COSC252: Programming Languages:

Lexical Analysis and Regular Languages

Jeremy Bolton, PhD

Asst Teaching Professor

A VERY special thanks to the authors of THE dragon book and other contributors at Stanford

GEORGETOWN UNIVERSITY

Notes

Notes: Read ALSU 1 – 2

Topics covered in CH1 will be revisited in more detail throughout

Sign up for Gradiance and complete Assignment #1

Outline

I. Lexical Analysis
   I. Scanning
   II. Regular Languages
      I. Regular Expressions
      II. Finite State Automata

The Basics of Programming Languages

- Before we can learn about each of these steps, we will formalize the concepts and nomenclature

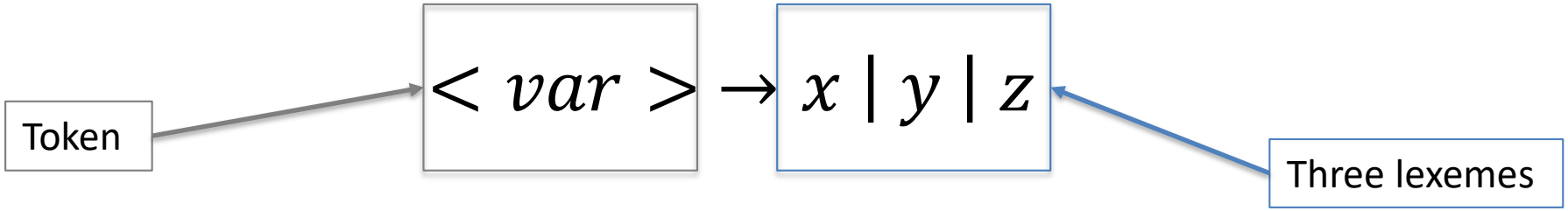
Overview: Characterizing a Language

- Noam Chomsky
  - MIT Linguist
  - Published work on categories of Languages / Grammars, 1950s
    - Two categories are commonly used in Programming Languages
      - Context-Free Languages
        - » Often used to characterize programming language sentence structure
      - Regular Languages
        - » Often used to characterize the structure of lexemes / tokens
- John Backus and Peter Naur developed a formal notation for generating a Context Free Language, (similar to the notation used by Chomsky)
  - BNF (Bachus Naur Form)

Recall: Regular Languages

- Remember: Tokens are categories of lexemes and lexemes are terminals (the smallest syntactic unit)
  - If the number of lexemes per token category are small, then we need only list them out. That is, there is no need to use a Regular Language to model all of the lexemes in the Token Class (Regular Language)

- In our previous example, there were only 3 possible variable tokens.



- If a token class has many possible lexemes, then a formal rule system can be used to help define all possible lexemes (rather than list them all out!). *For example, try listing out all possible variable names in C++.*

Overview: Regular Languages to define tokens.

- Could you use BNF to define the set of all possible variables or ints? YES. But instead we will use regular languages and regular expressions. (More on this later)

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow see\ regular\ expression\ for\ variables.$$
$$< int > \rightarrow see\ regular\ expression\ for\ integers.$$

Groups of abstractions: compositions of terminals and nonterminals. We use CFGs to model these constructs.

Groups of tokens (sets of terminals)

We will use regular grammars to model these constructs

GEORGETOWN UNIVERSITY

Overview: Regular Languages

- Both context free languages and regular languages are useful in programming languages (Chomsky)
- We have seen that BNF is a great way to formalize context free grammars (to define a context free language)

$$
\begin{aligned}
&< stmt > \rightarrow < var > = < expr > \\
&< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term > \\
&< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg > \\
&< neg > \rightarrow < var > \mid -< var > \\
&< var > \rightarrow x \mid y \mid z
\end{aligned}
$$

- BNF defines sequences of tokens (sentences) for a languages.
- Regular languages are often used to model the tokens of a language

GEORGETOWN UNIVERSITY

Overview: Regular Languages

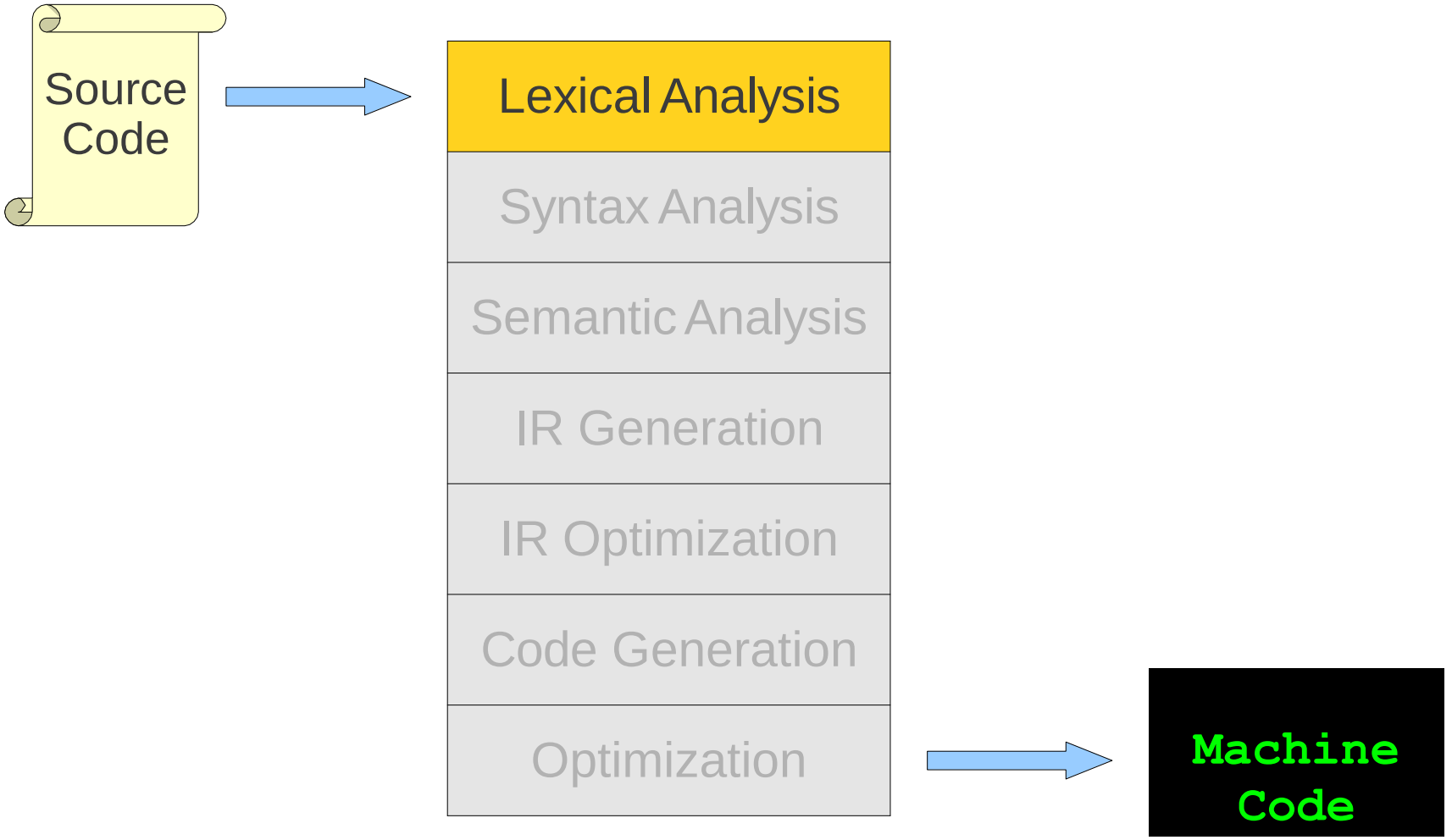
- Regular Languages used to model Tokens

- Regular Expression is a generator for a regular language (it defines a regular language)
  - Consists of characters and regular operations
  - Just as BNF is a generator and define a context free language.

- Regular Operations:
  - Concatenation
  - Repetition
    Repeat 0 or more times: *
    Repeat 1 or more times: +
  - Selection: |

  - Other common symbols used
    Optional: ?
    Any character: .
    Short hand for 1 element in a set: [ ]

Where We Are

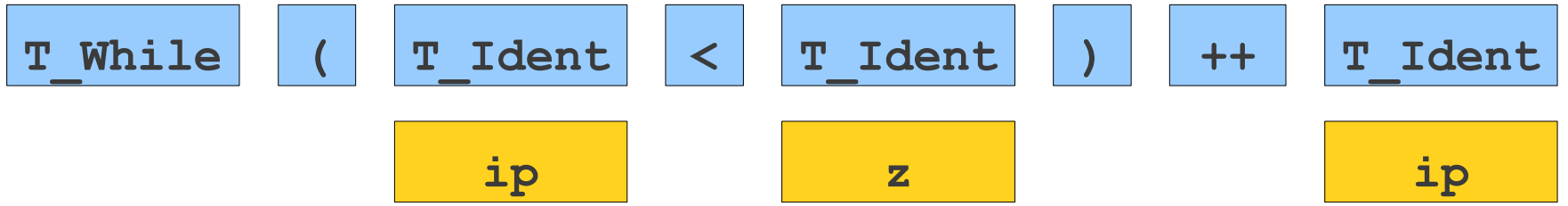

Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

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

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

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

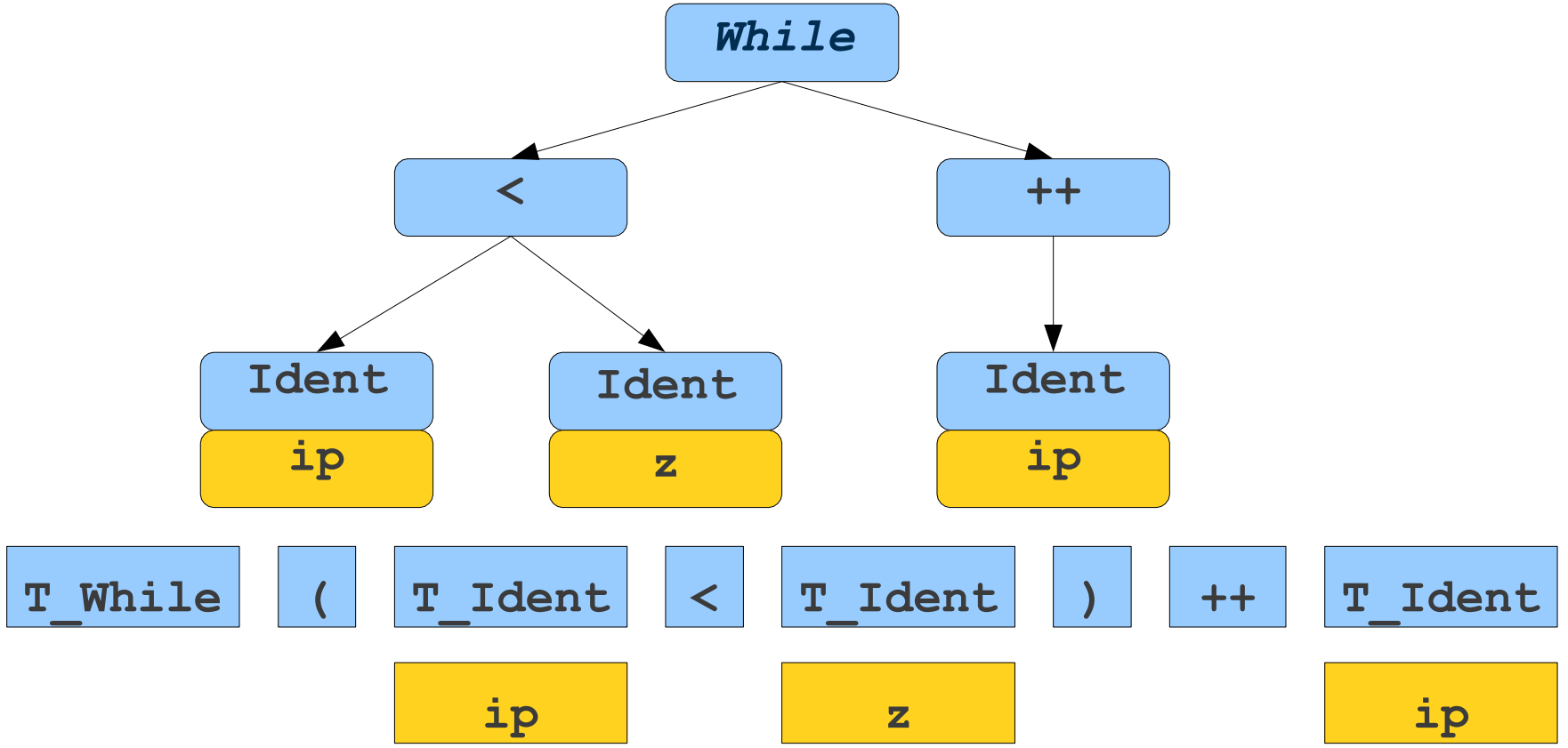
GEORGETOWN UNIVERSITY



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

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

GEORGETOWN UNIVERSITY



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

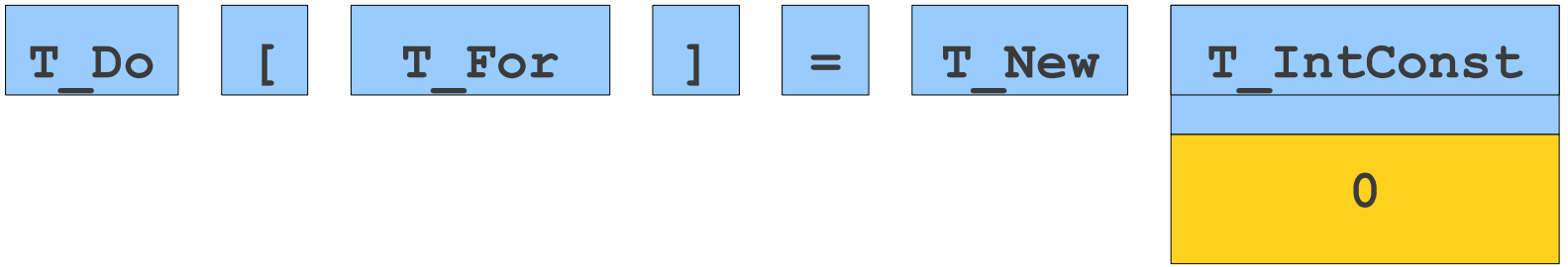
```
while (ip < z)
    ++ip;
```

GEORGETOWN UNIVERSITY

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

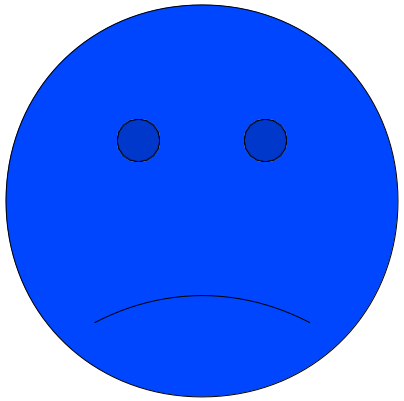
d	o	[	f	o	r	]		=		n	e	w		0	;

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



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

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



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

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

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

GEORGETOWN UNIVERSITY

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

T_While

Scanning a Source File

The piece of the original program fromwhich 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

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

T_While

Scanning a Source File

Scanning a Source File

Scanning a Source File

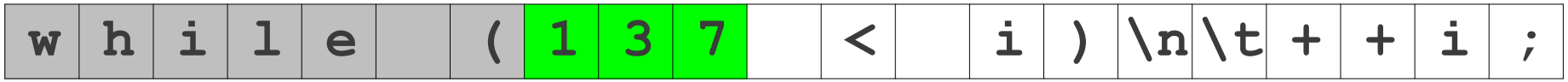
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While	(

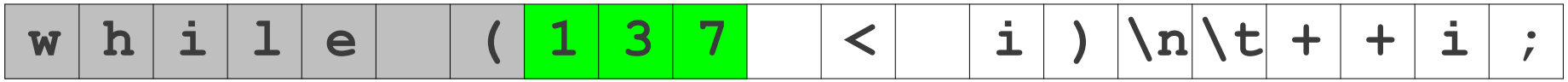
Scanning a Source File

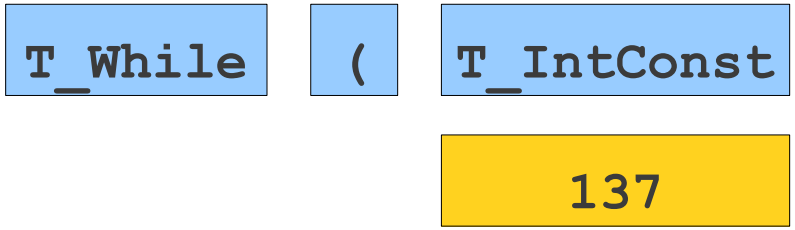
Scanning a Source File

Scanning a Source File

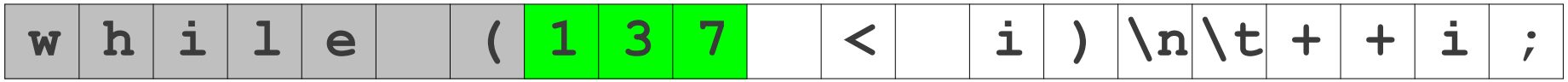


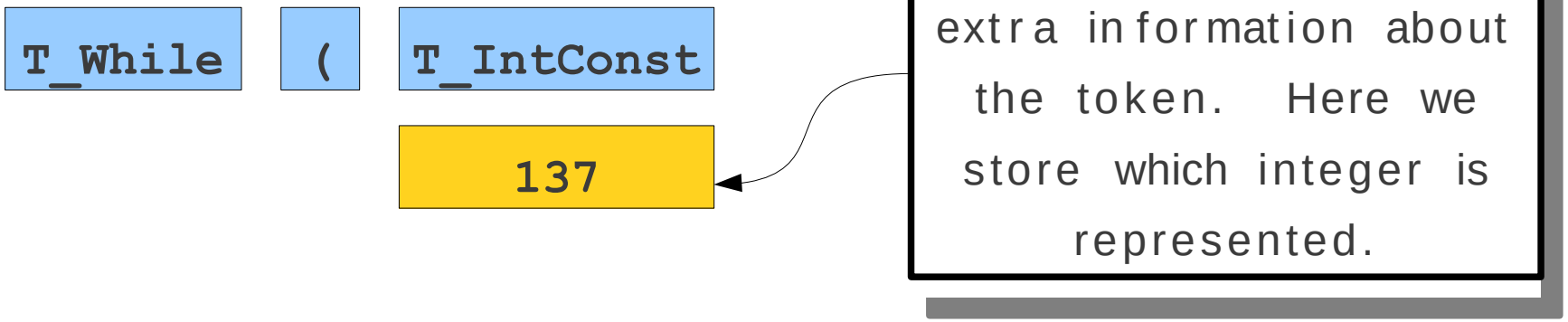
Scanning a Source File





Scanning a Source File





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	}
<<	;
=	<
(	[
)	]
++	

What Tokens are Useful Here?

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

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

Identifier
IntegerConstant
```

GEORGETOWN UNIVERSITY

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

- 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)))
```

Scanning is Hard

- C++: Nested template declarations

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

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

Thanks to Prof. Alex Aiken

Scanning is Hard

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

Scanning is Hard

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

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

Thanks to Prof. Alex Aiken

Scanning is Hard

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

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

Thanks to Prof. Alex Aiken

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 lexemes.

Thanks to Prof. Alex Aiken

Challenges in Scanning

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

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like `if` and `while` 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 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.
  - EG: Use 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, including the **`flex`** tool we will use in this course.

Atomic Regular Expressions

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

Compound Regular Expressions

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

Operator Precedence

- Regular expression operator precedence is

$$(R)$$

$$R^*$$

$$R_1 R_2$$

$$R_1 \mid R_2$$

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

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
1111101111011111

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 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:

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*

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*

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*

11110111
111111
0111
0

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*

11110111
111111
0111
0

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?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*)*

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*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

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*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

GEORGETOWN UNIVERSITY

Applied Regular Expressions

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

a^+ (.aa*)* @ aa*.aa* (.aa*)*

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

GEORGETOWN UNIVERSITY

Applied Regular Expressions

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

$$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

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

$$a^+ \quad (.a^+)^* \quad @ \quad a^+.a^+ \quad (.a^+)^*$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

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

$$a^+ \quad (.a^+)^* \quad @ \quad a^+ \quad (.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

Applied Regular Expressions

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

$$a^+(.a^+)^*@a^+(.a^+)^+$$

cs143@cs.stanford.edu
first.middle.last@mail.site.org
barack.obama@whitehouse.gov

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)

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)

42
+1370
-3248
-9999912

GEORGETOWN UNIVERSITY

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)

42
+1370
-3248
-9999912

Applied Regular Expressions

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

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

42
+1370
-3248
-9999912

GEORGETOWN UNIVERSITY

Applied Regular Expressions

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

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

42
+1370
-3248
-9999912

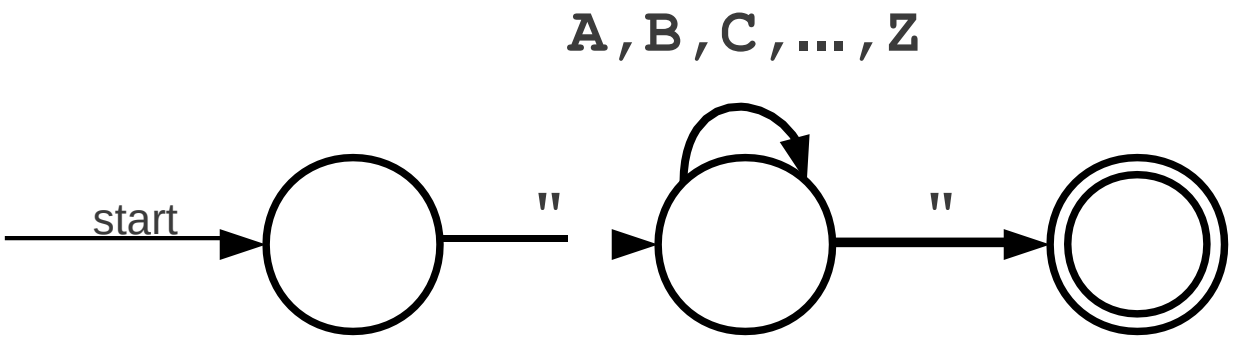
Matching Regular Expressions

GEORGETOWN UNIVERSITY

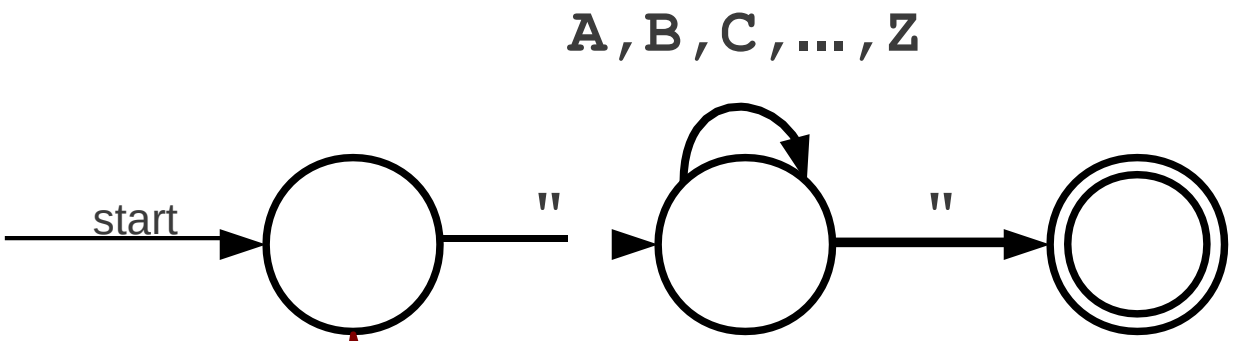
Implementing Regular Expressions

- 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.
- Automata are best explained by example...

A Simple Automaton

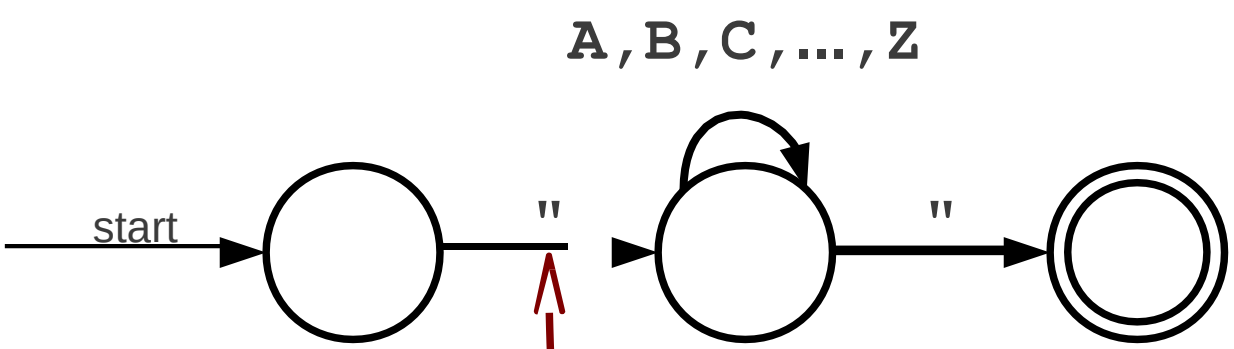


A Simple Automaton



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

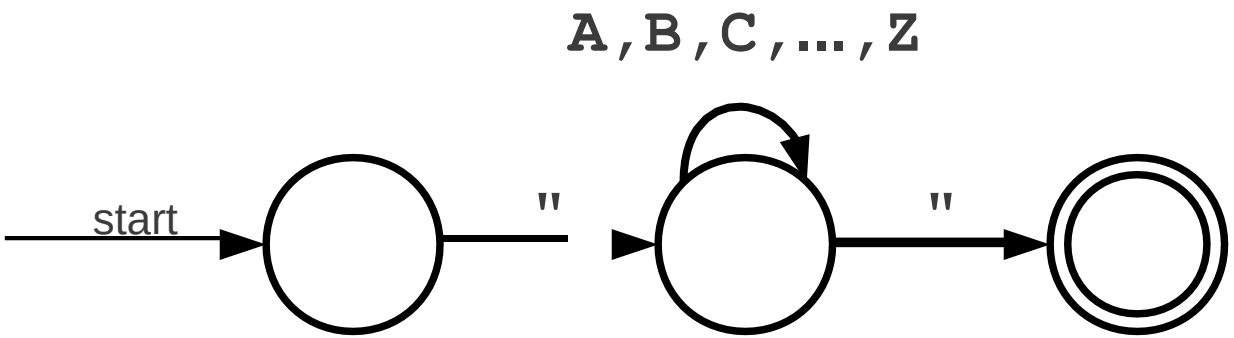
A Simple Automaton



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

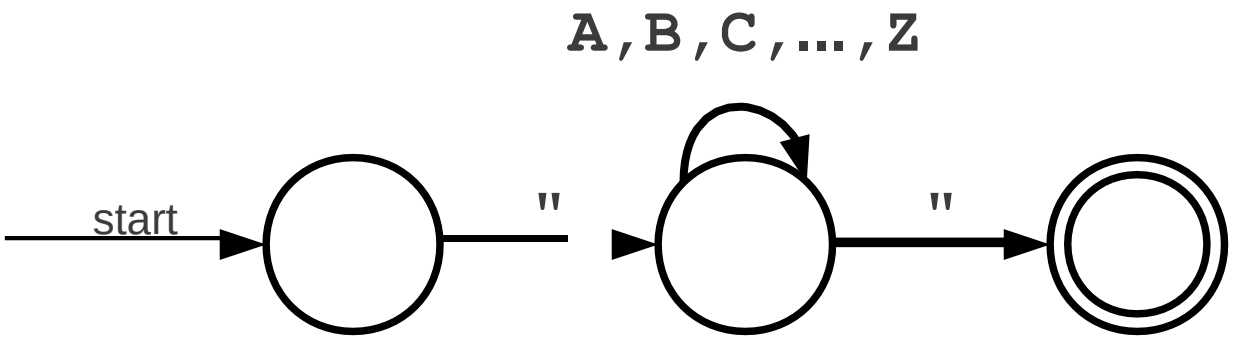
GEORGETOWN UNIVERSITY

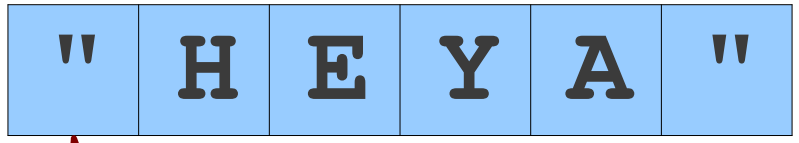
A Simple Automaton



"	H	E	Y	A	"

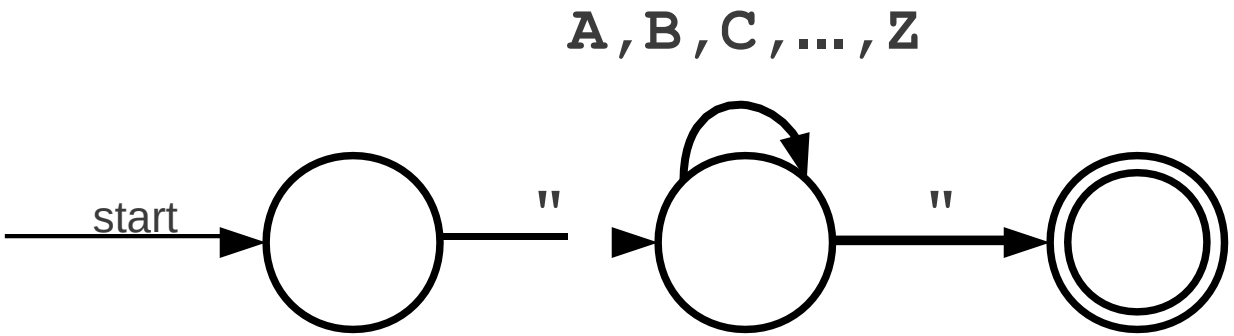
A Simple Automaton

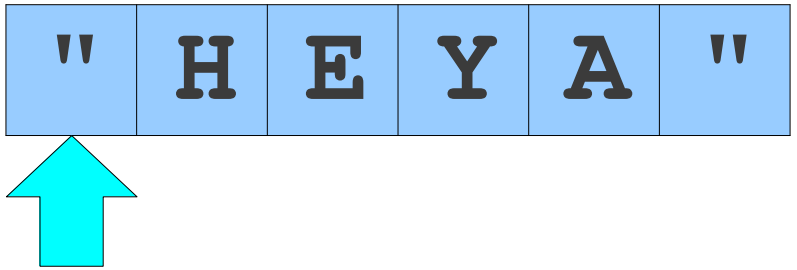




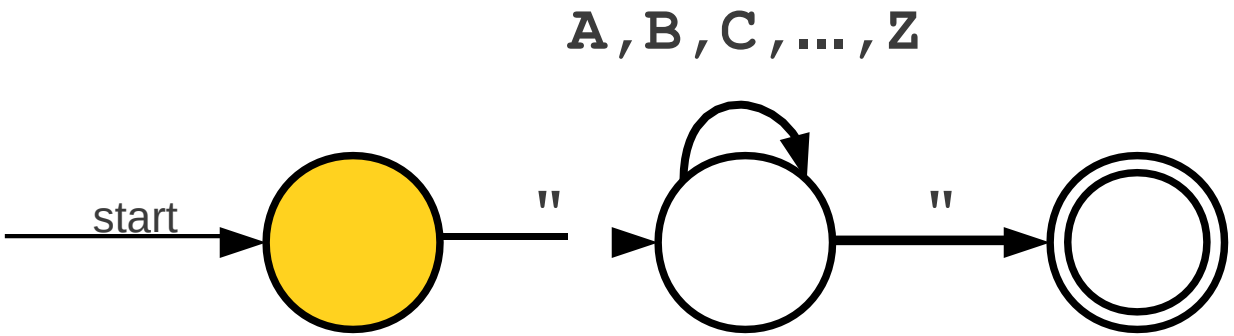
The automaton takes a string as input and decides whether to accept or reject the string.

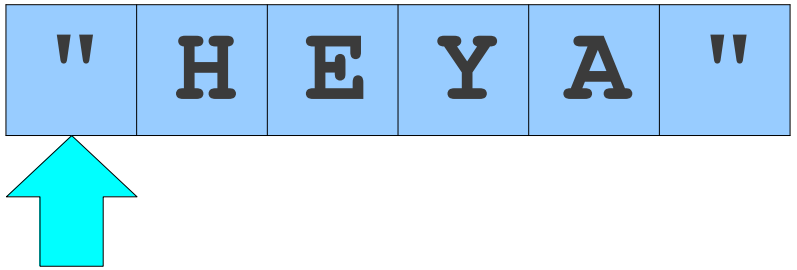
A Simple Automaton



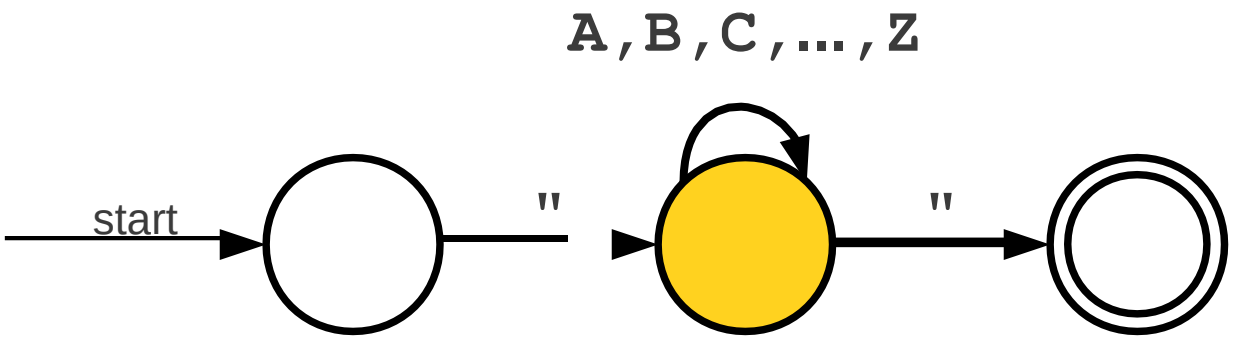


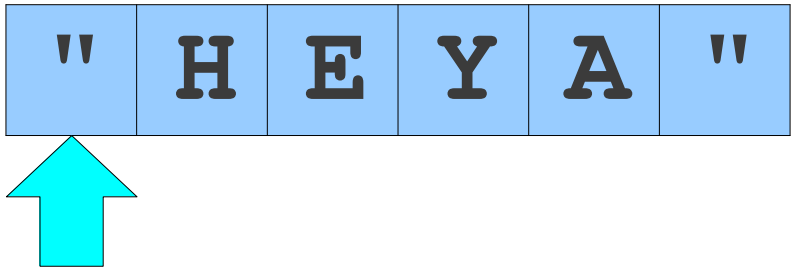
A Simple Automaton



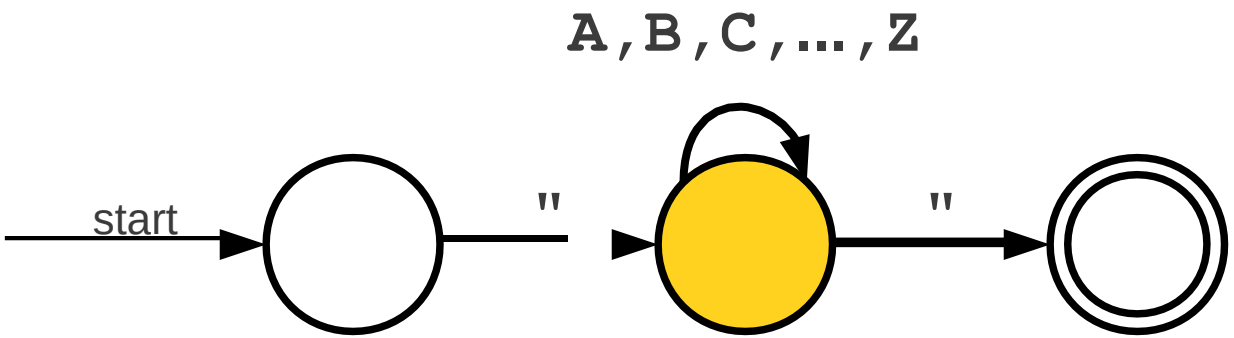


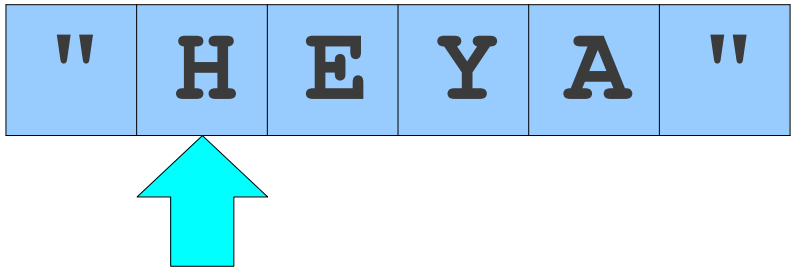
A Simple Automaton



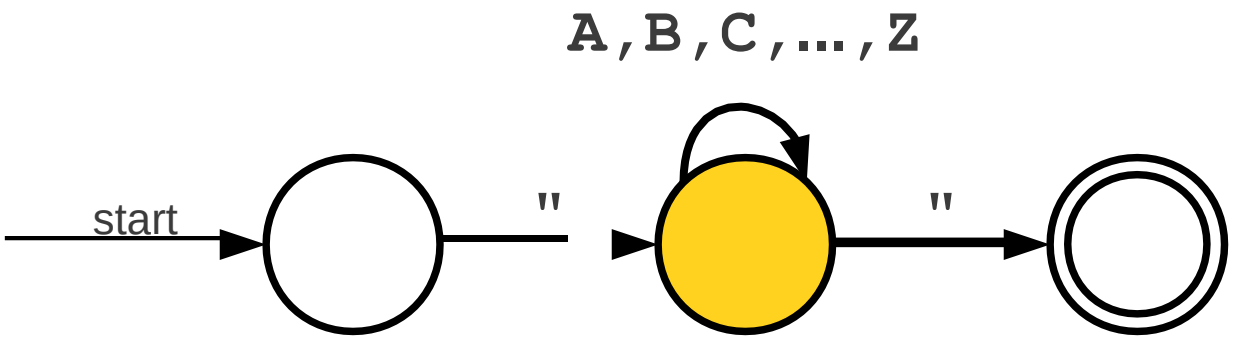


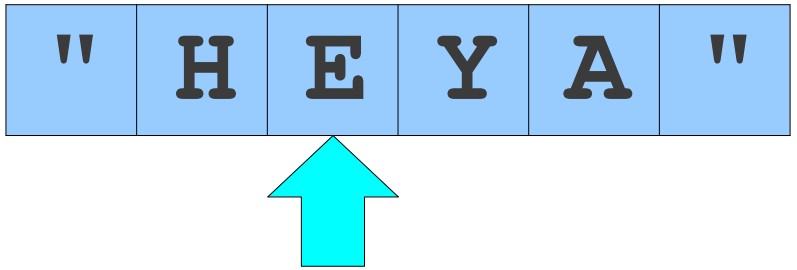
A Simple Automaton



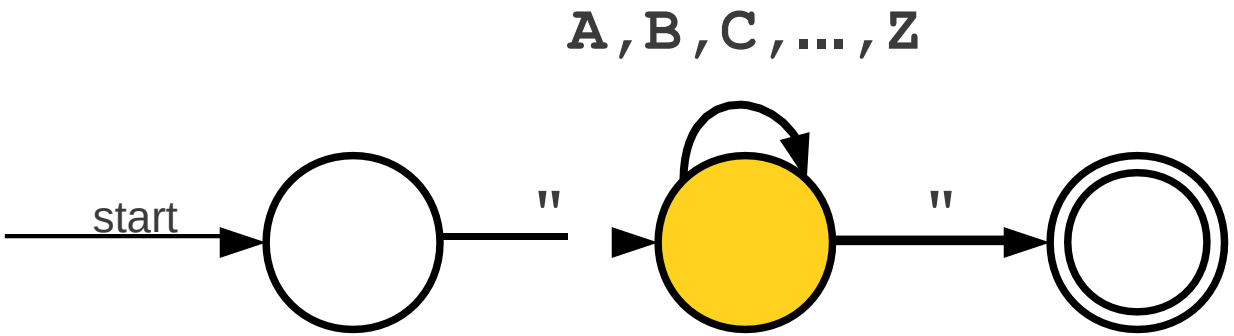


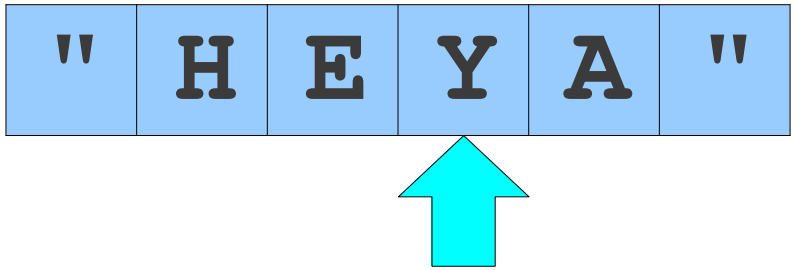
A Simple Automaton



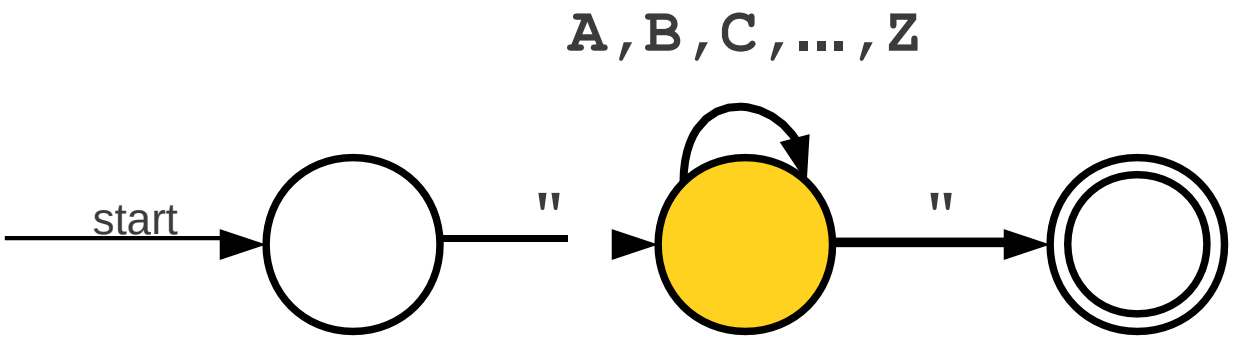


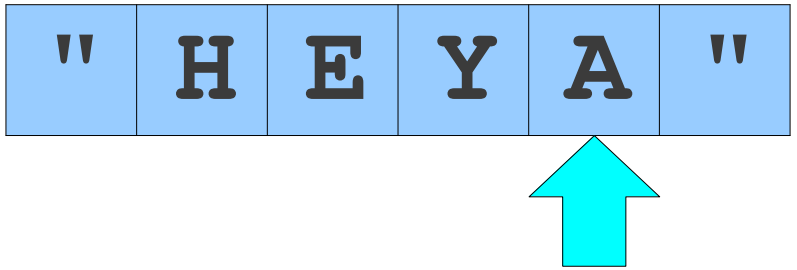
A Simple Automaton



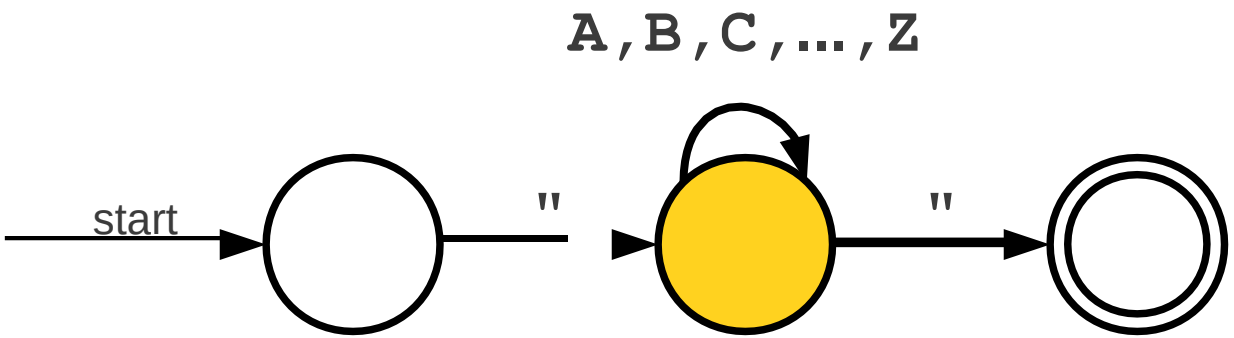


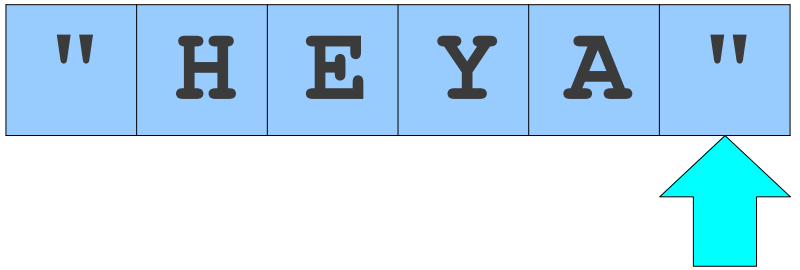
A Simple Automaton



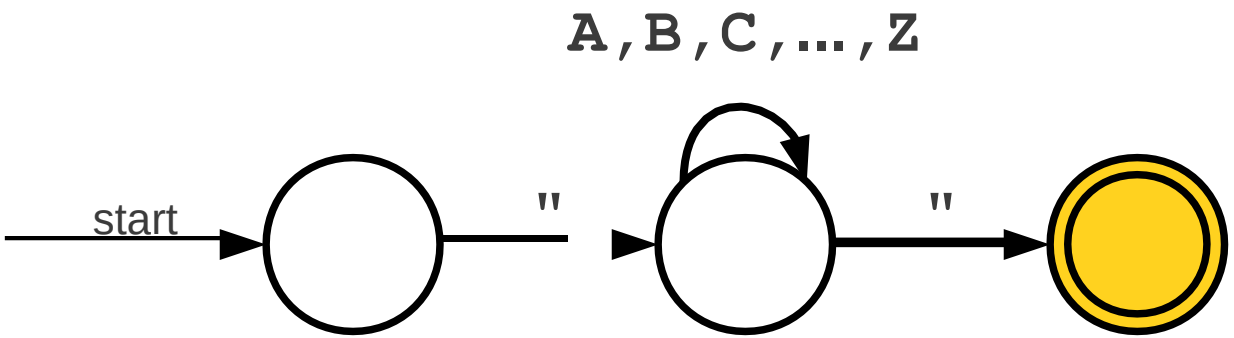


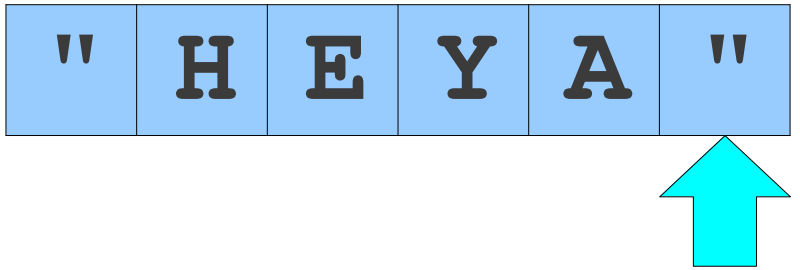
A Simple Automaton



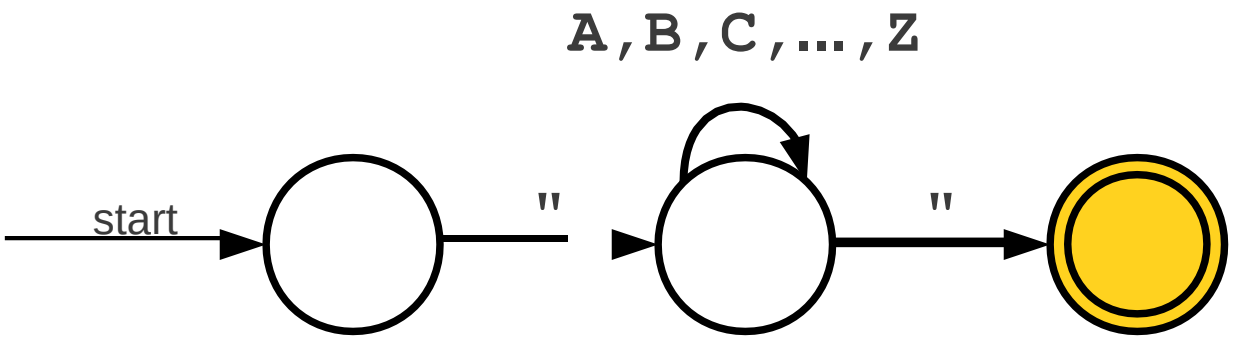


A Simple Automaton



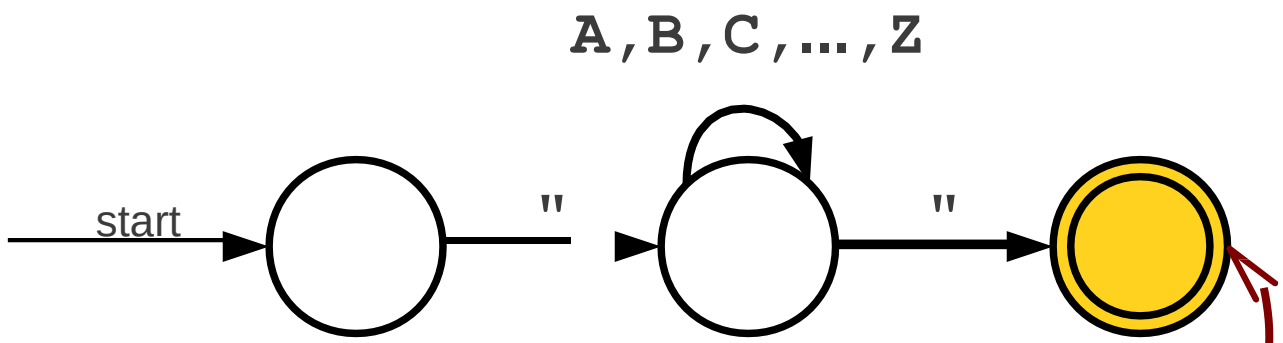


A Simple Automaton



"	H	E	Y	A	"

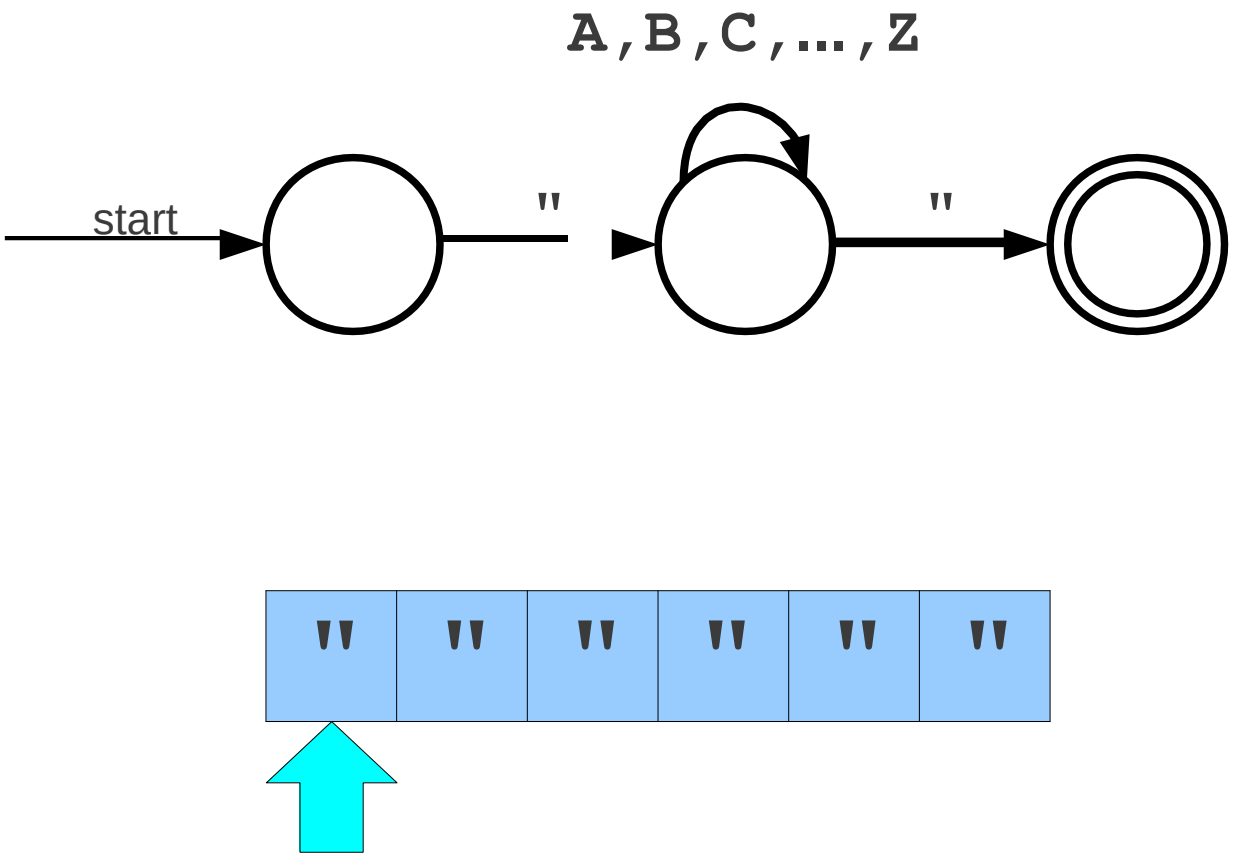
A Simple Automaton



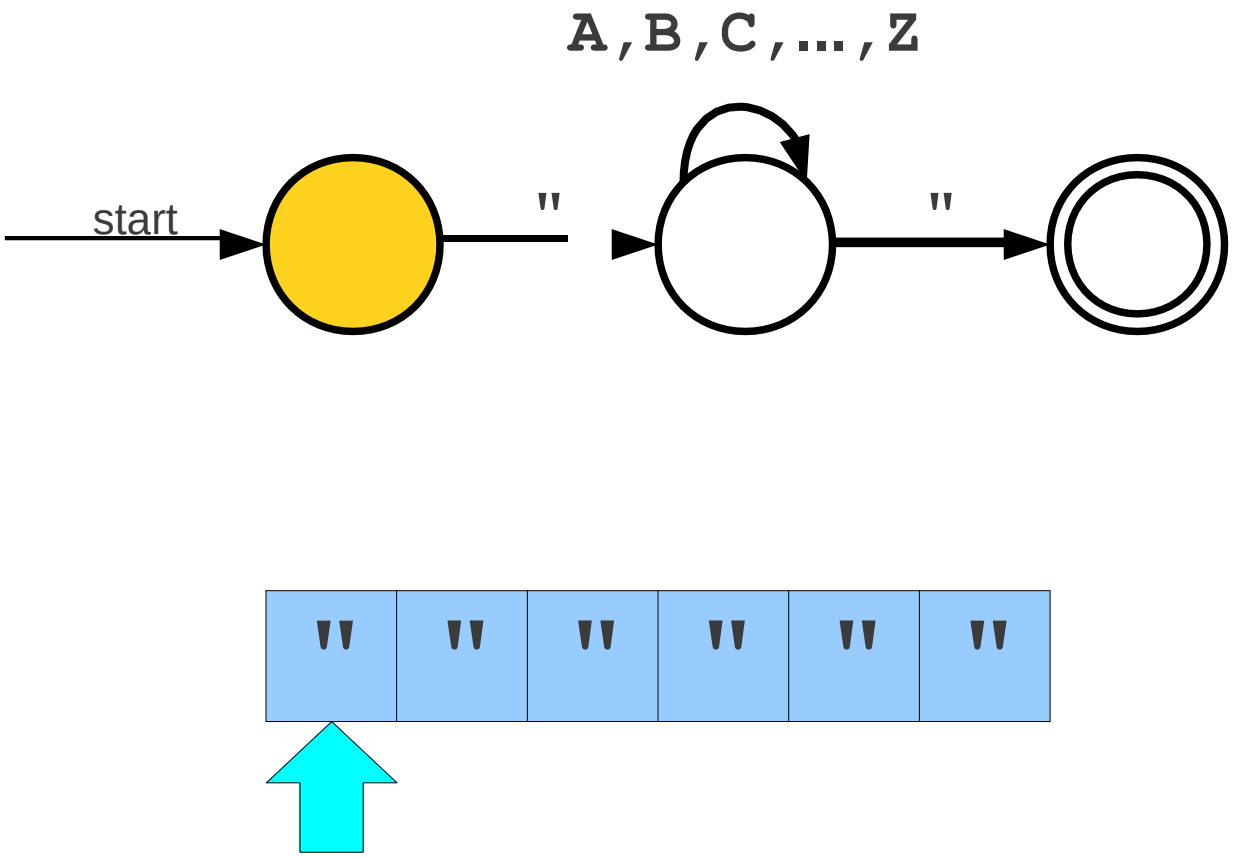
"	H	E	Y	A	"

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

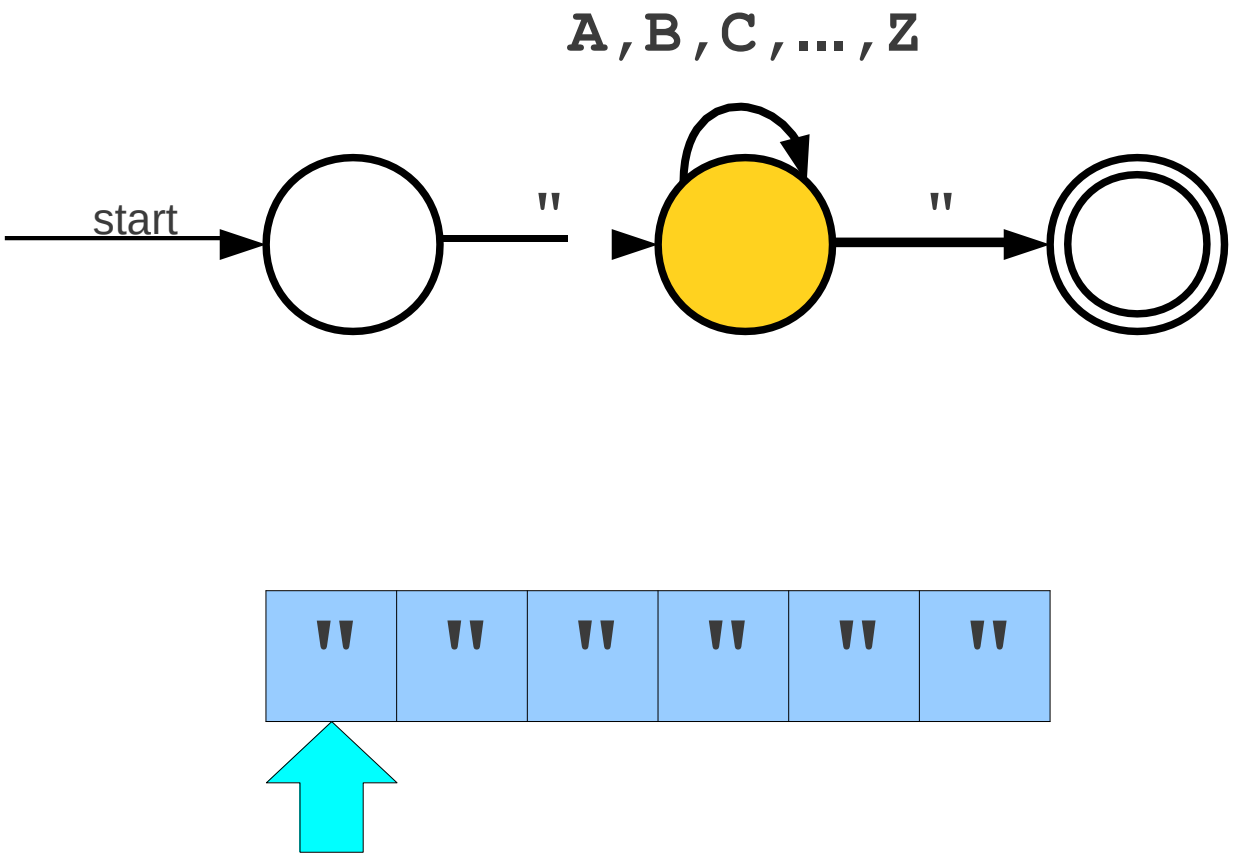
A Simple Automaton



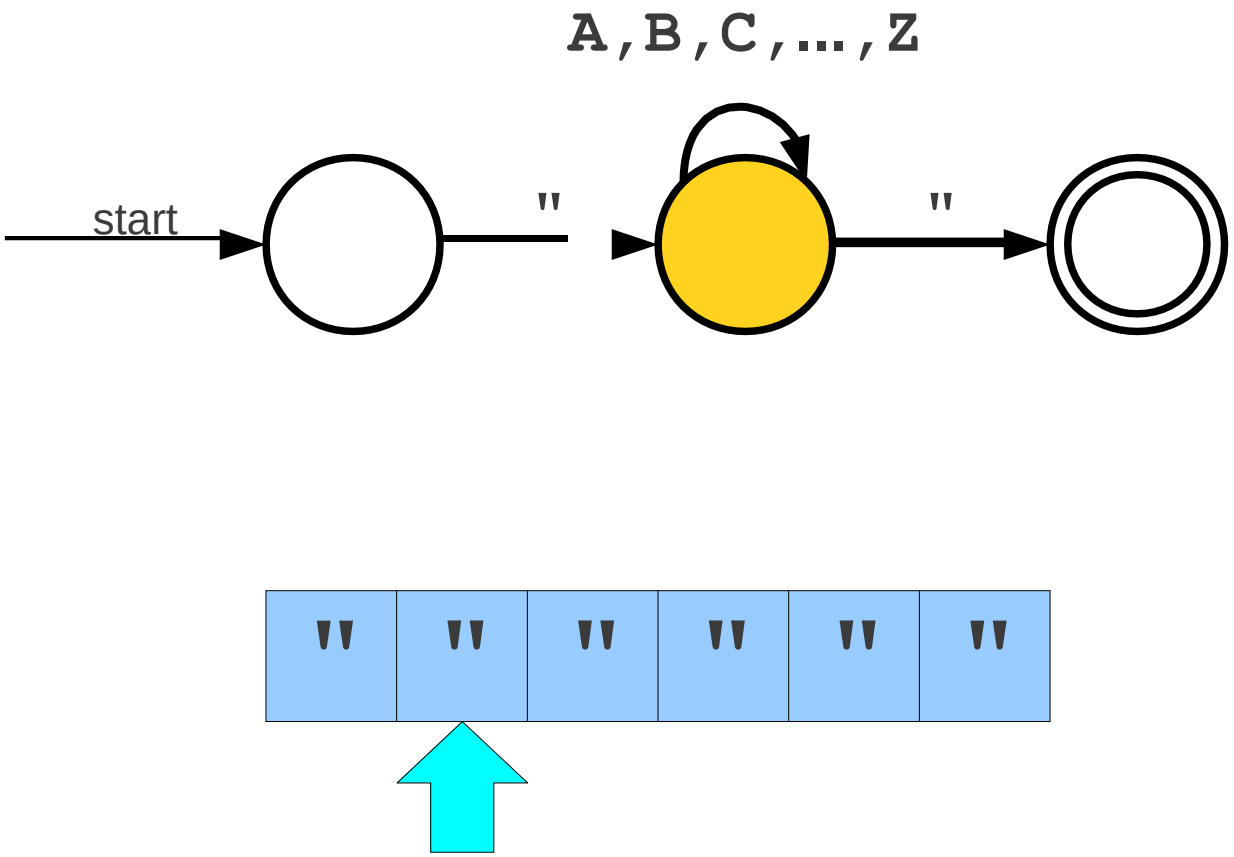
A Simple Automaton



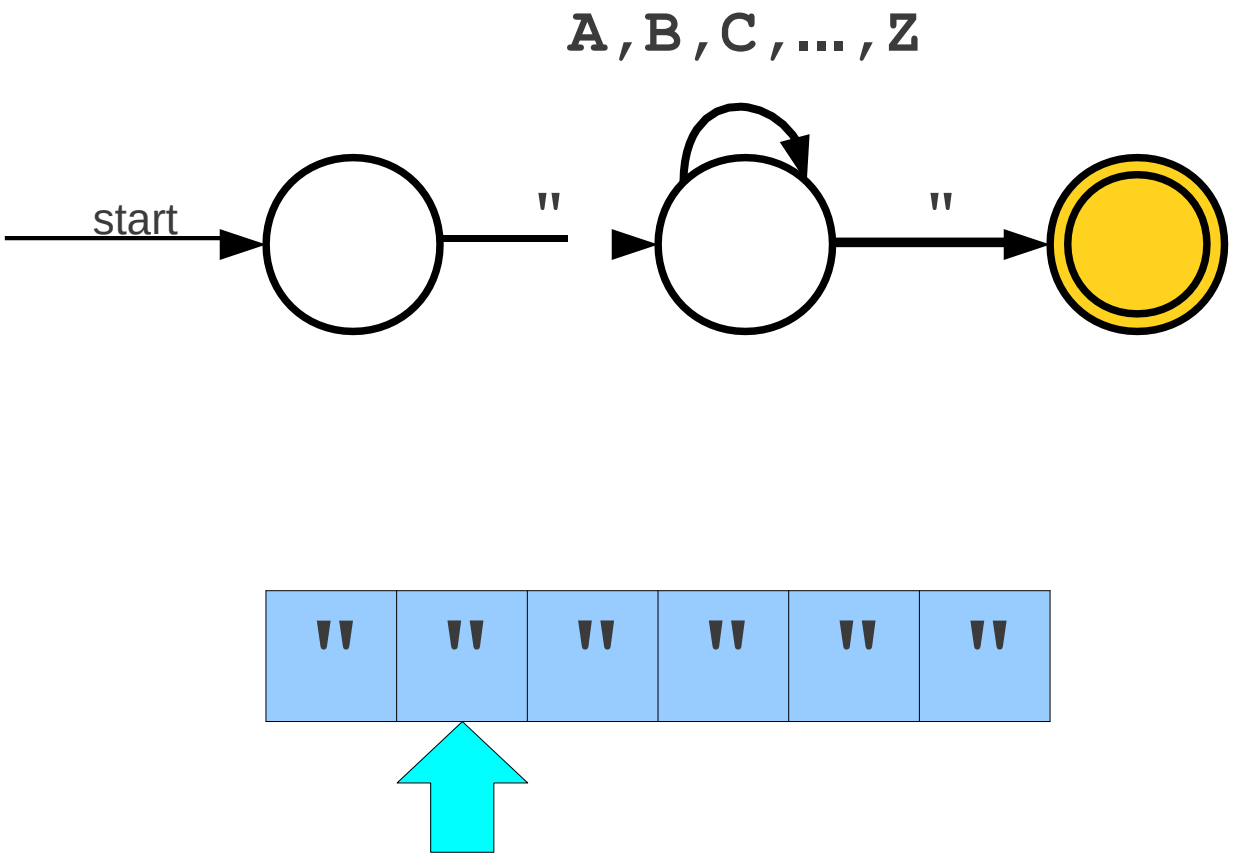
A Simple Automaton



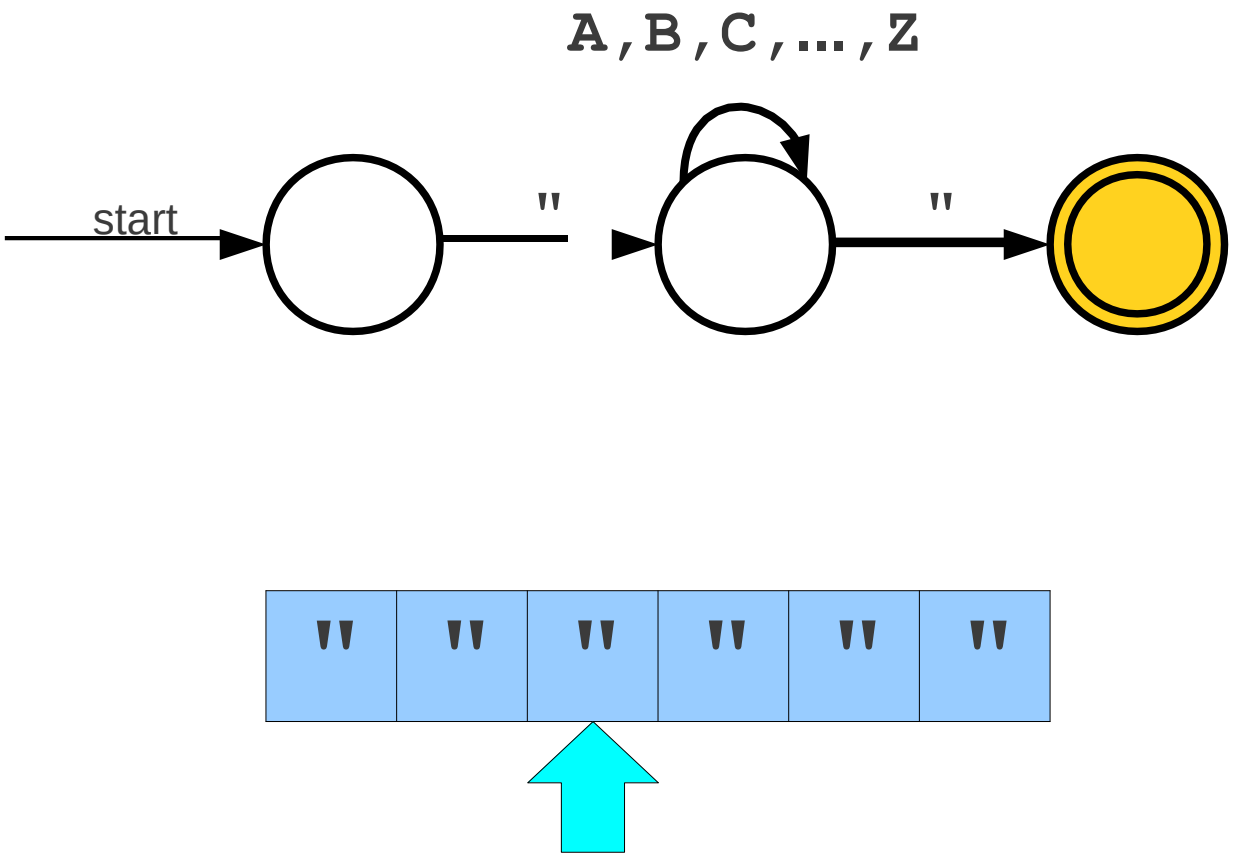
A Simple Automaton



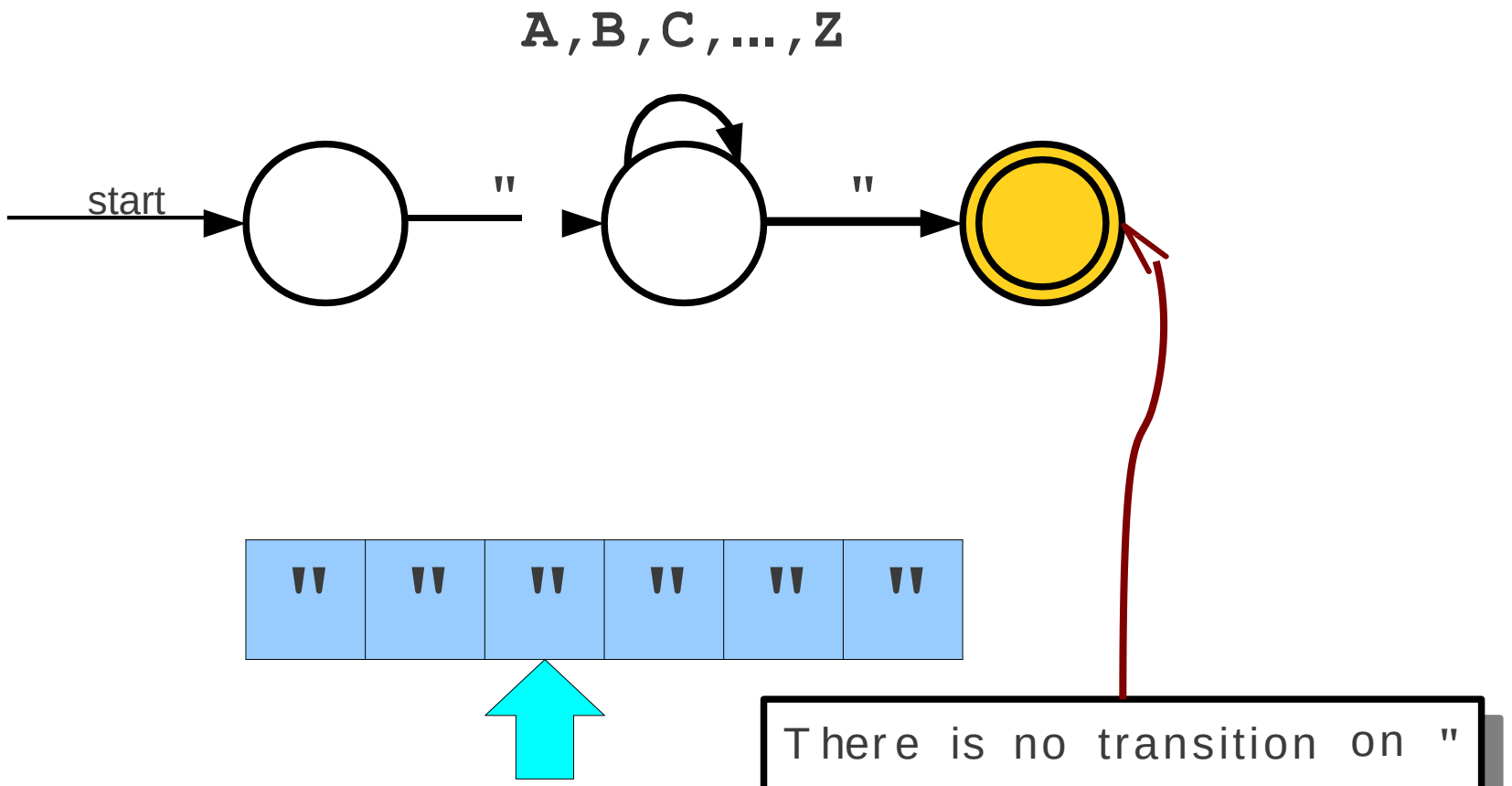
A Simple Automaton



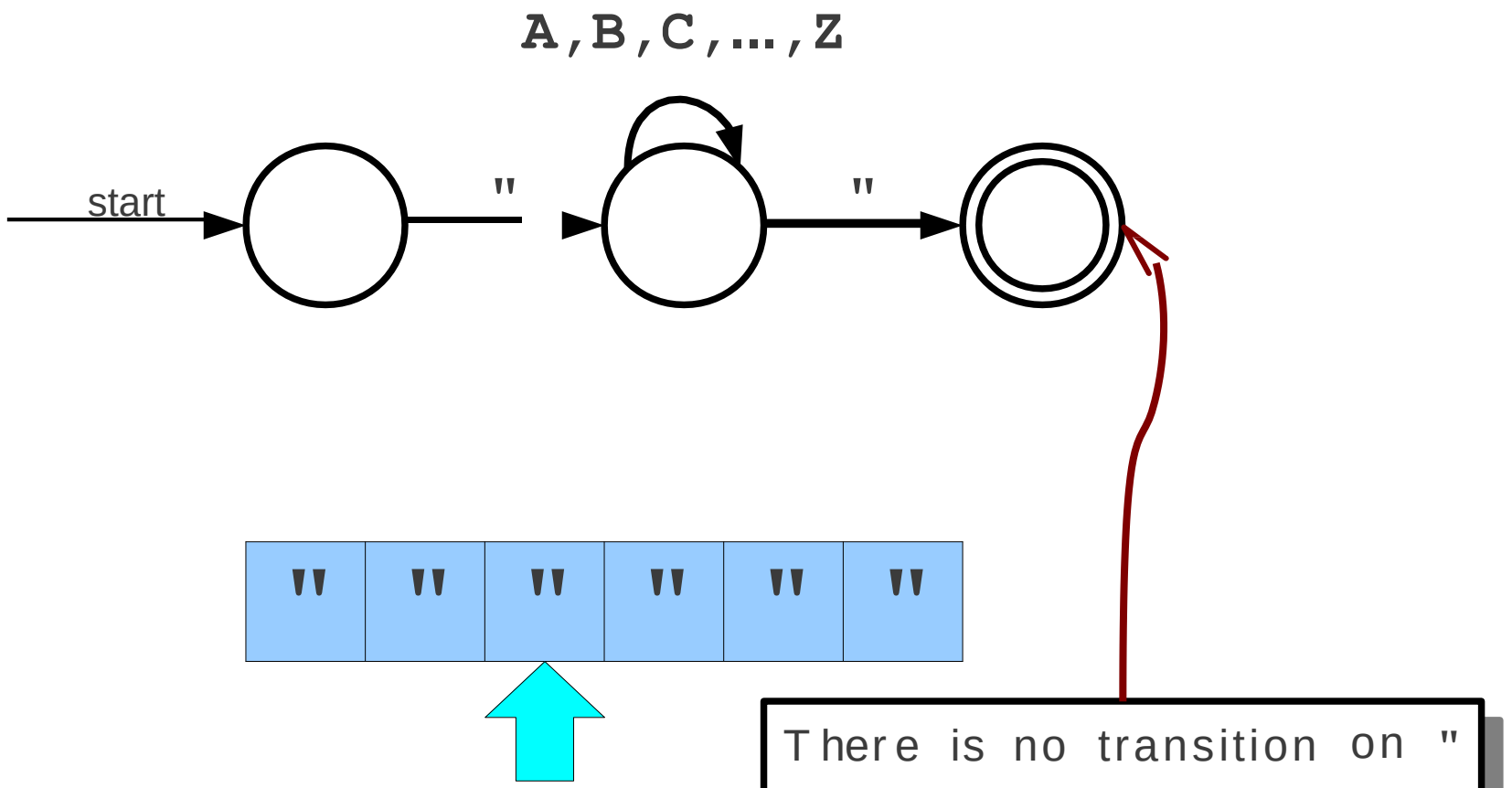
A Simple Automaton



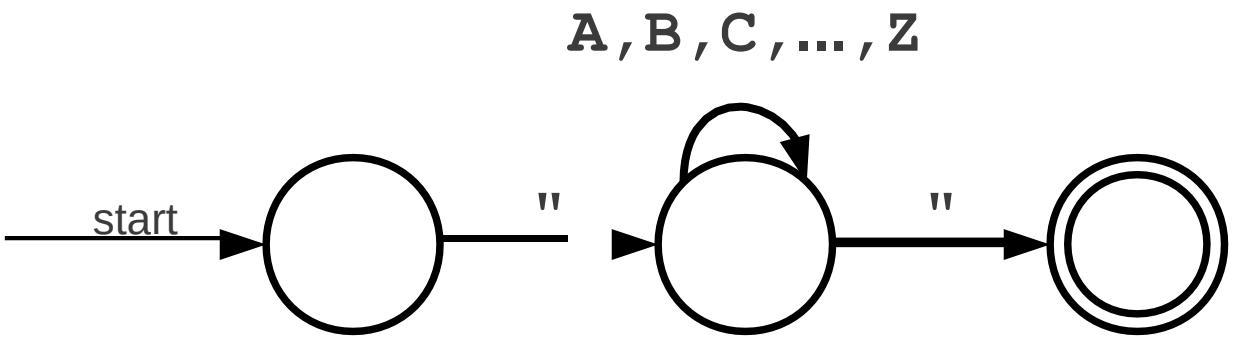
A Simple Automaton

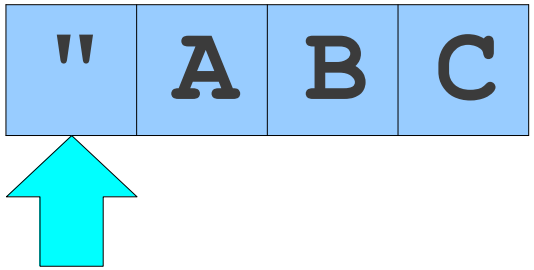


A Simple Automaton

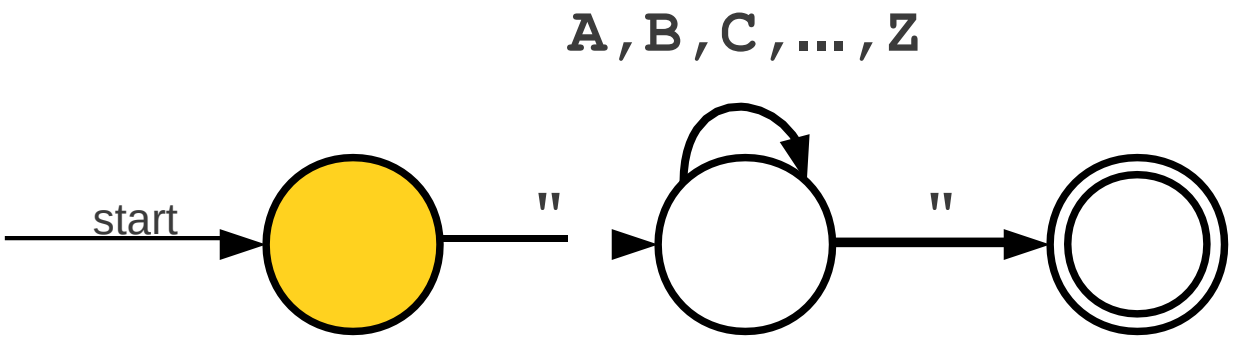


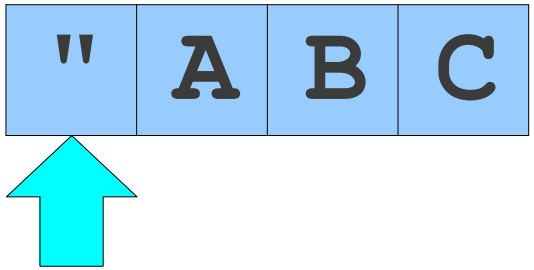
A Simple Automaton



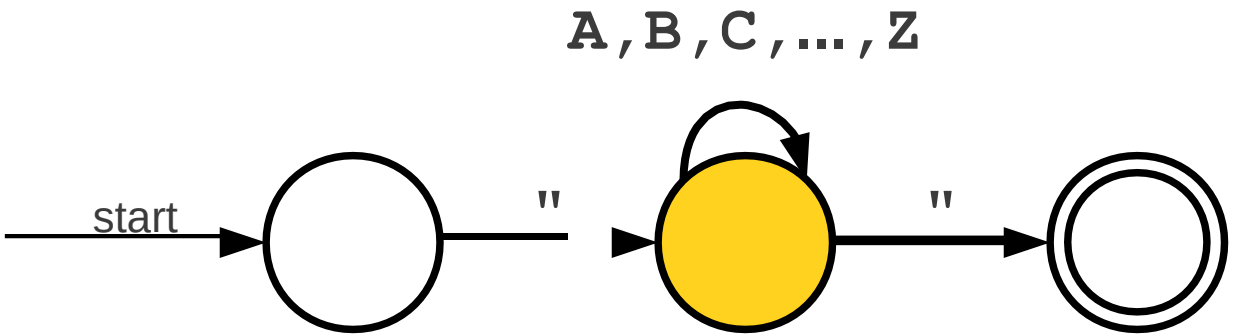


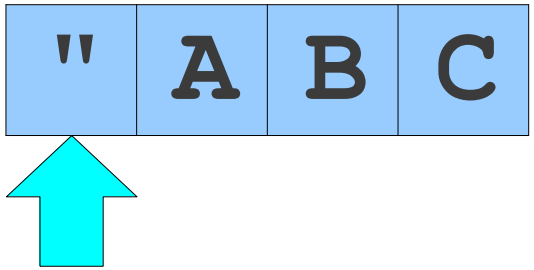
A Simple Automaton



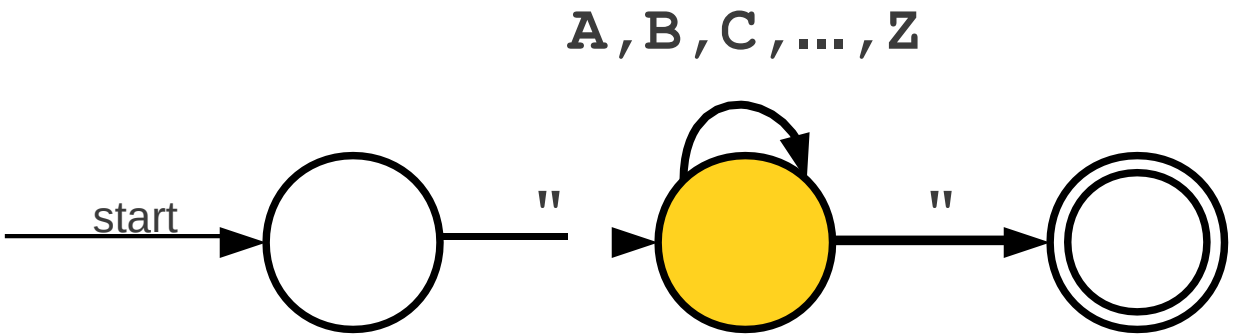


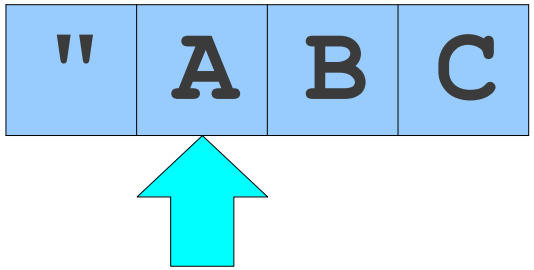
A Simple Automaton



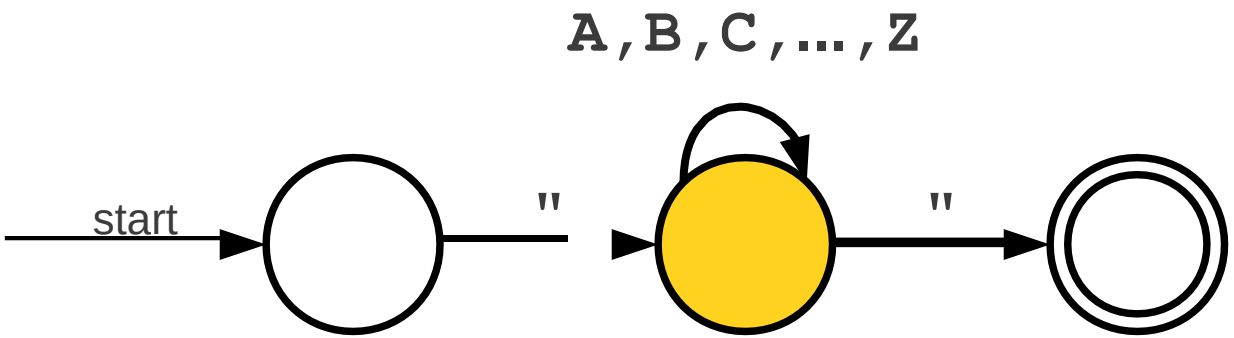


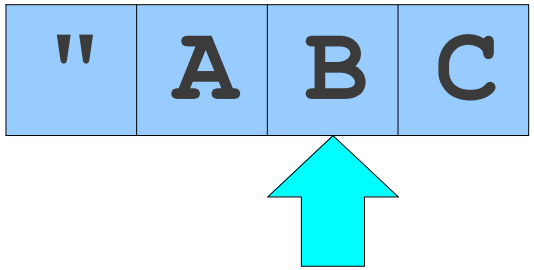
A Simple Automaton



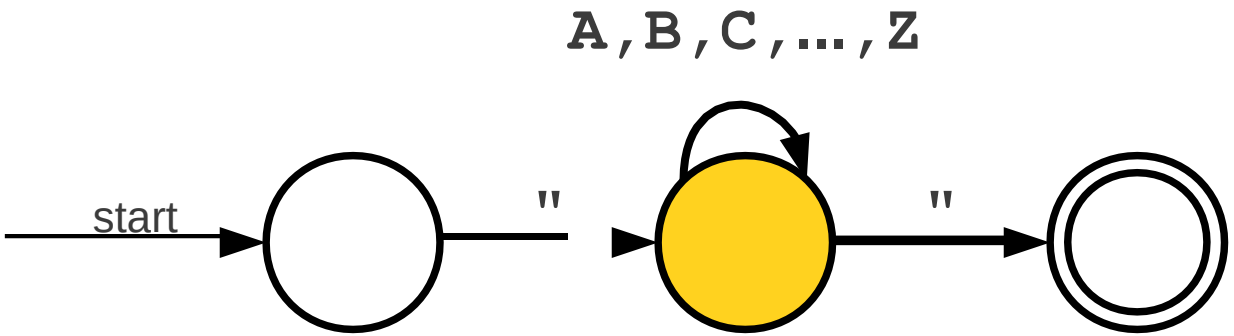


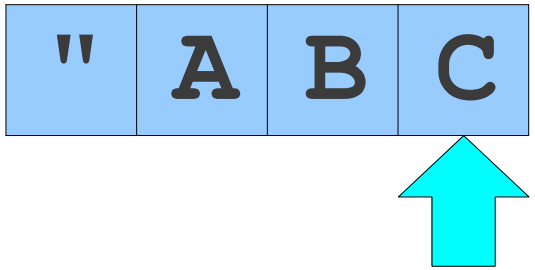
A Simple Automaton



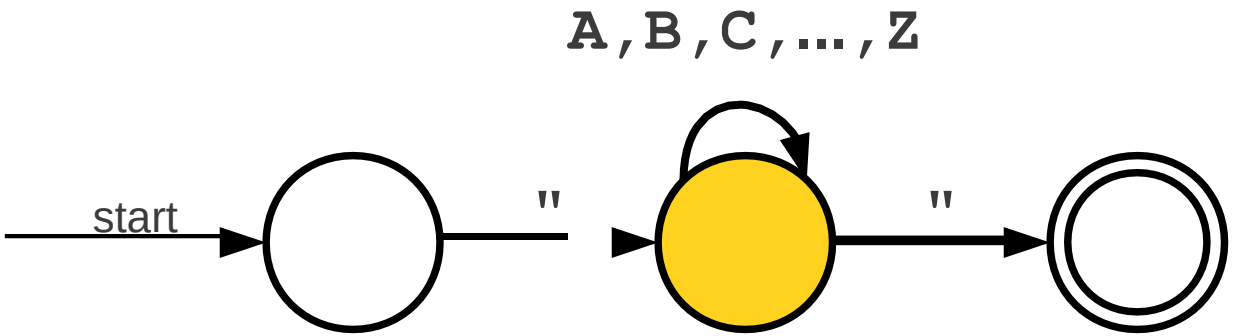


A Simple Automaton



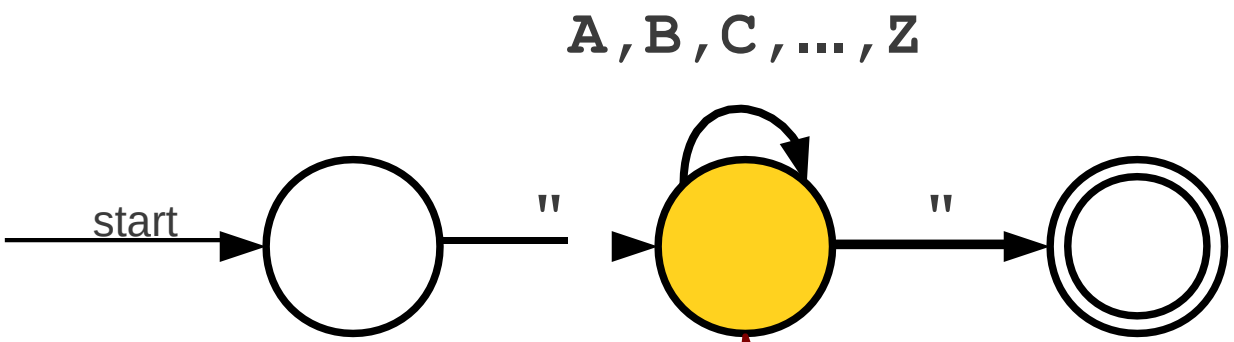


A Simple Automaton



"	A	B	C

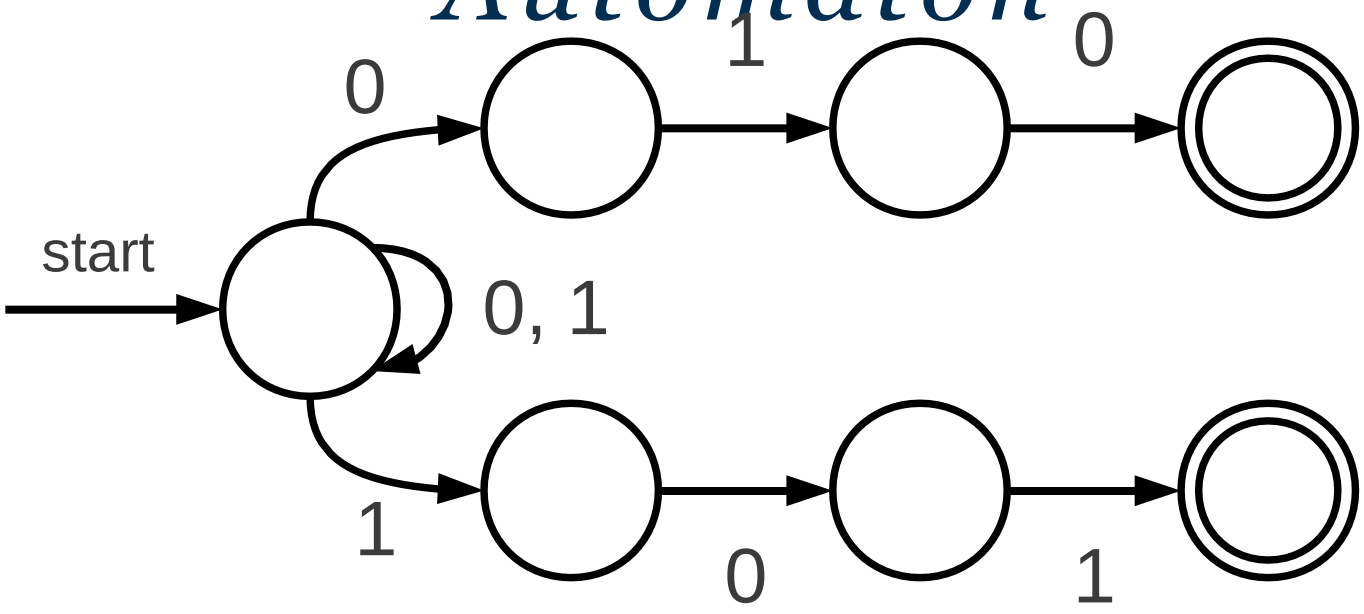
A Simple Automaton



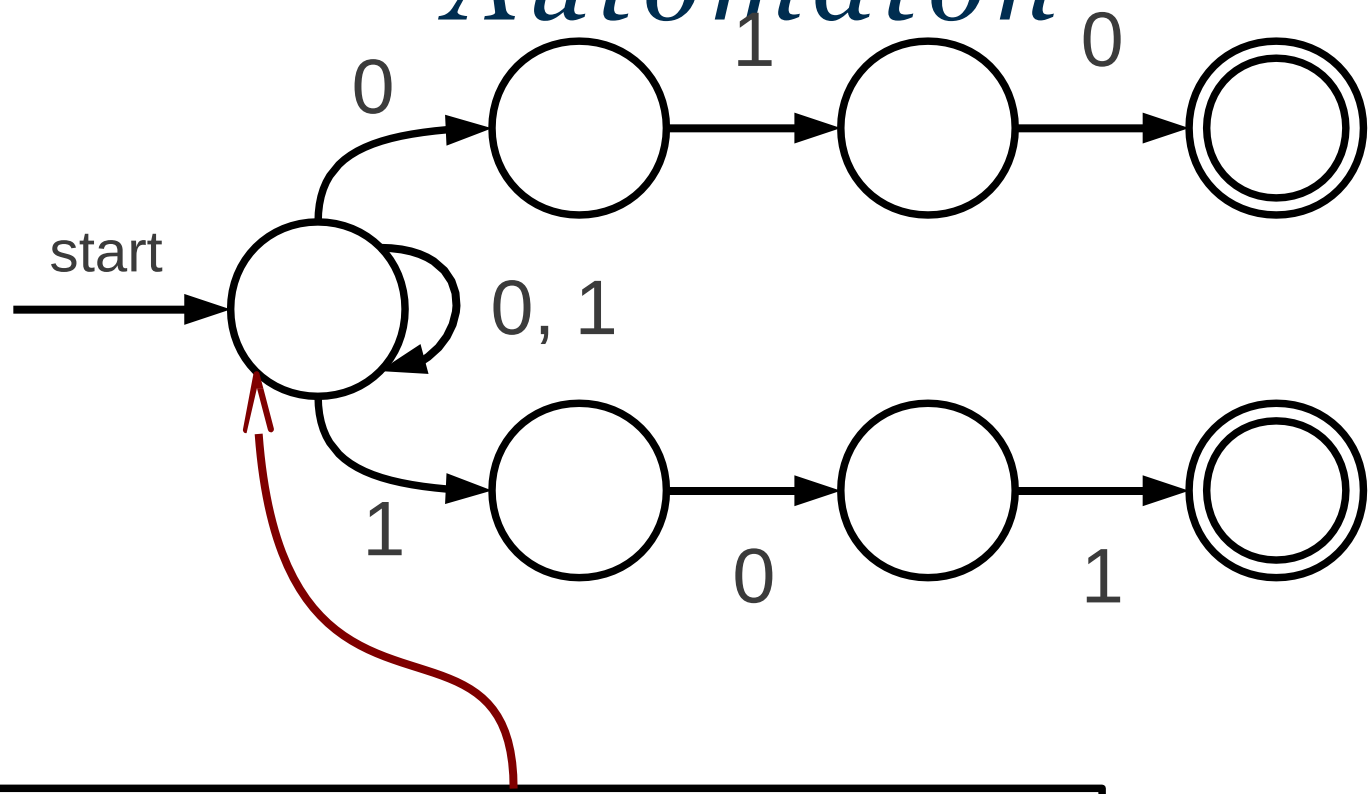
"	A	B	C

This is not an accepting state, so the automaton rejects.

A More Complex Automaton

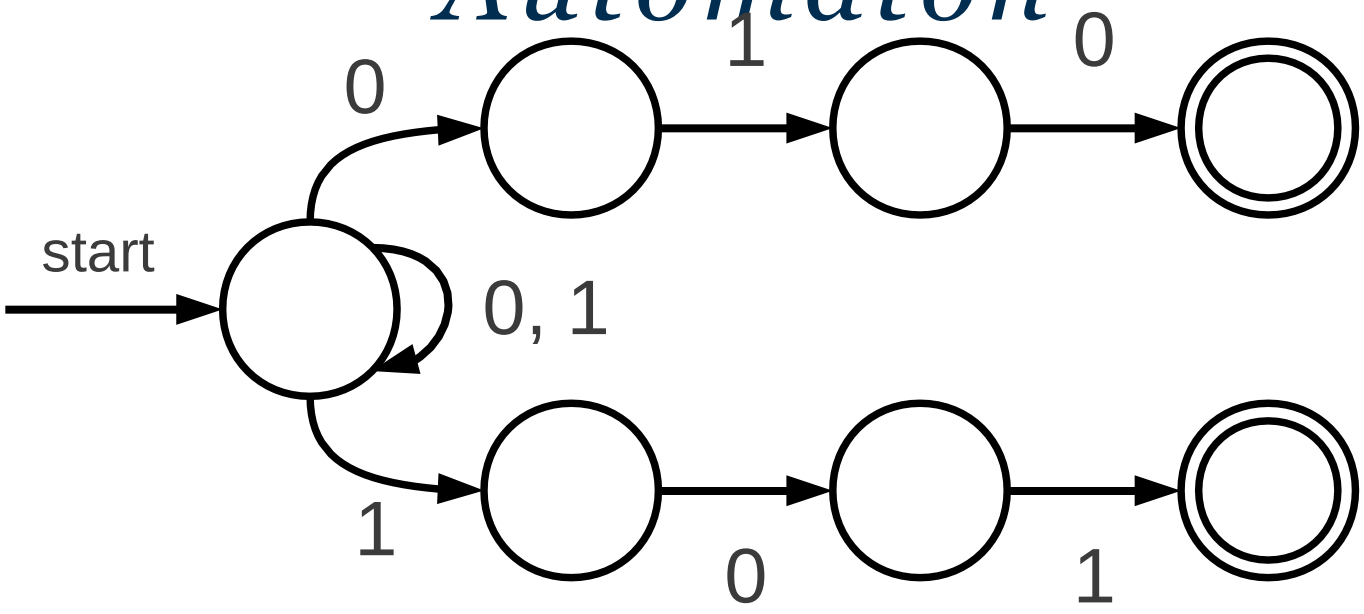


A More Complex Automaton

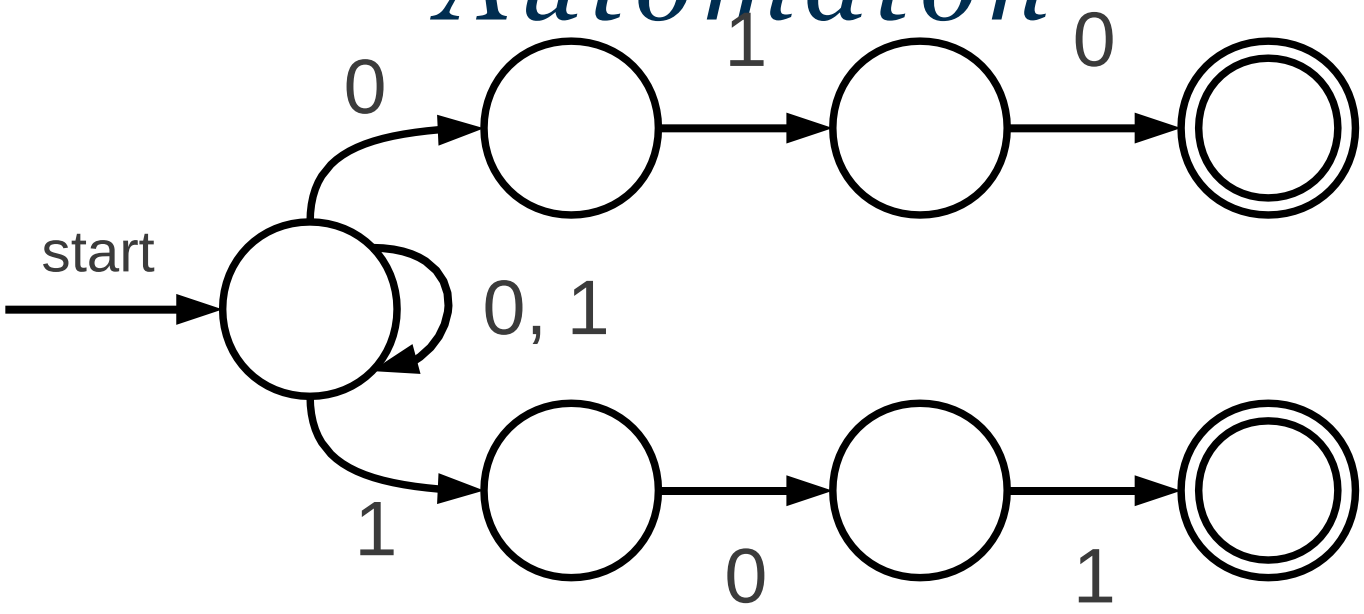


Notice that there are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow *both* transitions and enter multiple states.

A More Complex Automaton

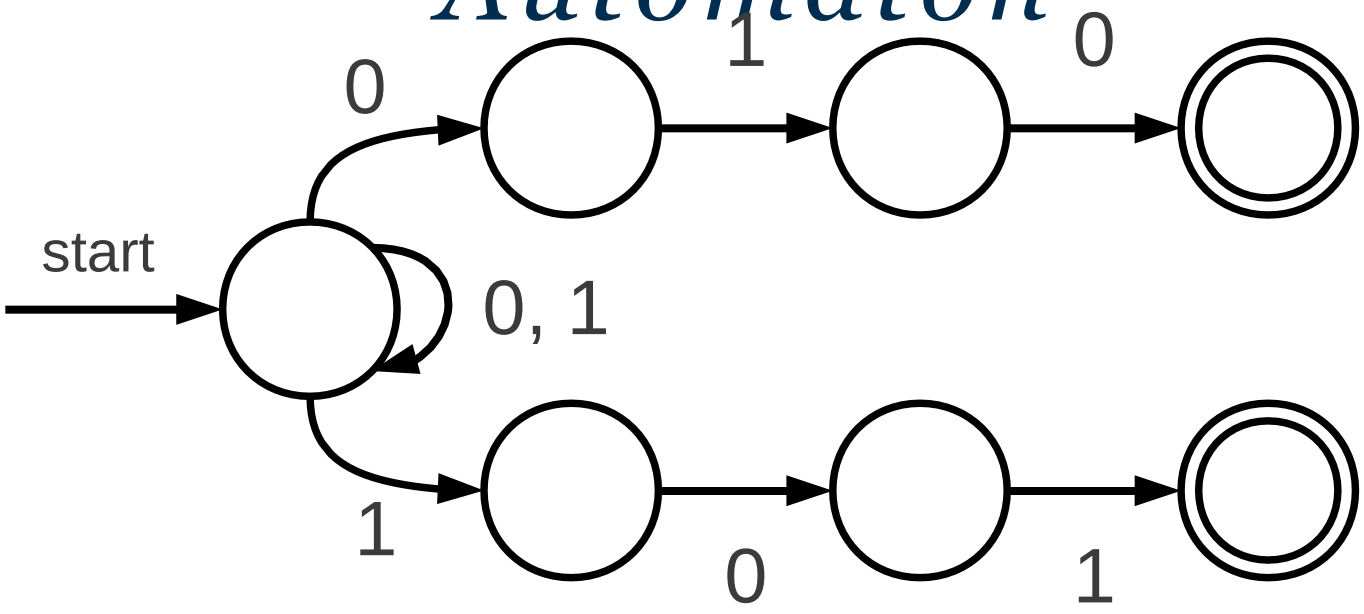


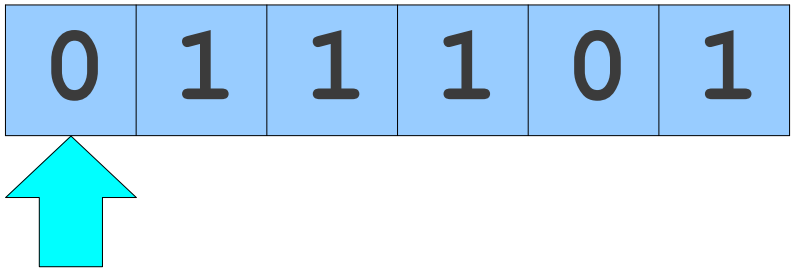
A More Complex Automaton



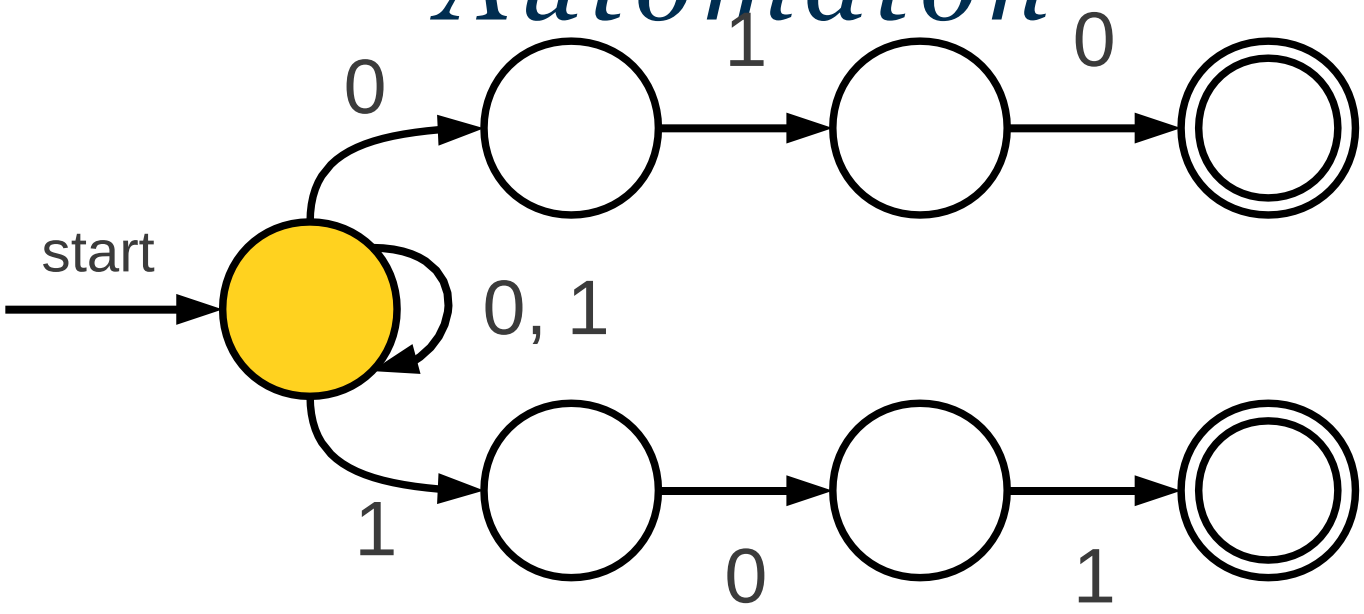
0	1	1	1	0	1

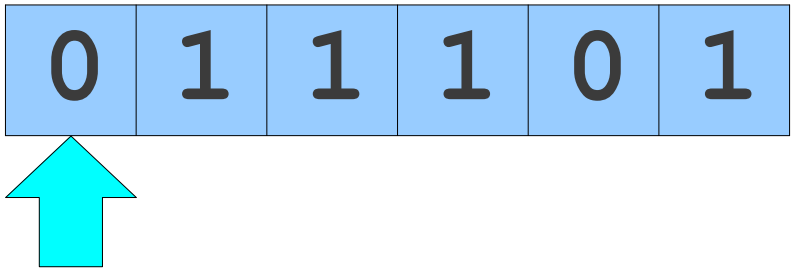
A More Complex Automaton



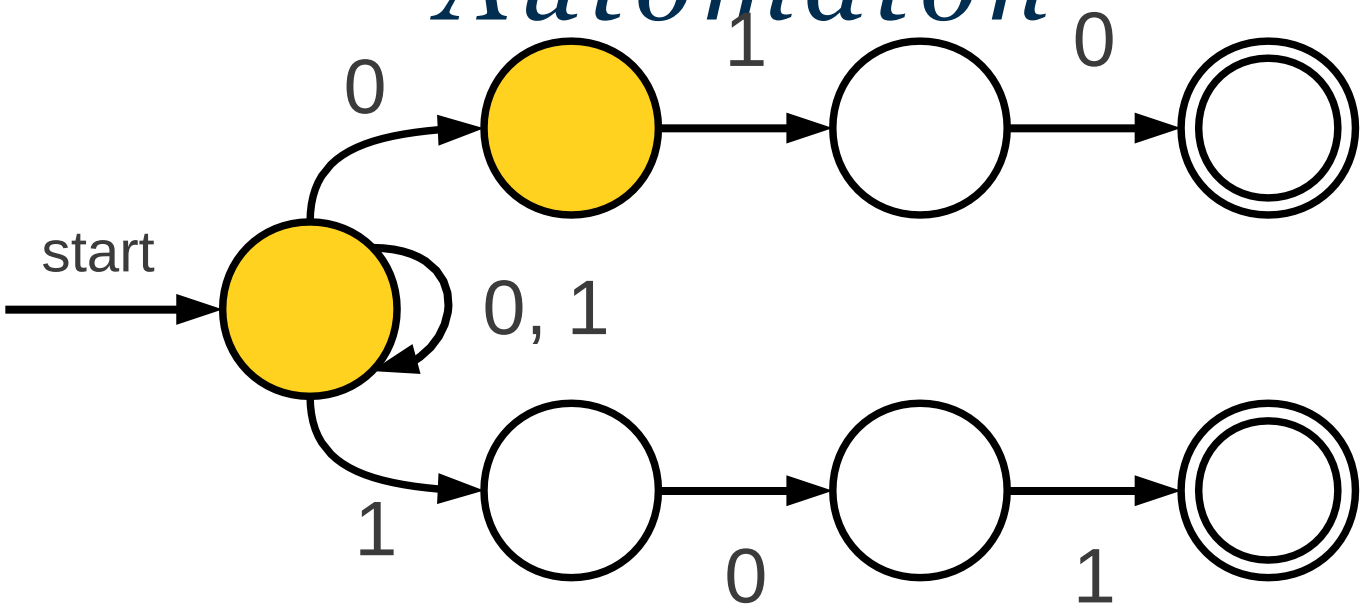


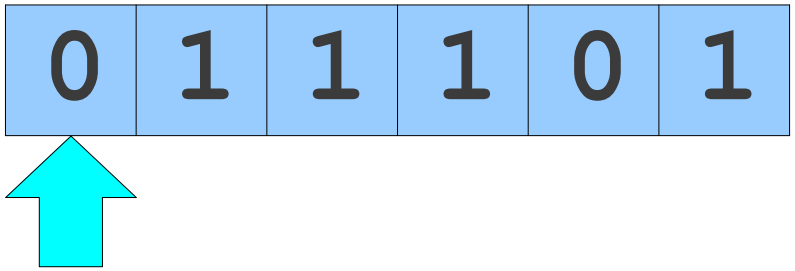
A More Complex Automaton



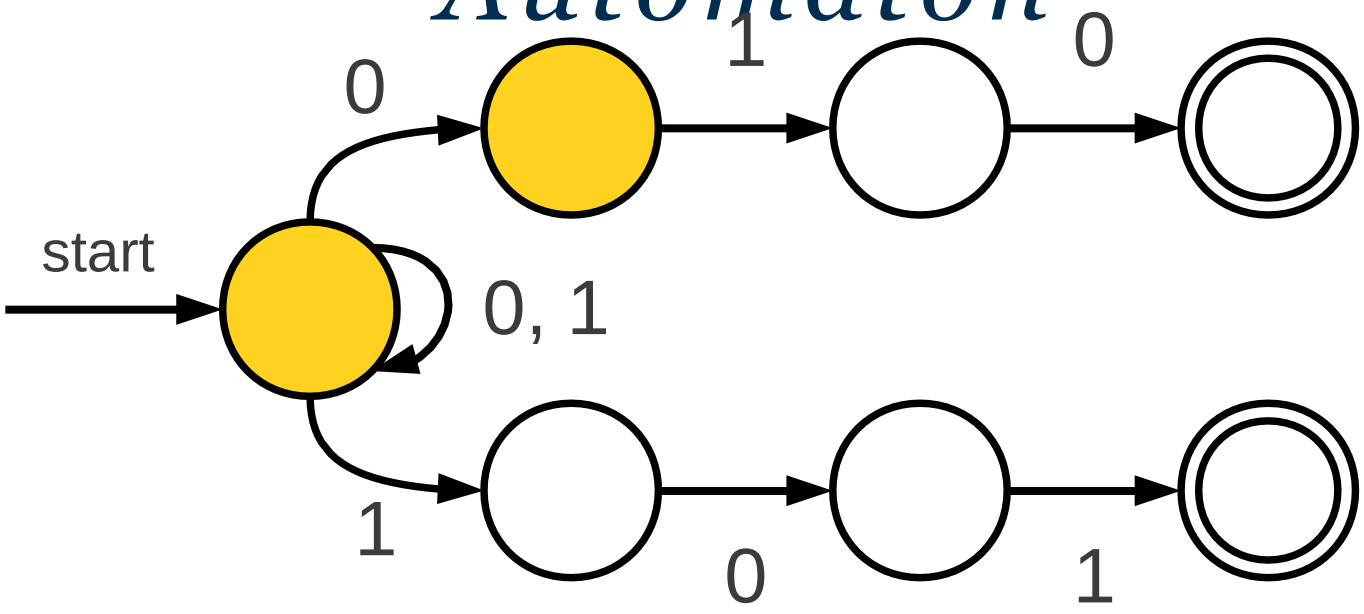


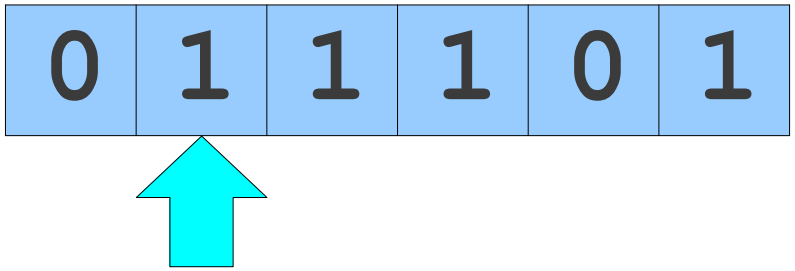
A More Complex Automaton



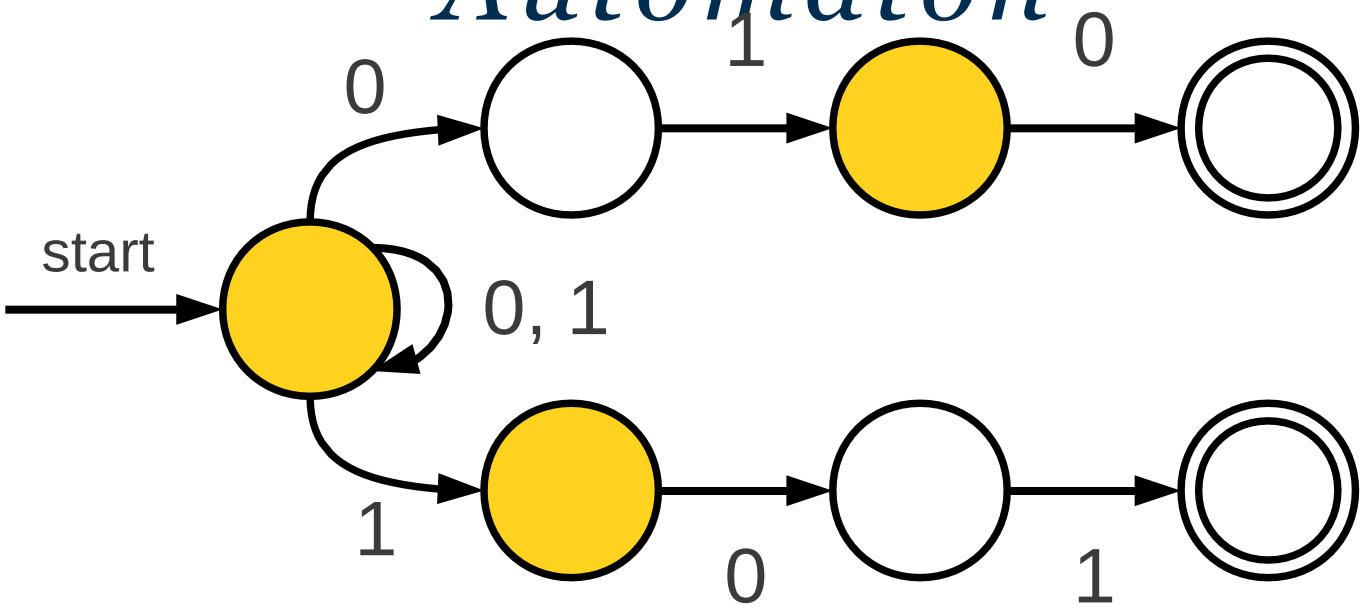


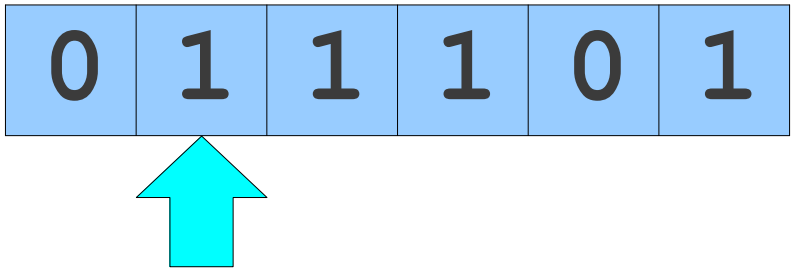
A More Complex Automaton



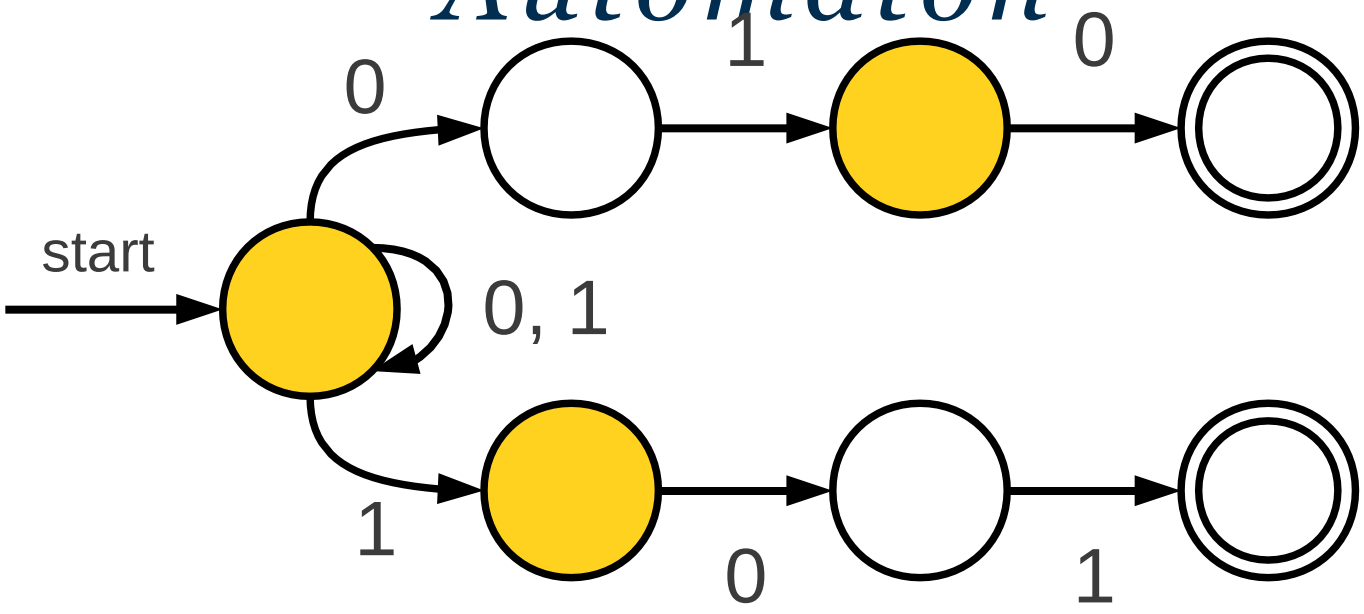


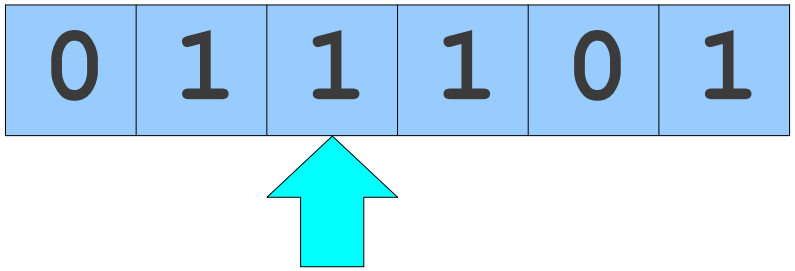
A More Complex Automaton



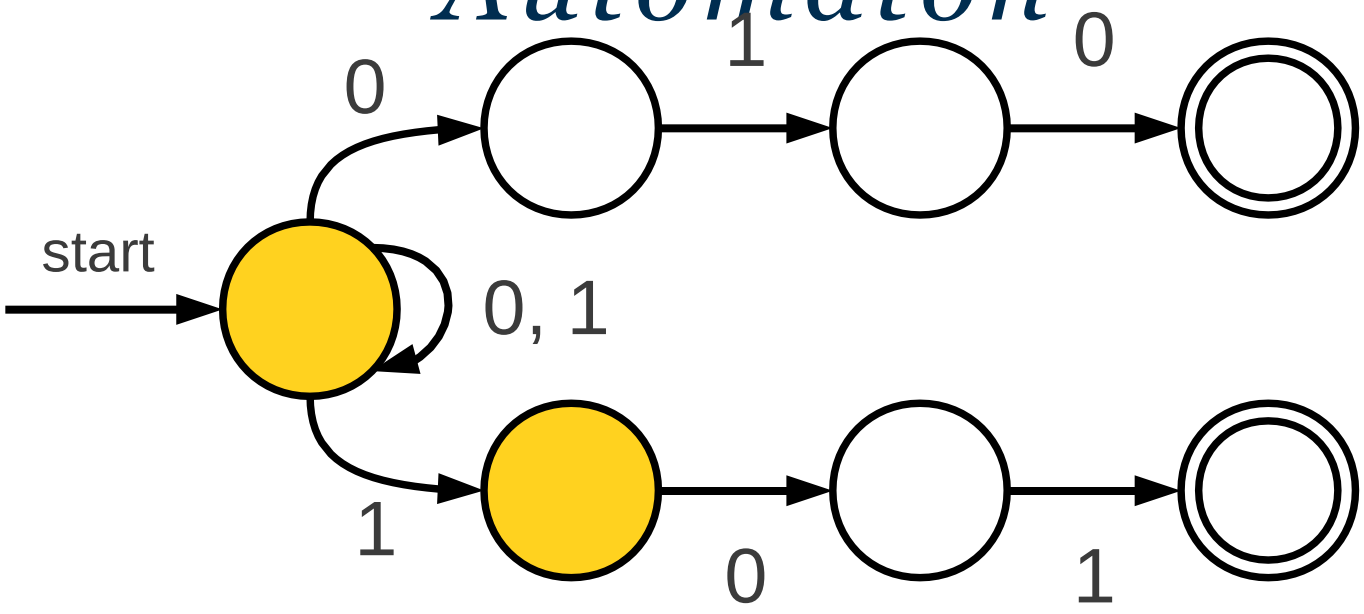


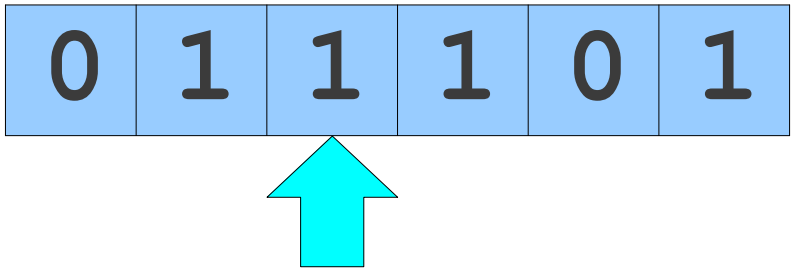
A More Complex Automaton



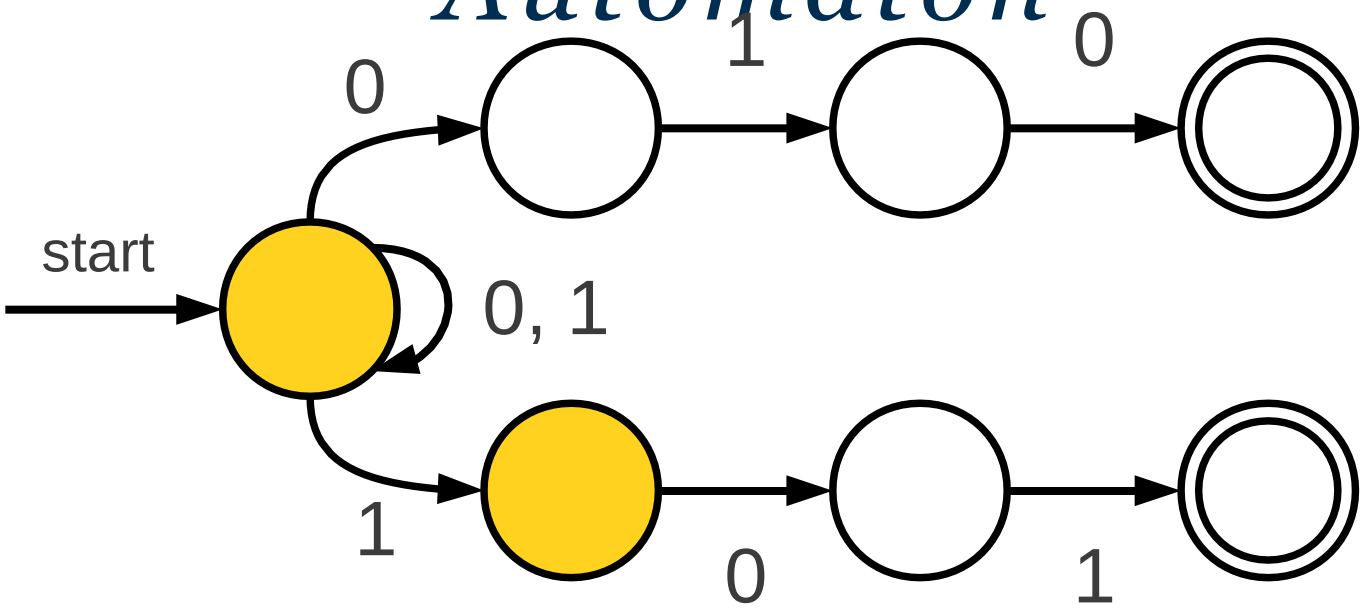


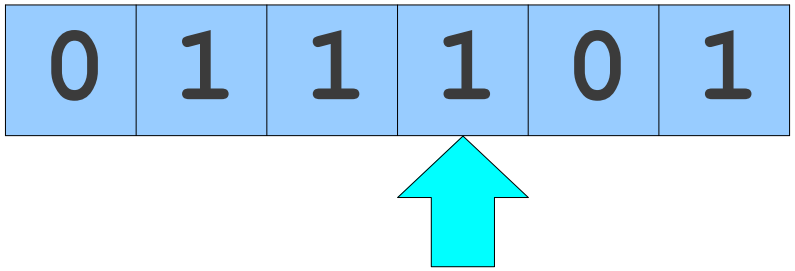
A More Complex Automaton



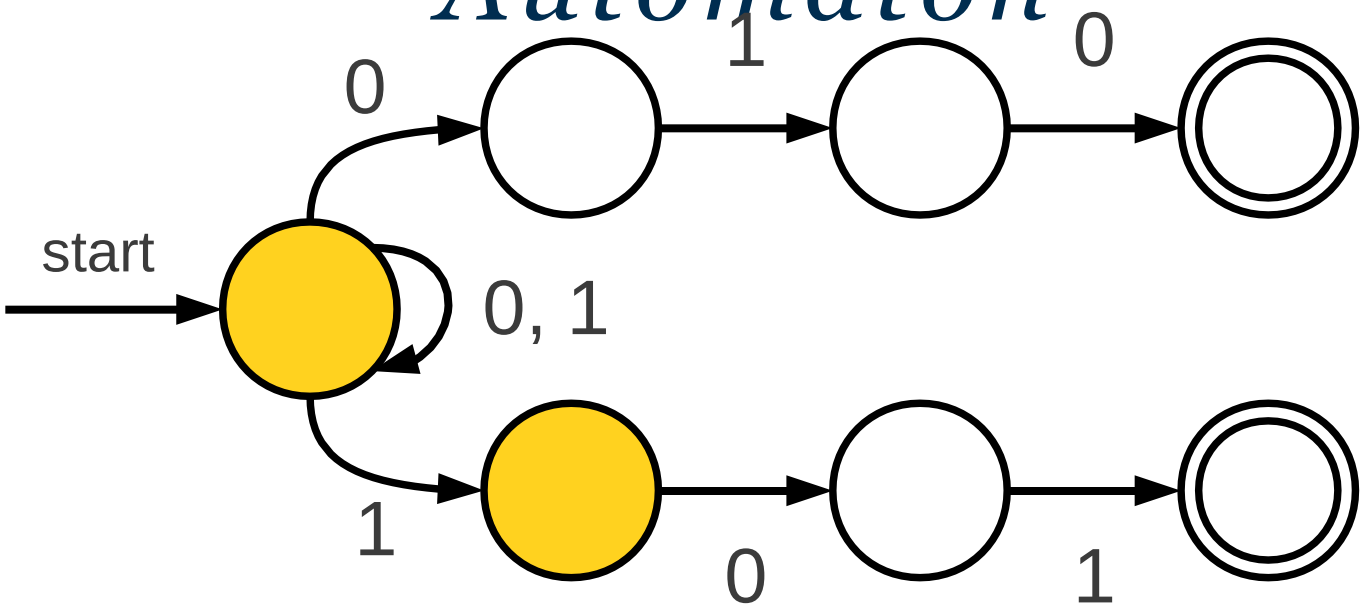


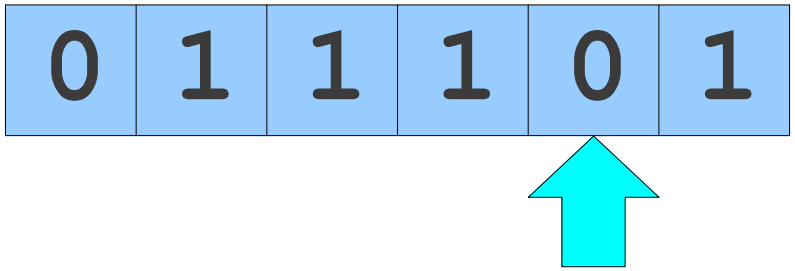
A More Complex Automaton



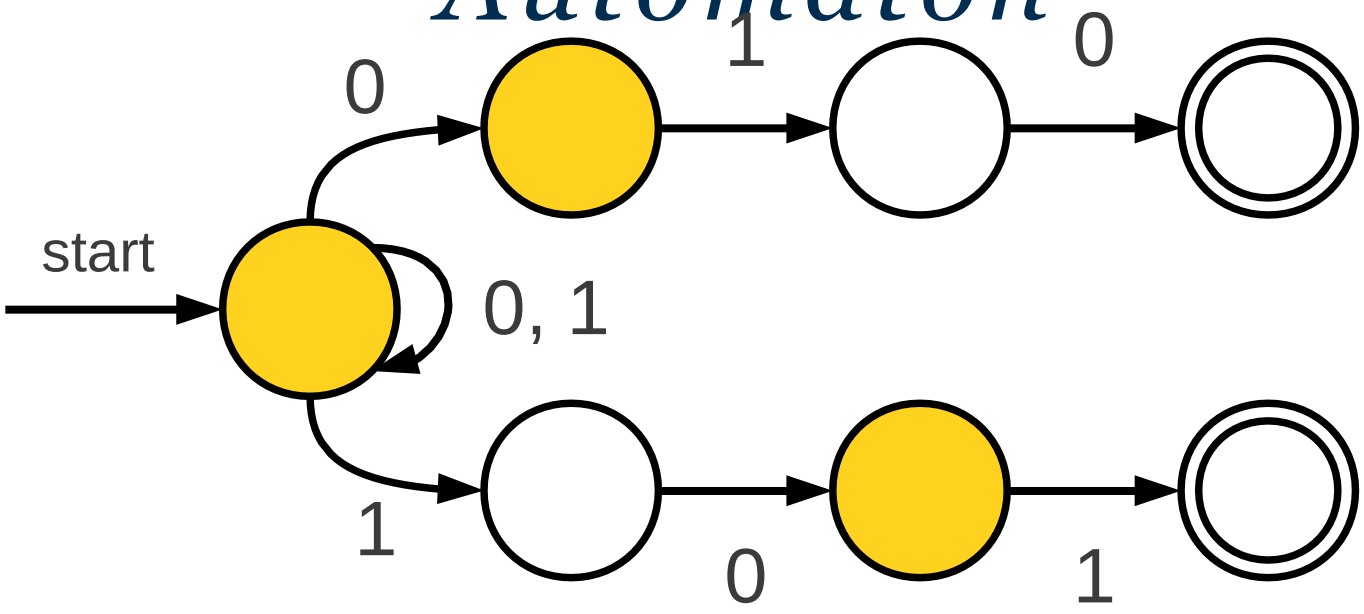


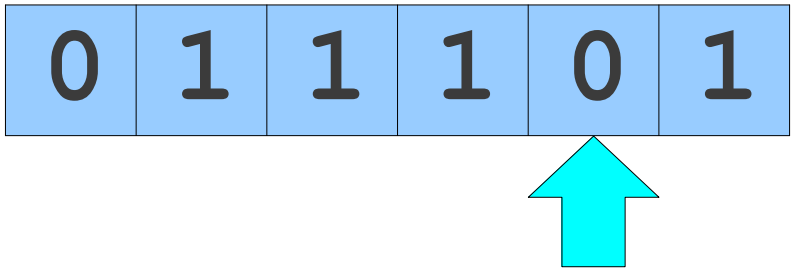
A More Complex Automaton



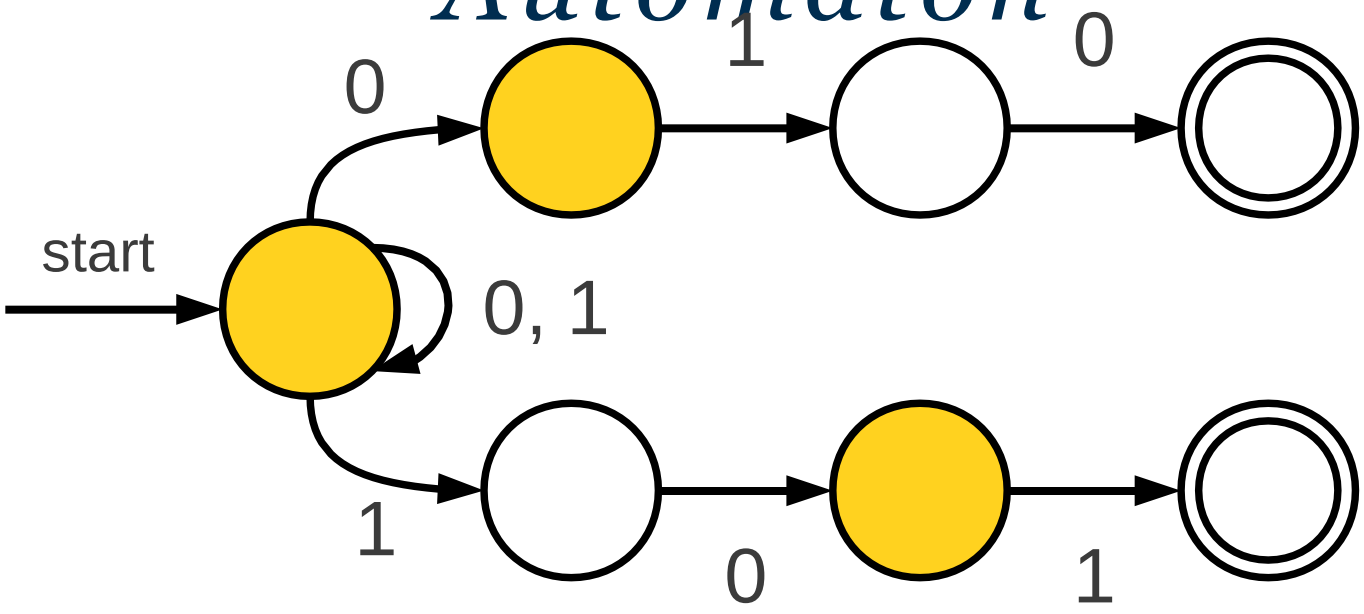


A More Complex Automaton



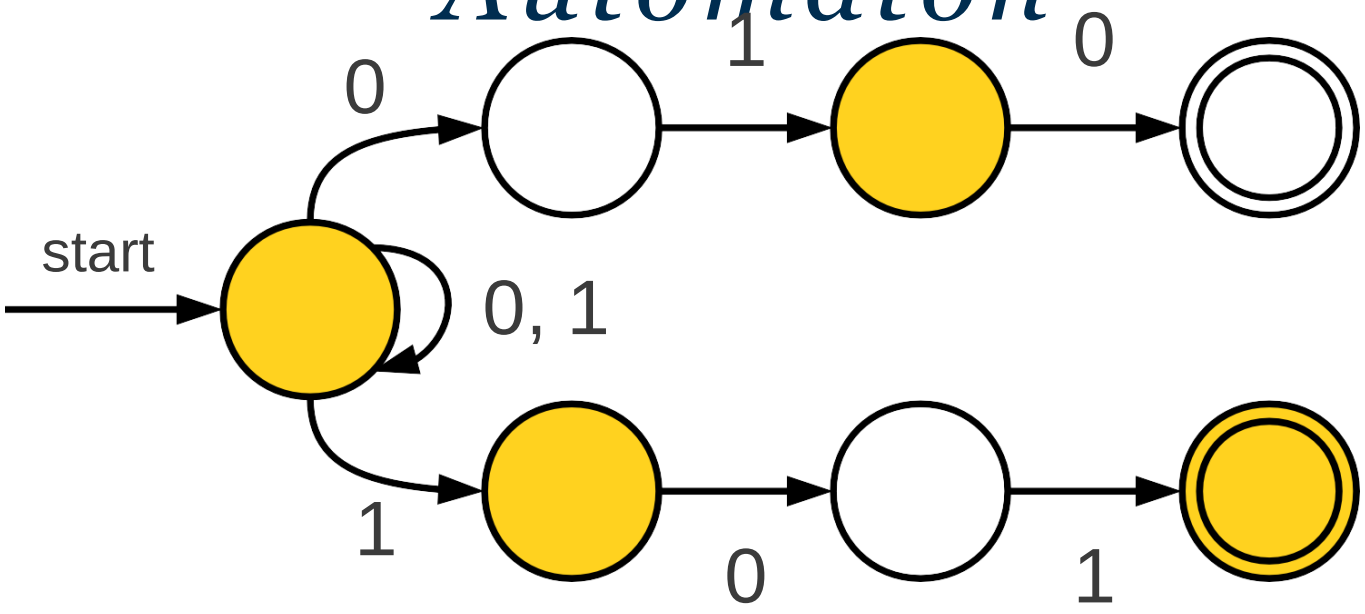


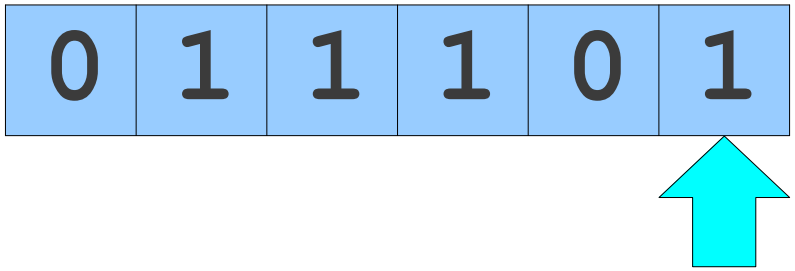
A More Complex Automaton



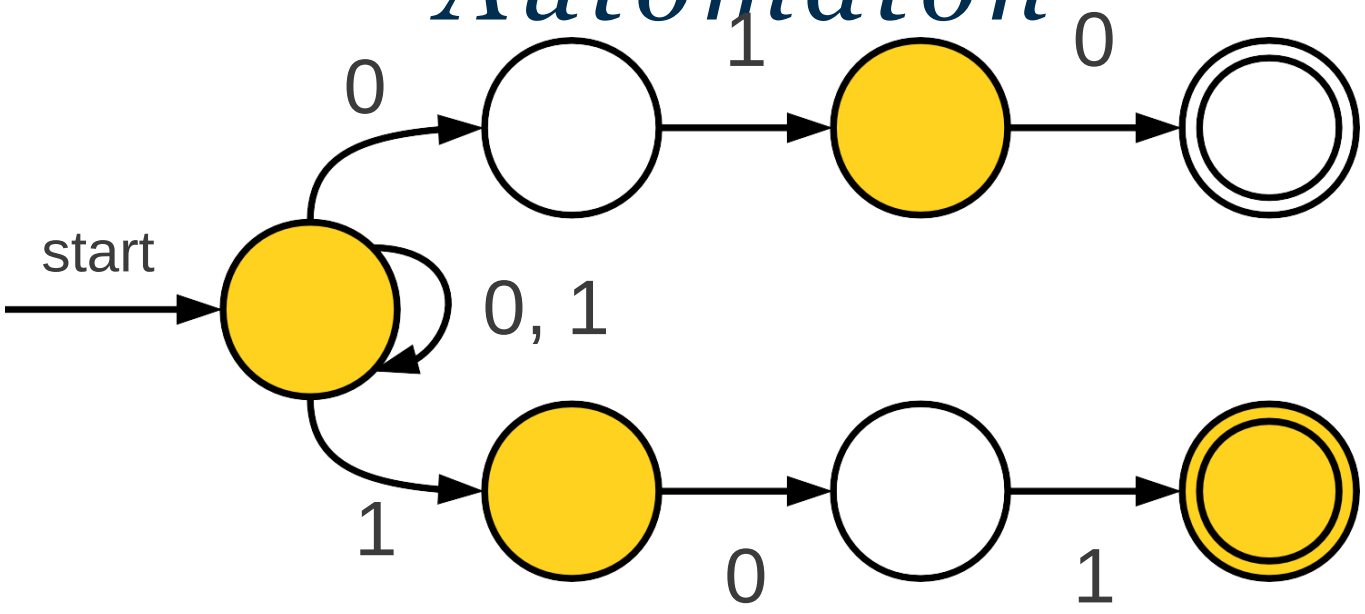
0	1	1	1	0	1

A More Complex Automaton



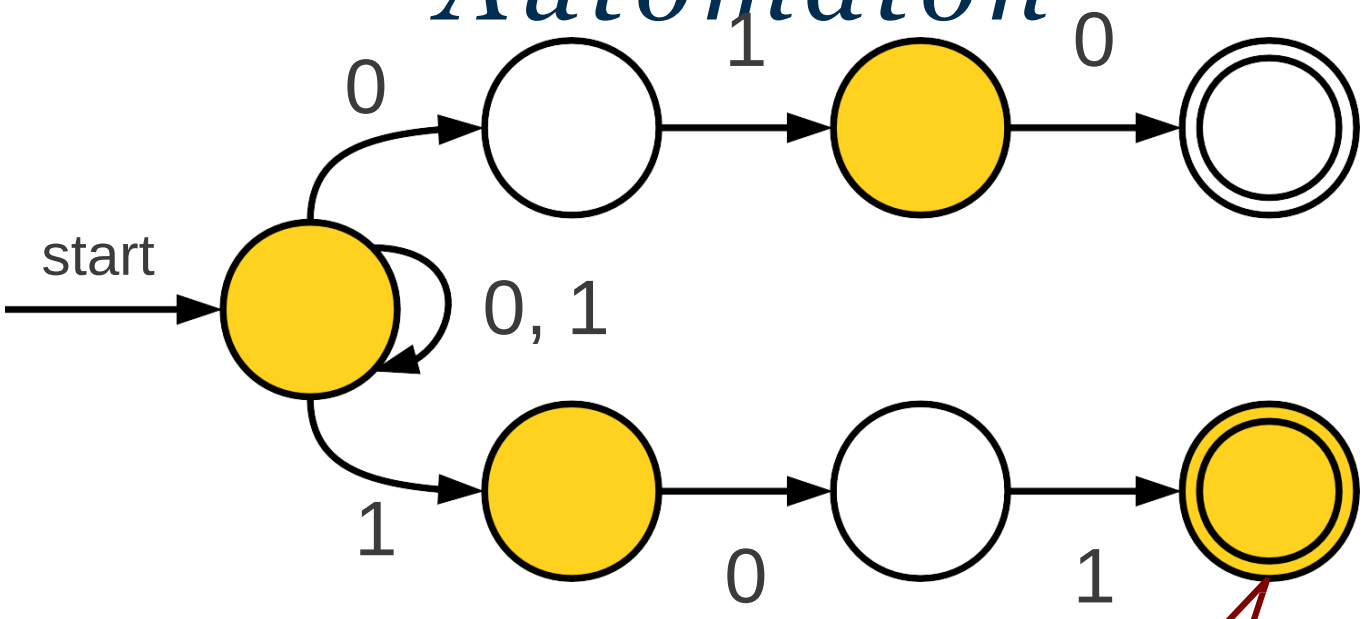


A More Complex Automaton



0	1	1	1	0	1

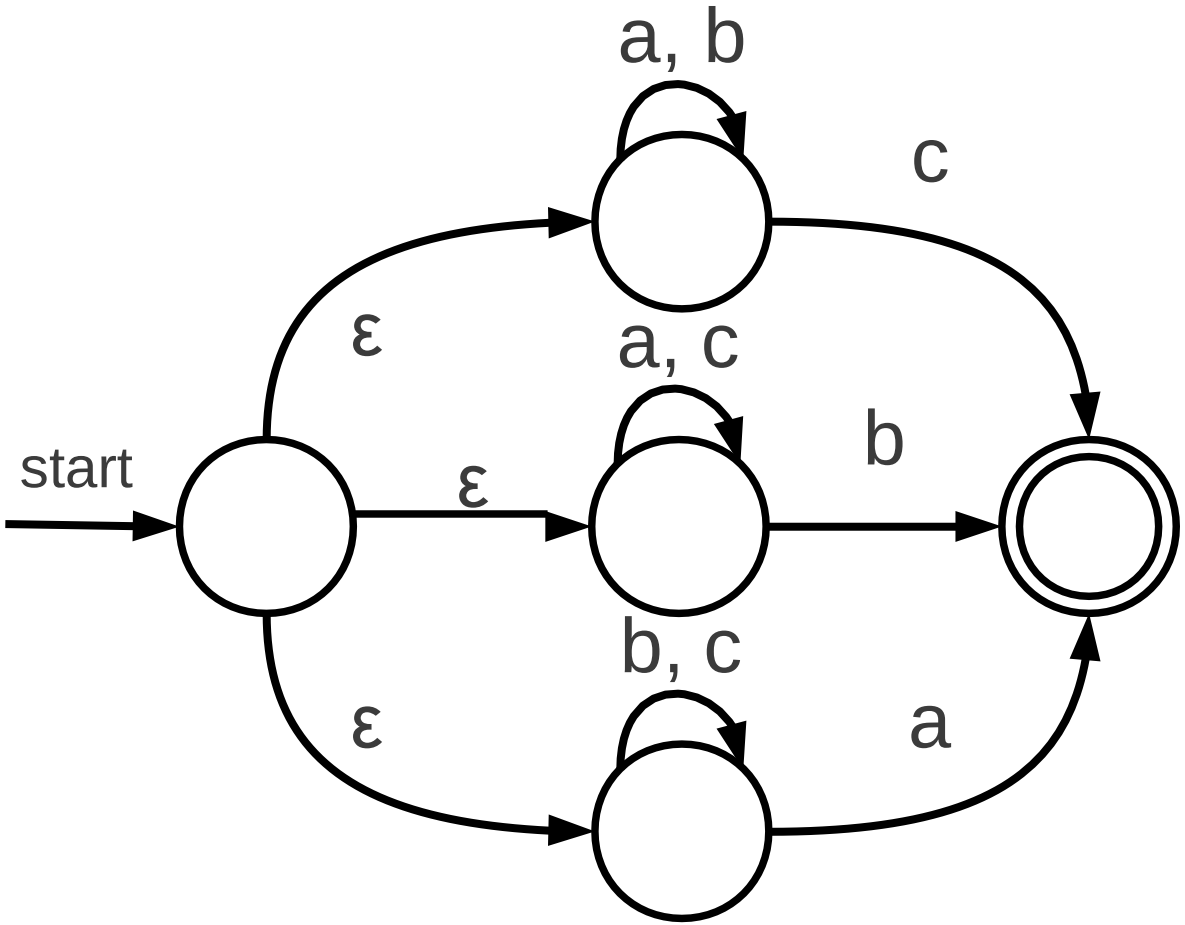
A More Complex Automaton



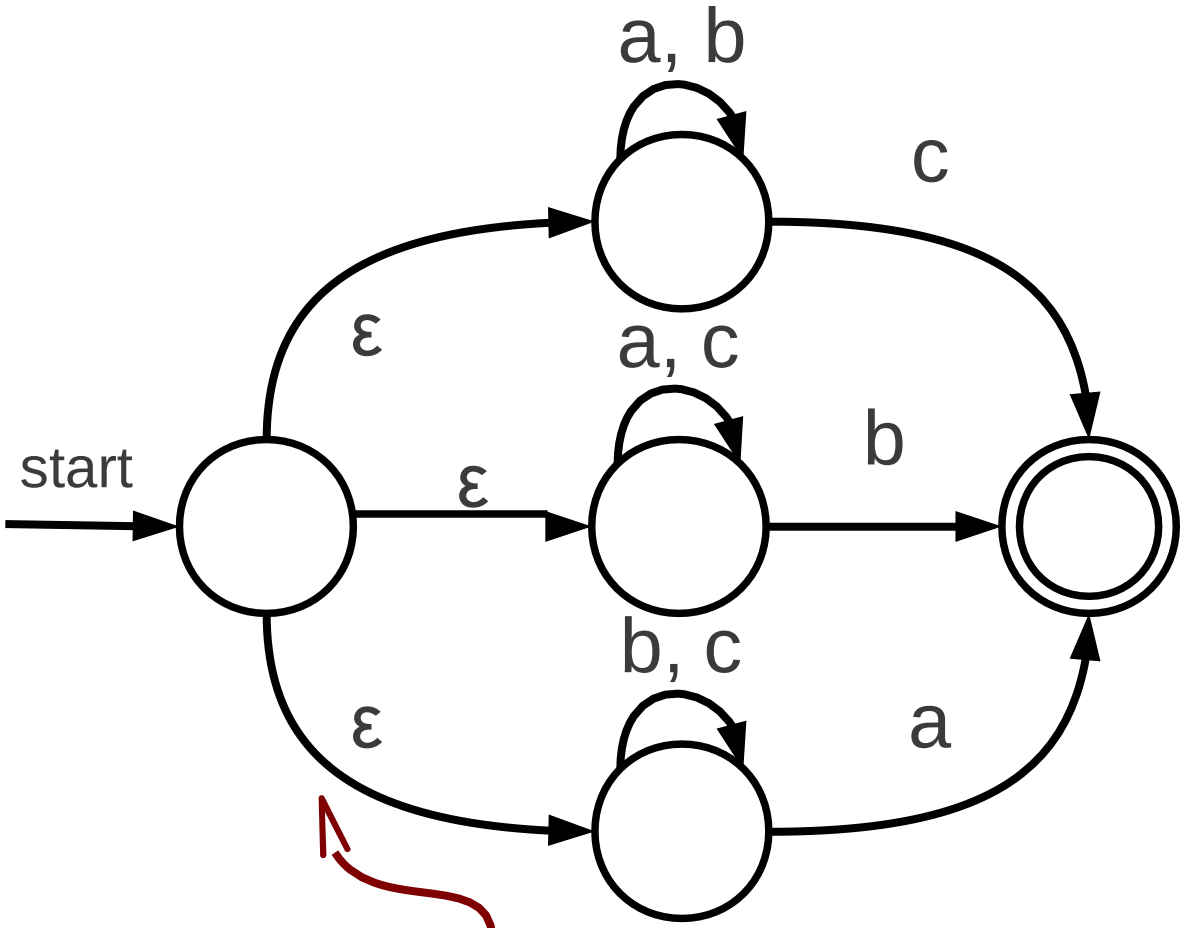
0	1	1	1	0	1

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

An Even More Complex Automaton



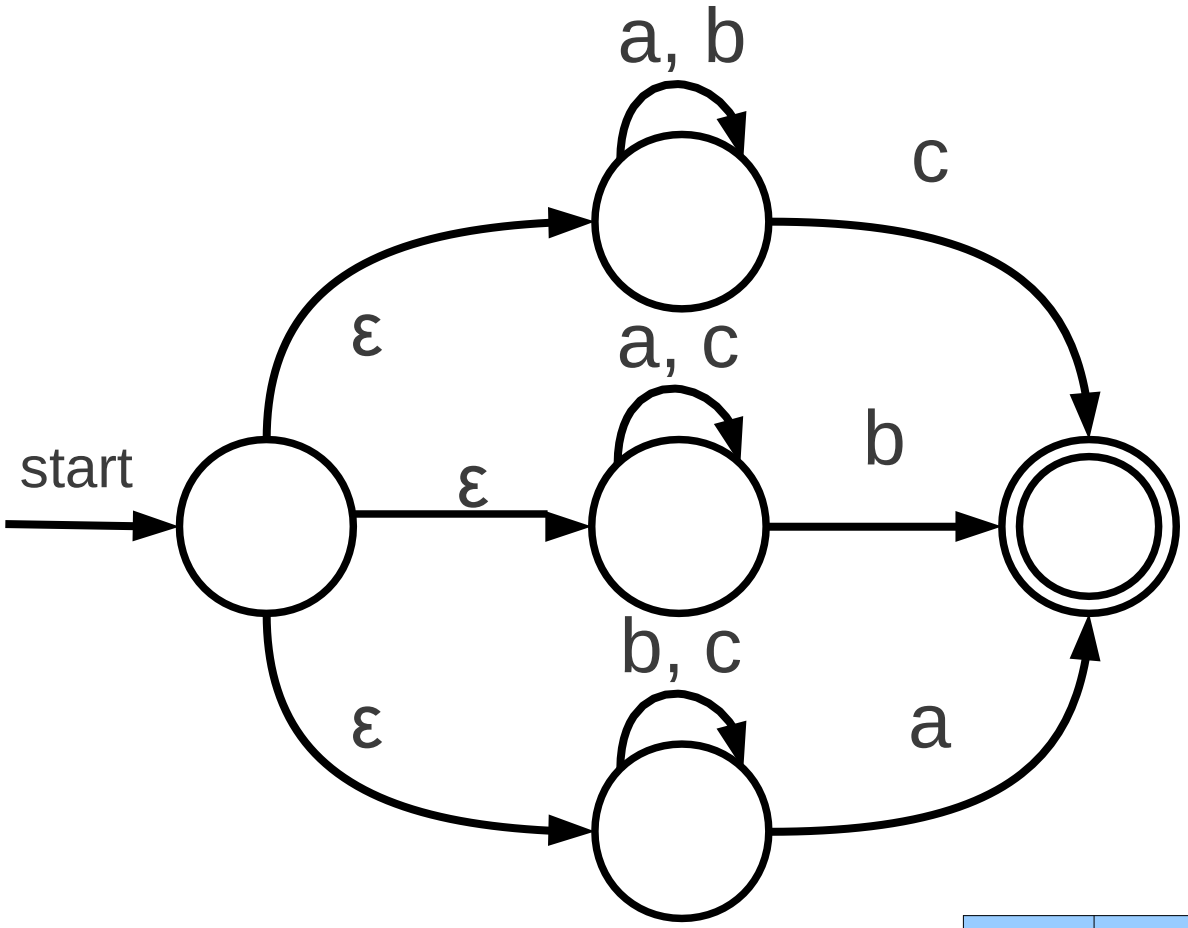
An Even More Complex Automaton

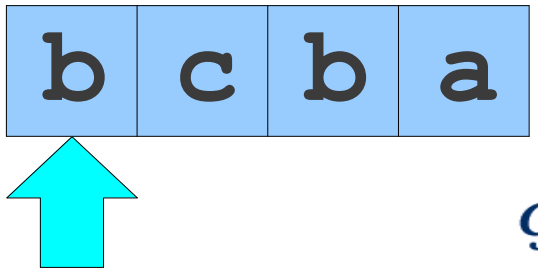


These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

GEORGETOWN UNIVERSITY

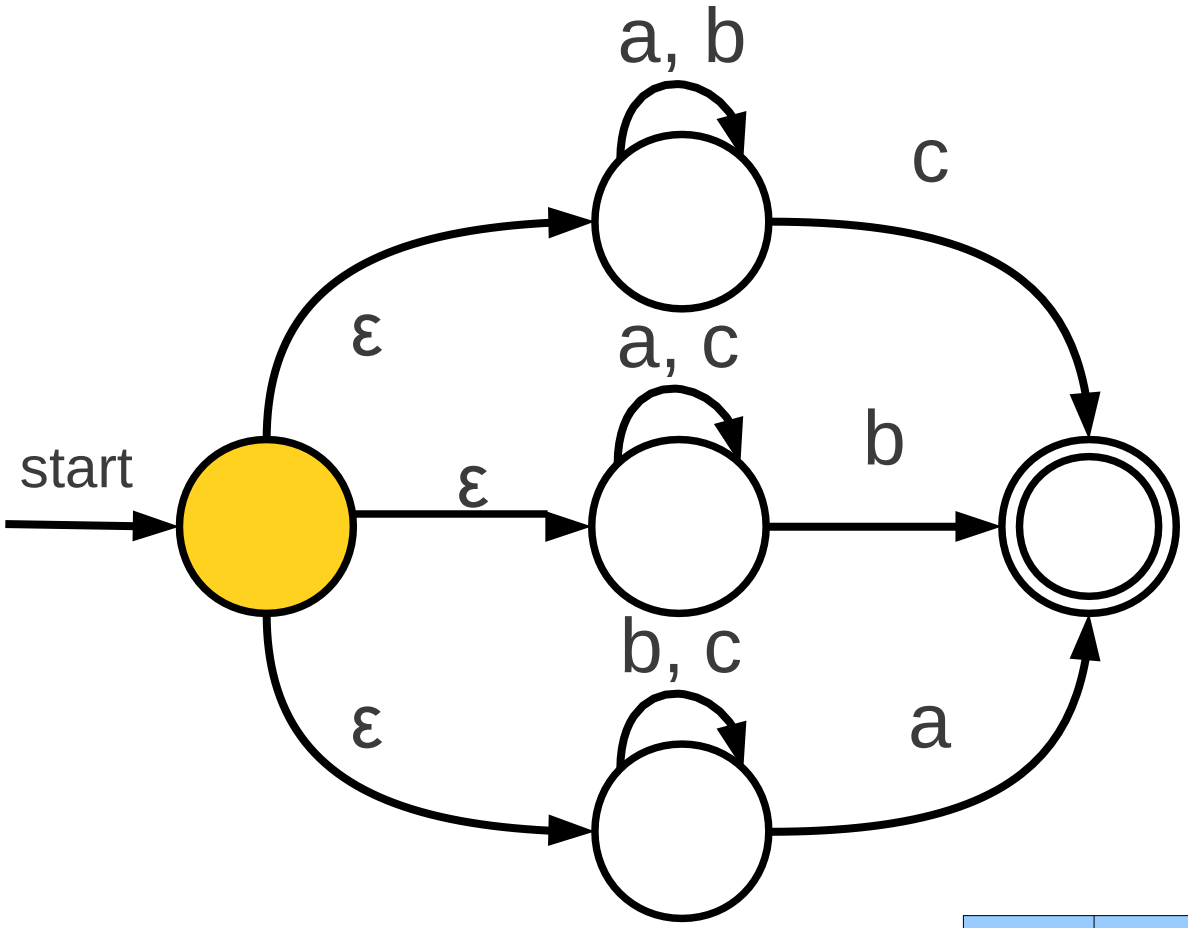
An Even More Complex Automaton

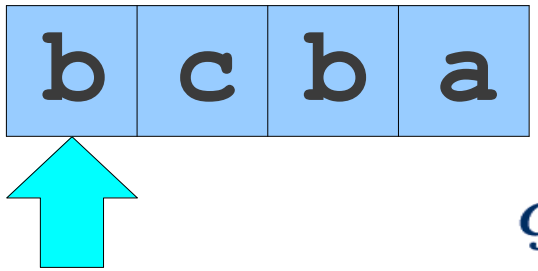




GEORGETOWN UNIVERSITY

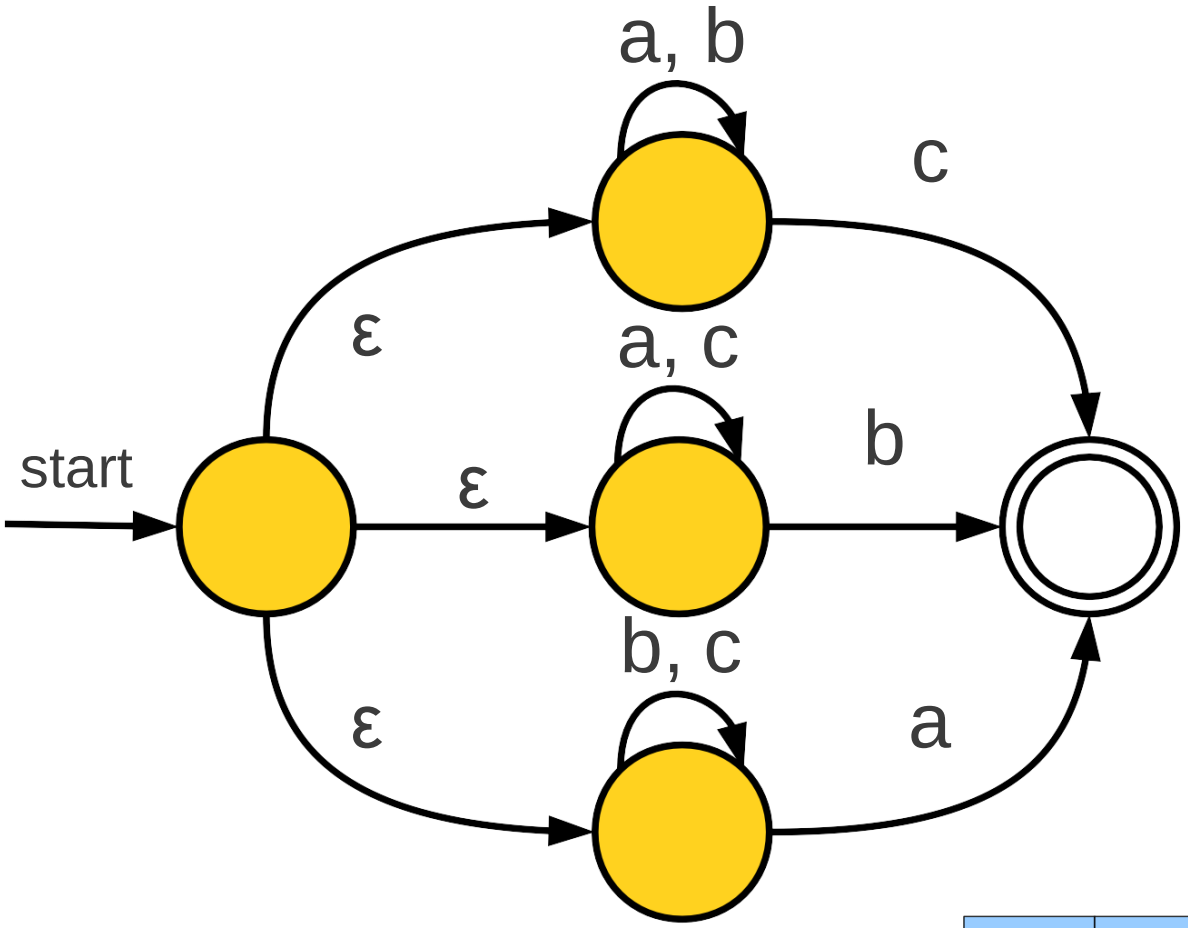
An Even More Complex Automaton

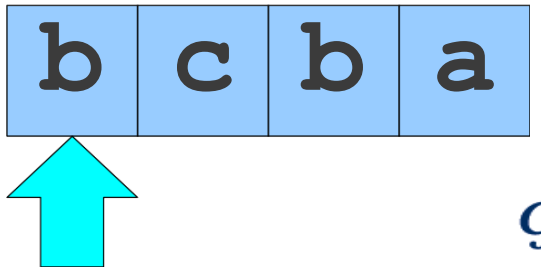




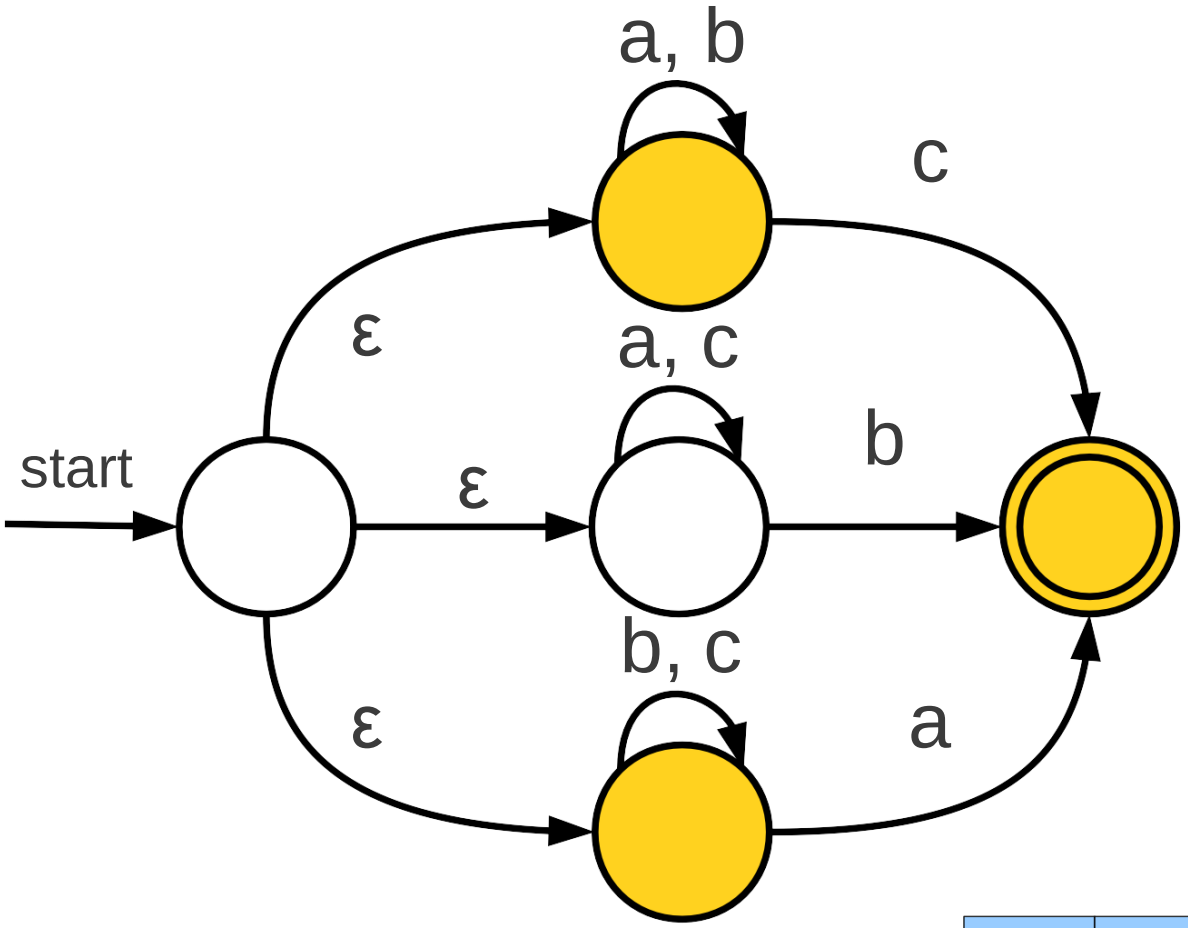
GEORGETOWN UNIVERSITY

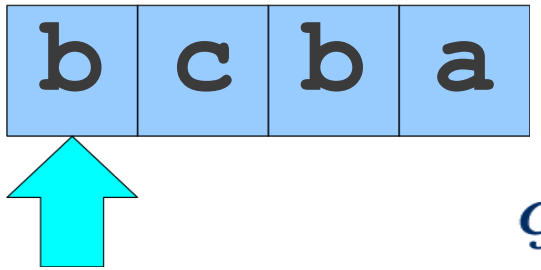
An Even More Complex Automaton



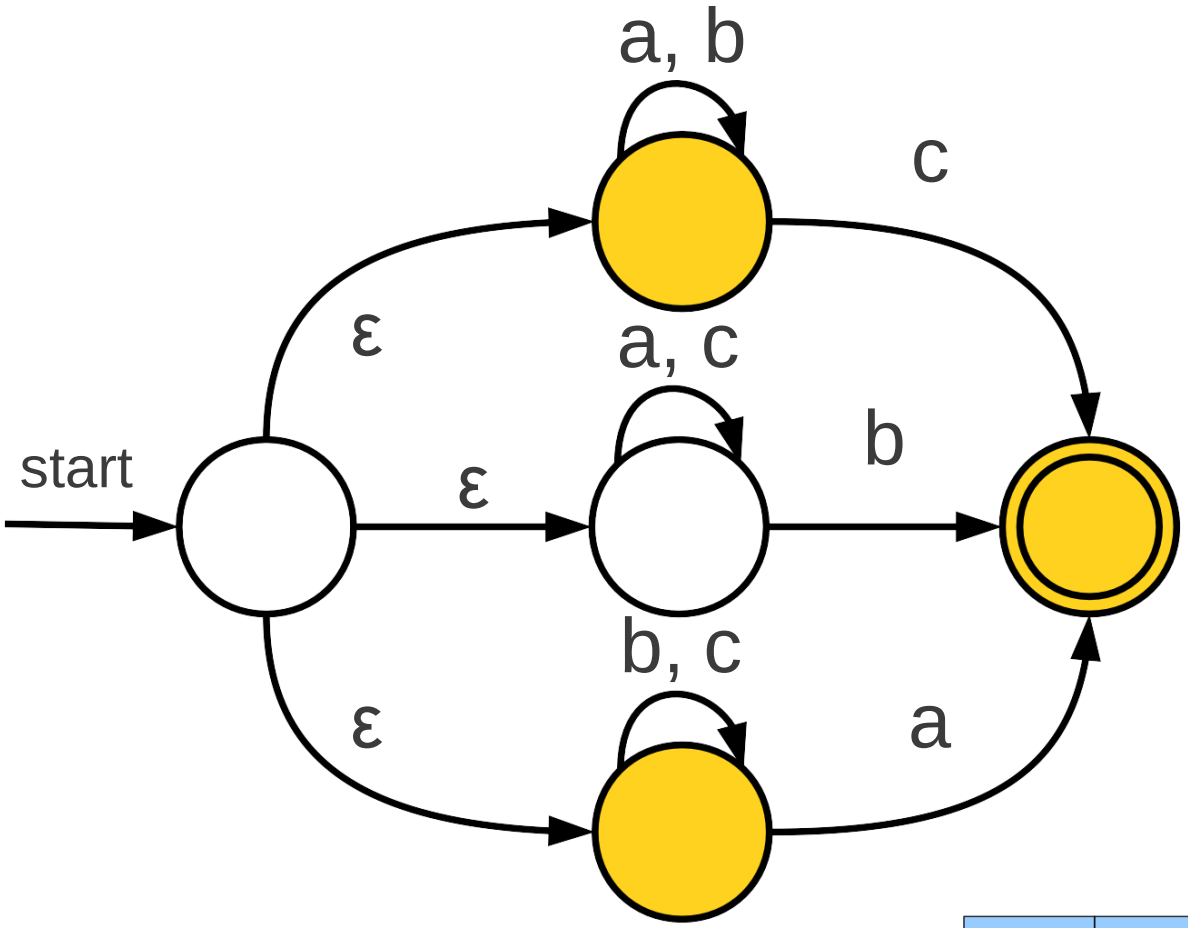


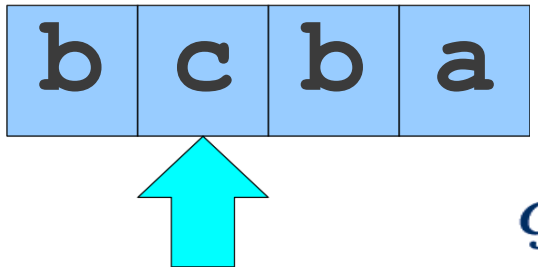
An Even More Complex Automaton





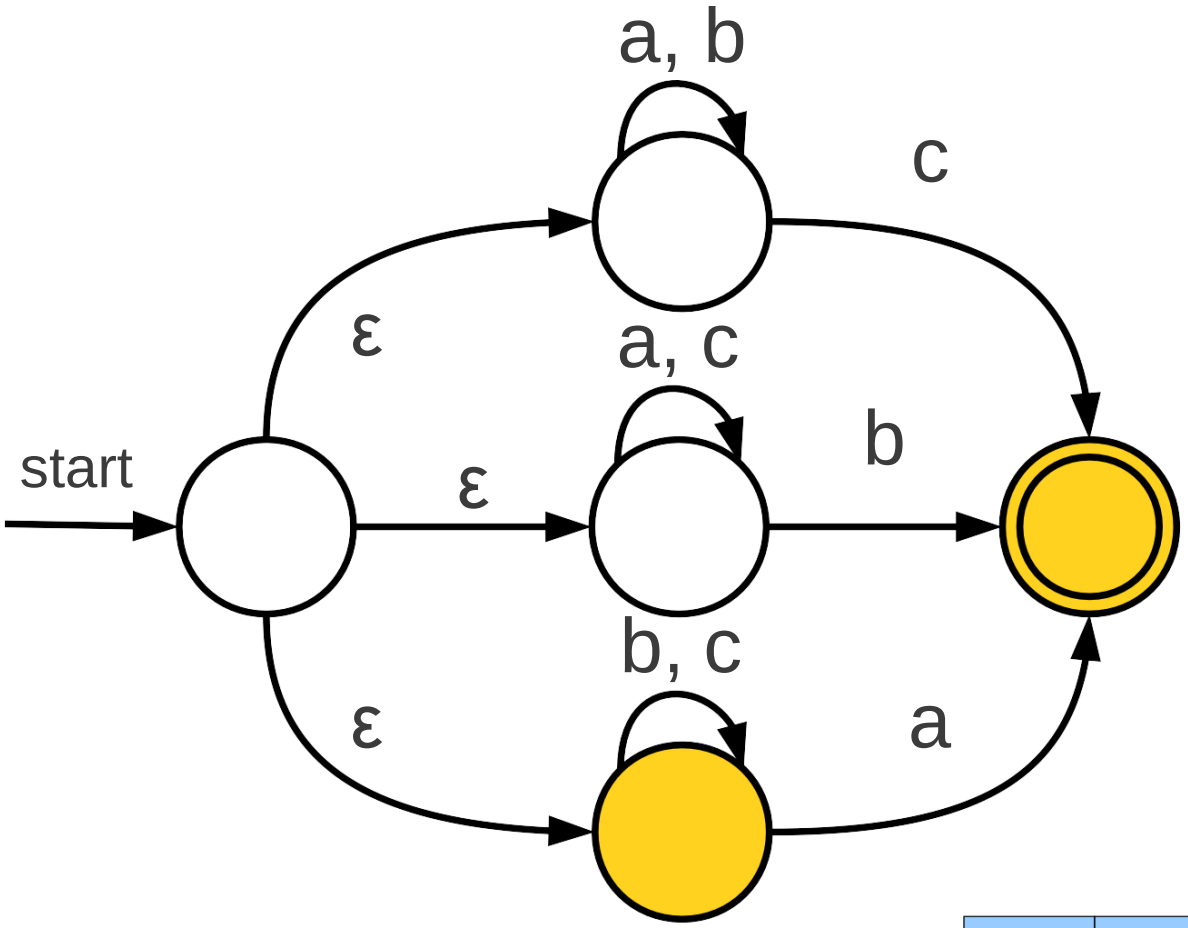
An Even More Complex Automaton

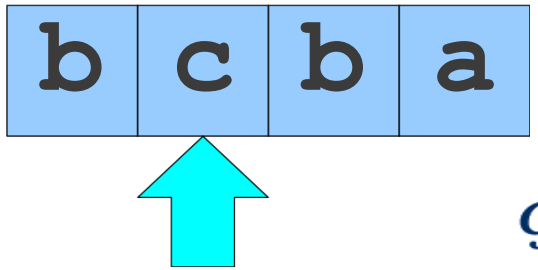




GEORGETOWN UNIVERSITY

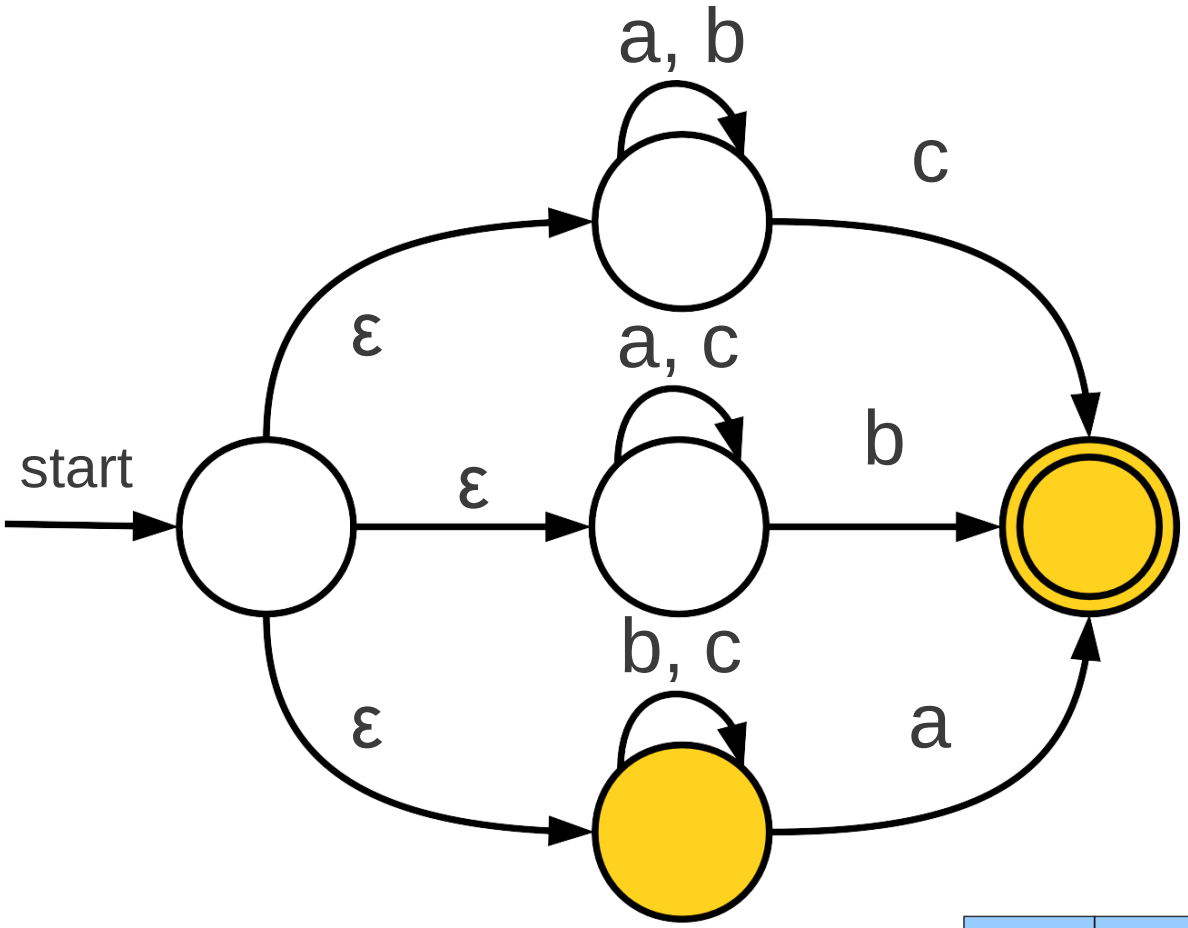
An Even More Complex Automaton

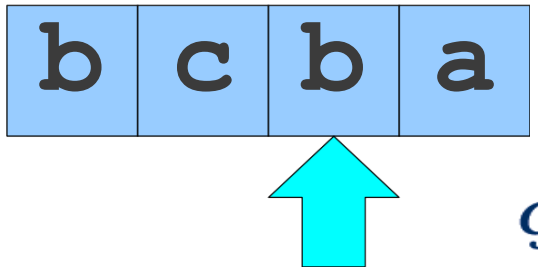




GEORGETOWN UNIVERSITY

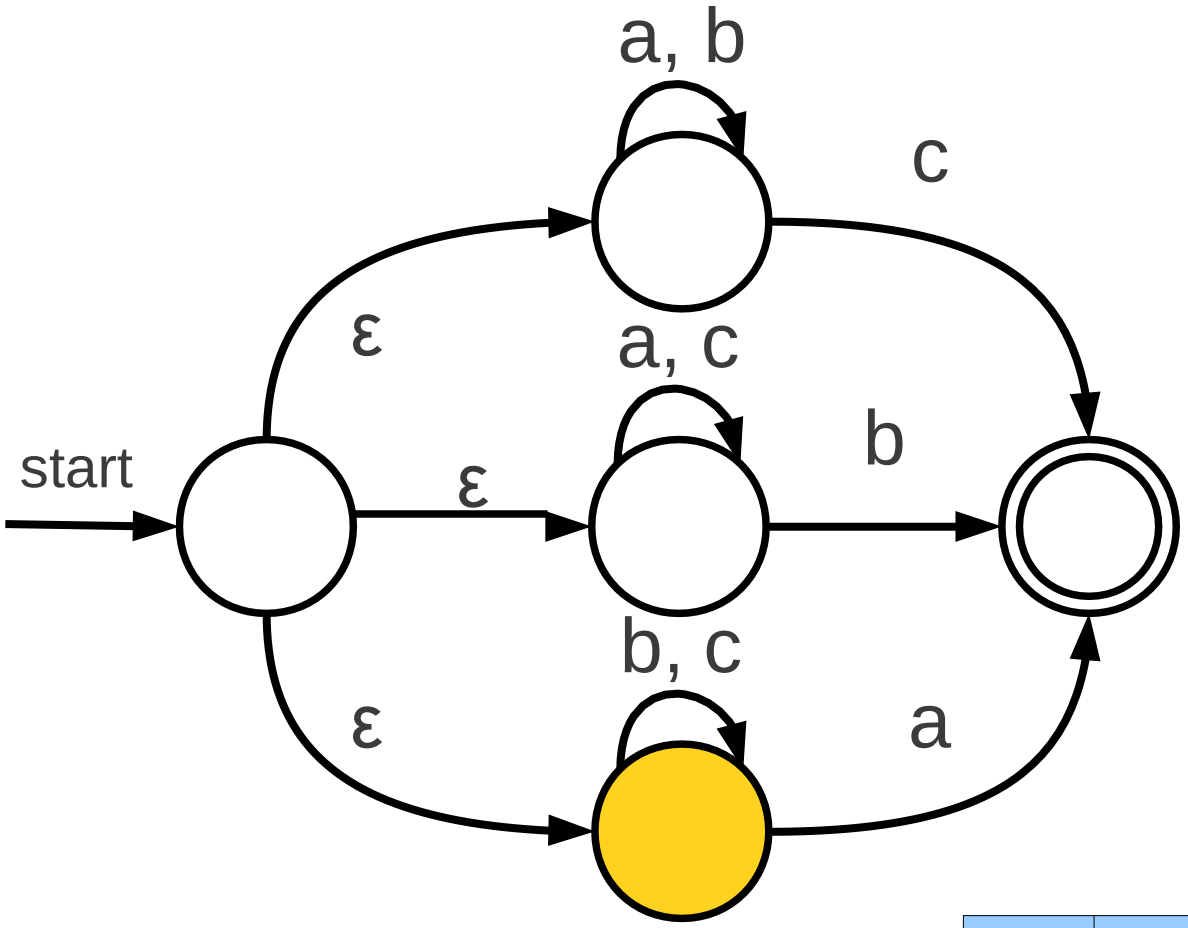
An Even More Complex Automaton

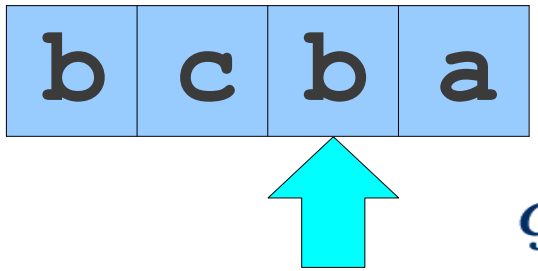




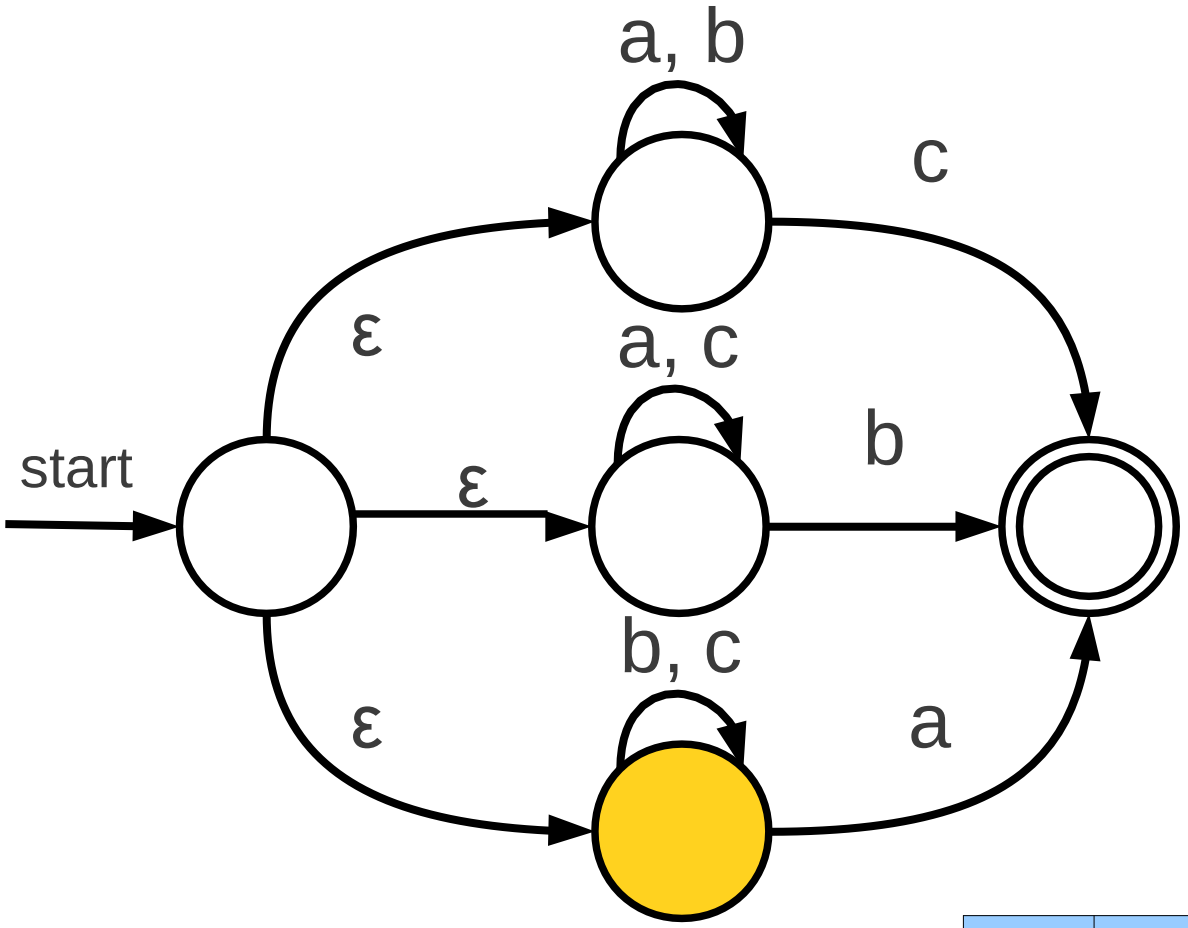
GEORGETOWN UNIVERSITY

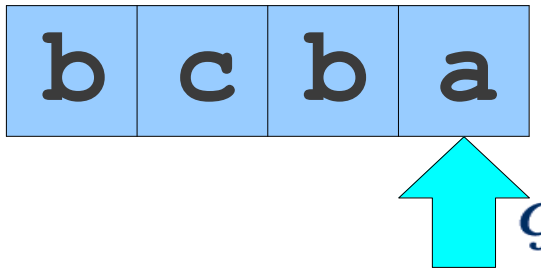
An Even More Complex Automaton





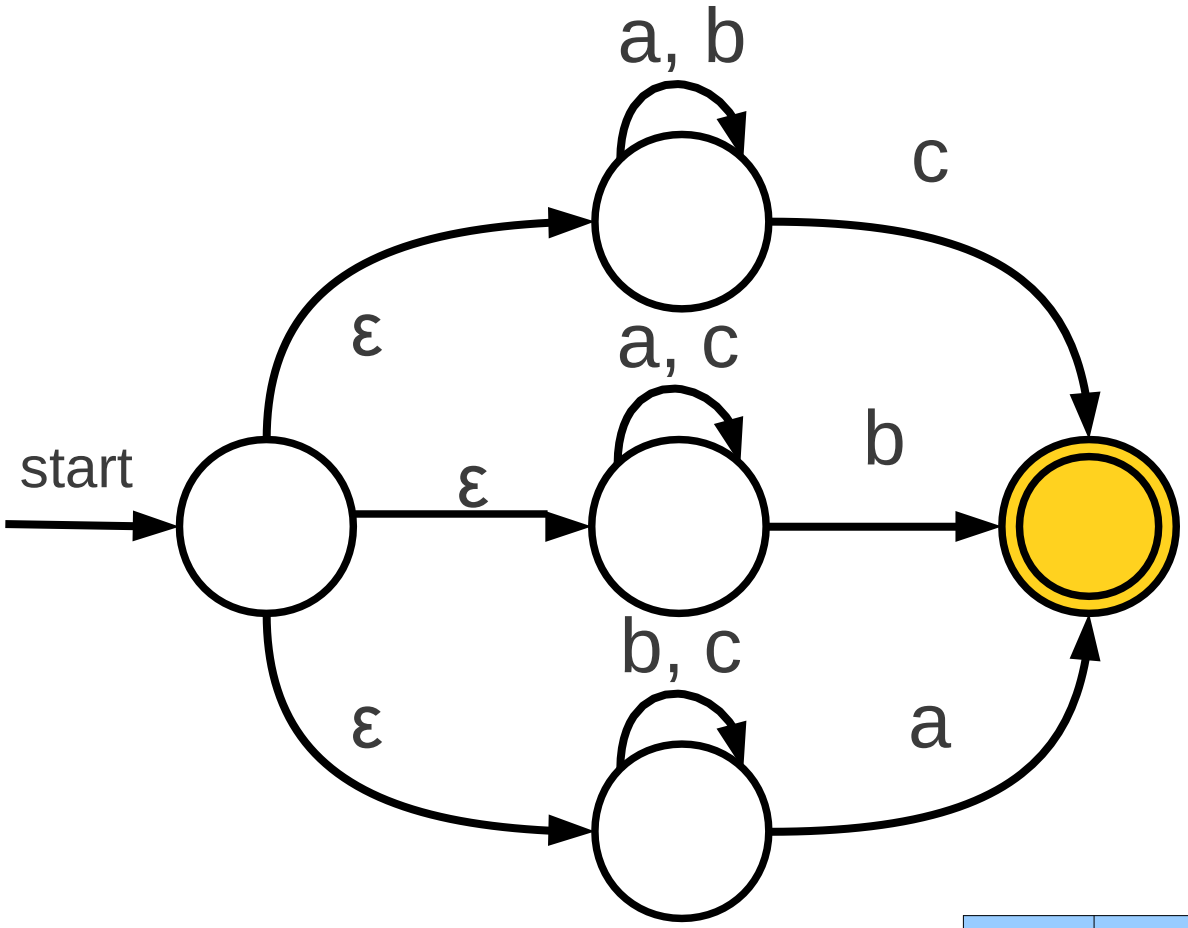
An Even More Complex Automaton

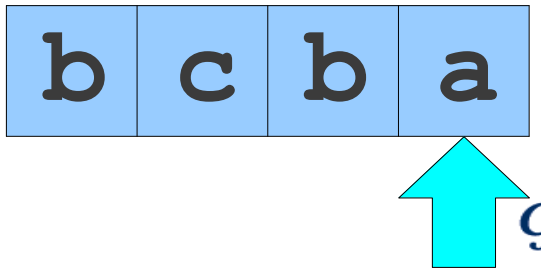




GEORGETOWN UNIVERSITY

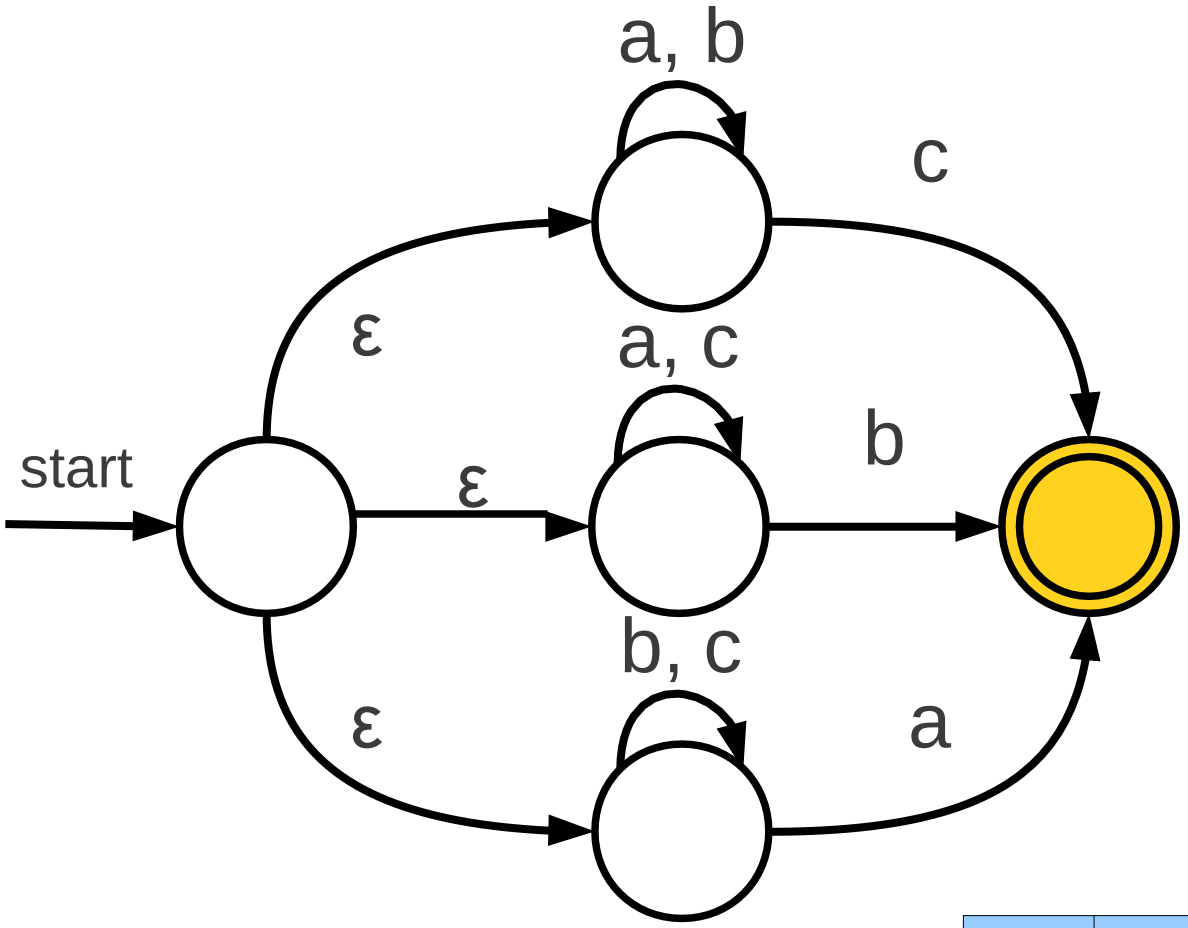
An Even More Complex Automaton





GEORGETOWN UNIVERSITY

An Even More Complex Automaton



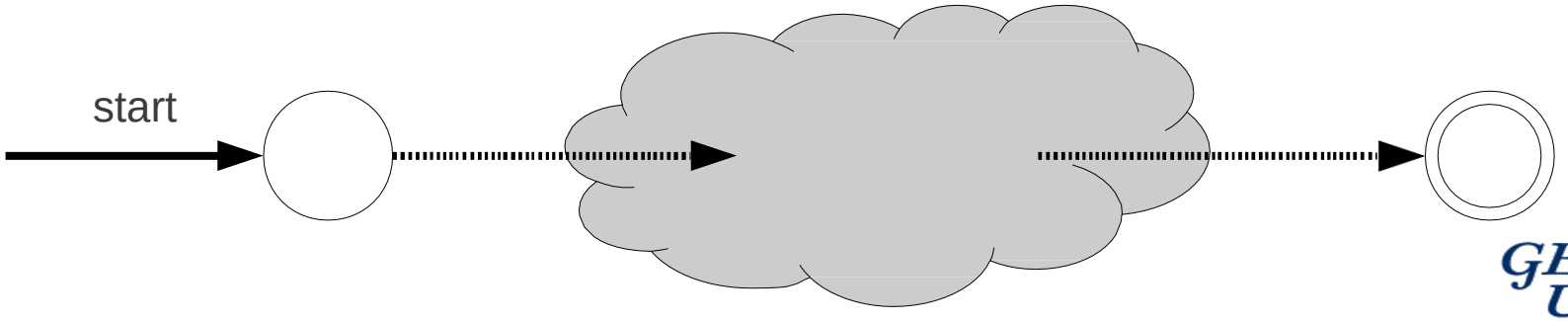
b	c	b	a

Simulating an NFA

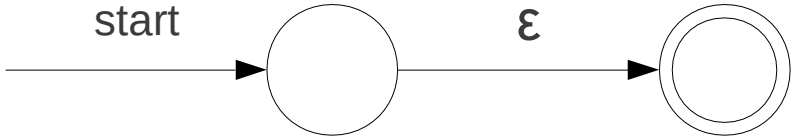
- Keep track of a set of states, initially the start state and everything reachable by ε-moves.
- For each character in the input:
  - Maintain a set of next states, initially empty.
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an ε-move to the set of next states.
- Complexity: $O(mn^2)$ for strings of length m and automata with n states.

From Regular Expressions to NFAs

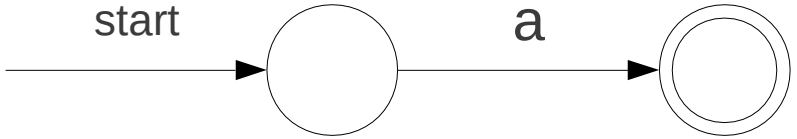
- There is a (beautiful!) procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting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



Base Cases

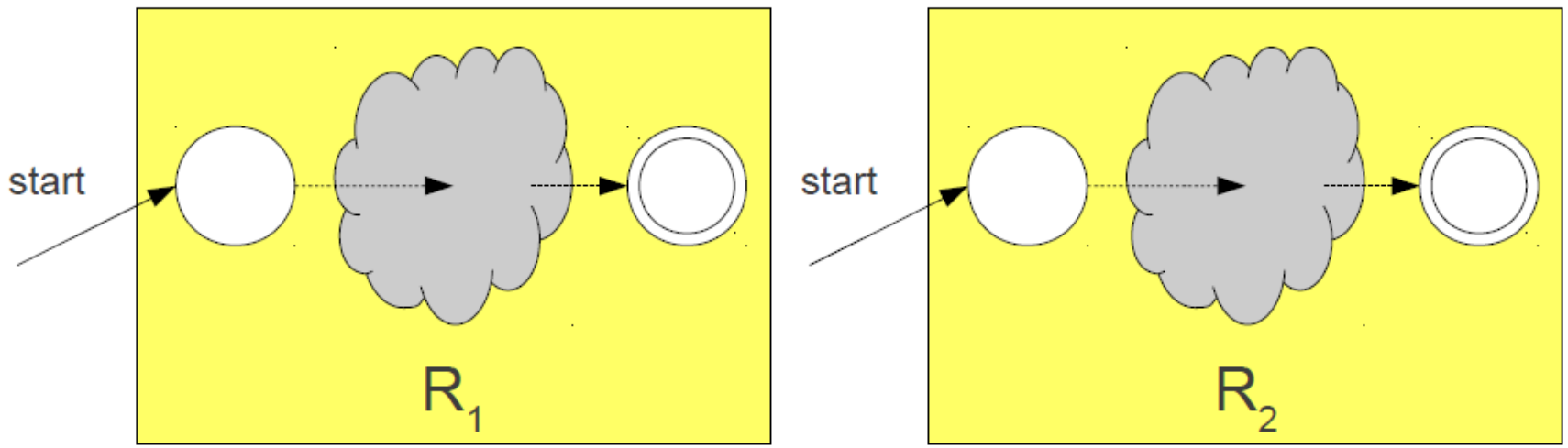


Automaton for ε

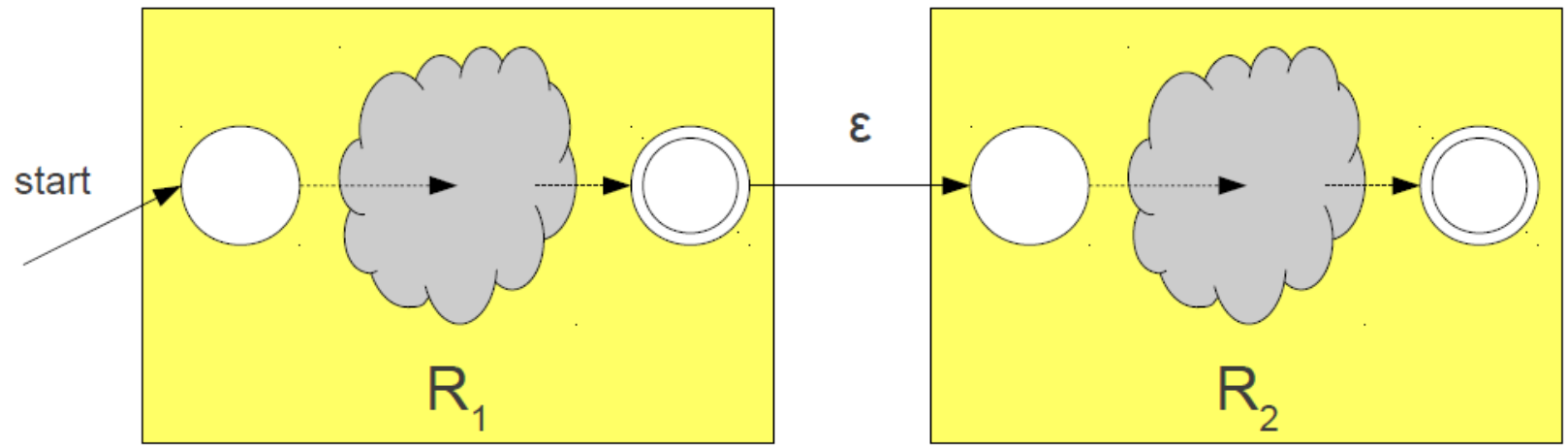


Automaton for single character **a**

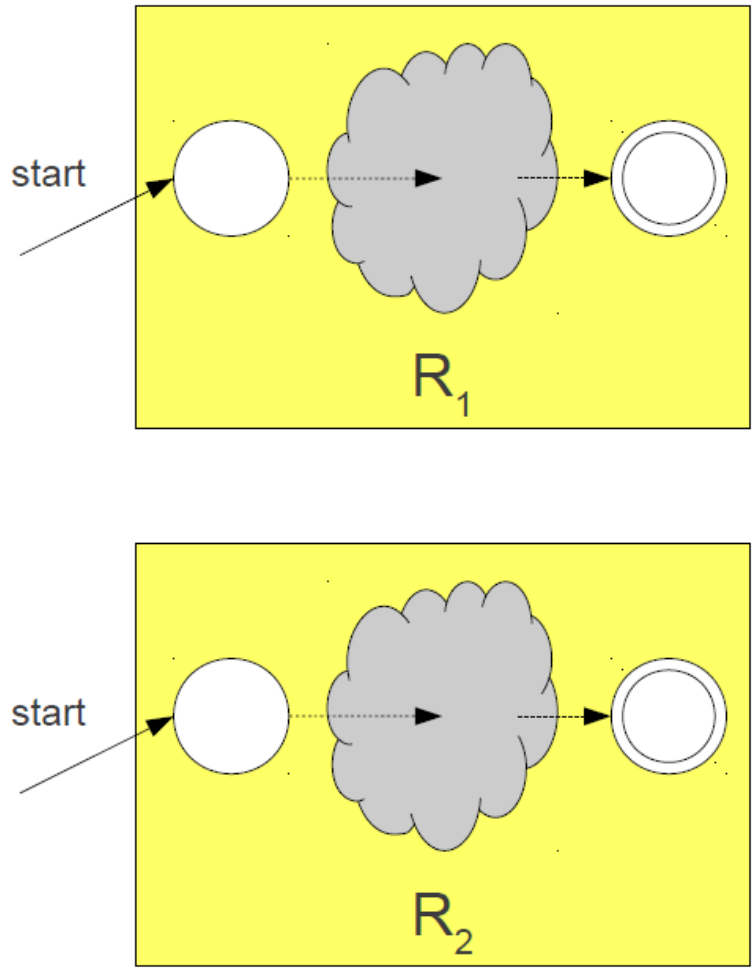
Construction for $R_1 R_2$



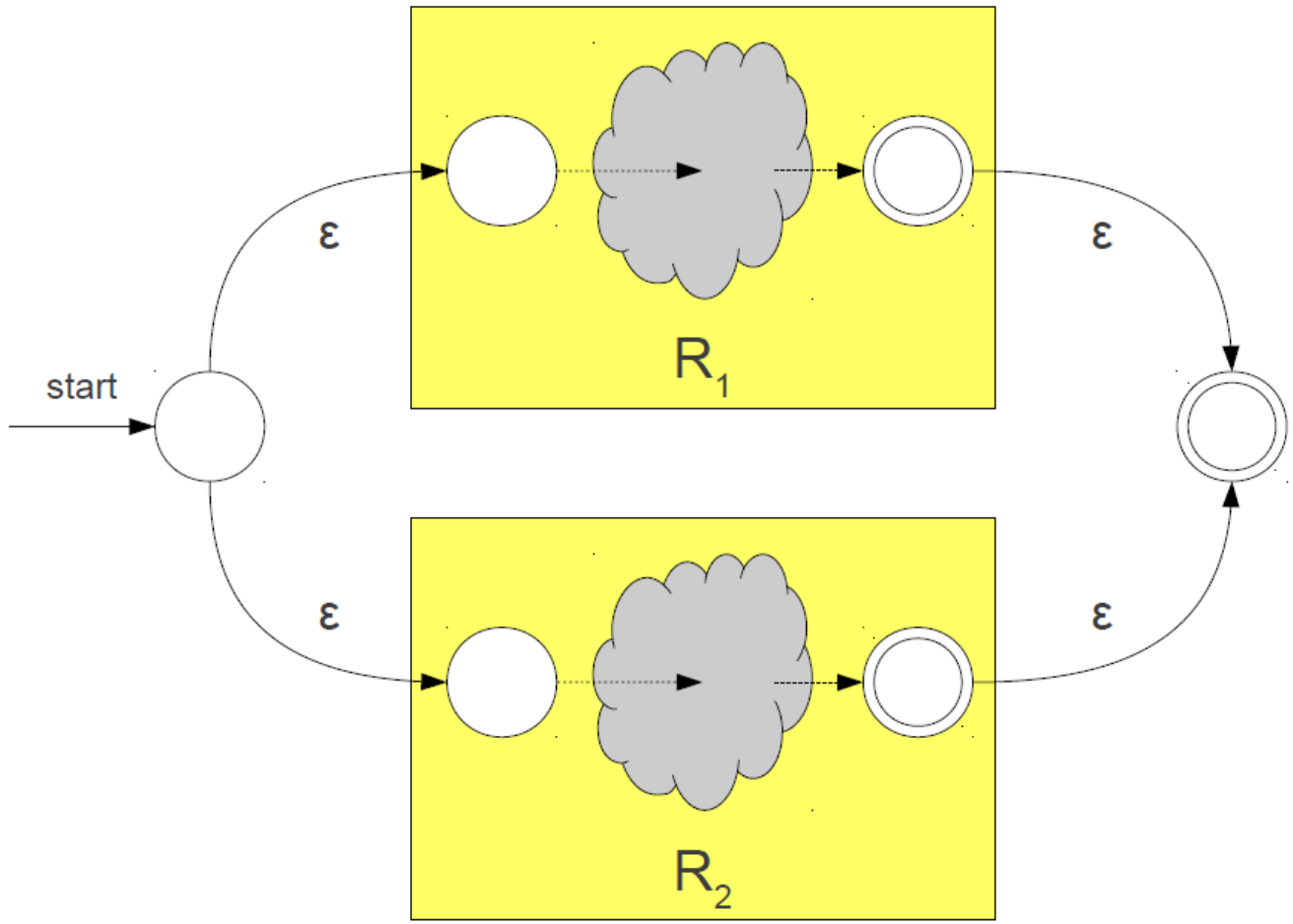
Construction for $R_1 R_2$



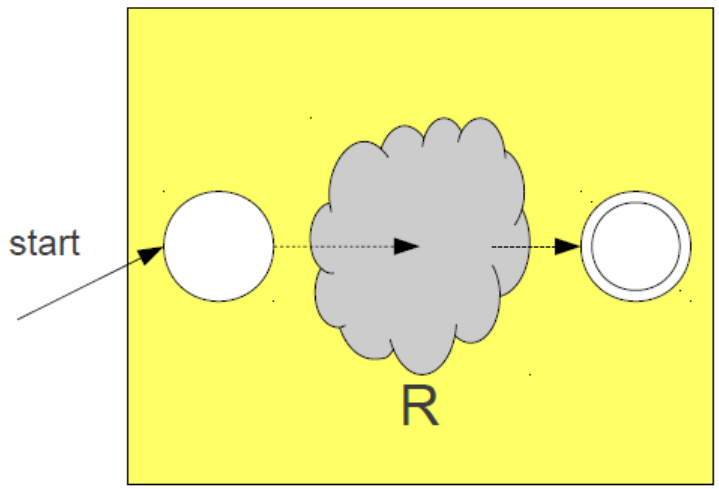
Construction for $R_1 \mid R_2$



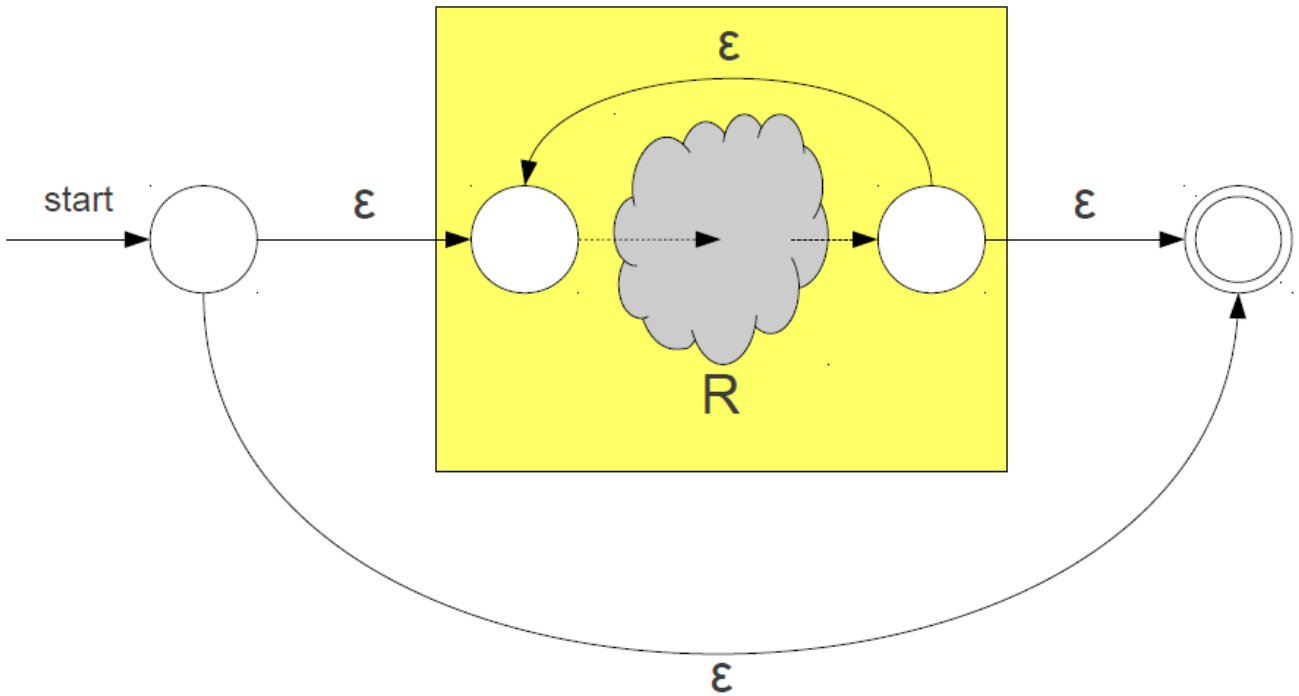
Construction for $R_1 \mid R_2$



Construction for R*



Construction for R*



Challenges in Scanning

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

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_]*
```

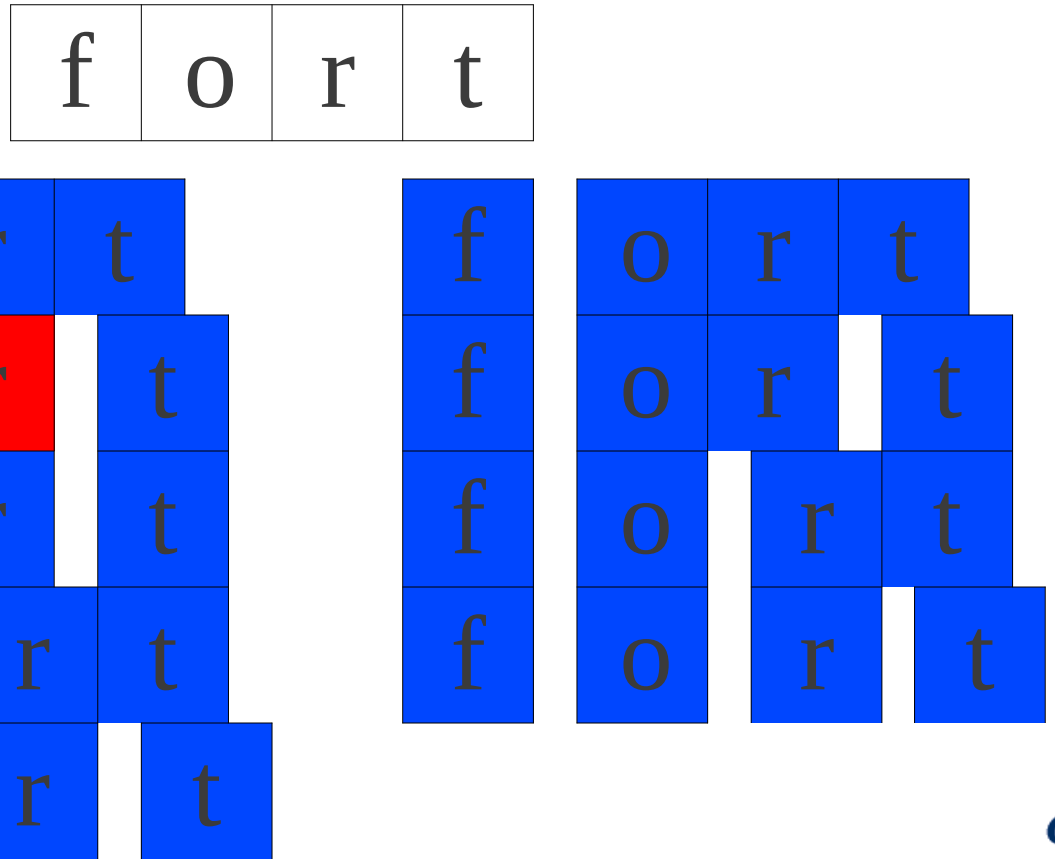
Lexing Ambiguities

```
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_]*
```



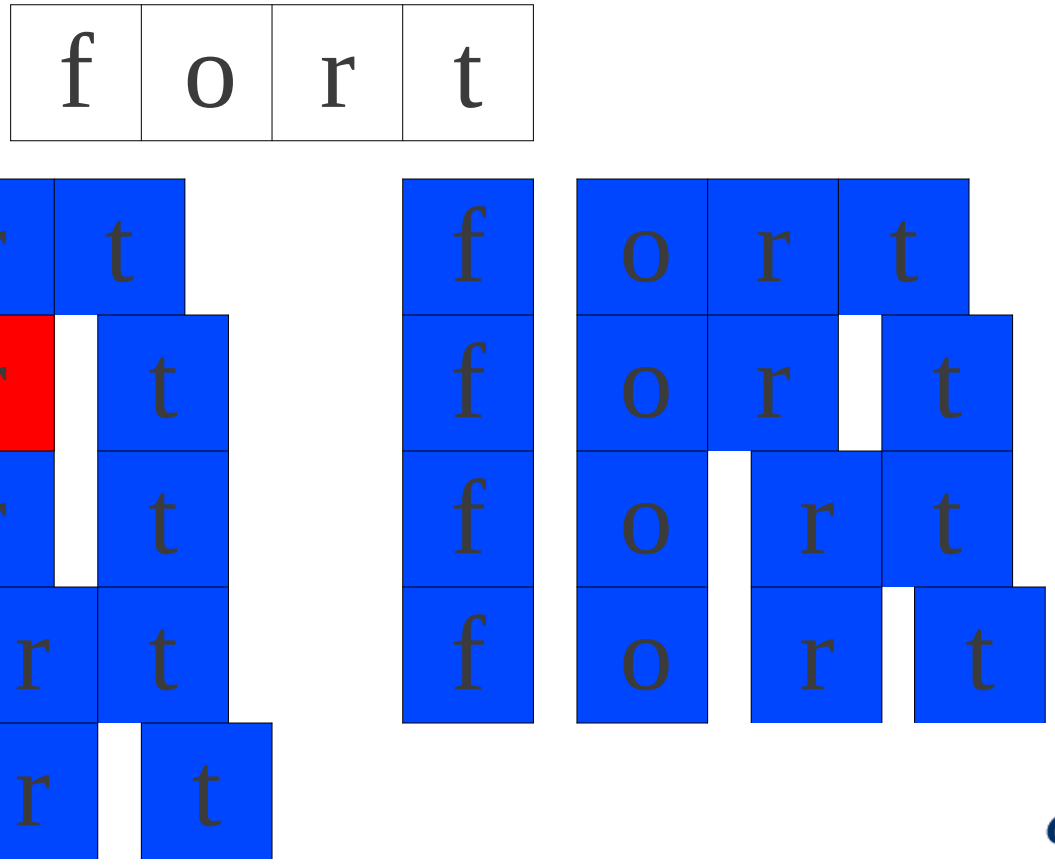
GEORGETOWN UNIVERSITY

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

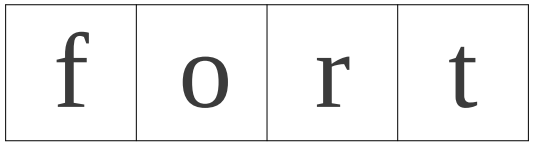
```
T_For          for
T_Identifier   [A-Za-z_][A-Za-z0-9_]*
```

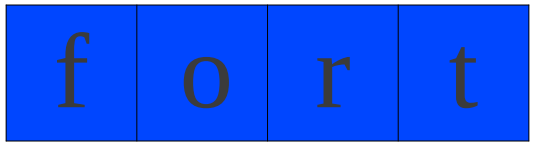


GEORGETOWN UNIVERSITY

Lexing Ambiguities

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





Implementing Maximal Munch

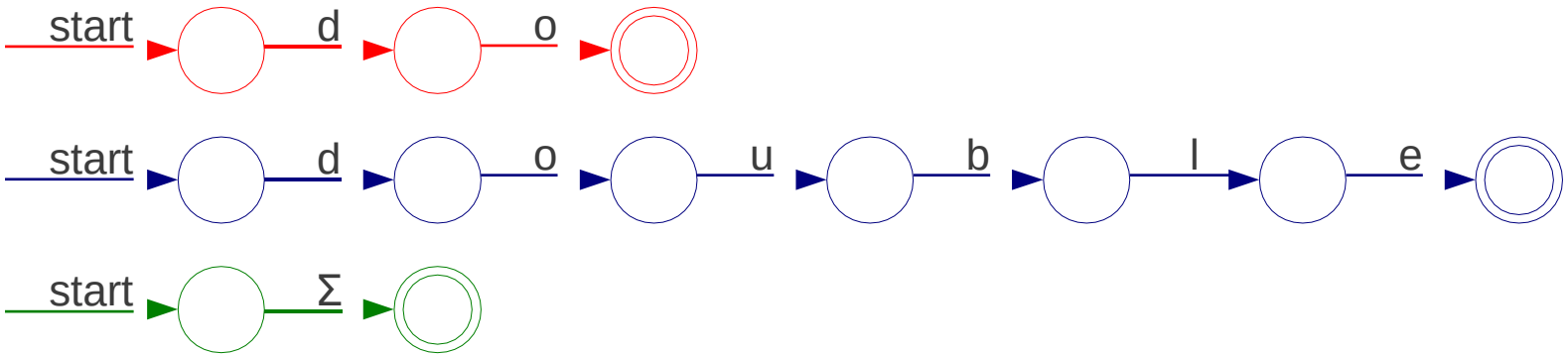
- Given a set of regular expressions, how can we use them to implement maximum munch?
- Idea:
  - Convert expressions to NFAs.
  - Run all NFAs in parallel, keeping track of the last match.
  - When all automata get stuck, report the last match and restart the search at that point.

Implementing Maximal Munch

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

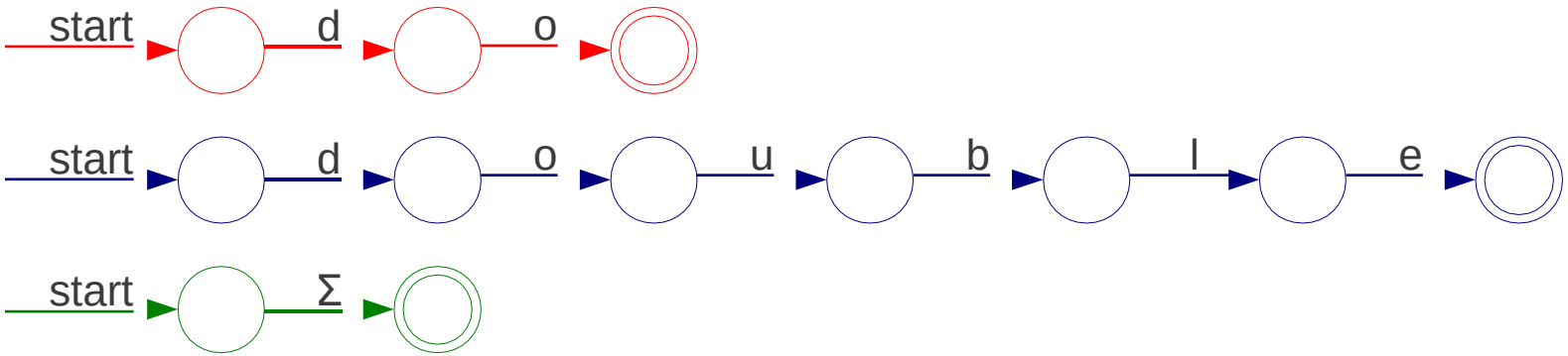
Implementing Maximal Munch

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



Implementing Maximal Munch

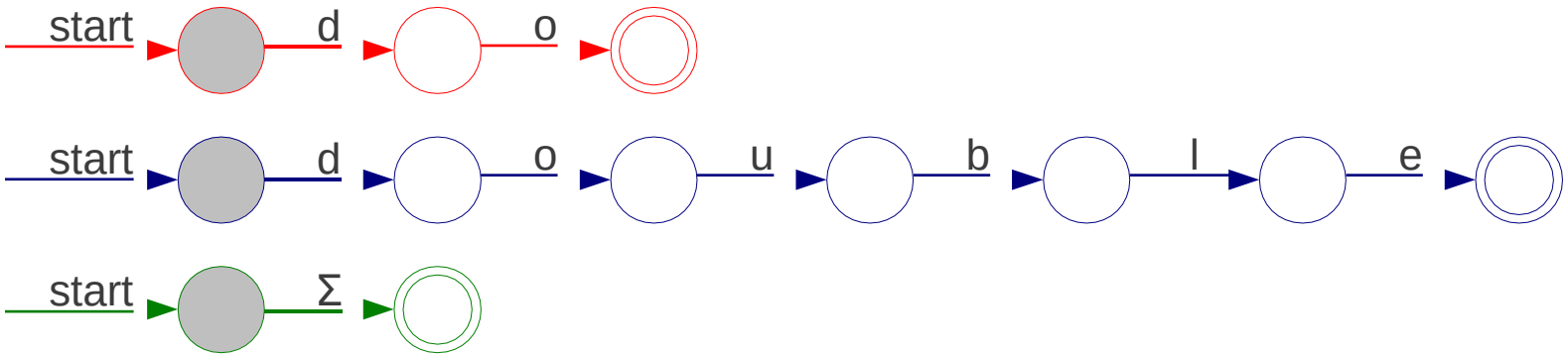
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

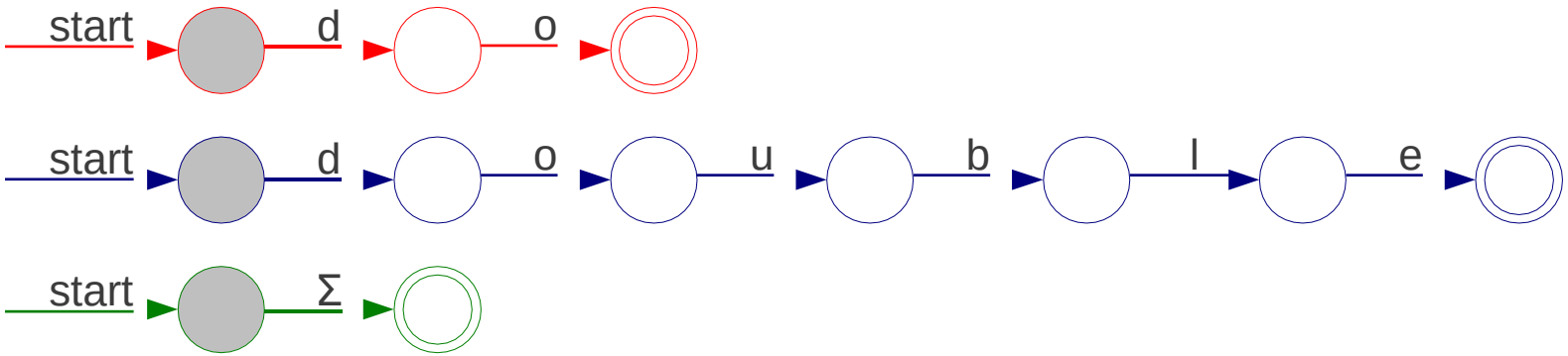
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

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

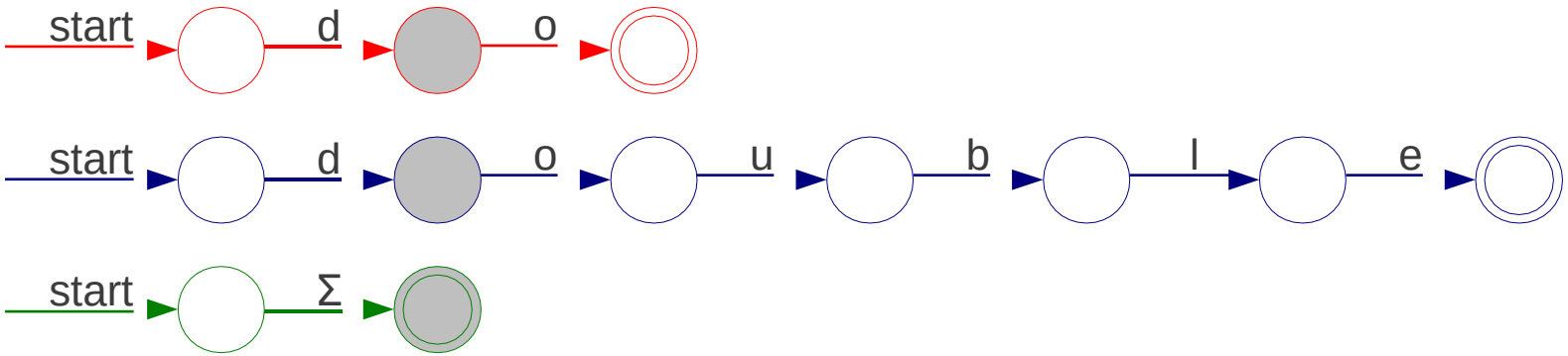


D	O	U	B	D	O	U	B	L	E

GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

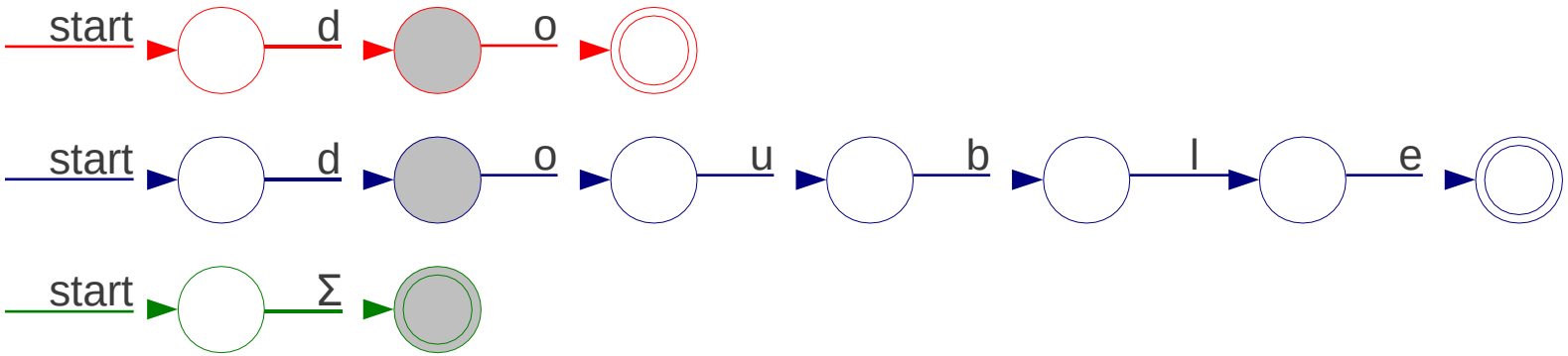


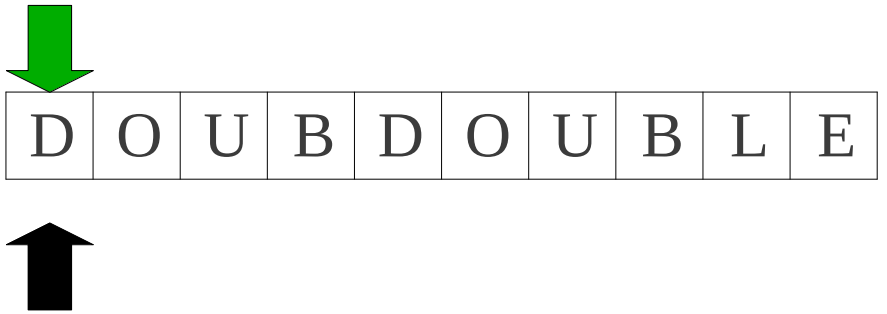
D	O	U	B	D	O	U	B	L	E

GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

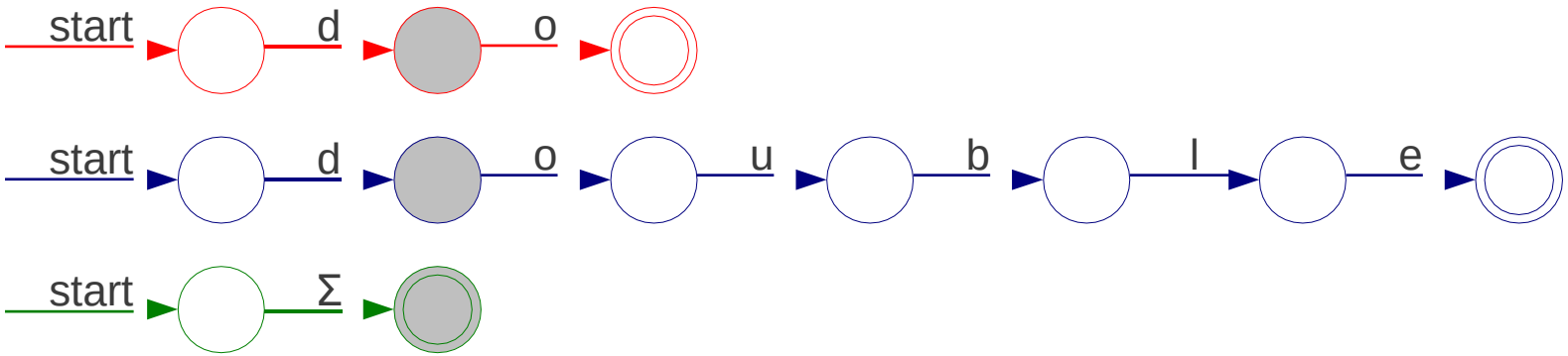


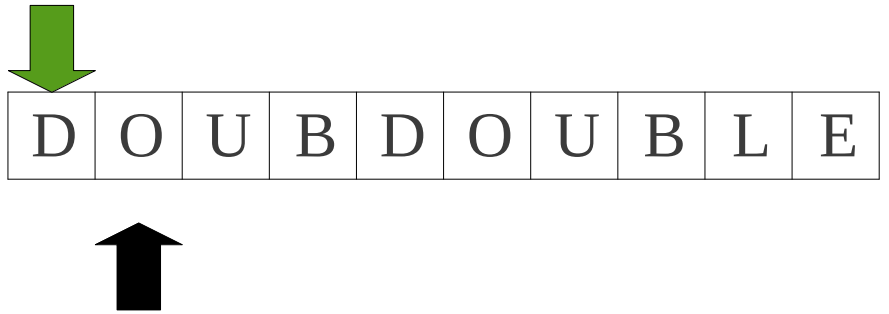


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

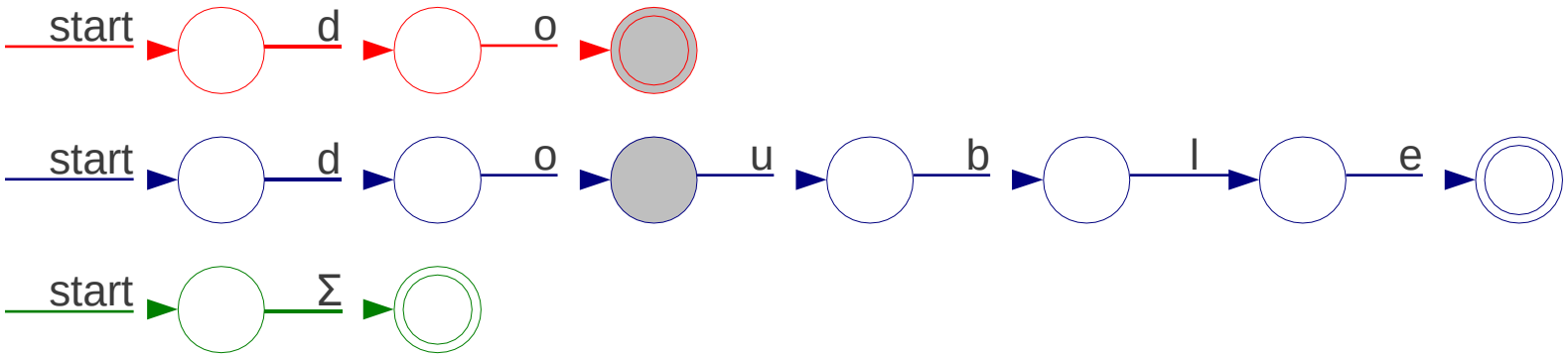


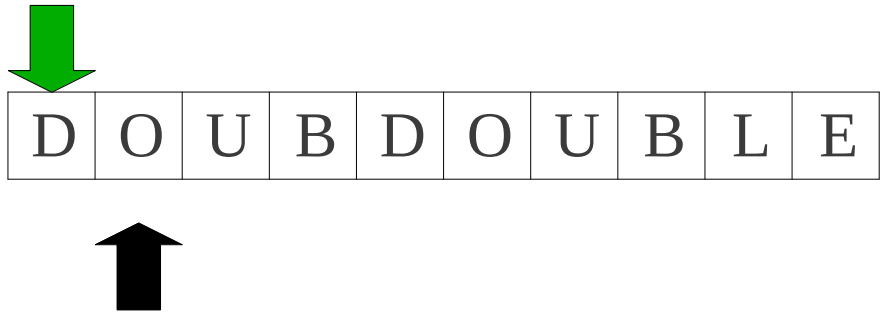


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

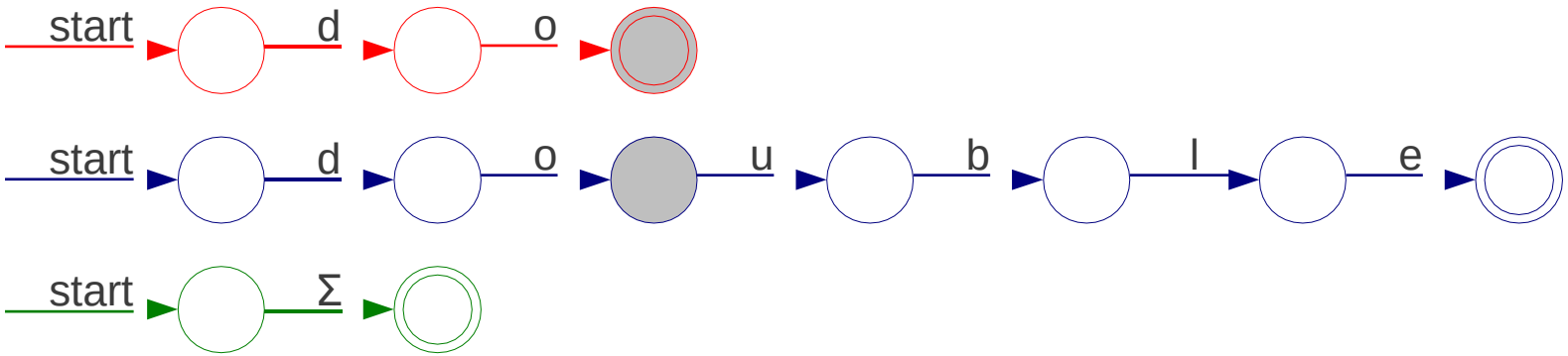


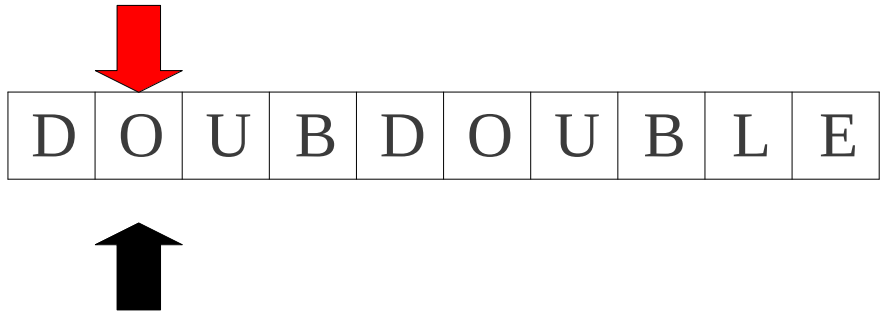


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

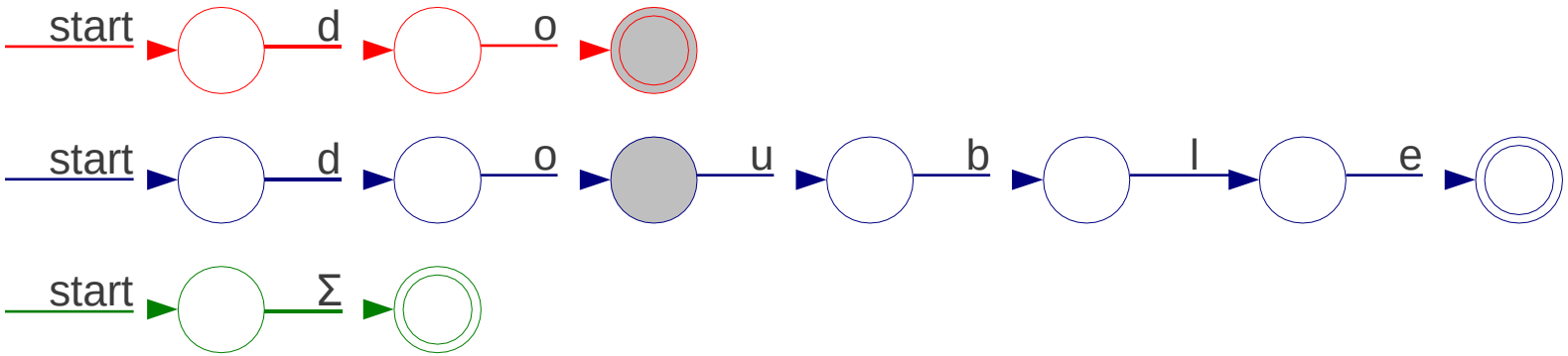


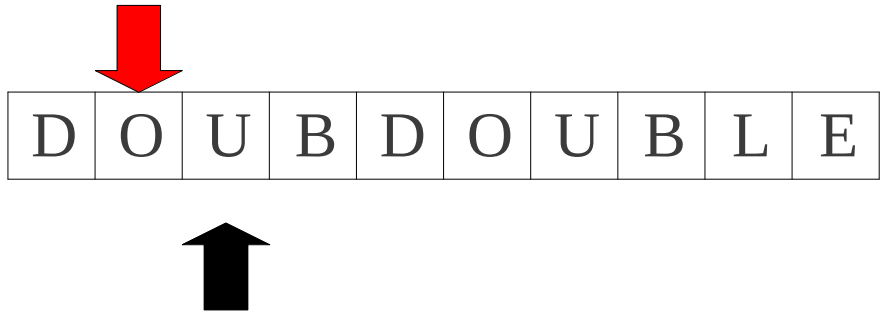


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

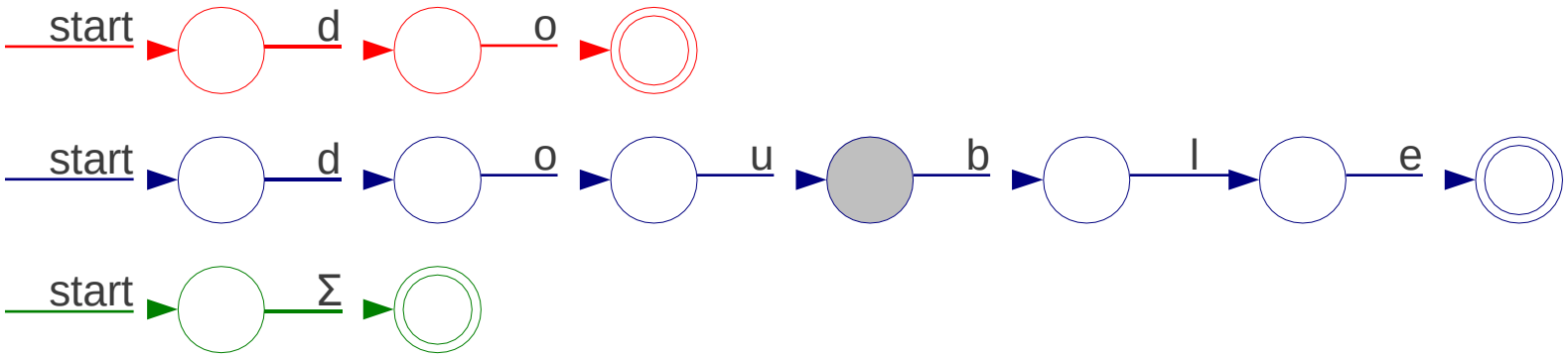


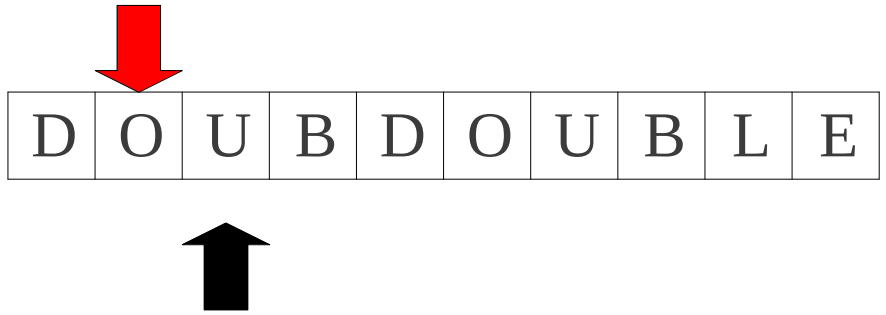


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

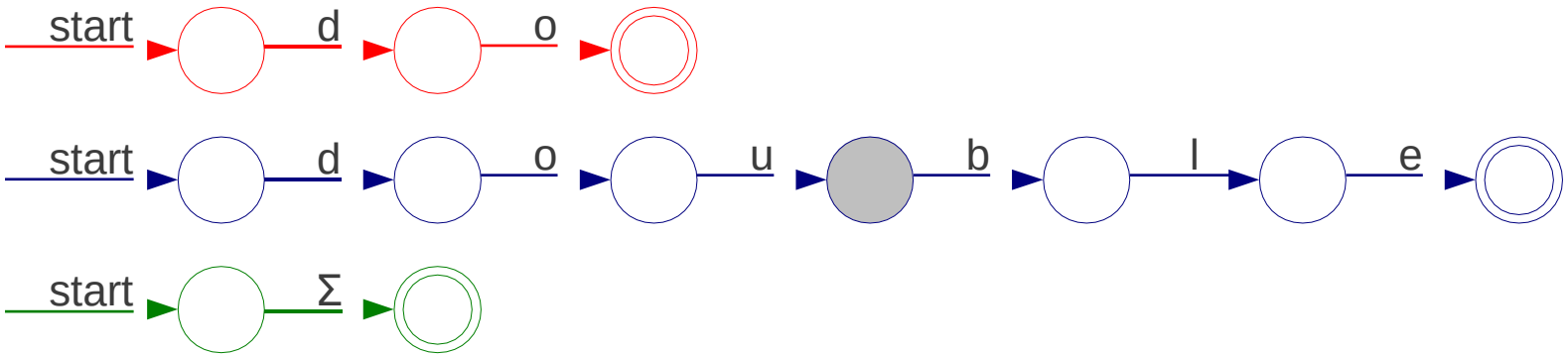


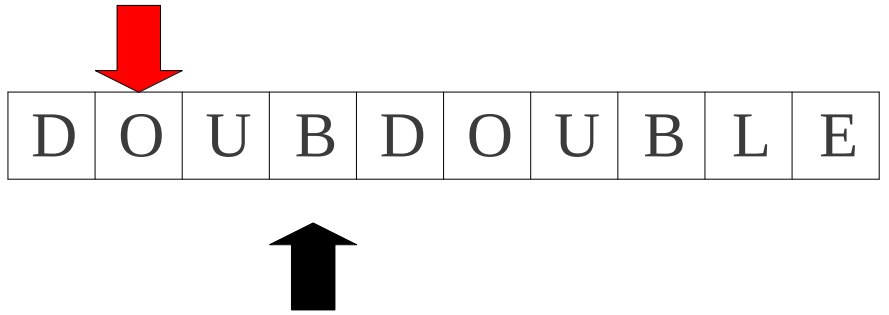


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

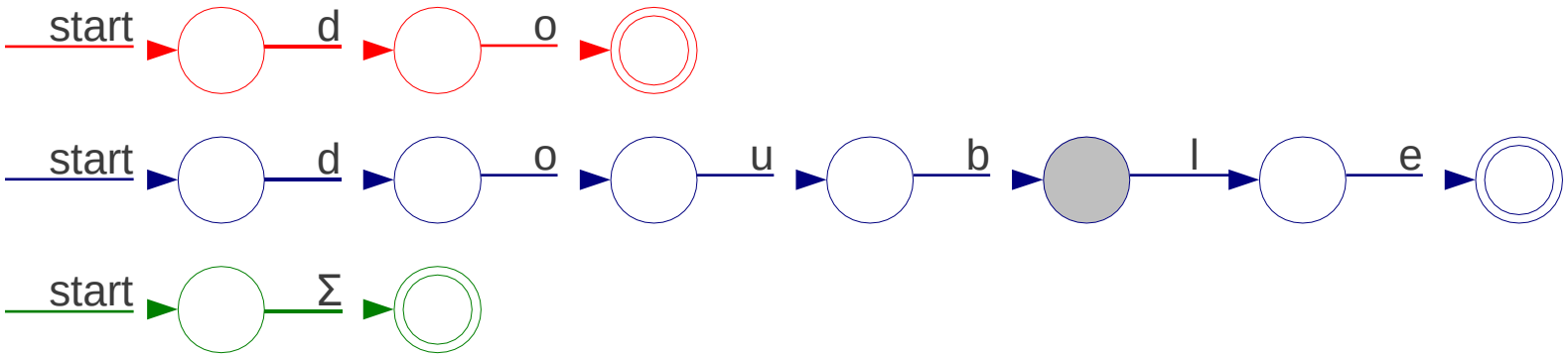


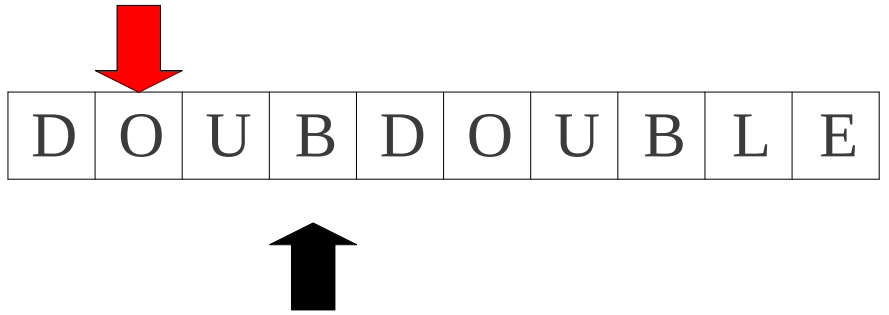


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

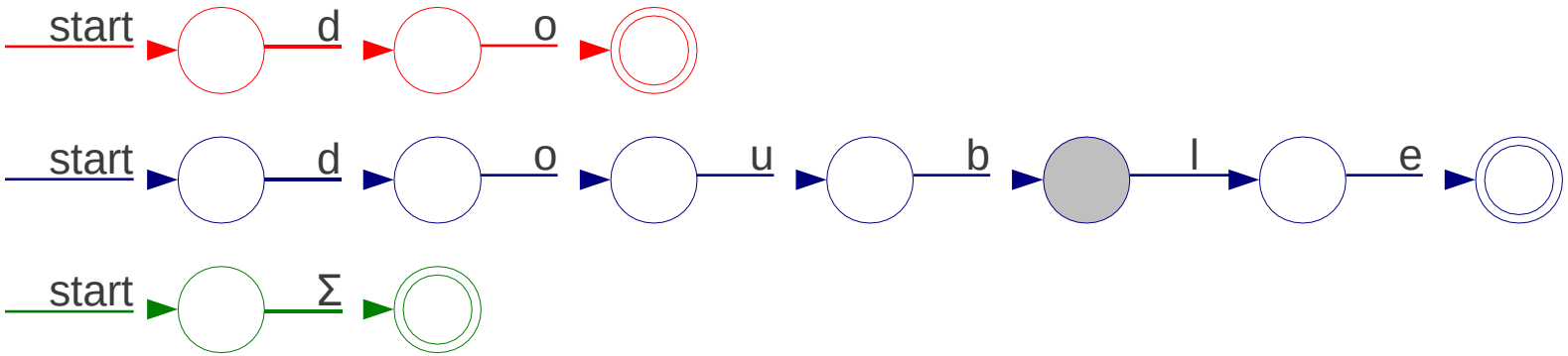


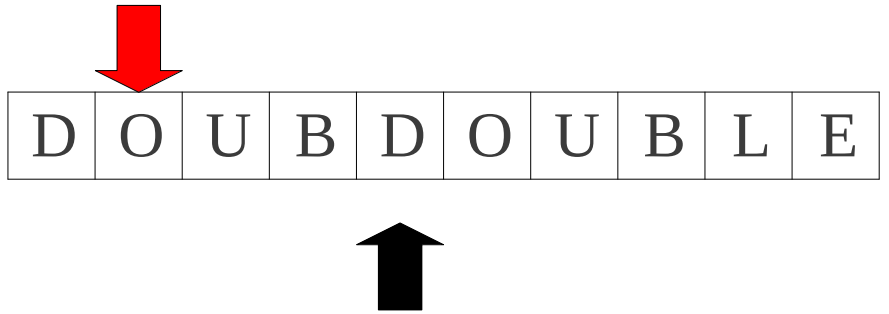


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

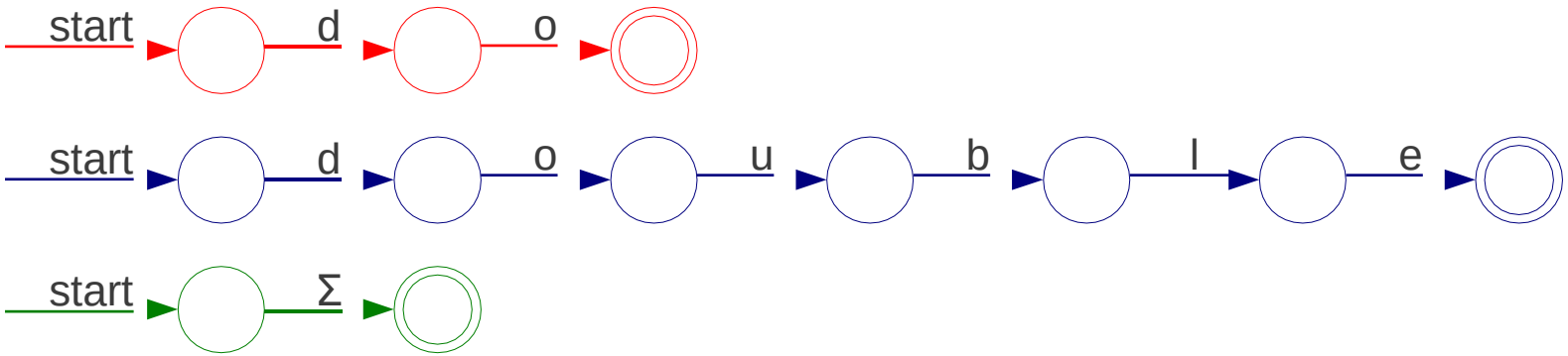


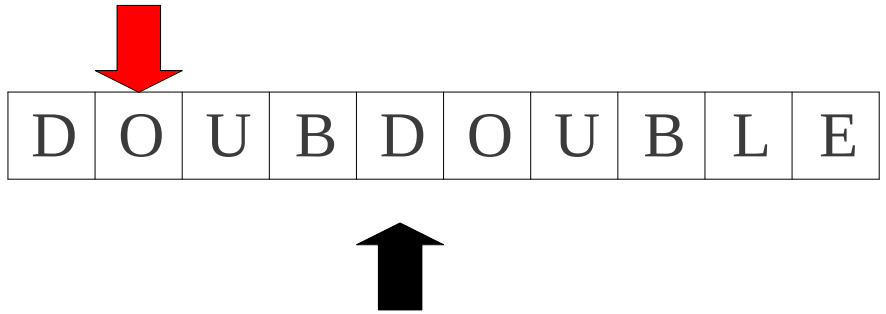


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

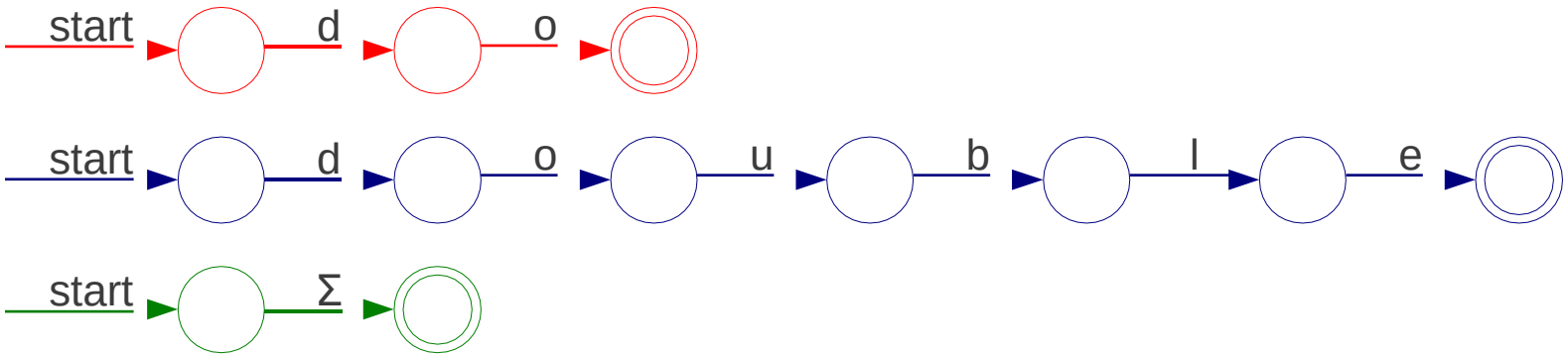


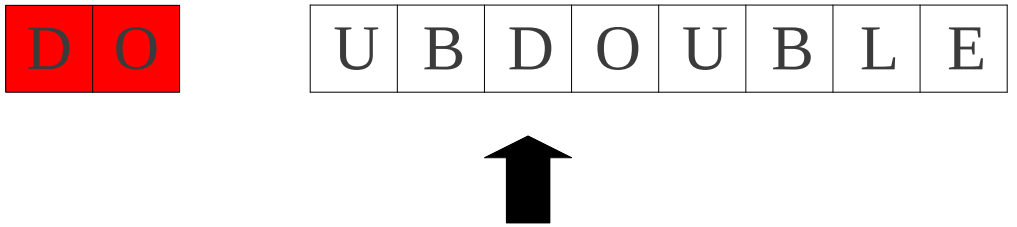


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

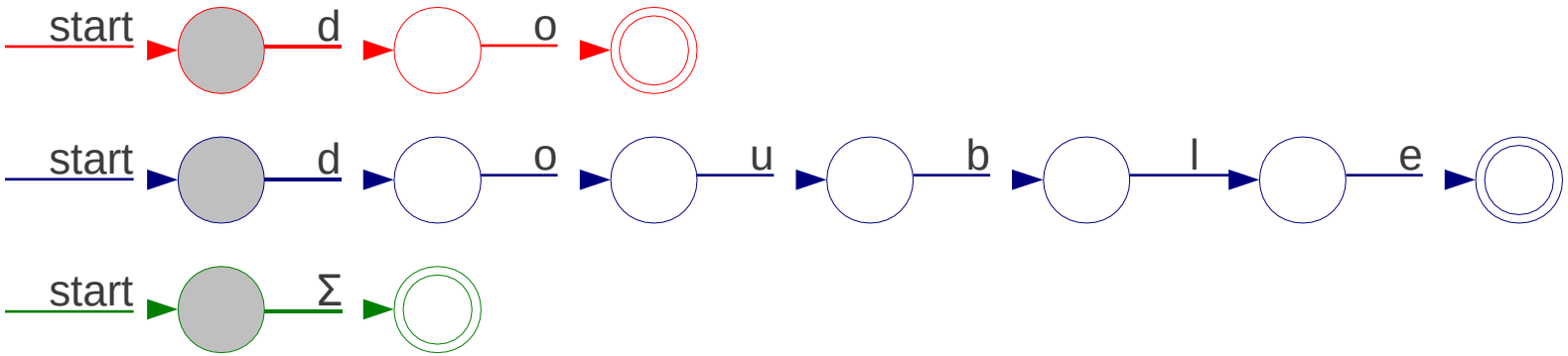


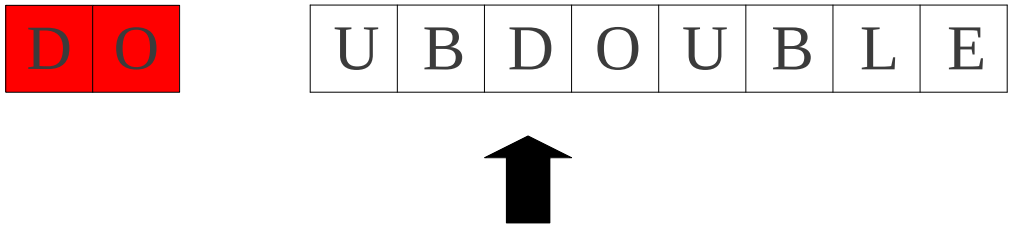


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

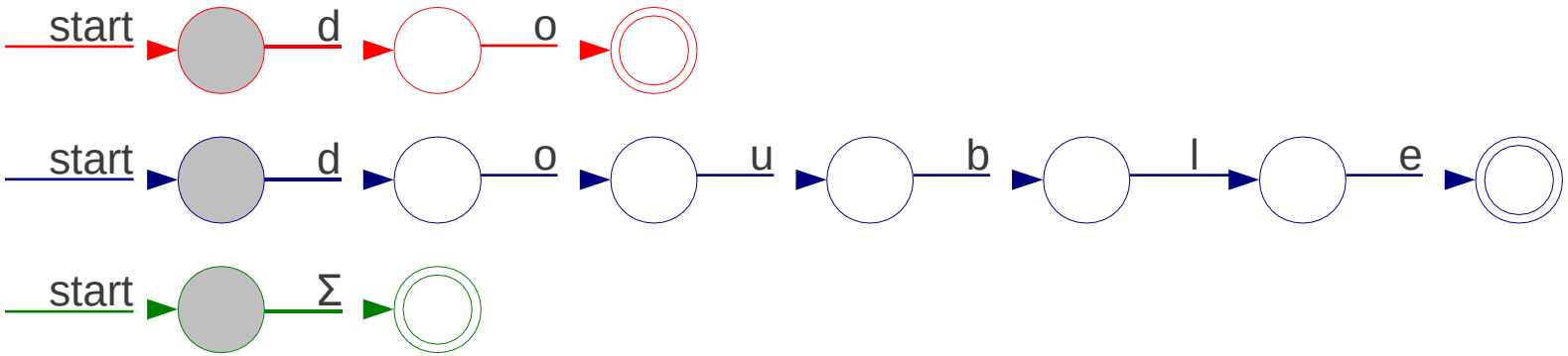




GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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



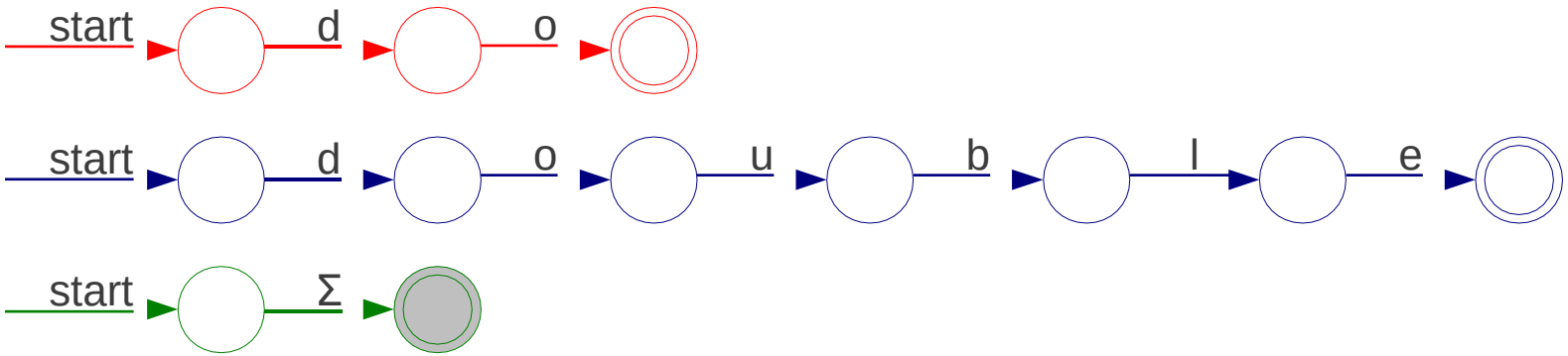
D	O

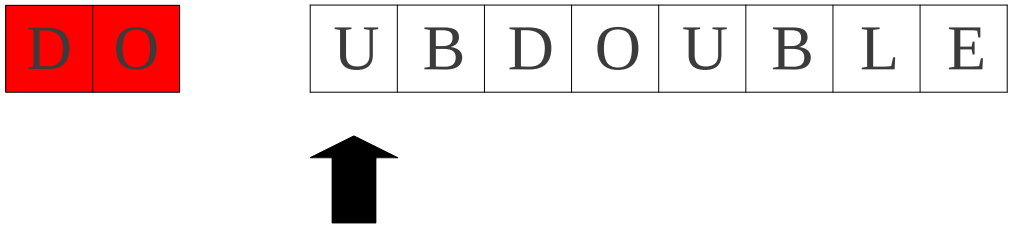
U	B	D	O	U	B	L	E

GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

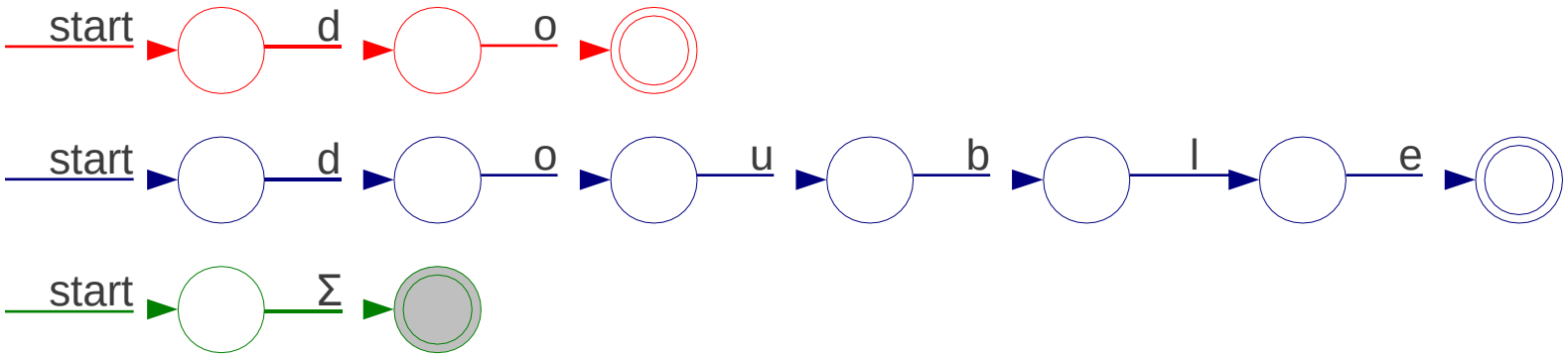


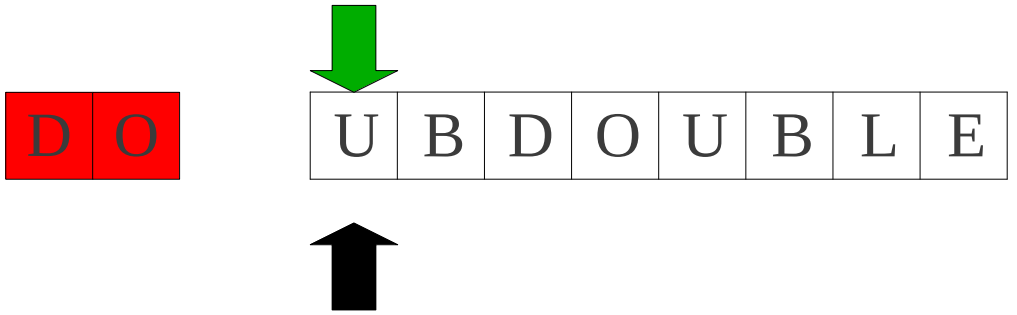


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

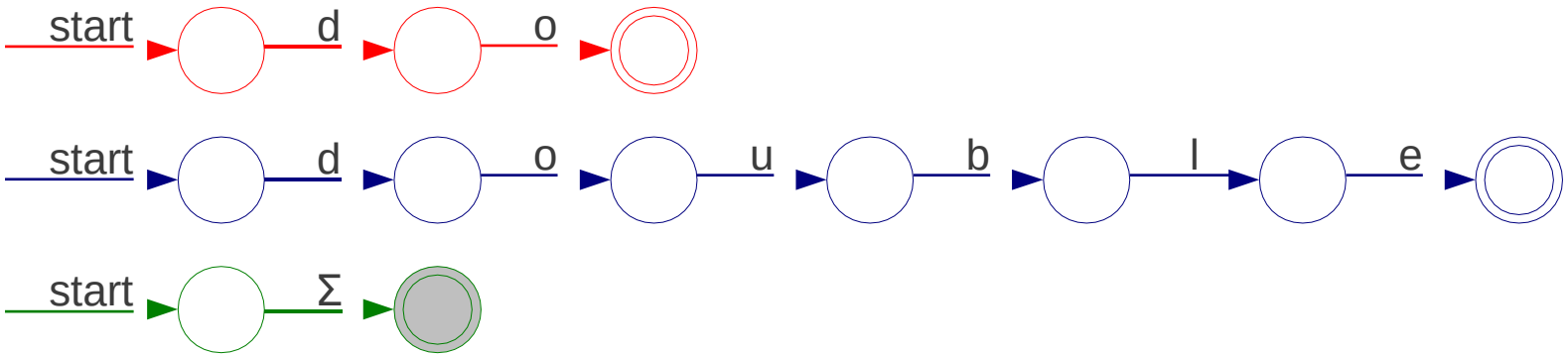


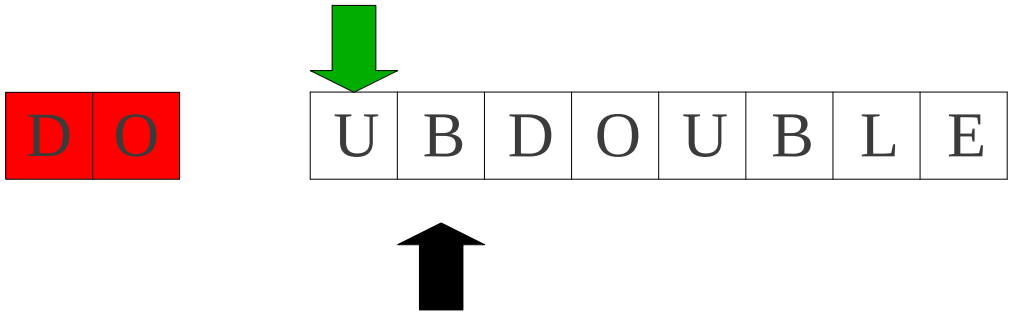


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

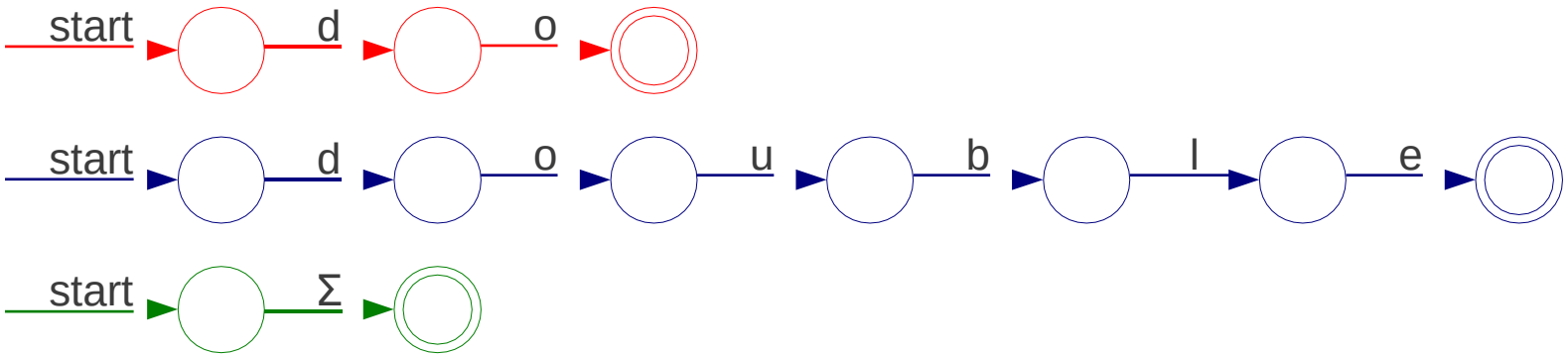


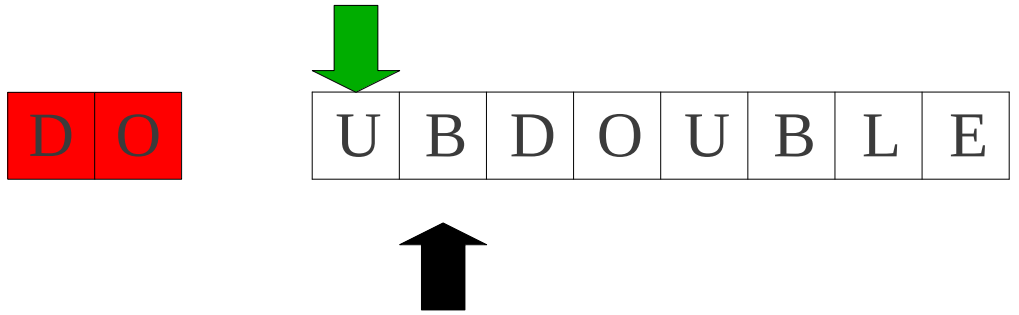


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

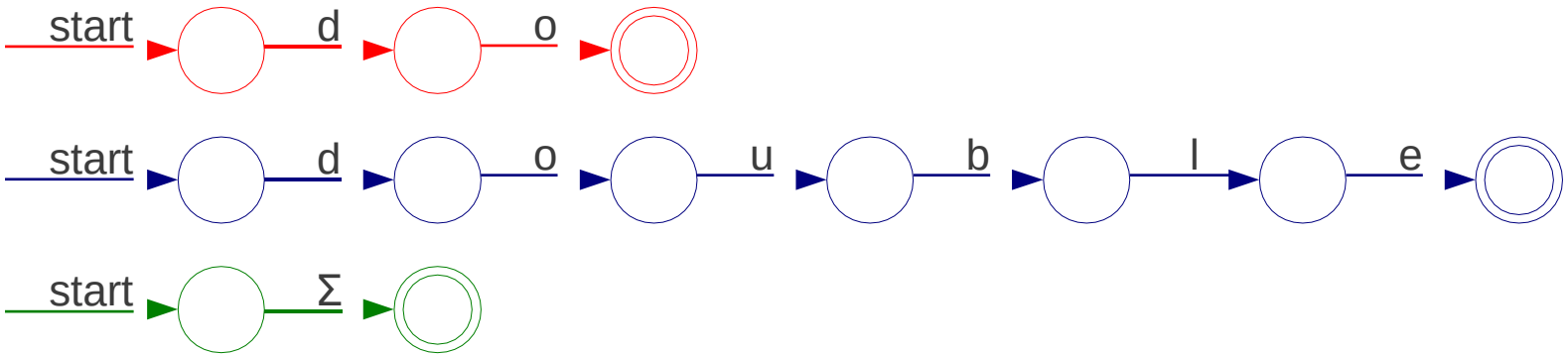


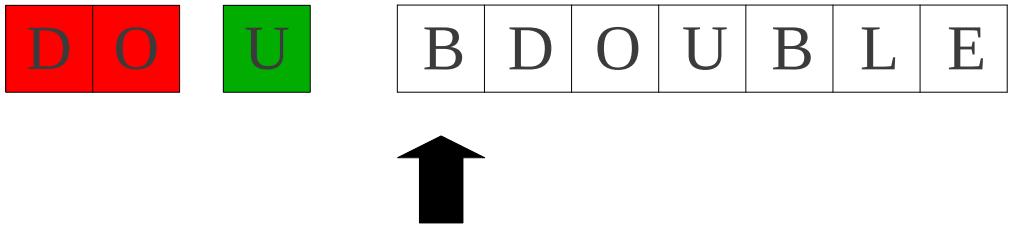


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

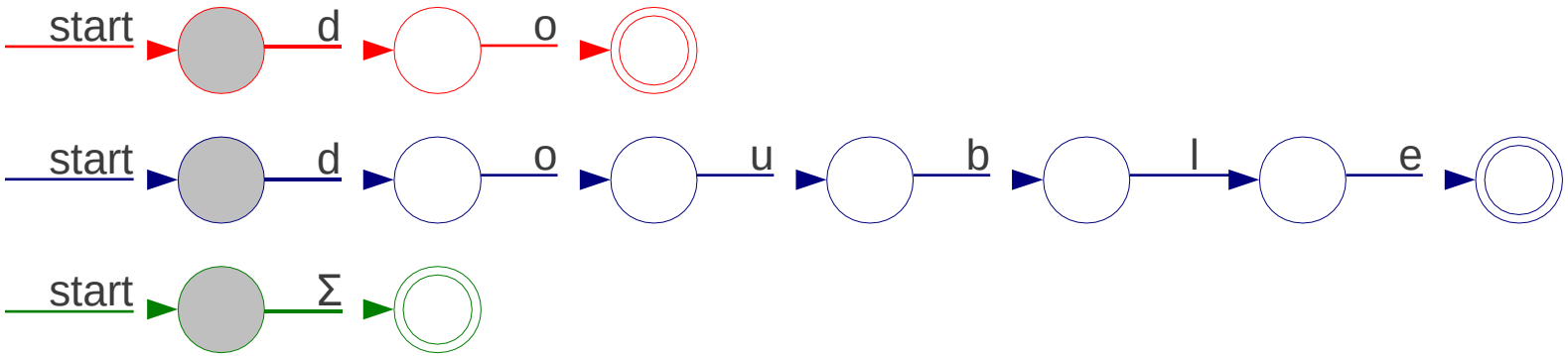


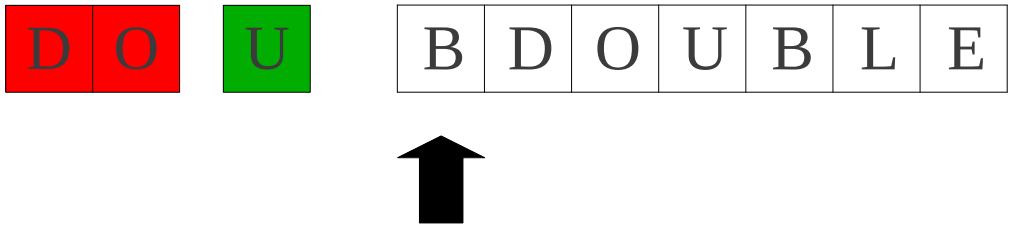


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

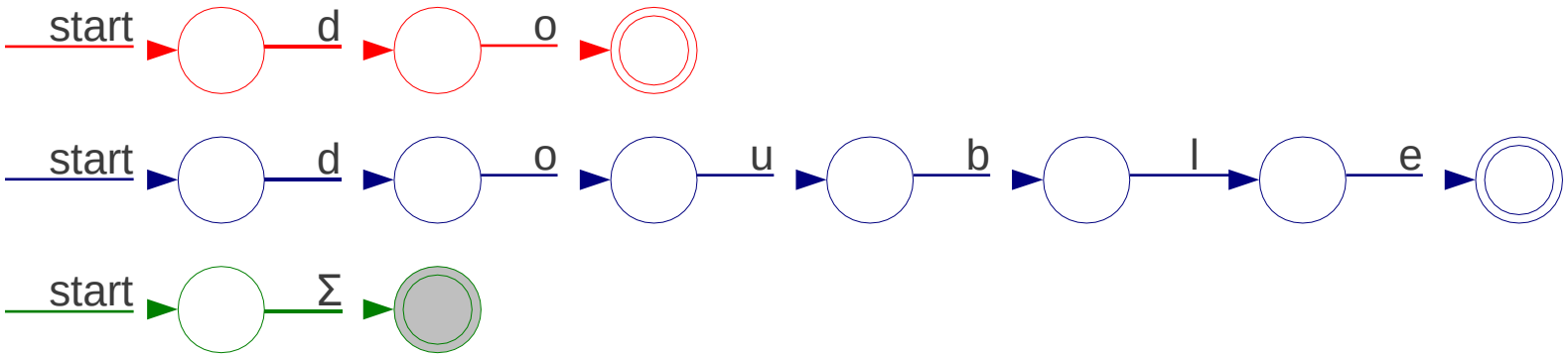


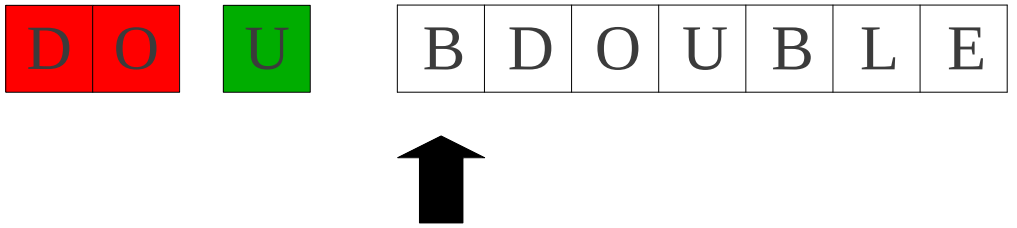


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

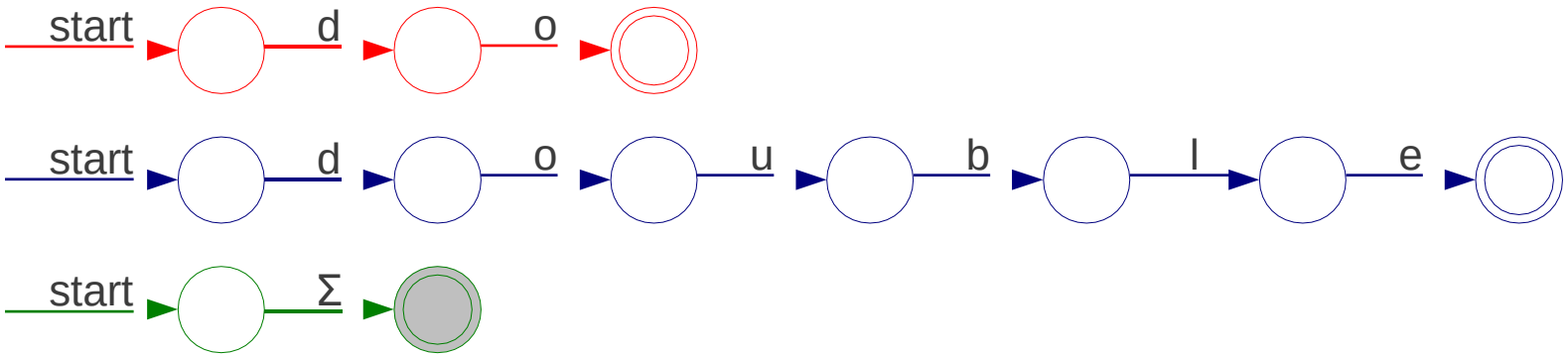


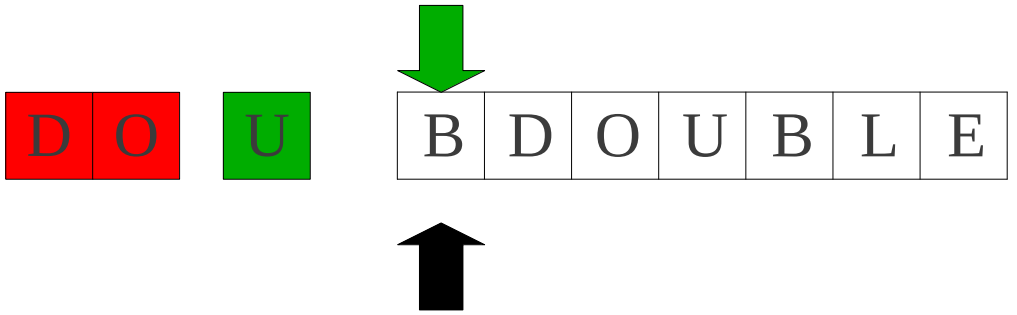


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

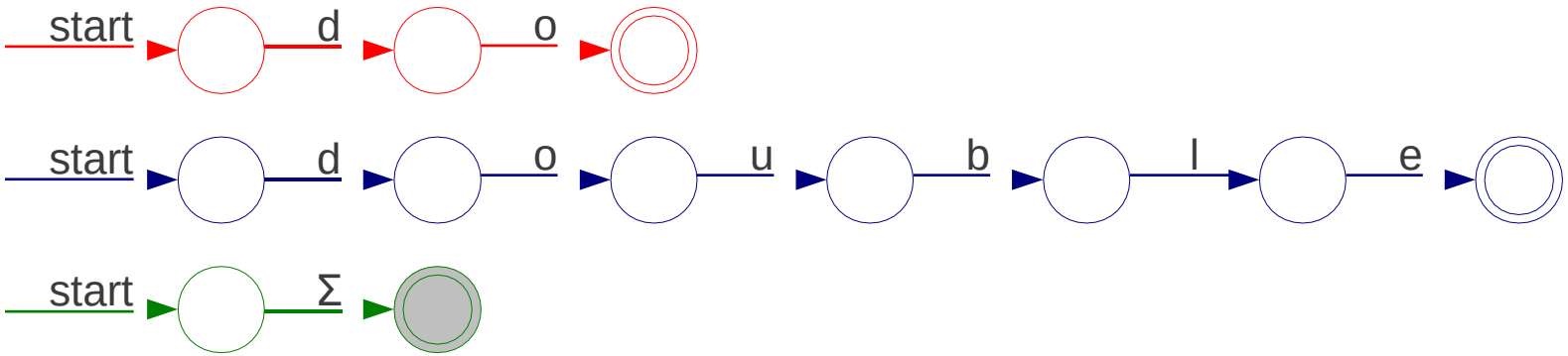


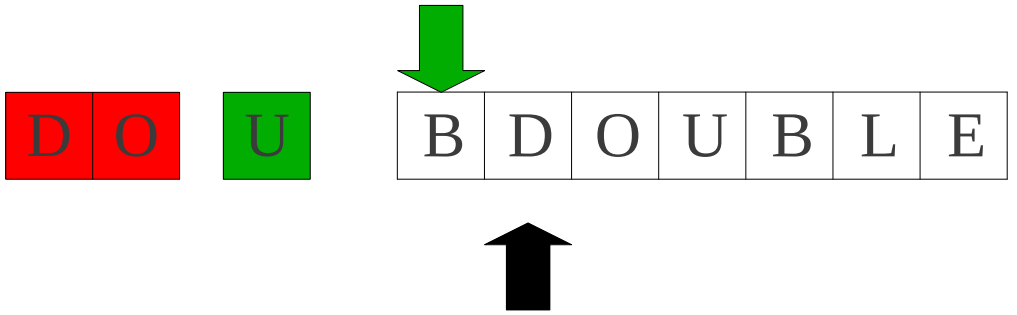


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

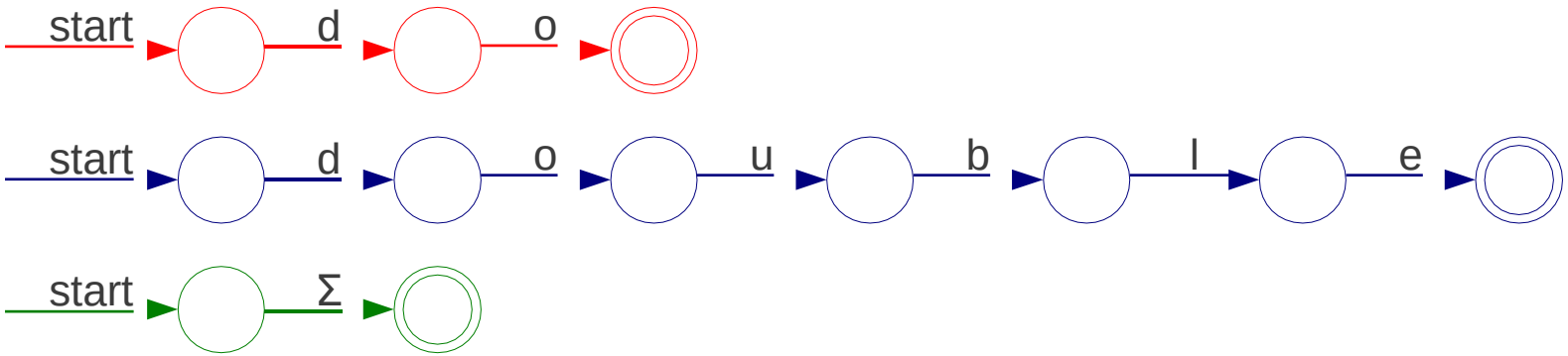


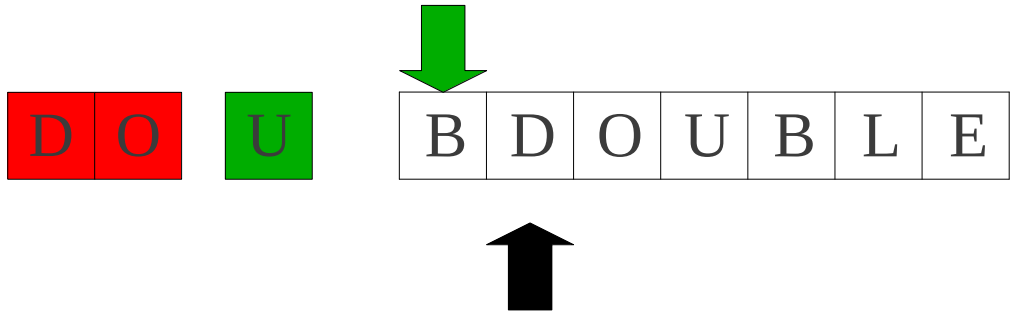


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

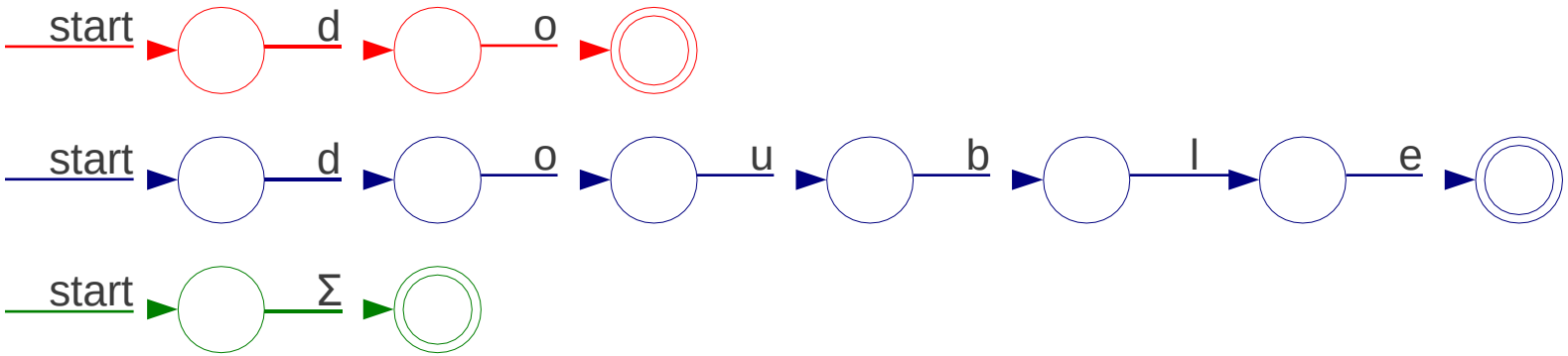


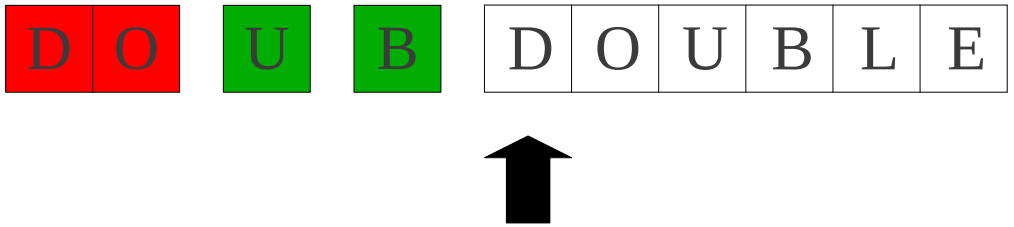


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

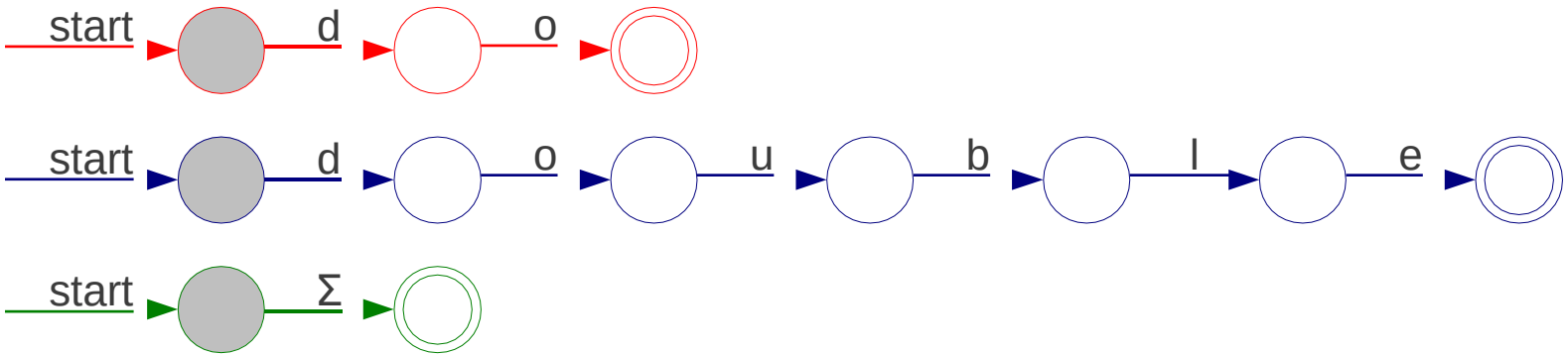


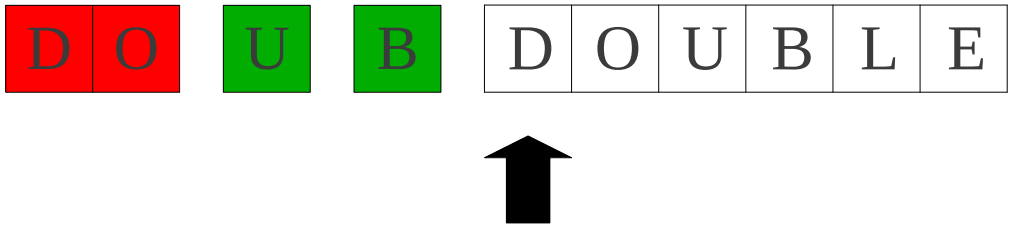


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

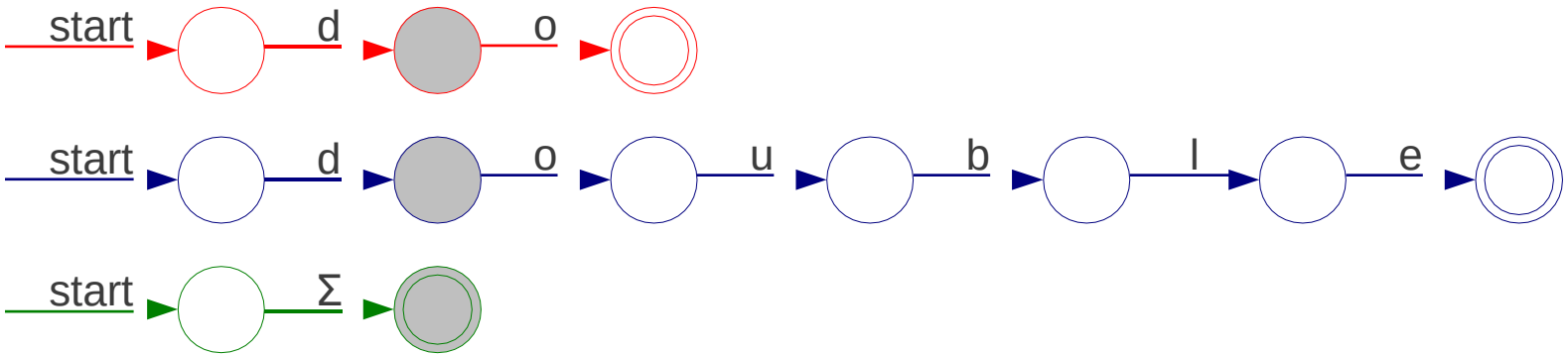


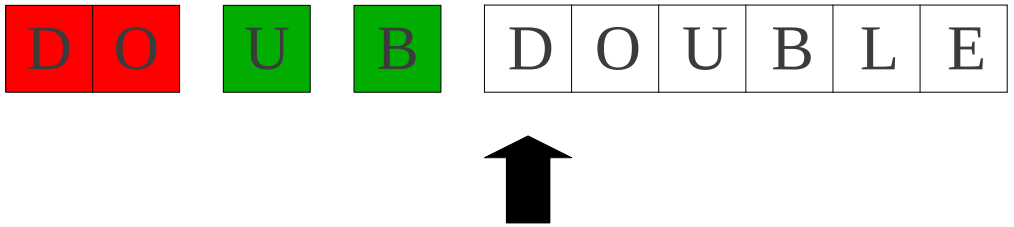


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

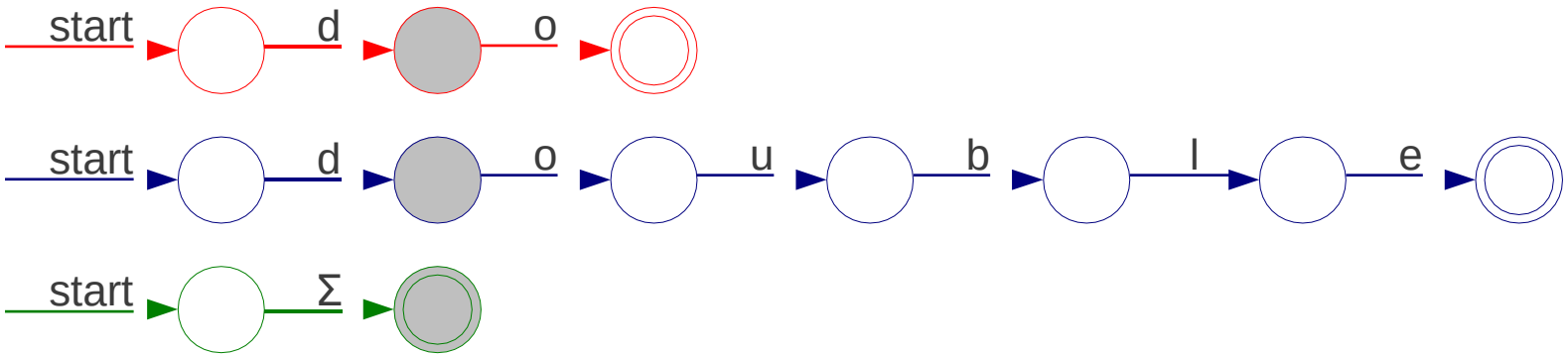


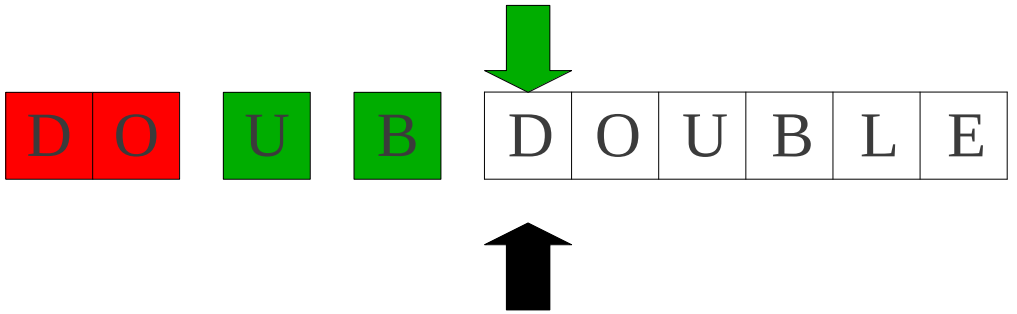


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

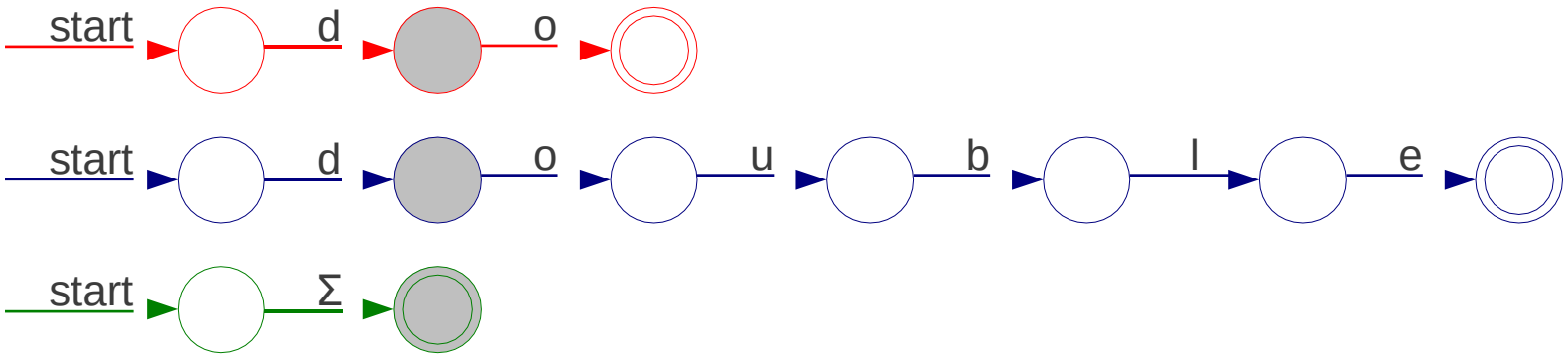


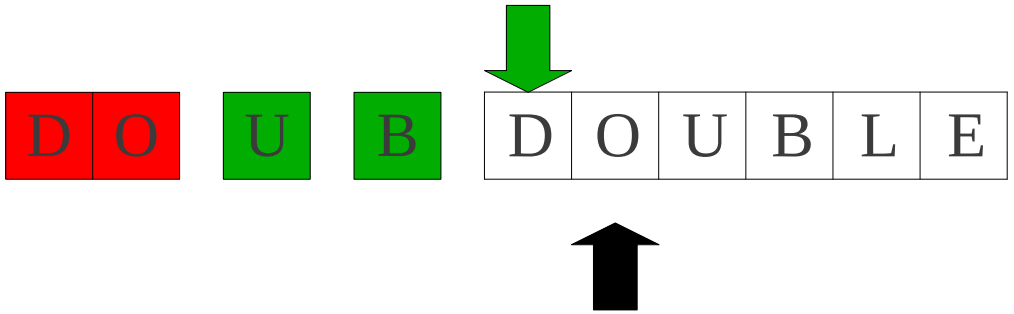


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

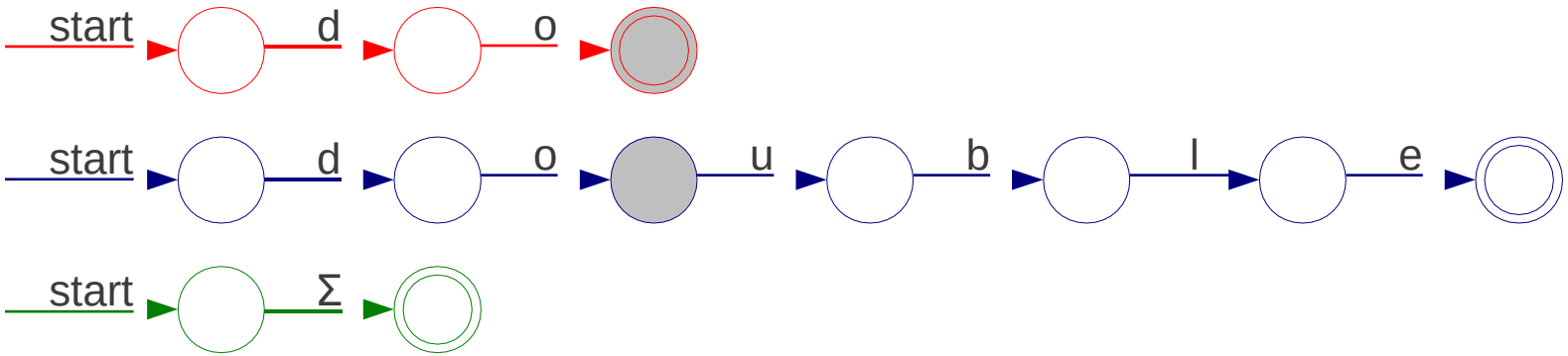


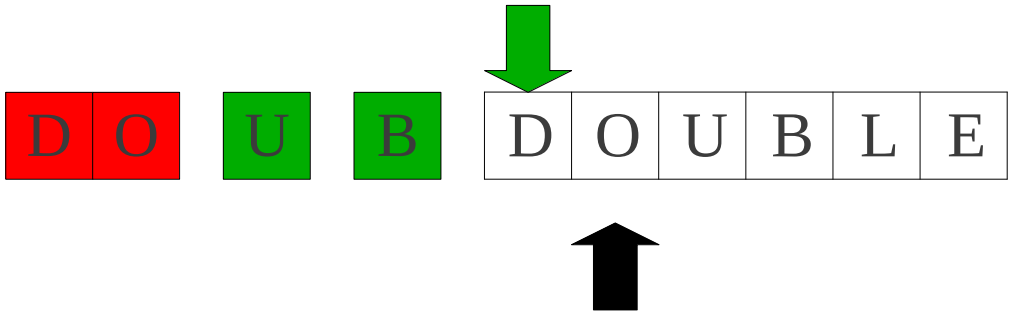


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

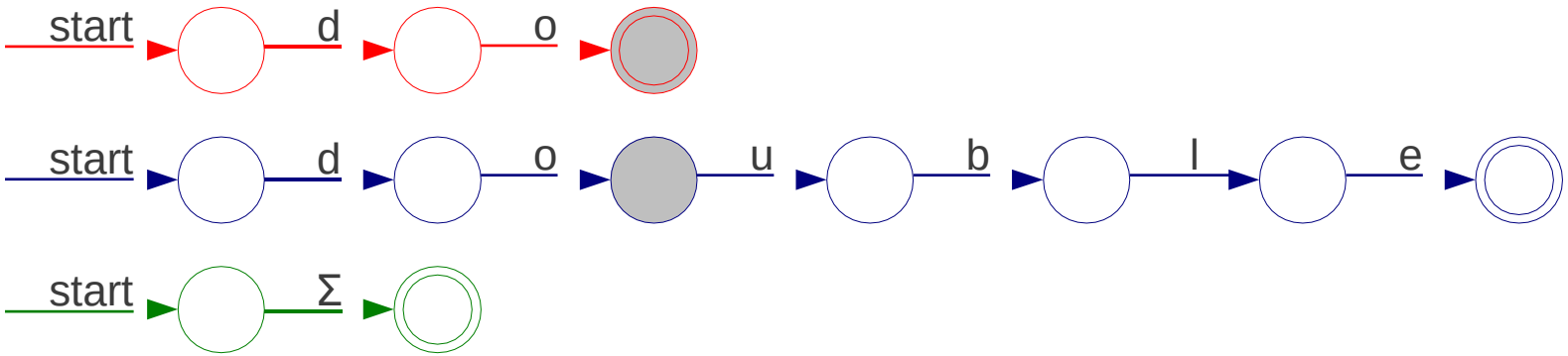


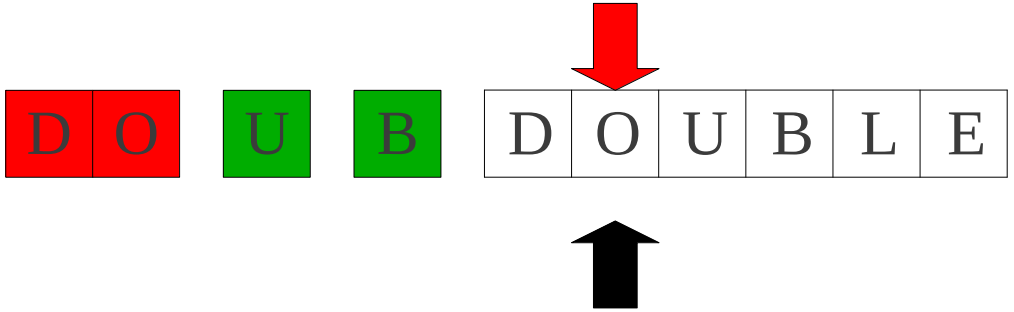


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

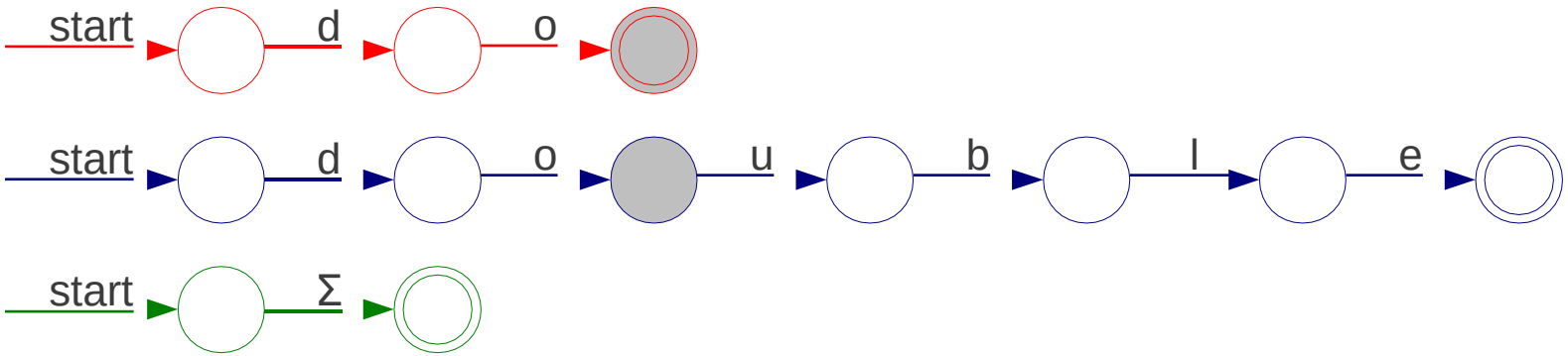


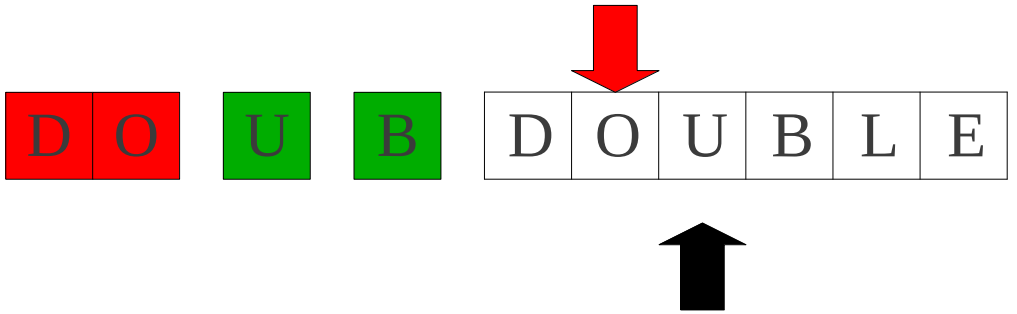


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

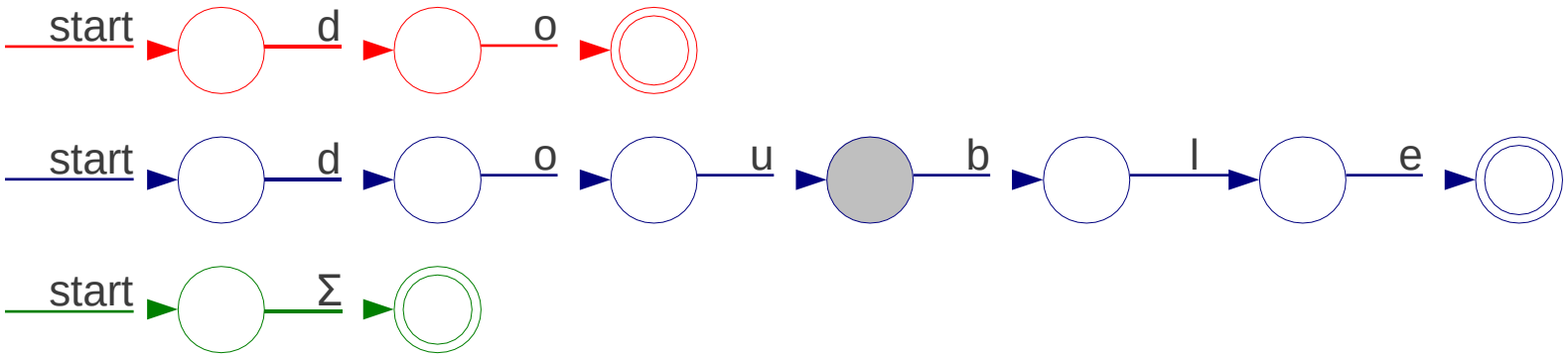


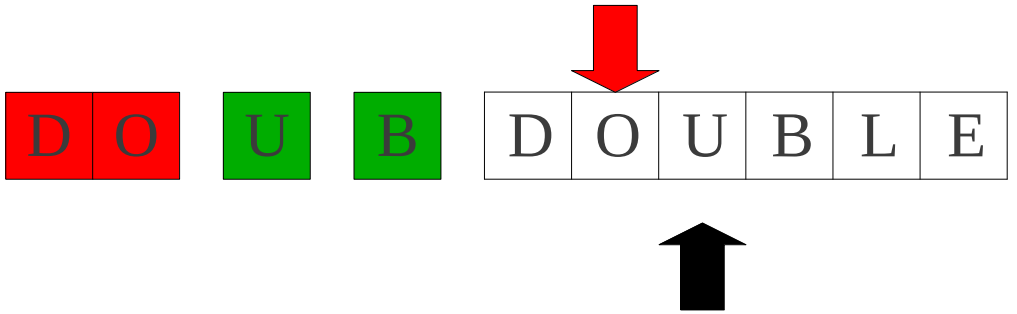


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

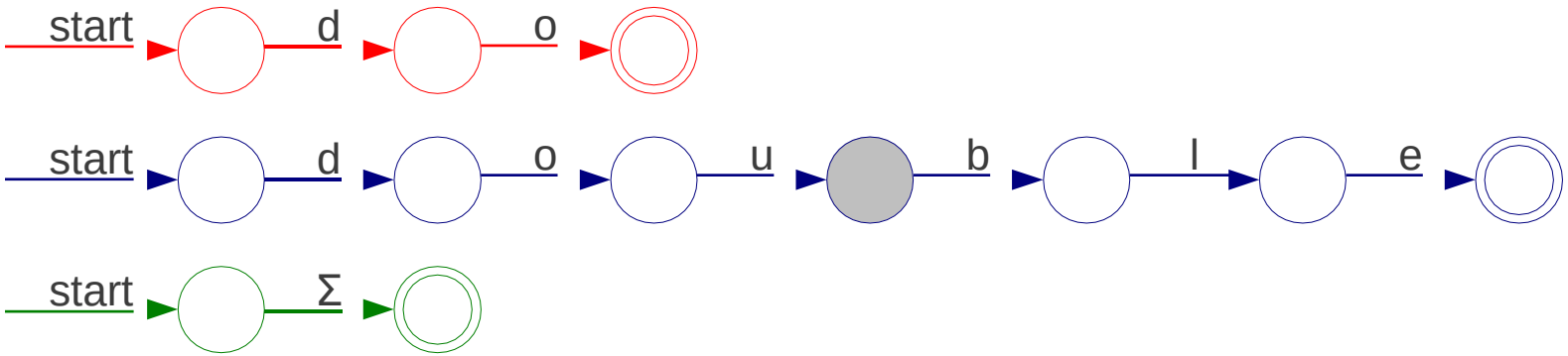


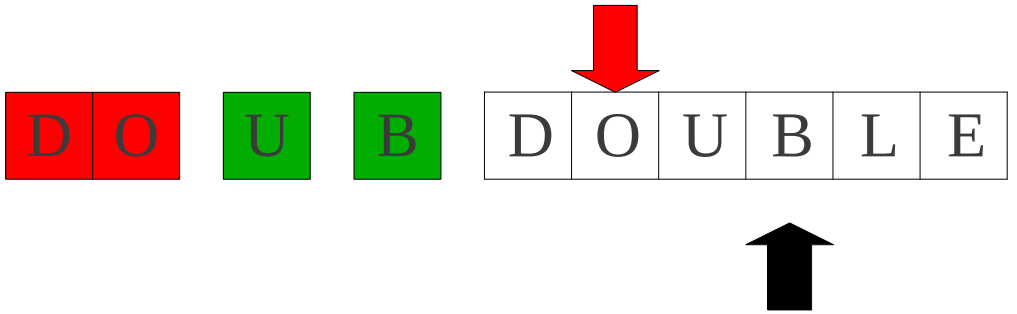


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

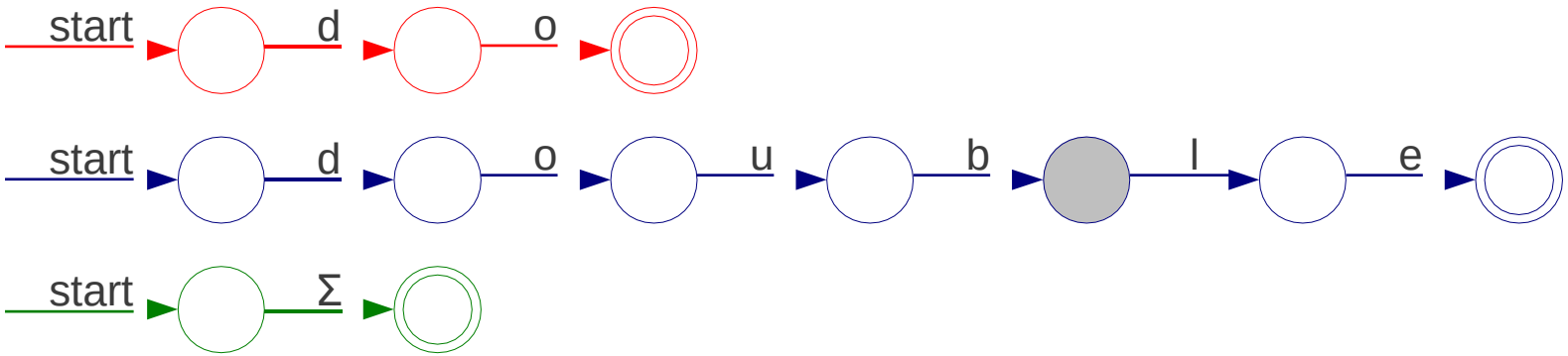


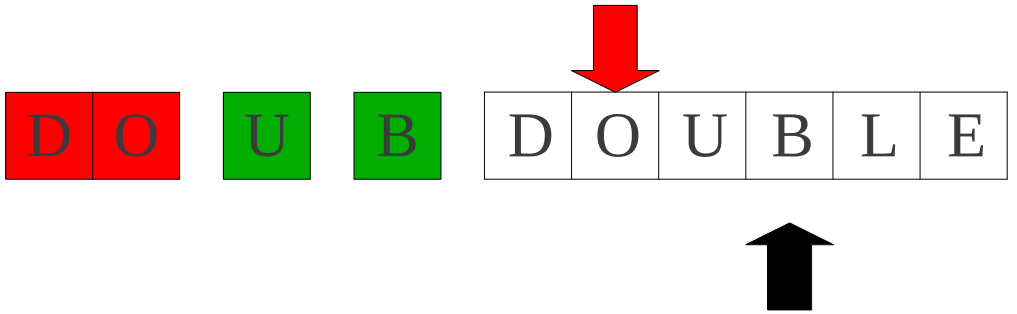


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

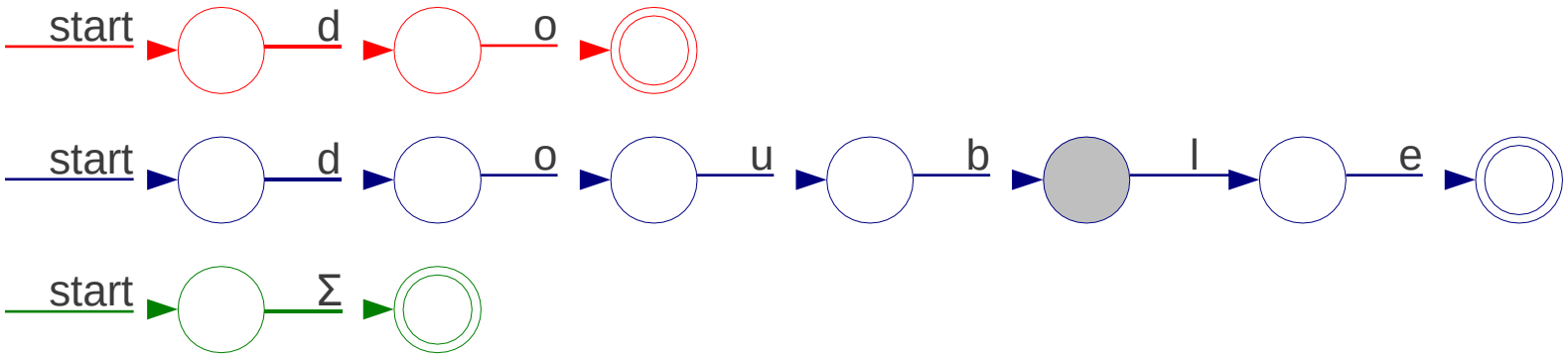


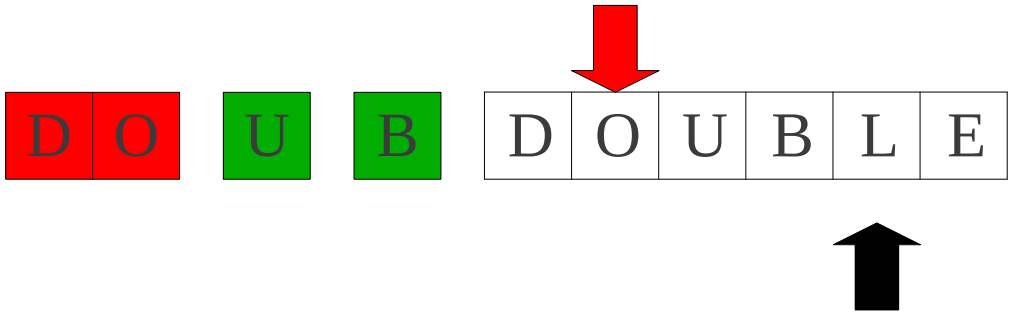


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

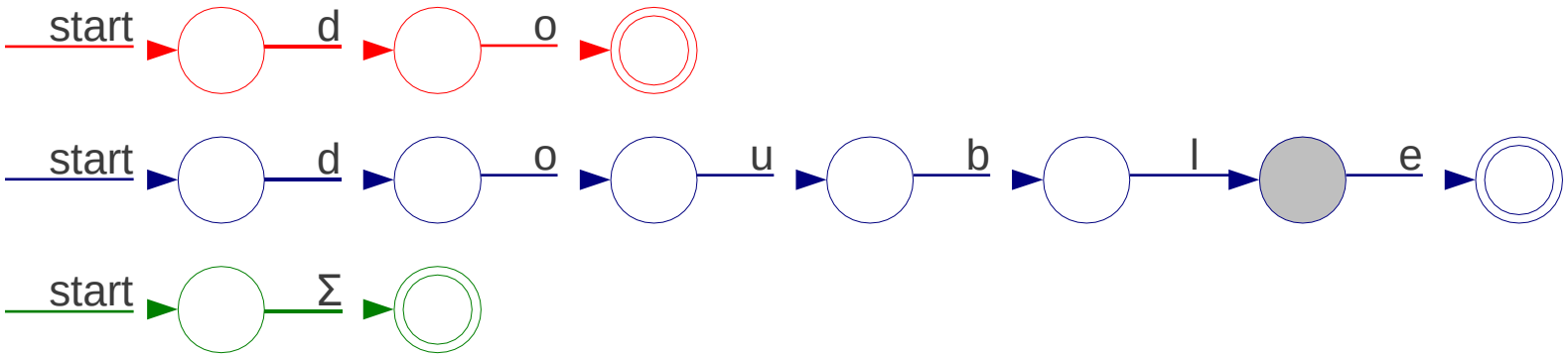


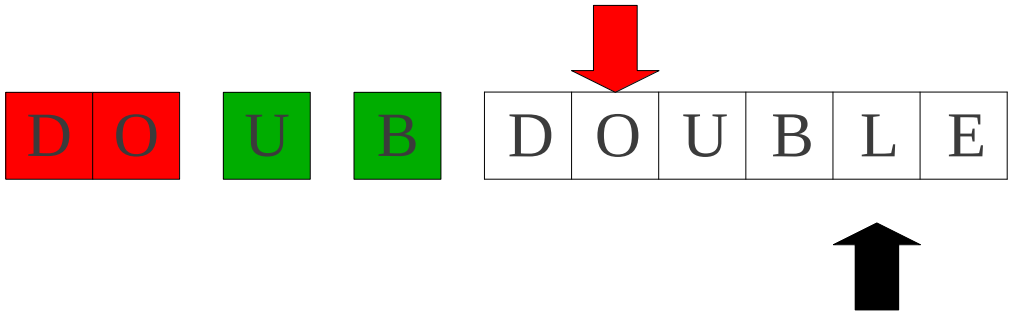


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

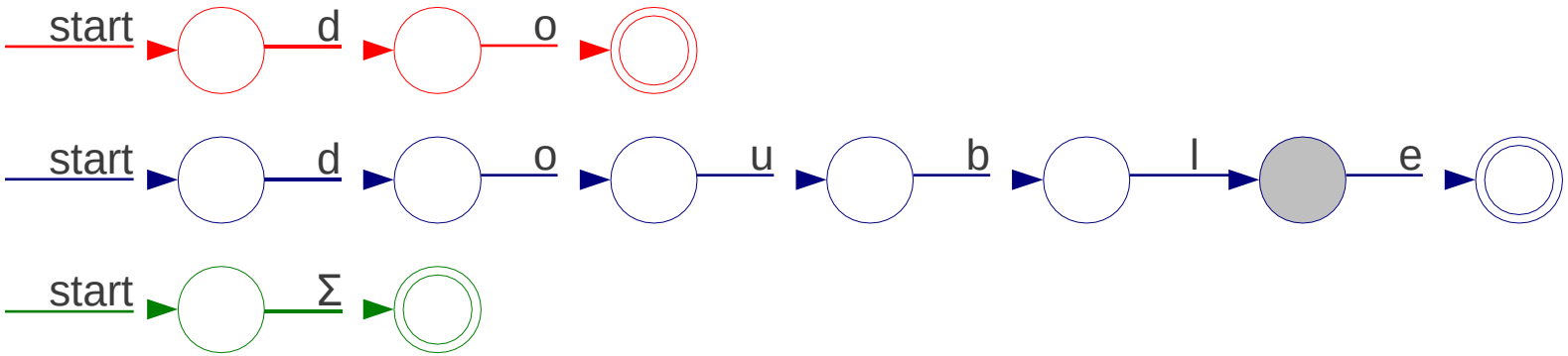


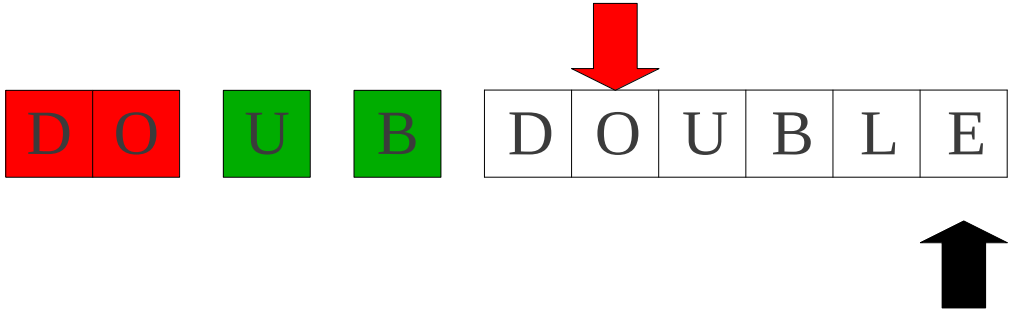


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

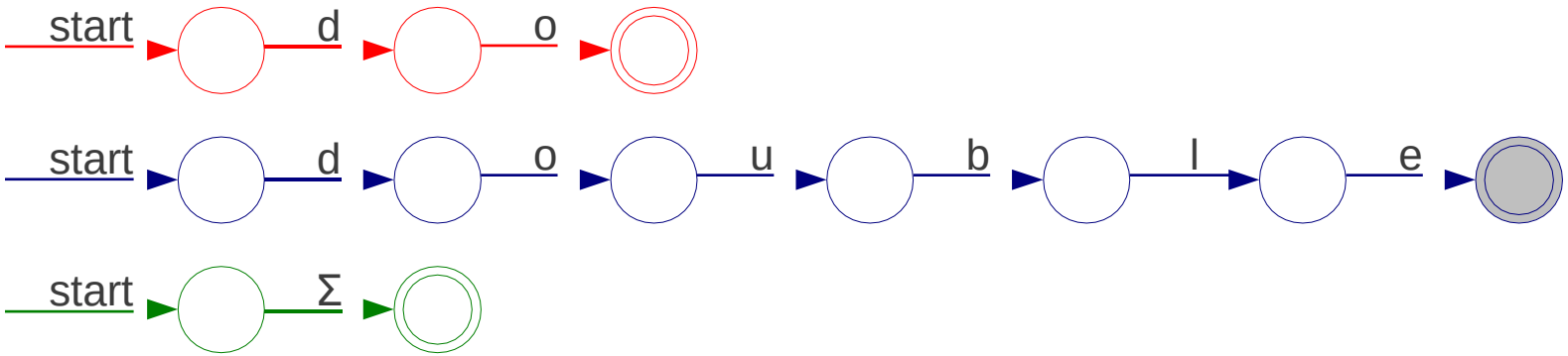


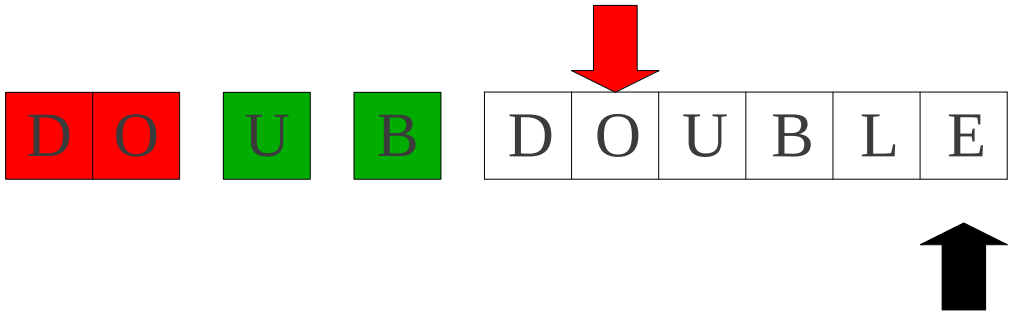


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

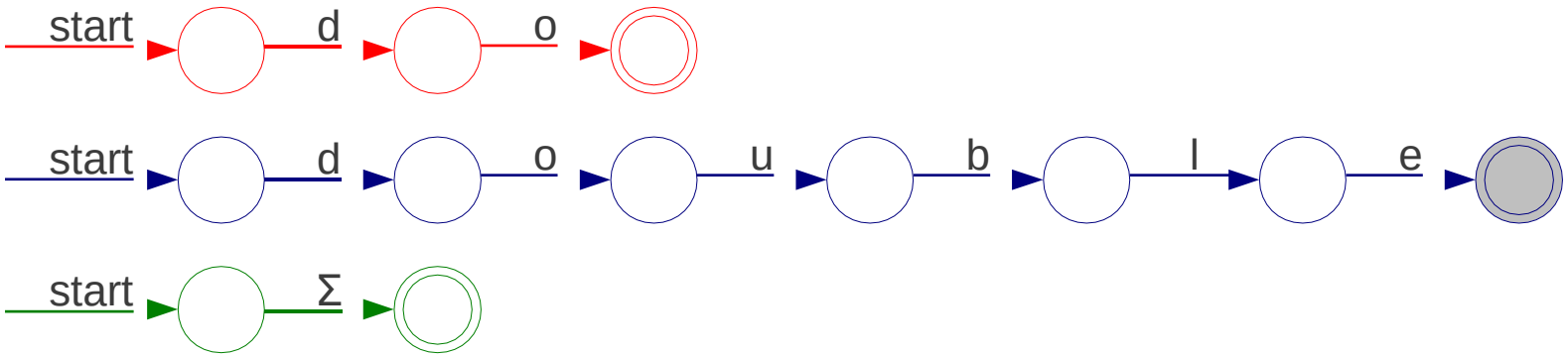


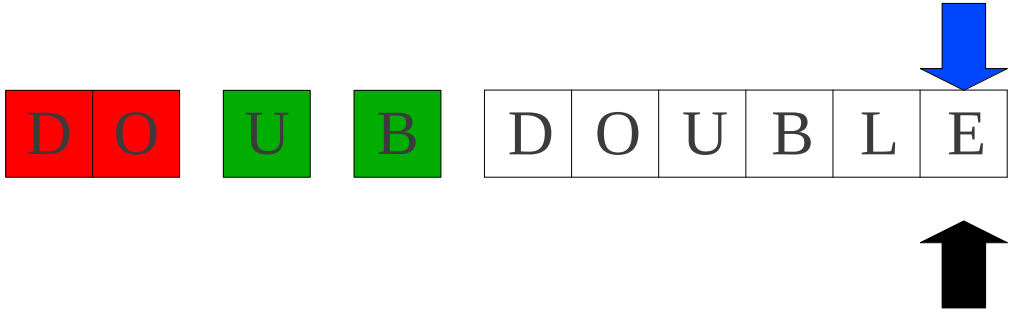


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

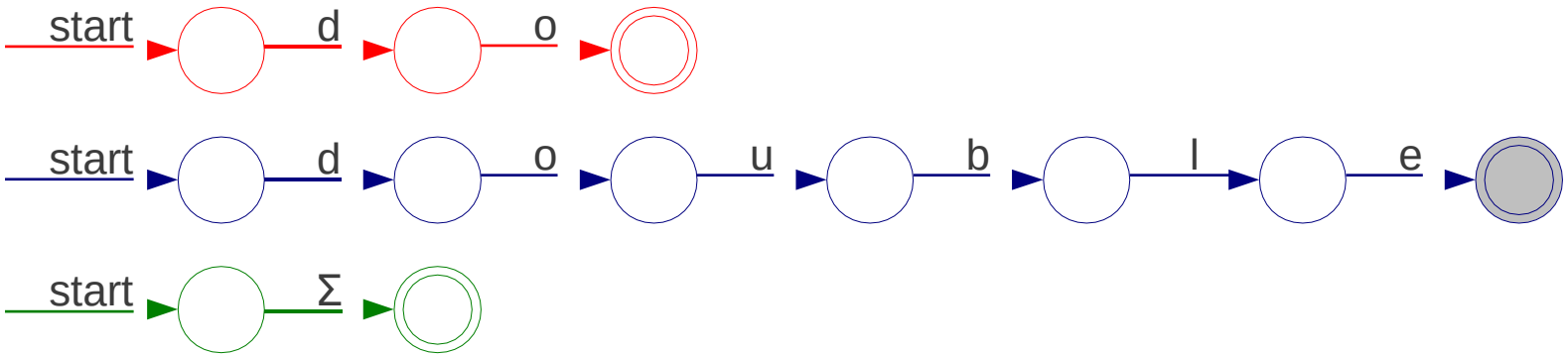


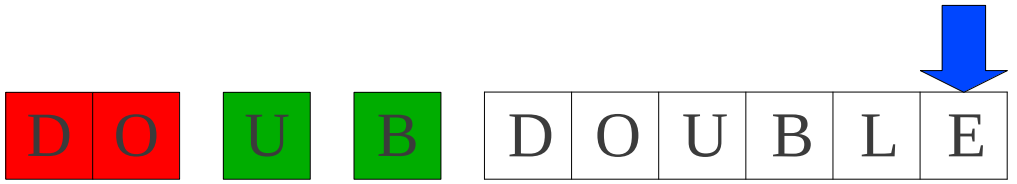


GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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

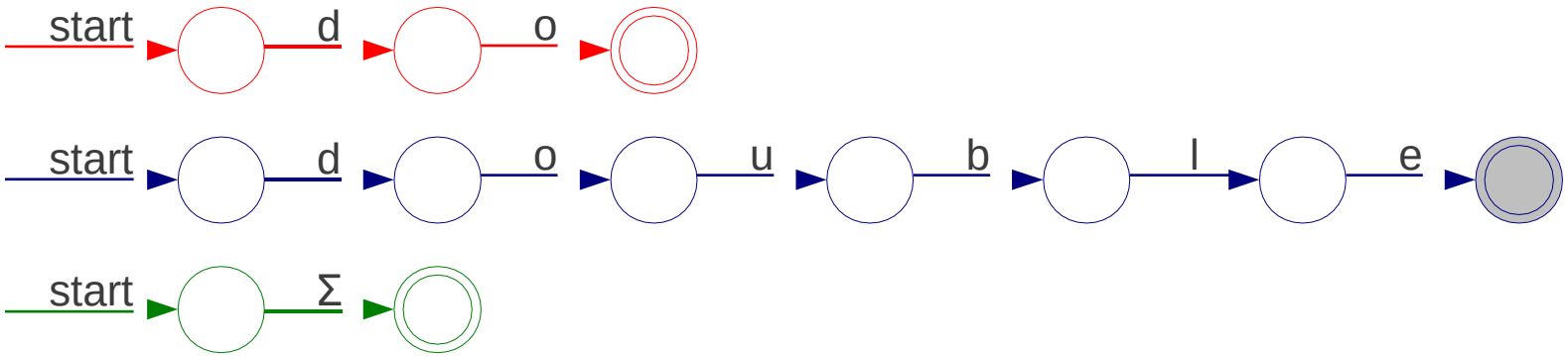




GEORGETOWN UNIVERSITY

Implementing Maximal Munch

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



GEORGETOWN UNIVERSITY

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.

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?

Challenges in Scanning

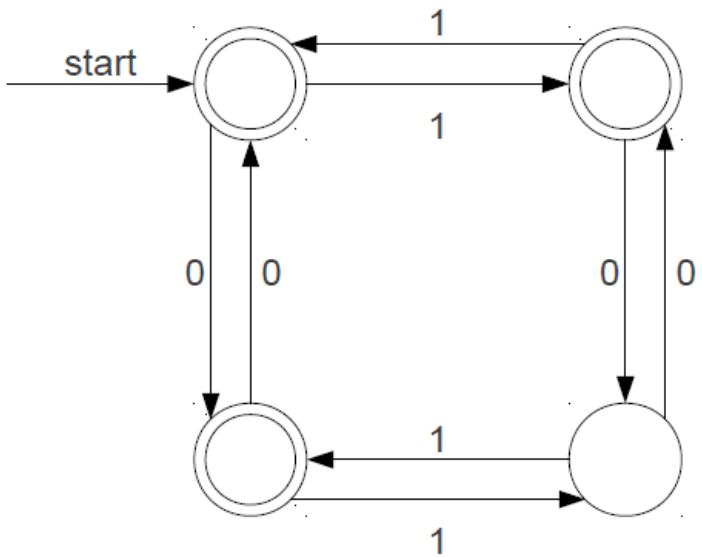
- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns
- efficiently?

DFAs

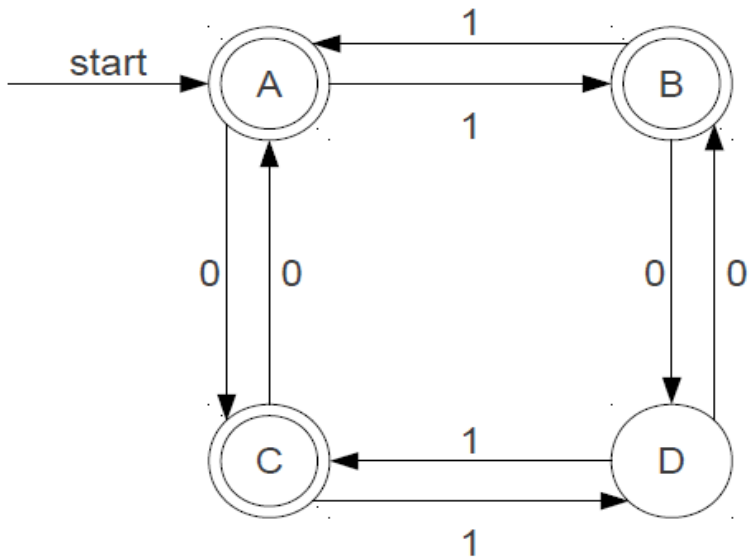
- The automata we've seen so far have all been NFAs.
- A **DFA** is like an NFA, but with tighter restrictions:
  - Every state must have **exactly one** transition defined for every letter.
  - ε-moves are not allowed.

A Sample DFA

A Sample DFA

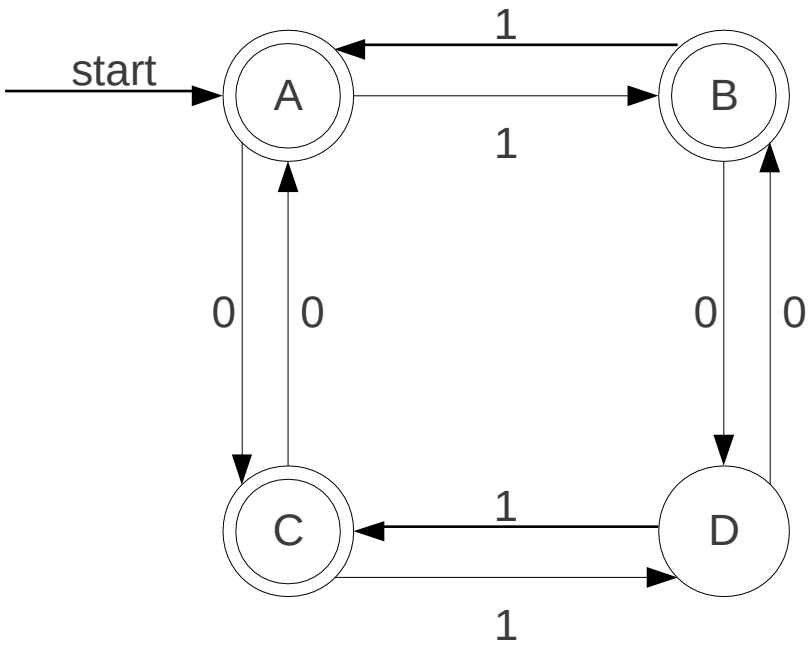


A Sample DFA



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

A Sample DFA



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

Code for DFAs

```
int kTransitionTable[kNumStates][kNumSymbols] = {
     {0, 0, 1, 3, 7, 1, …},
      …
};
bool kAcceptTable[kNumStates] = {
    false,
    true,
    true,
    …
};
bool simulateDFA(string input) {
    int state = 0;
    for (char ch: input)
        state = kTransitionTable[state][ch];
    return kAcceptTable[state];
}
```

Runs in time $O(m)$ on a string of length m.

Speeding up Matching

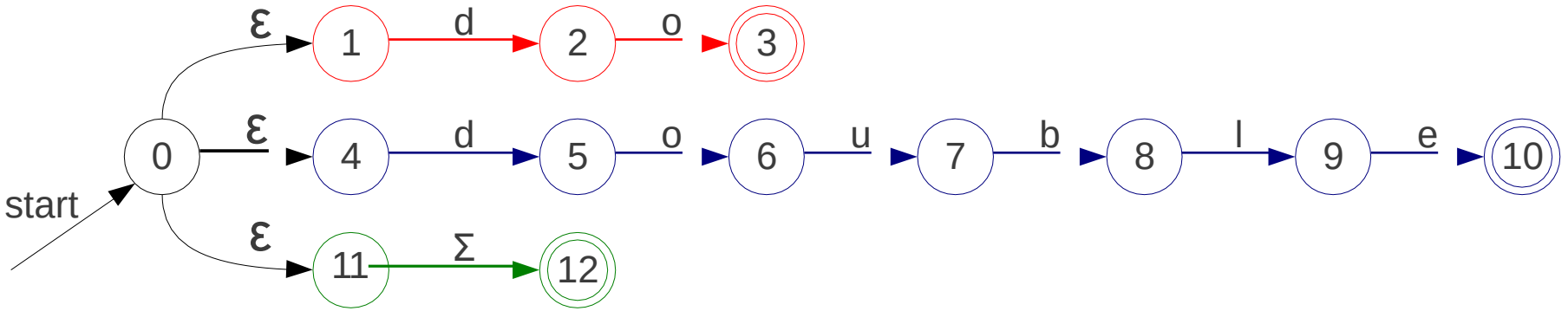
- In the worst-case, an NFA with n states takes time $O(mn^2)$ to match a string of length m. (Can you guess why ..?)
- DFAs, on the other hand, take only $O(m)$.
- There is another (beautiful!) algorithm to convert NFAs to DFAs.

Subset Construction

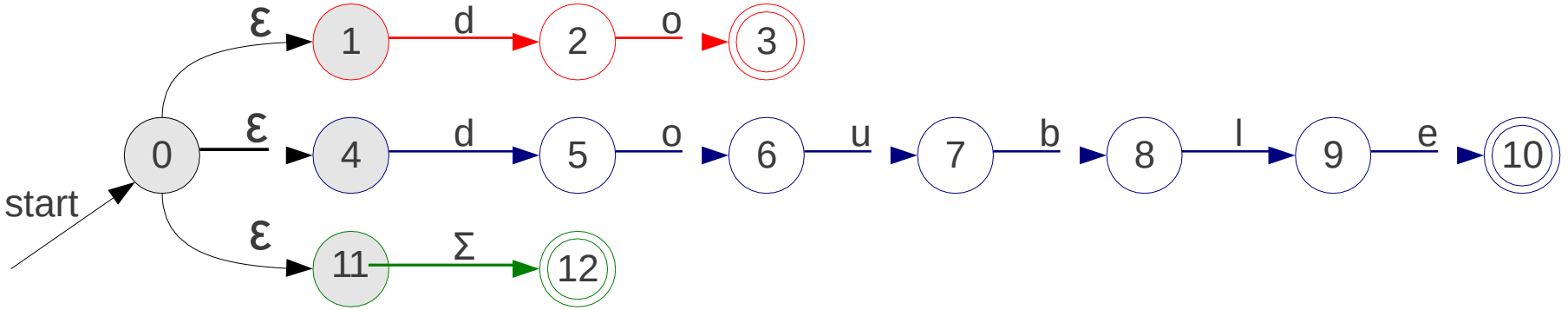
- NFAs can be in many states at once, while DFAs can only be in a single state at a time.
- Key idea: **Make the DFA simulate the NFA**.
- Have the states of the DFA correspond to the *sets of states* of the NFA.
- Transitions between states of DFA correspond to transitions between *sets of states* in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

From NFA to DFA

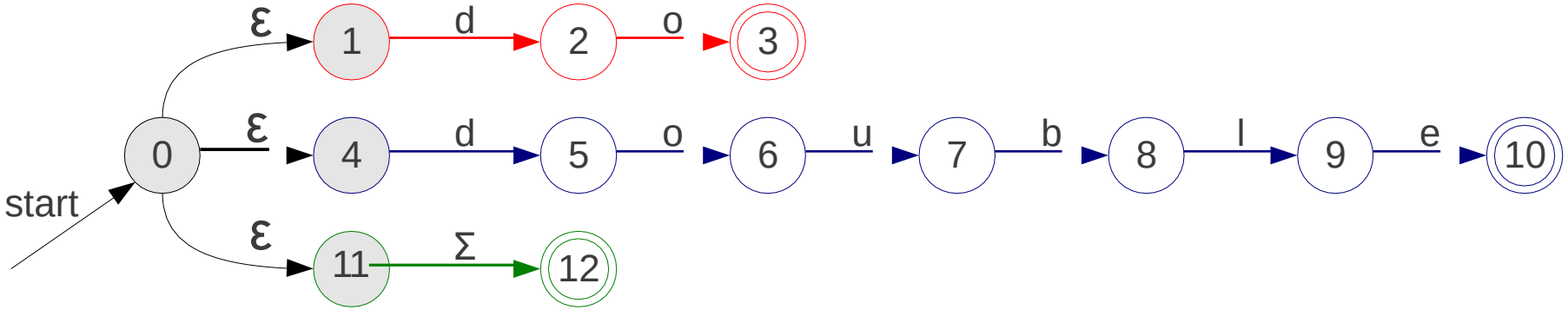
From NFA to DFA

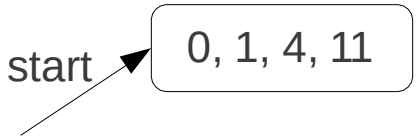


From NFA to DFA

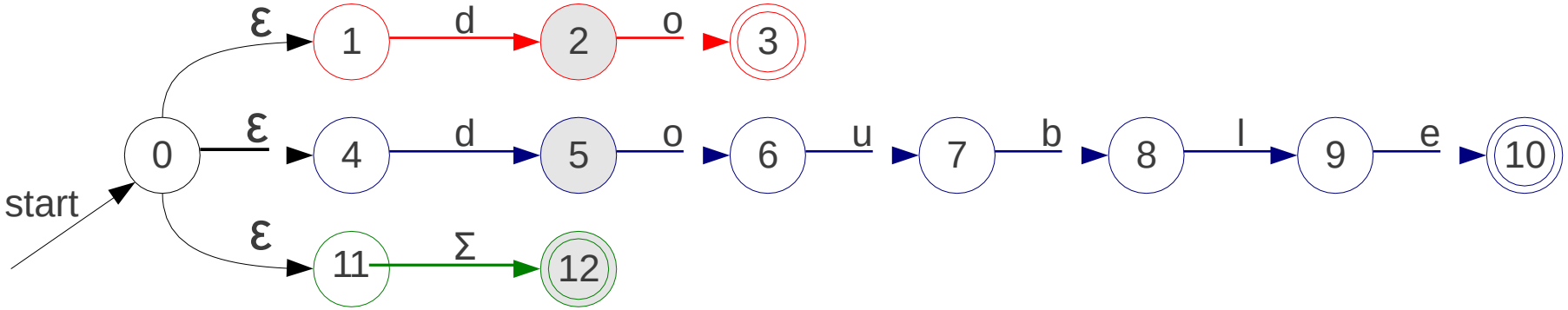


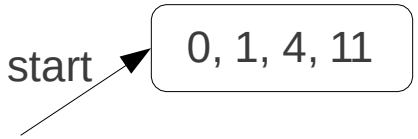
From NFA to DFA



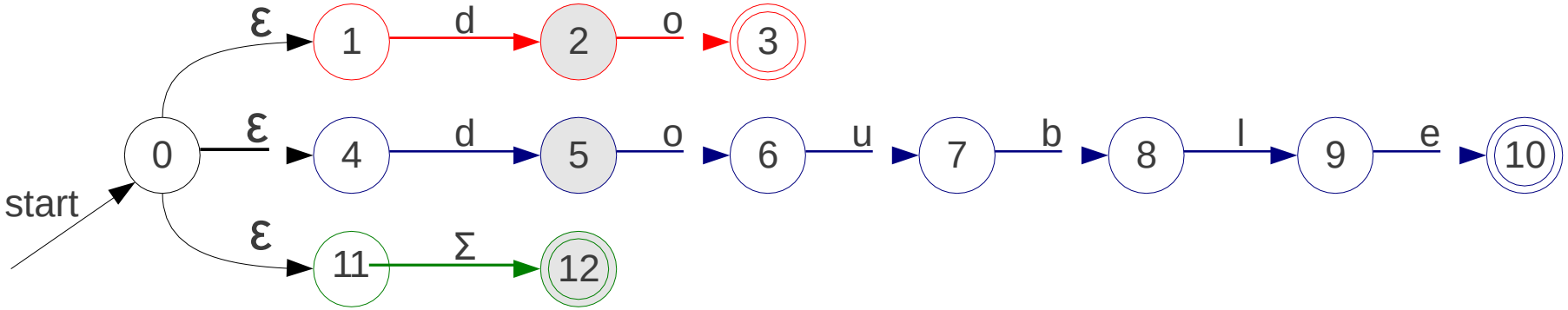


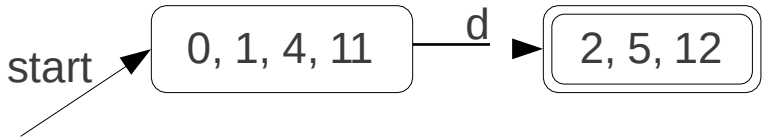
From NFA to DFA



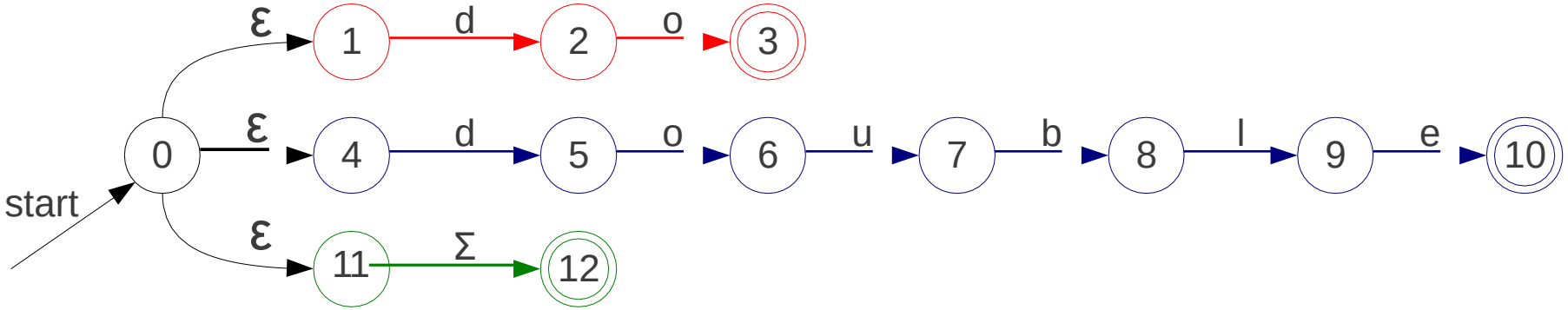


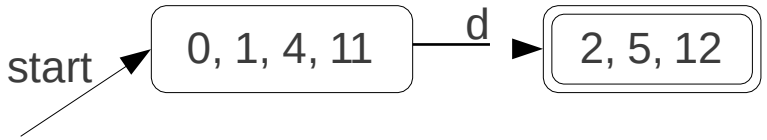
From NFA to DFA



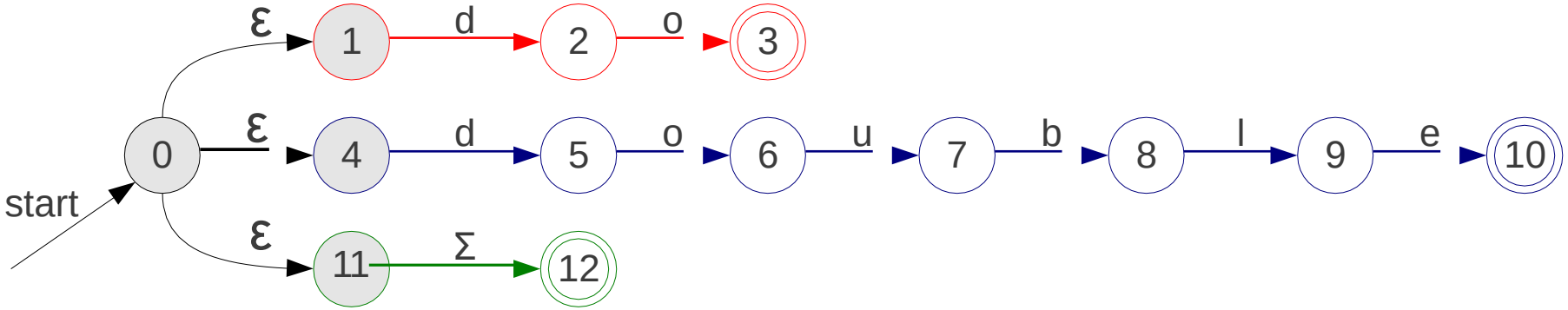


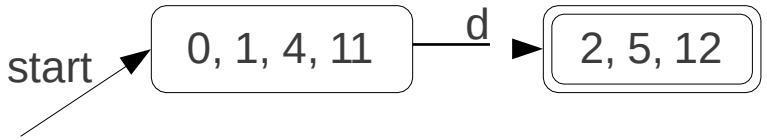
From NFA to DFA



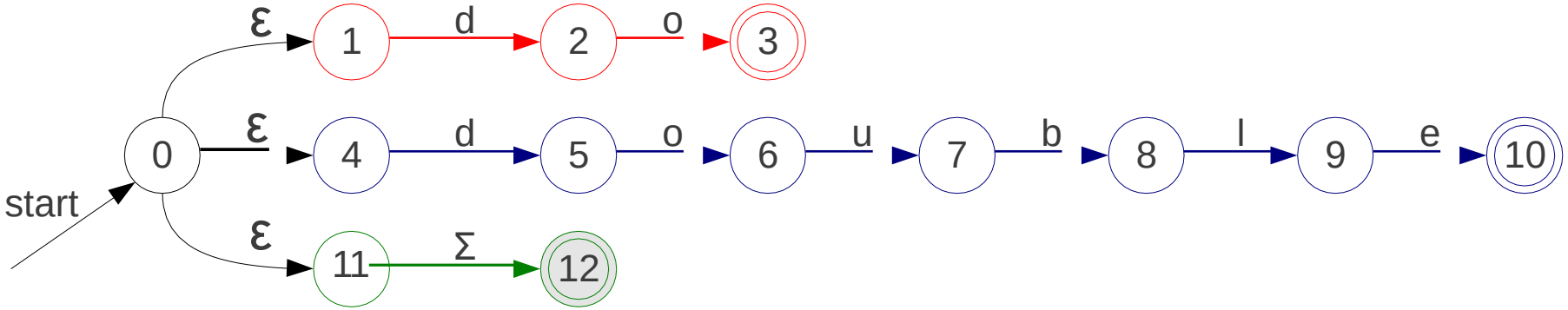


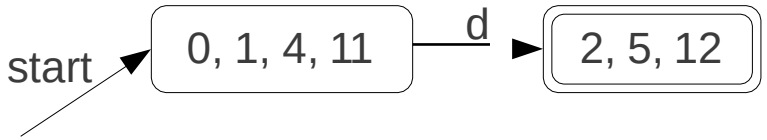
From NFA to DFA



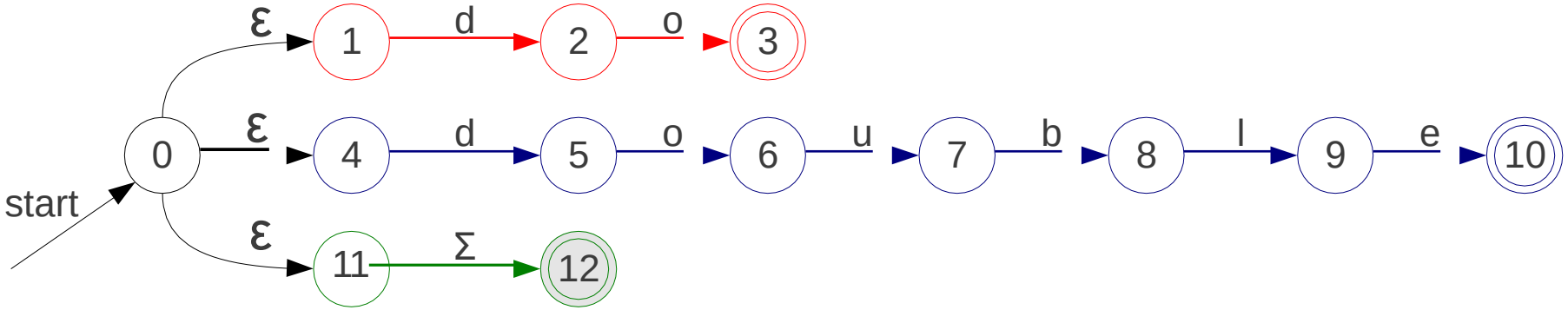


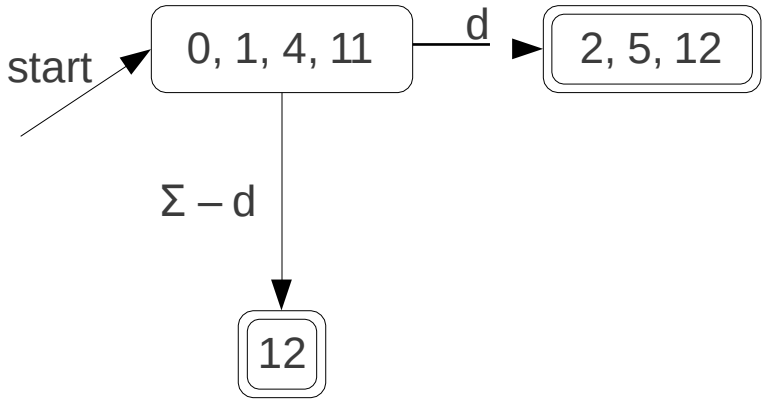
From NFA to DFA





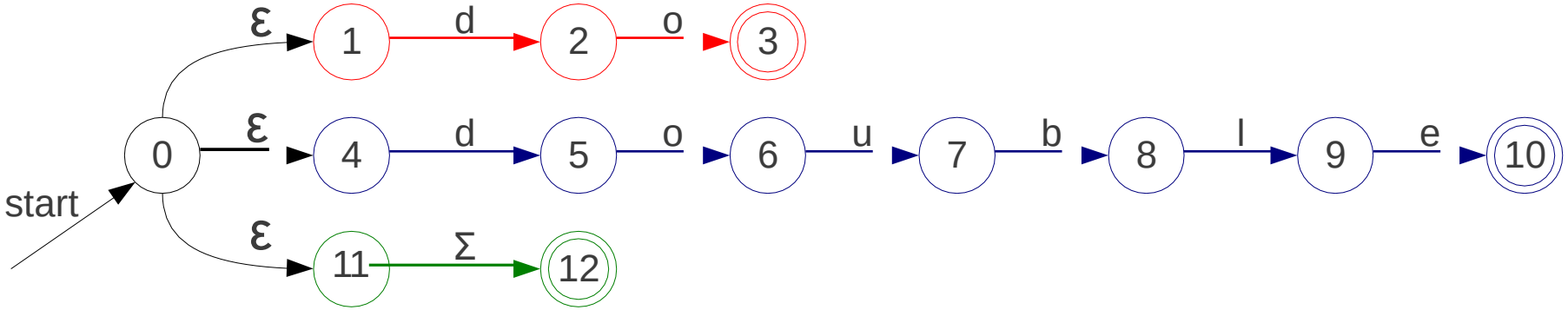
From NFA to DFA

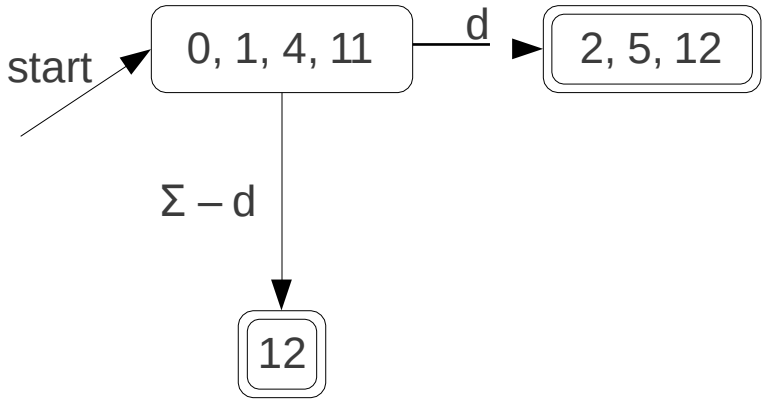




GEORGETOWN UNIVERSITY

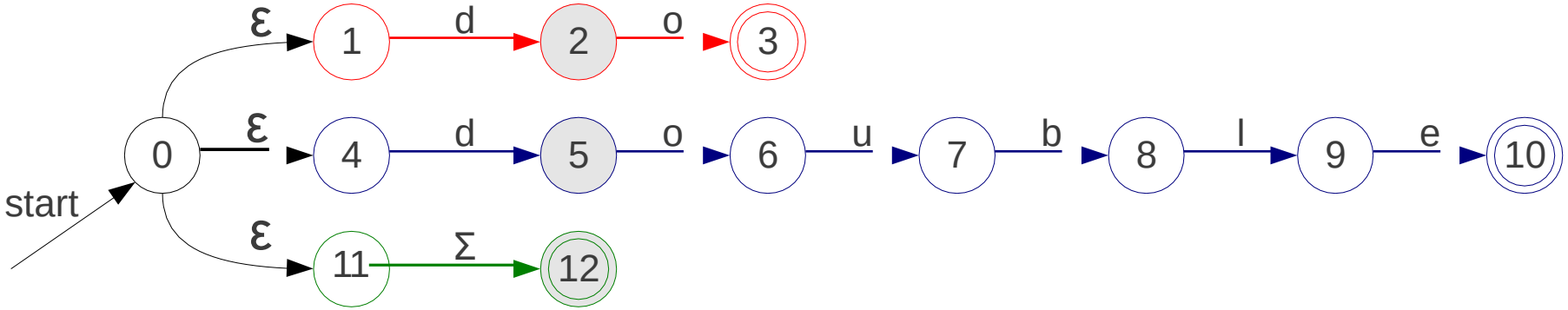
From NFA to DFA

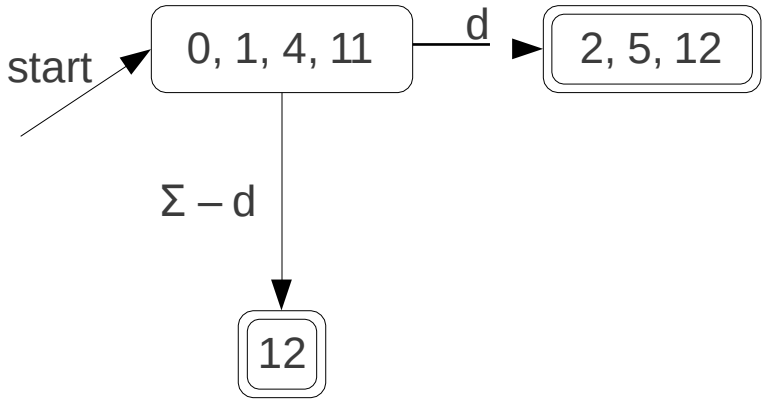




GEORGETOWN UNIVERSITY

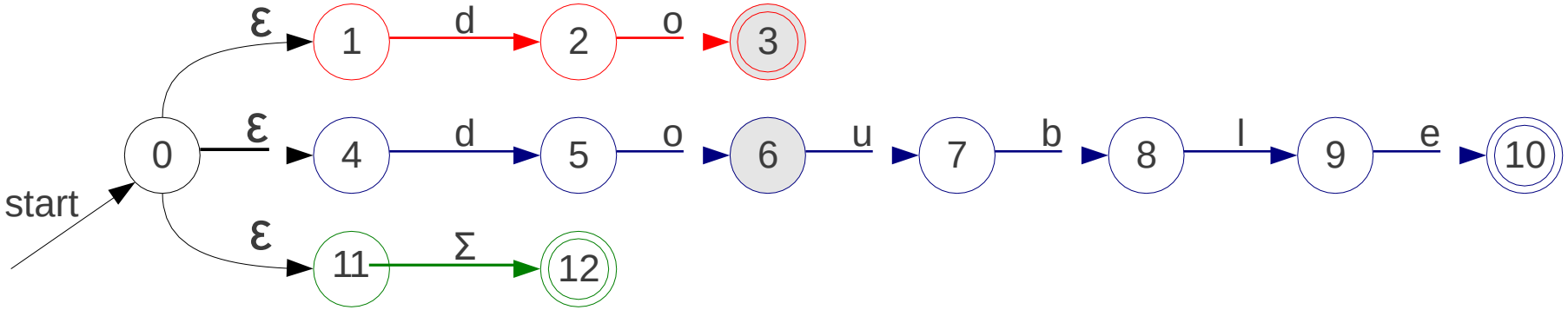
From NFA to DFA

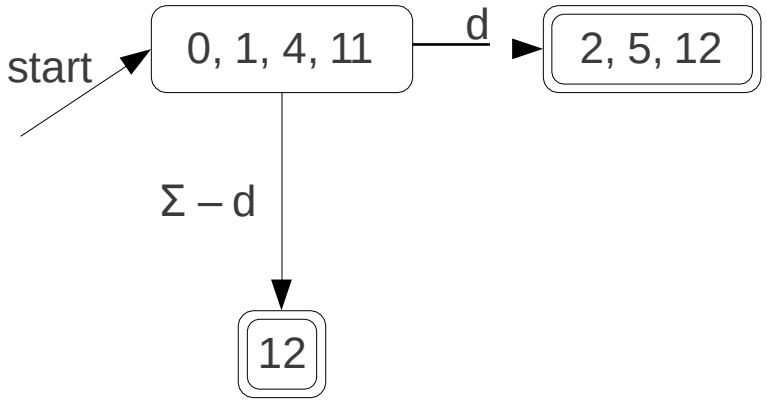




GEORGETOWN UNIVERSITY

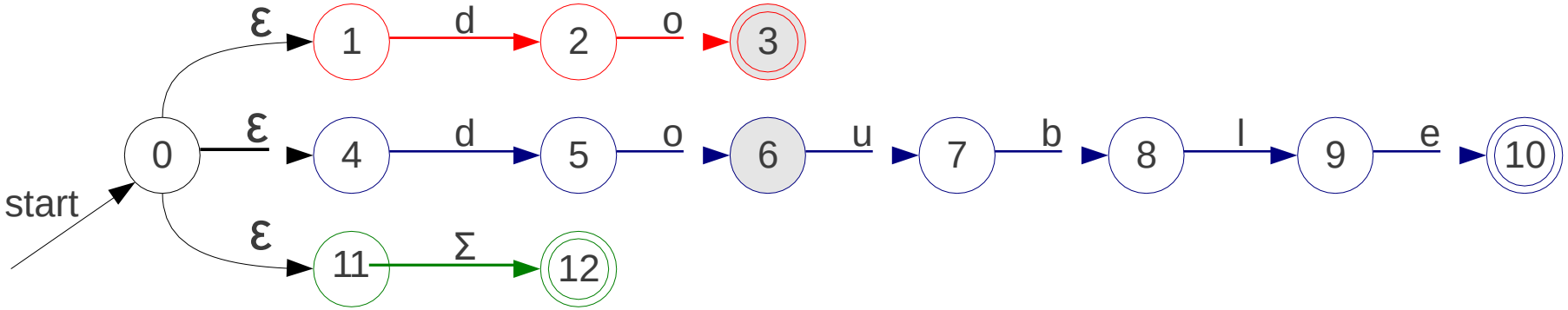
From NFA to DFA

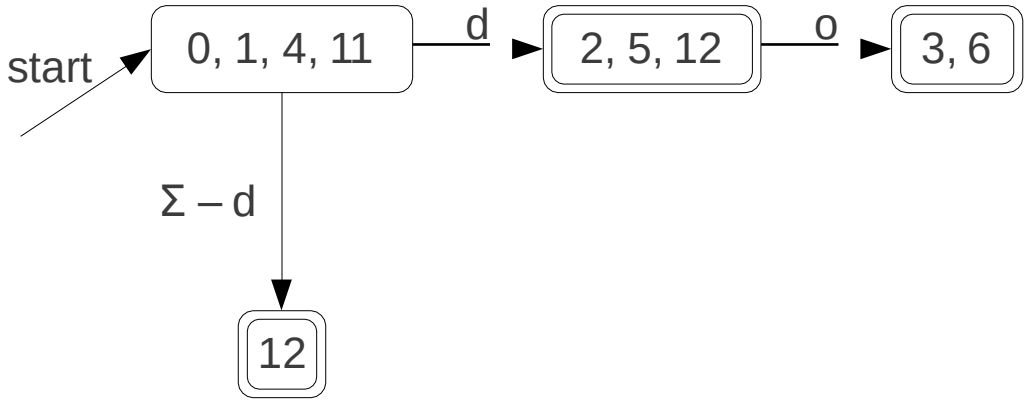




GEORGETOWN UNIVERSITY

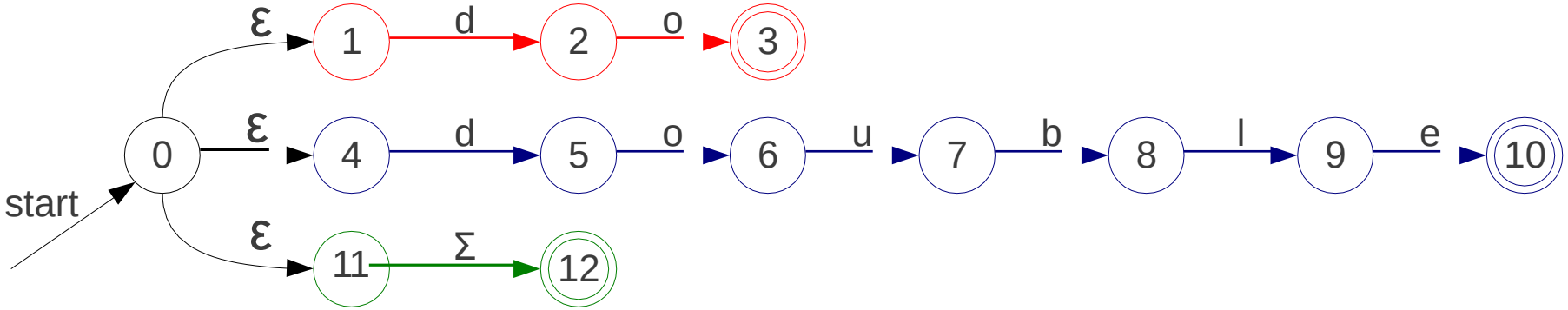
From NFA to DFA

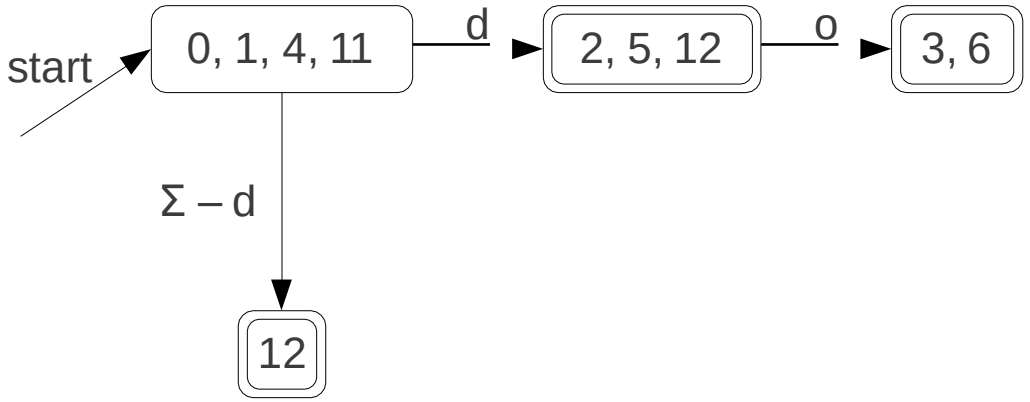




GEORGETOWN UNIVERSITY

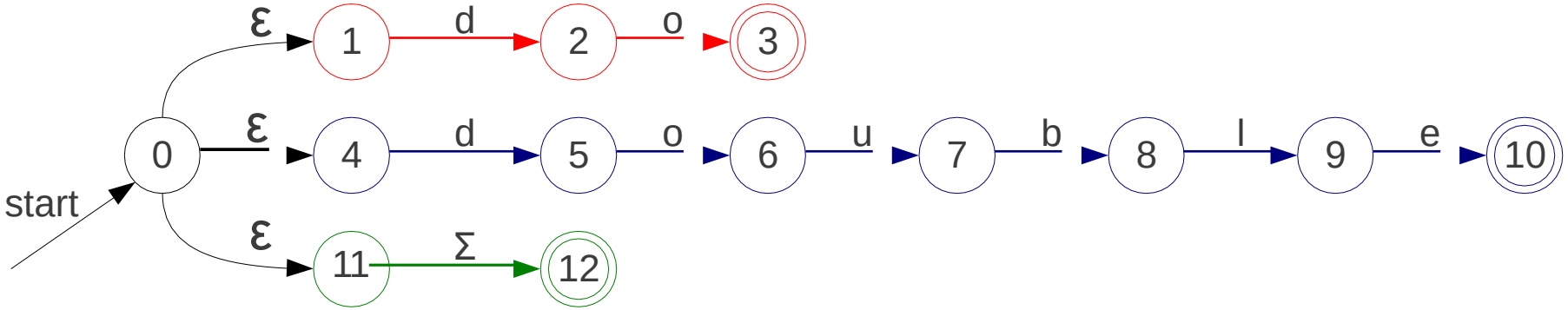
From NFA to DFA

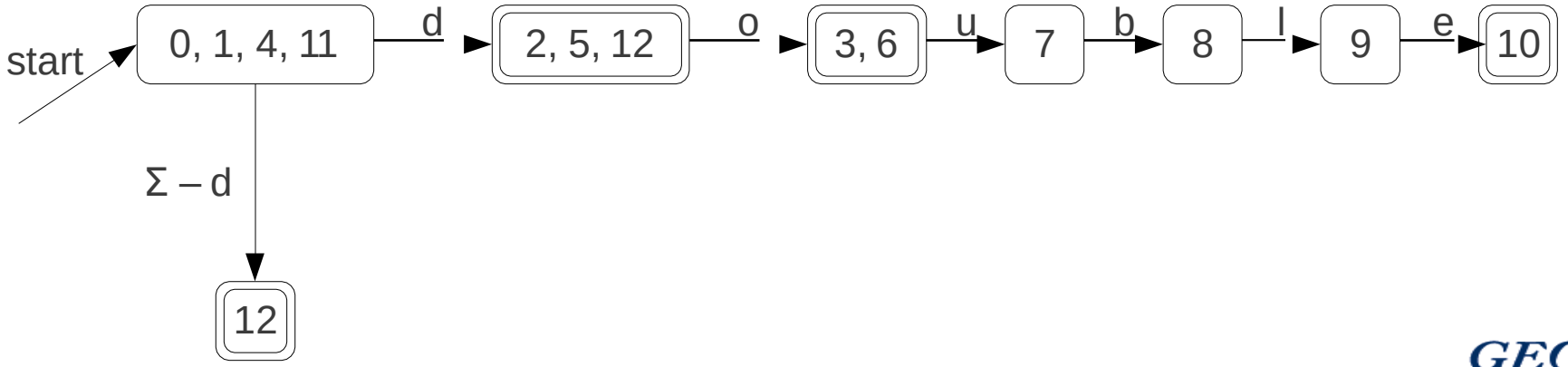




GEORGETOWN UNIVERSITY

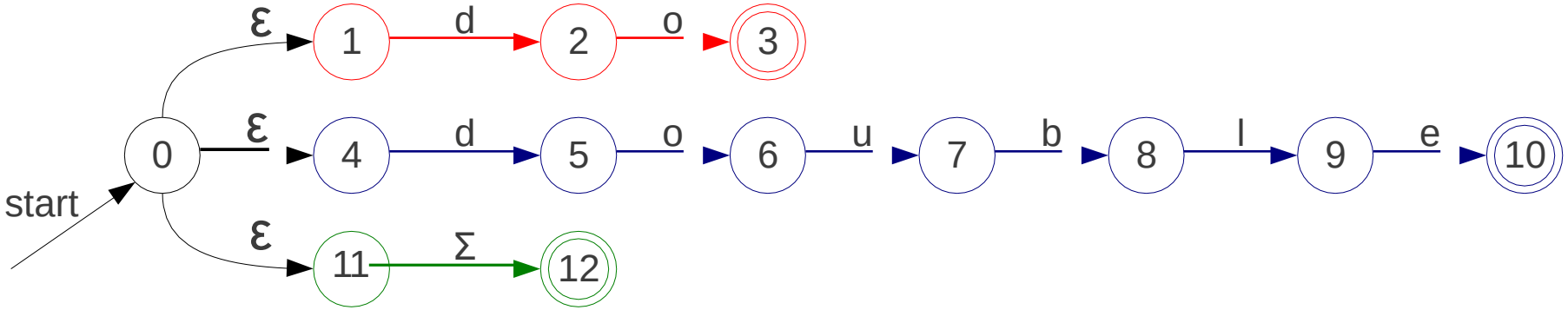
From NFA to DFA

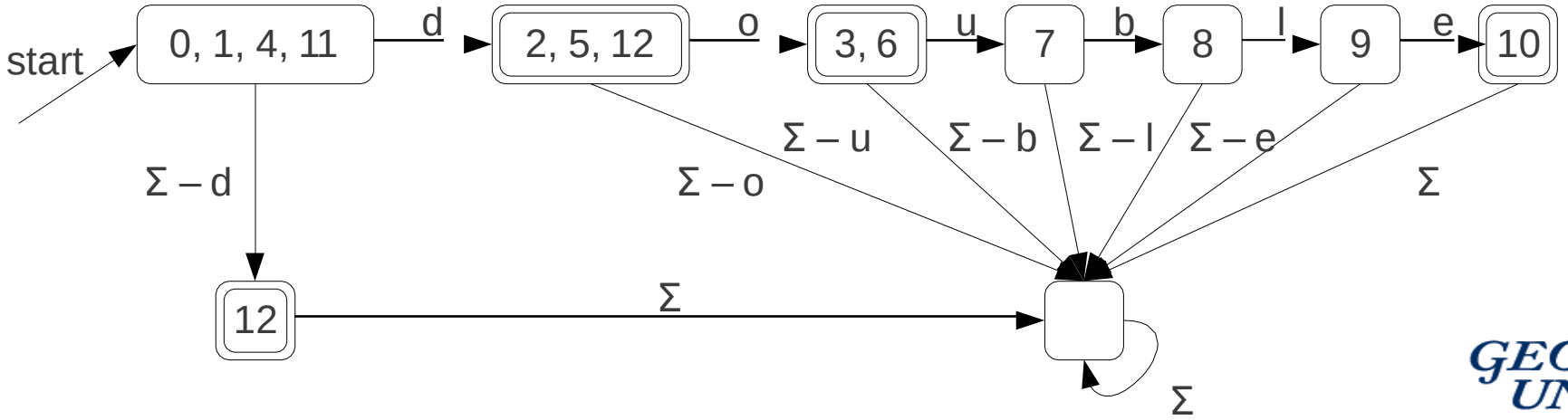




GEORGETOWN UNIVERSITY

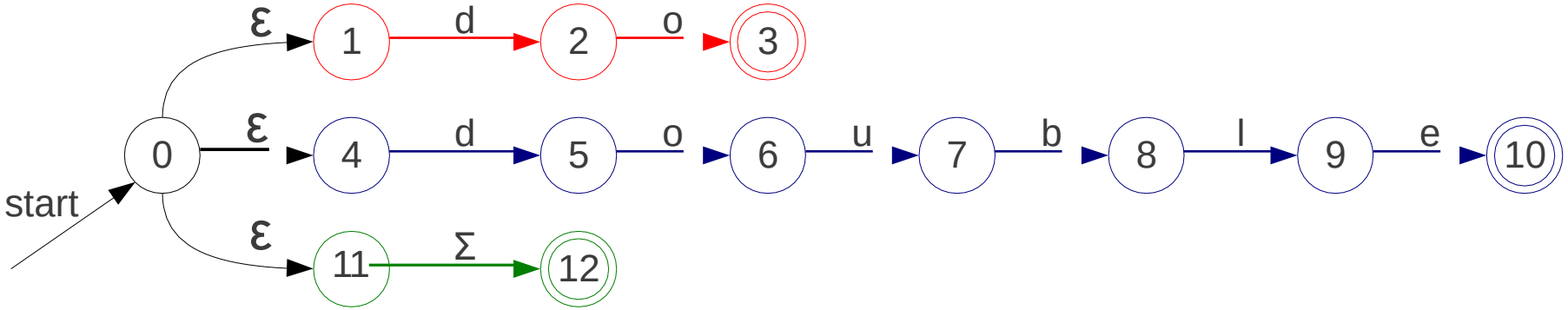
From NFA to DFA

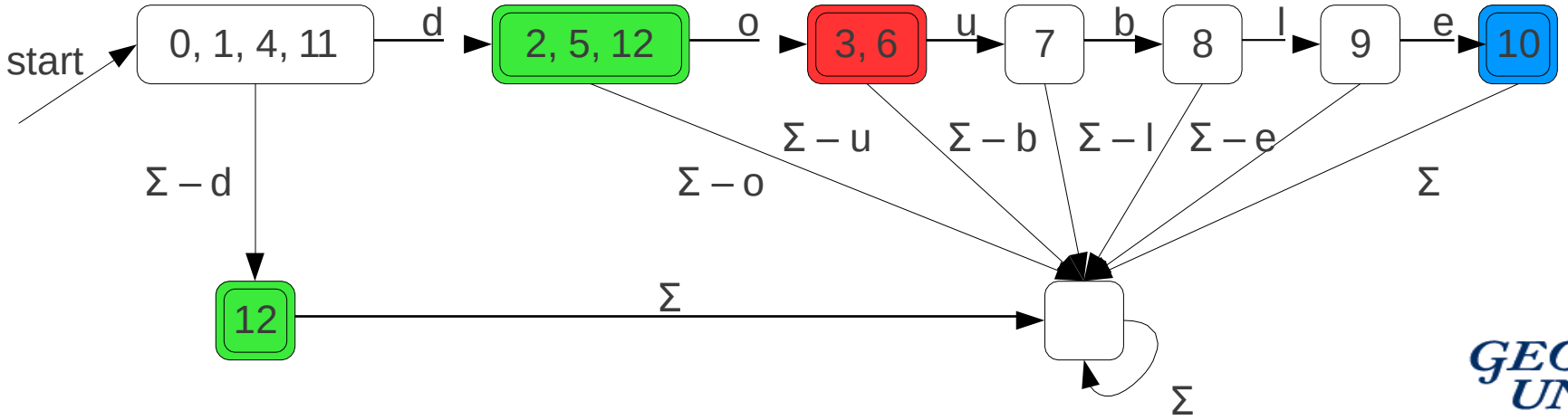




GEORGETOWN UNIVERSITY

From NFA to DFA





GEORGETOWN UNIVERSITY

Note

- Many programming Languages make us of whitespace to clearly separate many lexemes.

- Hint: We can account for this by defining a whitespace token

Summary

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

COSC252: Programming Languages:

Tokenizer

Jeremy Bolton, PhD
Adjunct Professor

GEORGETOWN UNIVERSITY

Revisit Compilers and Interpreters

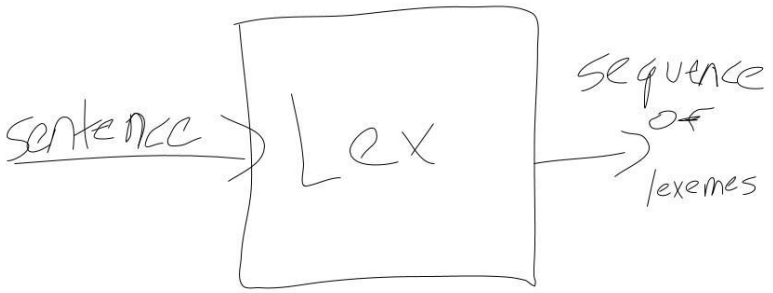
- Both Compilers and Interpreters require lexical and syntactic analysis
  - Compilers then translate
  - Interpreters then “simulate”

- Generally Lexical Analysis is separated from syntactic analysis
  - Simplicity: more reliable and structured
  - Efficiency: lexical analysis (and syntactic analysis) can be optimized
  - Portability: modularity and reusability

- We have learned about some of the theoretical constructs related to lexical and syntactic analysis – now lets look at their implementations

Lexer aka Tokenizer

- Lexical Analysis
  - First step in interpretation / compilation
  - Goal: identify (recognize) and categorize tokens



- Other implementation details to consider:
  - Comments
  - Spaces (white space)
  - “unseen” characters

- Building a Lexer
  1. Use a pre-existing tool: e.g. lex or flex
  2. DIYS: A FSM is a blue print for the your tokenizer.
     - For simplicity and clear design: 1 FSM per Token (lexeme group)

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Lexer Code (Generic Example)

```
tokenList lex(char* input)
{
	int pos = 0;
	tokenList tokens;
	token newToken;
	while(input[pos] != ‘EOF’){
		charClass c = getNextNonBlank( pos, input);
		swich(c){
			case ALPHA: newToken = lexId( input, pos ); break;
			case DIGIT: newToken = lexNum( input, pos ); break;
			case SPECIAL: newToken = lexSpecial( input, pos ); break;
		}
		tokenList.addToBack(newToken);
		//cout << ‘Next Token Category is ’ << newToken.cat << ‘. Lexeme is ‘<< newToken.lex << endl;
	}
return tokenList;
```

Lex an Identifier (example)

```
token lexVar( char* input, int &pos ){
        char lexeme[35];
        int length = 0;
        lexeme[length++] = input[pos++];
        charClass c = getNextChar( pos, input);
        while(c == ALPHA || c == DIGIT){
                lexeme[length++] = input[pos++];
                charClass c = getNextChar( pos, input);
        }
        return lexeme;
} // A very basic tokenizer … you should also include the token category in the token object
```

What does RegEx or FSM look like? Can we use the RegEx or FSM as a blueprint to design our lexer?

One more example to try (before parsers)

- Example:
  - Construct a regex to generate a language of binary strings that consists of n 0s followed by n 1s, where n is a non-neg integer: 0^n1^n
    - Construct the corresponding FSM as a recognizer

Ex (cont)

- Observations and Follow-up:
  - We cannot build a FSM for 0^n1^n , for an arbitrary non-neg n. (not finite … proof by contradiction)
  - (But I thought for every RegEx Generator there exists a FSM Recognizer)
    - Yes …
    - This implies 0^n1^n is not a RegEx!
    - Why not – Note that regular expressions consist of input alphabet elements combined with the regular operators
      - Regular Operators
        - » * arbitrary repetition
        - » concatenation
        - » | selection
  - This is not very intuitive since we can build an FSM for
    - 0^n1^n , where n = 1
    - 0^n1^n , where n = 2
    - Given our selection operator we can build an FSM for $0^11^1 \mid 0^21^2$
    - In fact we can build an FSM as long as we bound n: 0^n1^n , where $0 \leq n \leq 100$
    - But we cannot build one for any arbitrary n.

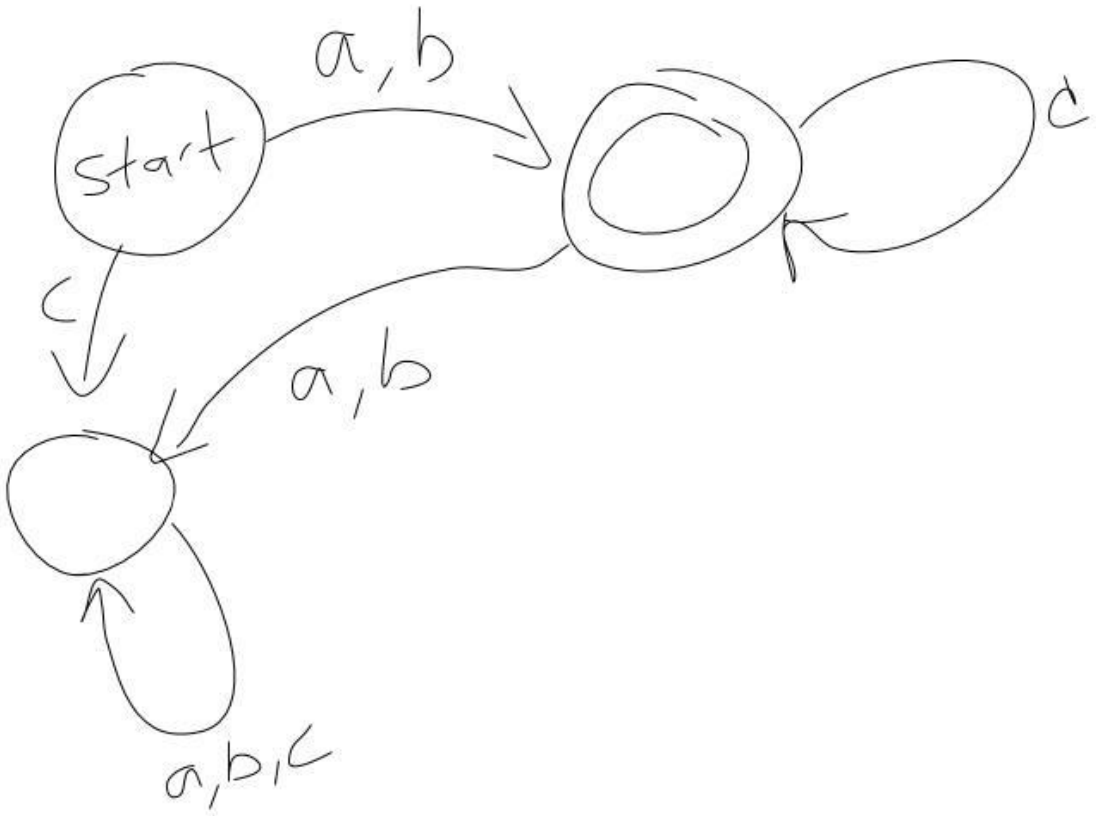
Appendix

GEORGETOWN UNIVERSITY

FSM example

- FSM to recognize (a | b) c*

- Nodes are states
- Edges indicate next character in left to right sequential scan
- The graph is traversed, beginning at start state. When a new character is encountered during scan, you traverse to the corresponding adjacent node.
- The string is accepted if you end in an accepting state (indicated by a double circle)



FSM formally

- A finite state machine is a 5-tuple:
  - (Q, Σ, δ, q_0, F)

  - Q: finite set of all states
  - Σ : alphabet (finite set of characters)
  - δ: state transition function, $\delta: Qx\Sigma \rightarrow$ Q
  - $q_0 \in Q$: start state
  - F$\subset Q$: set of accepting state(s)

FSM example with values

$Q := \{ Start, S_1, S_2 \}$

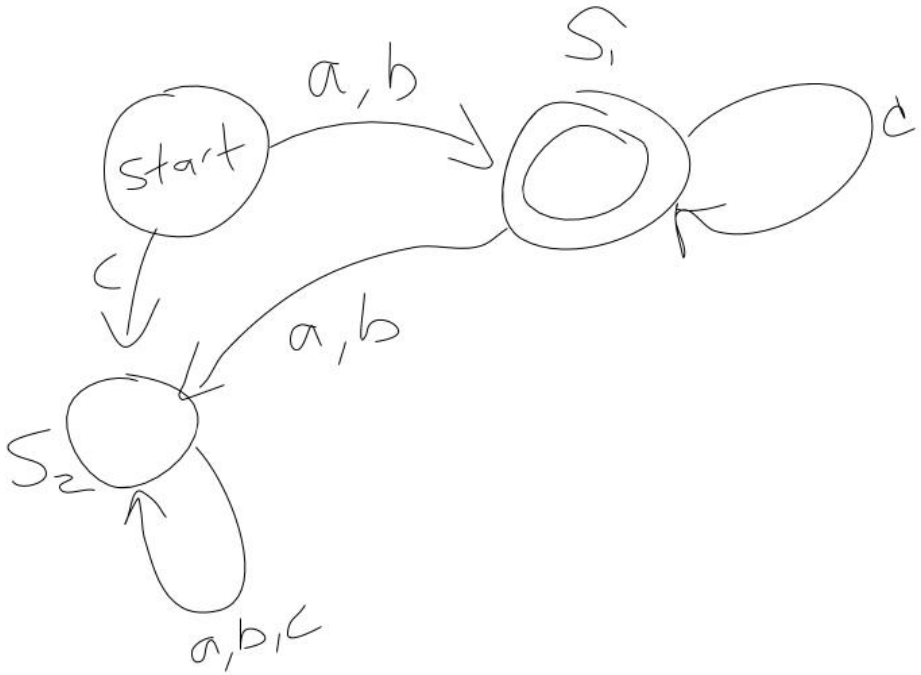
$\Sigma := \{ a, b, c \}$

$\delta :=$

	a	b	c
St	S_1	S_1	S_2
S_1	S_2	S_2	S_1
S_2	S_2	S_2	S_2

$q_0 := Start$

$F := \{ S_1 \}$



GEORGETOWN UNIVERSITY

In-Class
FSM Example: Design a FSM for a floating point token

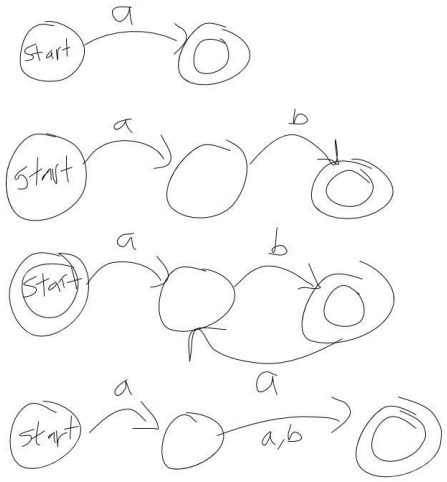
Example: RegEx and FSM

- Example:
  - Construct a regex to generate a language of binary strings that begin with an even number of 0s and ends with an odd number of 1s
    - Construct the corresponding FSM as a recognizer

RegEx to FSM

- A language L is a regular Language iff there exists a regular expressions generator. A language L is a regular Language iff there exists a finite state machine recognizer.
  - Note: for each Regular Expression, that generates a regular language L, there exists a FSM that recognizes L
  - Note: for each FSM, that recognizes a regular language L, there exists a RegEx that generates L

- Relationship between RegEx and FSM. Example $a, b \in \Sigma$
  - Any element a in the alphabet
  - Concatenation: ab
  - Repetition: (ab)*
  - Selection | : a(a | b)



GEORGETOWN UNIVERSITY

Usage of RLs and CFLs in PLs (revisited)
Use RLs to define rules for tokens. Use CFL for PL rules.

- Could you use BNF to define the set of all possible variables or ints? YES. But instead we will use regular languages and regular expressions. (More on this later)

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid -< var >$$
$$< var > \rightarrow see\ regular\ expression\ for\ variables.$$
$$< int > \rightarrow see\ regular\ expression\ for\ integers.$$

Groups of abstractions: compositions of terminals and nonterminals. We use CFGs to model these constructs.

It is intuitive to define our language in terms of its smallest syntactic unit (the most integral units that have meaning). However – these "lexemes" consist of parts (chars) themselves. Solution: Use 1 model for the (smallest syntactic units) lexemes and 1 model for the language.

Groups of tokens (sets of terminals)

We will use regular grammars to model these constructs

GEORGETOWN UNIVERSITY

We could simply use 1 model: CFLs

- This would certainly work since CFL recognizers and generators can recognize and generate any language a RL recognizer and generator can recognize and generate.

$$< stmt > \rightarrow < var > = < expr >$$
$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < neg > \mid < term > * < neg > \mid < term > / < neg >$$
$$< neg > \rightarrow < var > \mid - < var >$$

…
<var> -> <alpha> <alphaNum>
<alphaNum> -> <alpha> <alphaNum>| <num><alphaNum> |ϵ
$< alpha > \rightarrow a \mid b \mid \ldots \mid z$
$< num > \rightarrow 0 \mid 1 \mid \ldots \mid 9$

- So why not?
  - As noted, the form of tokens is notably less complicated than the form of programs. If we use a simpler model for the tokenizer, we can simplify this process.

- Where do we “draw the line” where does the token model end and the program language model begin
  - Intuitively: the tokens should describe the form of the most integral syntactic unit. The smallest unit for which we can supply meaning (semantics)

GEORGETOWN UNIVERSITY

Use of Language Models for PLs

- In general: RLs are used to model the tokens
  - Each token class is a regular language. *The input alphabet is the set of all characters.*
  - The job of the tokenizer is to “recognize” each token class
    - In a left to right scan of the input, the tokenizer can recognize the beginning and end of each token (given the rules: fsm of each token class). Thus creating a token list.
- In general: CFLs are used to model the programming language (int terms of the tokens / lexemes)
  - Each programming language is a CFL. *The input alphabet is the set of all tokens.*
  - The job of the parser is to “recognize” and produce a parse tree.
    - In a left to right scan of the input (token sequence), the parser will determine if the token sequence conforms with the rules of the grammar, thus creating a parse tree.

Syntactic Analyzer aka Parser

- Parsers construct (or trace out) the parse tree
  - Goals:
    - Determine if syntax is correct
    - The tree is explicitly constructed or “traced” out

- Two categories of parsers:
  - Top-Down: Tree is “constructed” from root down
  - Bottom-Up: Tree is “constructed” from leaves up
  - Example: 3 + 5 * 1

$$< expr > \rightarrow < term > \mid < expr > + < term > \mid < expr > - < term >$$
$$< term > \rightarrow < num > \mid < term > * < num > \mid < term > / < num >$$
$$< num > \rightarrow see\ regEx$$

GEORGETOWN UNIVERSITY

*The **Parsing Problem**: Top-Down Parsers*

- Top-Down
  - Nodes in parse tree are “visited” in pre-order.
  - Intuitive but not simple.
  - Creates a Left to Right, Left Most Derivation.

- Crux of top-down parsing.
  - During derivation, choosing which RHS when deriving leftmost non-terminal
  - (Mid derivation) Given a sentence in the form xAα ,
    - x is a sequence of terminals
    - A is leftmost non-terminal
    - α is mix of terminals and non-terminals (yet to be derived)

    - the parser must choose the “correct” RHS of A=> rule.
  - Its best if this decision can be made based on the first tokens of A’s RHSs.
    - This is true if all first terminals of A productions are different.

*The **Parsing Problem**: Bottom-Up Parsers*

- Bottom-Up
  - Tree “built” from leaves up.
  - aka shift-reduce parsers
  - Creates a left to right, right most derivation (in reverse)

- Crux of bottom up parser
  - At each reduction step, a RHS matching the substring of the input is replace by the LHS of the production (merging rather than expanding)

How hard is the general parsing problem?

- For any general, un-ambiguous grammar: $O(n^3)$
- Why? If the parser makes an incorrect decision, it may need to backtrack and rebuild the parse tree from the point of the mistake
- Most compilers are however O(n)
  - Syntax for PLs are chosen to permit “fast” parsing

Appendix

GEORGETOWN UNIVERSITY