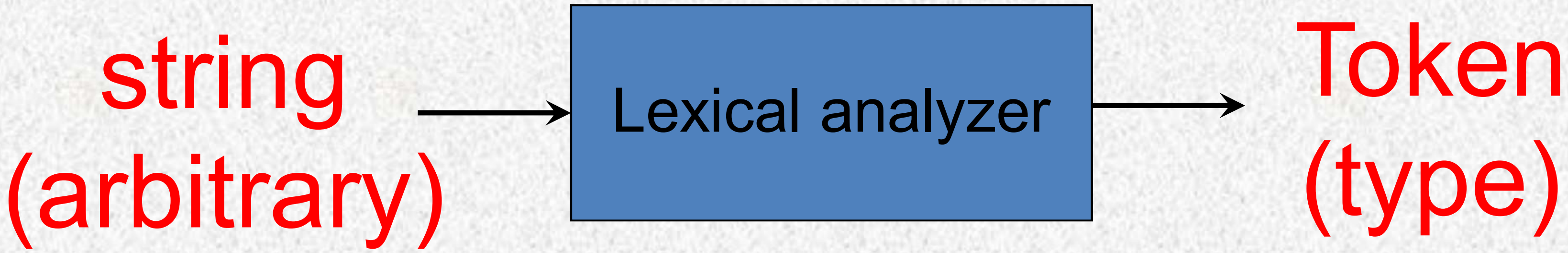
- A token corresponds to sets of strings (a type/category/class)
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

What are tokens for?

- Classify program substrings according to their roles
- Output of lexical analysis is a stream of tokens
- Parser relies on token distinctions
    - E.g. an identifier is treated differently from a keyword

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.



Strings are infinite

We need a method to describe the infinite strings with finite rules

Describe infinite strings with finite rules

- First, we define finite categories/types of tokens
  - Keywords, number, identifier, operator, etc.
- Secondly, we use finite rules to describe each type of token

How?

Formalisms of tokens

Regular languages

- Regular languages are used to define the category/type of a token in finite rules
- Three ways to describe a regular language
  - Grammar, Regular Expression, Finite Automaton
  - Equivalent to each other

Any grammar can be regarded as a generating device: derive infinite set of strings (i.e. language)

Formally define Languages

- An *alphabet* table Σ is a finite set of symbols (characters)
- A *string* *s* is a finite sequence of symbols from Σ
  - |*s*| denotes the length of string *s*
  - ε denotes the empty string, thus |ε| = 0
- A *language* is a specific set of strings over some fixed alphabet Σ (a subset of all possible strings)

Examples of languages

Type-III:
Alphabet = English characters
Language = English words

Not every string of English characters is an English word!

Type-II:
Alphabet = English characters
Language = English sentences

Not every string of English characters is an English word!

Examples of languages

Type-III:
Alphabet = ASCII
Language = C tokens

Not every string of ASCII characters is a C token!

Type-II:
Alphabet = ASCII
Language = C programs

Not every string of ASCII characters is a C program!

Examples of languages

Alphabet = English characters
Language = English words

Not every string of English characters is an English word!

Alphabet = ASCII
Language = C programs

Regular language is (Type-III) language
--- regular expression
--- finite automaton

Regular Expression

Finite Automaton

Regular Expression

Finite Automaton

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (i.e. 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.`**

Identifier: strings of letters or digits, starting with a letter
letter = ‘A’ | . . . | ‘Z’ | ‘a’ | . . . |‘z’
identifier = letter (letter | digit)*

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, that is to repeat R for 0-n times
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>* is idempotent</td></tr>
</table>

Regular Expression v.s. Regular Language

- Regular expression can represent a set of strings, which form a regular language

Let Σ= {a, b}

The regular expression a l b denotes the language {a, b} .

(alb)(alb) denotes {aa, ab, ba, bb} , the language of all strings of length two over the alphabet.
Another regular expression for the same language is
aa l ab l ba l bb.

a* denotes the language consisting of all strings of zero or more a's, that is, {ε, a, aa, aaa, ... }.

Regular Expression v.s. Regular Language

Let Σ= {a, b}

(a I b) * denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {E, a, b, aa, ab, ba, bb, aaa, ... }. Another regular expression for the same language is (a* b *)*.

a I a* b denotes the language {a, b, ab, aab, aaab, ... }, that is, the string a and all strings consisting of zero or more a's and ending in b.

Sample 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)*

Sample 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)*

Sample 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

Sample 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

Sample Regular Expressions

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

Sample 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)

Sample 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)

Sample 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

Sample 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

Sample 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

Sample 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+)+

abc@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

More examples

Keyword: “else” or “if” or “begin” or …
‘else’ | ‘if’ | ‘begin’ | . . .

Integer: a non-empty string of digits
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
integer = digit digit*
Abbreviation: $A^+ = AA^*$

Identifier: strings of letters or digits, starting with a letter
letter = ‘A’ | . . . | ‘Z’ | ‘a’ | . . . |‘z’
identifier = letter (letter | digit)*

Is (letter | digit*) the same?*

Regular Expression

Finite Automaton

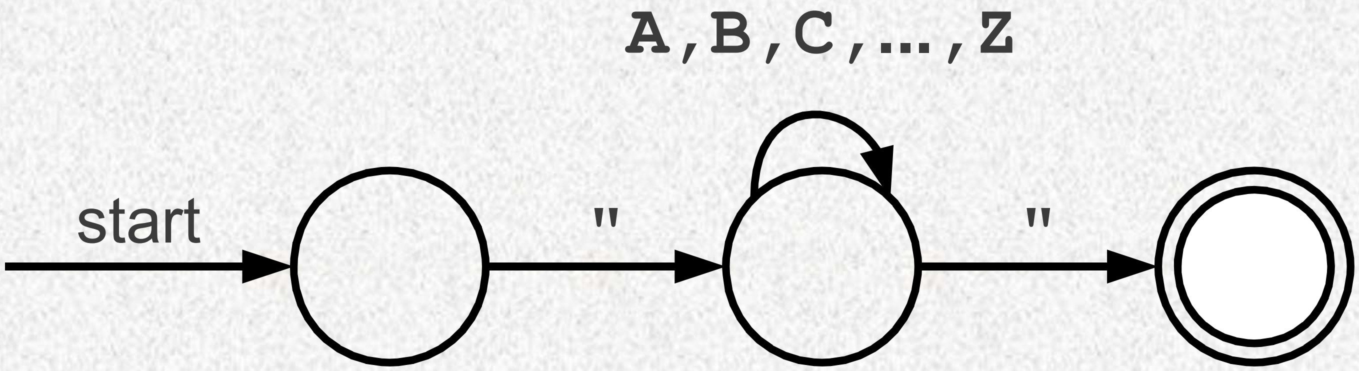
Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
  - Regular expressions = specification
  - Finite automata = implementation
- 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.

Finite Automatons

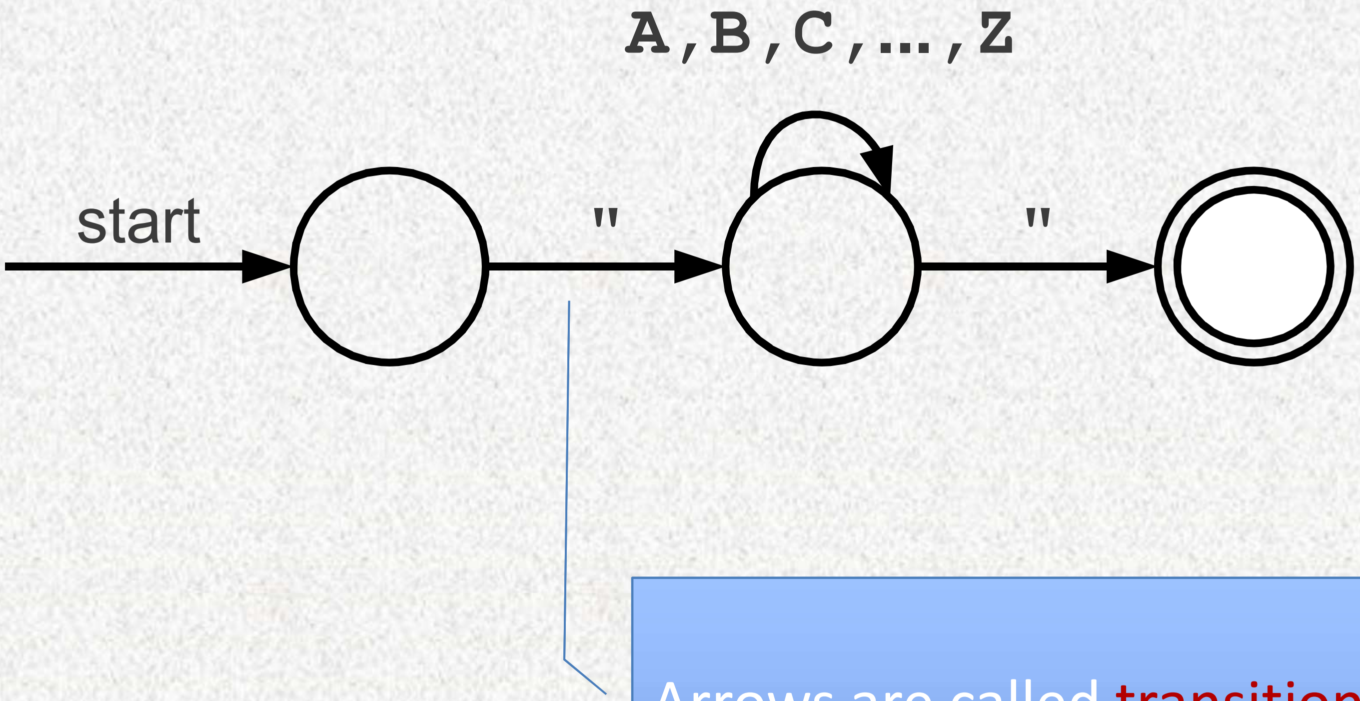
- A finite automaton is a 5-tuple (S,Σ,δ,s_0,F)
  - A set of states S --- nodes
  - An input alphabet Σ
  - A transition function $\delta(S_i, a)=S_j$
  - A start state S_0
  - A set of accepting states $F \subseteq S$

A Simple Automaton



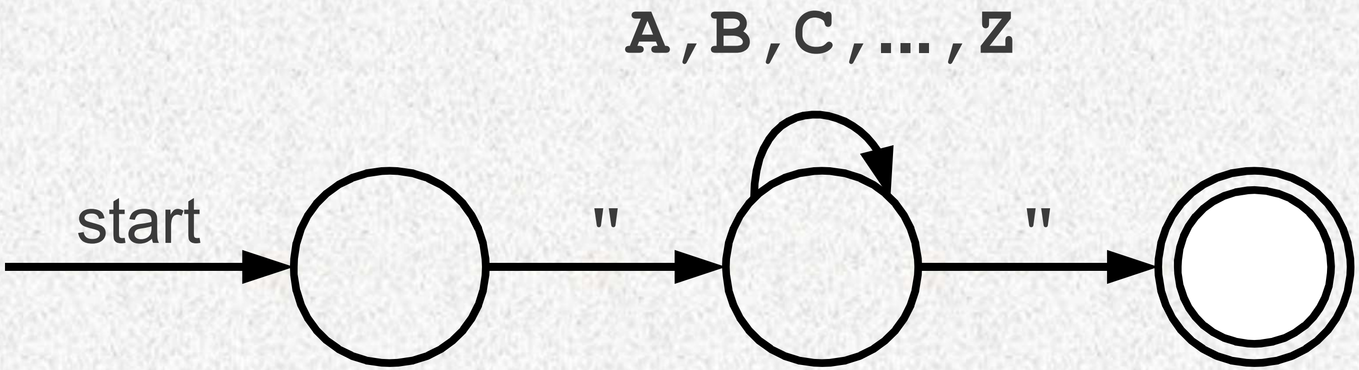
Transition diagrams have a collection of nodes or circles, called states.

A Simple Automaton



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

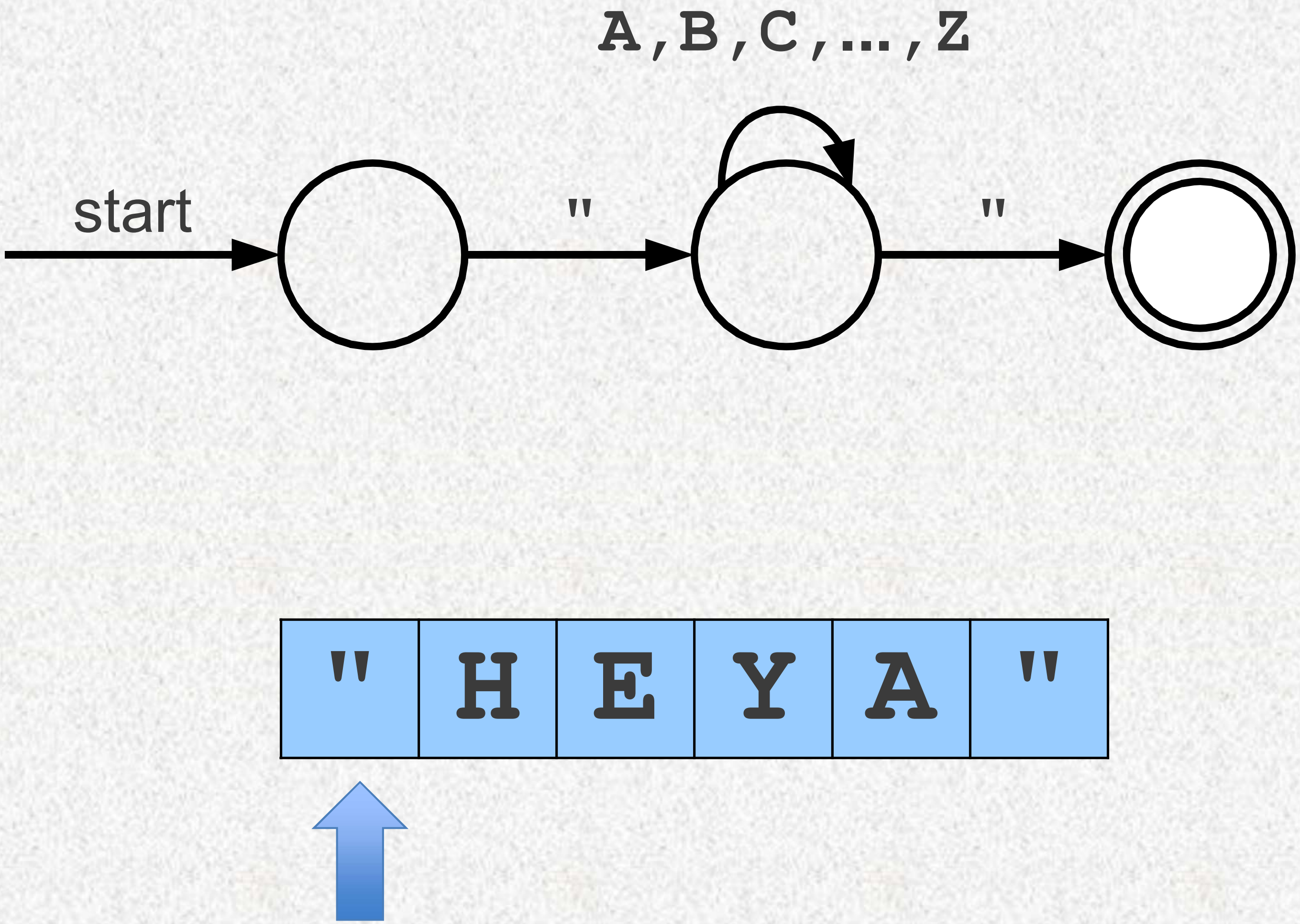
A Simple Automaton



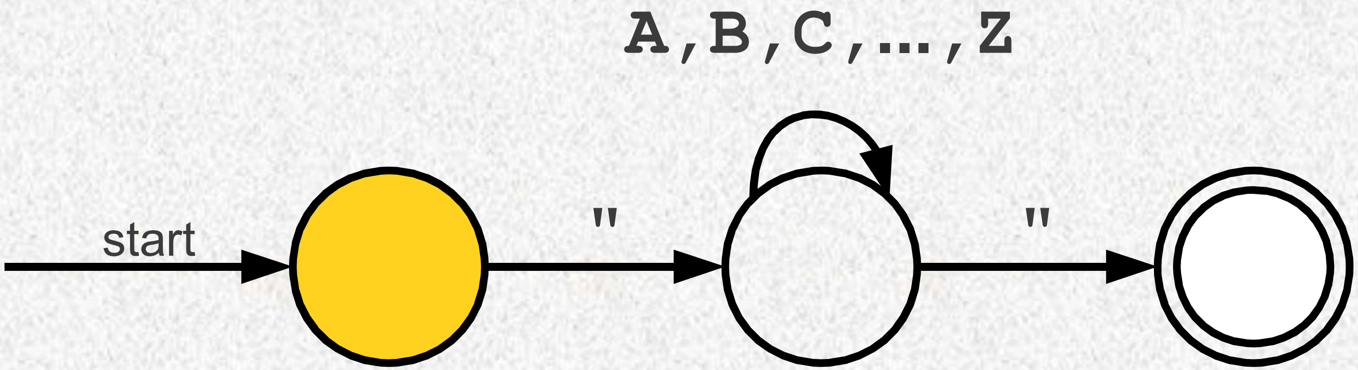
"	H	E	Y	A	"

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

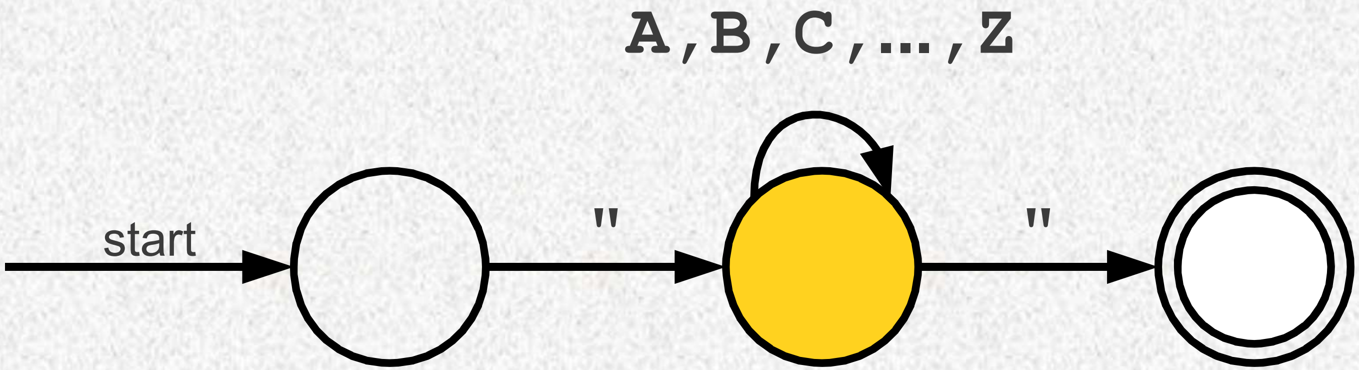
A Simple Automaton



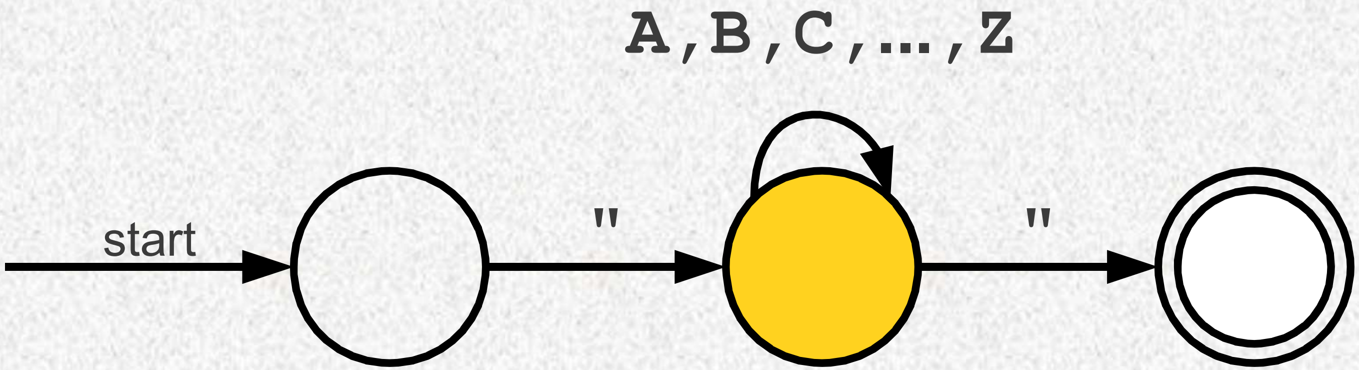
A Simple Automaton

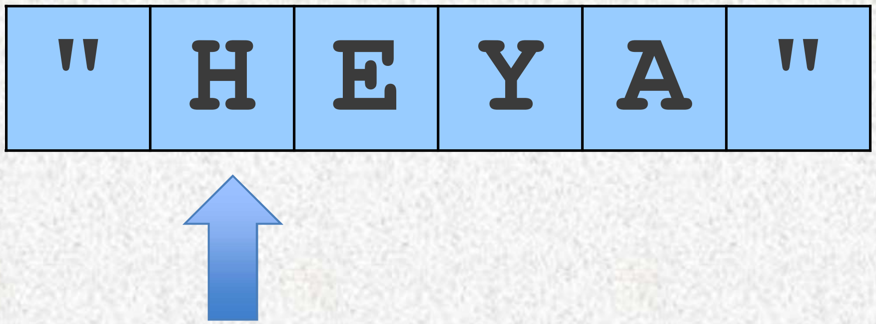


A Simple Automaton

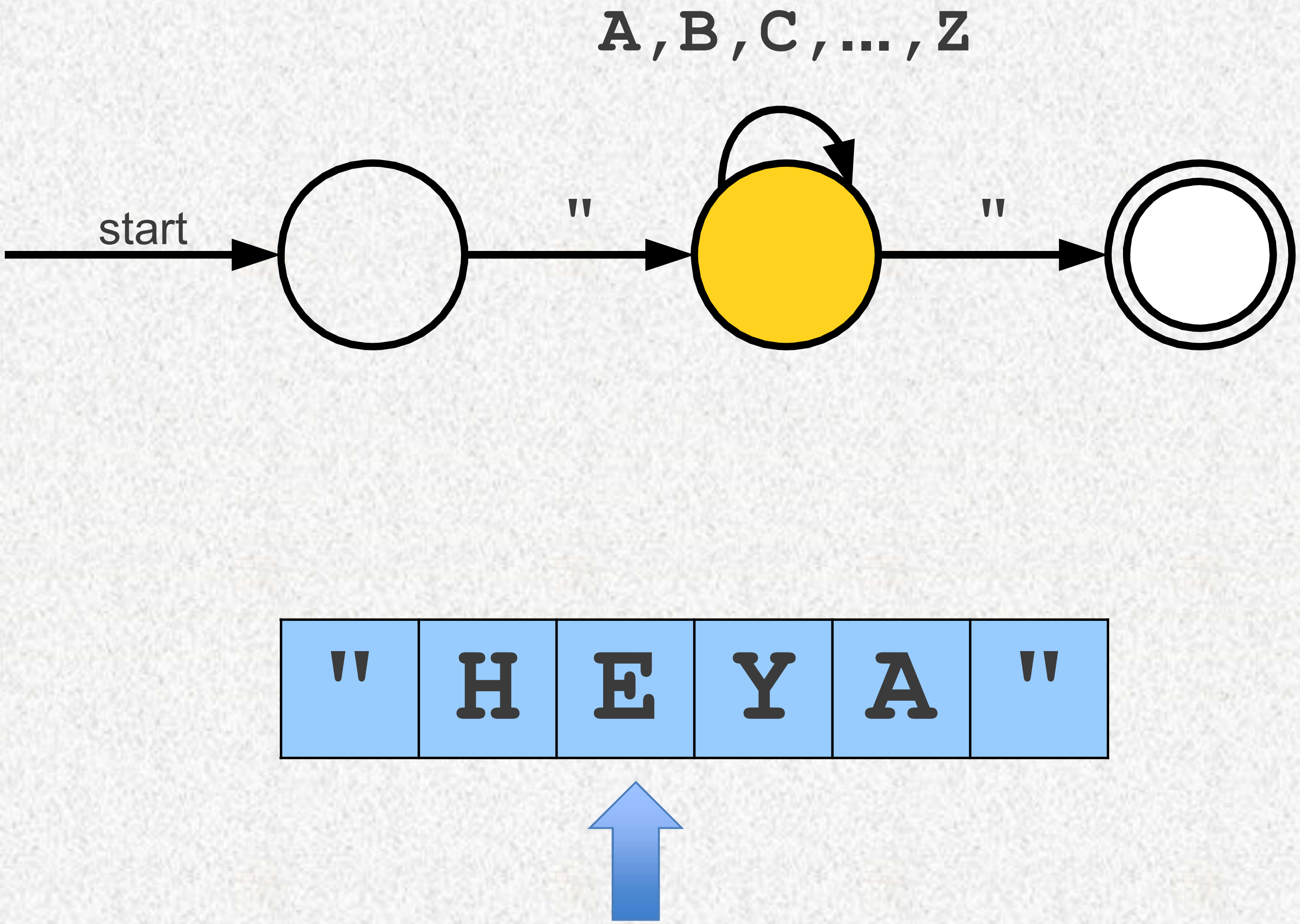


A Simple Automaton

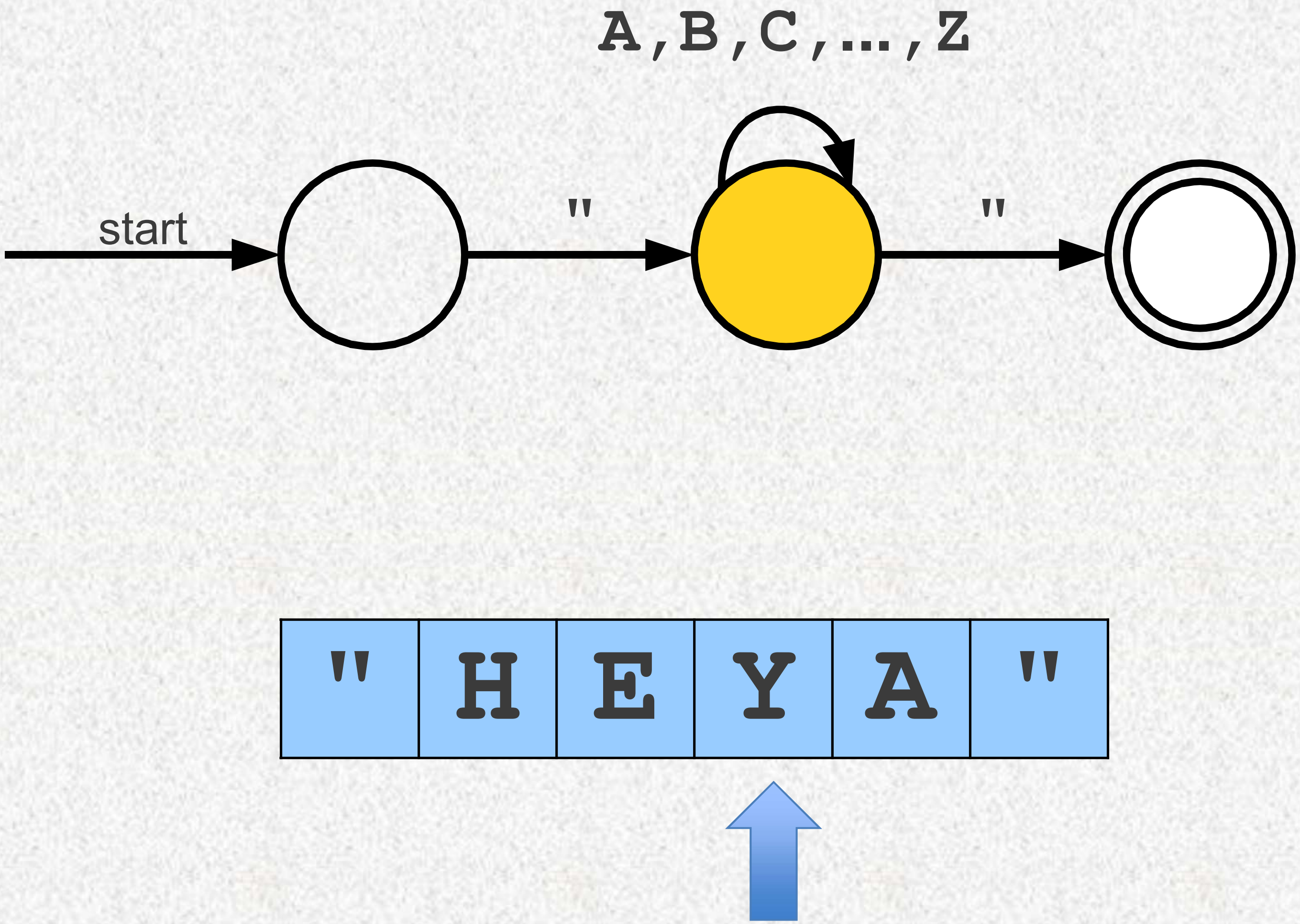




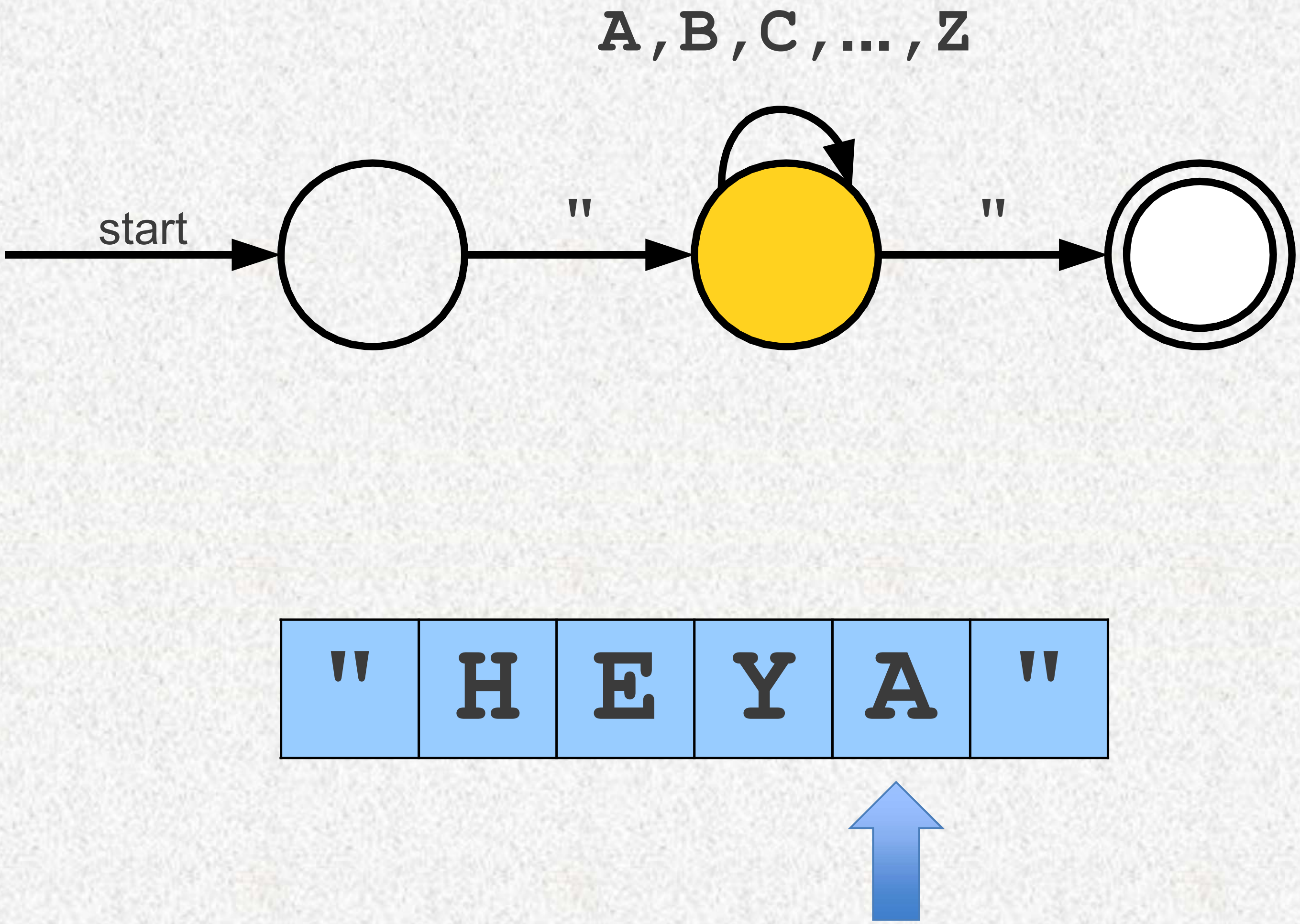
A Simple Automaton



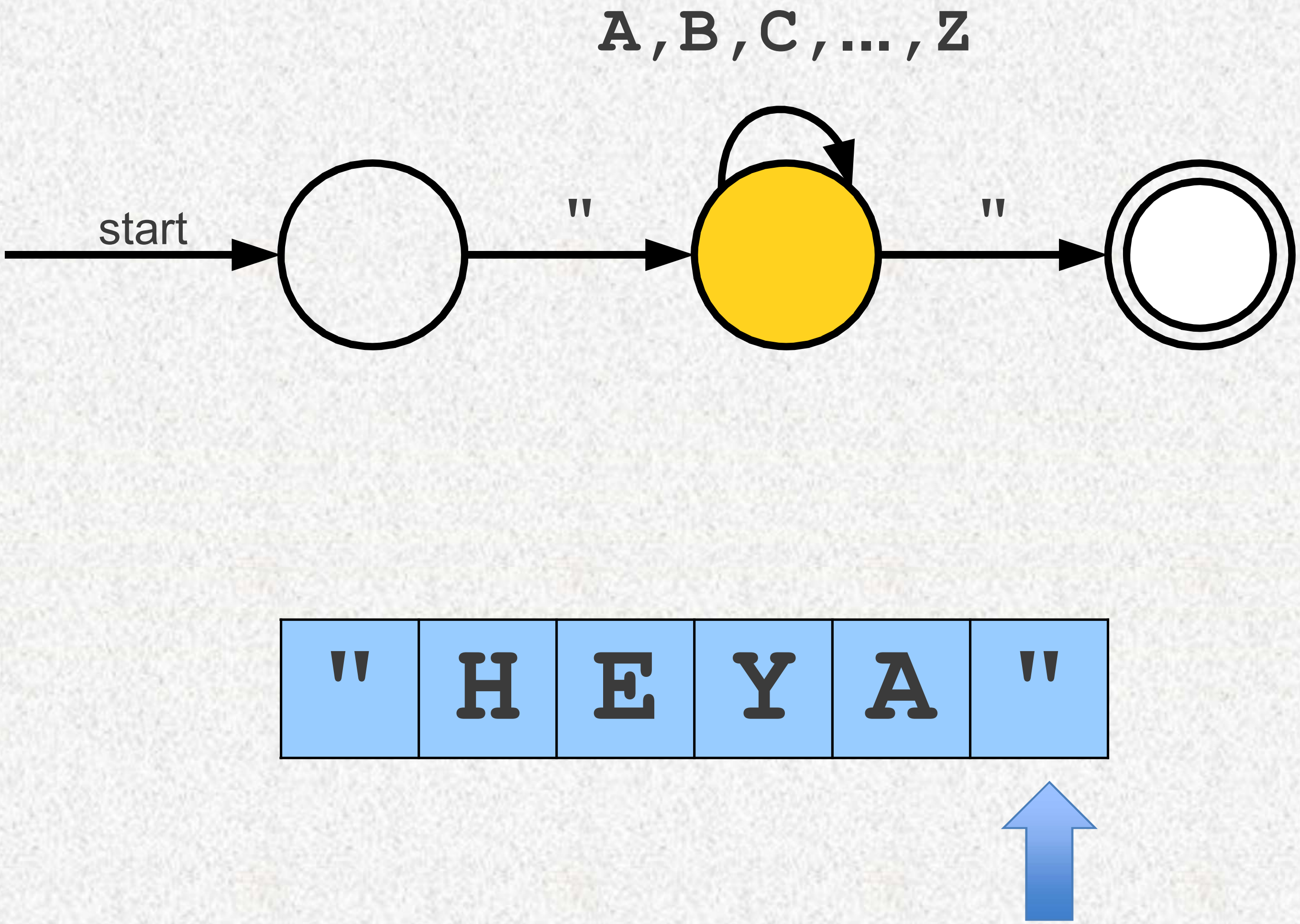
A Simple Automaton



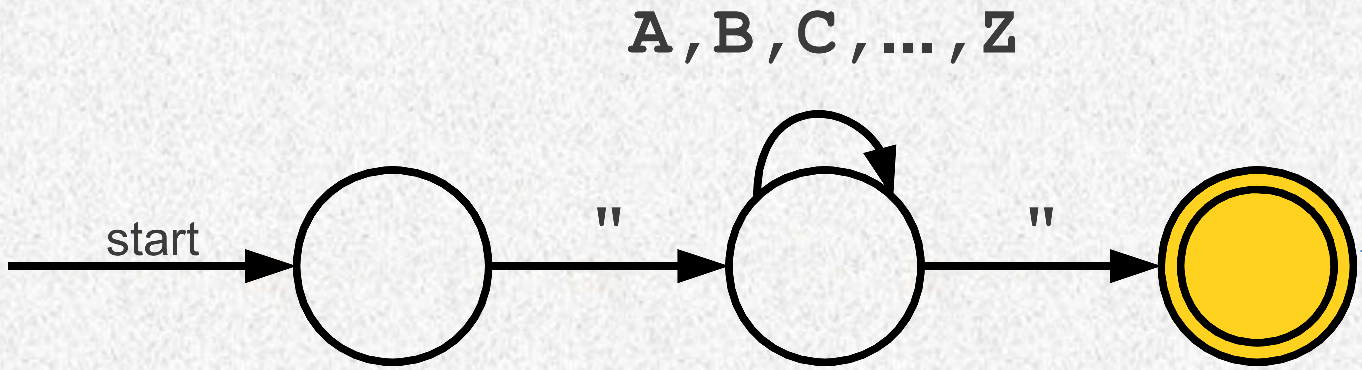
A Simple Automaton



A Simple Automaton

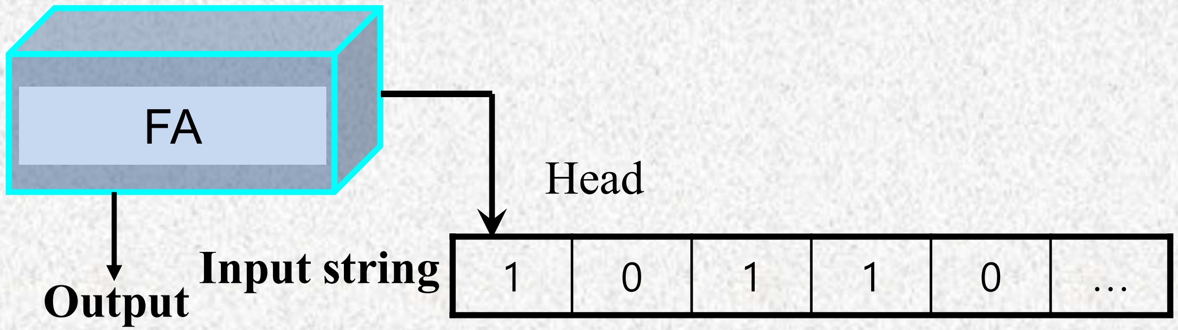


A Simple Automaton



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

Finite Automatons

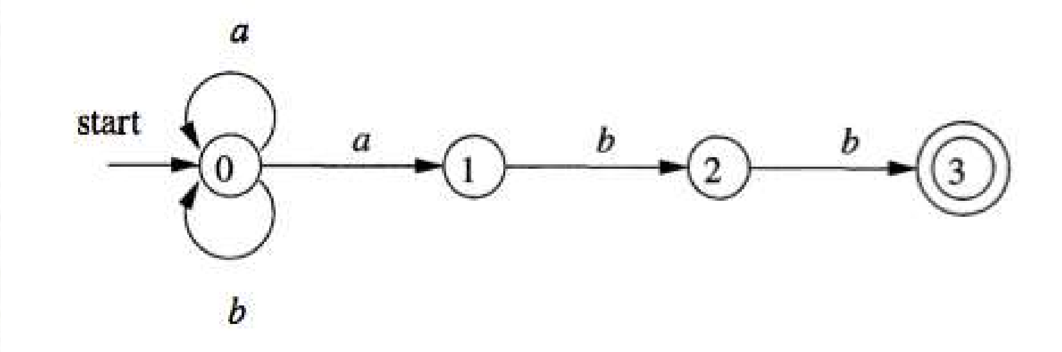


- Input: a string
- Output： accept if the scanning of input string reaches its EOF and the FA reaches an accepting state; reject otherwise

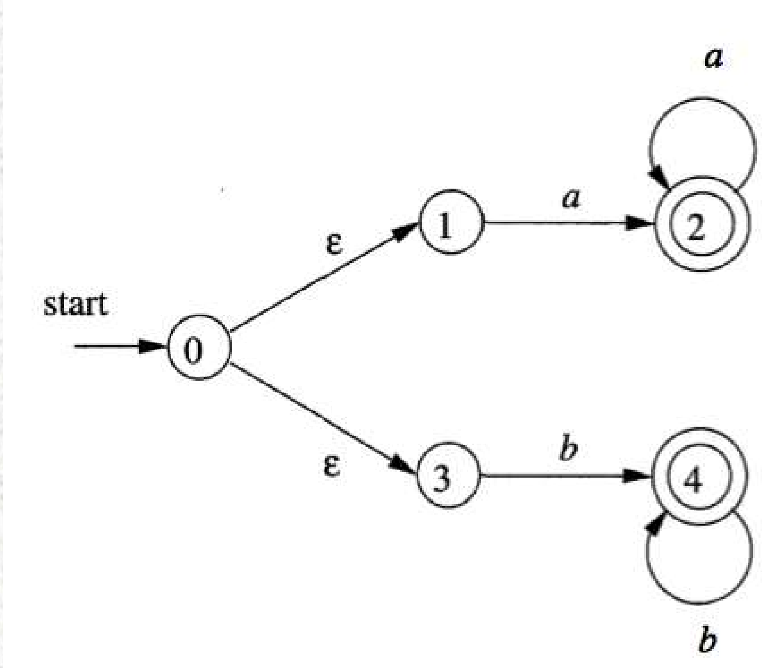
Strings accepted by an FA

- An FA *accepts an input string x* **iff** there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an FA is the set of input strings it accepts, such as (**a**|**b**)*__abb__ for the example NFA

Strings accepted by an FA

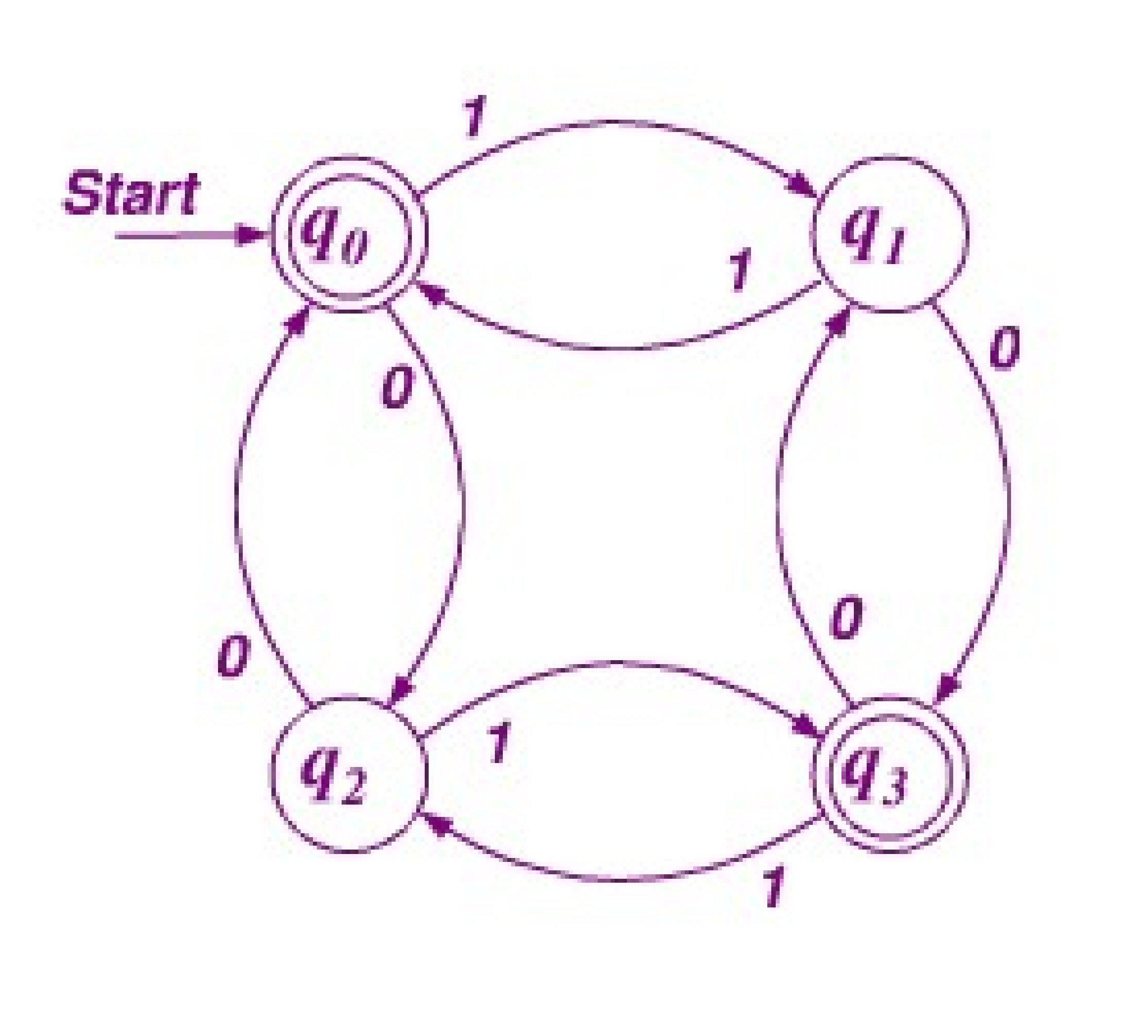


(a|b)*abb



aa*|bb*

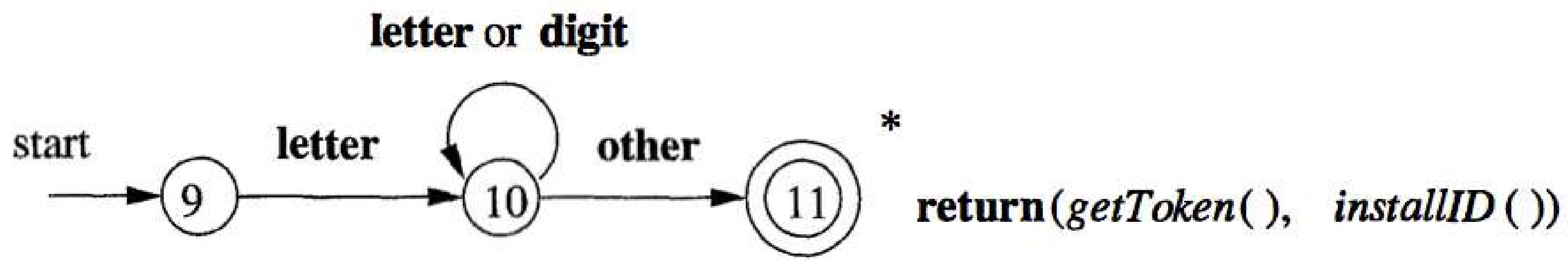
A More Complex Automaton



"1010": accept

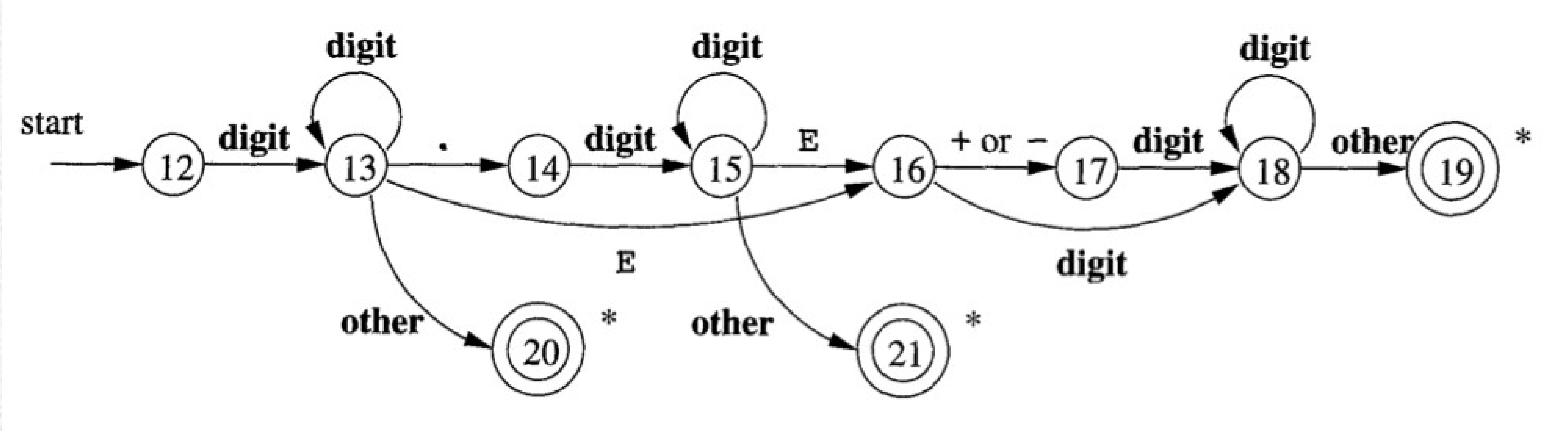
"101": reject

A More Complex Automaton



h	i	1	2	3

A More Complex Automaton



1	2	.	3	7	5

Finite Automata

- Finite automata is a recognizer
- Given an input string, they simply say "yes" or "no" about each possible input string

Nondeterministic Finite Automata (NFA)

- Definition: an NFA is a 5-tuple (S,Σ,δ,s_0,F) where
  - S is a finite set of *states*
  - Σ is a finite set of *input symbol alphabet*
  - δ is a *mapping* from $S\times \Sigma \cup \{\varepsilon\}$ to a set of states
  - $S_0 \subseteq S$ is the set of *start states*
  - $F \subseteq S$ is the set of *accepting* (or *final*) *states*

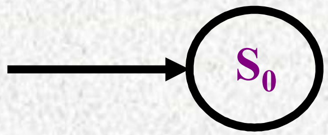
Nondeterministic Finite Automata (NFA)

- Transition Graph

Node: State

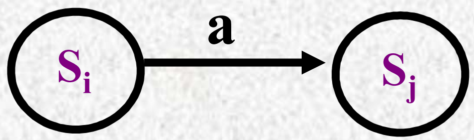
- Non-terminal state: 

- Terminal state: 

- Starting state: 

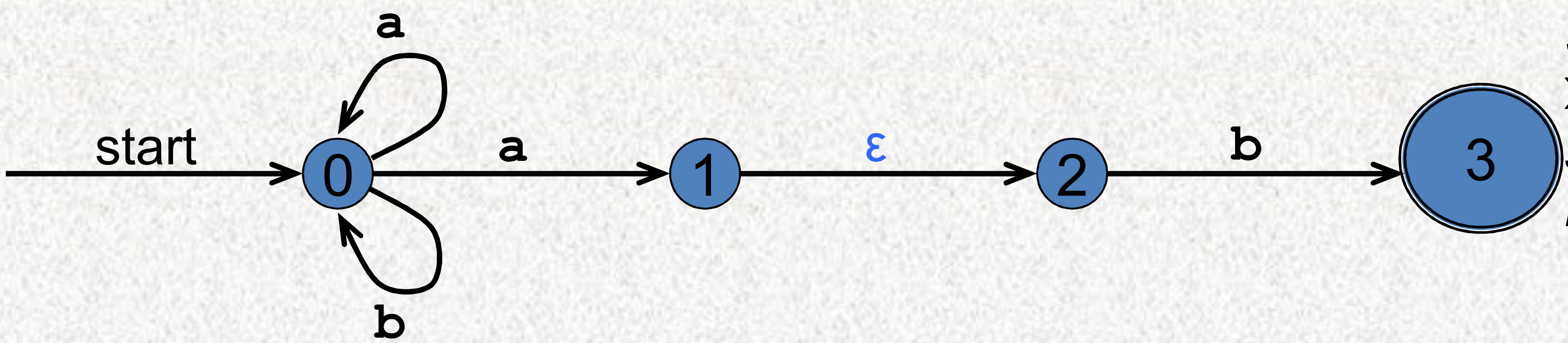
Edge: state transition

$f(S_i,a)=S_j$



Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*



$S = \{0,1,2,3\}$
$\Sigma = \{\mathbf{a},\mathbf{b}\}$
$s_0 = 0$
$F = \{3\}$

Nondeterministic Finite Automata (NFA)

- **Transit table**
  - Line： State
    - Starting state： in general, the first line， or label “+”；
    - Terminal state: “*” or “-”；
  - Column： All symbols in Σ
  - Cell： state transition mapping

Transition Table

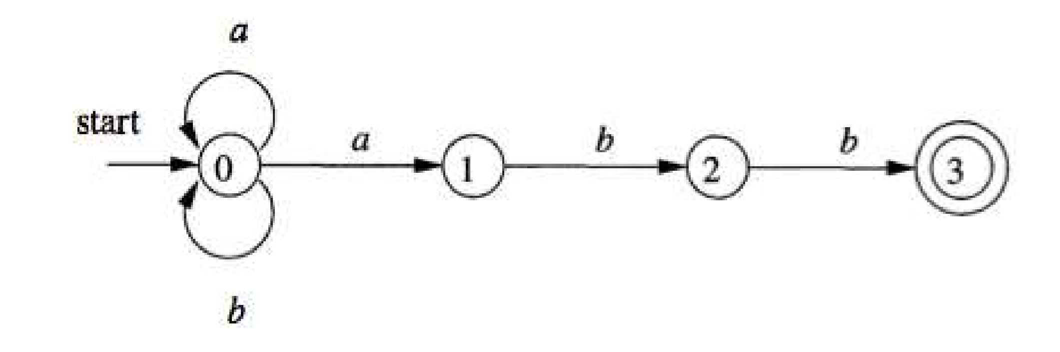
- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0,\mathbf{a}) = \{0,1\}$
$\delta(0,\mathbf{b}) = \{0\}$
$\delta(1,\mathbf{b}) = \{2\}$
$\delta(2,\mathbf{b}) = \{3\}$

→

State	*Input* **a**	*Input* **b**
0	{0,1}	{0}
1		{2}
2		{3}

NFA Example 2



Transition Table

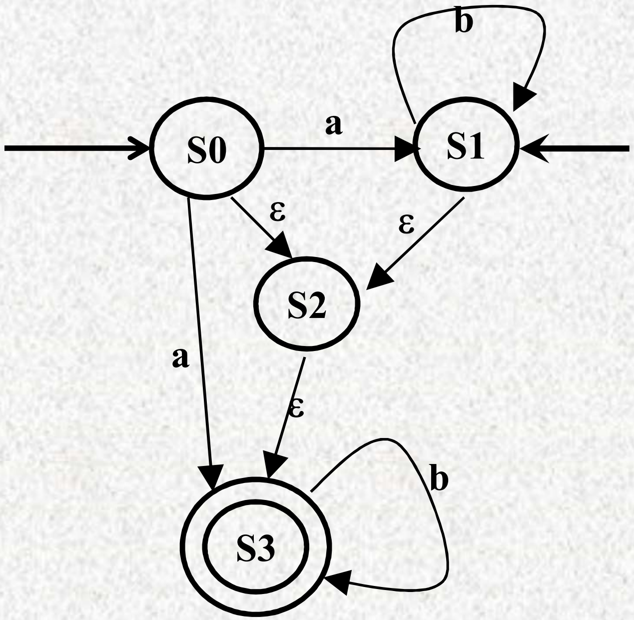
STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Acceptance of input strings

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

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

NFA Example 3



	a	b	ε
$S0^+$	**{S1,S3}**		{S2}
$S1^+$		{S1}	{S2}
S2			{S3}
$S3^-$		{S3}	

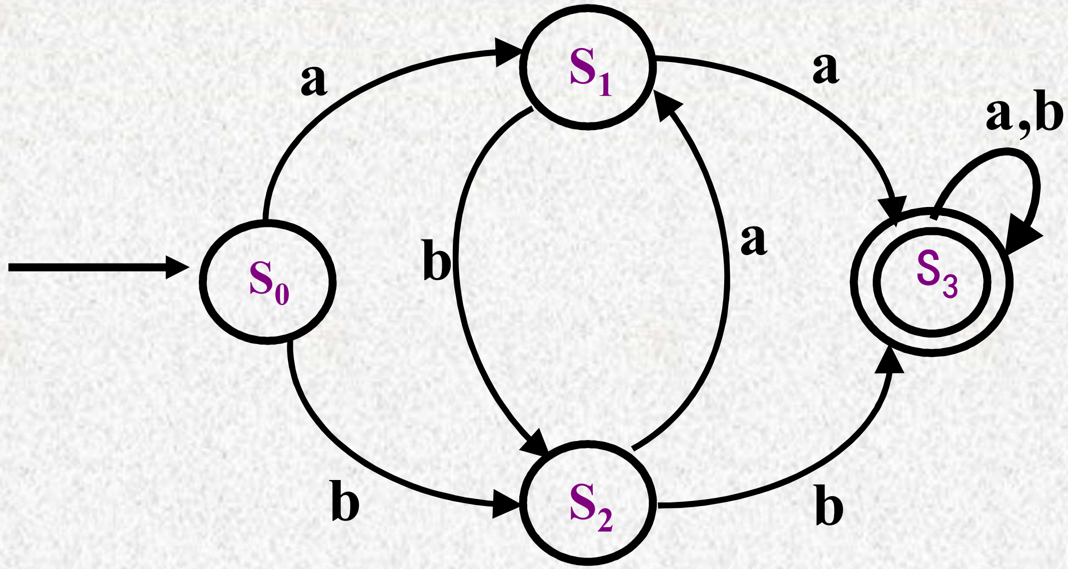
Deterministic Finite Automata (DFA)

- Definition: an DFA is a 5-tuple (S,Σ,δ,s_0,F), is a special case of NFA
  - There are no moves on input ε, and
  - For each state s and input symbol a, there is exactly one edge out of s labeled a.

Deterministic Finite Automata (DFA)

- DFA M=({S0, S1, S2, S3}, {a,b}, f, S0, {S3}), :

f (S0, a)=S1	f (S2, a)=S1
f (S0, b)=S2	f (S2, b)= S3
f (S1, a)= S3	f (S3, a)= S3
f (S1, b)= S2	f (S3, b)= S3



Deterministic Finite Automata (DFA)

- For example, DFA M=({0,1,2,3,4},{a,b}, δ,{0},{3})
- δ(0, a) = 1 δ (0, b) = 4

δ (1, a) = 4 δ (1, b) = 2

δ (2, a) = 3 δ (2, b) = 4

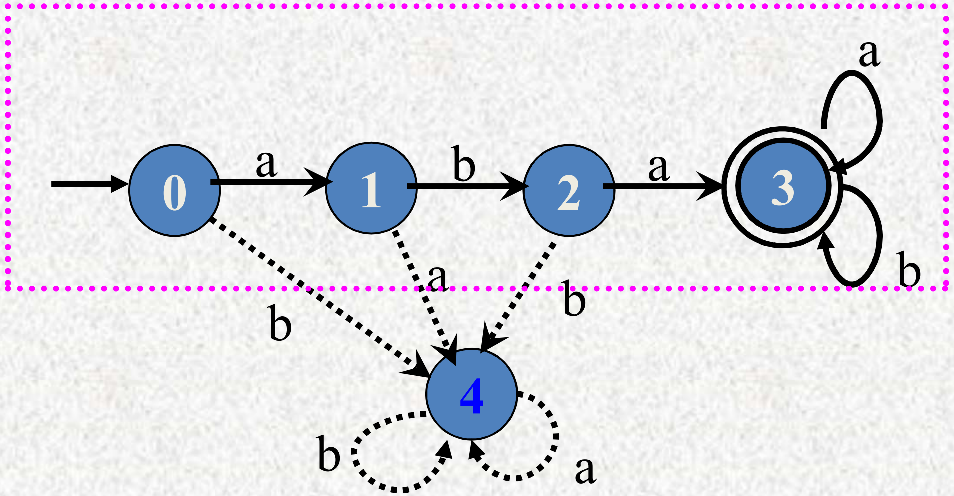
δ(3, a) = 3 δ (3, b) = 3

δ (4, a) = 4 δ (4, b) = 4

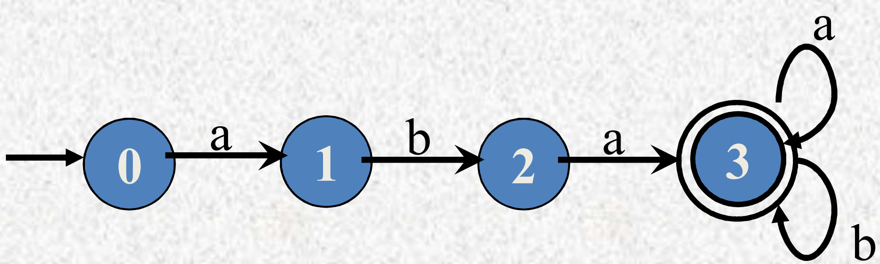
	a	b
0^+	1	4
1	4	2
2	3	4
3^-	3	3
4	4	4

Deterministic Finite Automata (DFA)

	a	b
0+	1	4
1	4	2
2	3	4
3⁻	3	3
4	4	4



Deterministic Finite Automata (DFA)



	a	b
0+	1	⊥
1	⊥	2
2	3	⊥
3⁻	3	3

⟷

	a	b
0+	1	
1		2
2	3	
3⁻	3	3

Deterministic Finite Automata (DFA)

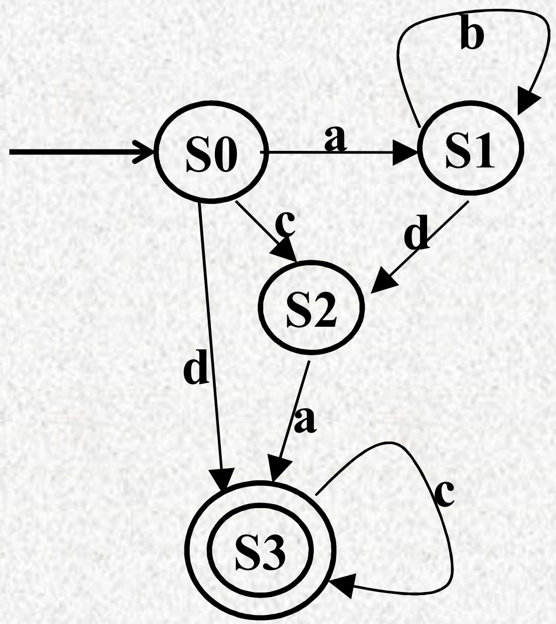
Σ: {a, b, c, d}

S: {S0, S1, S2, S3}

Start: S0

Terminal: {S3}

f: {(S0,a)→ S1, (S0,c)→S2,
(S0,d)→S3, (S1,b)→S1,
(S1,d)→S2, (S2,a)→S3,
(S3, c)→S3}



NFA v.s. DFA

NFA v.s. DFA

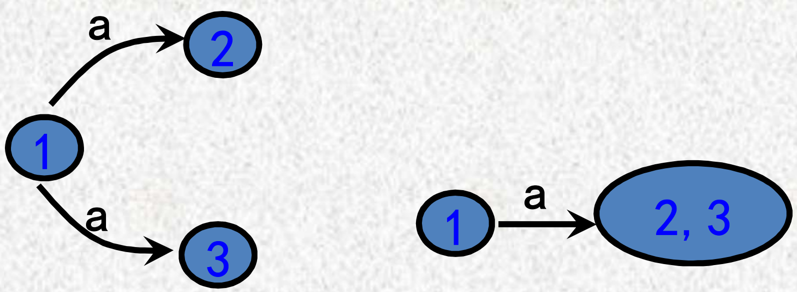
	DFA	NFA
Initial	Single starting state	A set of starting states
ε dege	Not allowed	Allowed
δ(S, a)	S' or ⊥	{S1, …, Sn} or ⊥
Implementation	Deterministic	Nondeterministic

- DFA accepts an input string with only one path
- NFA accepts an input string with possibly multiple paths

Construct DFA from NFA

- Construct DFA from NFA
  - For any NFA, there exists an equivalent DFA
  - Idea of construction: eliminate the uncertainty
  - Merge N states in NFA into one single state
    - Eliminate ε

    - Eliminate multiple mapping

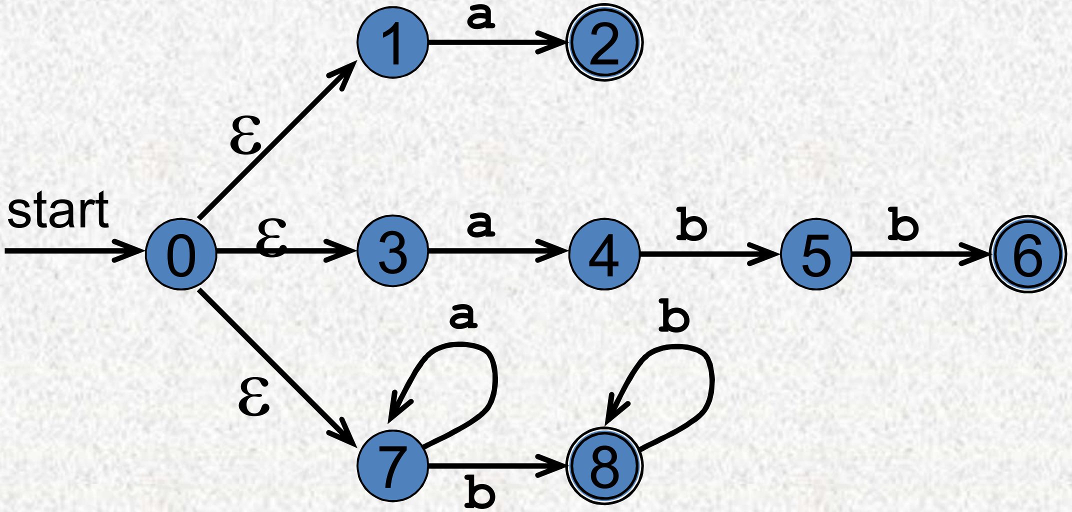


Construct DFA from NFA

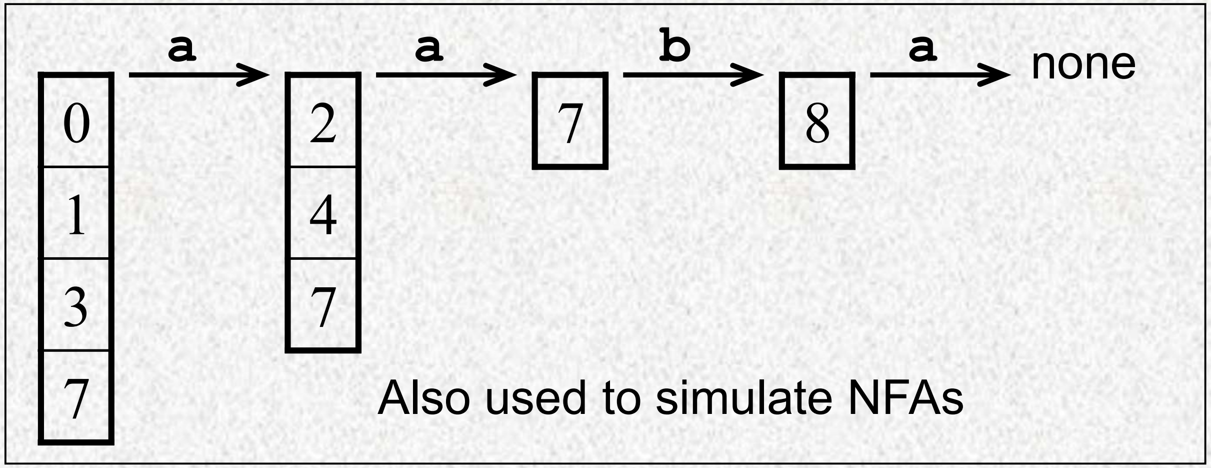
- **INPUT**: An NFA N.
- **OUTPUT**: A DFA D accepting the same language as N.
- **METHOD**: The algorithm constructs a transition table Dtran for D. Each state of D is a set of NFA states, and we construct Dtran so D will simulate "in parallel" all possible moves N can make on a given input string.

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.

ε-*closure* and *move* Examples



ε-*closure*({0}) = {0,1,3,7}
move({0,1,3,7},**a**) = {2,4,7}
ε-*closure*({2,4,7}) = {2,4,7}
move({2,4,7},**a**) = {7}
ε-*closure*({7}) = {7}
move({7},**b**) = {8}
ε-*closure*({8}) = {8}
move({8},**a**) = ∅



The Subset Construction Algorithm

- 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.

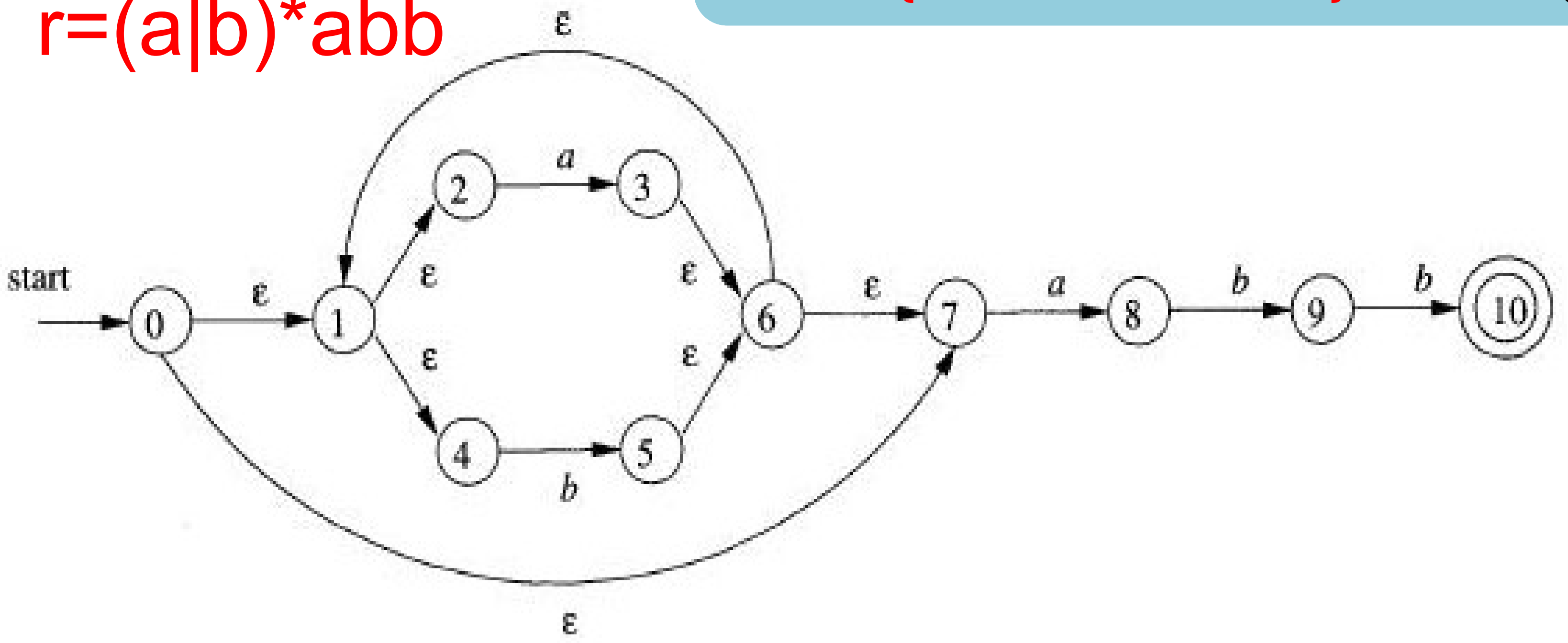
The Subset Construction Algorithm

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Subset Construction Example 1

First，Initial state of NFA is ε-closure(0)，
i.e. A={0，1，2，4，7}， Σ = {a,b}

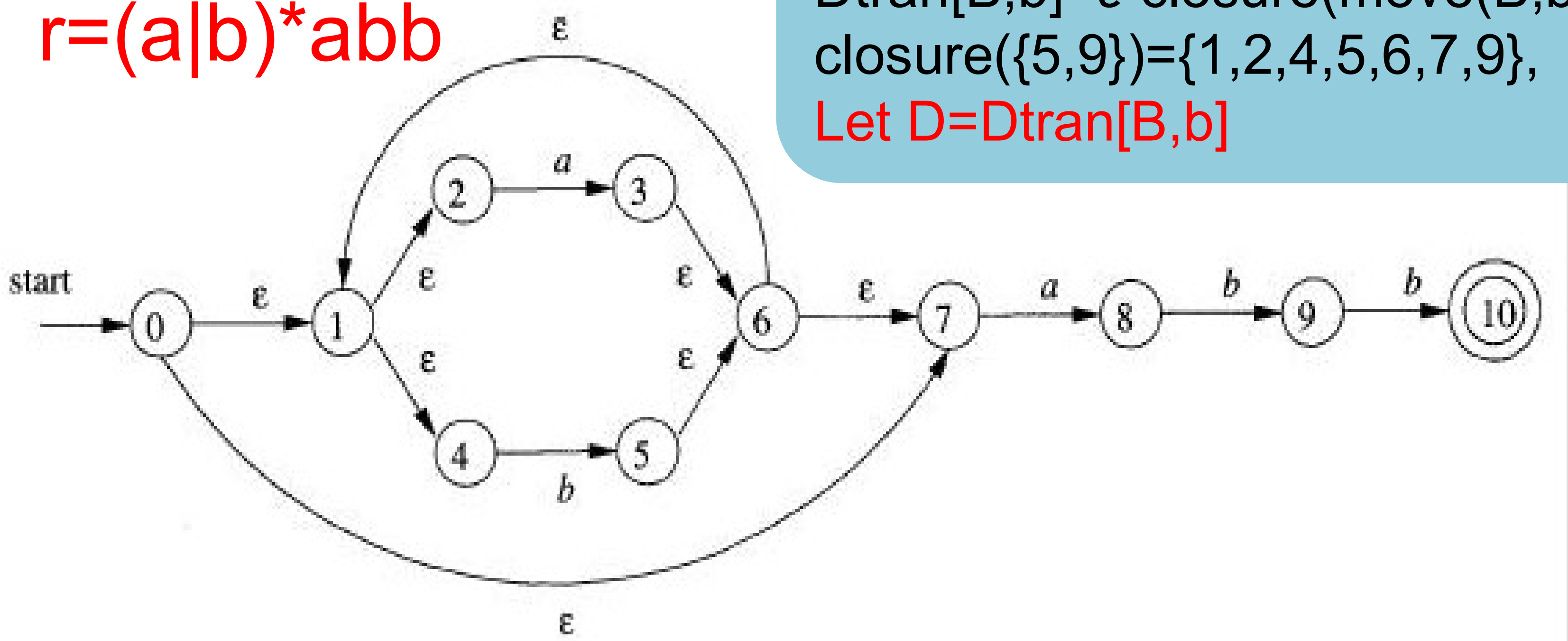
r=(a|b)*abb



Dtran[A,a]=ε-closure(move(A,a))=ε-closure({3,8})={1,2,3,4,6,7,8},
Let B=Dtran[A,a]
Dtran[A,b]=ε-closure(move(A,b))=ε-closure({5})={1,2,4,6,7},
Let C=Dtran[A,b]

Subset Construction Example 1

r=(a|b)*abb

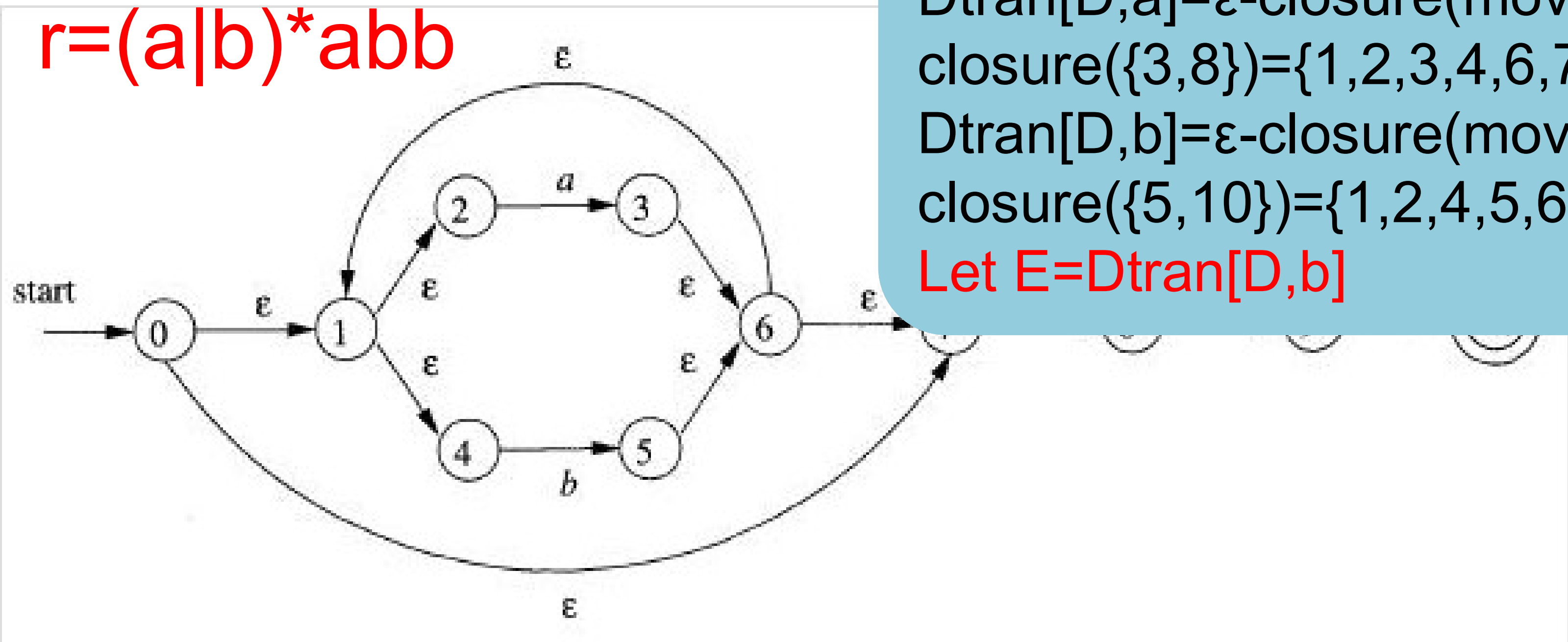


Dtran[B,a]=ε-closure(move(B,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[B,b]=ε-closure(move(B,b))=ε-closure({5,9})={1,2,4,5,6,7,9},
Let D=Dtran[B,b]

Dtran[C,a]=ε-closure(move(C,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[C,b]=ε-closure(move(C,b))=ε-closure({5})={1,2,4,6,7}=C

Subset Construction Example 1

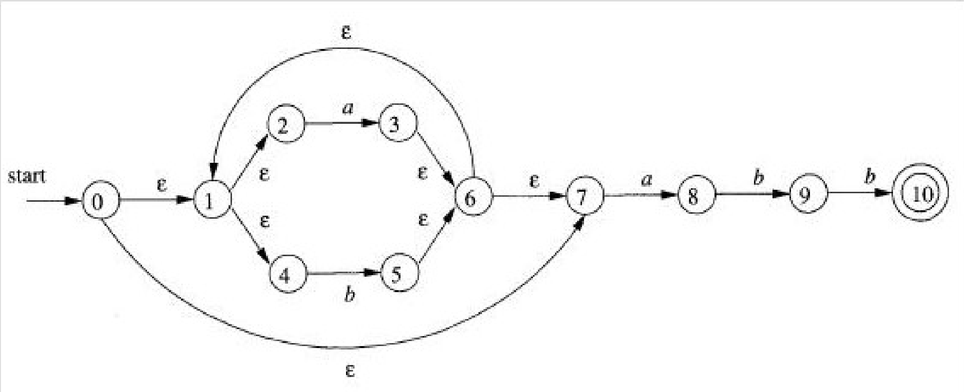
r=(a|b)*abb



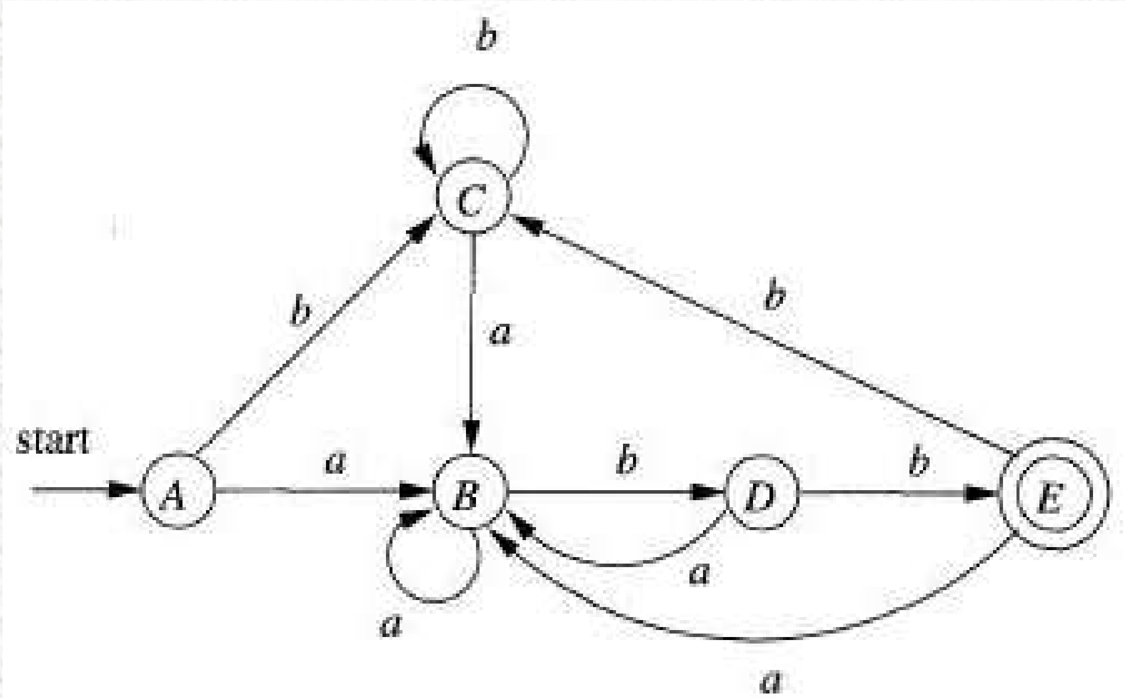
Dtran[D,a]=ε-closure(move(D,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[D,b]=ε-closure(move(D,b))=ε-closure({5,10})={1,2,4,5,6,7,10},
Let E=Dtran[D,b]

Dtran[E,a]=ε-closure(move(E,a))=ε-closure({3,8})={1,2,3,4,6,7,8}=B
Dtran[E,b]=ε-closure(move(E,b))=ε-closure({5})={1,2,4,6,7}=C

Subset Construction Example 1



NFA STATE	DFA STATE	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 3, 5, 6, 7, 10}	E	B	C



Homework-W2

Homework – week 2

- pp.125, Exercise 3.3.2 (a)(c), 3.3.5 (a)(e)
- pp.151-152, Exercise 3.6.3, Exercise 3.6.4
- pp.152, Exercise 3.6.5
- pp. 166, Exercise 3.7.1 (b)