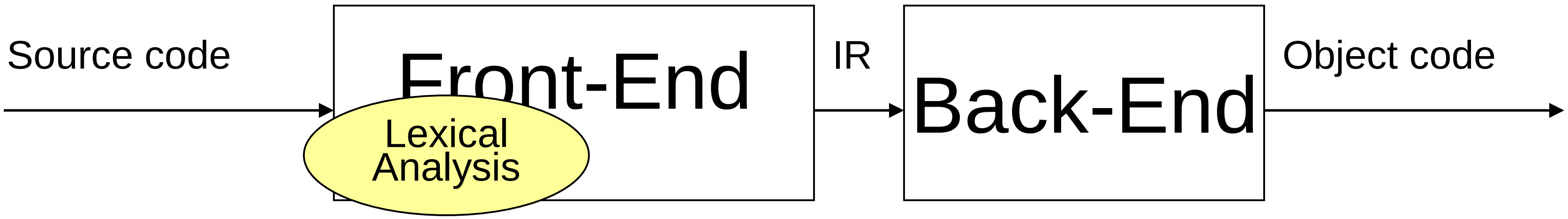
Compiler Design

Lecture 3:
Lexical Analysis

Dr. Momtazi
momtazi@aut.ac.ir

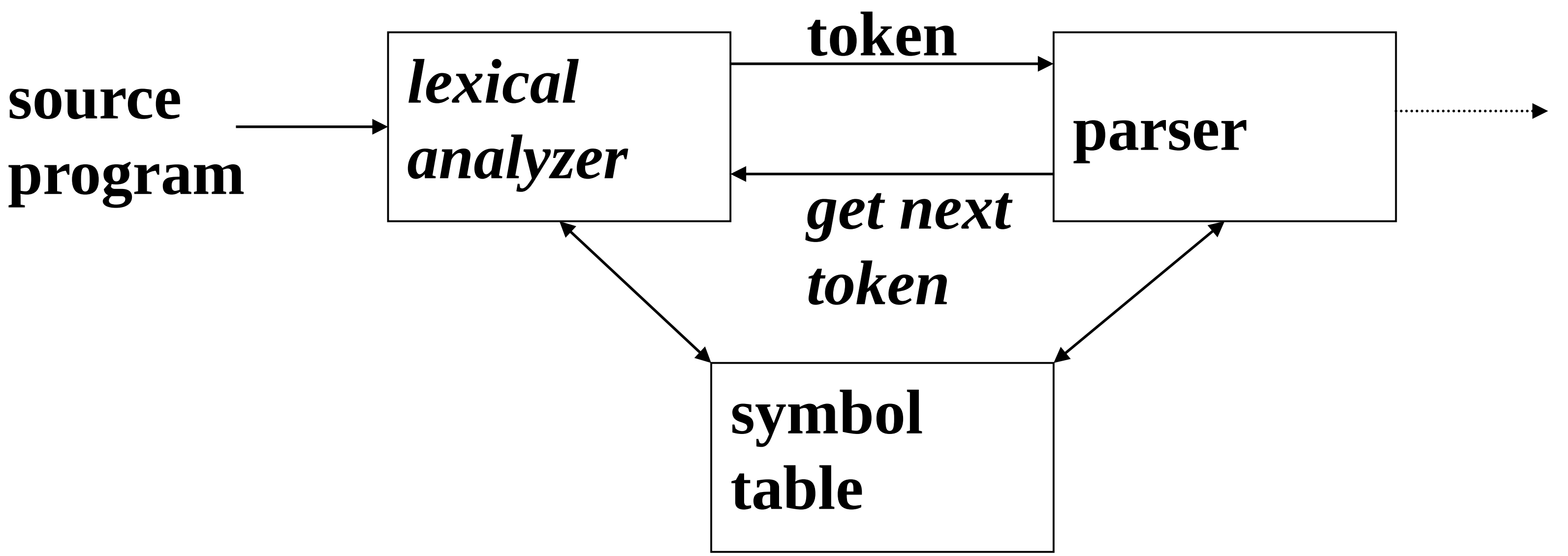
based on the slides of the course book

Lexical Analyzer in Perspective



- Lexical Analysis:
  - Reading characters and producing sequences of tokens.

Lexical Analyzer in Perspective



- Important issue:
  - What are responsibilities of each box ?
  - Focus on lexical analyzer and parser

Why to Separate Lexical Analysis and Parsing

- Simplicity of design
- Improving compiler efficiency
- Enhancing compiler portability

Outline

- **Definition**
- Associating Lexemes with Tokens
- Matching Regular Expressions
- From RE to Automata
- Real-world Application
- Error Recovery
- Toward Automation

General Definition

- First step in any translation:
  - Determine whether the text to be translated is well constructed in terms of the input language.
  - Syntax is specified with parts of speech - syntax checking matches parts of speech against a grammar.
- In natural languages, mapping words to part of speech is idiosyncratic.
- In formal languages, mapping words to part of speech is syntactic.
  - Reserved keywords are important

Some Definitions

- A token is a pair a token name and an optional token attribute
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Why Lexical Analysis?

- We want to specify **lexical patterns** (to derive tokens):
  - Some parts are easy:
    - *WhiteSpace* $\rightarrow$ *blank* | *tab* | *WhiteSpace blank* | *WhiteSpace tab*
    - Keywords and operators (if, then, =, +)
    - Comments (/* followed by */ in C, // in C++, % in latex, ...)
  - Some parts are more complex:
    - Identifiers (letter followed by - up to *n* - alphanumerics…)
    - Numbers

We need a notation that could lead to an implementation!

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
relation	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ surrounded by “	“core dumped”

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

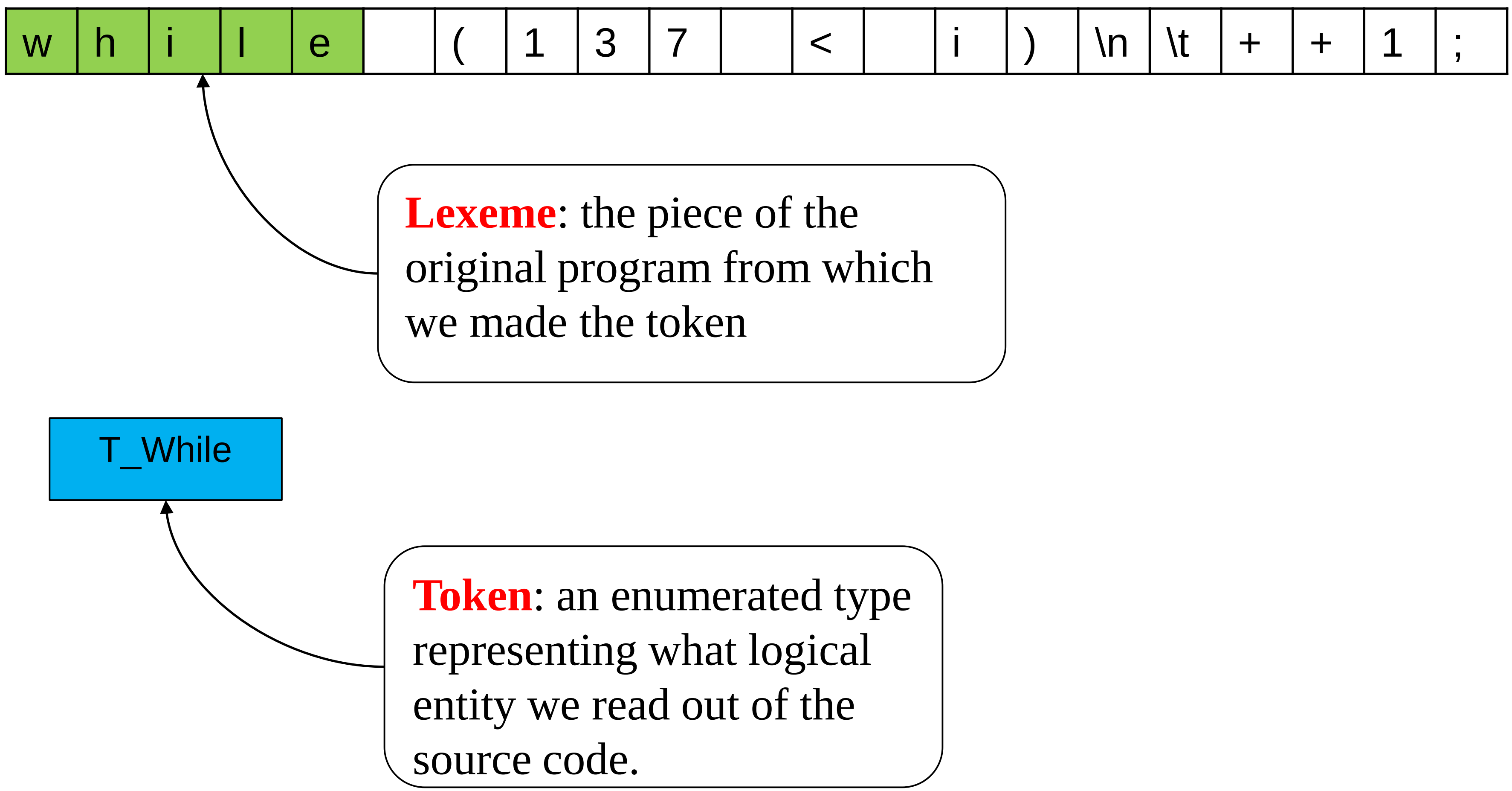
Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File



Scanning a Source File

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

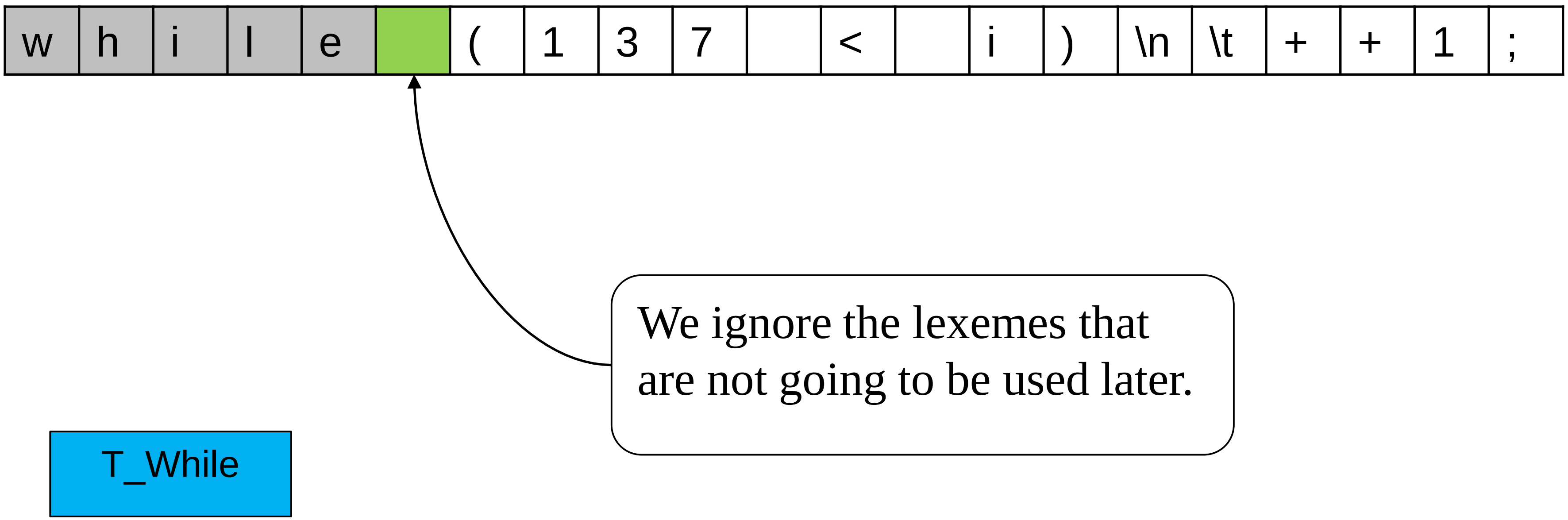
T_While

Scanning a Source File

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

T_While

Scanning a Source File



Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

(

Scanning a Source File

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

T_While

(

Scanning a Source File

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

T_While

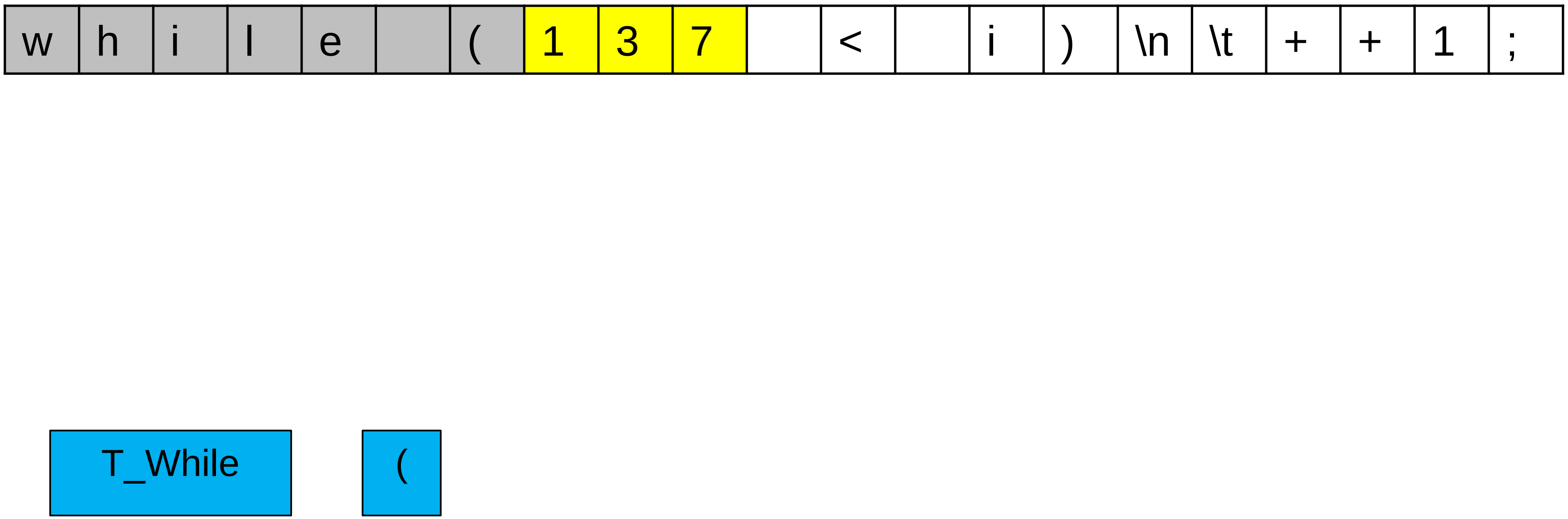
(

Scanning a Source File

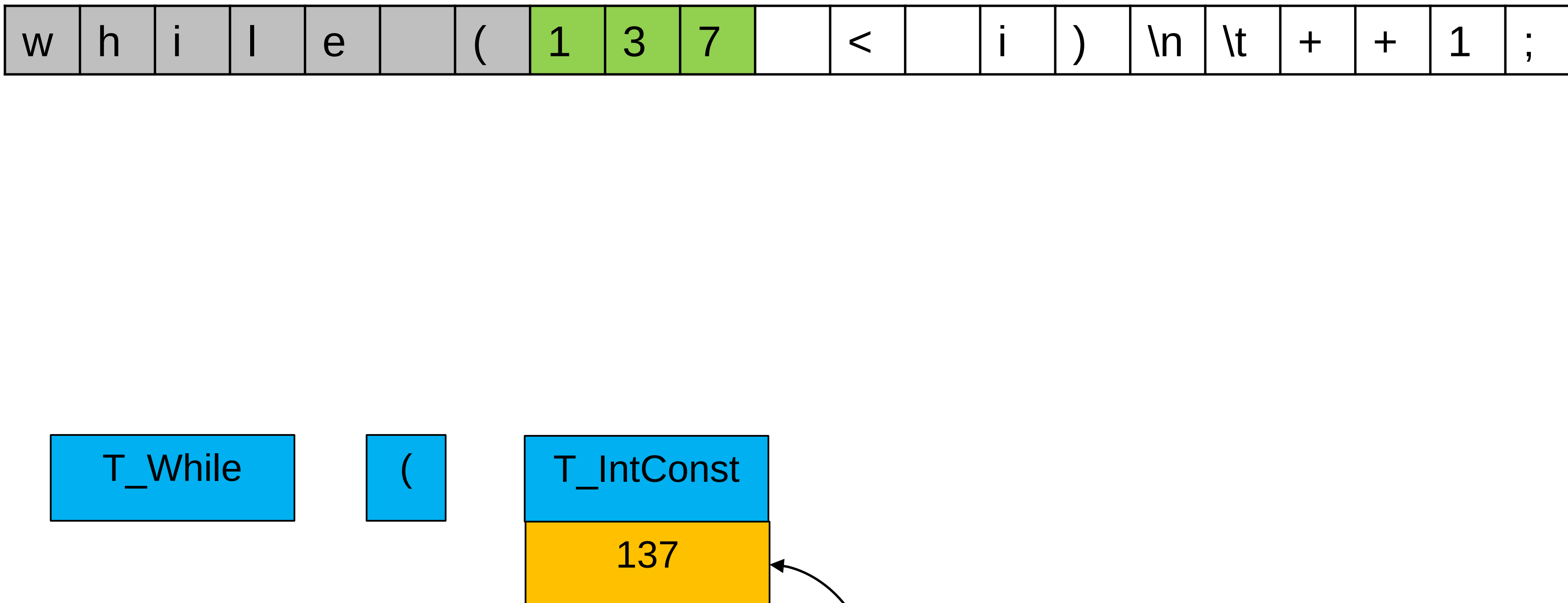
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	1	;

T_While	(

Scanning a Source File



Scanning a Source File



Some tokens can have **attributes** that store extra information about the token.

Tokenizer

- 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 Difficulties

- FORTRAN: Whitespace is irrelevant

```
DO 5 I  =  1.25
```

Scanning Difficulties

- FORTRAN: Whitespace is irrelevant

```
DO 5 I  =  1.25
  DO5I  =  1.25
```

- Can be difficult to tell when to partition input.

Scanning Difficulties

- C++: Nested template declarations

vector<vector<int>> myVector

Scanning Difficulties

- C++: Nested template declarations

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

- Again, can be difficult to determine where to split.

Scanning Difficulties

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

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

Scanning Difficulties

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

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

- Can be difficult to determine how to label lexemes.

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?

Outline

- Definition
- **Associating Lexemes with Tokens**
- Matching Regular Expressions
- From RE to Automata
- Real-world Application
- Error Recovery
- Toward Automation

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** }.

Lexeme construction

- 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.
- Over the course of this class, we will use all of these approaches.

Some Definitions

- A context-free grammar, G, is a 4-tuple, $G=(S,N,T,P)$, where:

    S: starting symbol

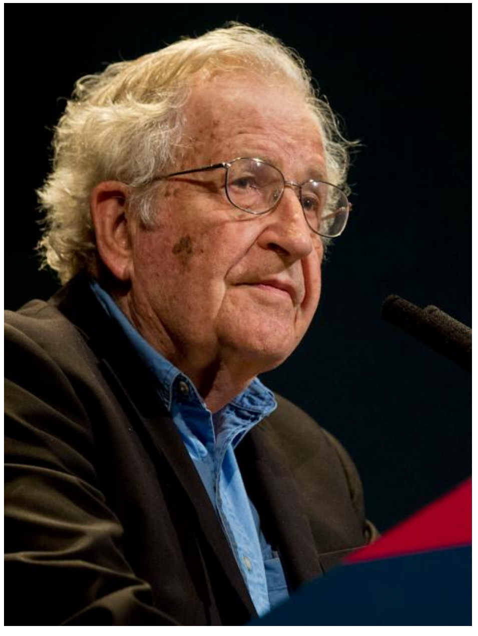
    N: set of non-terminal symbols

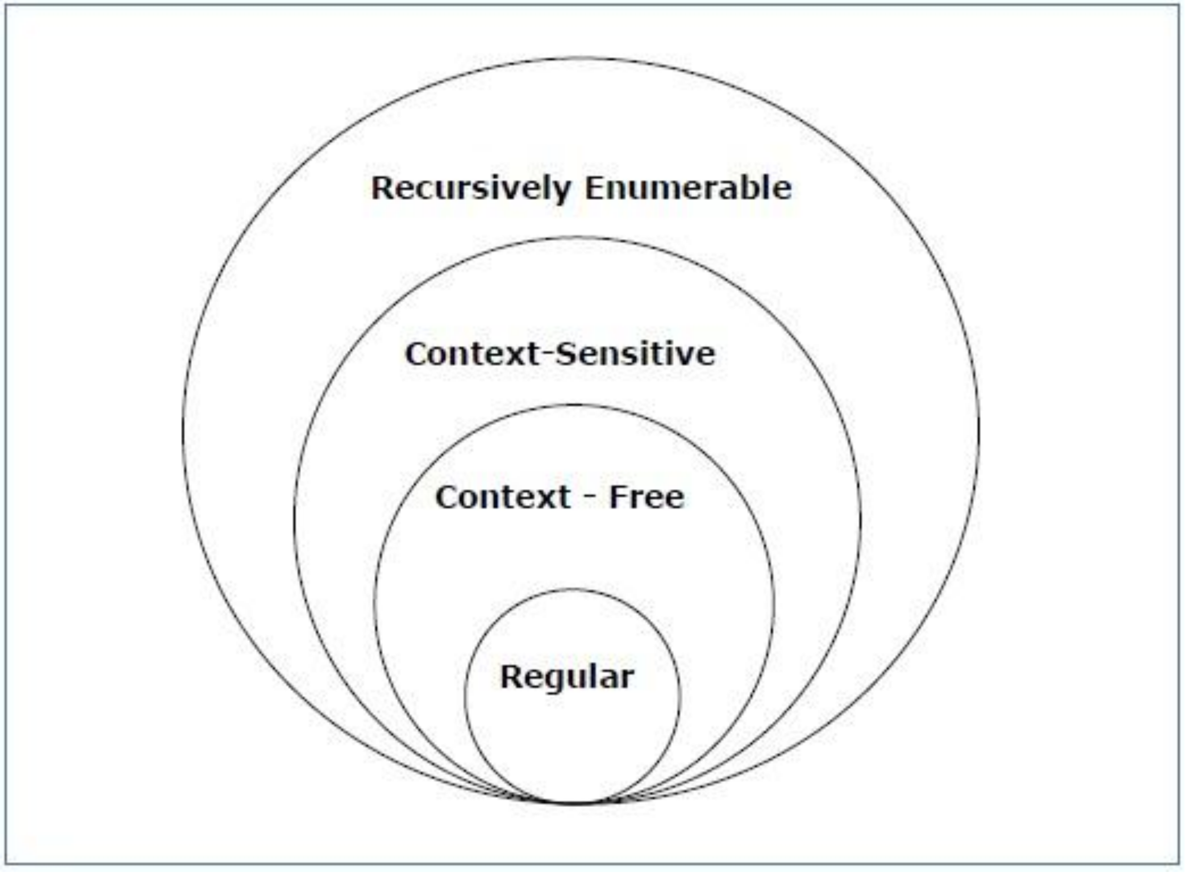
    T: set of terminal symbols

    P: set of production rules

- A language is the set of all terminal productions of G.

Chomsky Hierarchy





Chomsky Hierarchy

1. Unrestricted $\alpha \rightarrow \beta$
2. Context-Sensitive $\alpha A\beta \rightarrow \alpha\,\gamma\beta$
3. Context-Free $A \rightarrow \gamma$
4. Regular $A \rightarrow a \mid aB$ or

   $A \rightarrow a \mid Ba$

Chomsky Hierarchy

Unrestricted	Turing machine
Context-Sensitive	Linear-bounded non-deterministic Turing machine
Context-Free	Non-deterministic pushdown automaton
Regular	Finite state automaton

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, including the **flex** tool we will use in this course.

Language & Regular Expressions

- A Regular expression is a set of rules / techniques for constructing sequences of symbols (strings) from an alphabet.

- Let Σ be an alphabet, r a regular expression. Then L(r) is the language that is characterized by the rules of r.

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**.

Regular Expressions

- **Regular Expression** (RE) (over a vocabulary V):
  - ε is a RE denoting the empty set $\{\varepsilon\}$.
  - If $a \in$ V then a is a RE denoting $\{a\}$.
  - If r_1, r_2 are REs then:
    - r_1* denotes zero or more occurrences of r_1;
    - $r_1 r_2$ denotes concatenation;
    - $r_1 \mid r_2$ denotes either r_1 or r_2;
  - Or more compact as:
    - *[a-d]* for $a \mid b \mid c \mid d$;
    - r^+ for rr*;
    - r? for $r \mid \varepsilon$

Operator Precedence

- Regular expression operator precedence is

(R)

R*

R1R2

R1 | R2

- Example: how to parse **ab*c|d** ?

Operator Precedence

- Regular expression operator precedence is

(R)

R*

R1R2

R1 | R2

- Example: how to parse **ab*c|d** ?

((a(b*))c)|d

Formal Language Operations

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
concatenation of L and M written LM	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L written L*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L* denotes "zero or more concatenations of " L
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of " L

Formal Language Operations Examples

$L = \{A, B, C, D\}$ **$D = \{1, 2, 3\}$**

$L \cup D =$

$LD =$

$L^2 =$

Formal Language Operations Examples

$L = \{A, B, C, D\}$ $\quad$ $D = \{1, 2, 3\}$

$L \cup D = \{A, B, C, D, 1, 2, 3\}$

$LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$

$L^2 = \{AA, AB, AC, AD, BA, BB, BC, BD, CA, \ldots DD\}$

Formal Language Operations Examples

$L = \{A, B, C, D\}$ $D = \{1, 2, 3\}$

$L \cup D = \{A, B, C, D, 1, 2, 3\}$

$LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}$

$L^2 = \{ AA, AB, AC, AD, BA, BB, BC, BD, CA, \ldots DD\}$

$L^4 = L^2 \; L^2 = ??$

$L^* = \{$ All possible strings of L plus $\in \}$

$L^+ = L^* - \in$

$L\,(L \cup D) = ??$

$L\,(L \cup D)^* = ??$

Examples

- $integer \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid 1 \mid 2 \mid \ldots \mid 9)+$
- $integer \rightarrow (+ \mid - \mid \varepsilon)\ (0 \mid (1 \mid 2 \mid \ldots \mid 9)\ (0 \mid 1 \mid 2 \mid \ldots \mid 9)^*)$
- $decimal \rightarrow integer.(0 \mid 1 \mid 2 \mid \ldots \mid 9)^*$
- $identifier \rightarrow [a\text{-}zA\text{-}Z]\ [a\text{-}zA\text{-}Z0\text{-}9]^*$

Not all languages can be described by regular expressions. But, we don't care for now.

Regular Expressions Example

$$(0 \mid 1)^*00(0 \mid 1)^*$$

Regular Expressions Example

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

- A regular expression for strings containing **00** as a substring:

11011100101

0000

111111011110011111

Regular Expressions Example

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

Regular Expressions Example

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

- A regular expression for strings of length exactly four:

0000

1010

1111

1000

Regular Expressions Example

- A regular expression for strings that contain at most one zero:

Regular Expressions Example

- A regular expression for strings that contain at most one zero:

11110111

111111

0111

0

Regular Expressions Example

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

- A regular expression for strings that contain at most one zero:

11110111

111111

0111

0

Regular Expressions Example

1*0?1*

- A regular expression for strings that contain at most one zero:

11110111

111111

0111

0

Regular Expressions Example

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

Regular Expressions Example

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

cs143@cs.stanford.edu

first.middle.last@mail.site.org

barack.obama@whitehouse.gov

Regular Expressions Example

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

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

cs143@cs.stanford.edu

first.middle.last@mail.site.org

barack.obama@whitehouse.gov

Regular Expressions Example

a+(.a+)*@a+(.a+)+

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

cs143@cs.stanford.edu

first.middle.last@mail.site.org

barack.obama@whitehouse.gov

Regular Expressions Example

- A regular expression for even numbers:

Regular Expressions Example

- A regular expression for even numbers:

42

+1370

-3248

-9999912

Regular Expressions Example

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

- A regular expression for even numbers:

42

+1370

-3248

-9999912

Regular Expressions Example

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

- A regular expression for even numbers:

42

+1370

-3248

-9999912

Regular Expressions Example

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

- A regular expression for even numbers:

42

+1370

-3248

-9999912

Outline

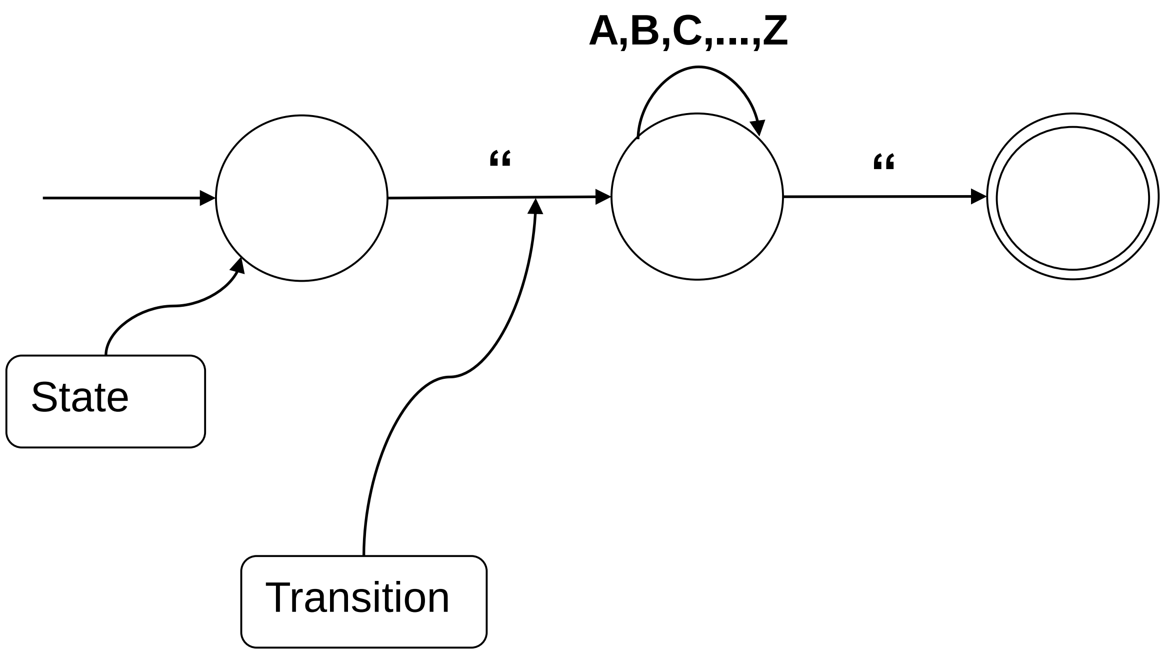
- Definition
- Associating Lexemes with Tokens
- **Matching Regular Expressions**
- From RE to Automata
- Real-world Application
- Error Recovery
- Toward Automation

Implementing Regular Expressions

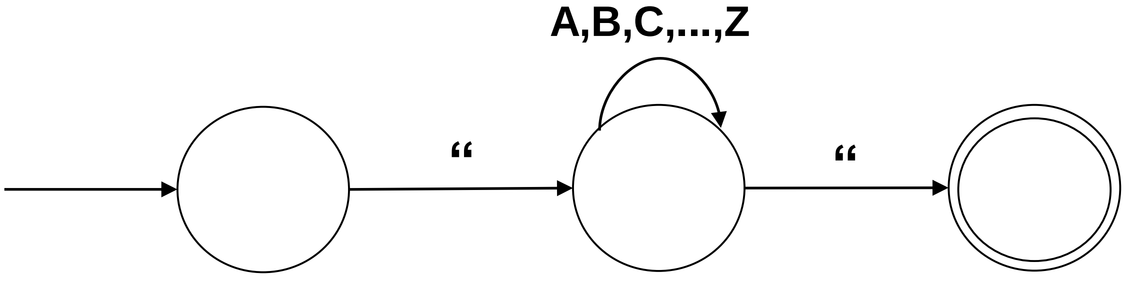
- Regular expressions can be implemented using **finite automata**.
- There are two main kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata)
  - **DFA**s (**deterministic** finite automata)

- Automata are best explained by example...

A Simple Automaton

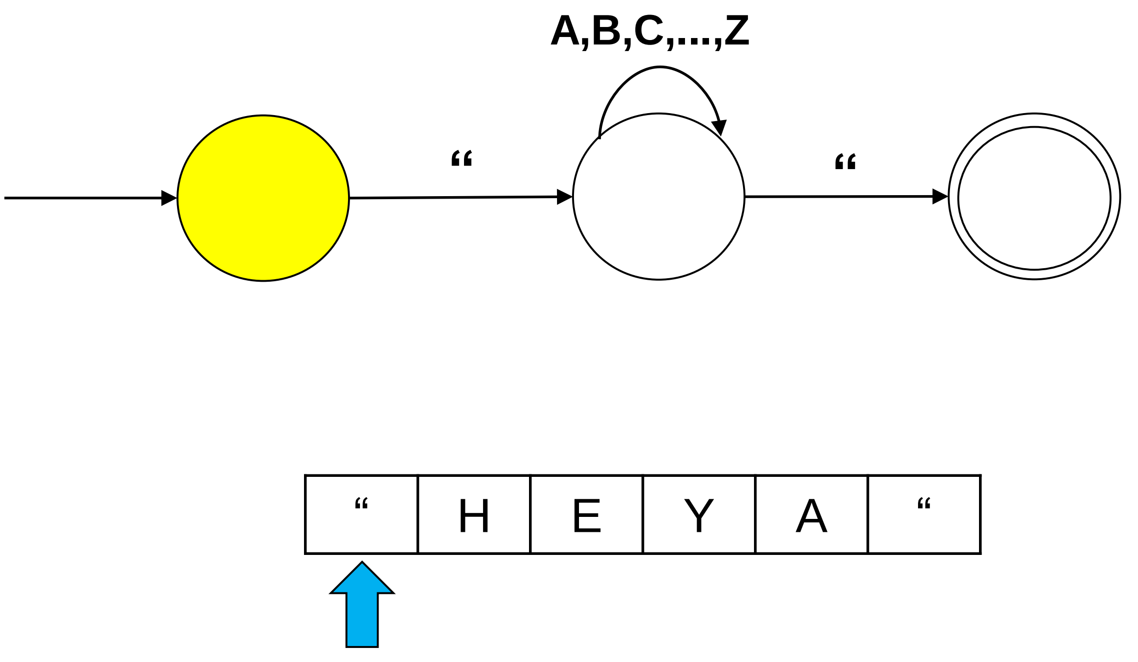


A Simple Automaton

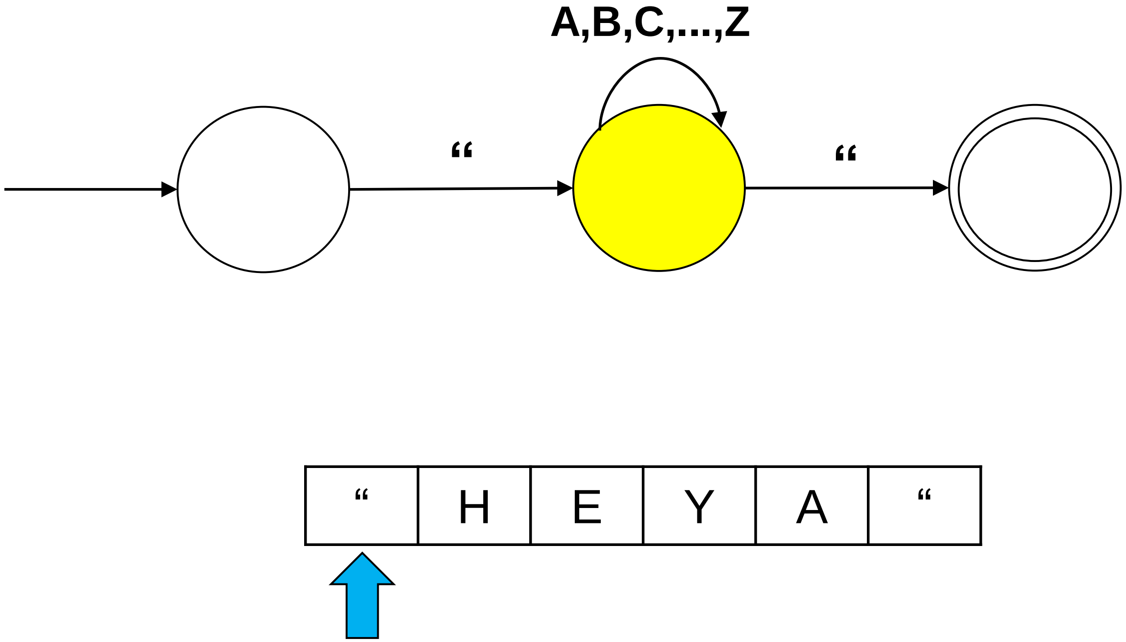


“	H	E	Y	A	“

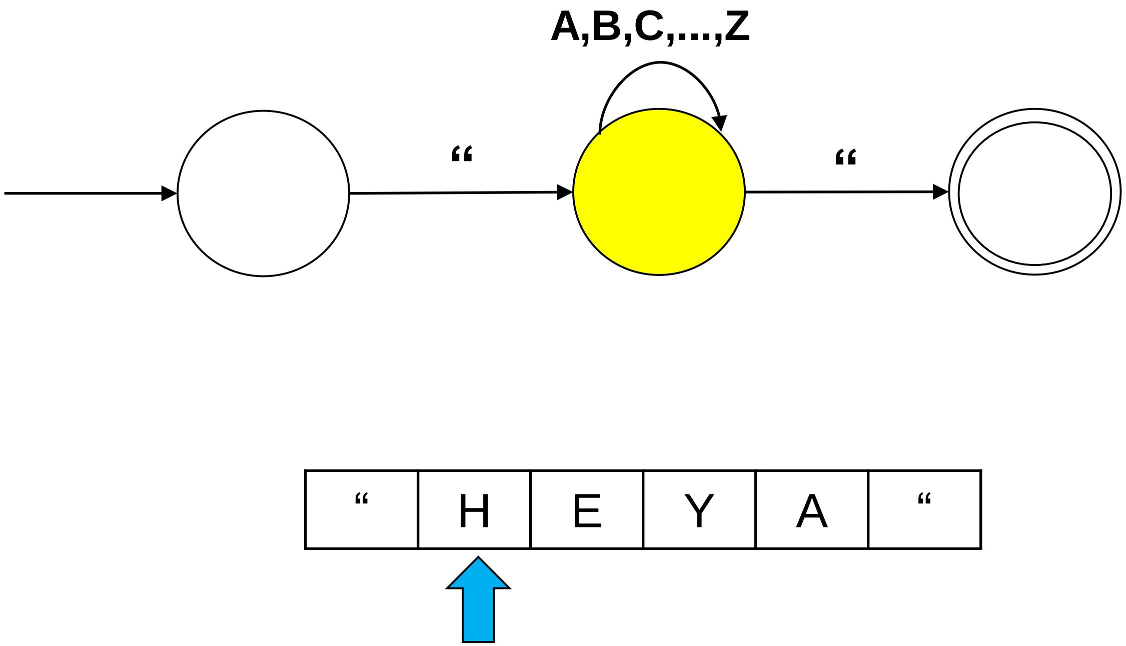
A Simple Automaton



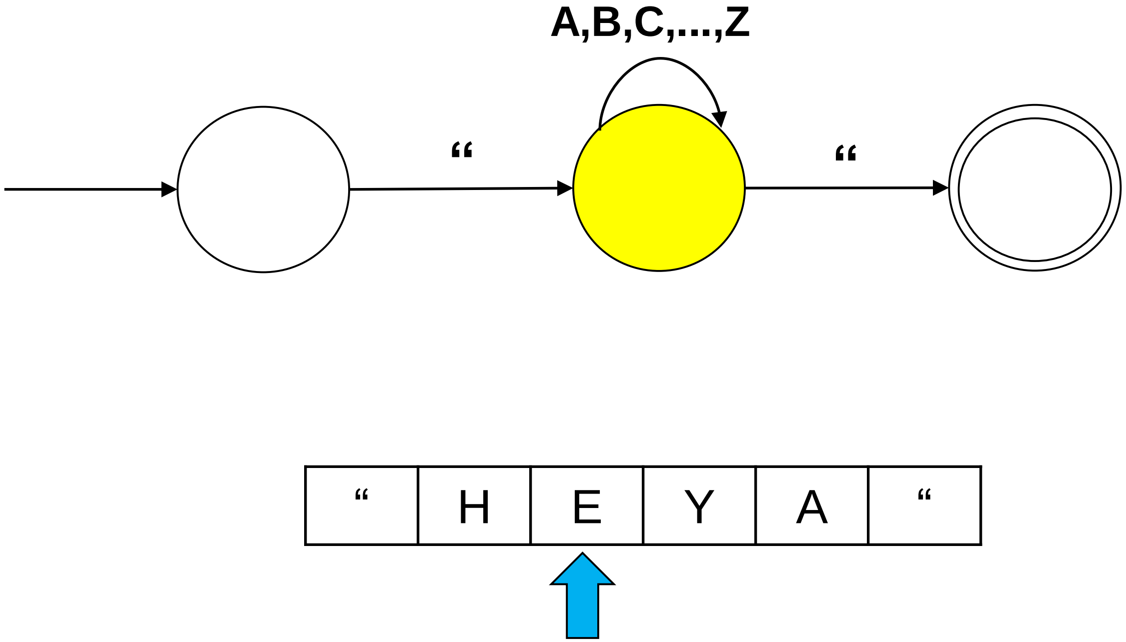
A Simple Automaton



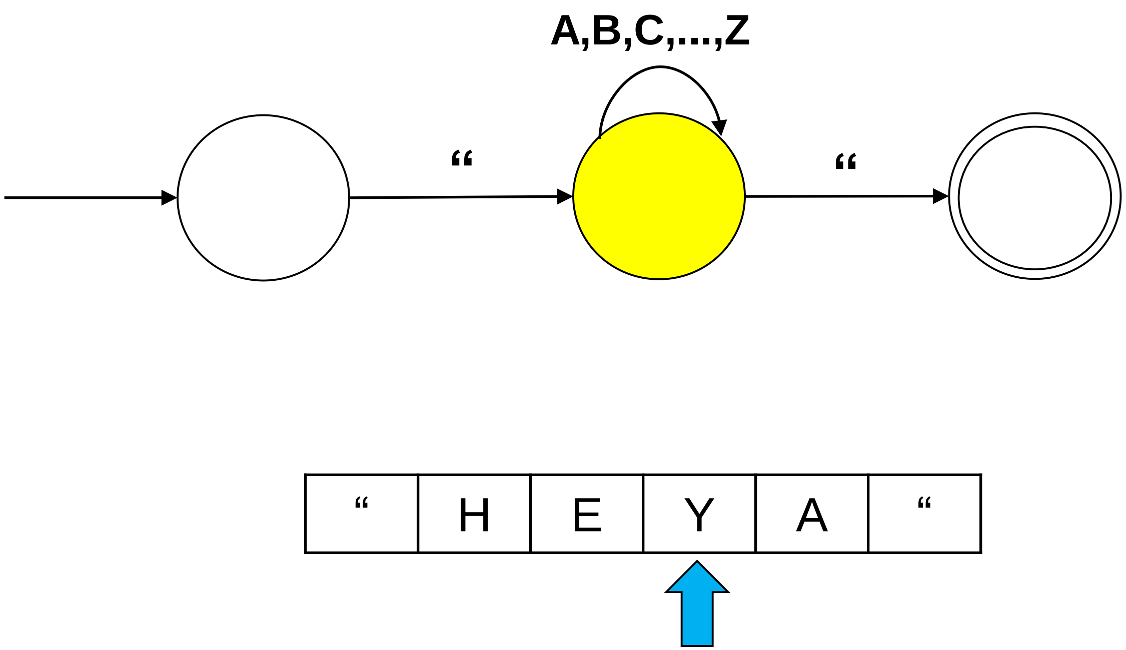
A Simple Automaton



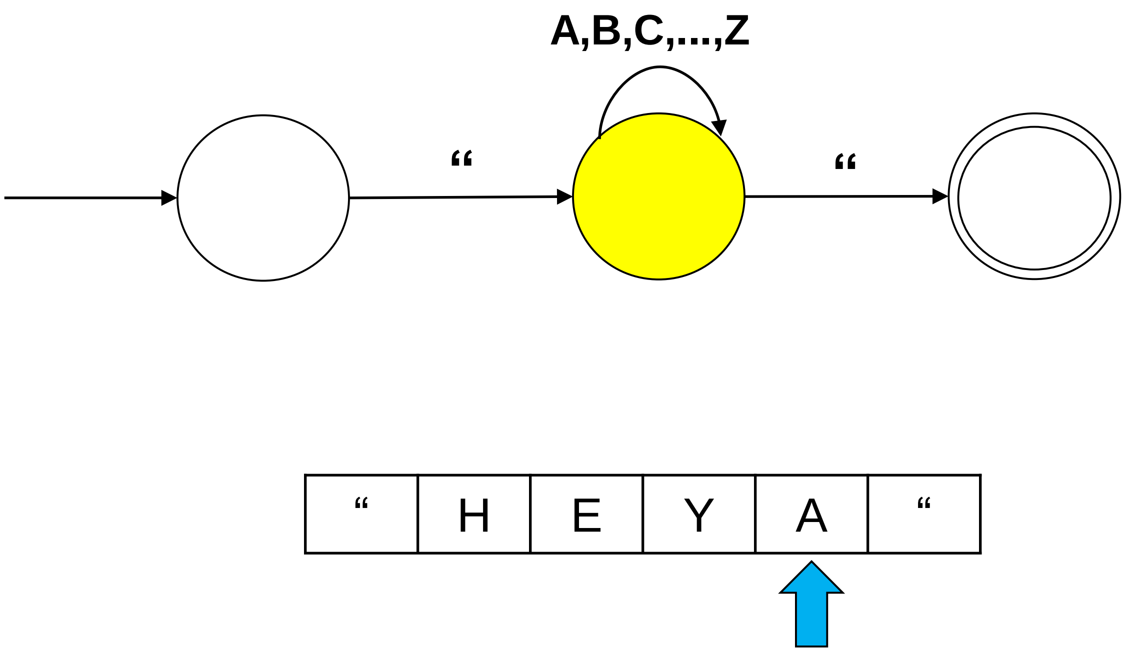
A Simple Automaton



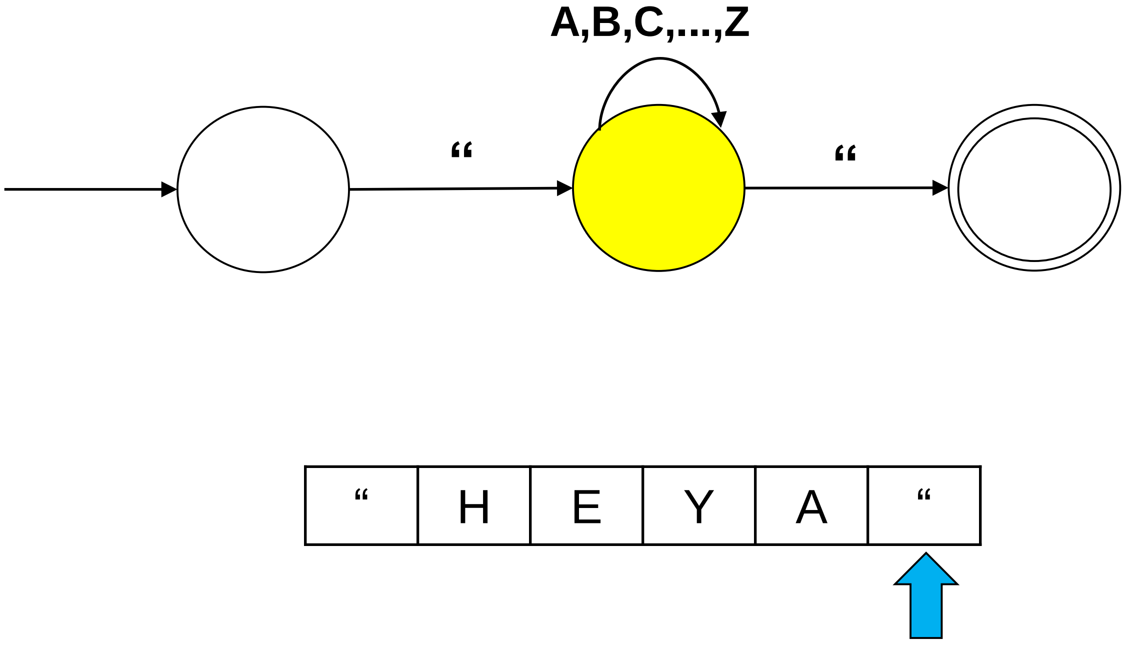
A Simple Automaton



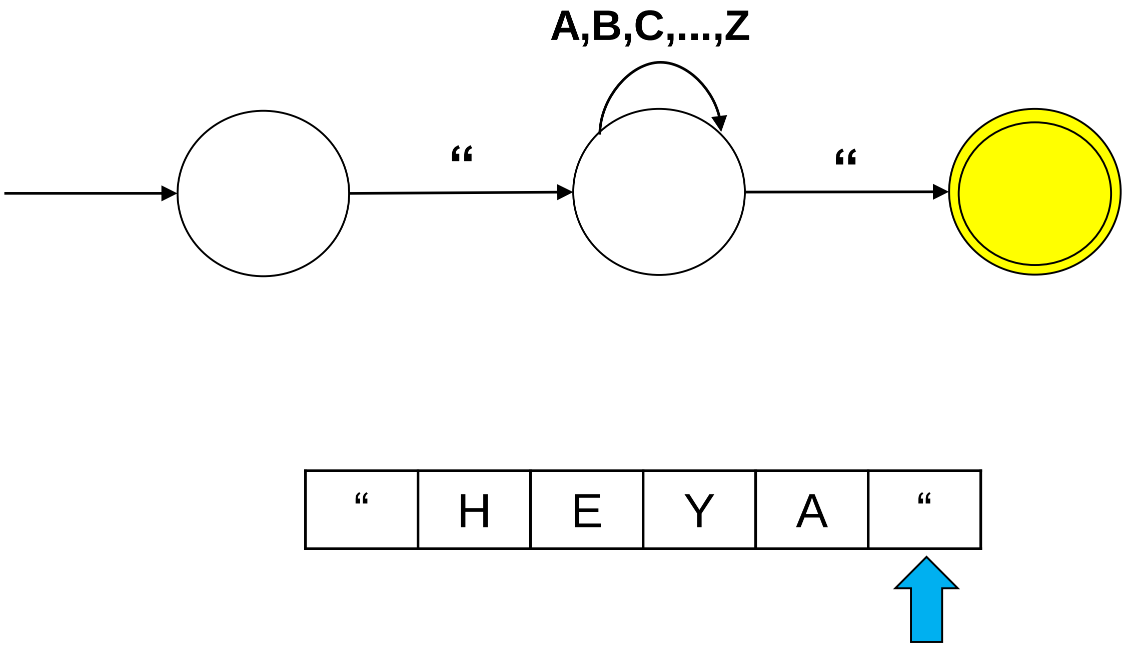
A Simple Automaton



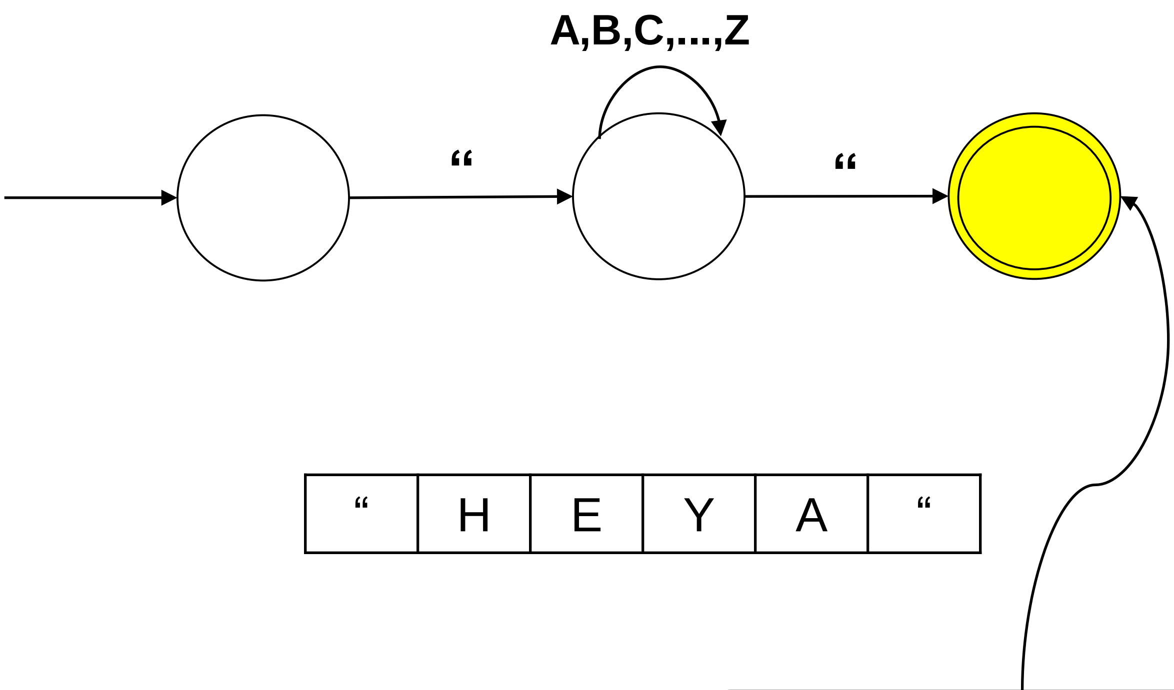
A Simple Automaton



A Simple Automaton

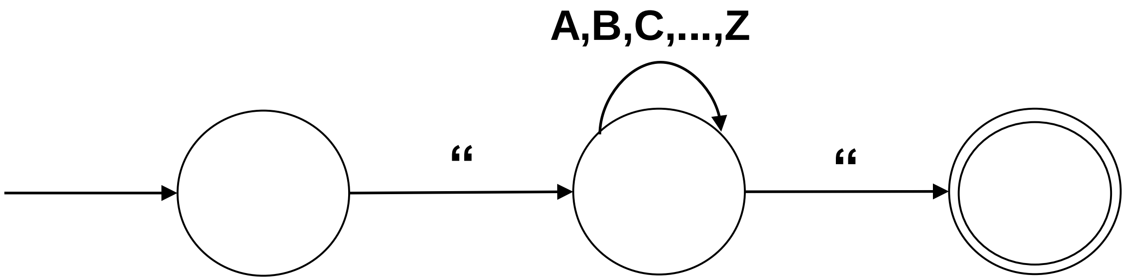


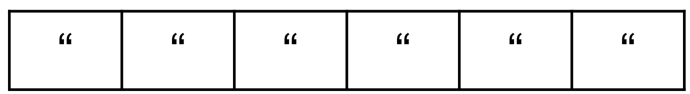
A Simple Automaton



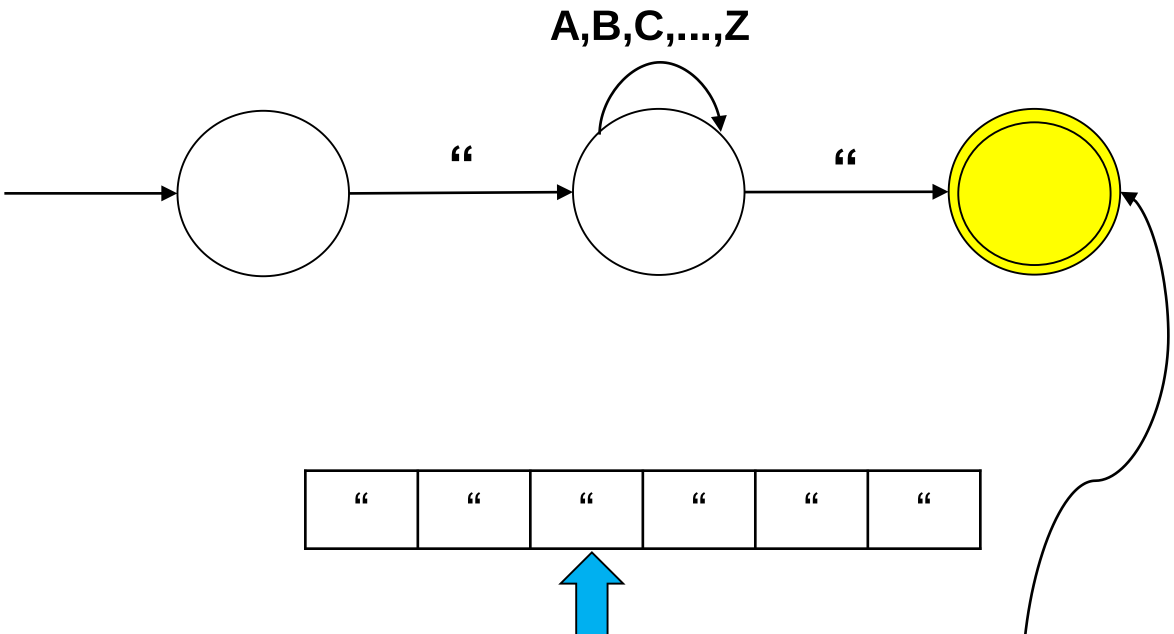
The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.

A Simple Automaton



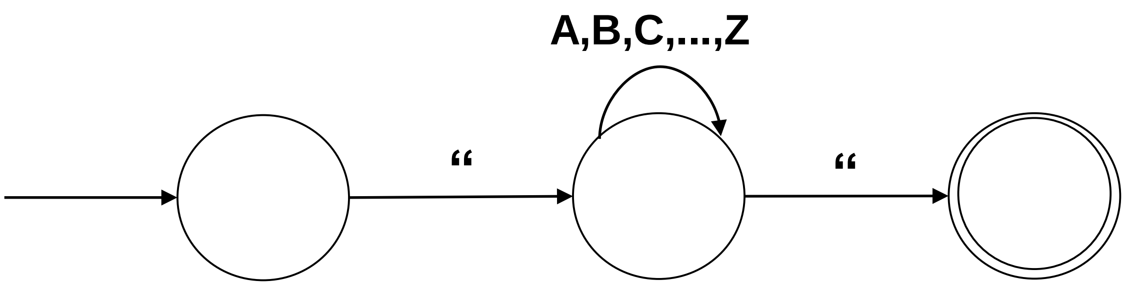


A Simple Automaton



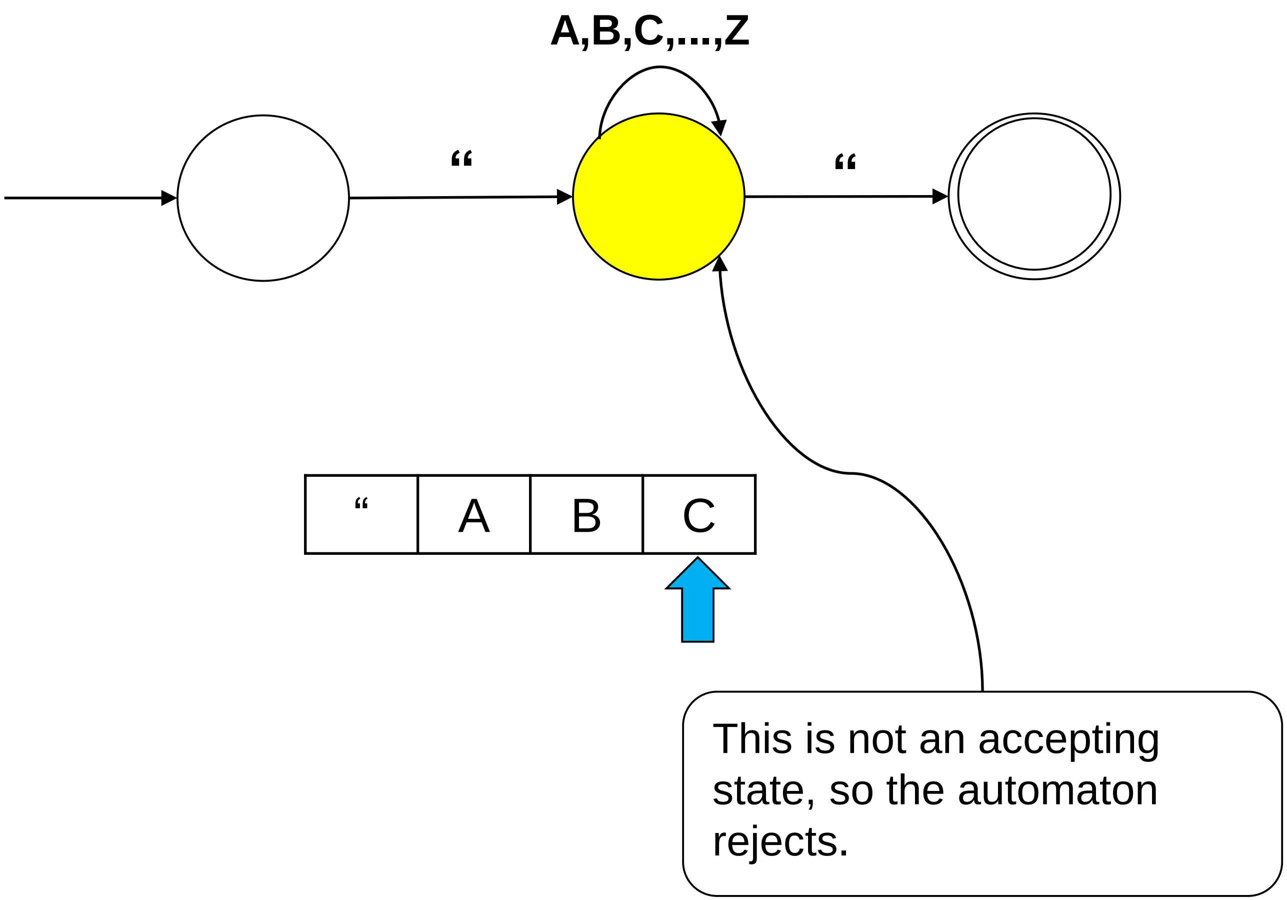
There is no transition on "
here, so the automaton **dies**
and rejects.

A Simple Automaton

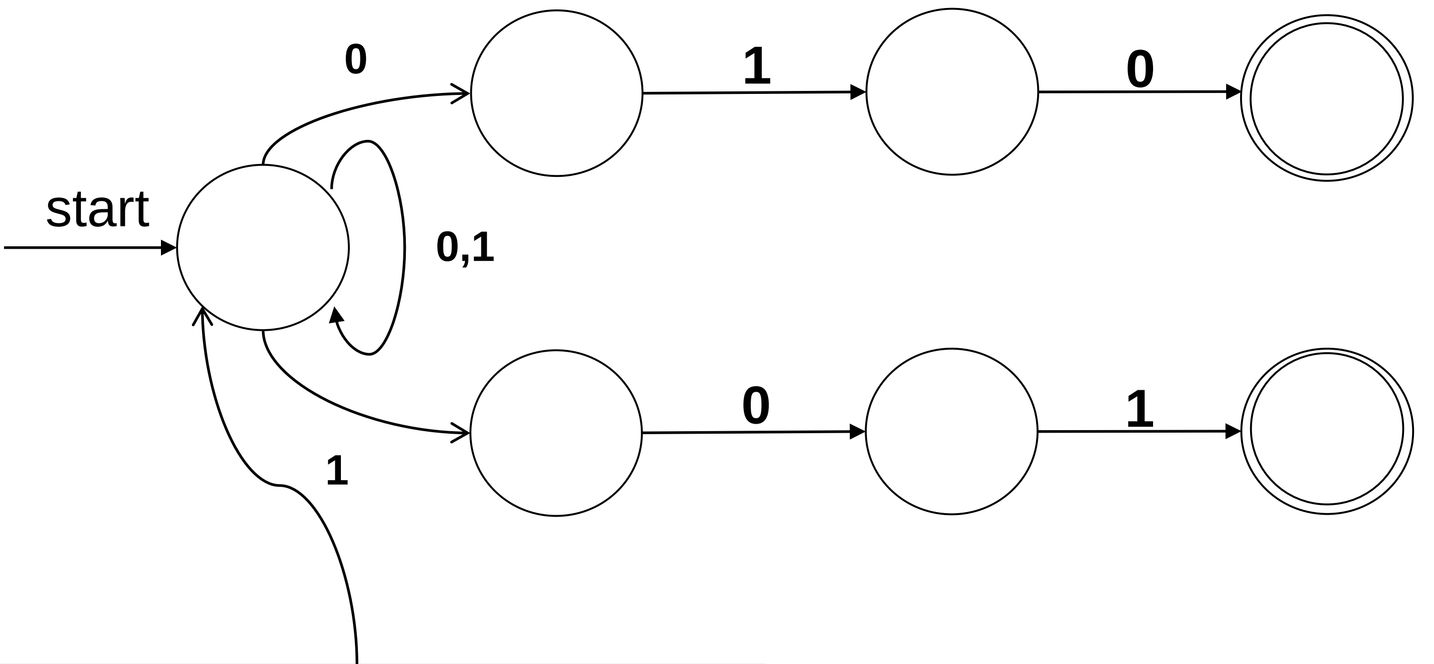


“	A	B	C

A Simple Automaton

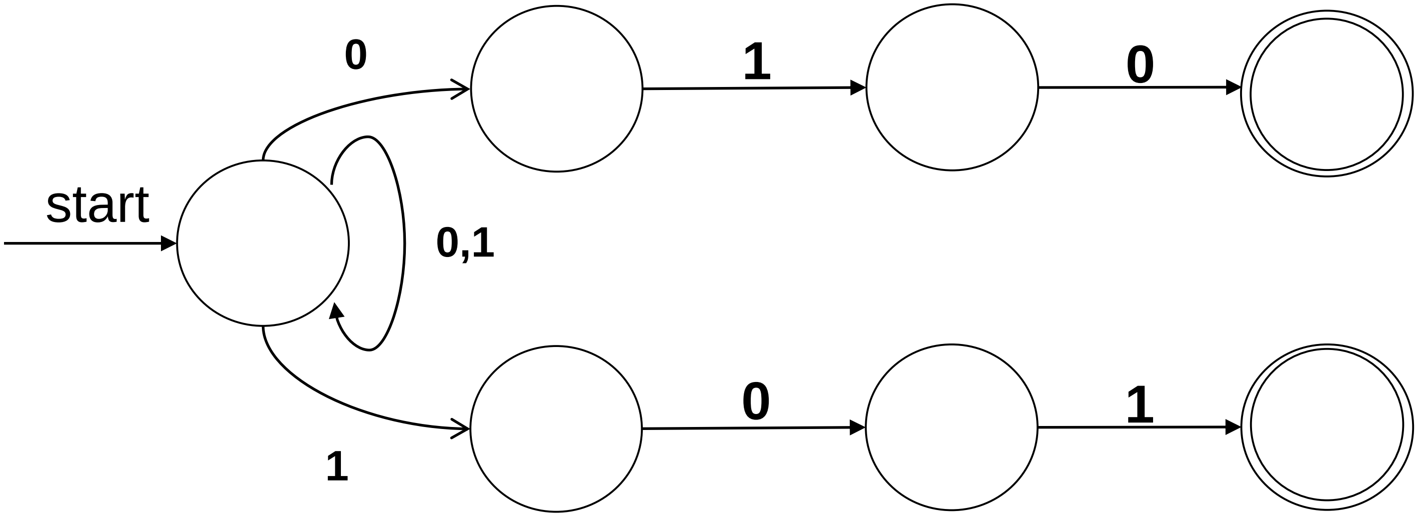


A More Complex Automaton



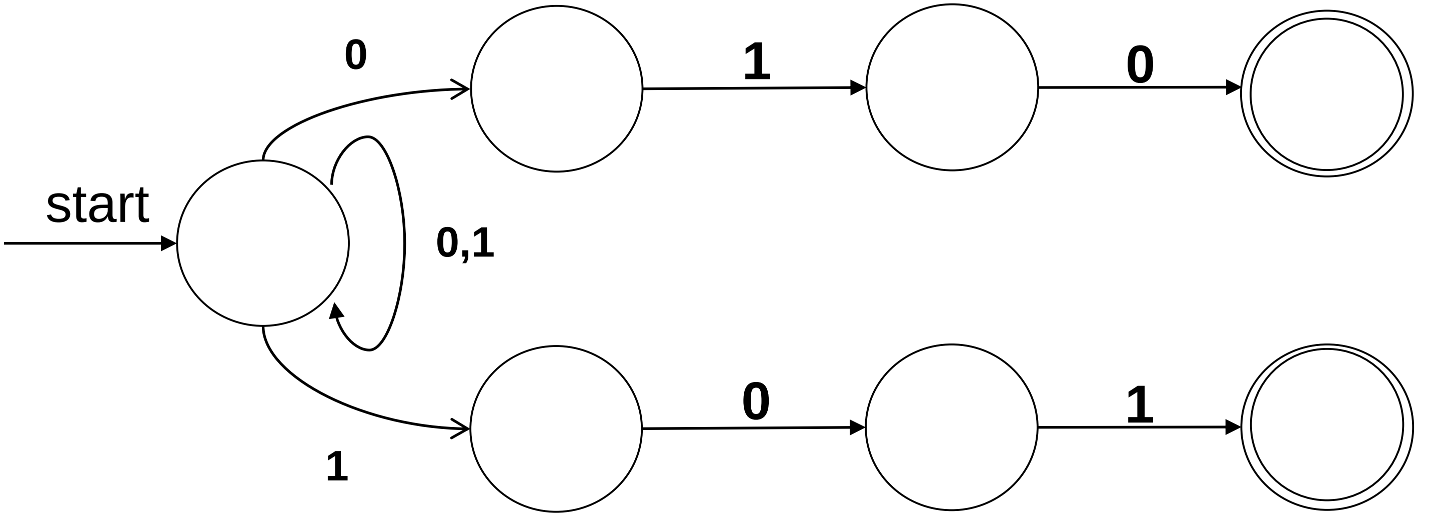
There are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow both transitions and enter multiple states.

A More Complex Automaton



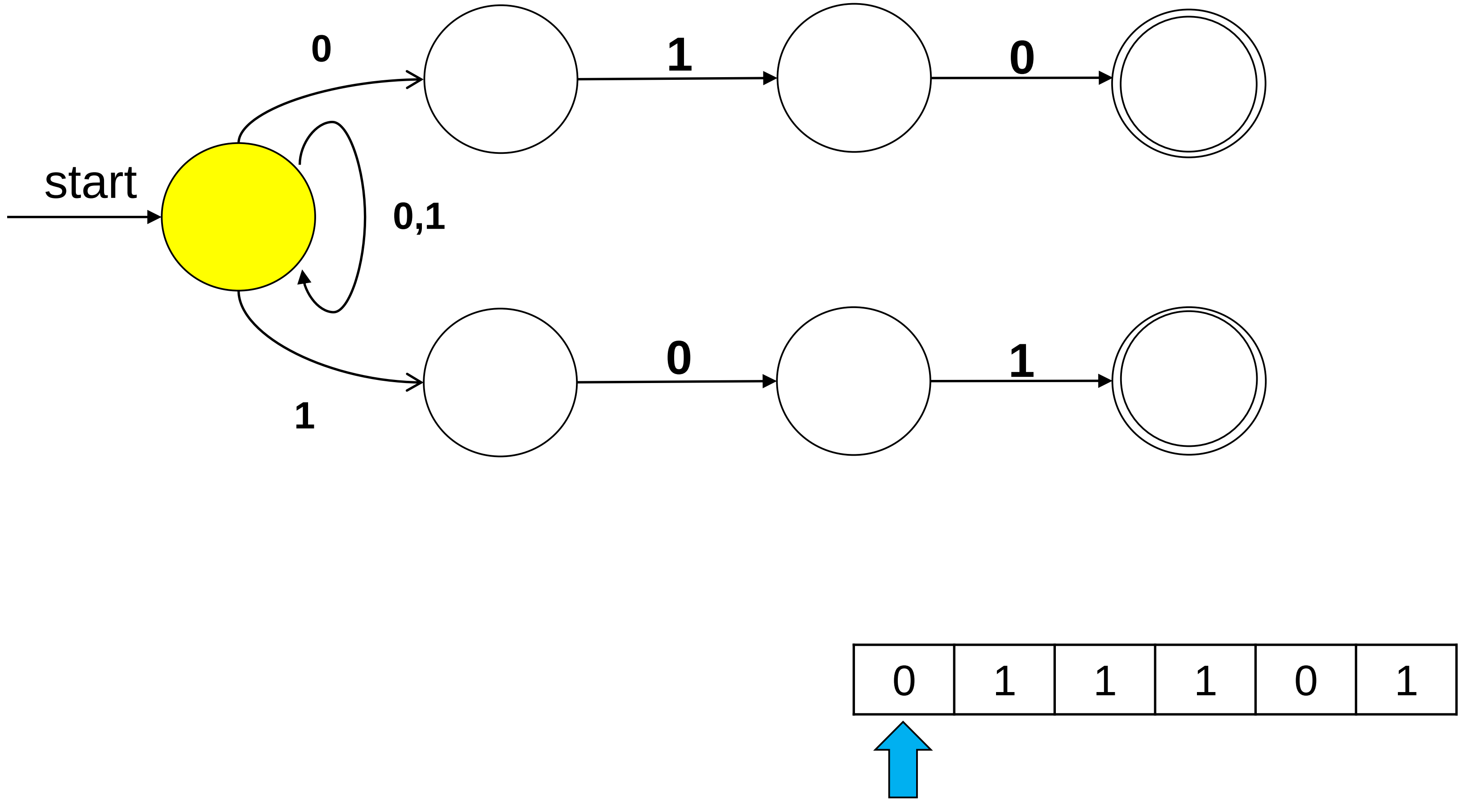
0	1	1	1	0	1

A More Complex Automaton

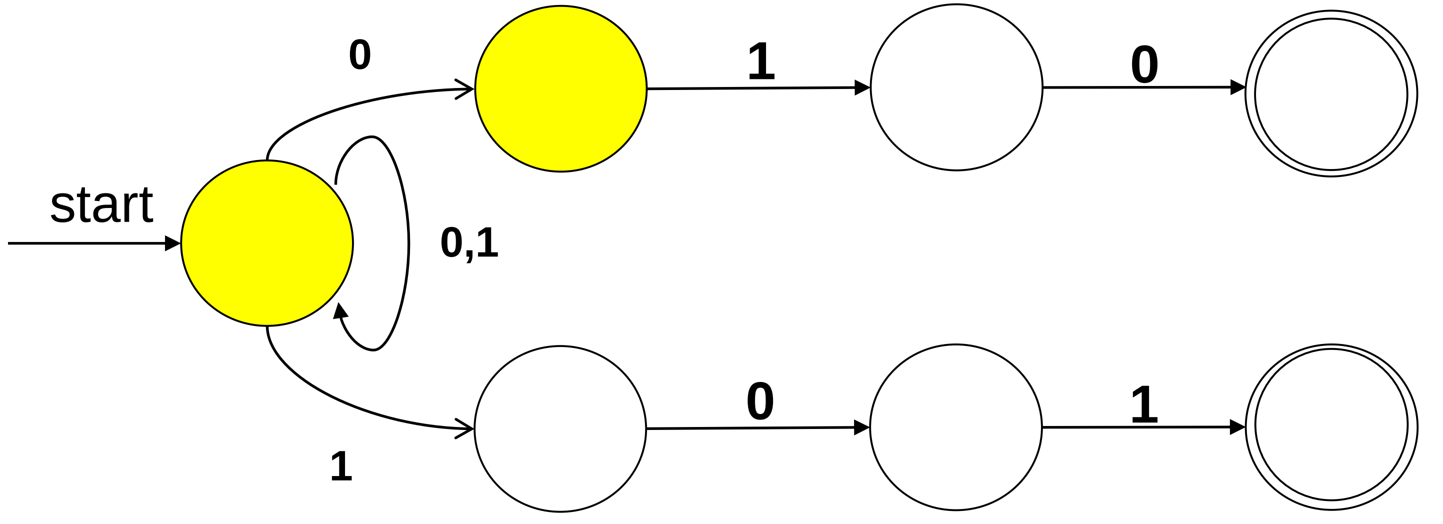


0	1	1	1	0	1

A More Complex Automaton

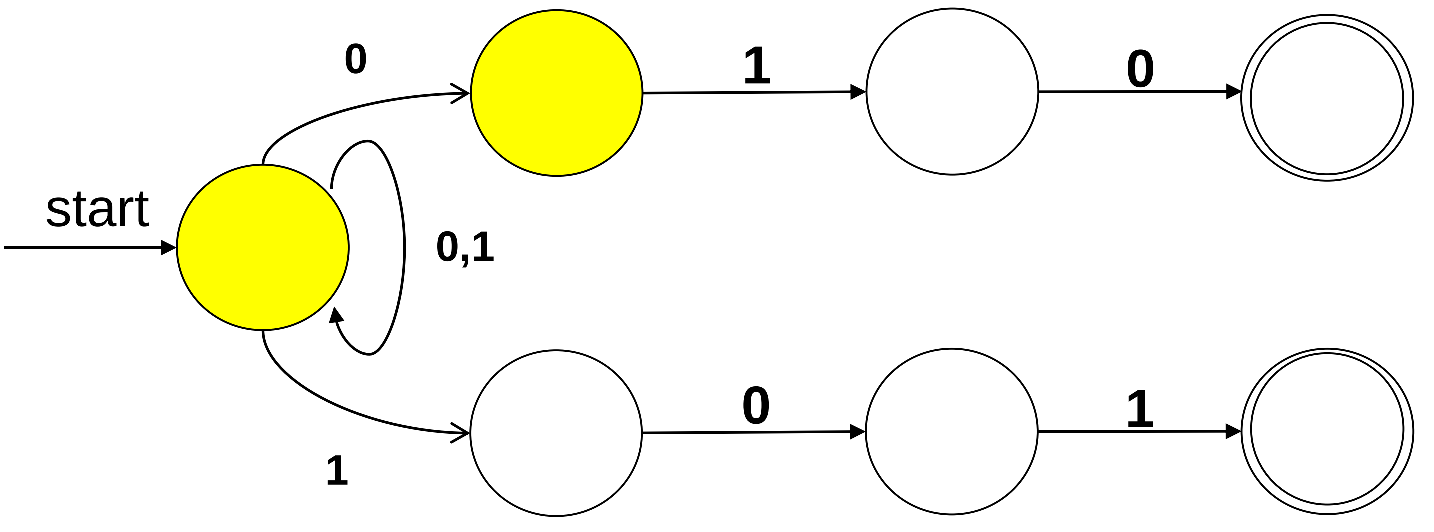


A More Complex Automaton



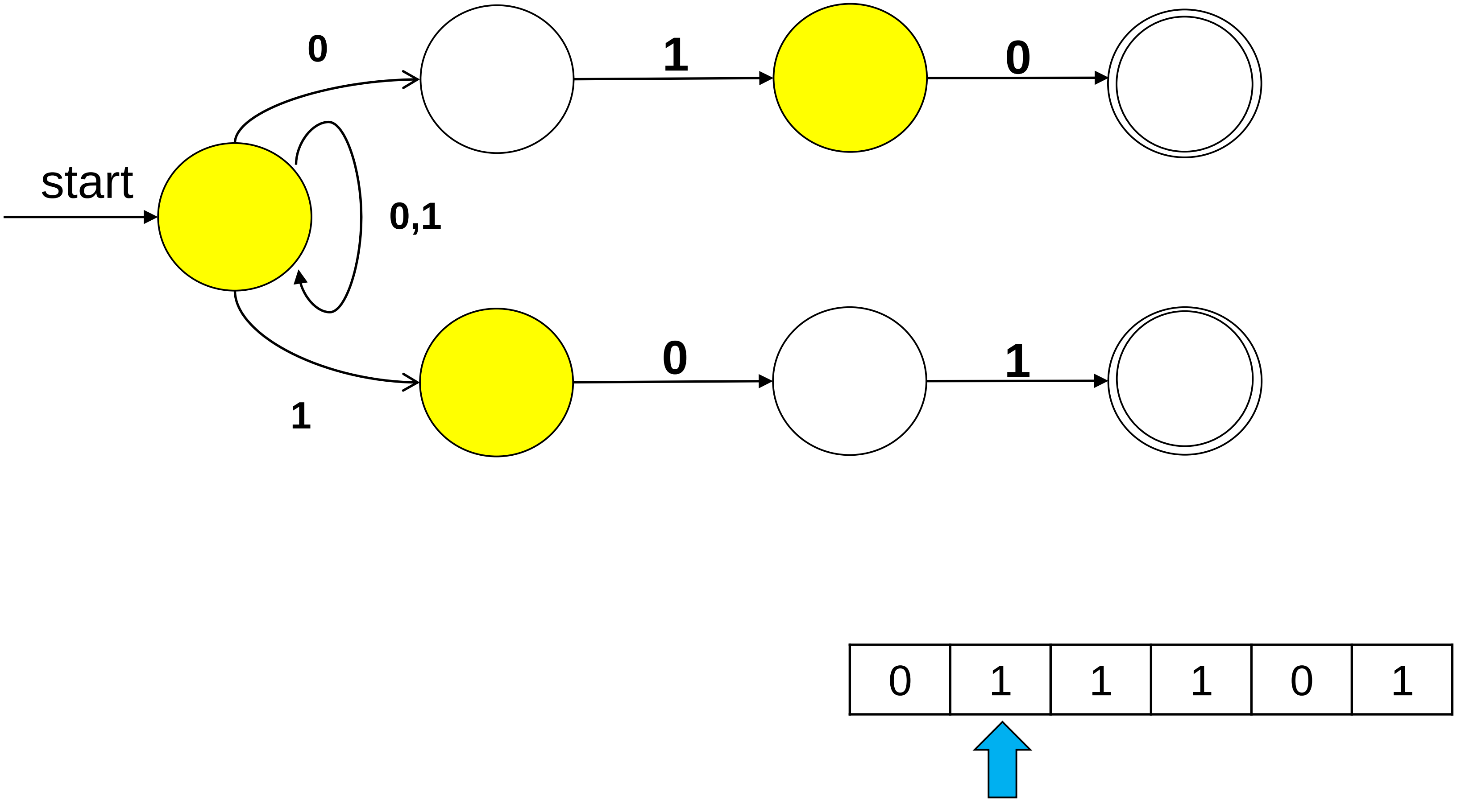
0	1	1	1	0	1

A More Complex Automaton

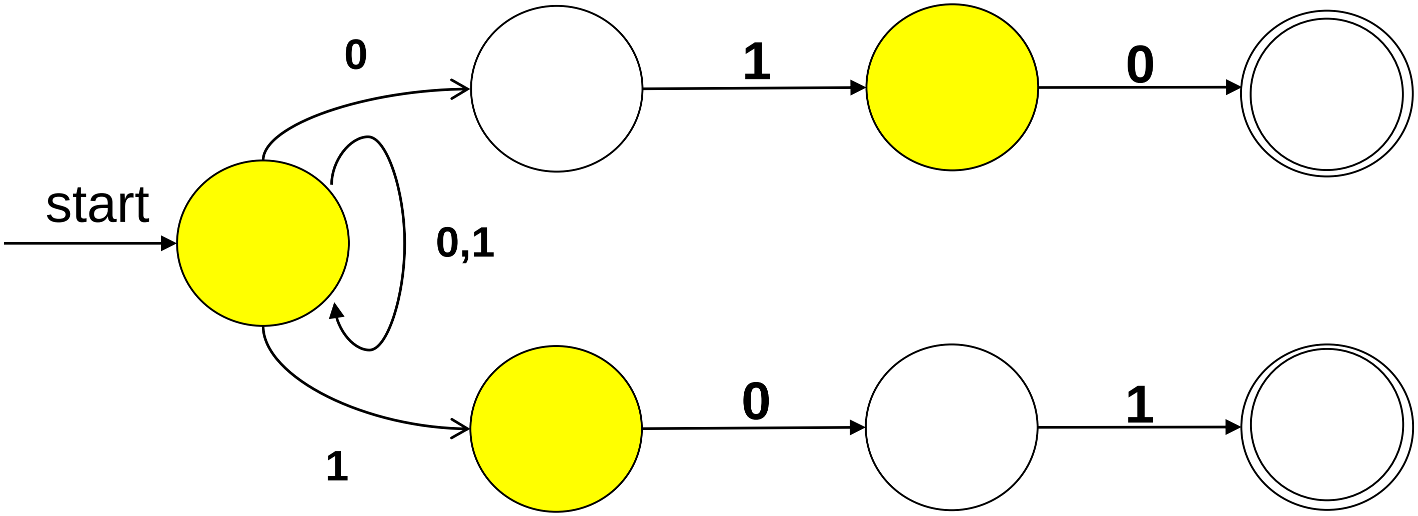


0	1	1	1	0	1

A More Complex Automaton

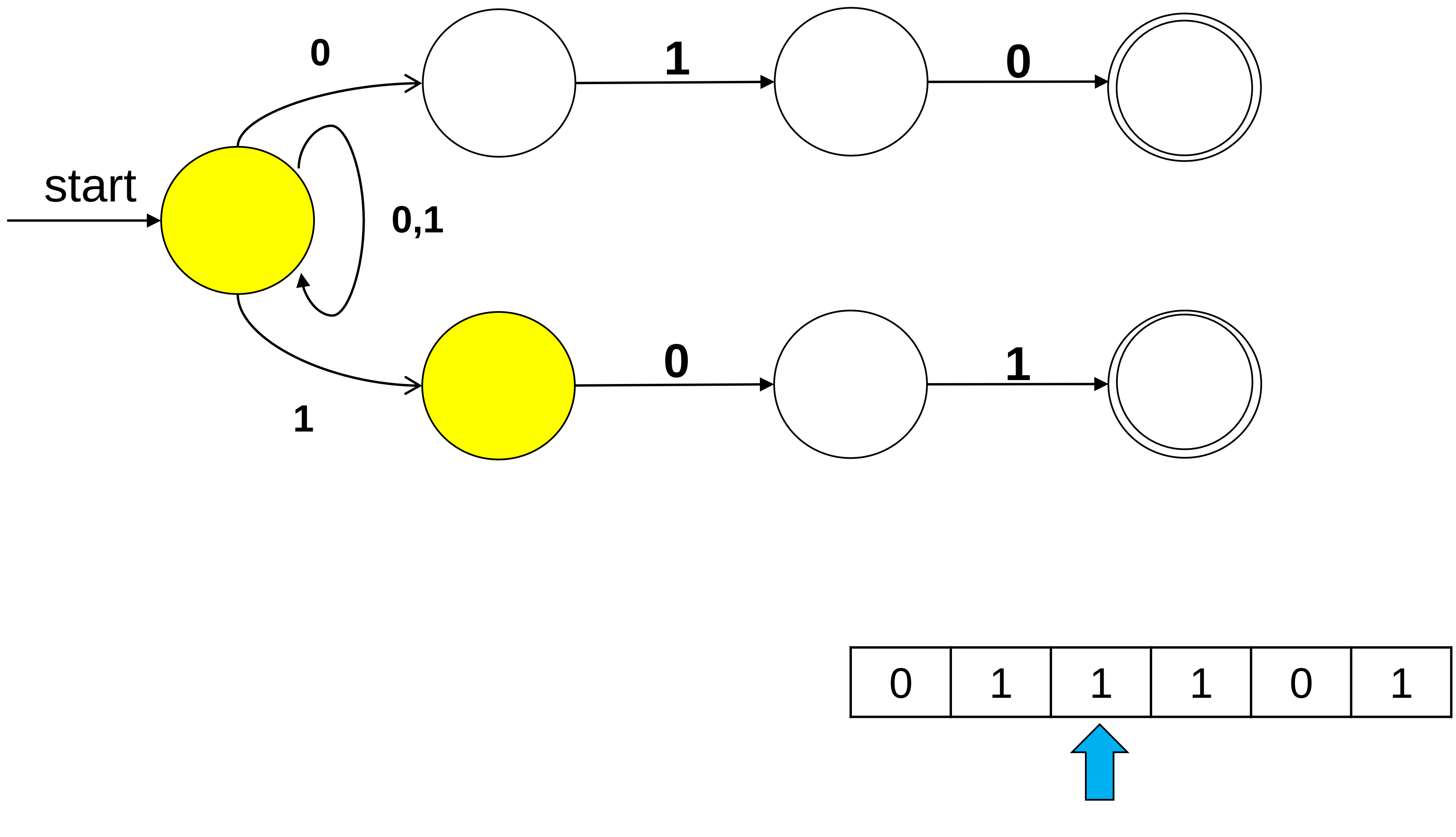


A More Complex Automaton

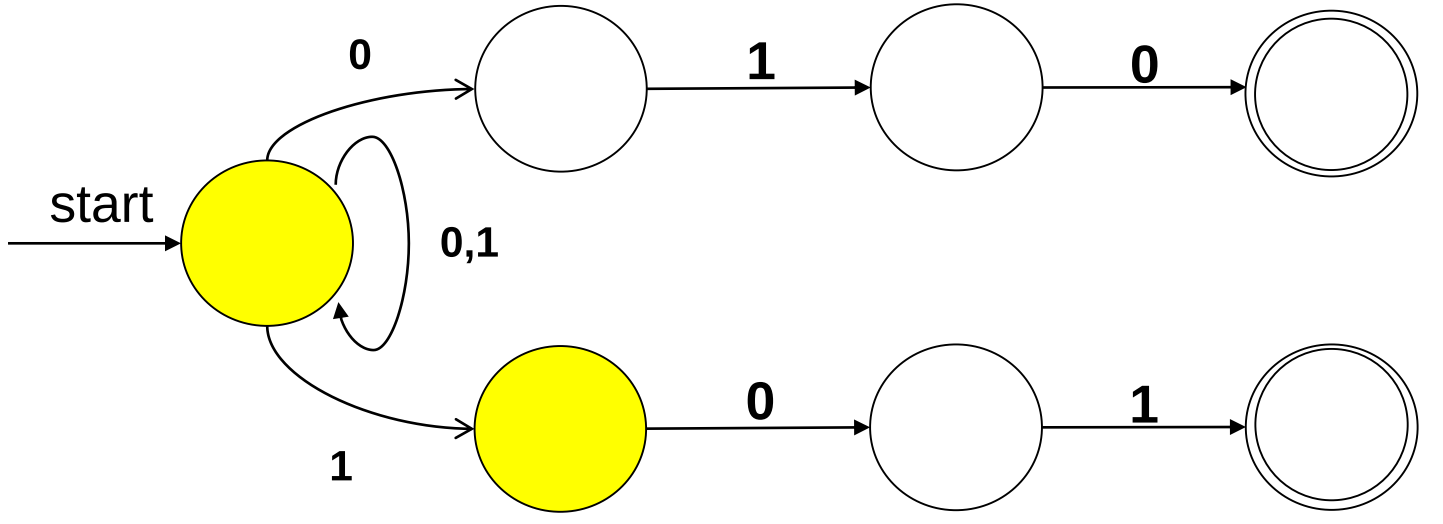


0	1	1	1	0	1

A More Complex Automaton

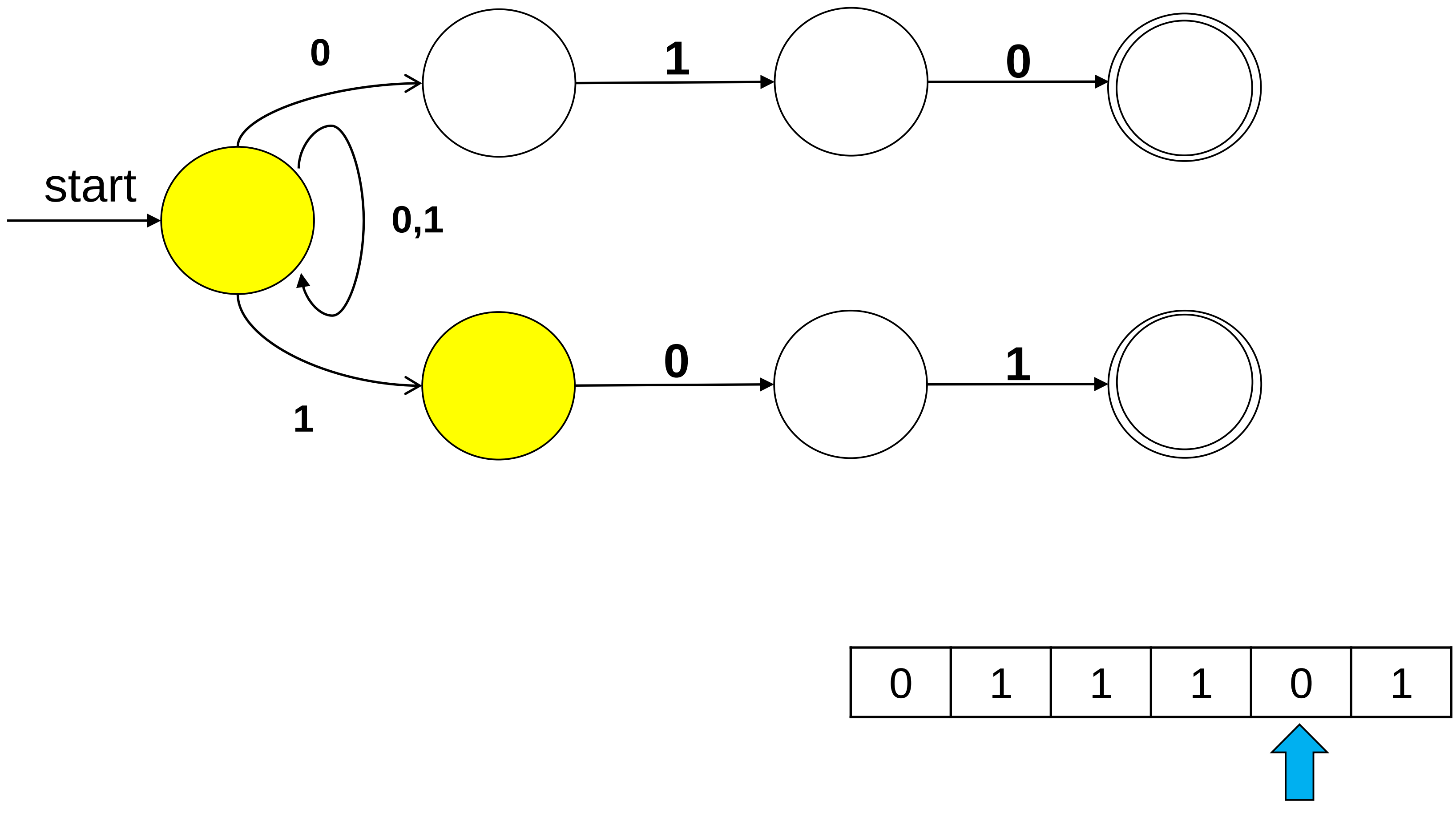


A More Complex Automaton

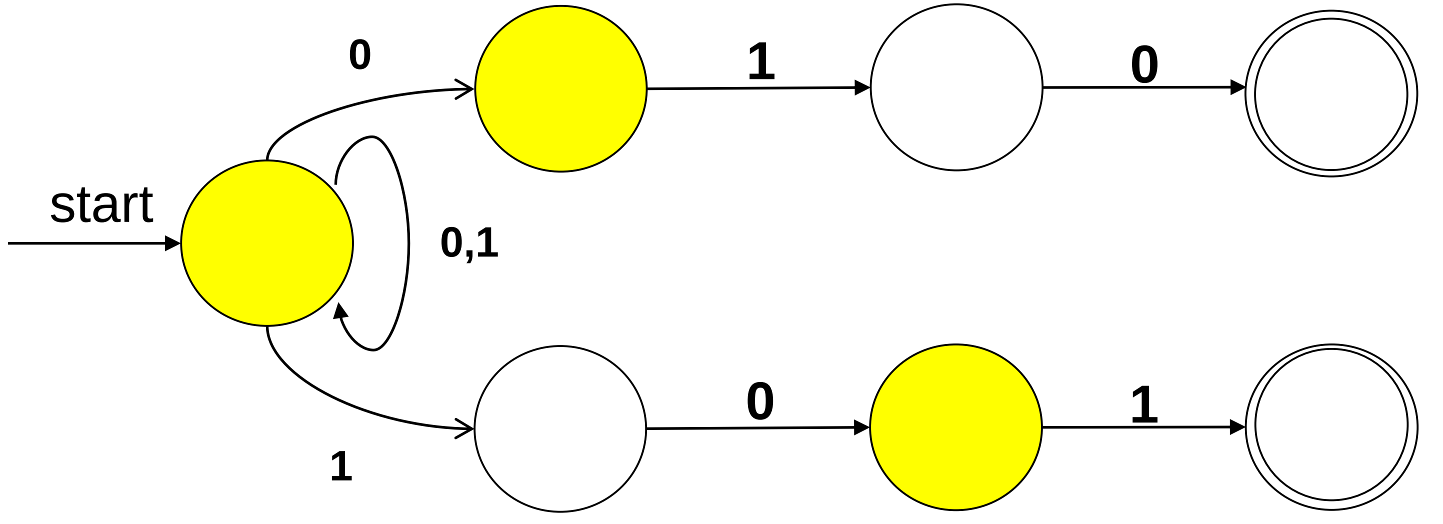


0	1	1	1	0	1

A More Complex Automaton

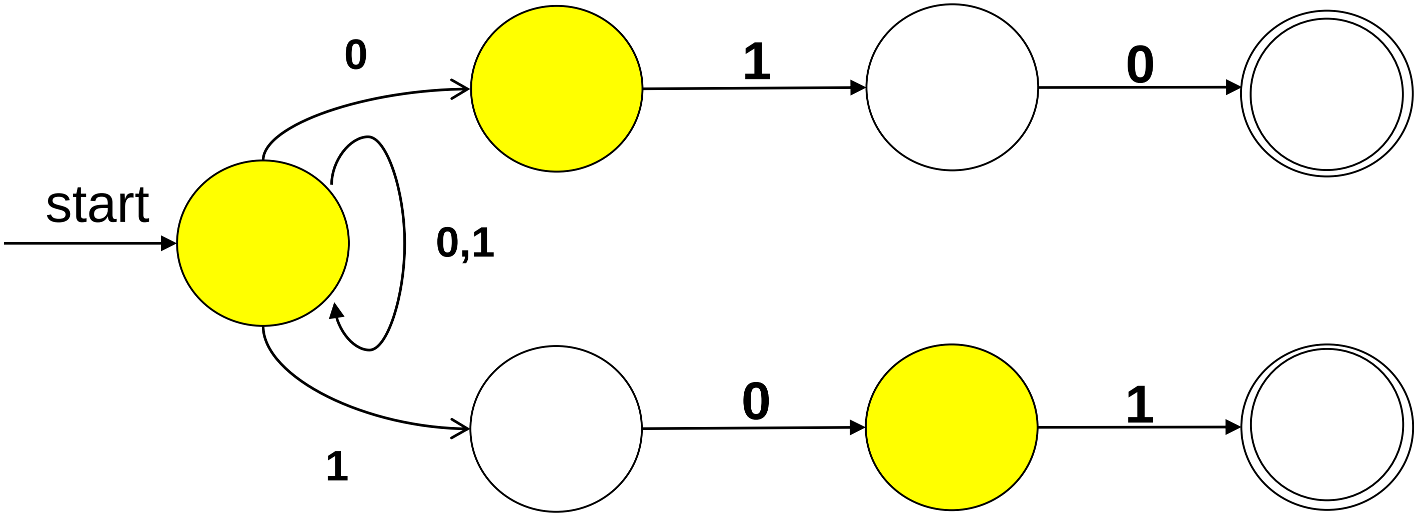


A More Complex Automaton



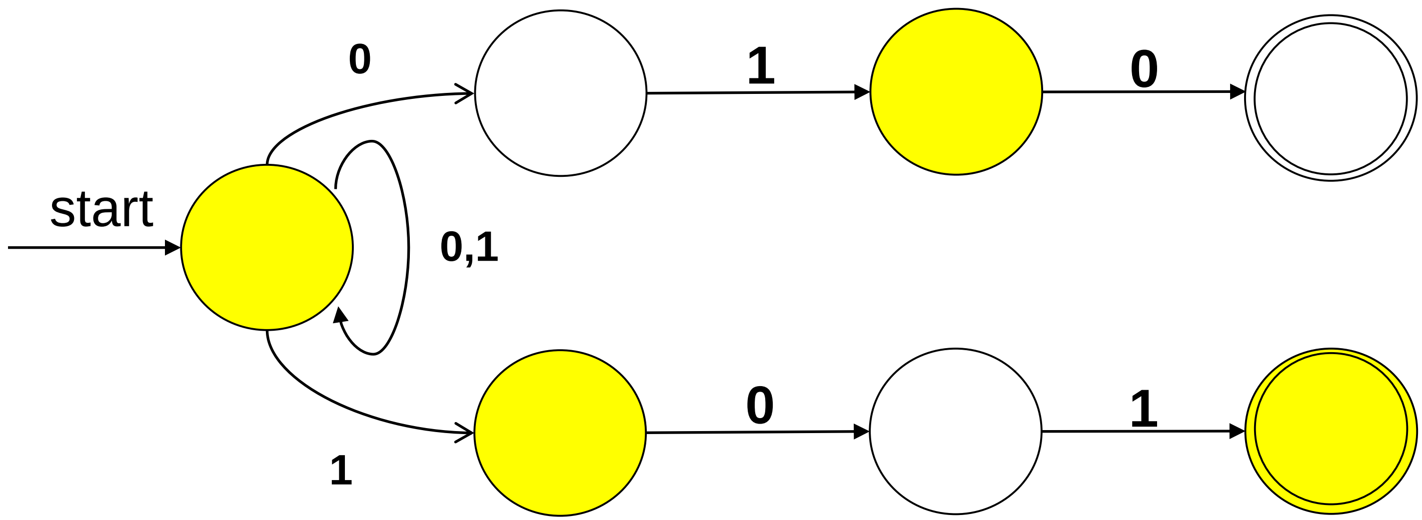
0	1	1	1	0	1

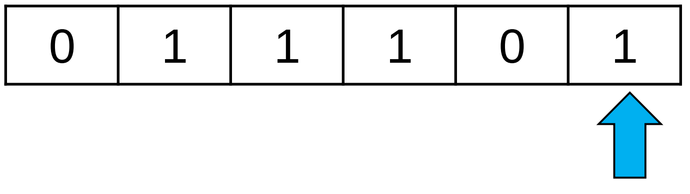
A More Complex Automaton



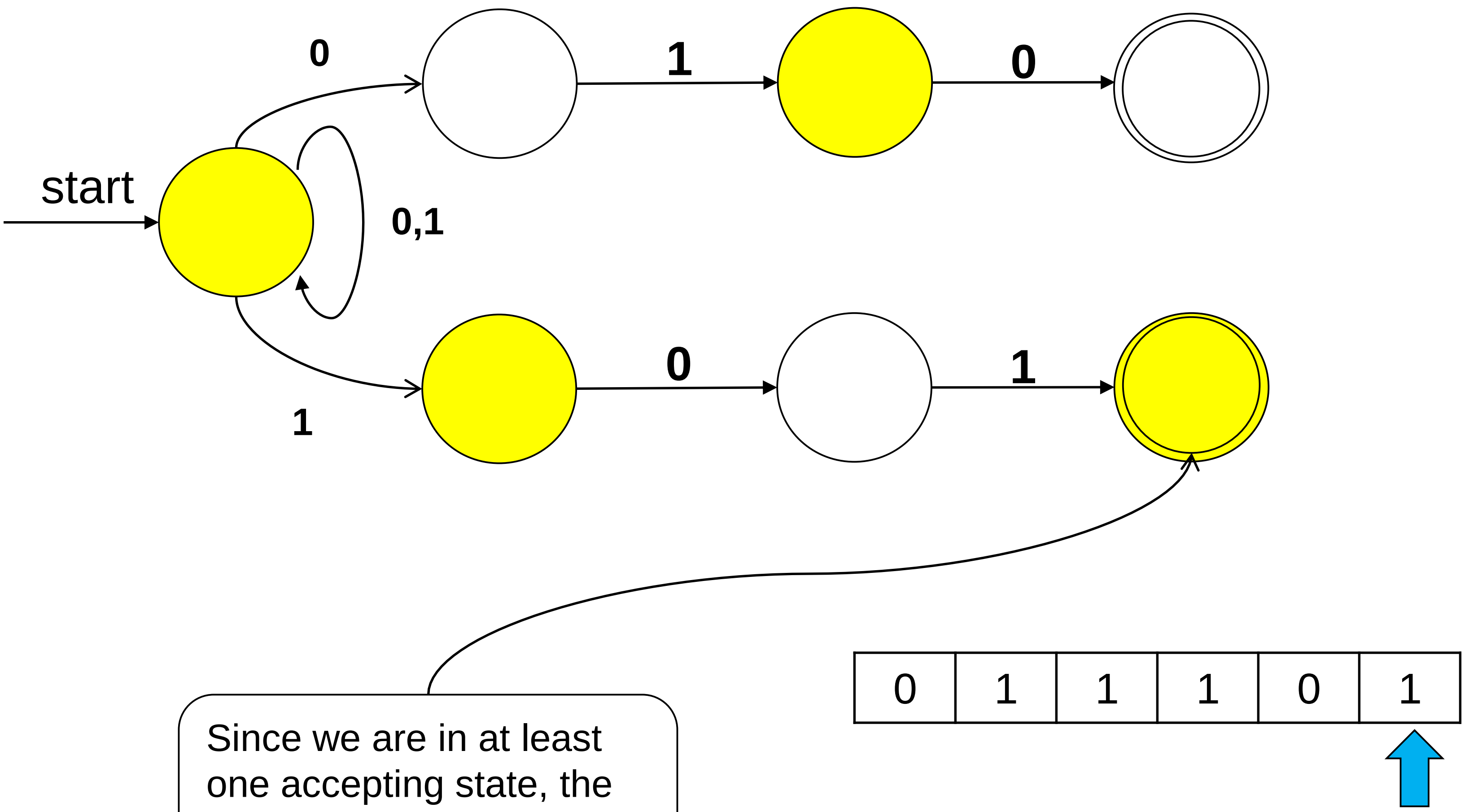
0	1	1	1	0	1

A More Complex Automaton



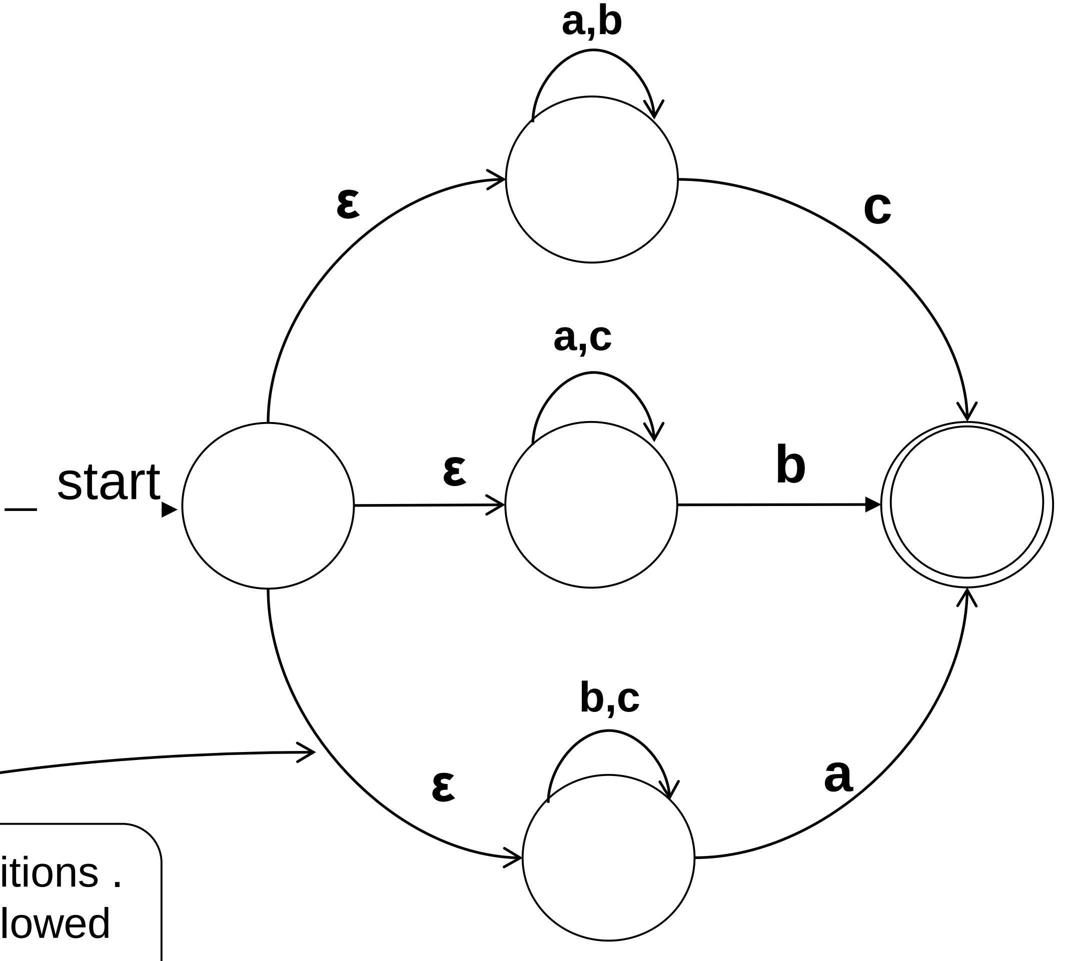


A More Complex Automaton



Since we are in at least one accepting state, the automaton accepts.

An Even More Complex Automaton



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

Simulating an NFA

- Keep track of a set of states, initially the start state and everything reachable by ε-moves.

- For each character in the input:
  - Maintain a set of next states, initially empty.
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an ε-move to the set of next states.

Outline

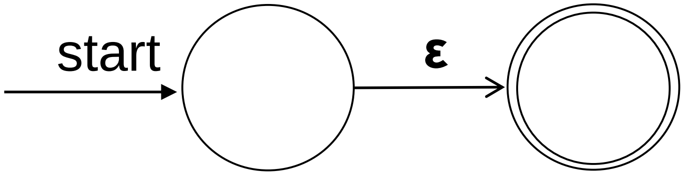
- Definition
- Associating Lexemes with Tokens
- Matching Regular Expressions
- **From RE to Automata**
- Real-world Application
- Error Recovery
- Toward Automation

From Regular Expressions to NFAs

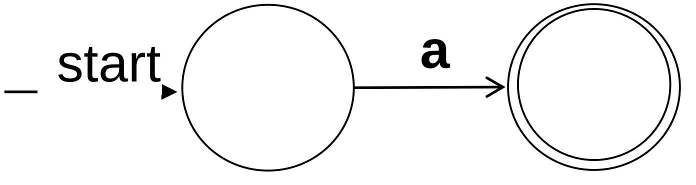
- There is an straightforward procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting state.
- These restrictions are stronger than necessary, but make the construction easier.

Basic Cases

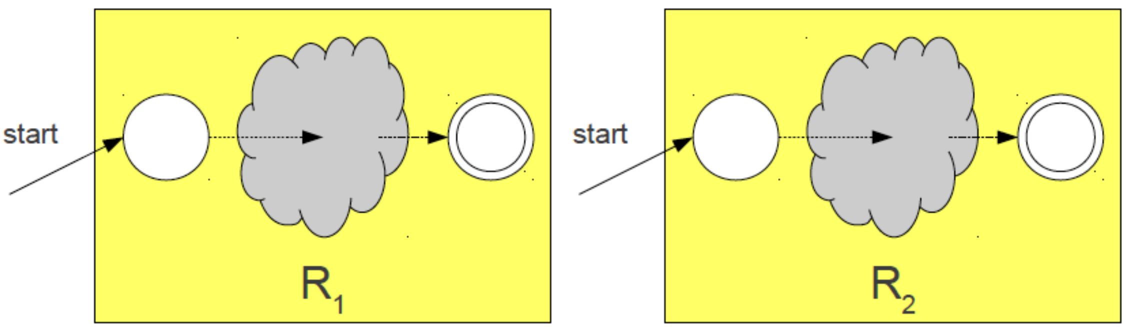
Automaton for ε



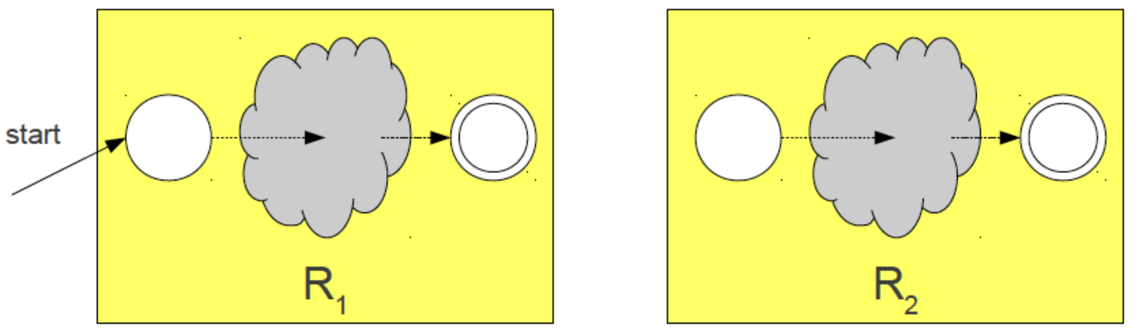
Automaton for single character a



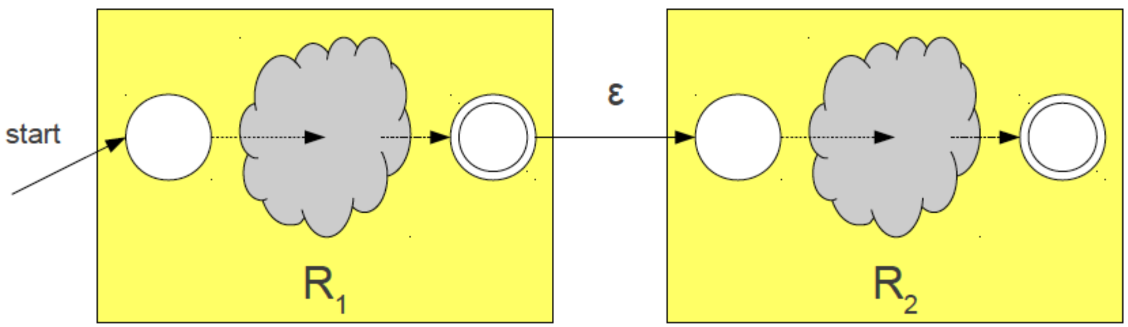
Construction for R_1R_2



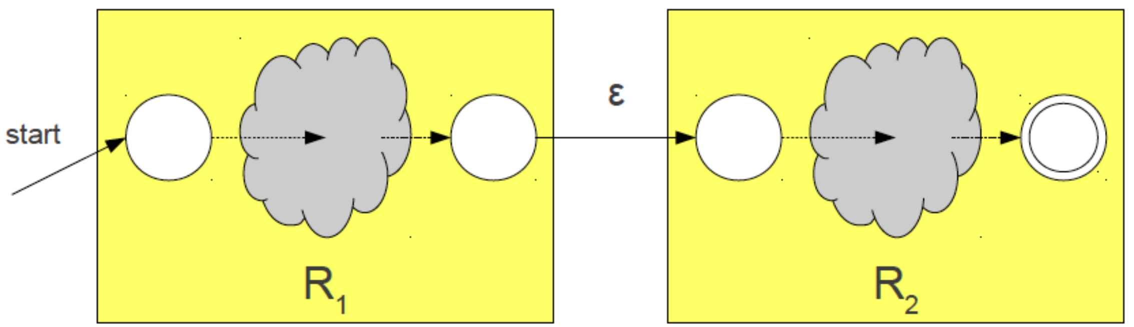
Construction for R_1R_2



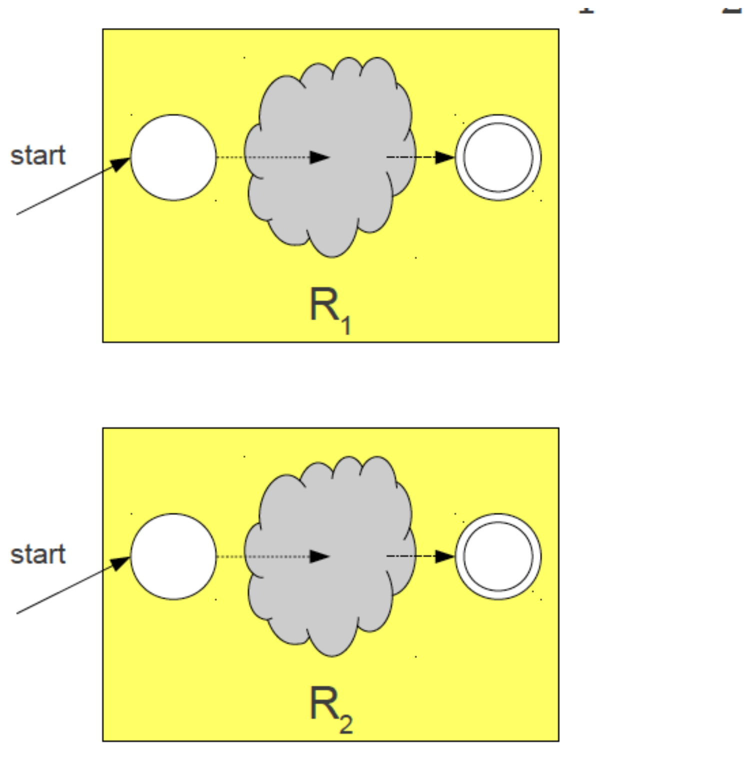
Construction for R_1R_2



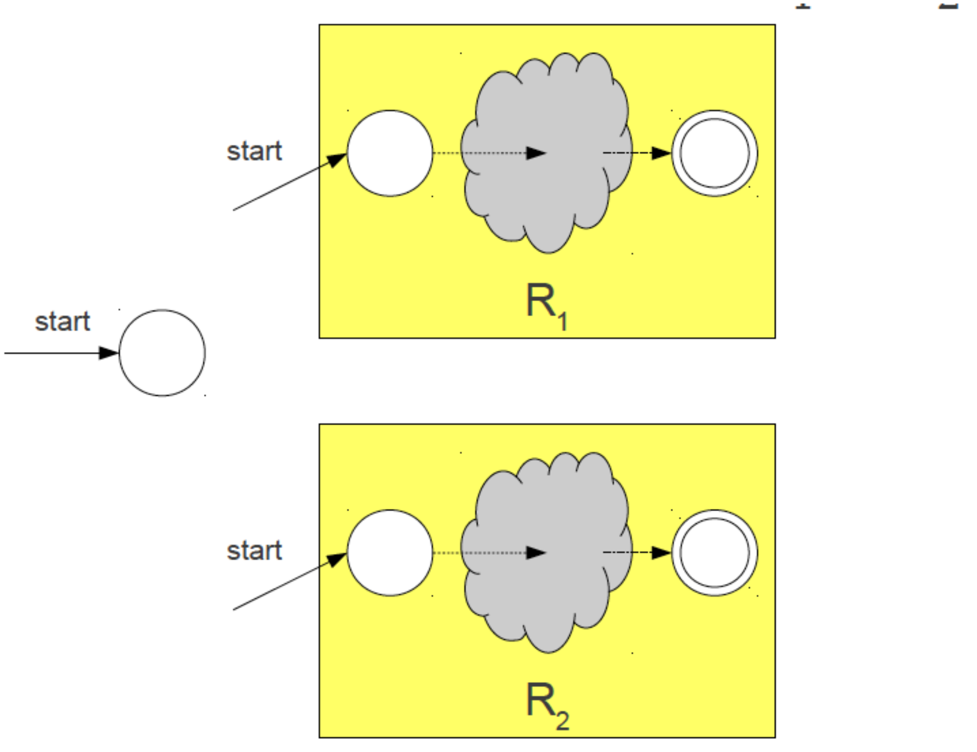
Construction for R_1R_2



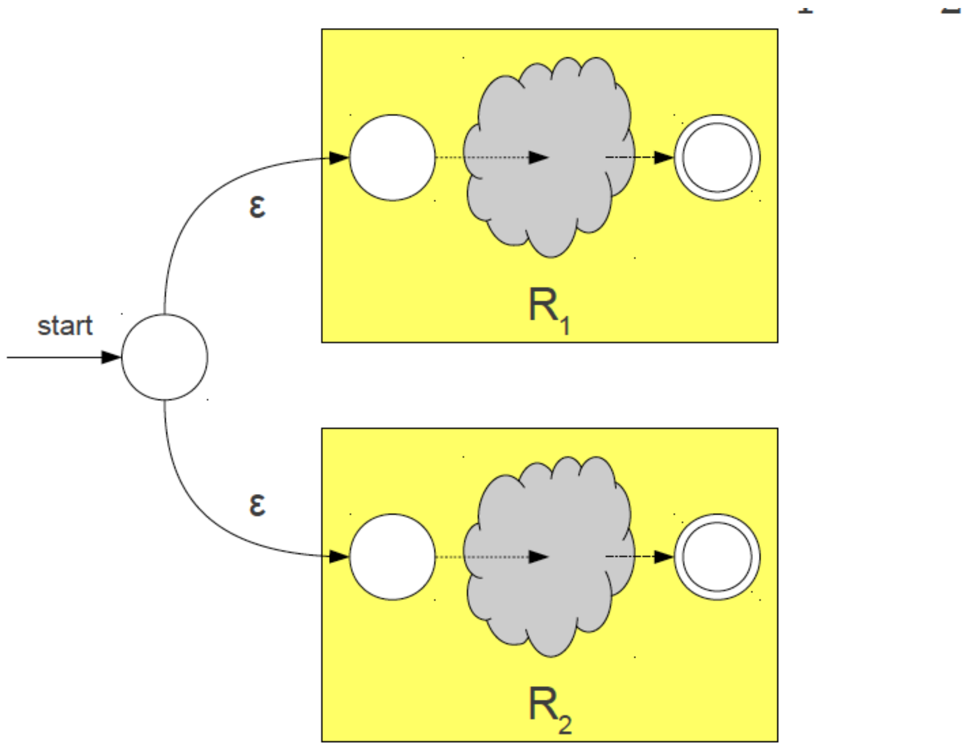
Construction for $R_1 \mid R_2$



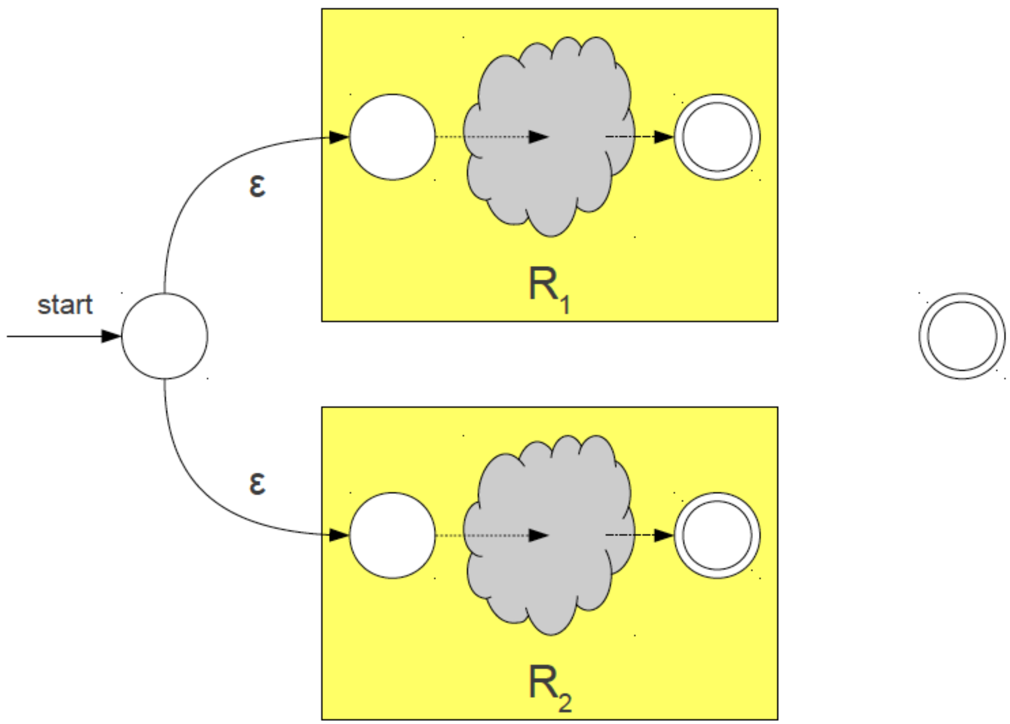
Construction for $R_1 \mid R_2$



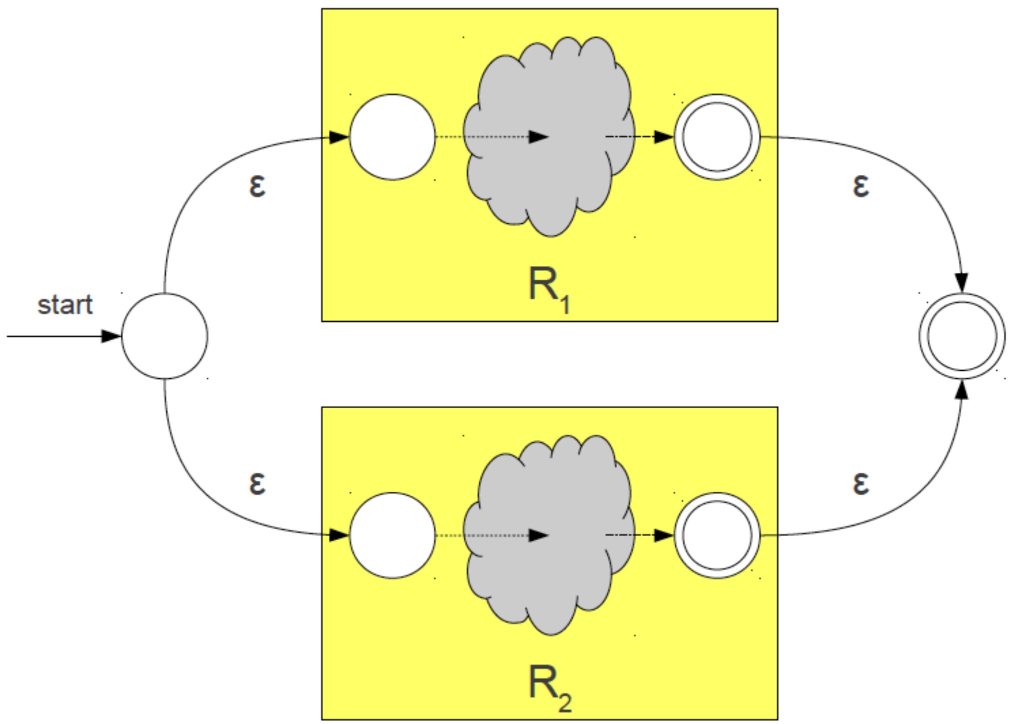
Construction for $R_1 \mid R_2$



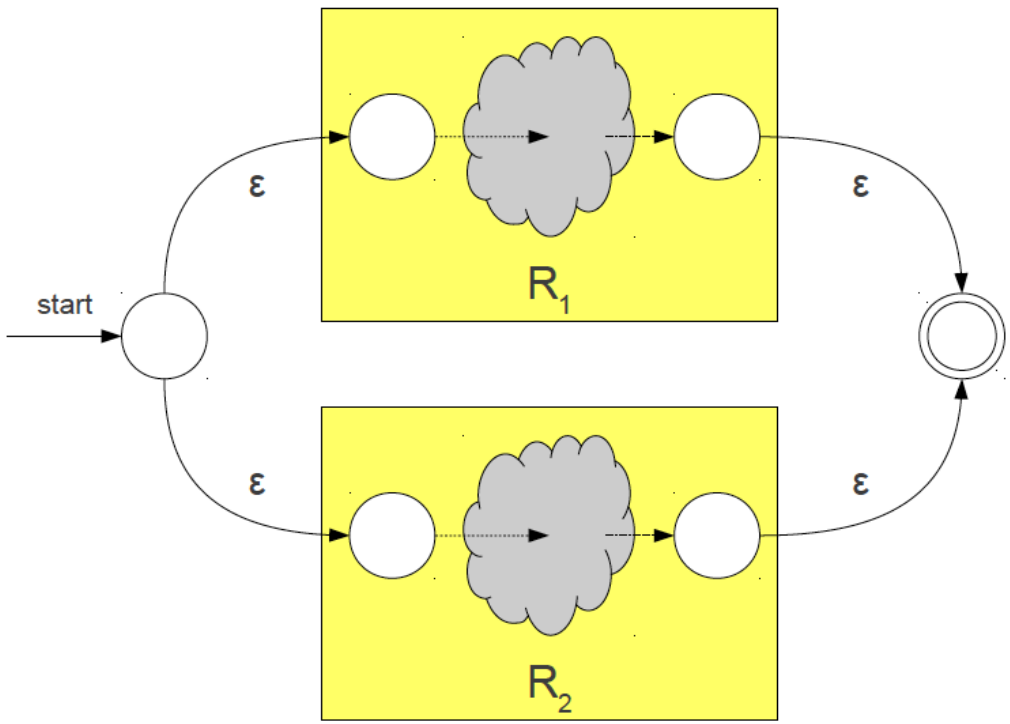
Construction for $R_1 \mid R_2$



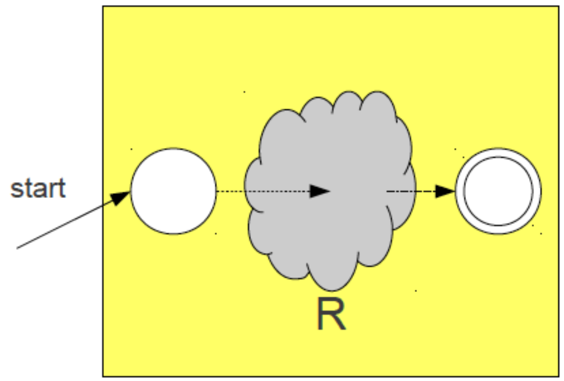
Construction for $R_1 | R_2$



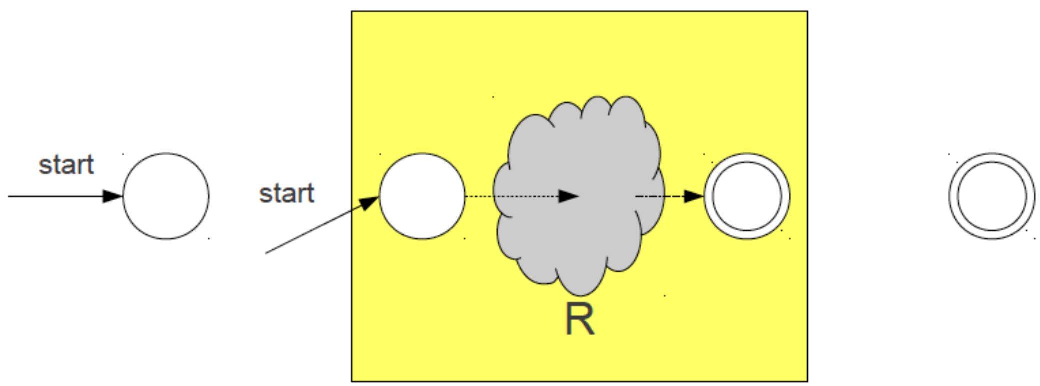
Construction for $R_1 \mid R_2$



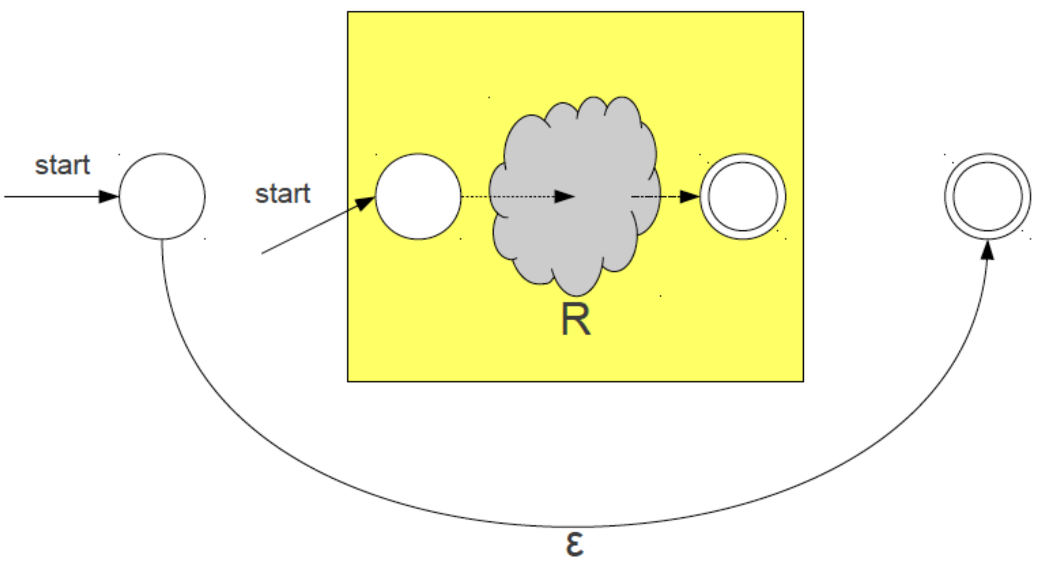
Construction for R*



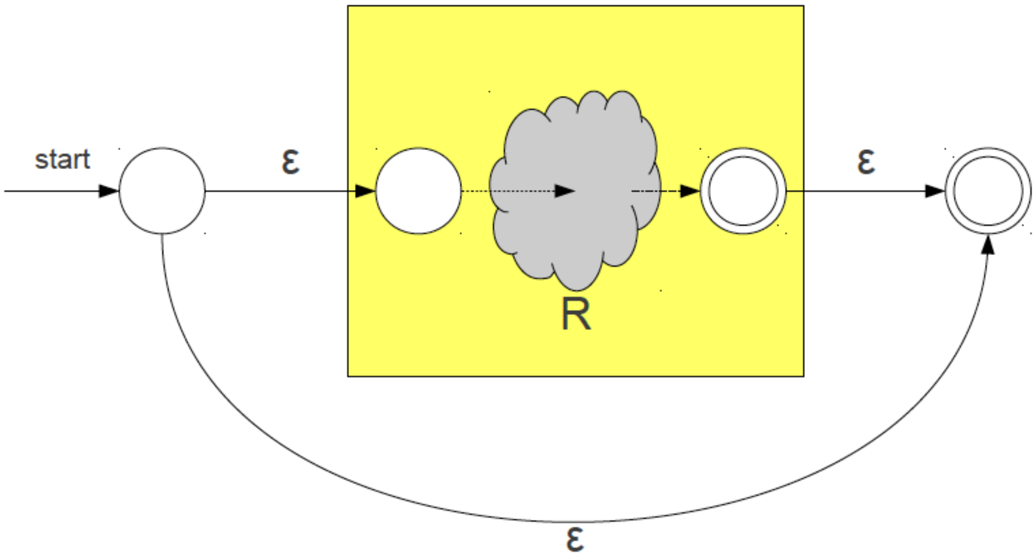
Construction for R*



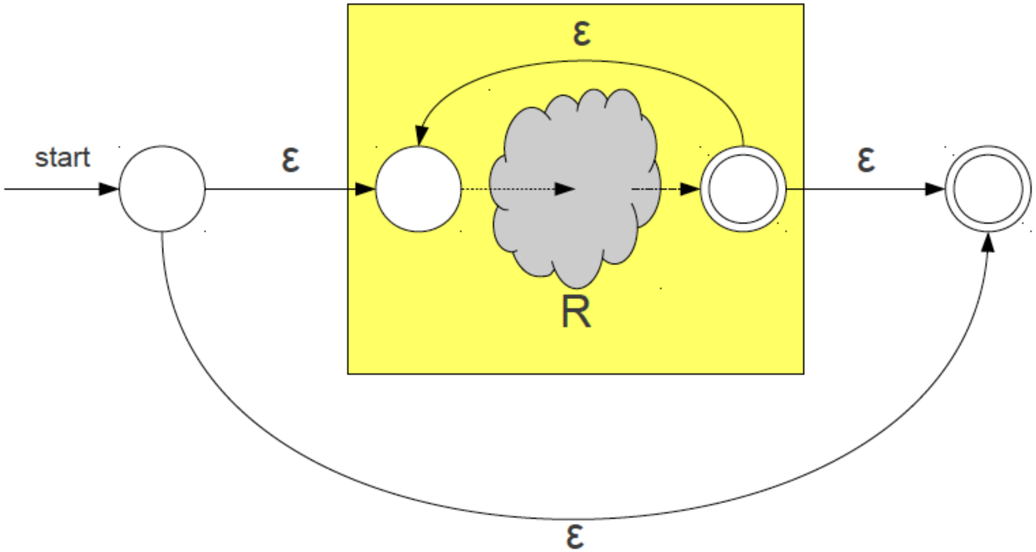
Construction for R*



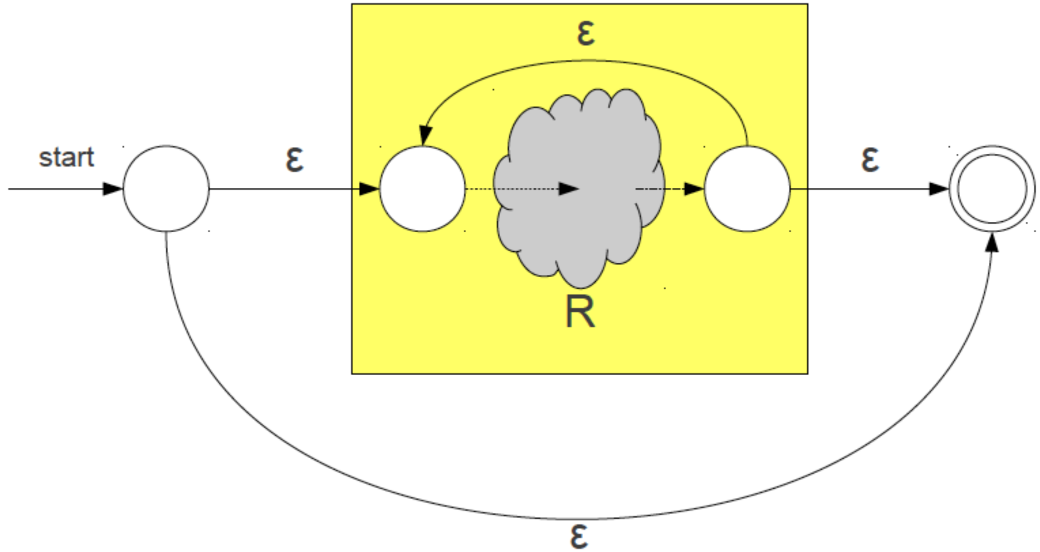
Construction for R*



Construction for R*

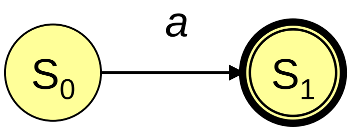


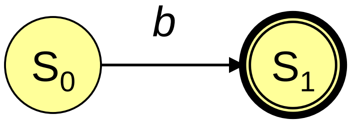
Construction for R*

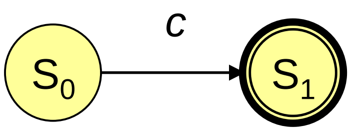


Example: Construct the NFA of *a (b|c)**

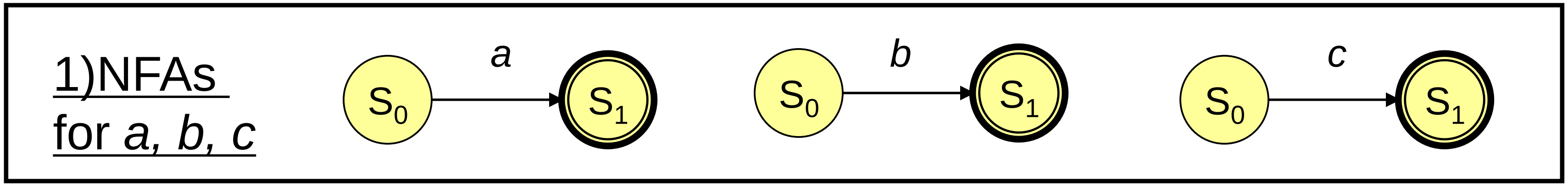
1)NFAs
for *a, b, c*

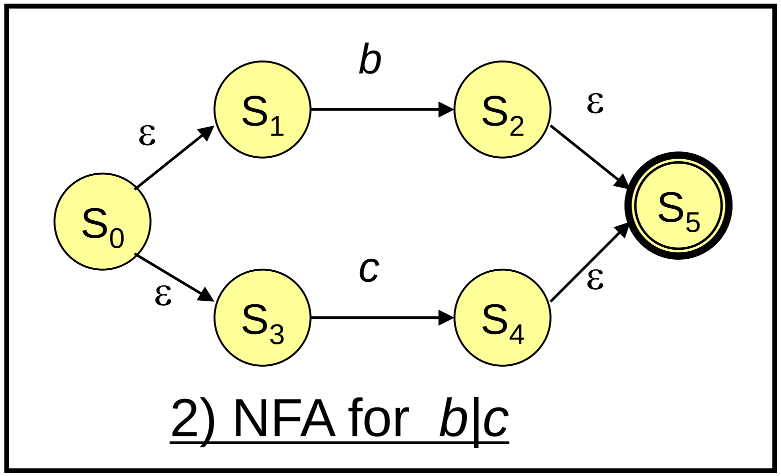






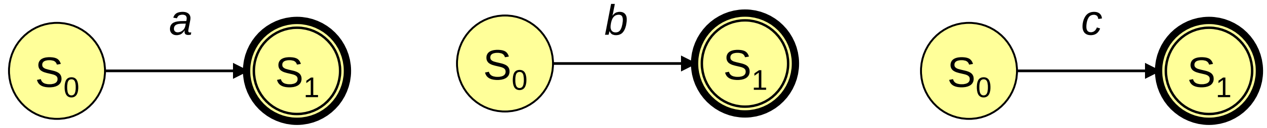
Example: Construct the NFA of *a (b|c)**

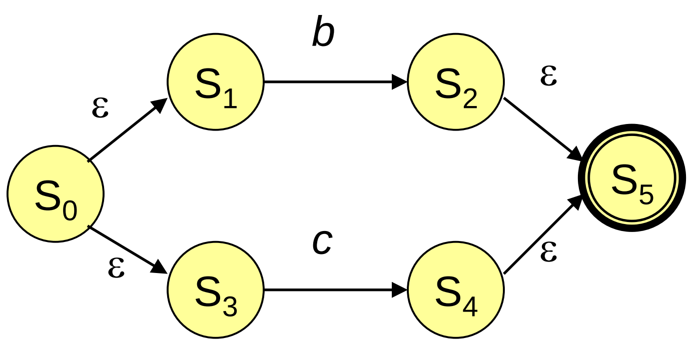




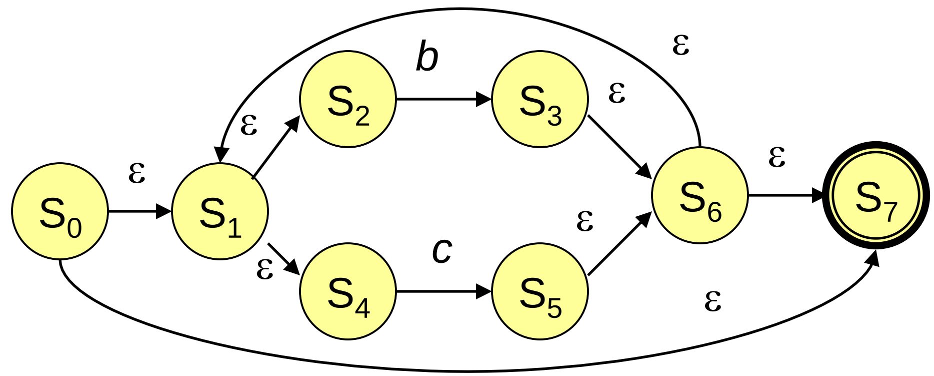
Example: Construct the NFA of *a (b|c)**

1)NFAs for *a, b, c*





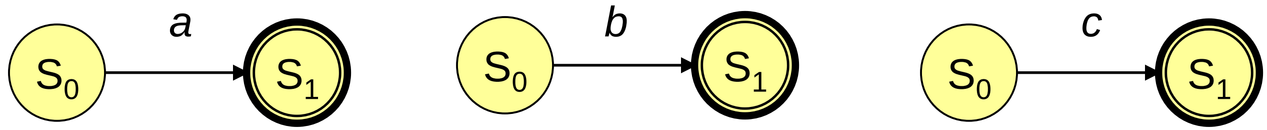
2) NFA for *b|c*

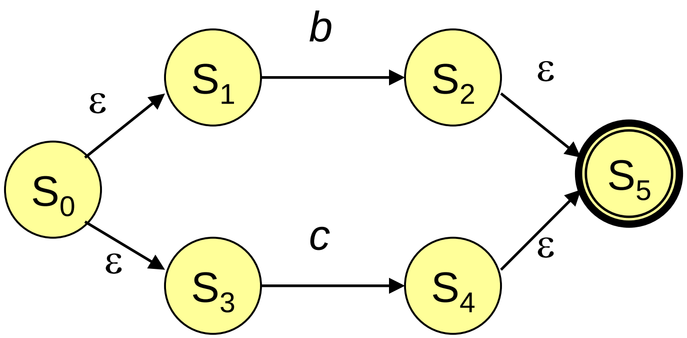


3) NFA for *(b|c)**

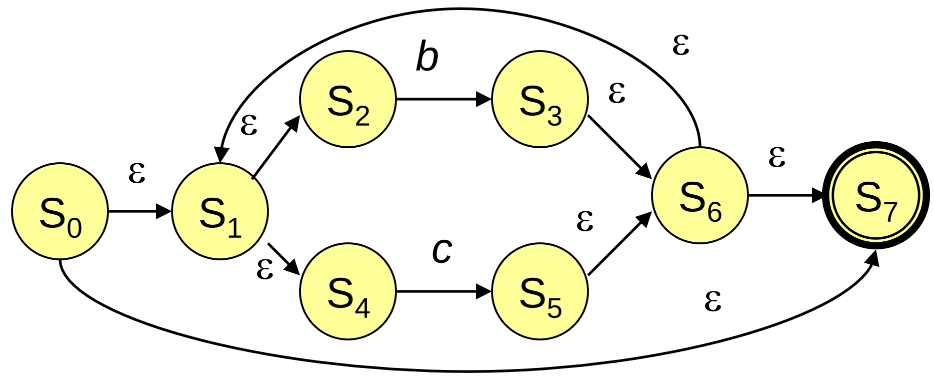
Example: Construct the NFA of *a (b|c)**

1)NFAs for *a, b, c*



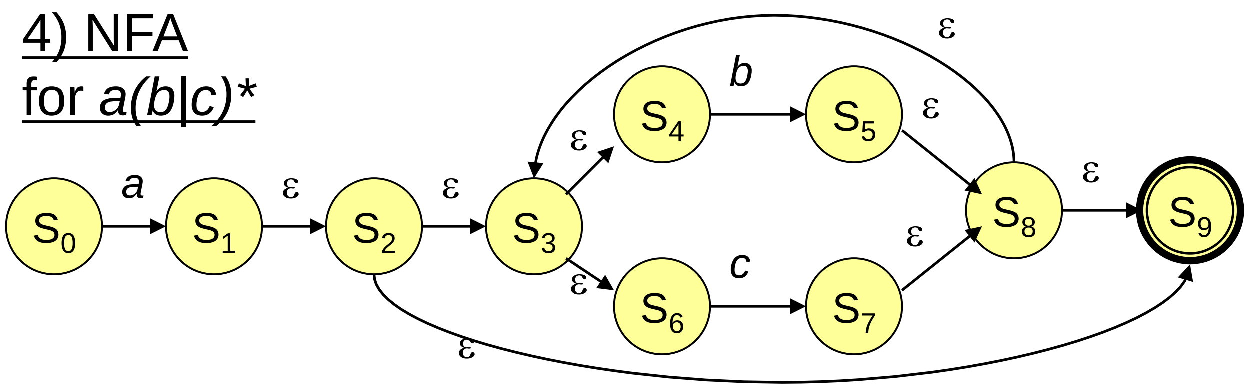


2) NFA for *b|c*

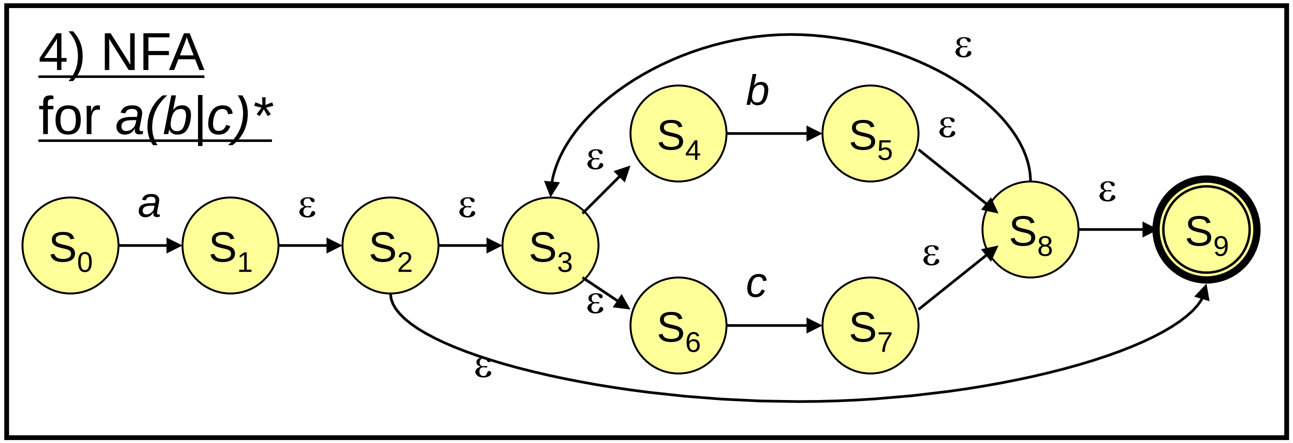


3) NFA for *(b|c)**

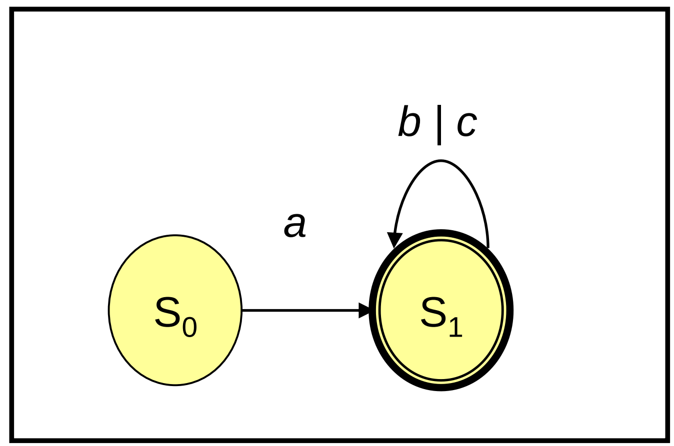
4) NFA for *a(b|c)**



Example: Construct the NFA of *a (b|c)**



Of course, a human would design a simpler one…
But, we can automate production of the complex one...



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.

Speeding up Matching

- In the worst-case, an NFA with n states takes time $O(mn^2)$ to match a string of length m.

- DFAs, on the other hand, take only O(m).

- There is another straightforward algorithm (Subset Construction) to convert NFAs to DFAs.

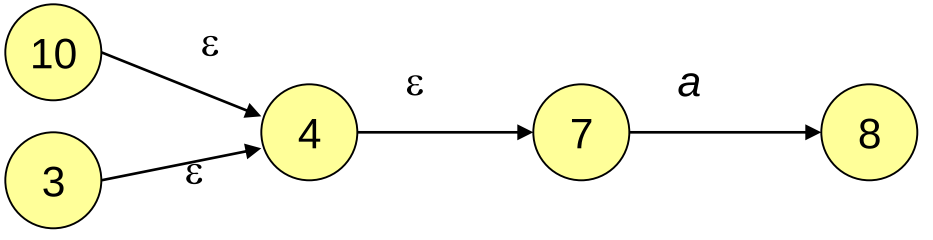
NFA to DFA: two key functions

- **move(s_i,a):** the (union of the) set of states to which there is a transition on input symbol **a** from state s_i

- **ε-closure(s_i):** the (union of the) set of states reachable by **ε** from s_i.

NFA to DFA: two key functions

- Example:
  - ε-closure(3)={3,4,7}
  - ε-closure(10)={4,7,10};
  - move(7,*a*)=8;



- The Algorithm starts with the ε-closure of s_0 from NFA.
- Do for each unmarked state until there are no unmarked states:
  - for each symbol take their ε-closure(move(state,symbol))

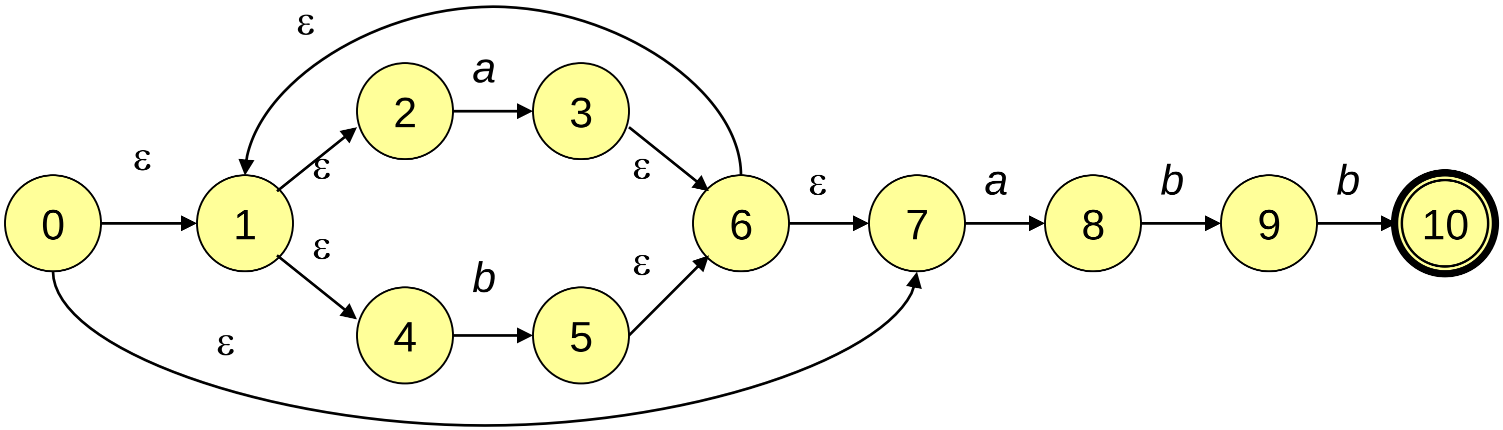
NFA to DFA: Algorithm

Initially, ε-closure the start state

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 then add U as unmarked to Dstates
        Dtable[T,a]:=U

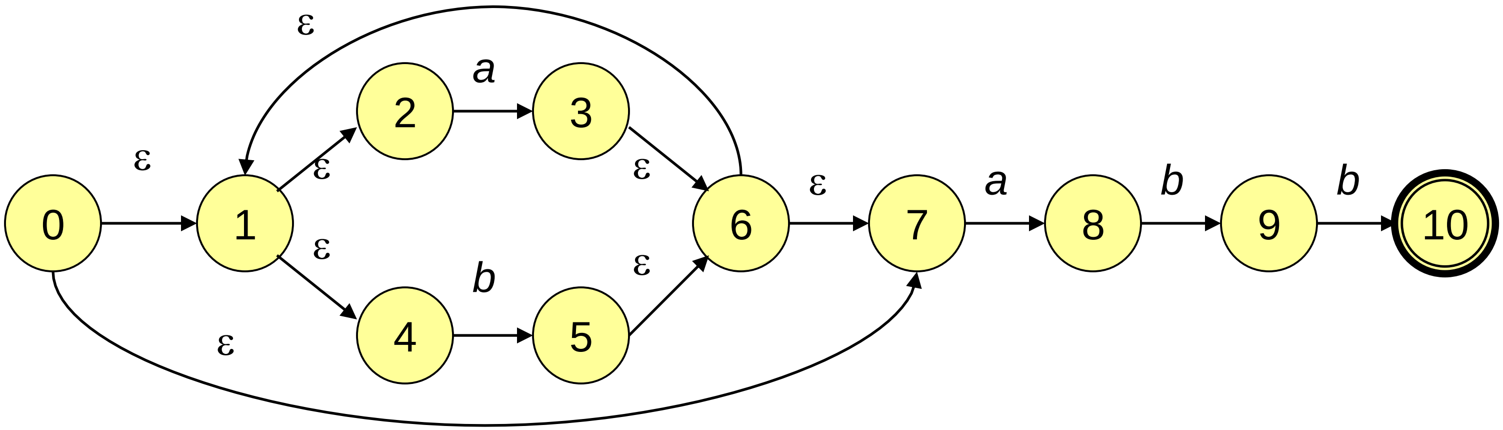
- Dstates (set of states for DFA) and Dtable form the DFA.
- Each state of DFA corresponds to a set of NFA states that NFA could be in after reading some sequences of input symbols.

NFA to DFA: Example



- A=ε-closure(0)={0,1,2,4,7}
- for each input symbol (*a* and *b*):
  - B=ε-closure(move(A,*a*))=ε-closure({3,8})={1,2,3,4,6,7,8}
  - C=ε-closure(move(A,*b*))=ε-closure({5})={1,2,4,5,6,7}
  - Dtable[A,*a*]=B; Dtable[A,*b*]=C

NFA to DFA: Example

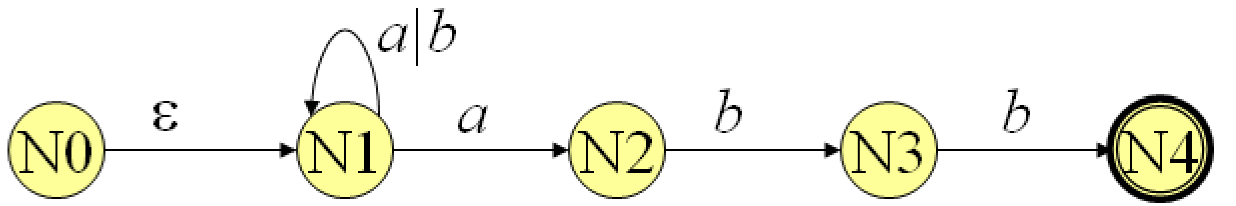


- A={0,1,2,4,7},
- Dtable[A,*a*]=**B**; Dtable[A,*b*]=**C**
  - B=ε-closure({3,8})={1,2,3,4,6,7,8} , C=ε-closure({5})={1,2,4,5,6,7}

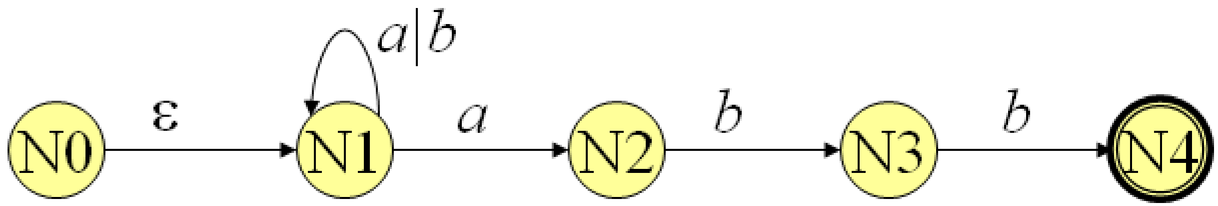
- Dtable[B,*a*]=B; Dtable[B,*b*]=**D**; Dtable[C,*a*]=B; Dtable[C,*b*]=C;
  - D=ε-closure({5,9})={1,2,4,5,6,7,9};

- Dtable[D,*a*]=B; Dtable[D,*b*]=**E**; Dtable[E,*a*]=B; Dtable[E,*b*]=C;
  - E= ε-closure({5,10})={1,2,4,5,6,7,10};

NFA to DFA: Example
(Another NFA for the same RE)



NFA to DFA: Example (Another NFA for the same RE)



Iteration	State	Contains	ε-closure(move(s,*a*))	ε-closure(move(s,*b*))
0	A	N0,N1	N1,N2	N1
1	B	N1,N2	N1,N2	N1,N3
	C	N1	N1,N2	N1
2	D	N1,N3	N1,N2	N1,N4
3	E	N1,N4	N1,N2	N1

- iteration 3 adds nothing new, so the algorithm stops.
- state E contains N4 (final state)

Outline

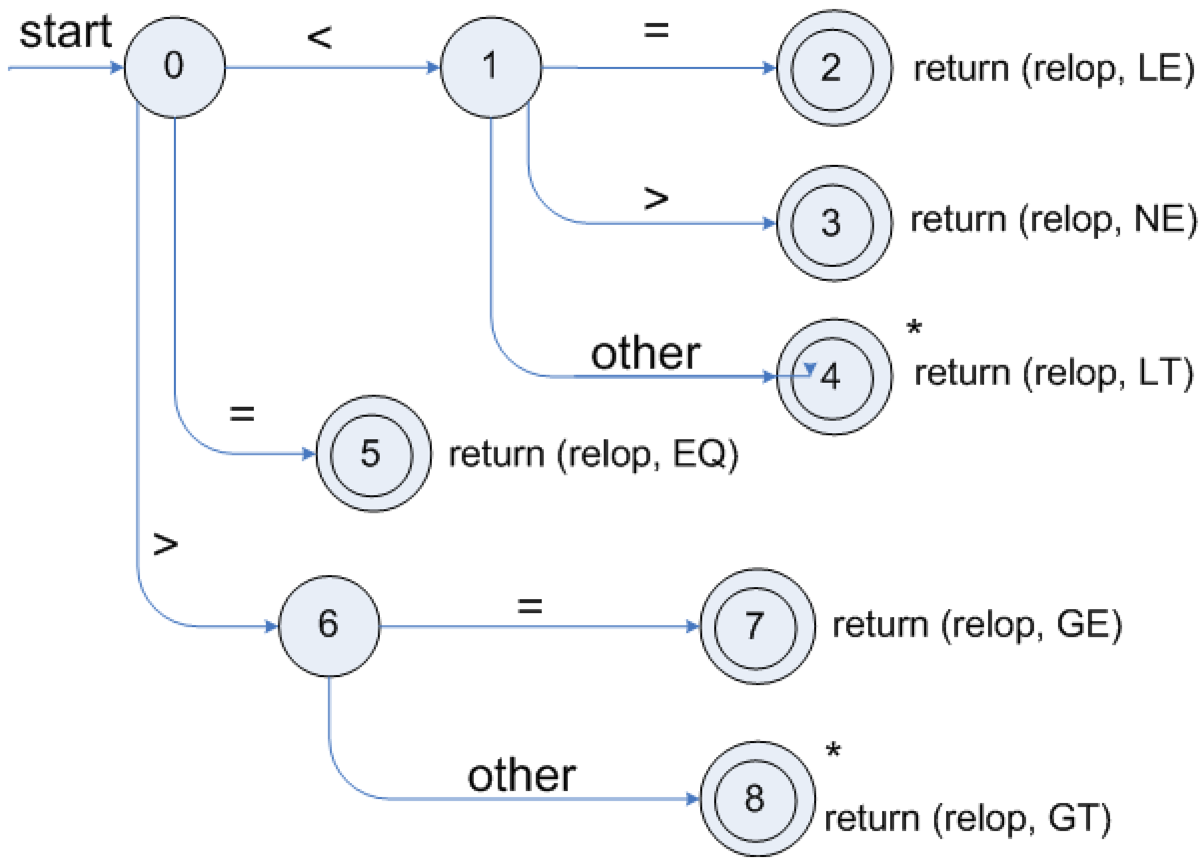
- Definition
- Associating Lexemes with Tokens
- Matching Regular Expressions
- From RE to Automata
- **Real-world Application**
- Error Recovery
- Toward Automation

Relop

Regular Expression	Token	Attribute-Value
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
< >	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

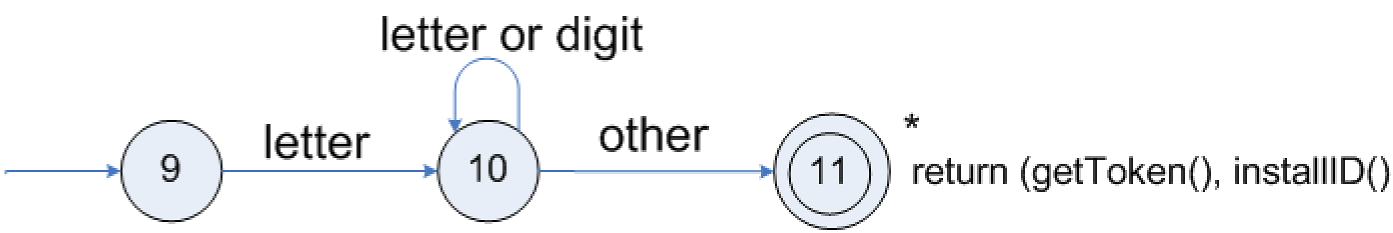
Transition diagrams

- Transition diagram for relop



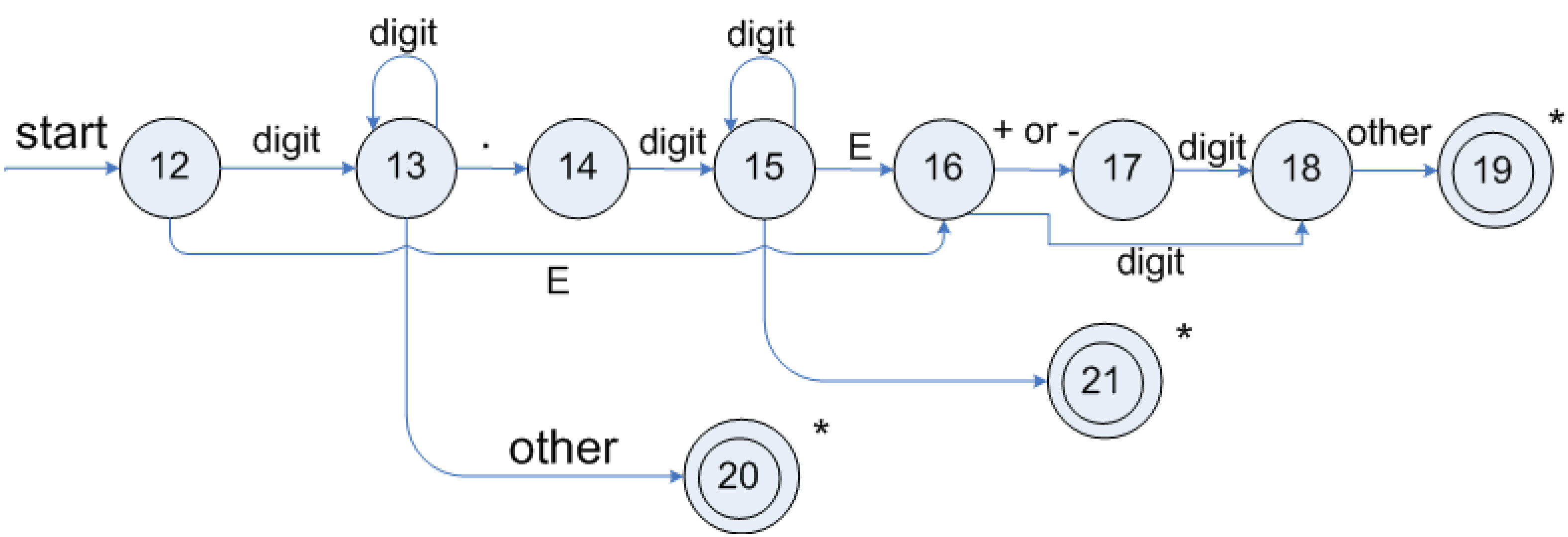
Transition diagrams

- Transition diagram for reserved words and identifiers



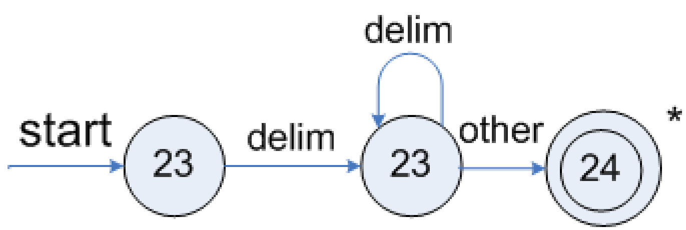
Transition diagrams

- Transition diagram for unsigned numbers



Transition diagrams

- Transition diagram for whitespace



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?

Whitespace Tokens

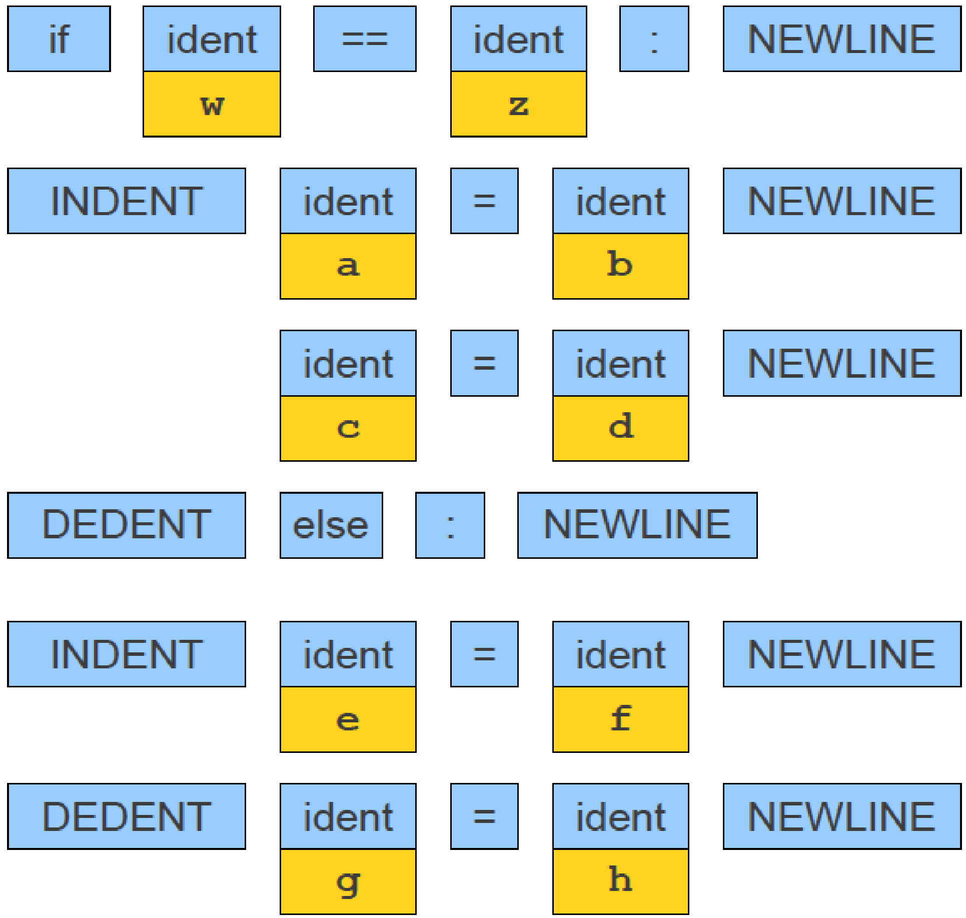
- Special tokens inserted to indicate changes in levels of indentation.
- NEWLINE marks the end of a line.
- INDENT indicates an increase in indentation.
- DEDENT indicates a decrease in indentation.
- Note that INDENT and DEDENT encode change in indentation, not the total amount of indentation.

Scanning Python

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

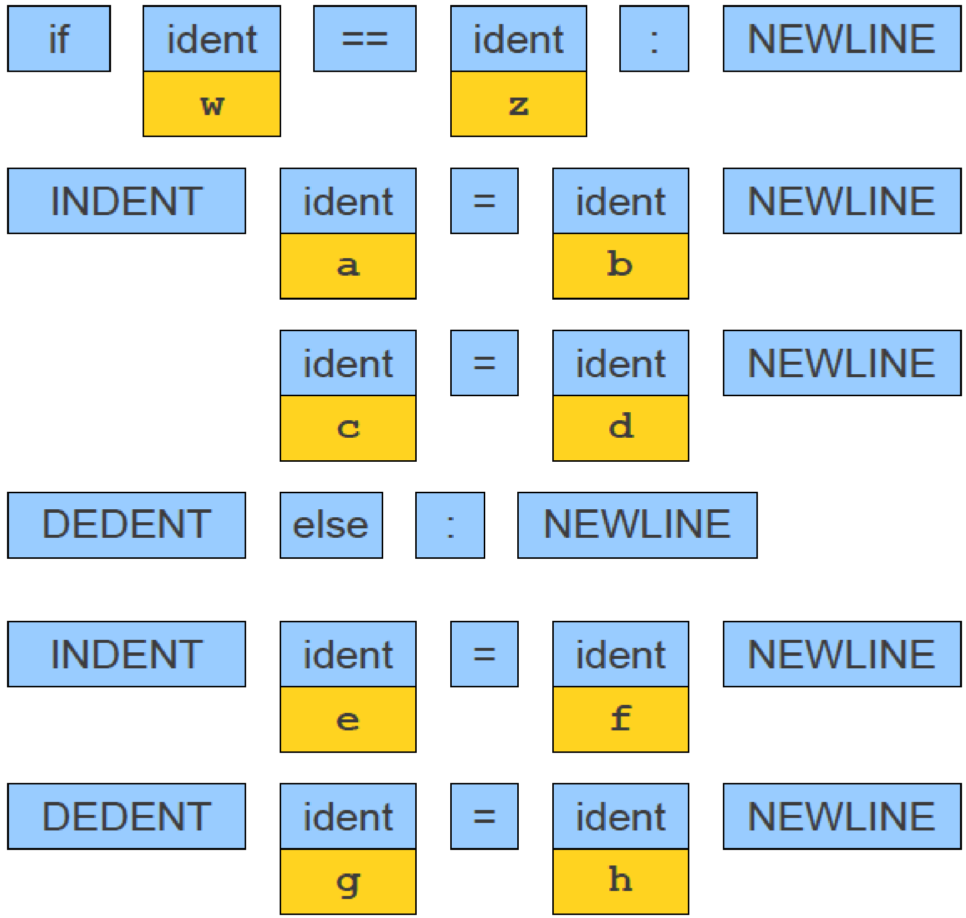
Scanning Python

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



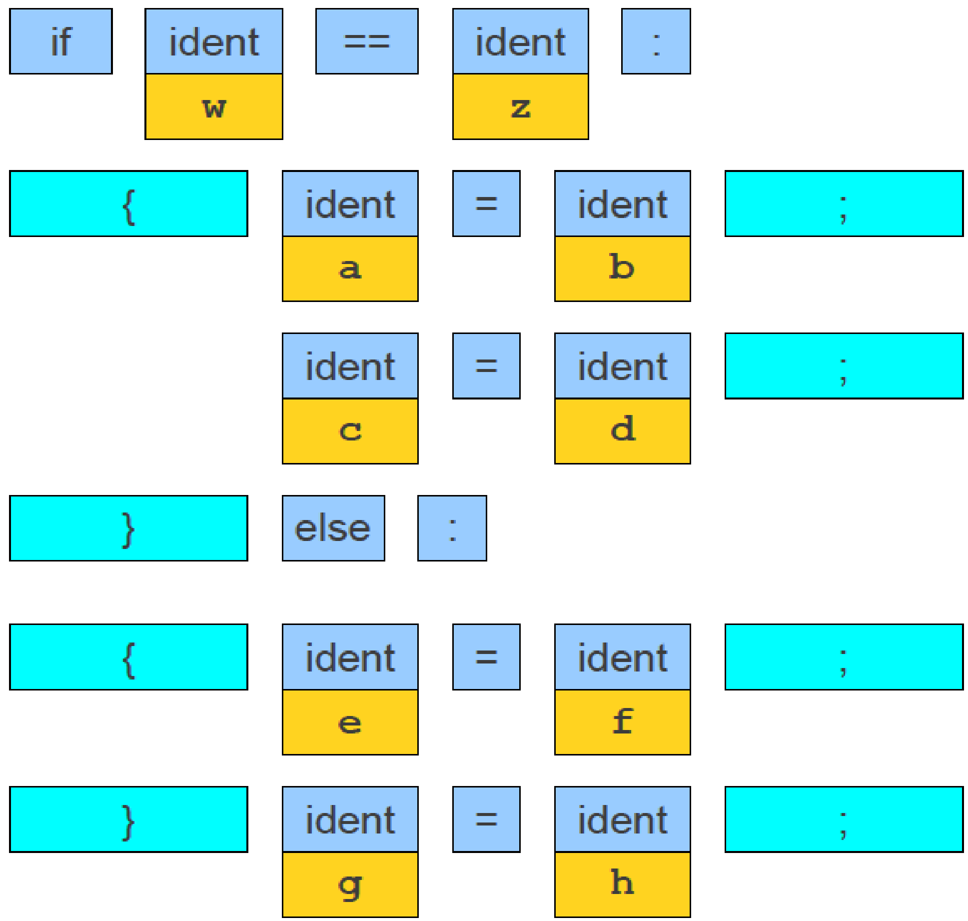
Scanning Python

```
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;
```



Where to INDENT/DEDENT?

- Scanner maintains a stack of line indentations keeping track of all indented contexts so far.
- Initially, this stack contains 0, since initially the contents of the file aren't indented.
- On a newline:
  - See how much whitespace is at the start of the line.
  - If this value exceeds the top of the stack:
    - Push the value onto the stack.
    - Emit an INDENT token.
- Otherwise, while the value is less than the top of the stack:
  - Pop the stack.
  - Emit a DEDENT token.

Source: http://docs.python.org/reference/lexical_analysis.html

General Practical Considerations

- Poor language design may complicate lexical analysis:
  - **`if then then = else; else else = then`** (PL/I)
  - **`DO5I=1,25`** vs **`DO5I=1.25`**
    - (Fortran: urban legend has it that an error like this caused a crash of an early NASA mission)
  - The development of a sound theoretical basis has influenced language design positively.

General Practical Considerations

- Template syntax in C++:
  - **`aaaa<mytype>`**
  - **`aaaa<mytype<int>>`**
    - (>> is an operator for writing to the output stream)
  - The lexical analyser treats the >> operator as two consecutive > symbols. The confusion will be resolved by the parser (by matching the <, >)

Outline

- Definition
- Associating Lexemes with Tokens
- Matching Regular Expressions
- From RE to Automata
- Real-world Application
- **Error Recovery**
- Toward Automation

Lexical errors

- Some errors are out of power of lexical analyzer to recognize

- It is able to recognize some errors when no pattern is found for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters
- Minimal Distance

Outline

- Definition
- Associating Lexemes with Tokens
- Matching Regular Expressions
- From RE to Automata
- Real-world Application
- Error Recovery
- **Toward Automation**

Implementation using DFA

- Option 1: Implement by hand using procedures
- Option 2: Use tool to generate table driven parser

Implement Using Procedures

- One procedure for each token
- Each procedure reads one character
- Choices implemented using if and switch statements

Implement Using Procedures

```
static char nextch;       // next unprocessed input character
void getch() { ... }      // advance to next input char

public Token getToken() {
    Token result;
    skipWhiteSpace();
    if (no more input) {
        result = new Token(Token.EOF); return result;
    }
    switch(nextch) {
        case '(': result = new Token(Token.LPAREN); getch(); return result;
        case ‘)': result = new Token(Token.RPAREN); getch(); return result;
        case ‘;': result = new Token(Token.SCOLON); getch(); return result;
        …
        case ‘0': ... case ‘9':
        …
        case 'a': ... case 'z':
        …
```

Implement Using Procedures

- Proc:
  - Straightforward to write
  - Fast

- Cons
  - A fair amount of tedious work
  - May have subtle differences from the language specification

Implementation Using Transition Table

- Rows: states of DFA
- Columns: input characters
- Entries: action
  - Go to next state
  - Accept token, go to start state
  - Error

Implementation Using Transition Table

- An easy (computerized) implementation of a transition diagram is a **<u>transition table</u>**: a column for each input symbol and a row for each state. An entry is a set of states that can be reached from a state on some input symbol.

- E.g.:

state	'r'	digit
0	1	-
1	-	2
2(final)	-	2

Implementation Using Transition Table

- If we know the transition table and the final state(s) we can build directly a recognizer that detects acceptance:

```
char=input_char();
state=0;    // starting state
while (char != EOF) {
    state ← table(state,char);
    if (state == '-') return failure;
    word=word+char;
    char=input_char();
}
if (state == FINAL) return acceptance; else return
  failure;
```

Example

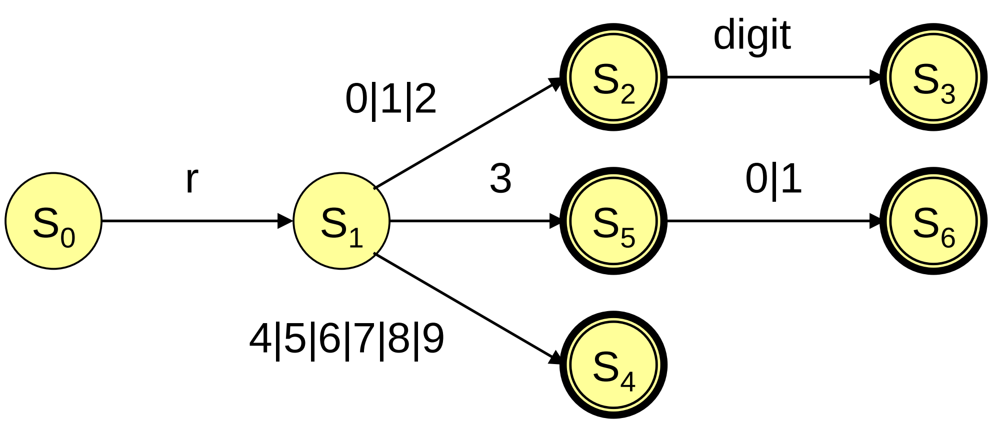
- What is this RE for?
- Produce the DFA for it:

$Register \rightarrow r\ ((0|1|2)\ (Digit|\varepsilon)\ |\ (4|5|6|7|8|9)\ |\ (3|30|31))$

Example

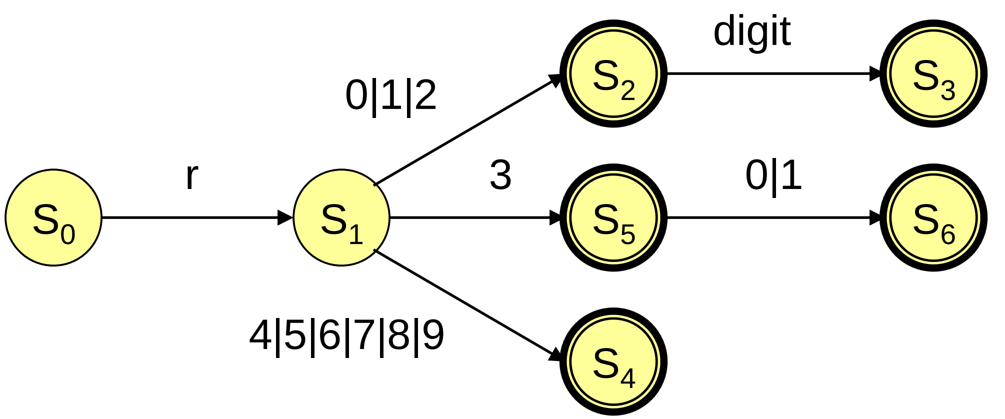
- What is this RE for?
- Produce the DFA for it:

Register $\rightarrow r\ ((0|1|2)\ (Digit|\varepsilon)\ |\ (4|5|6|7|8|9)\ |\ (3|30|31))$



recognize r0 through r31

Example



State	'r'	0,1	2	3	4,5,…,9
0	1	-	-	-	-
1	-	2	2	5	4
2(final)	-	3	3	3	3
3(final)	-	-	-	-	-
4(final)	-	-	-	-	-
5(final)	-	6	-	-	-
6(final)	-	-	-	-	-

Automatic Lexical Analyser Construction

- To convert a specification into code:
  - Write down the RE for the input language
  - Convert the RE to a NFA
  - Build the DFA that simulates the NFA
  - Shrink the DFA

Lex/Flex: Generating Lexical Analysers

- Flex is a tool for generating scanners
  - Programs which recognized lexical patterns in text
- Lex input consists of 3 sections:
  - Regular expressions
  - Pairs of regular expressions and C code
  - Auxiliary C code

Lex/Flex: Generating Lexical Analyzers

- When the lex input is compiled, it generates as output a C source file lex.yy.c
  - The source contains a routine yylex()
- After compiling the C file, the executable will start isolating tokens from the input according to the regular expressions
  - For each token, the associated code will be executed
  - The array char yytext[] contains the representation of a token

Flex Example

```
%{
#define ERROR -1
int line_number=1;
%}
whitespace    [ \t]
letter        [a-zA-Z]
digit         [0-9]
integer       ({digit}+)
l_or_d        ({letter}|{digit})
identifier    ({letter}{l_or_d}*)
operator      [-+*/]
separator     [;,(){}]
%%
{integer}     {return 1;}
{identifier} {return 2;}
{operator}|{separator}  {return (int)yytext[0];}
{whitespace} {}
\n            {line_number++;}
.             {return ERROR;}
%%
int yywrap(void) {return 1;}
int main() {
    int token;
    yyin=fopen("myfile","r");
    while ((token=yylex())!=0)
       printf("%d %s \n", token, yytext);
    printf("lines %d \n",line_number);
}
```

Input file ("myfile")

```
123+435+34=aaaa
329*45/a-34*(45+23)**3
bye-bye
```

Output:

```
 1 123
43 +
 1 435
43 +
 1 34
-1 =
 2 aaaa
 1 329
42 *
 1 45
47 /
 2 a
45 -
 1 34
42 *
40 (
 1 45
43 +
 1 23
41 )
42 *
42 *
 1 3
 2 bye
45 -
 2 bye
lines 4
```

Summary

- Lexical Analysis turns a stream of characters into a stream of tokens
  - A largely automatic process.
    - REs are powerful enough to specify scanners
    - DFAs have good properties for an implementation

Summary

- Lexical analysis splits input text into tokens holding a lexeme and an attribute
- Lexemes are sets of strings often defined with regular expressions
- The generalized transition diagram is a **finite automaton.** It can be:
  - **Deterministic (**DFA)
  - **Non-Deterministic** (NFA)
- Regular expressions can be converted to NFAs and from there to DFAs

Reading

- Aho2, Sections 2.2; 3.1-3.4; 3.5 (lex); 3.6-3.7; 3.9.6
- Aho1, pp. 25-29; 84-87; 92-105; 113-125; 141-144, 105-111 (lex)
- Hunter, Chapter 2 (too detailed); Sec. 3.1 -3.3 (too condensed)
- Grune 1.9; 2.1-2.5; 2.1.6.1-2.1.6.6; pp.86-96
- Cooper, Sections 2.1-2.3; 2.4-2.4.3; pp.55-72

Question?