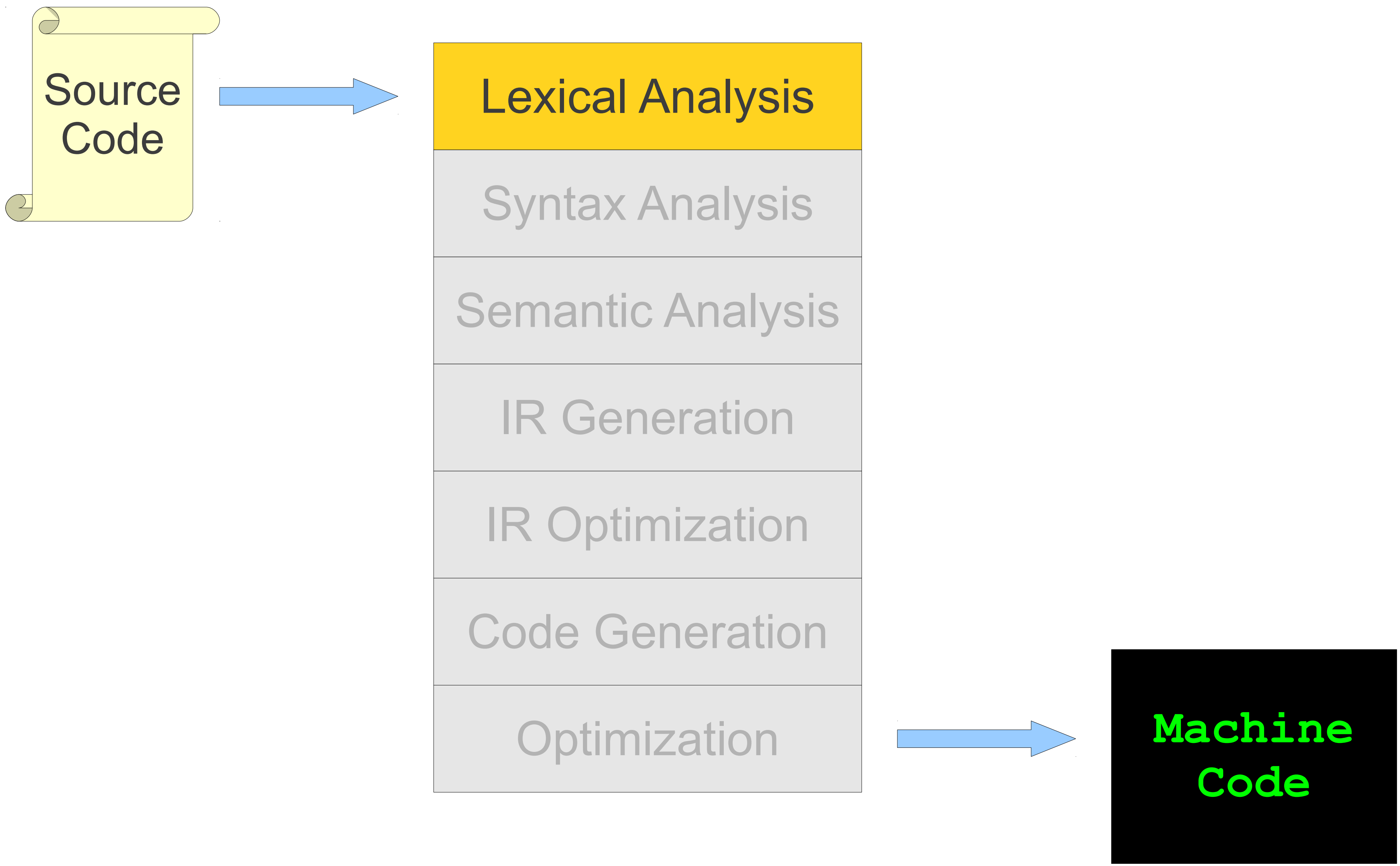
Lexical Analysis

Announcements

- **Programming Assignment 1 Out**
  - Due Monday, July 9 at 11:59 PM.
- Four handouts (all available online):
  - Decaf Specification
  - Lexical Analysis
  - Intro to `flex`
  - Programming Assignment 1

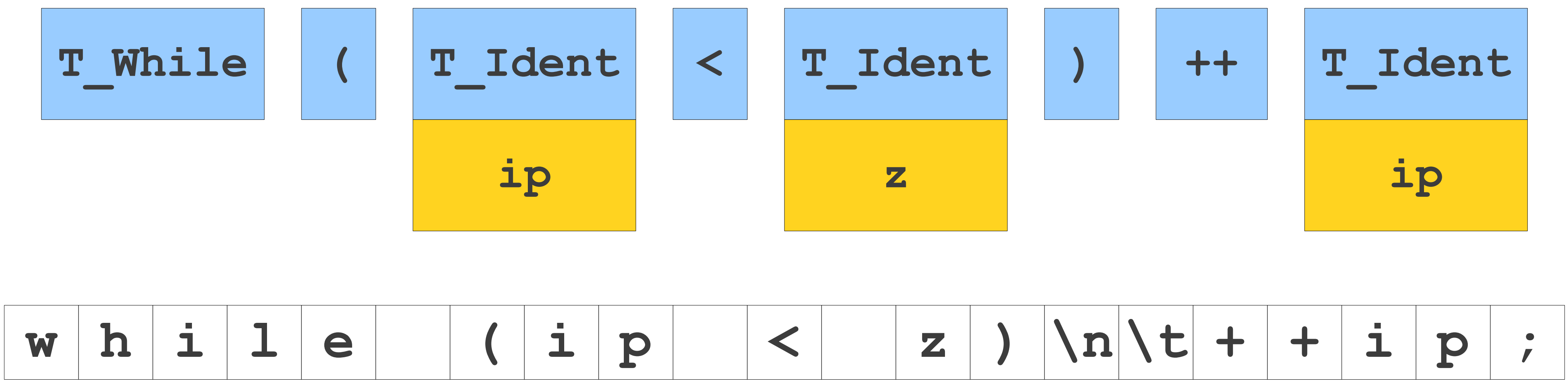
Where We Are


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

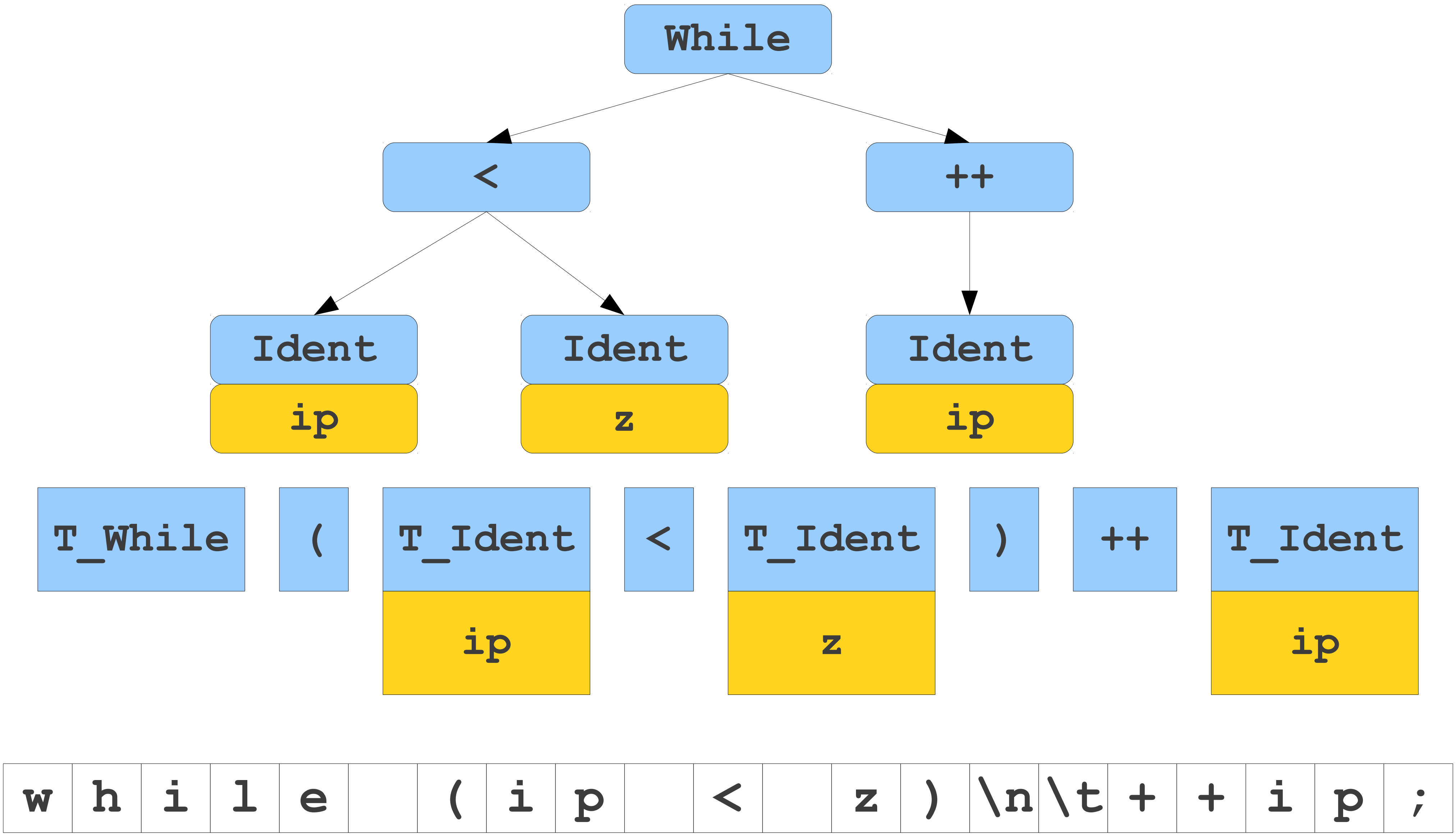
```
while (ip < z)
    ++ip;
```

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

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




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

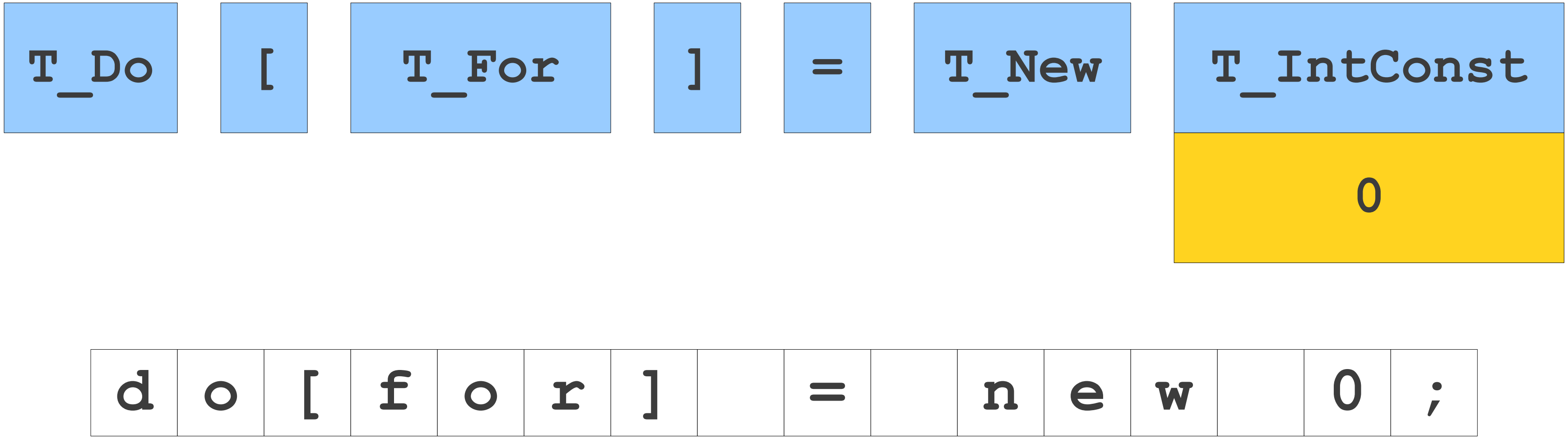

While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

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

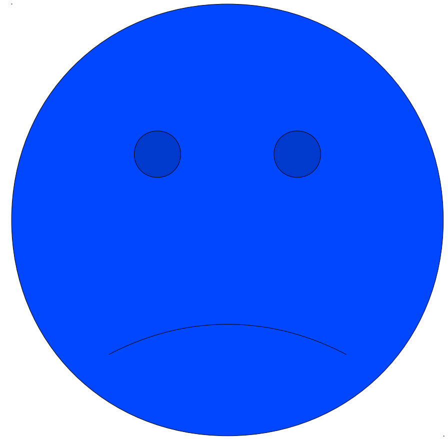
```
do[for] = new 0;
```

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

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




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



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

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

do[for] = new 0;

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

`T_While`

Scanning a Source File

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

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

`T_While`

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

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Sometimes we will discard a lexeme rather than storing it for later use. Here, we ignore whitespace, since it has no bearing on the meaning of the program.

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While

Scanning a Source File

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

T_While	(

Scanning a Source File

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

T_While	(

Scanning a Source File

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

T_While	(

Scanning a Source File

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

T_While	(

Scanning a Source File

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

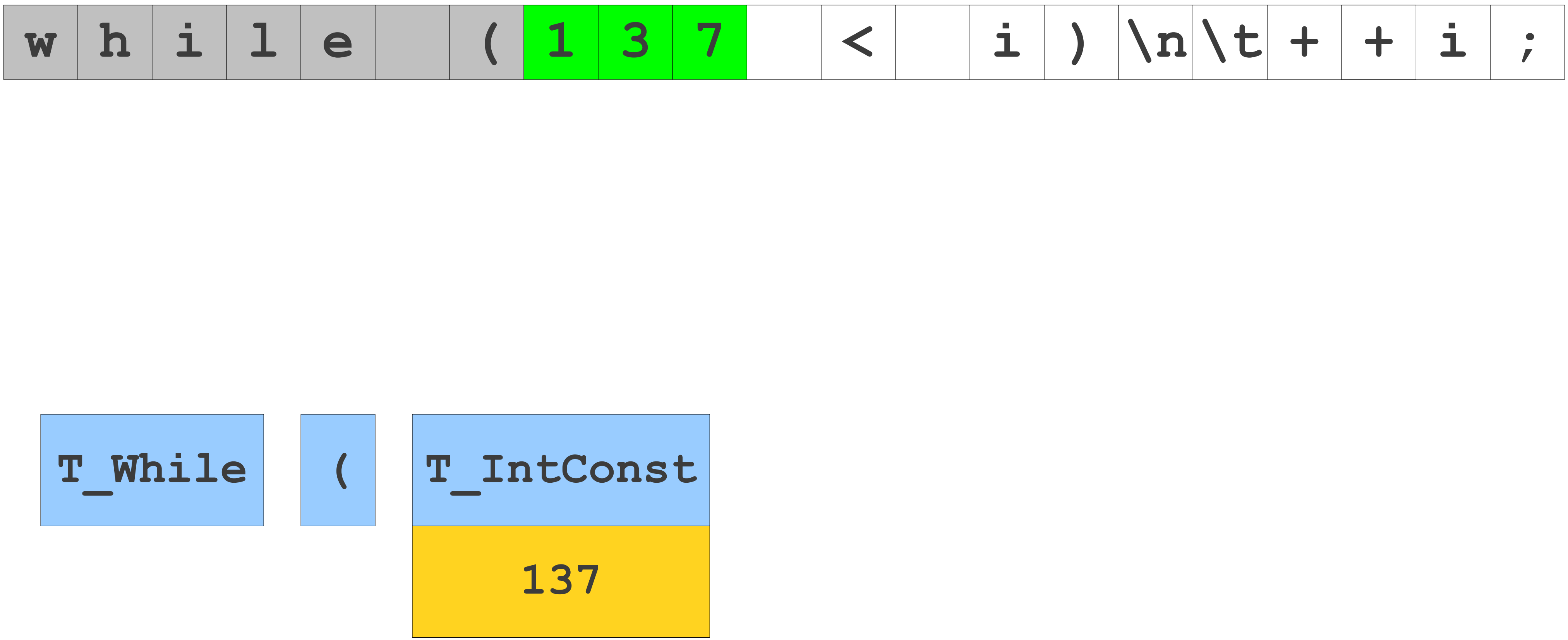
T_While	(

Scanning a Source File

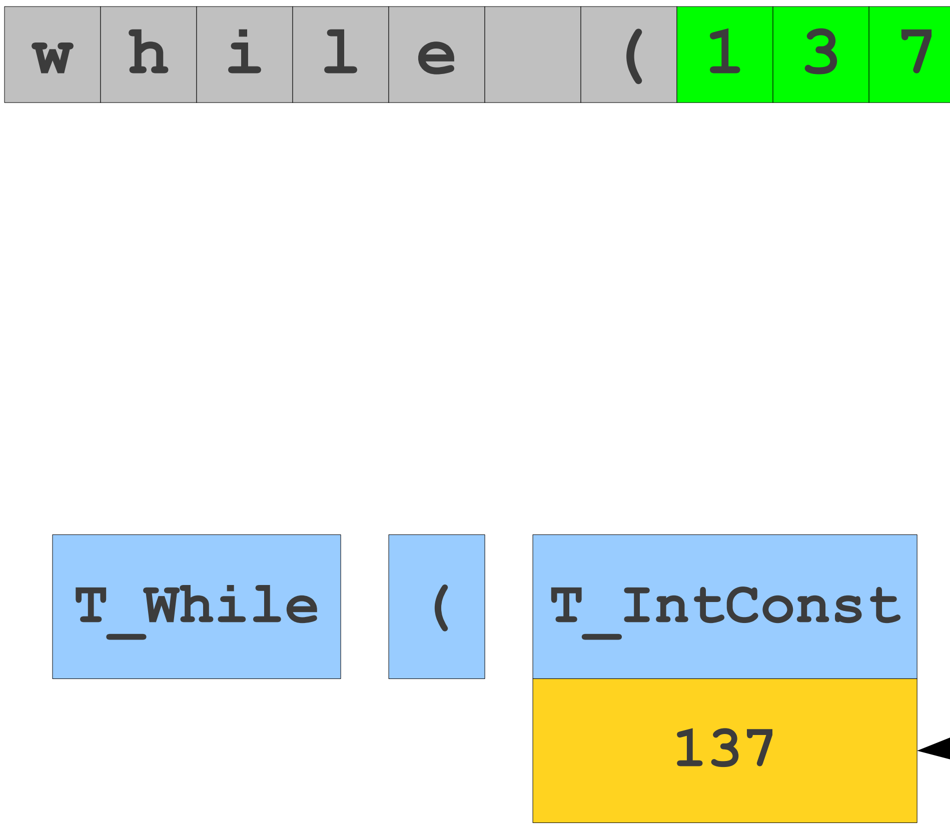
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_While (

Scanning a Source File



Scanning a Source File



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

Goals of Lexical Analysis

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

Choosing Tokens

What Tokens are Useful Here?

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

What Tokens are Useful Here?

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

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

What Tokens are Useful Here?

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

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

Identifier
IntegerConstant
```

Choosing Good Tokens

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

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I   = 1.25
```

Thanks to Prof. Alex Aiken

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

```
DO 5 I = 1,25
 DO5I  = 1.25
```

- Can be difficult to tell when to partition input.

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

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.

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
```

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.

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.
  - Define the language using a regular expression.
- We can use these compact descriptions of the language to define sets of strings.
- Over the course of this class, we will use all of these approaches.

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (the *regular languages*).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems, 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, $\mathbf{R_1R_2}$ 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, $\mathbf{R_1 \mid R_2}$ 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
11111011110011111

Simple Regular Expressions

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

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

11011100101
0000
11111011110011111

Simple Regular Expressions

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

Simple Regular Expressions

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

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

Simple Regular Expressions

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

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

Simple Regular Expressions

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

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

0000
1010
1111
1000

Simple Regular Expressions

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

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

0000
1010
1111
1000

Simple Regular Expressions

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

(0|1){4}

0000
1010
1111
1000

Simple Regular Expressions

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

$$\mathbf{1^*(0 \mid \varepsilon)1^*}$$

Simple Regular Expressions

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

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

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

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

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

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

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

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

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

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

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

Applied Regular Expressions

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

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

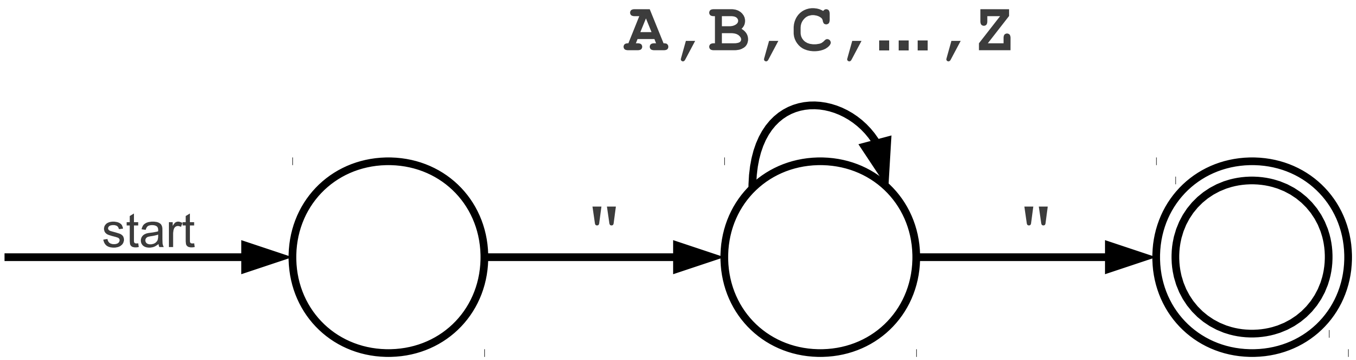
42
+1370
-3248
-9999912

Matching Regular Expressions

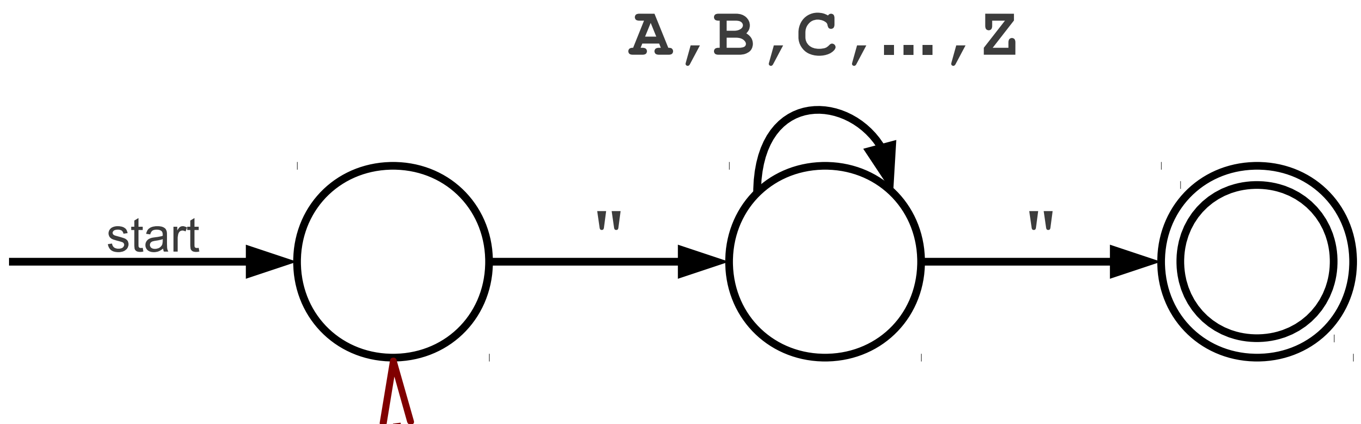
Implementing Regular Expressions

- 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

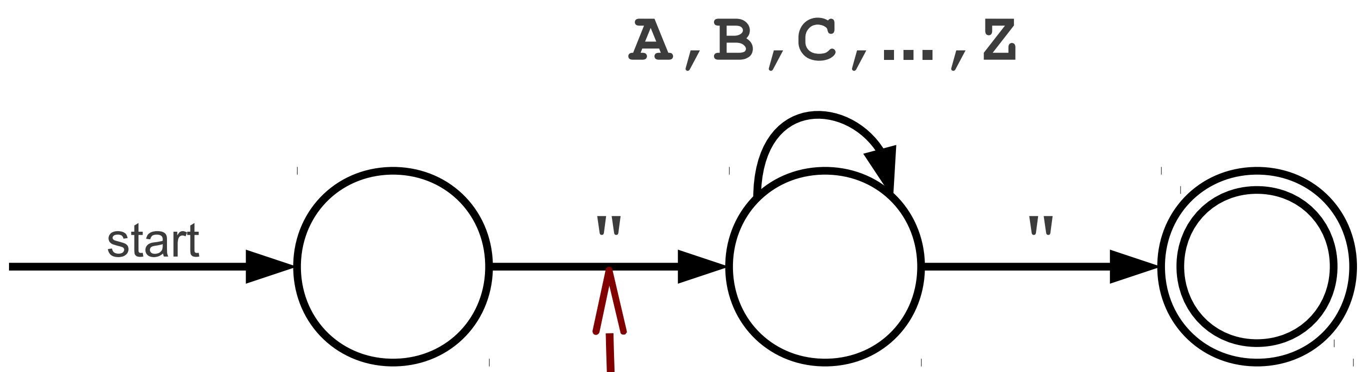


A Simple Automaton



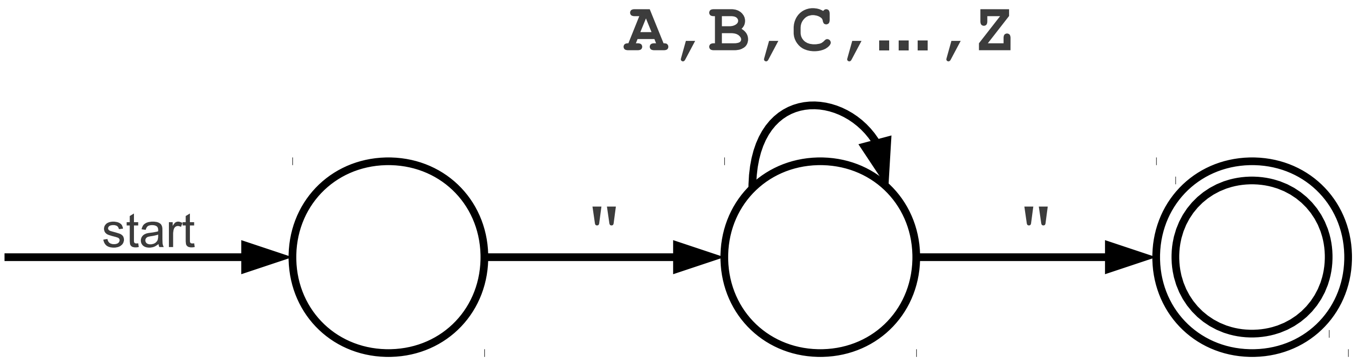
Each circle is a state of the automaton. The automaton's configuration is determined by what state(s) it is in.

A Simple Automaton



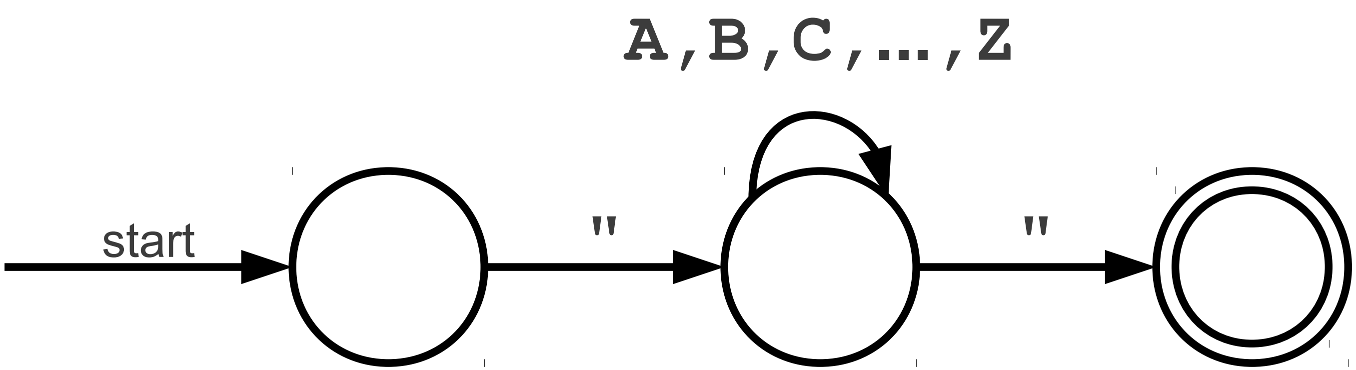
These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

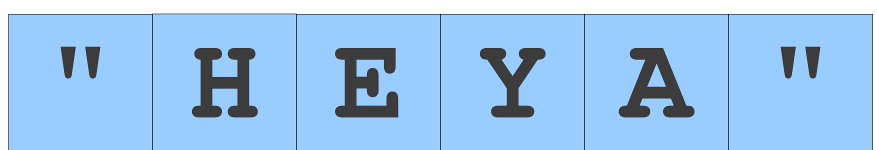
A Simple Automaton



"	H	E	Y	A	"

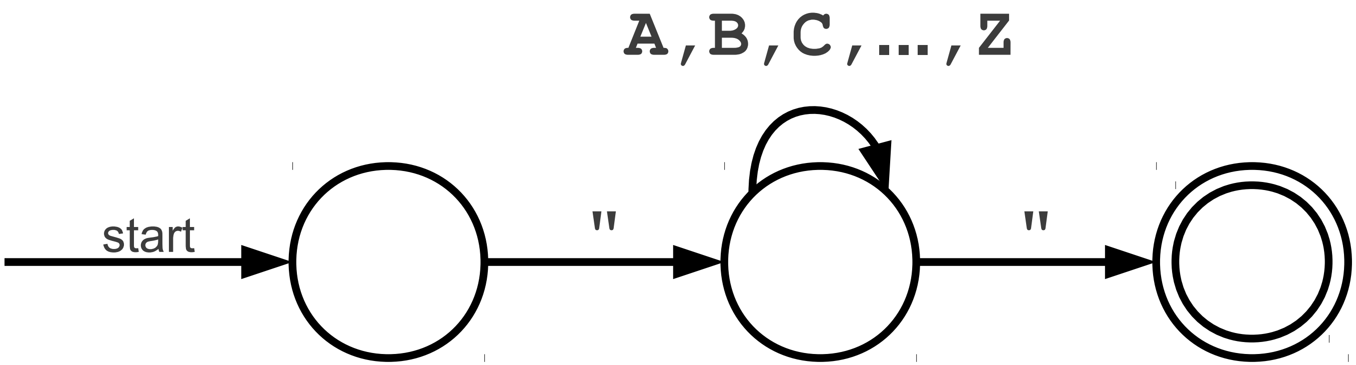
A Simple Automaton



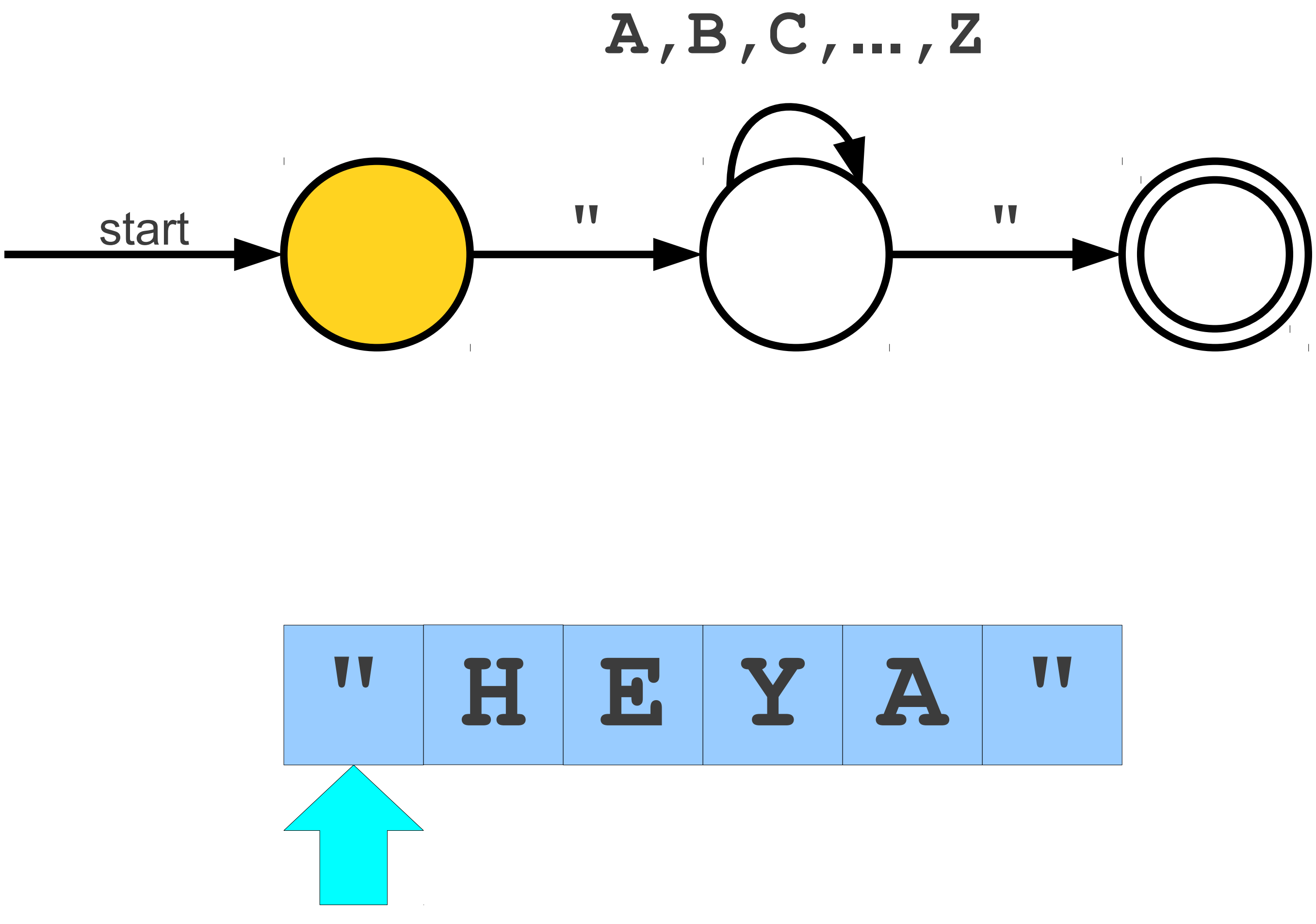


The automaton takes a string as input and decides whether to accept or reject the string.

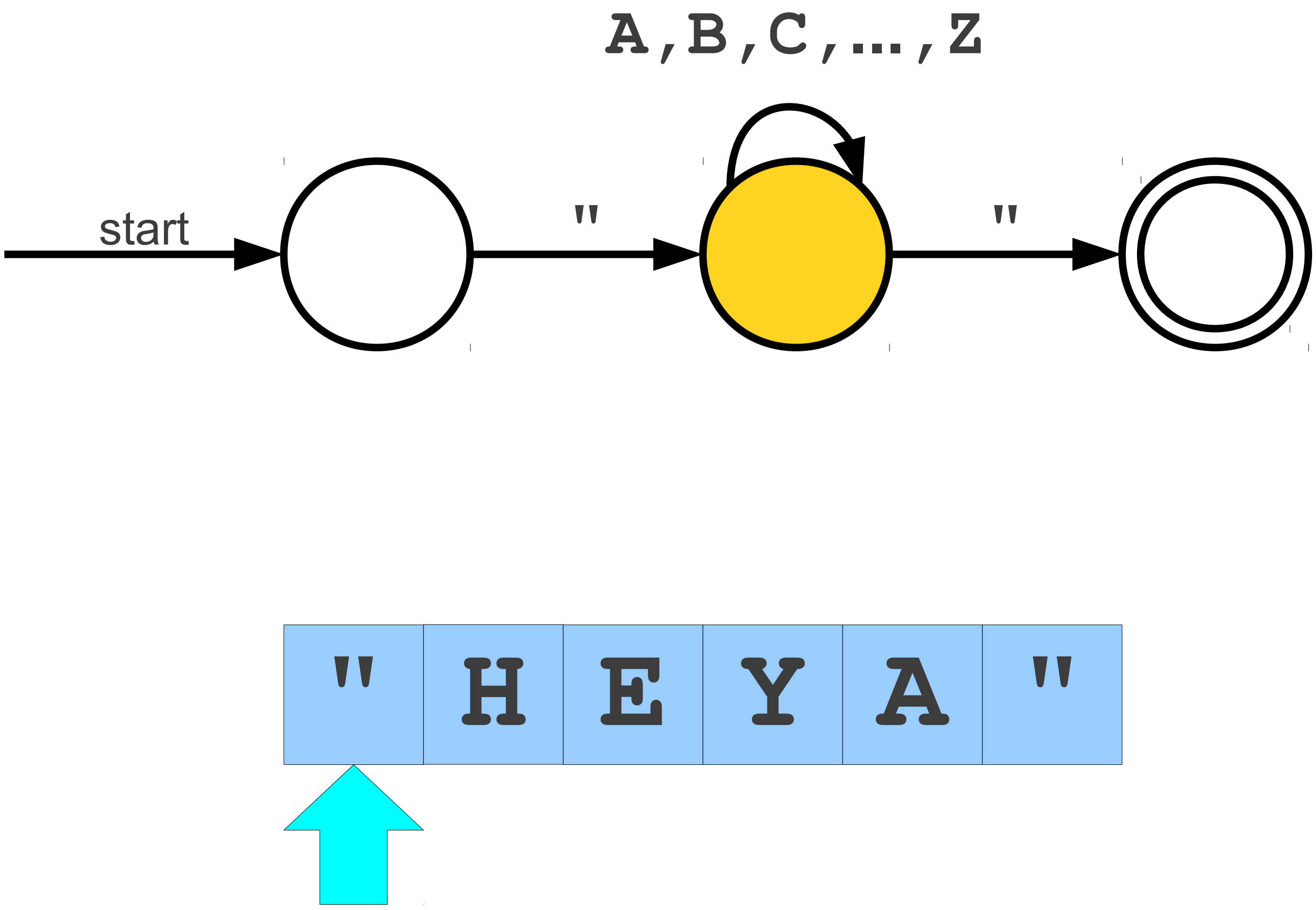
A Simple Automaton



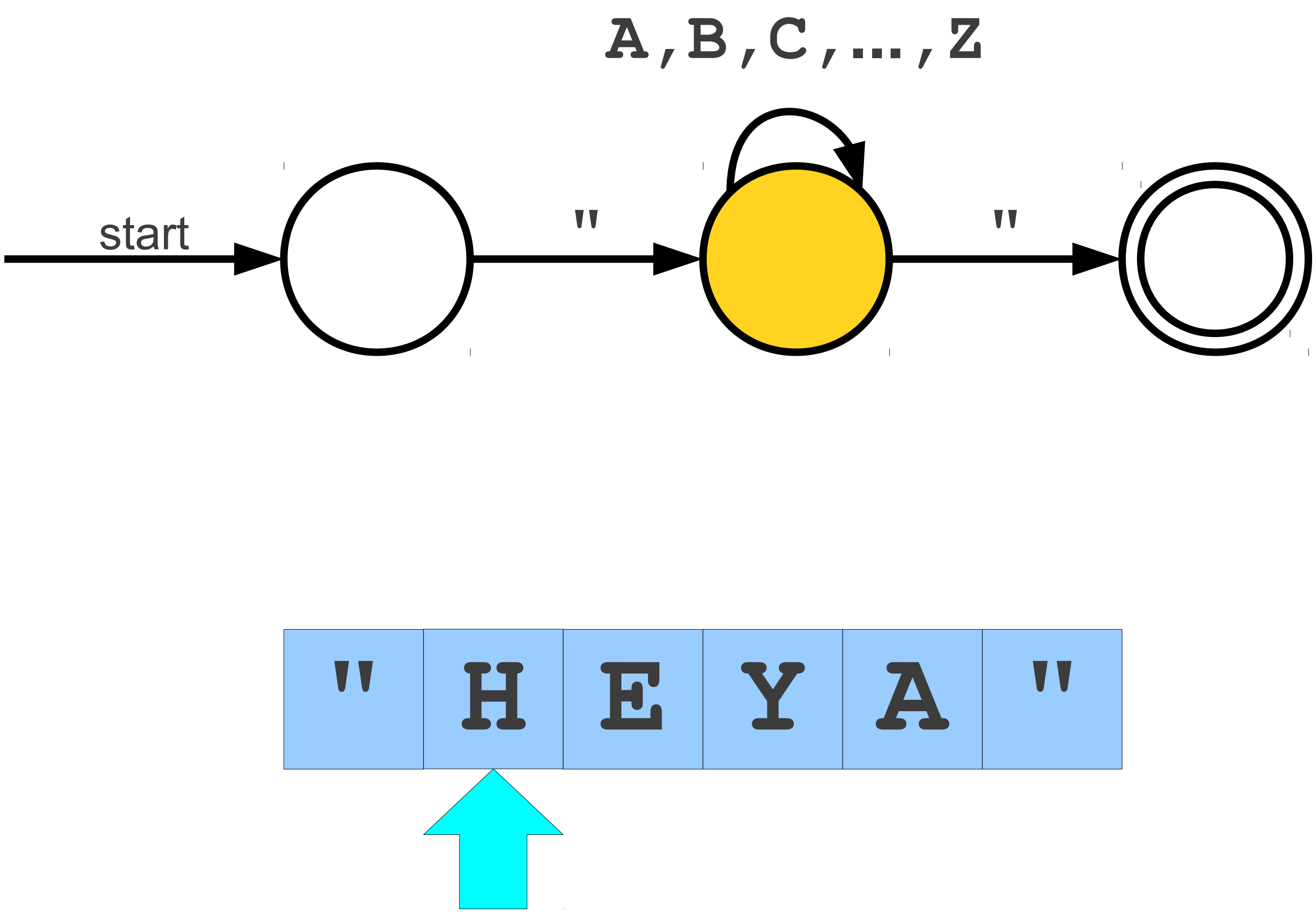
A Simple Automaton



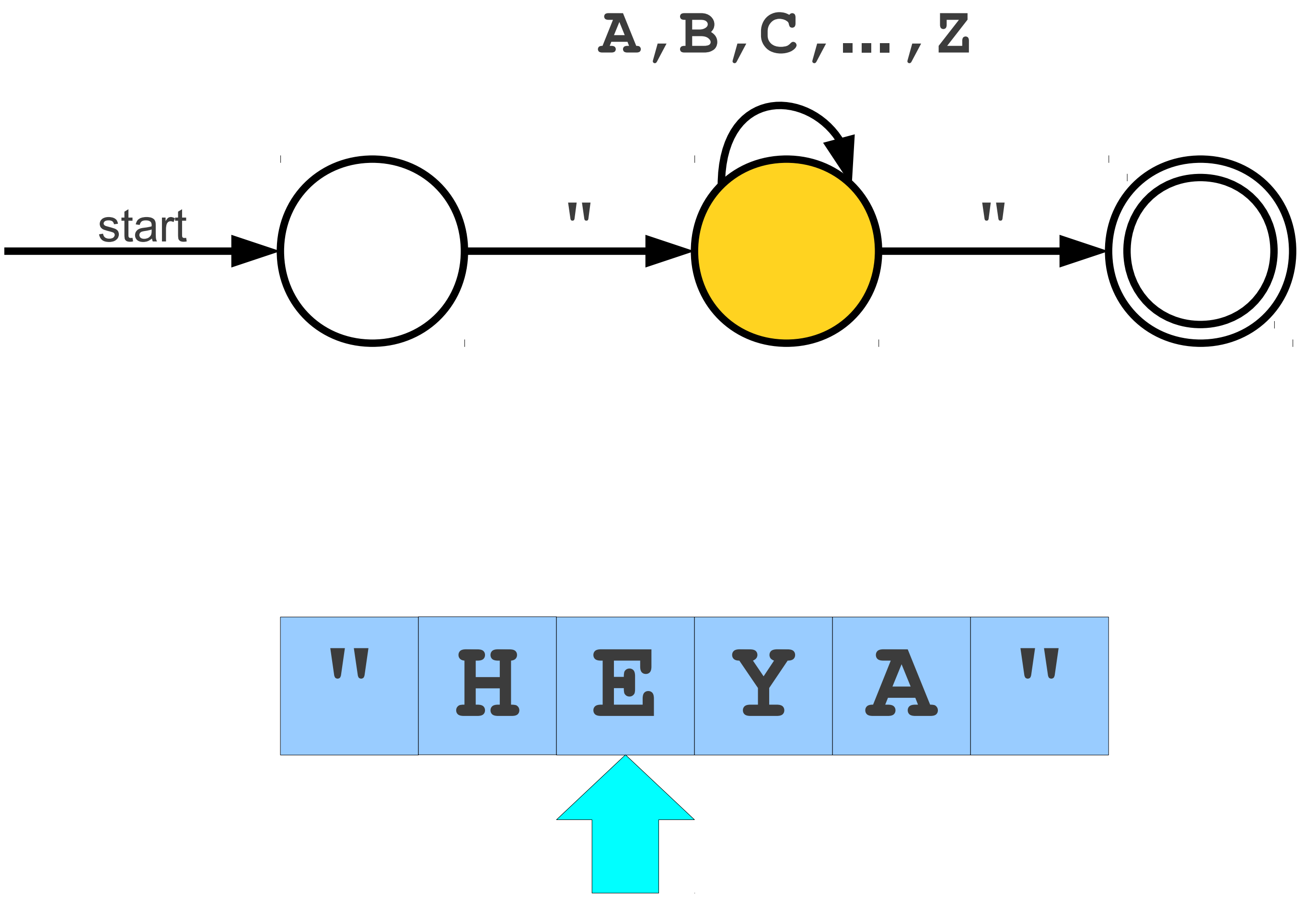
A Simple Automaton



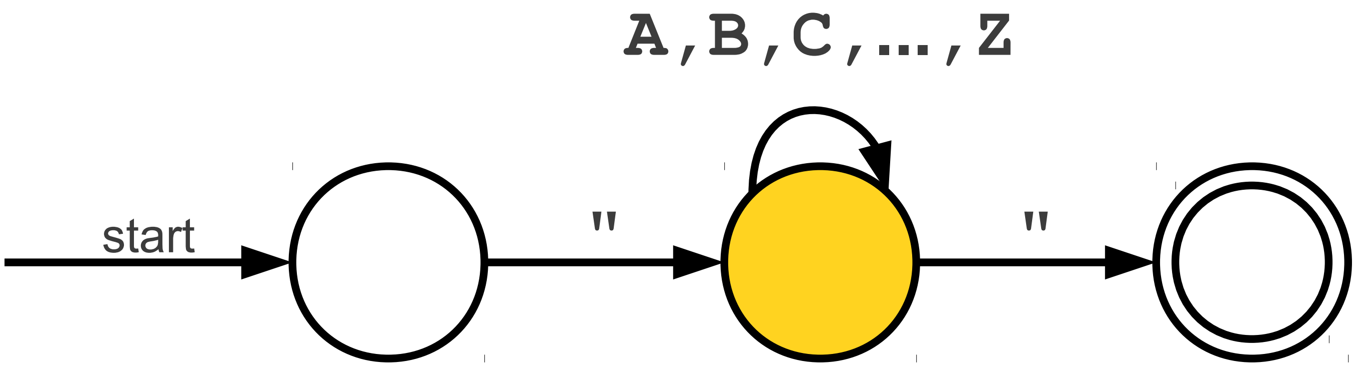
A Simple Automaton

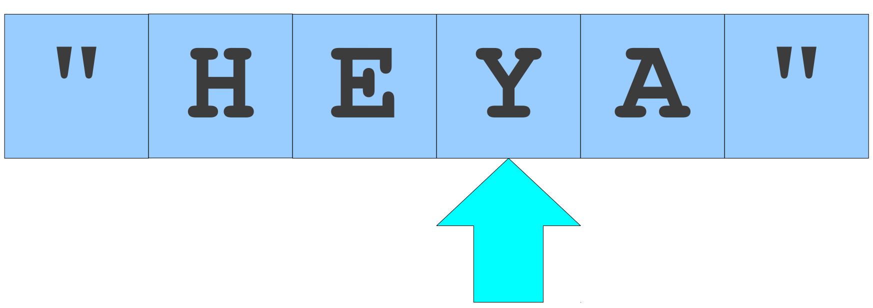


A Simple Automaton

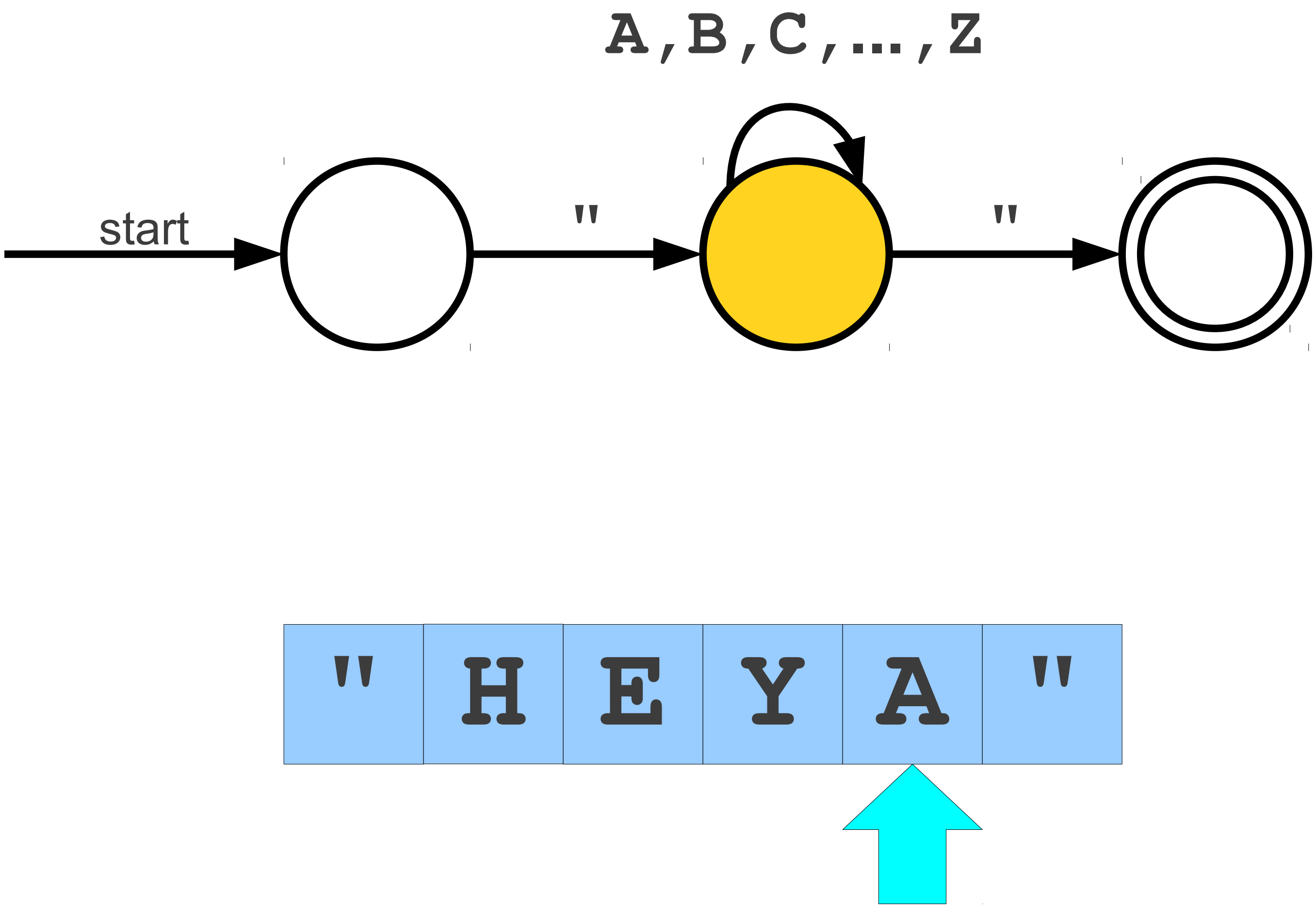


A Simple Automaton

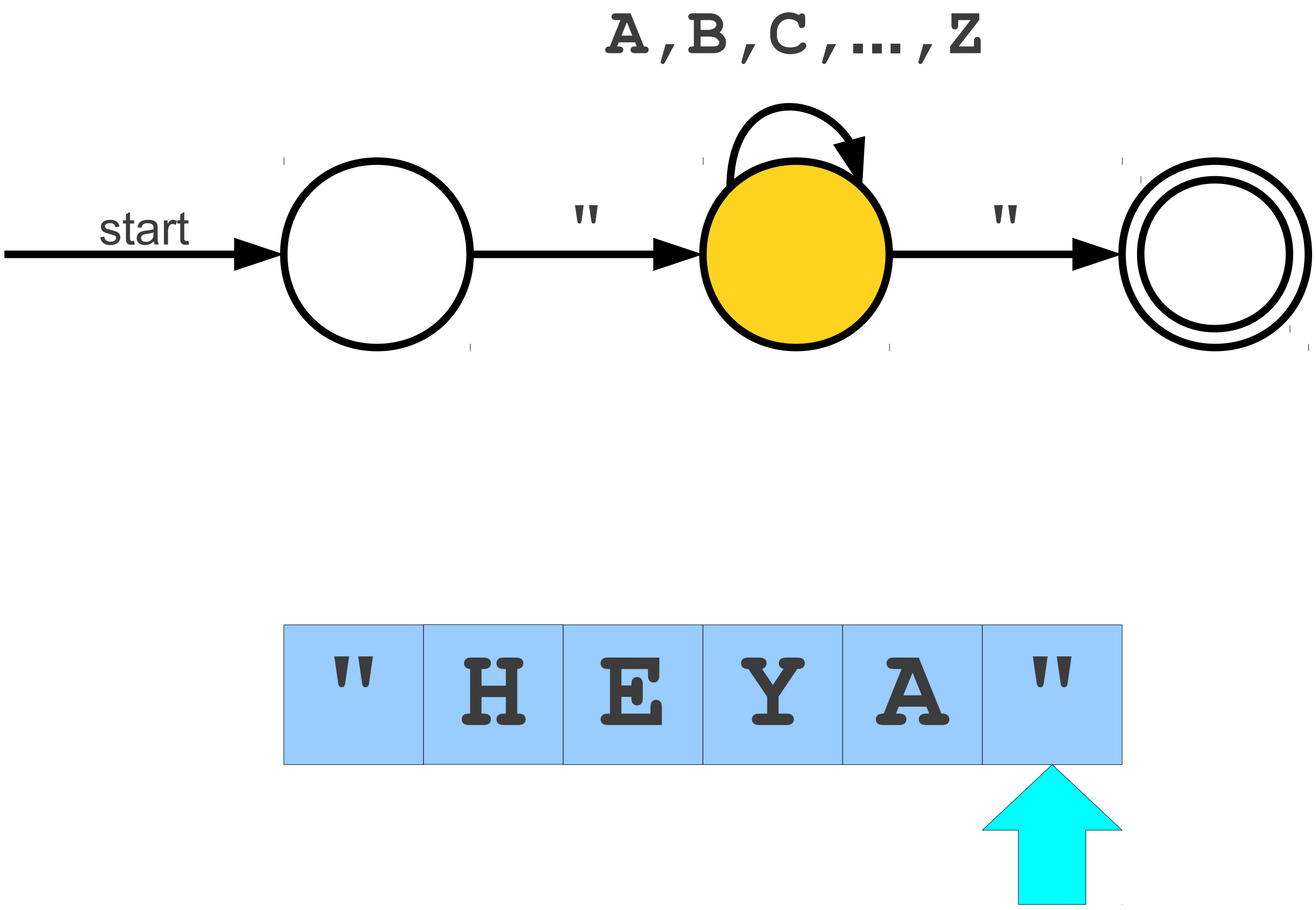




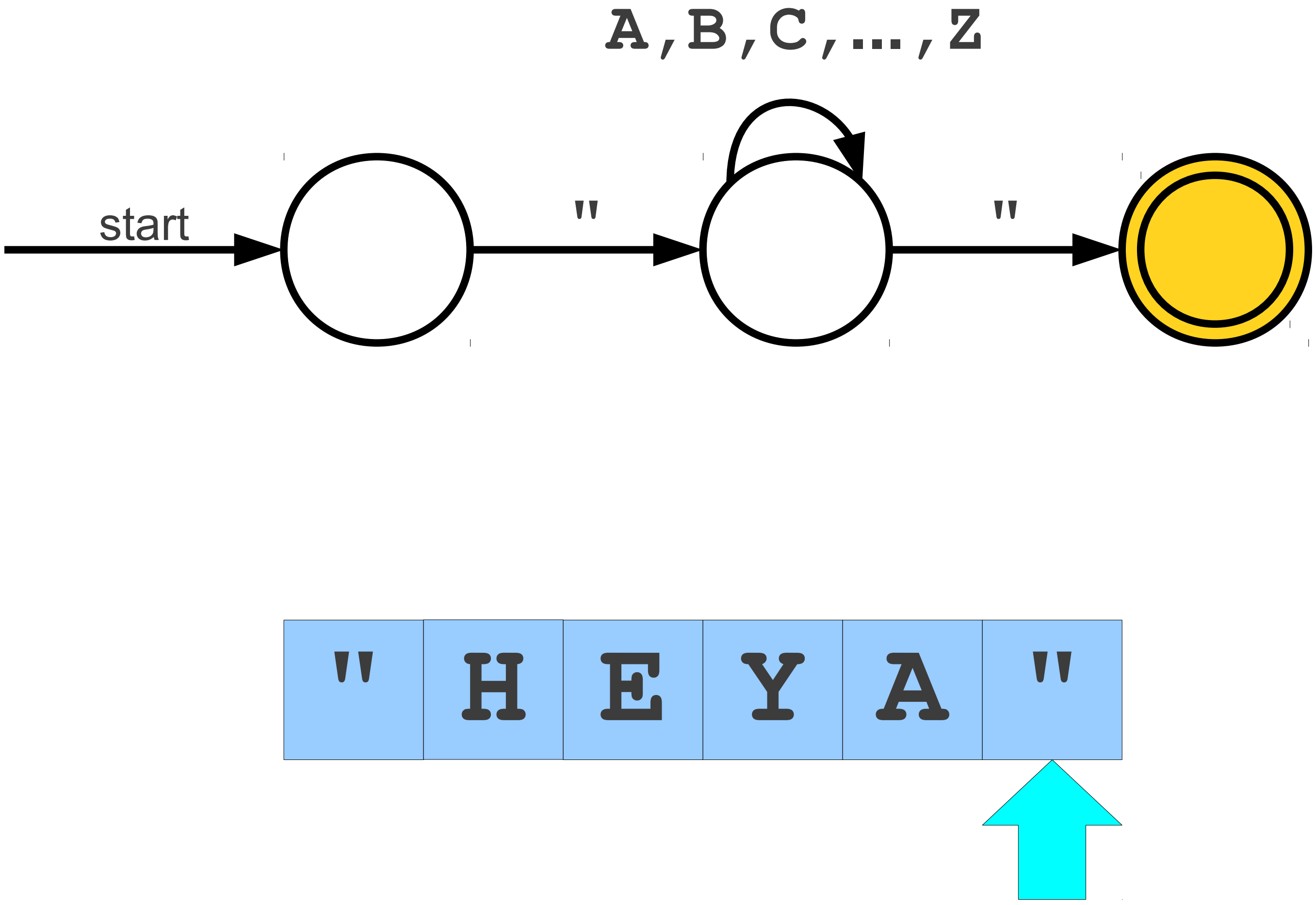
A Simple Automaton



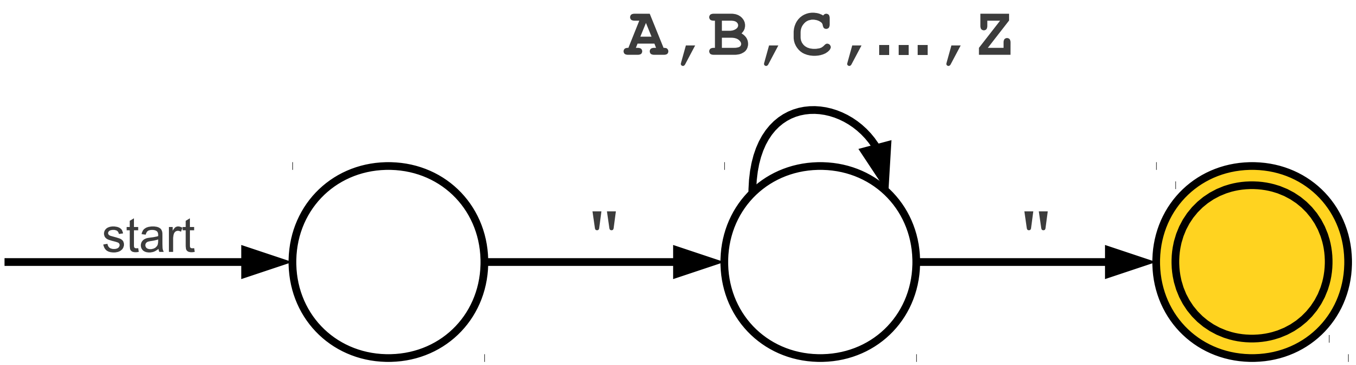
A Simple Automaton

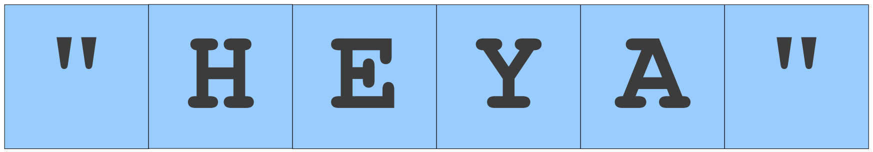


A Simple Automaton

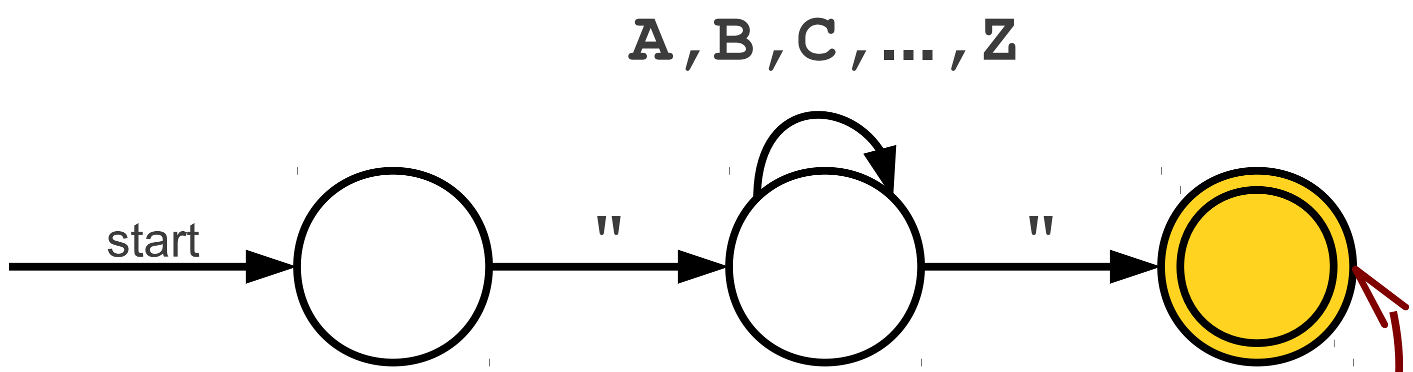


A Simple Automaton





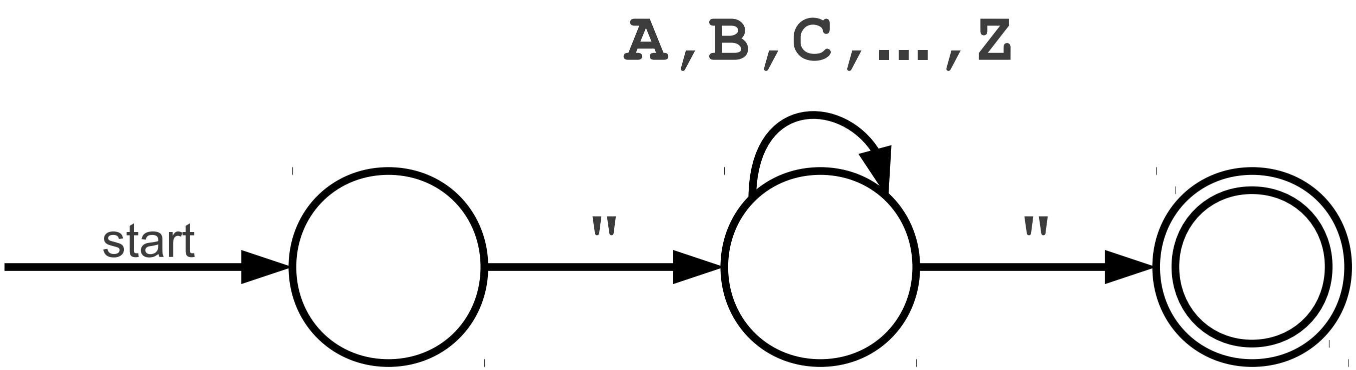
A Simple Automaton

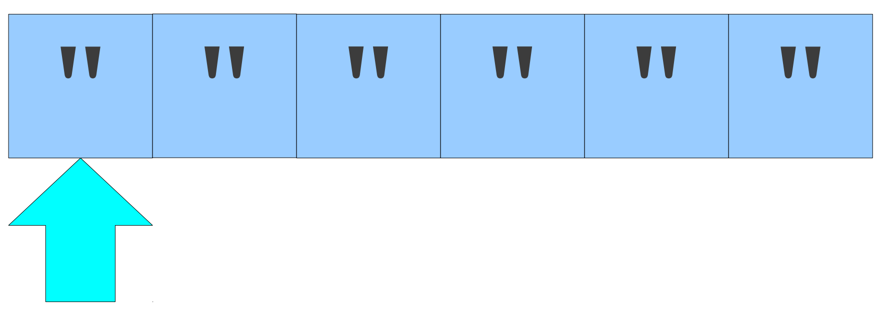


"	H	E	Y	A	"

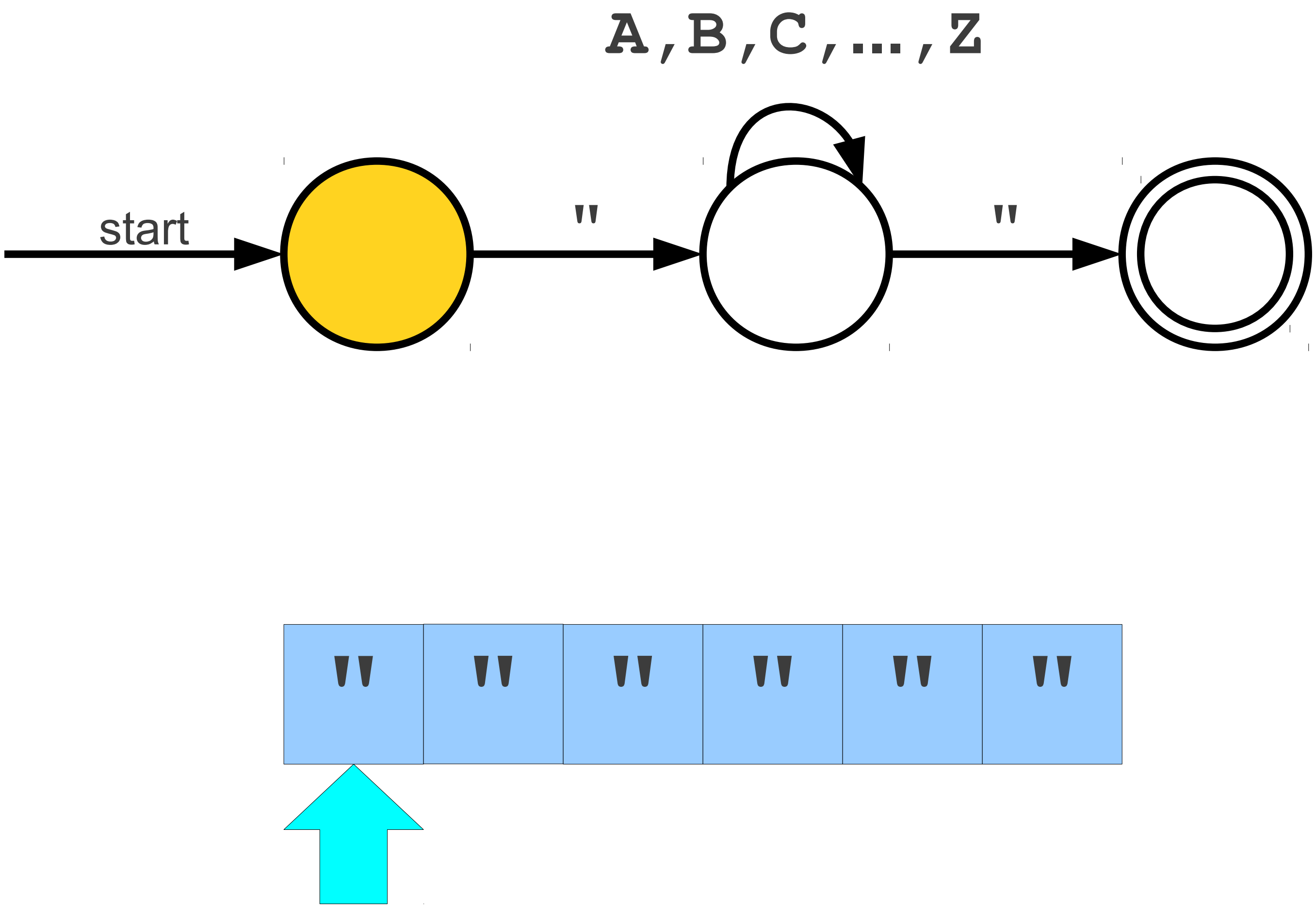
The double circle indicates that this state is an **accepting state.** The automaton accepts the string if it ends in an accepting state.

A Simple Automaton

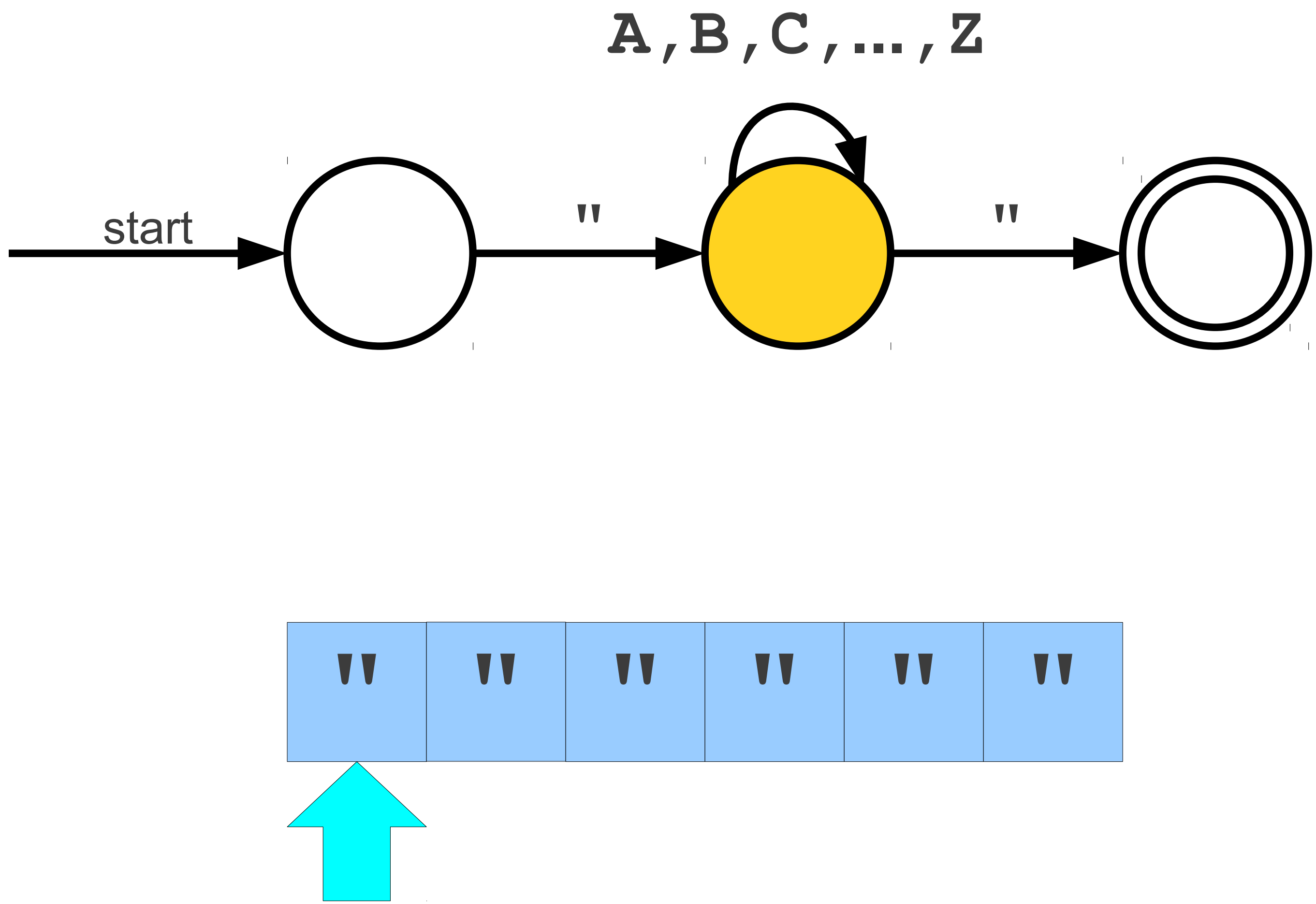




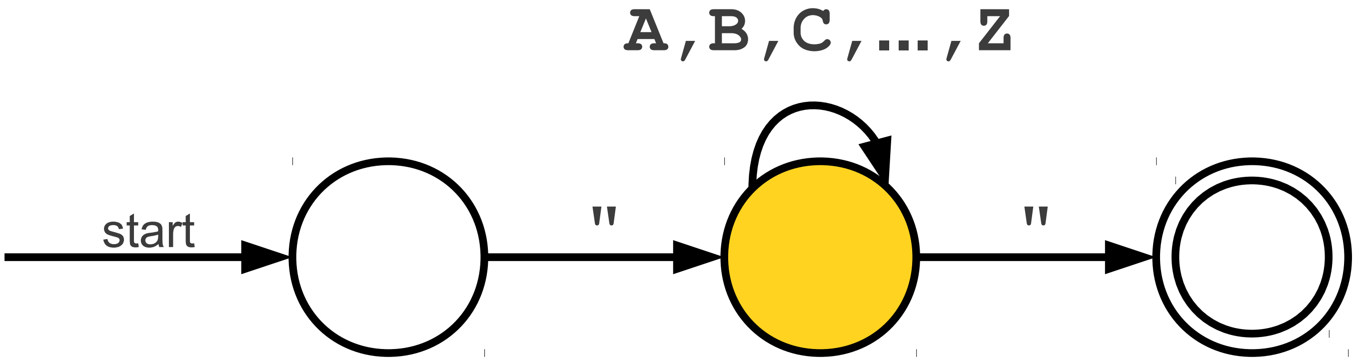
A Simple Automaton

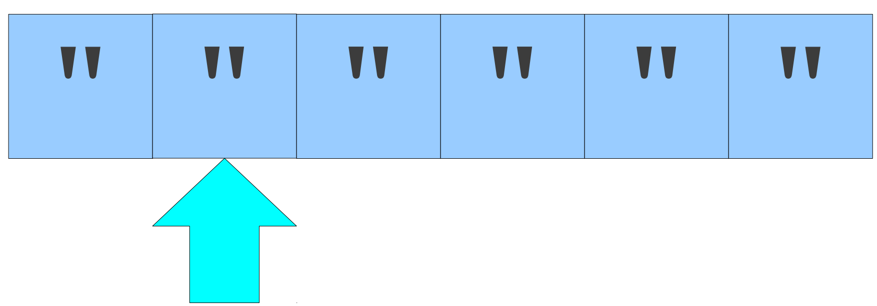


A Simple Automaton

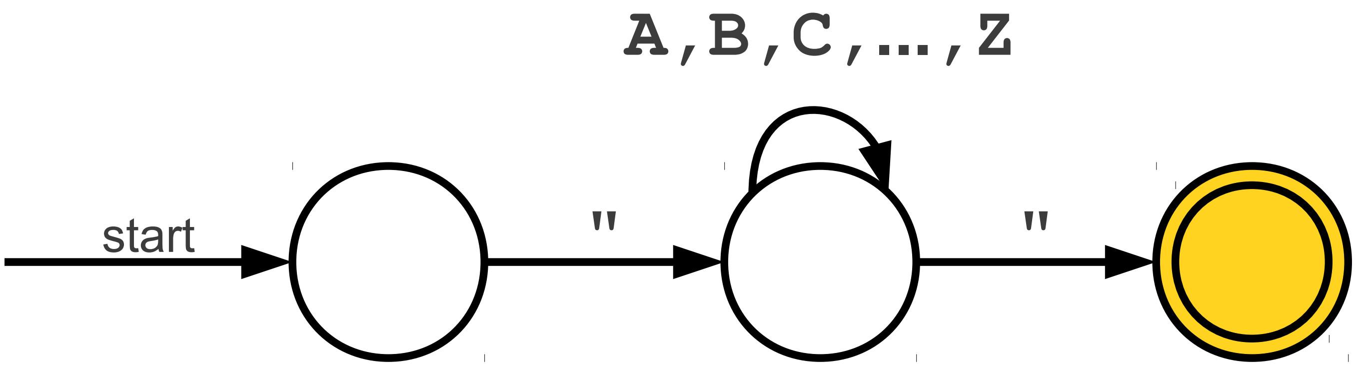


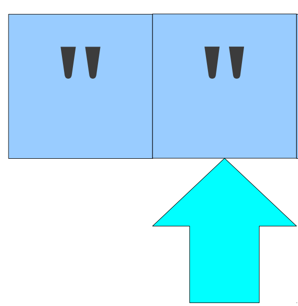
A Simple Automaton



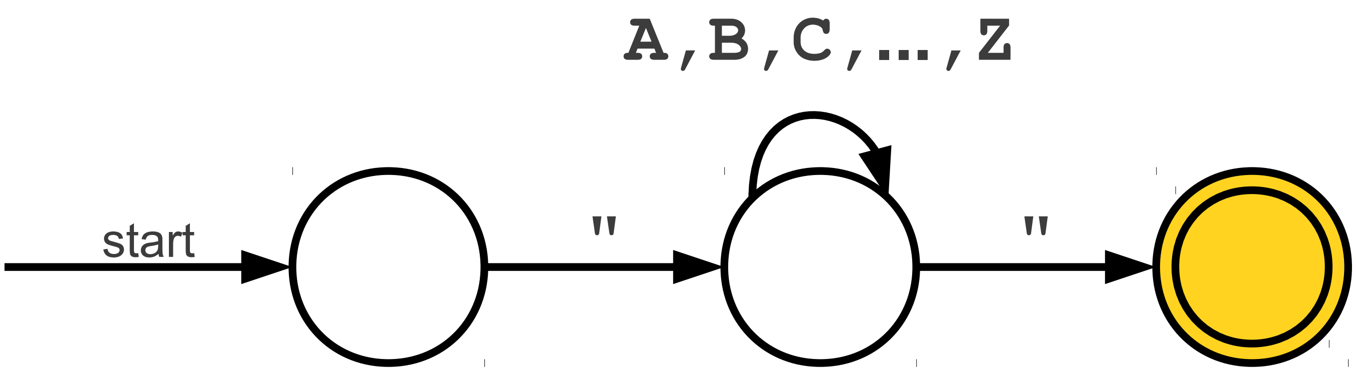


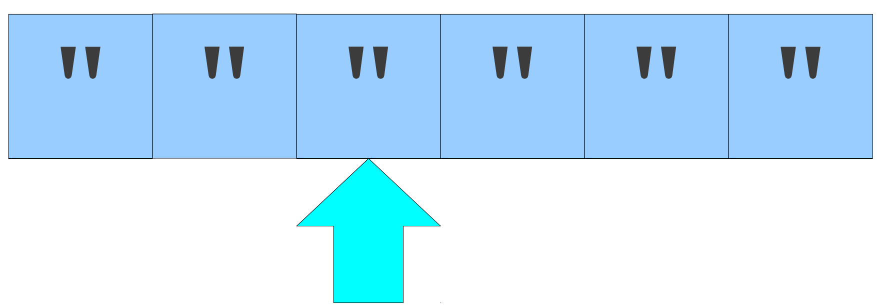
A Simple Automaton



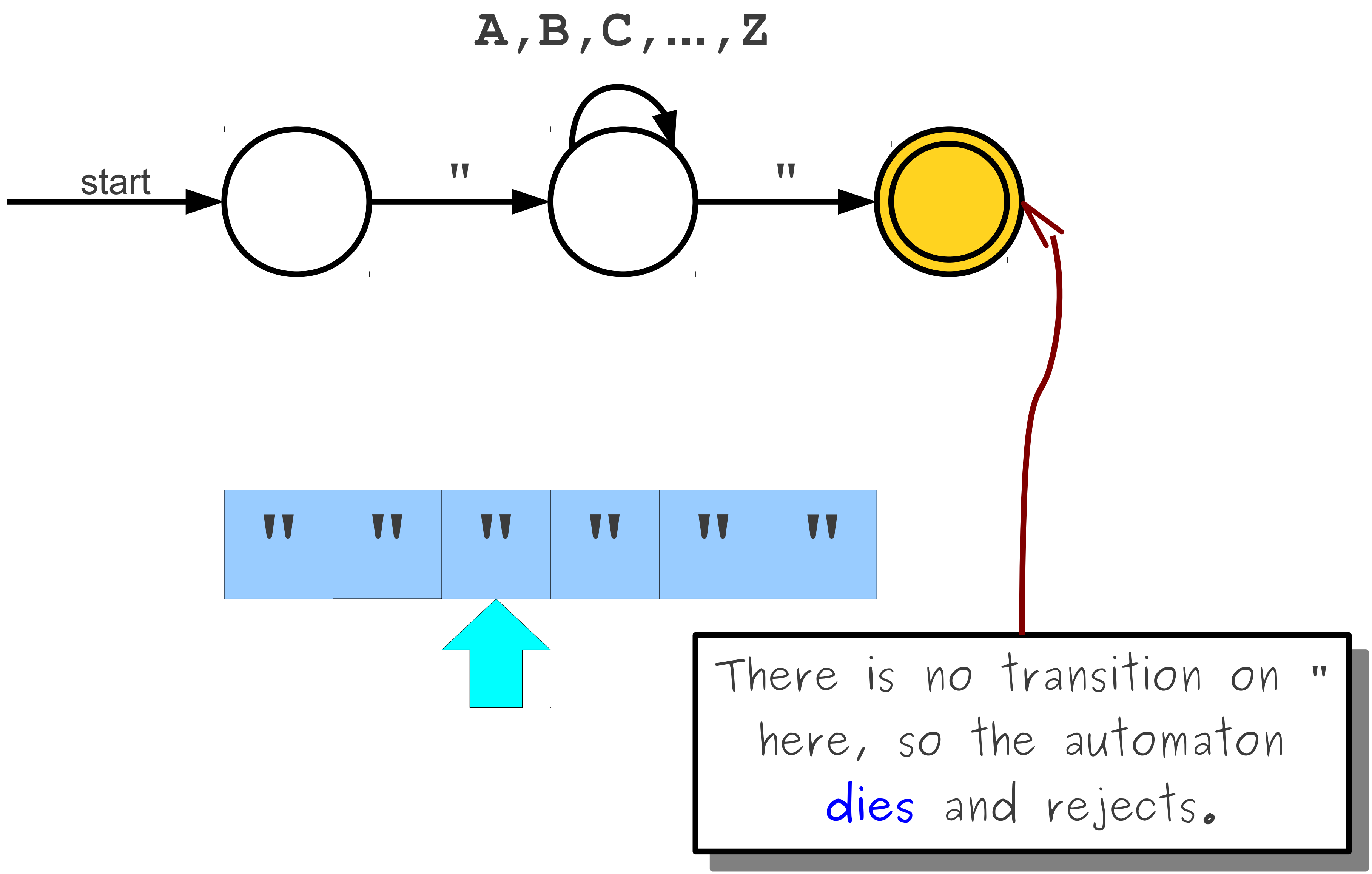


A Simple Automaton

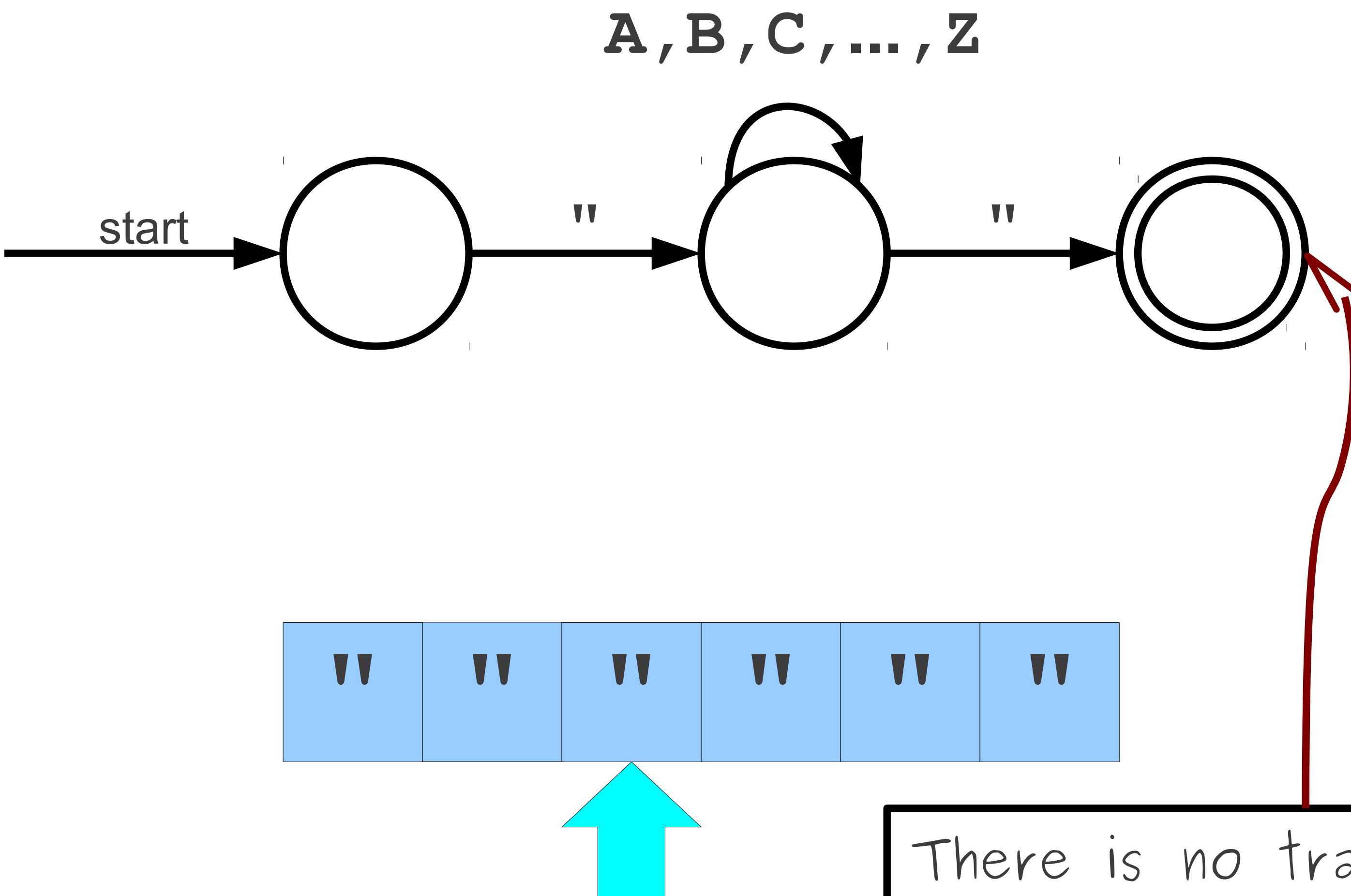




A Simple Automaton

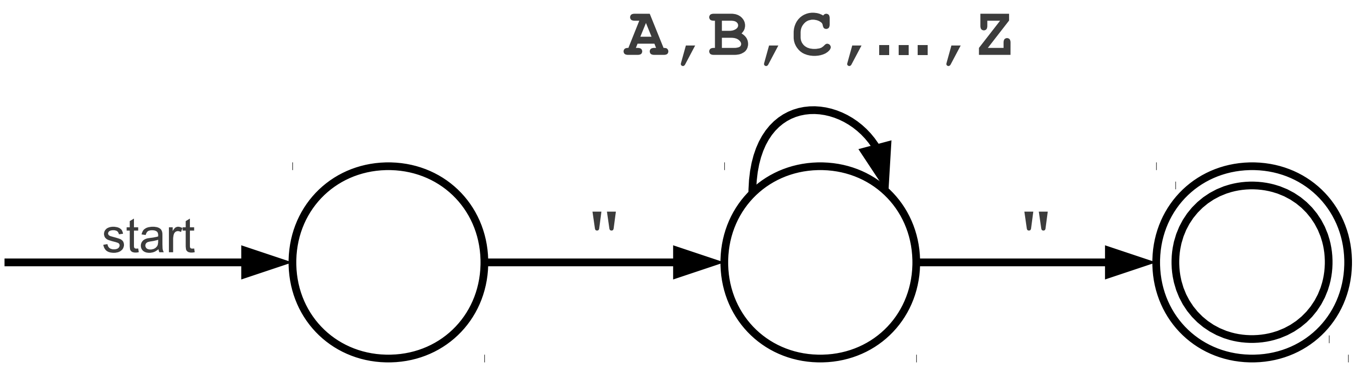


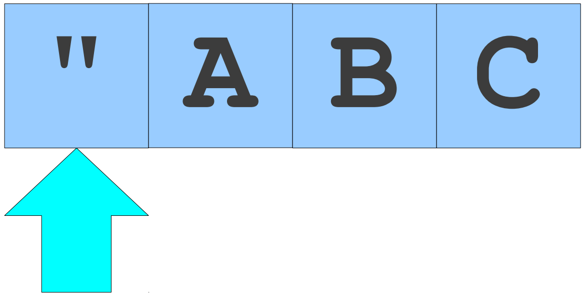
A Simple Automaton



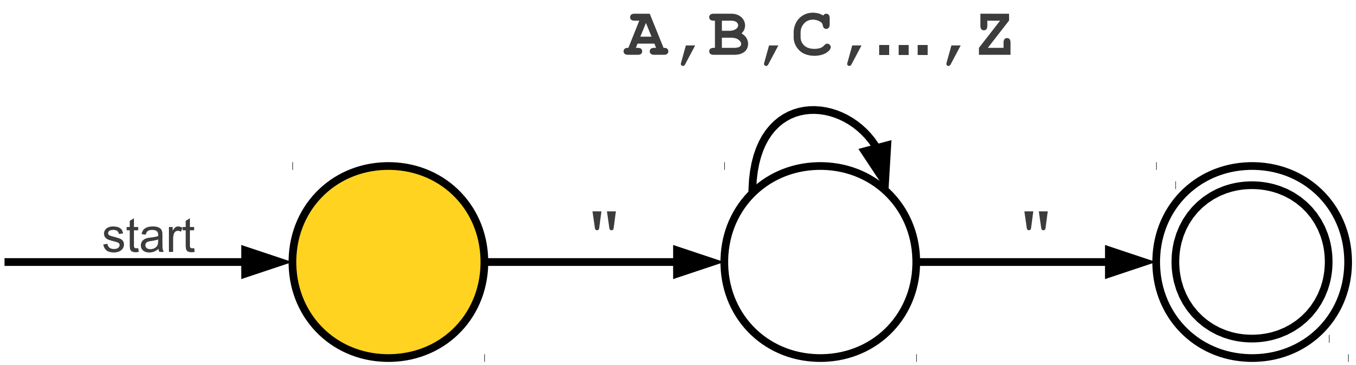
There is no transition on "
here, so the automaton
dies and rejects.

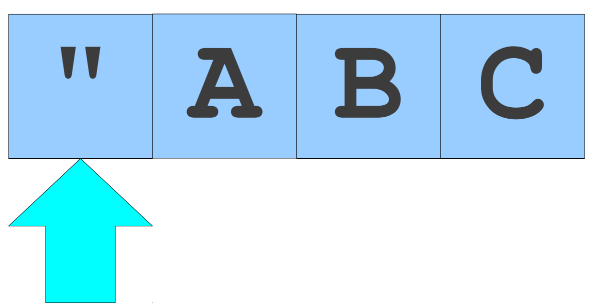
A Simple Automaton



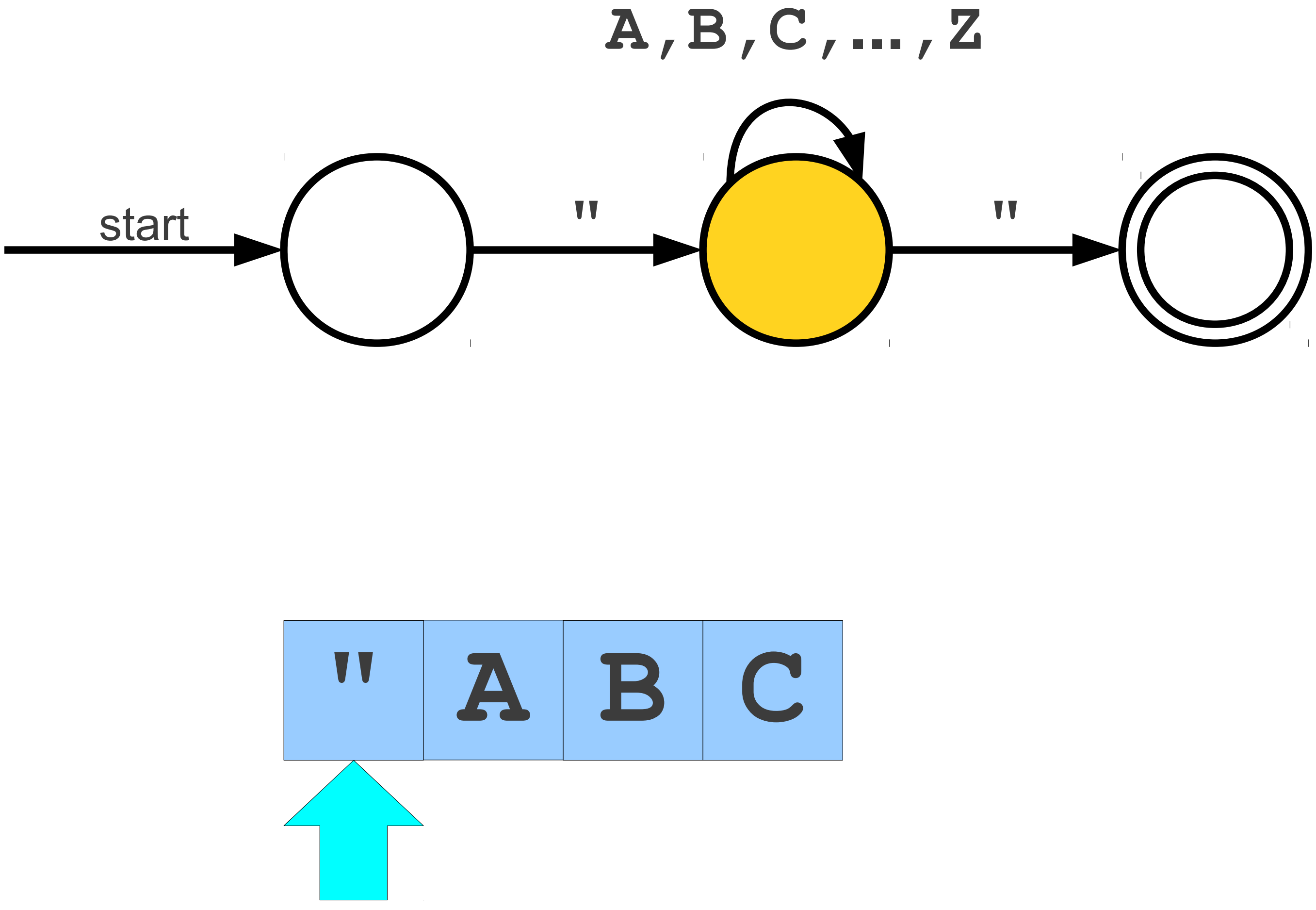


A Simple Automaton

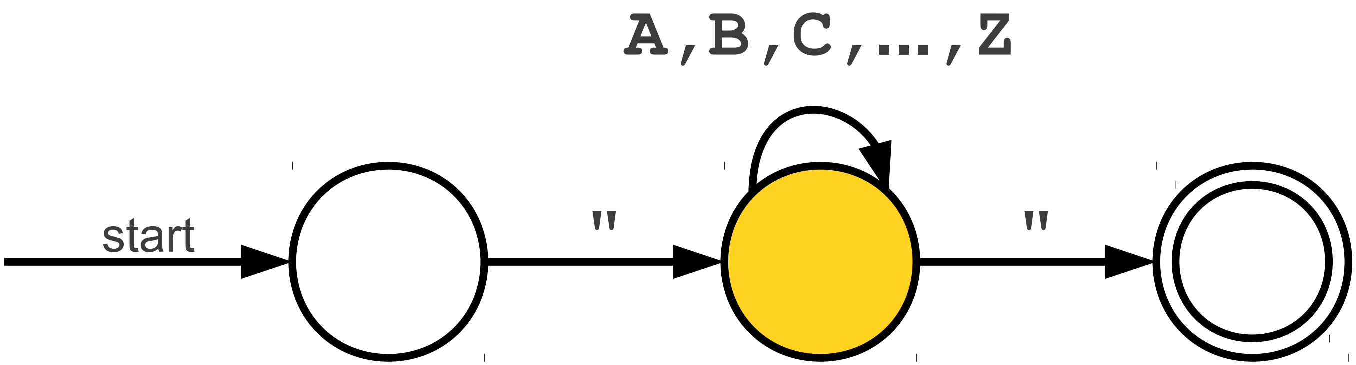


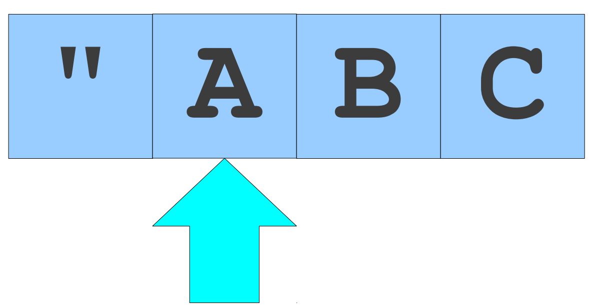


A Simple Automaton

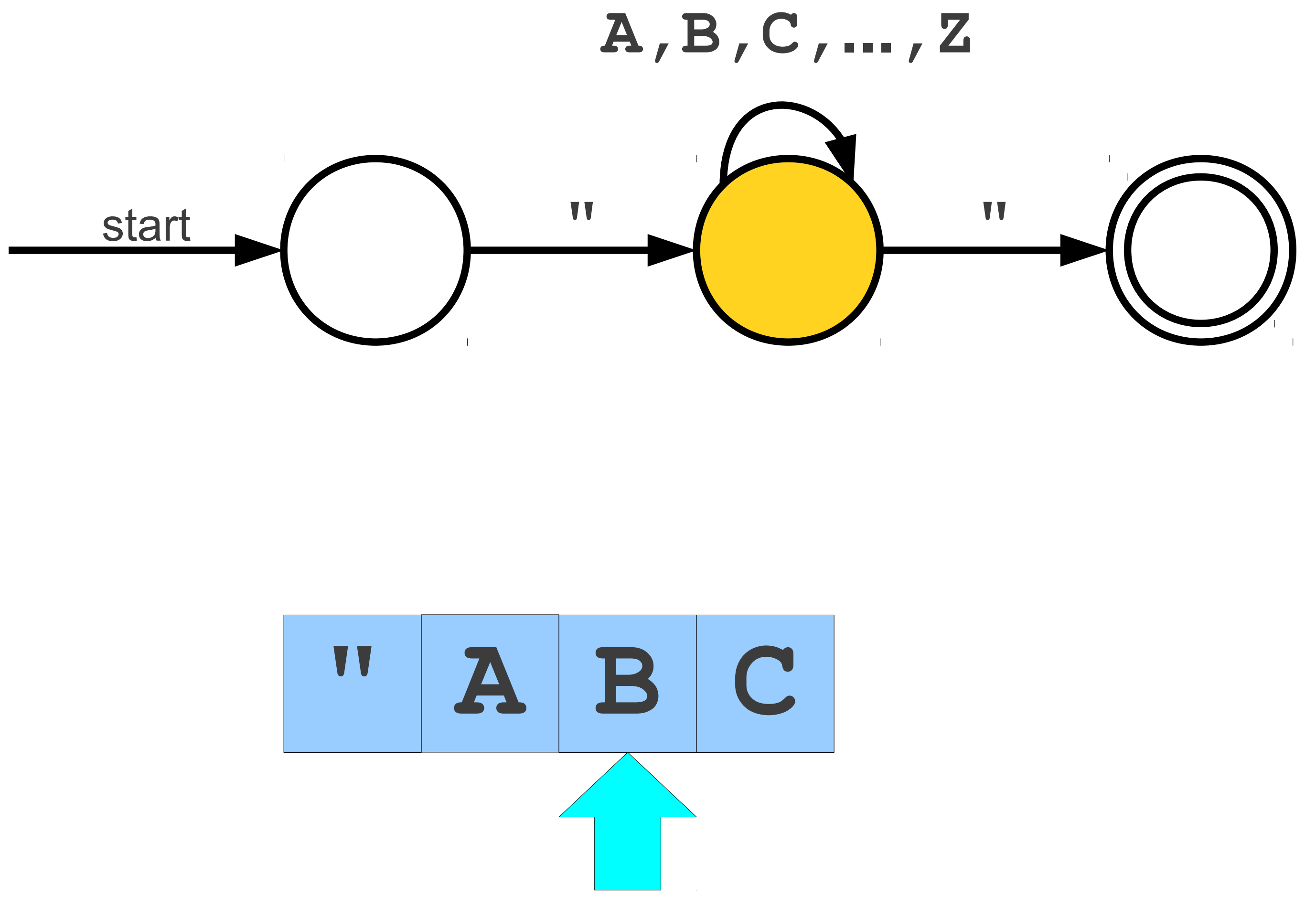


A Simple Automaton

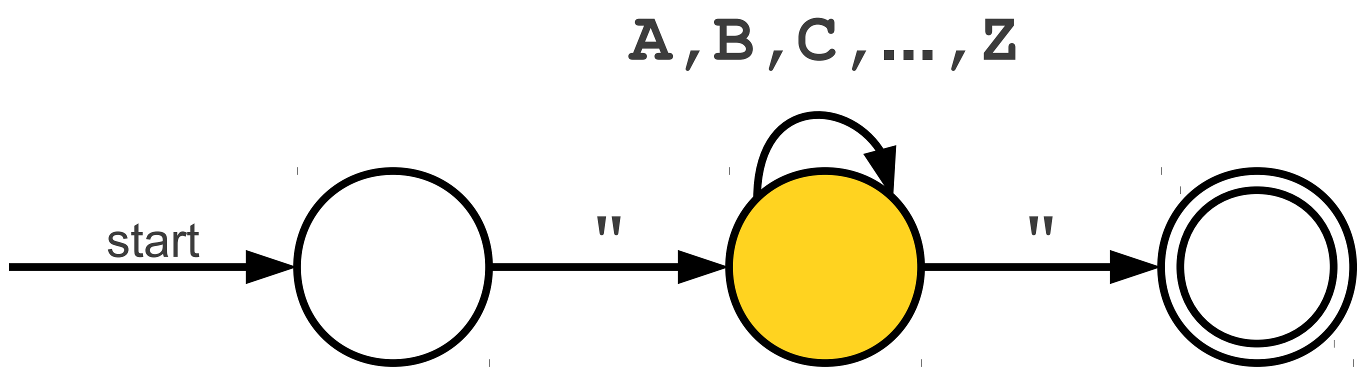


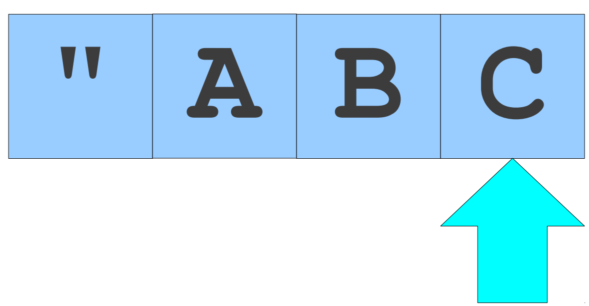


A Simple Automaton

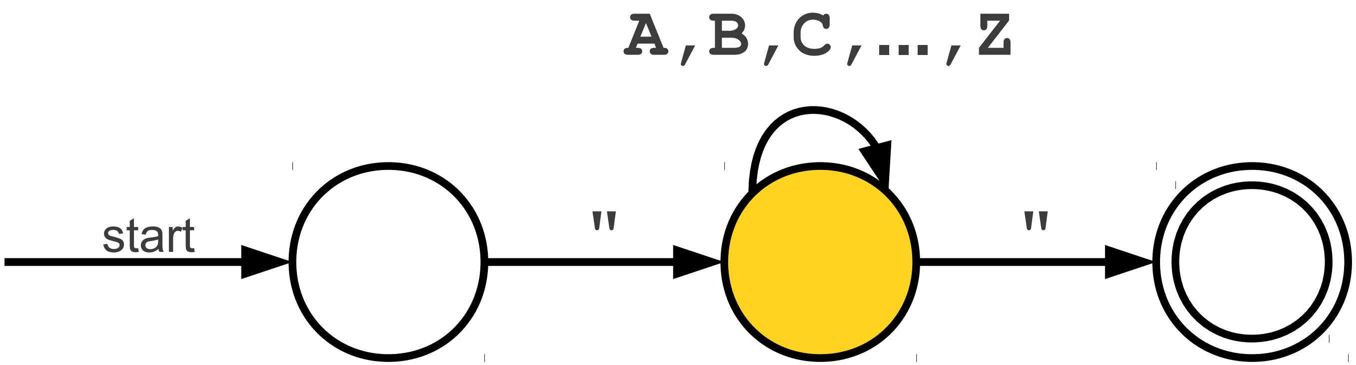


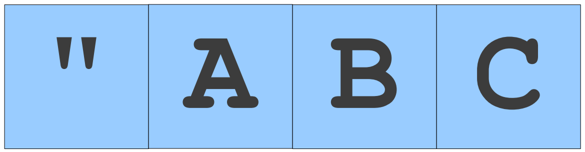
A Simple Automaton



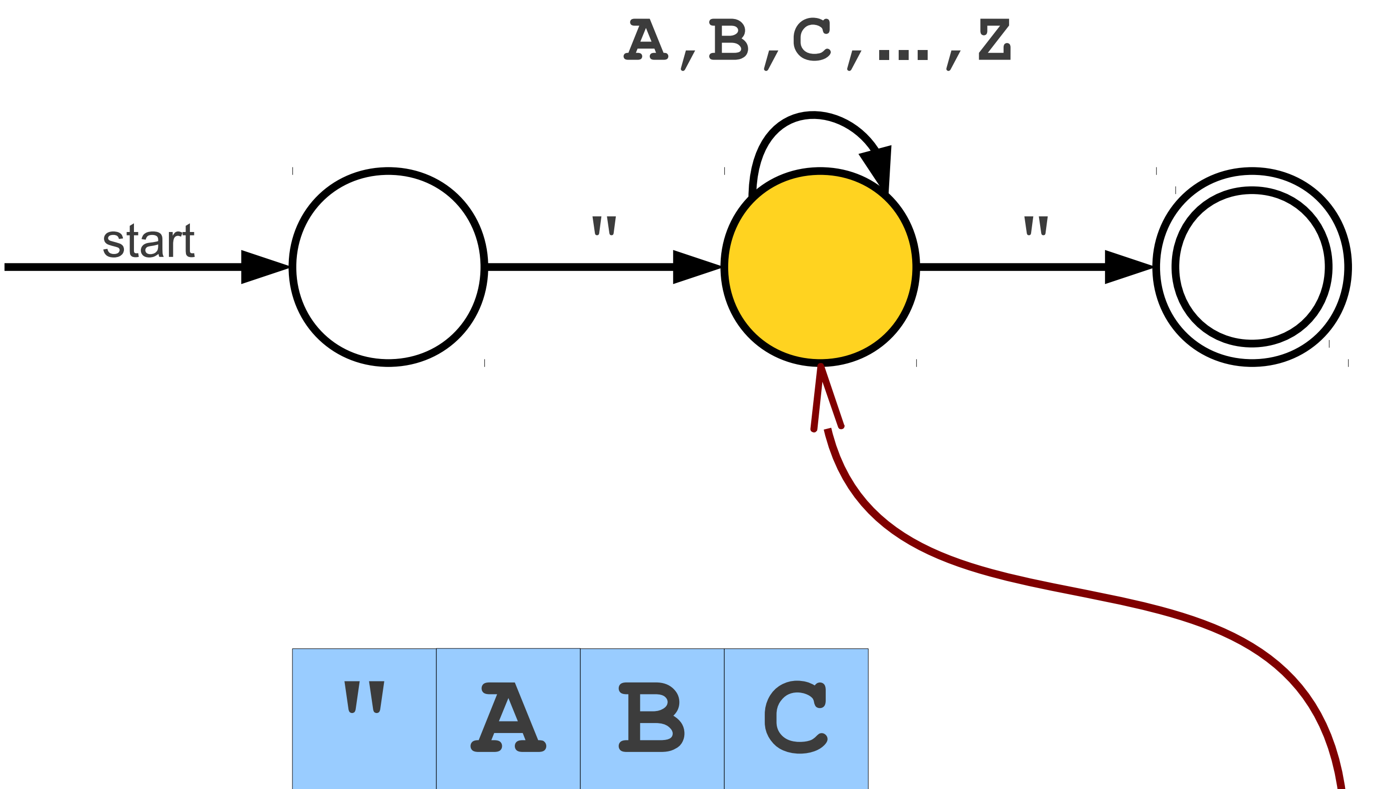


A Simple Automaton



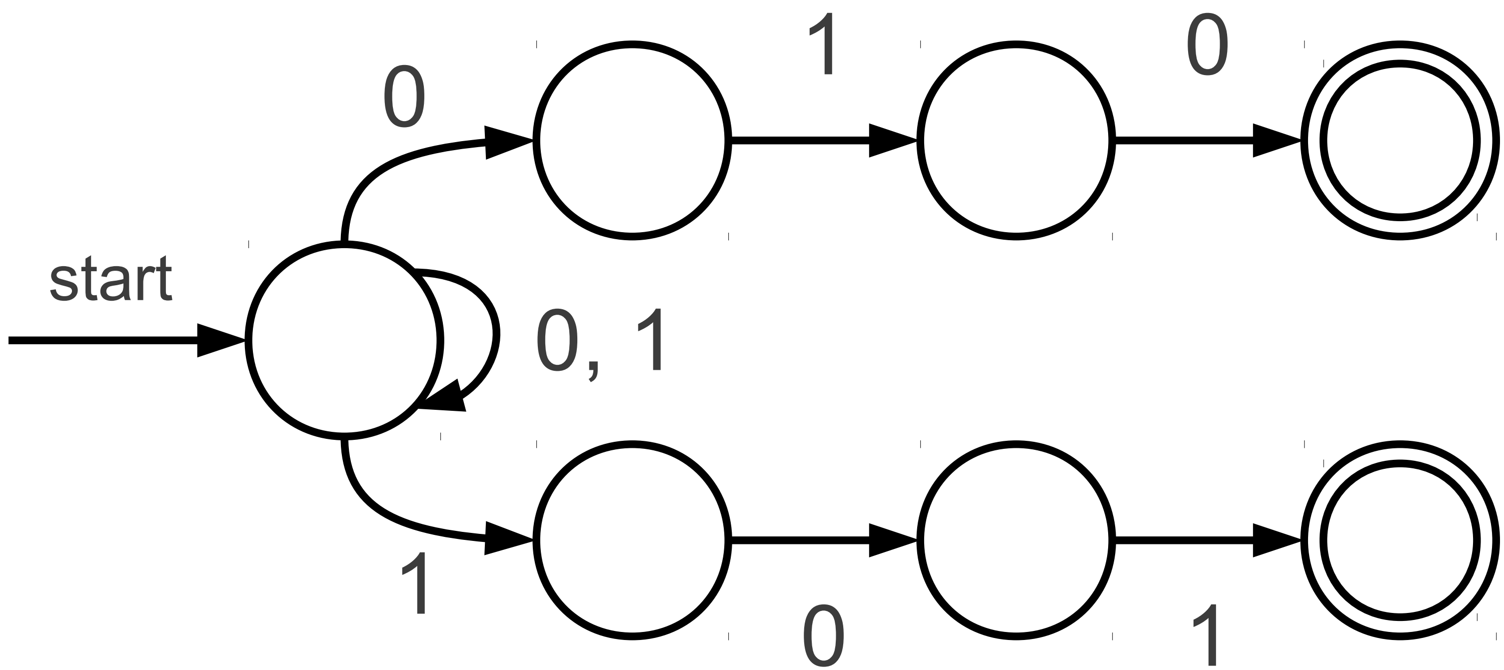


A Simple Automaton

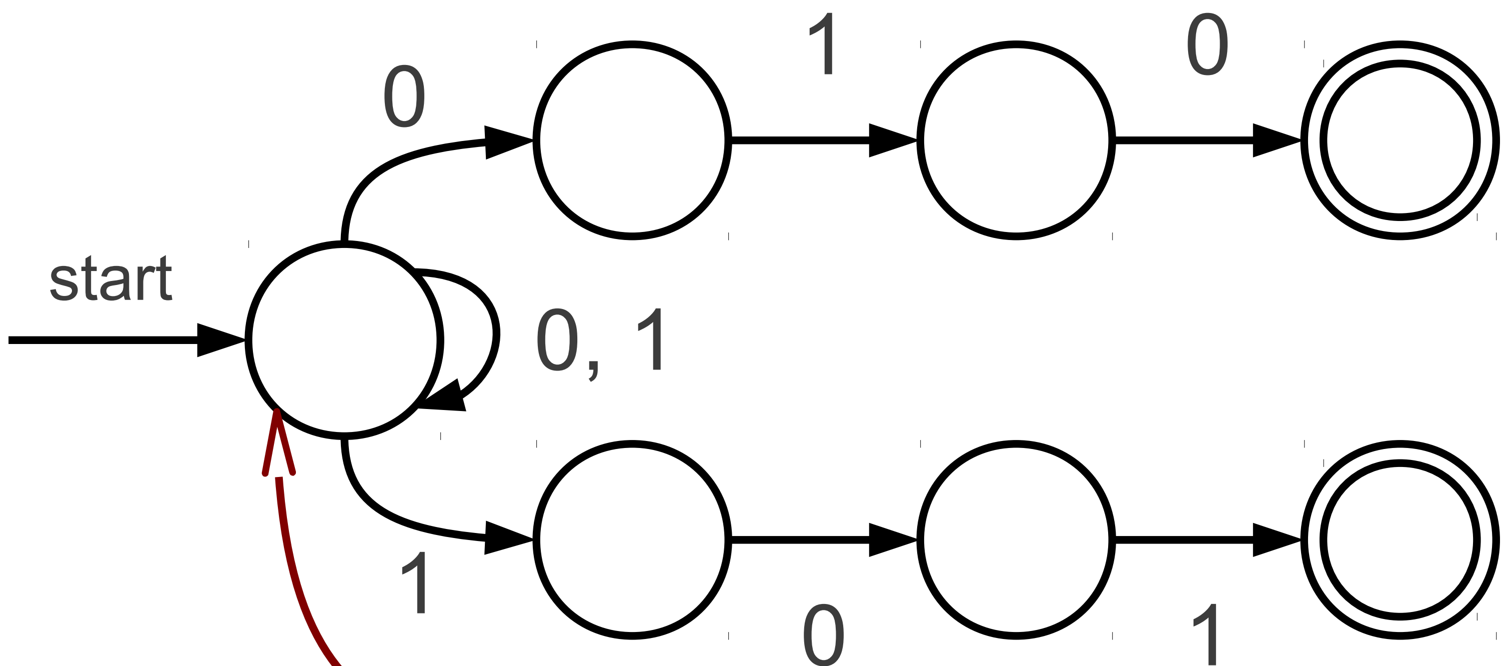


This is not an accepting state, so the automaton rejects.

A More Complex Automaton

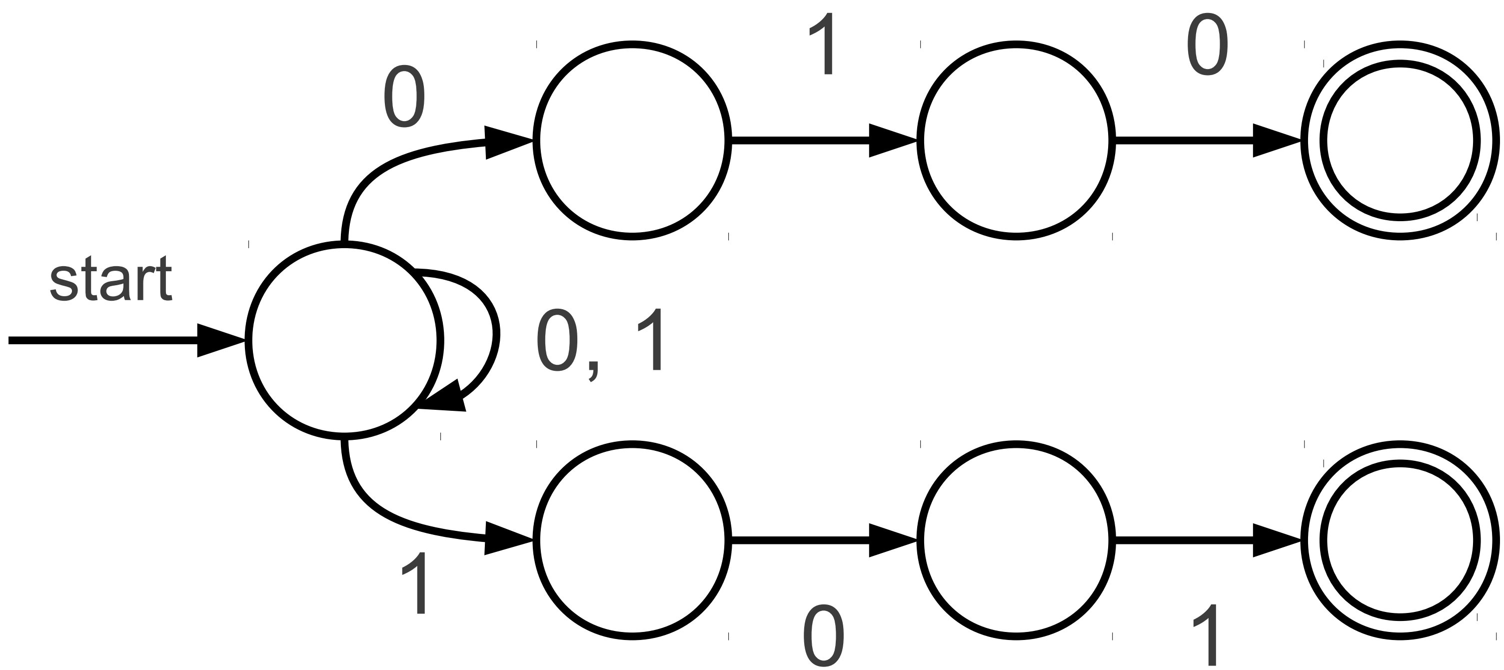


A More Complex Automaton

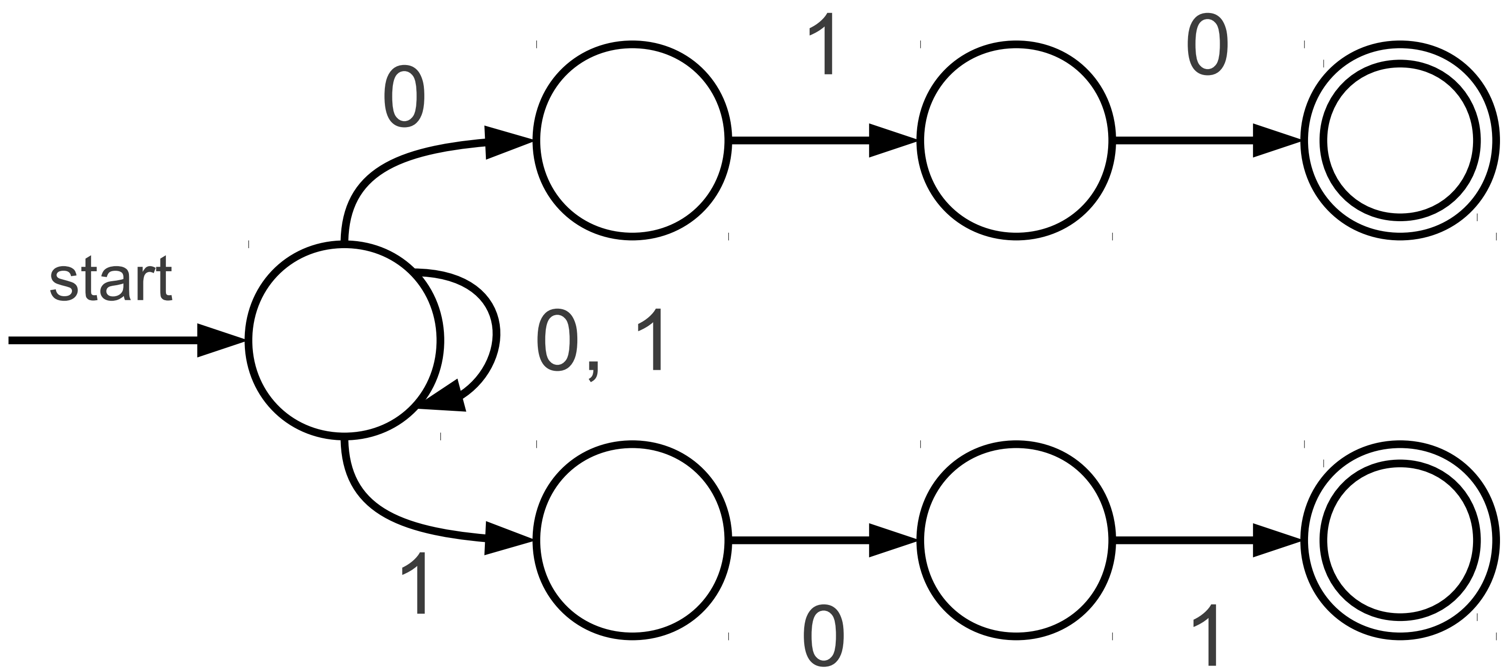


Notice that there are multiple transitions defined here on *0* and *1*. If we read a *0* or *1* here, we follow *both* transitions and enter multiple states.

A More Complex Automaton

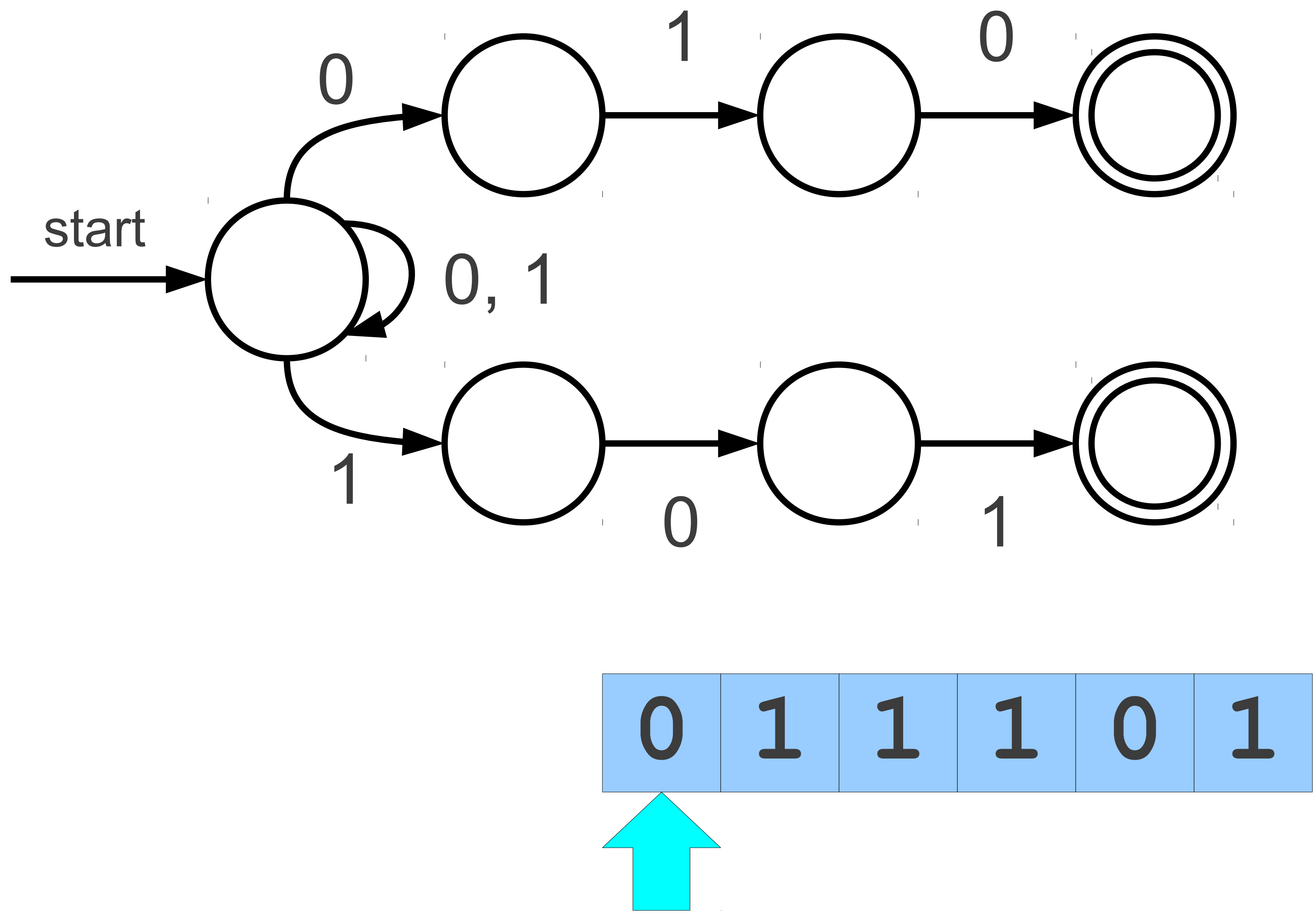


A More Complex Automaton

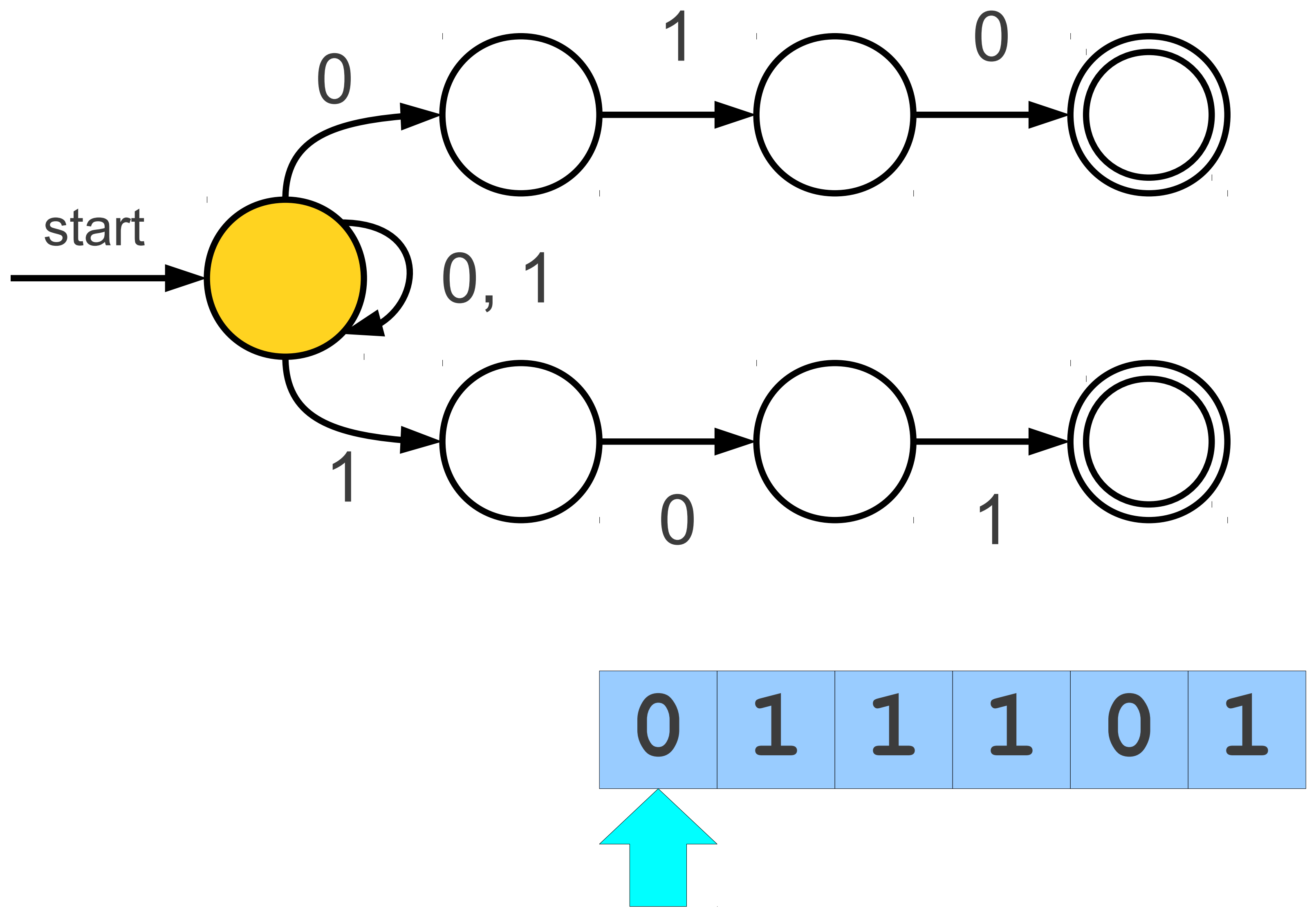


0	1	1	1	0	1

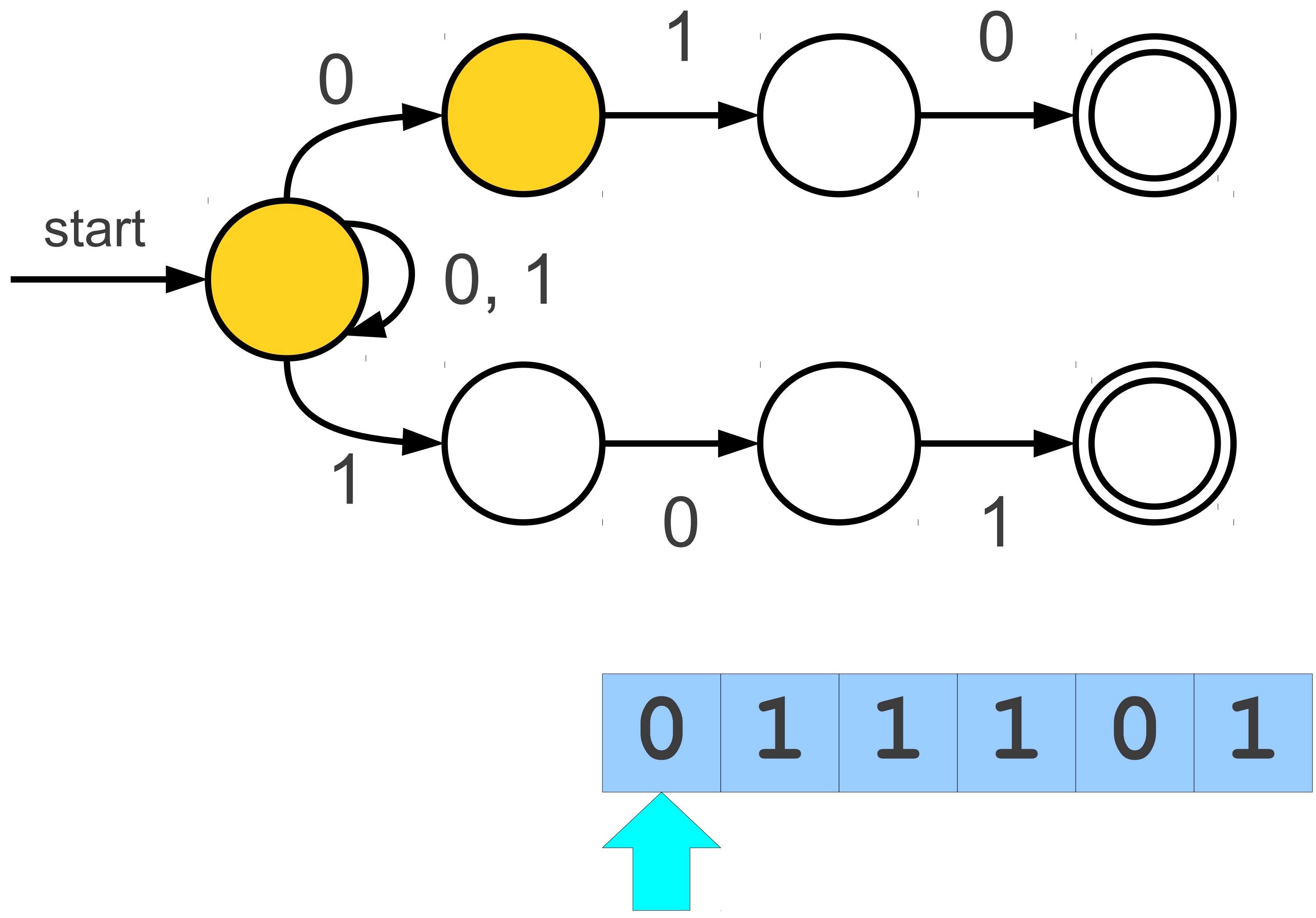
A More Complex Automaton



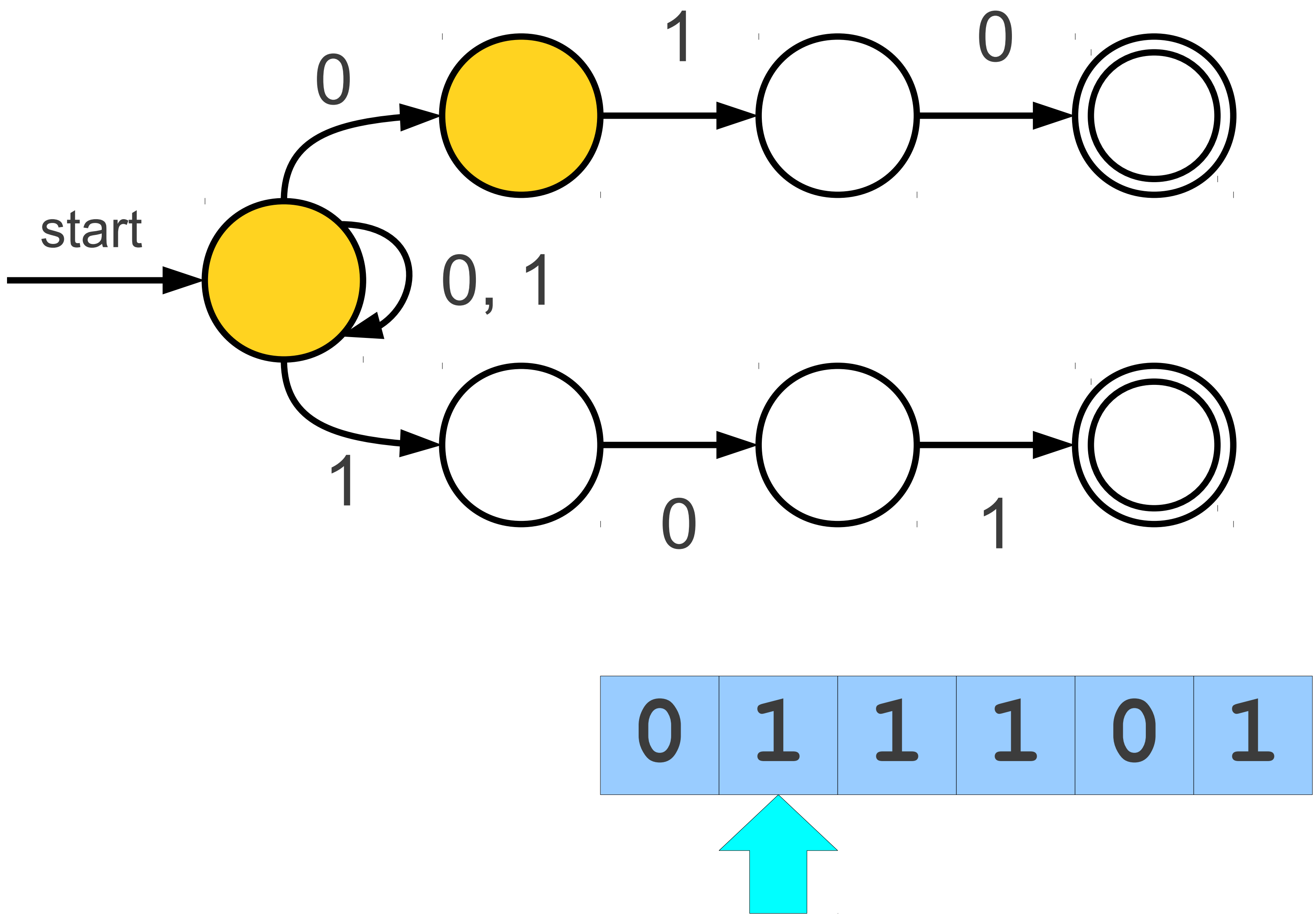
A More Complex Automaton



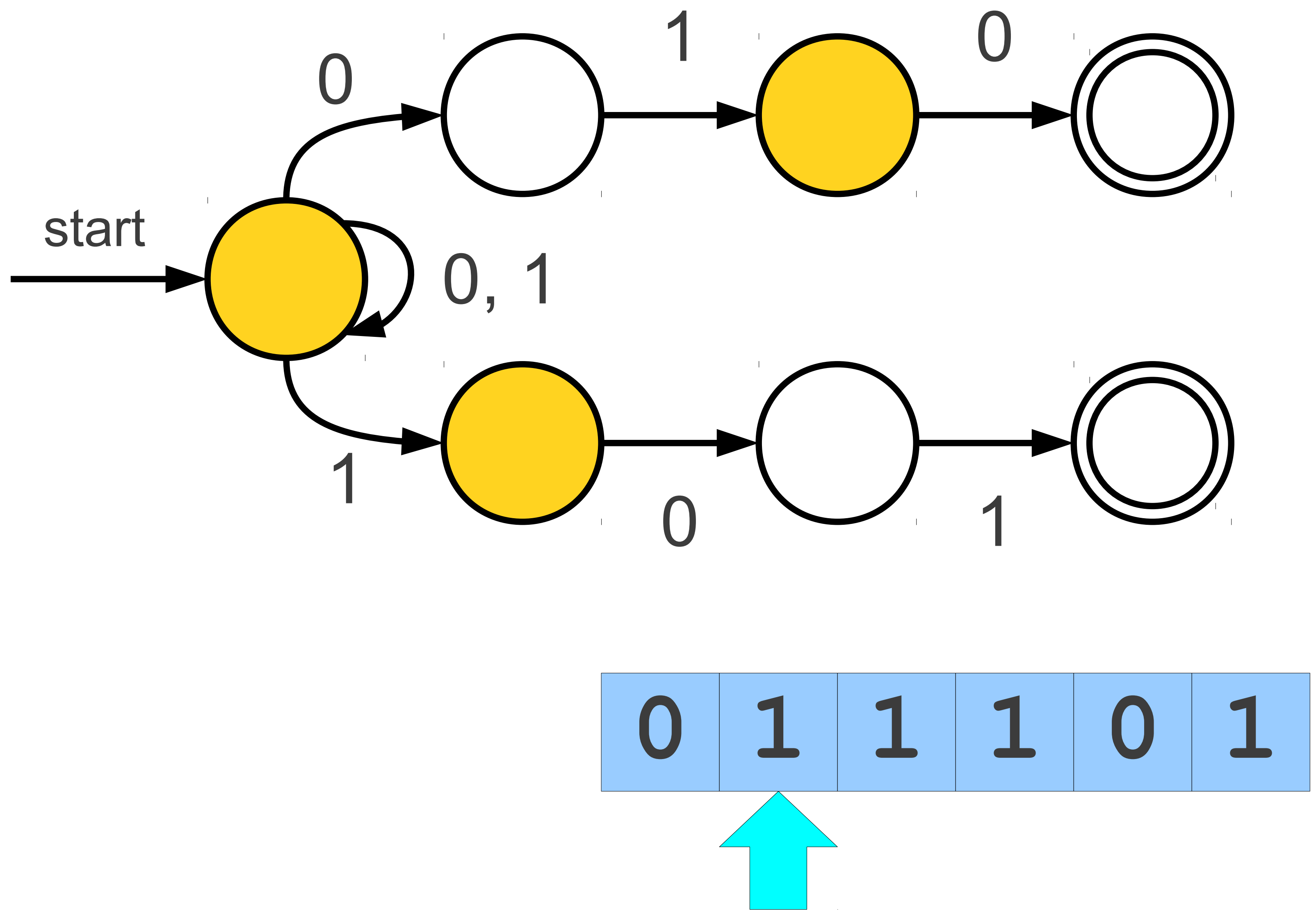
A More Complex Automaton



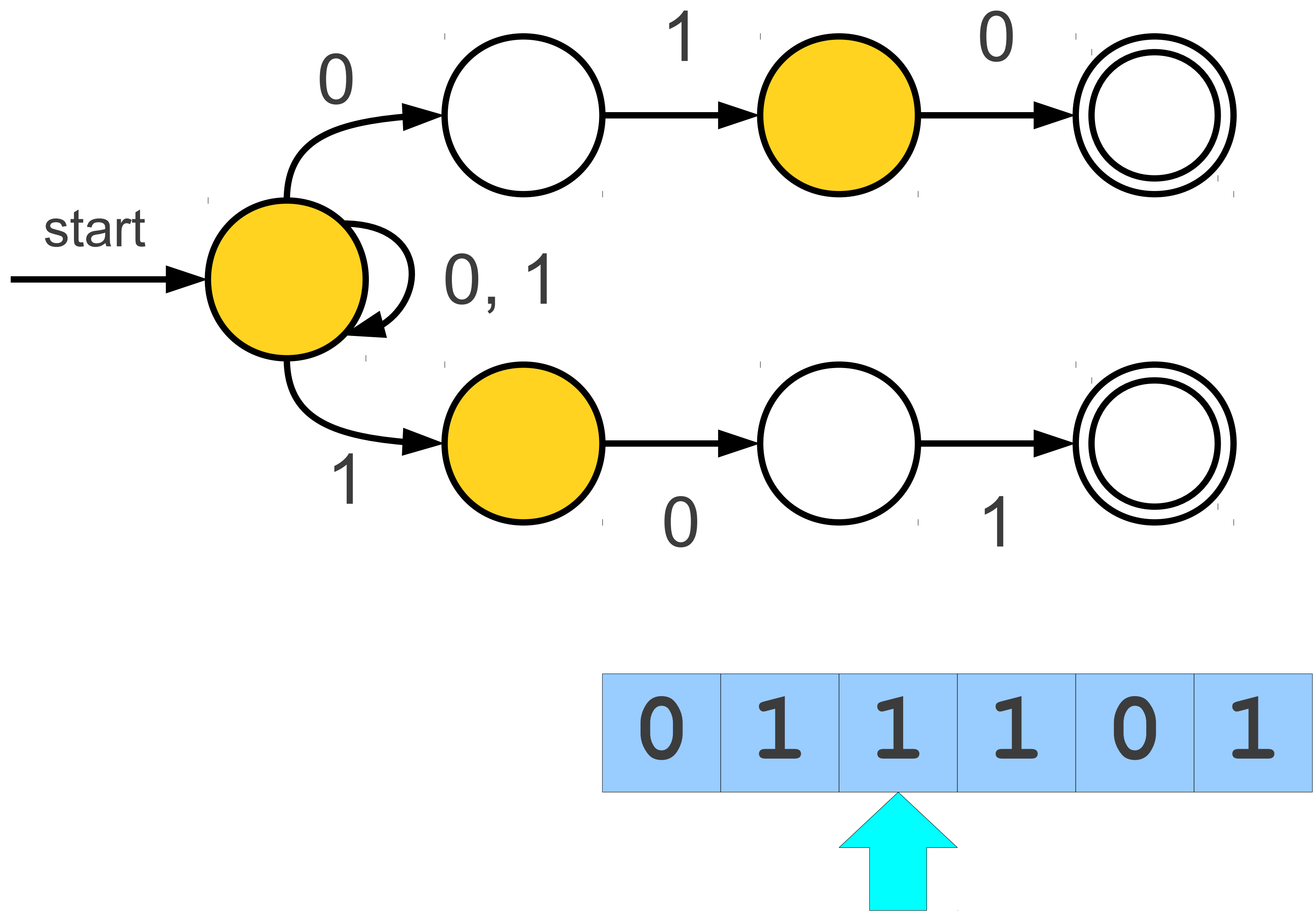
A More Complex Automaton



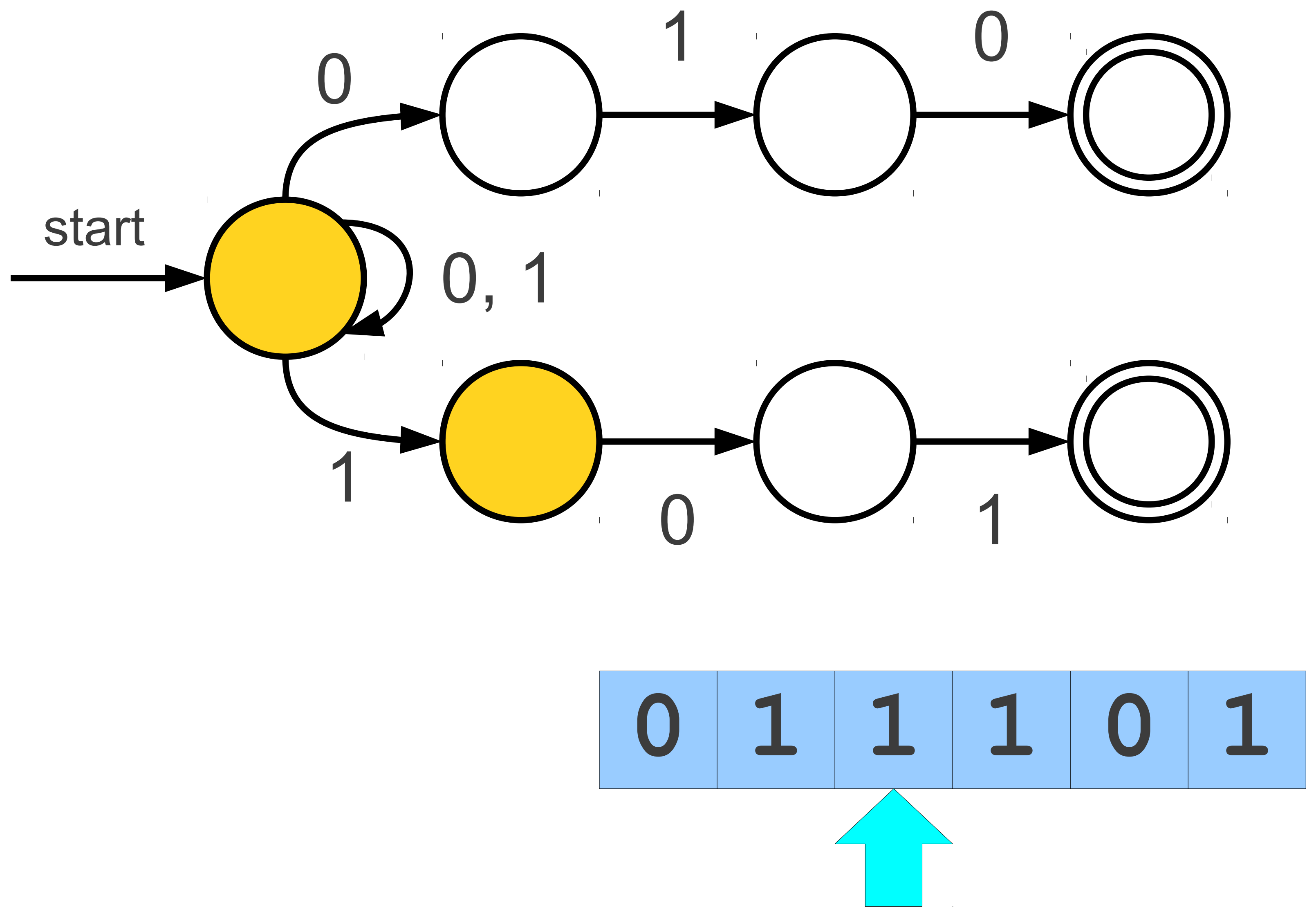
A More Complex Automaton



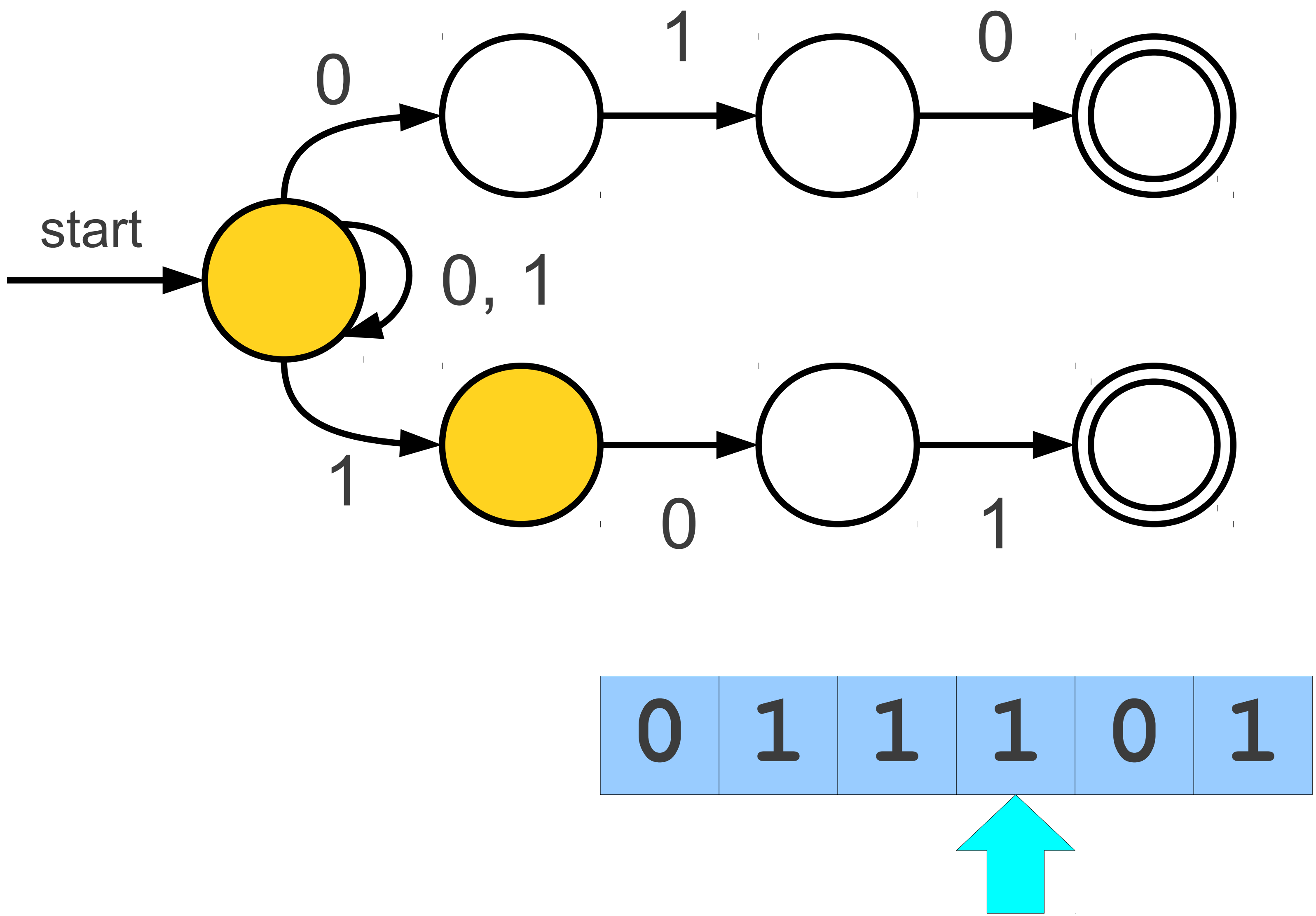
A More Complex Automaton



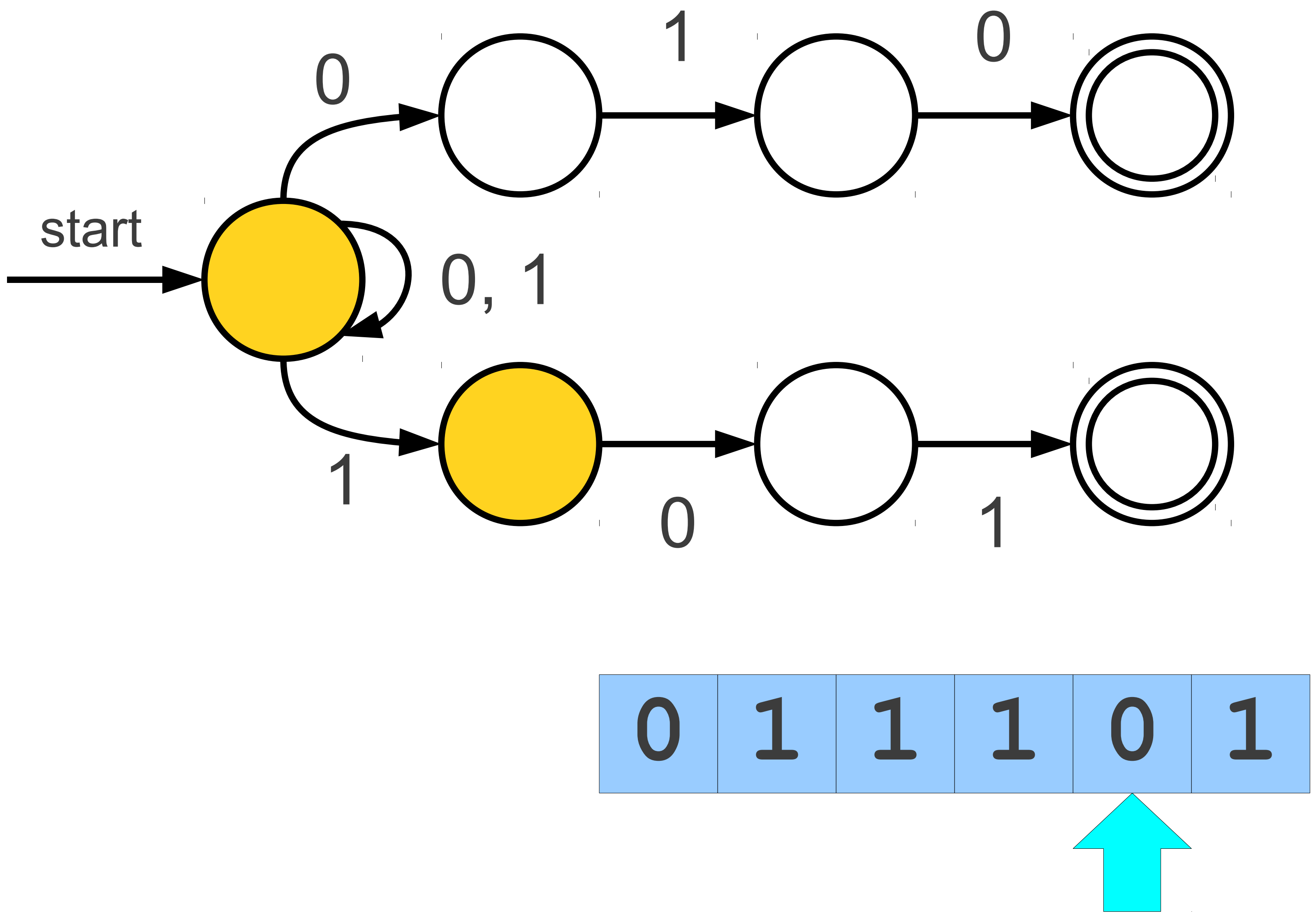
A More Complex Automaton



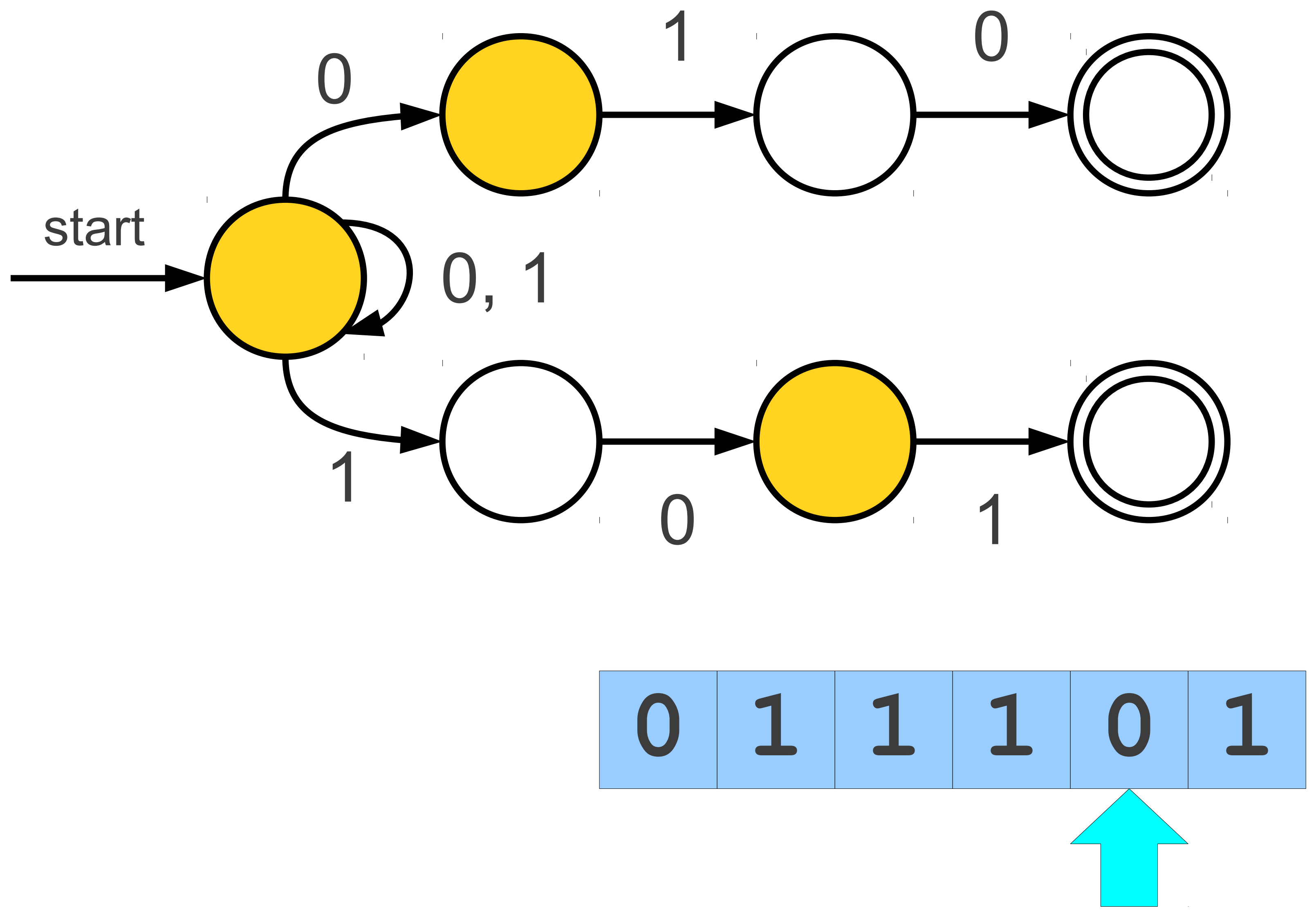
A More Complex Automaton



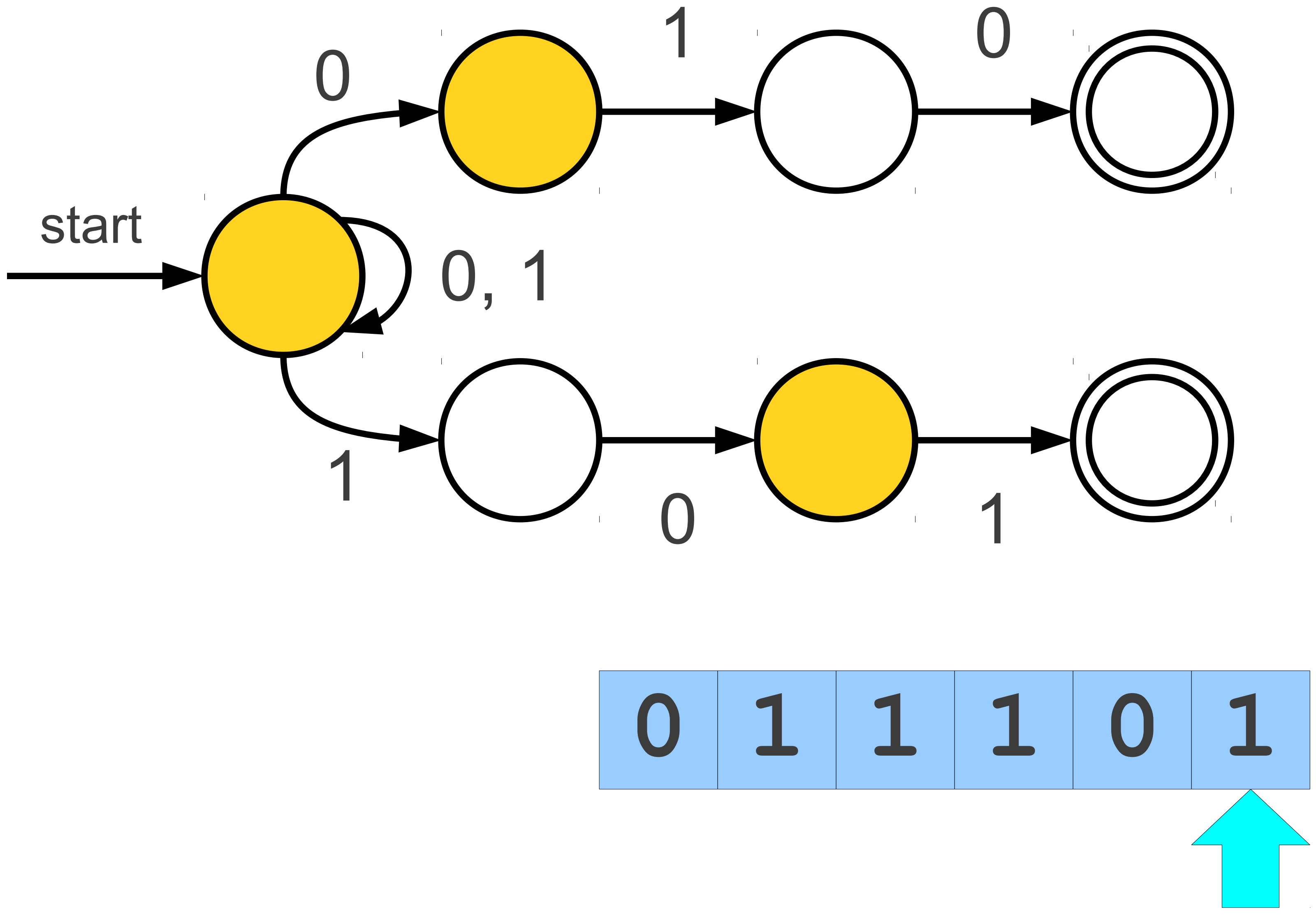
A More Complex Automaton



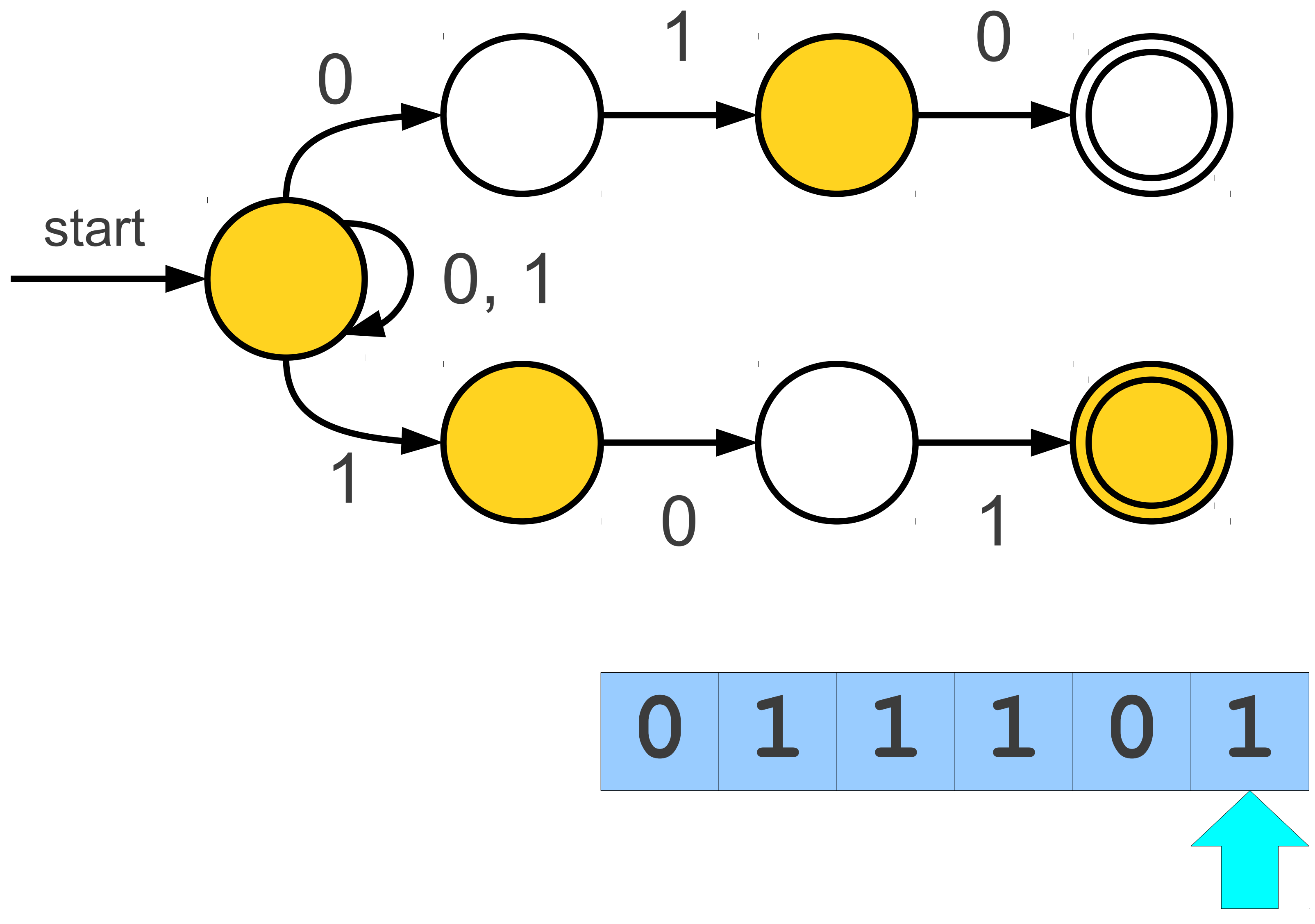
A More Complex Automaton



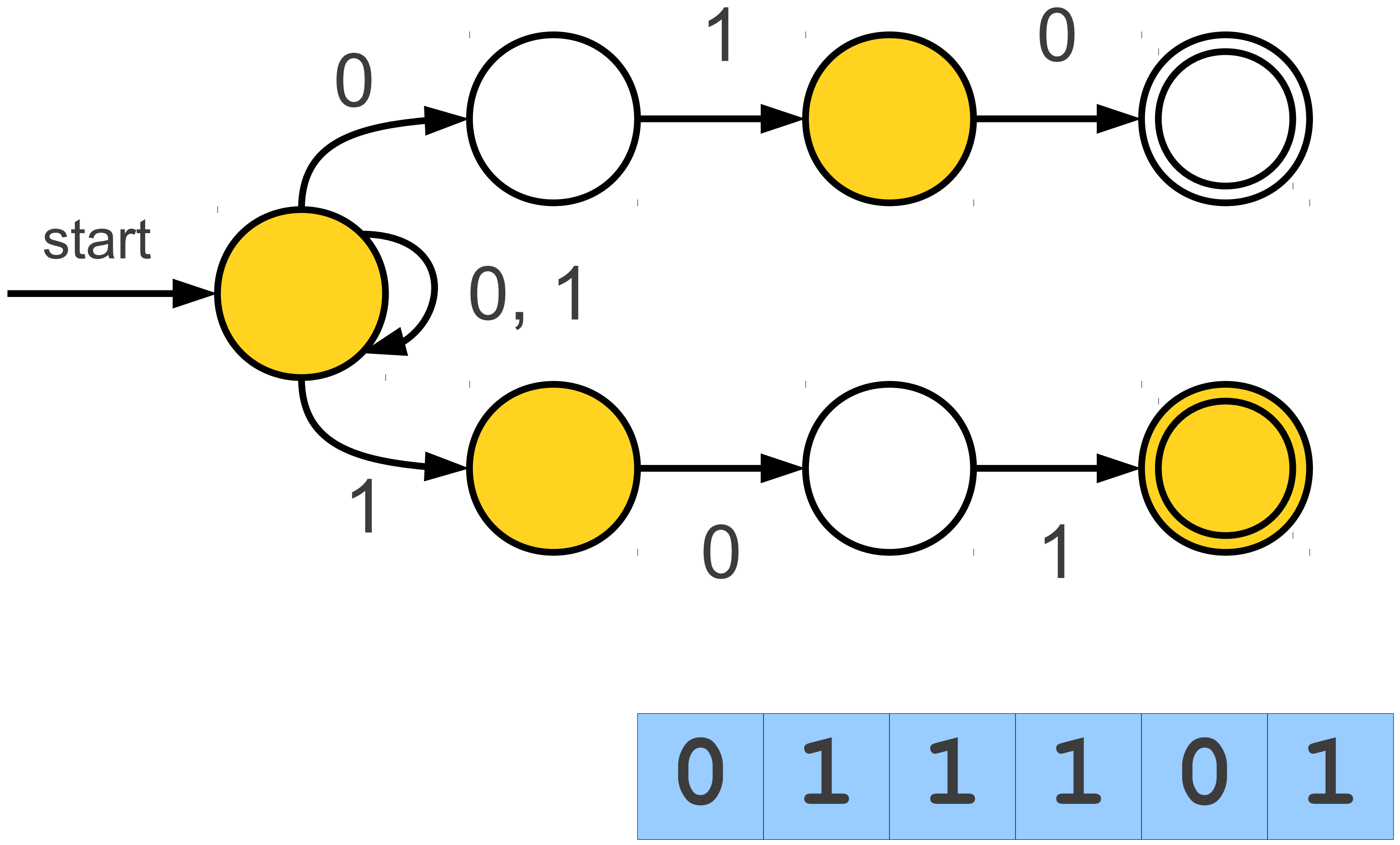
A More Complex Automaton



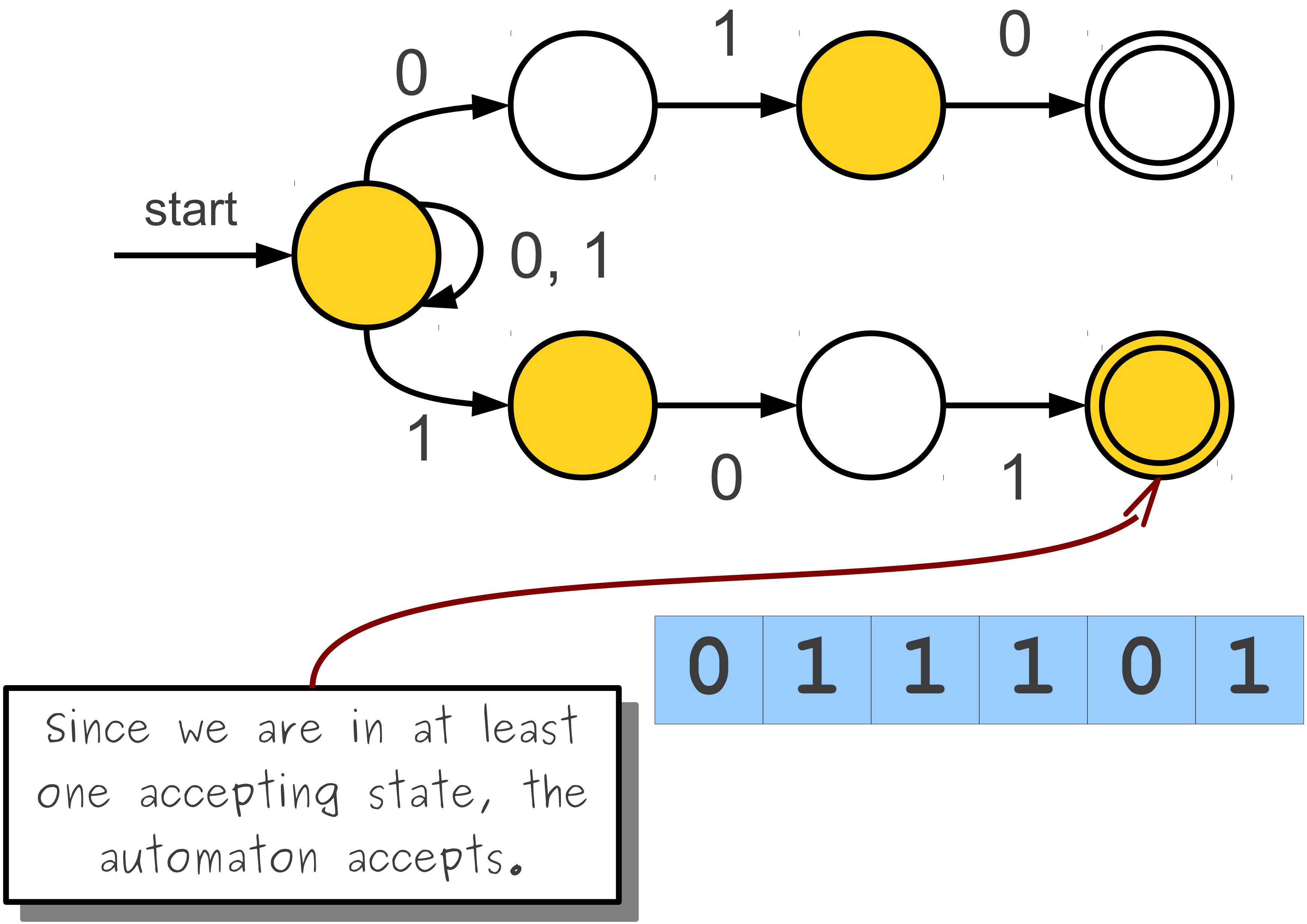
A More Complex Automaton



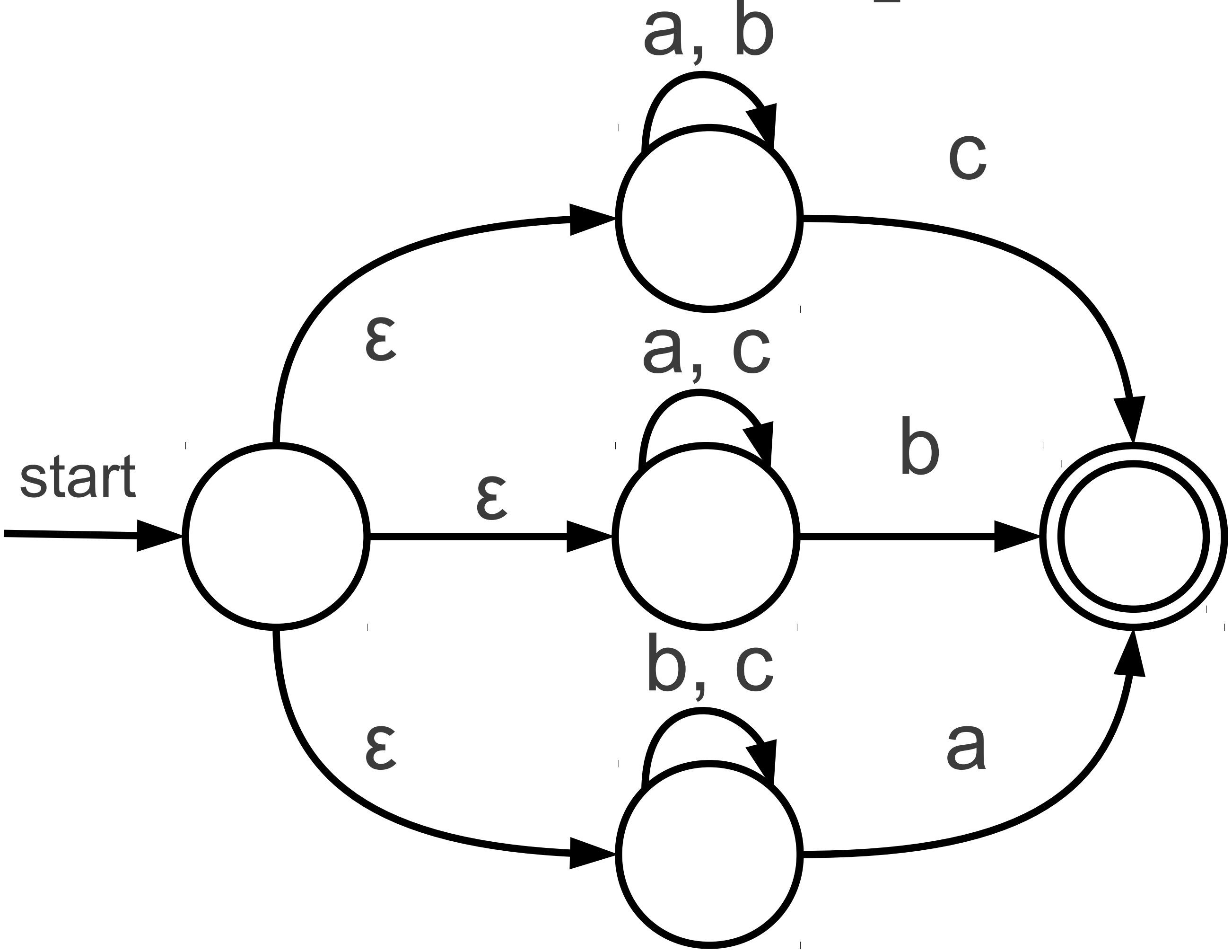
A More Complex Automaton



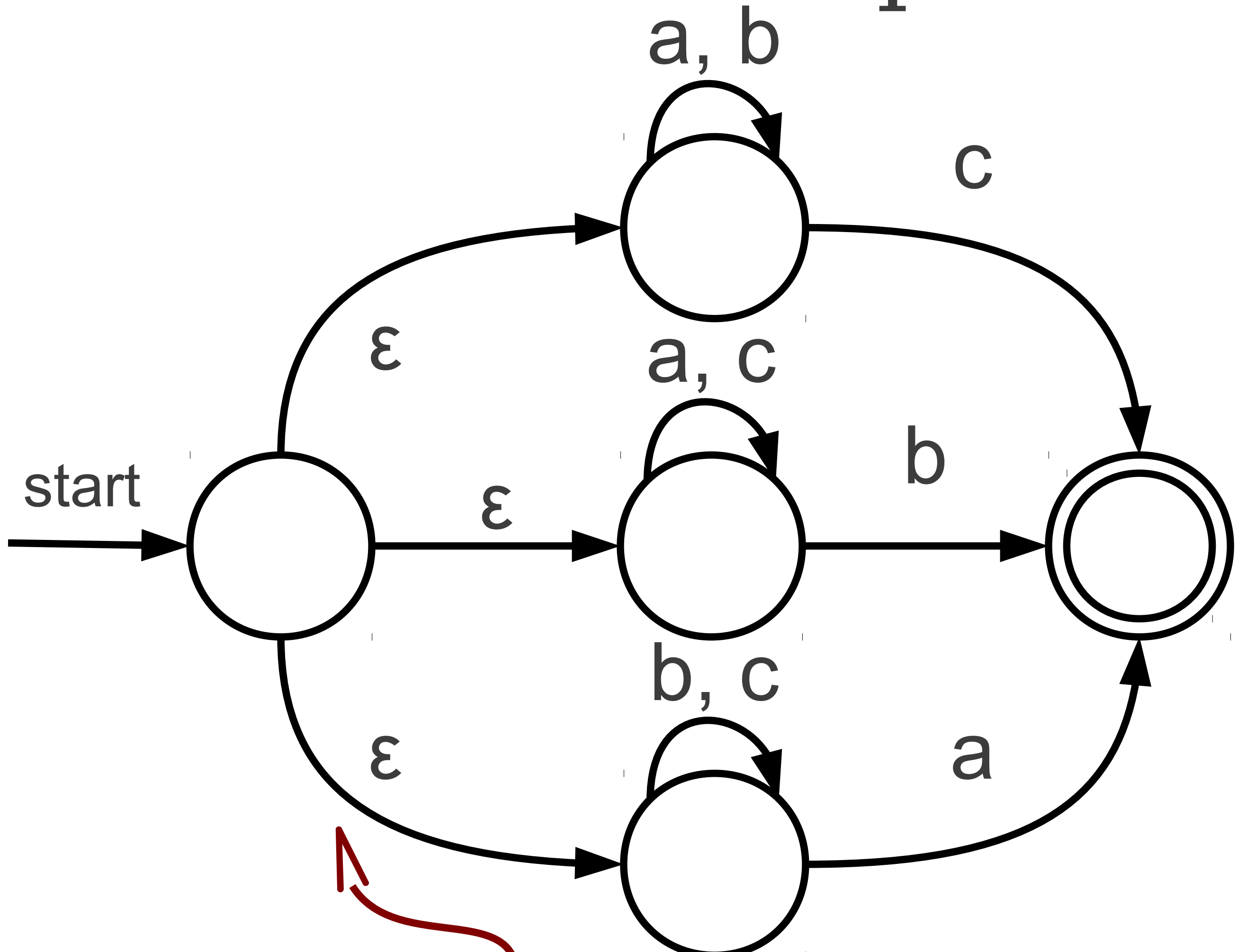
A More Complex Automaton



An Even More Complex Automaton

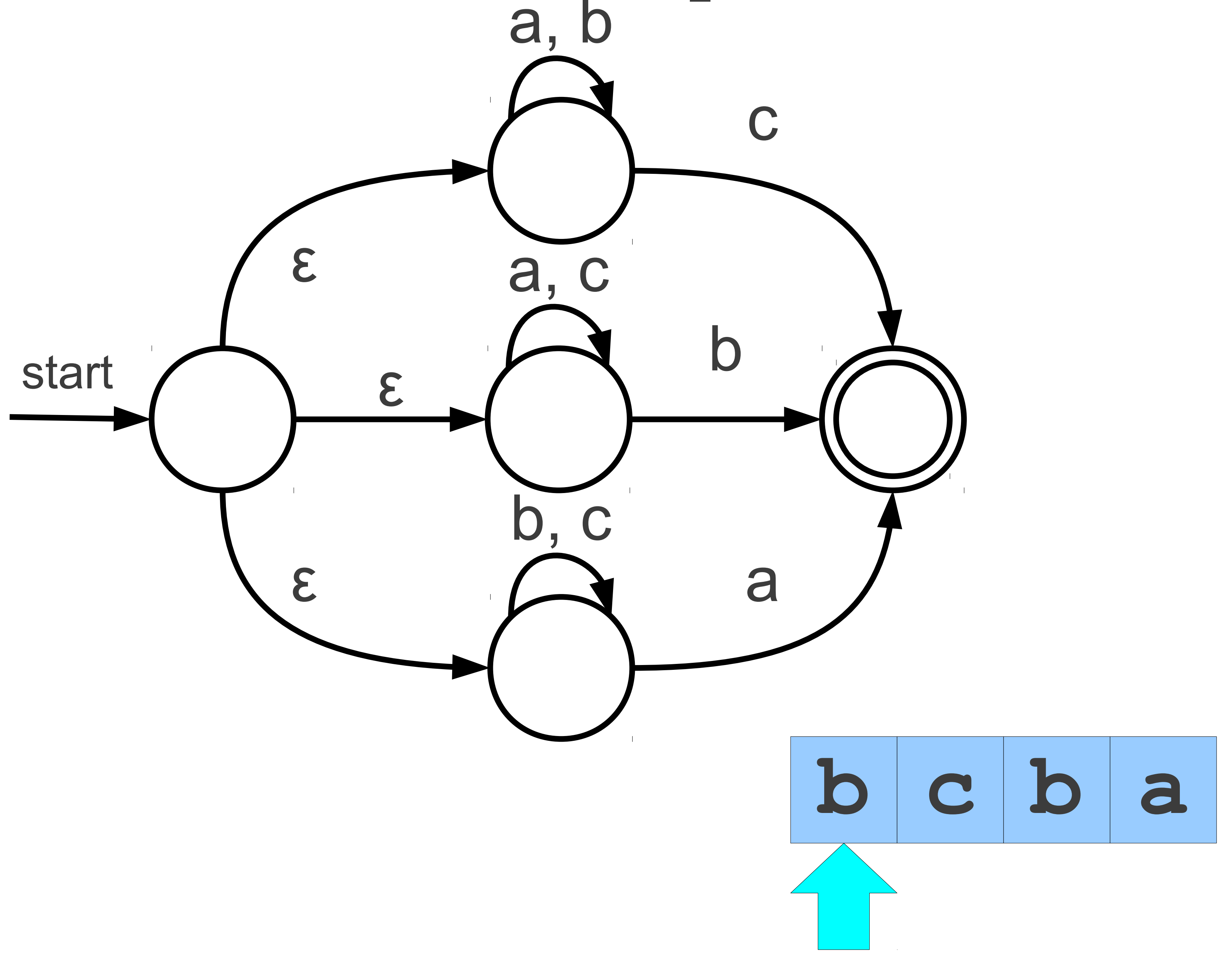


An Even More Complex Automaton

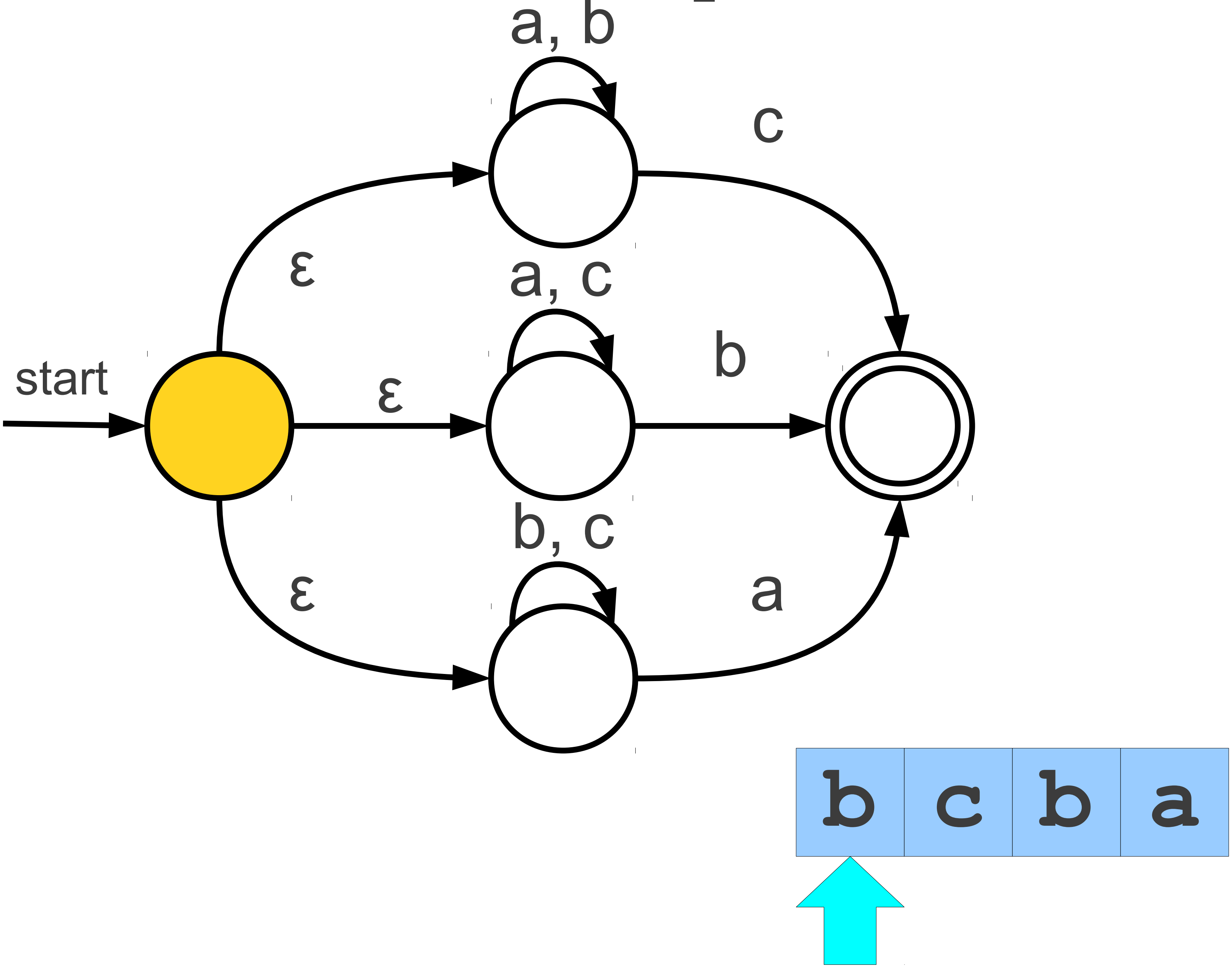


These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

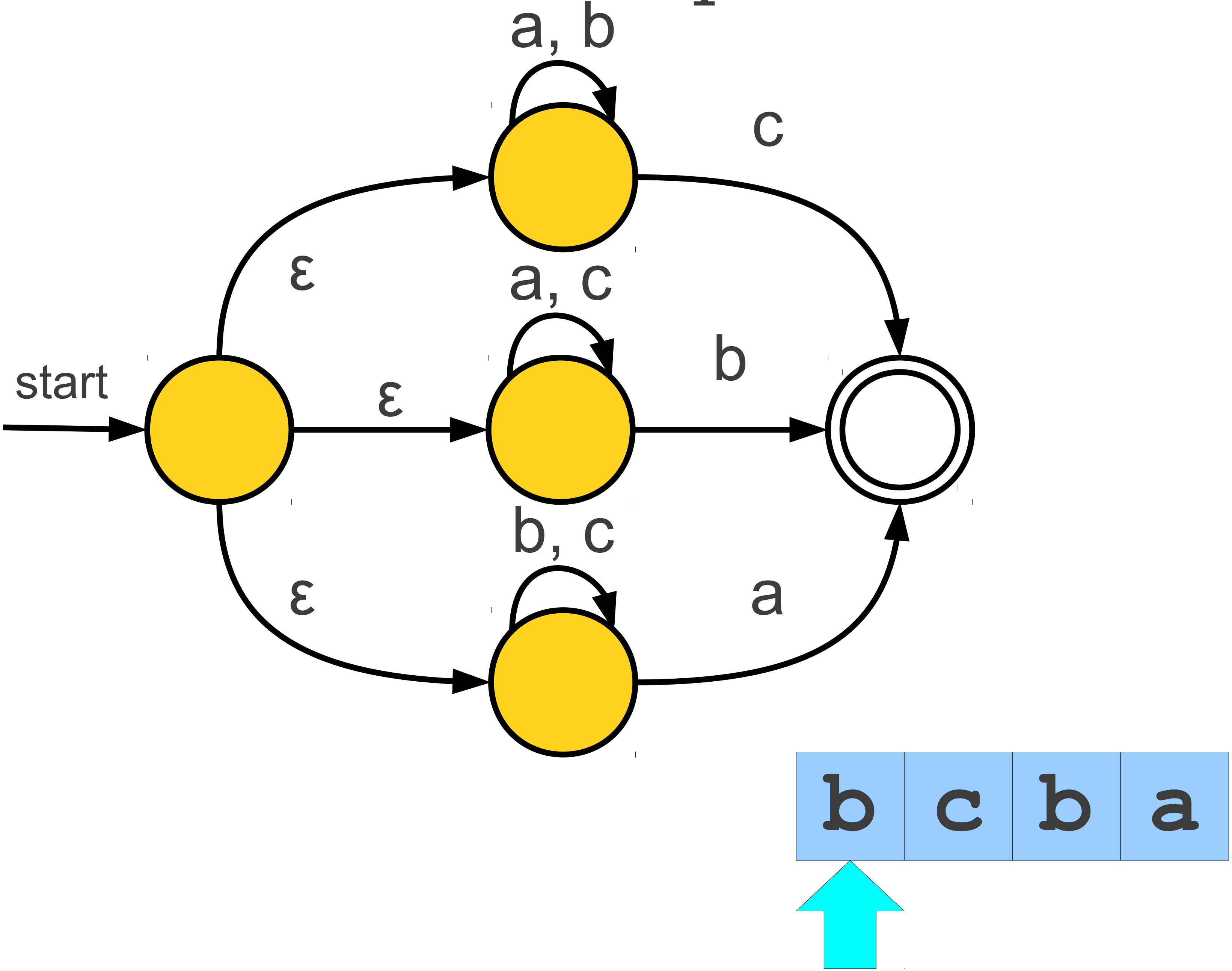
An Even More Complex Automaton



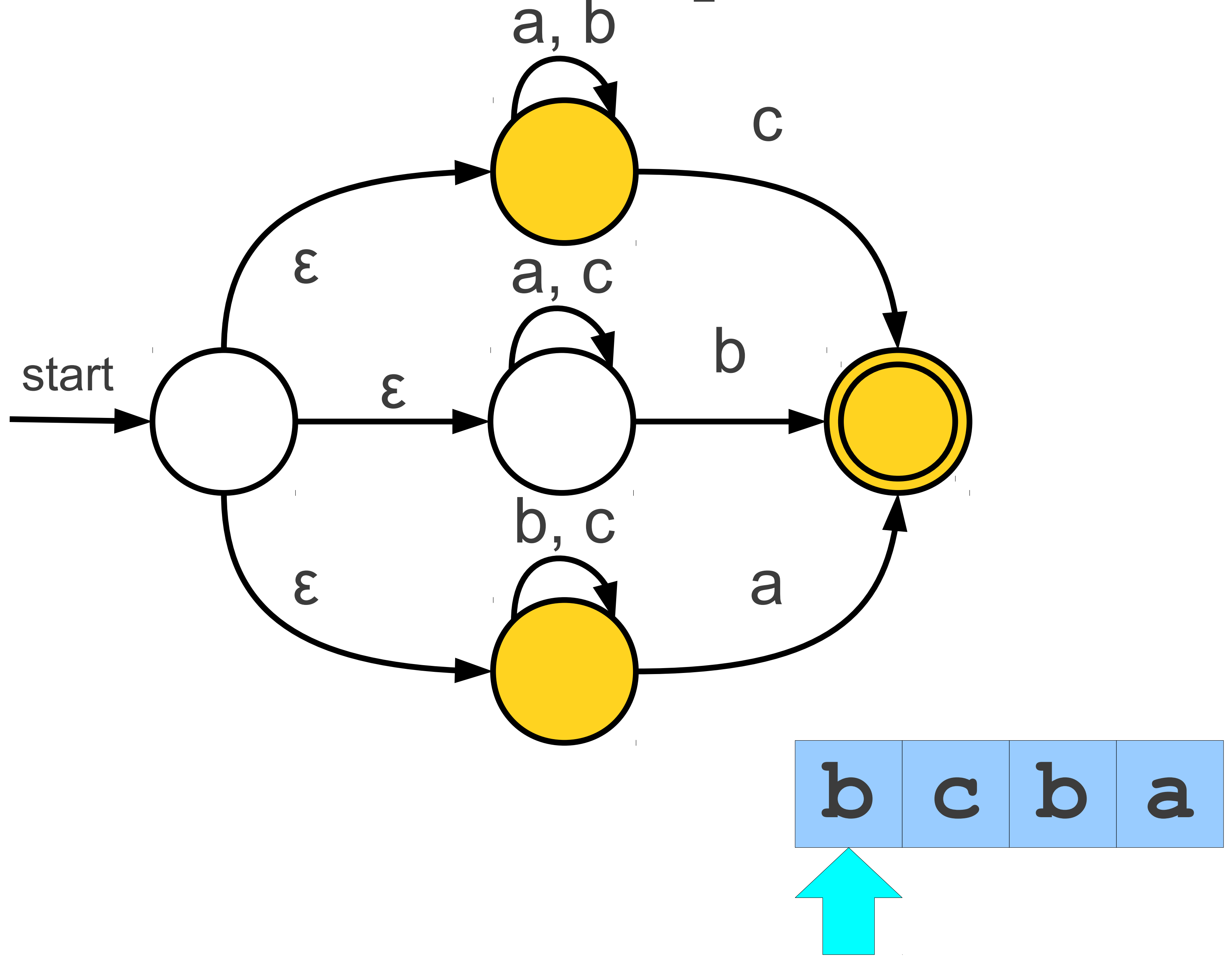
An Even More Complex Automaton



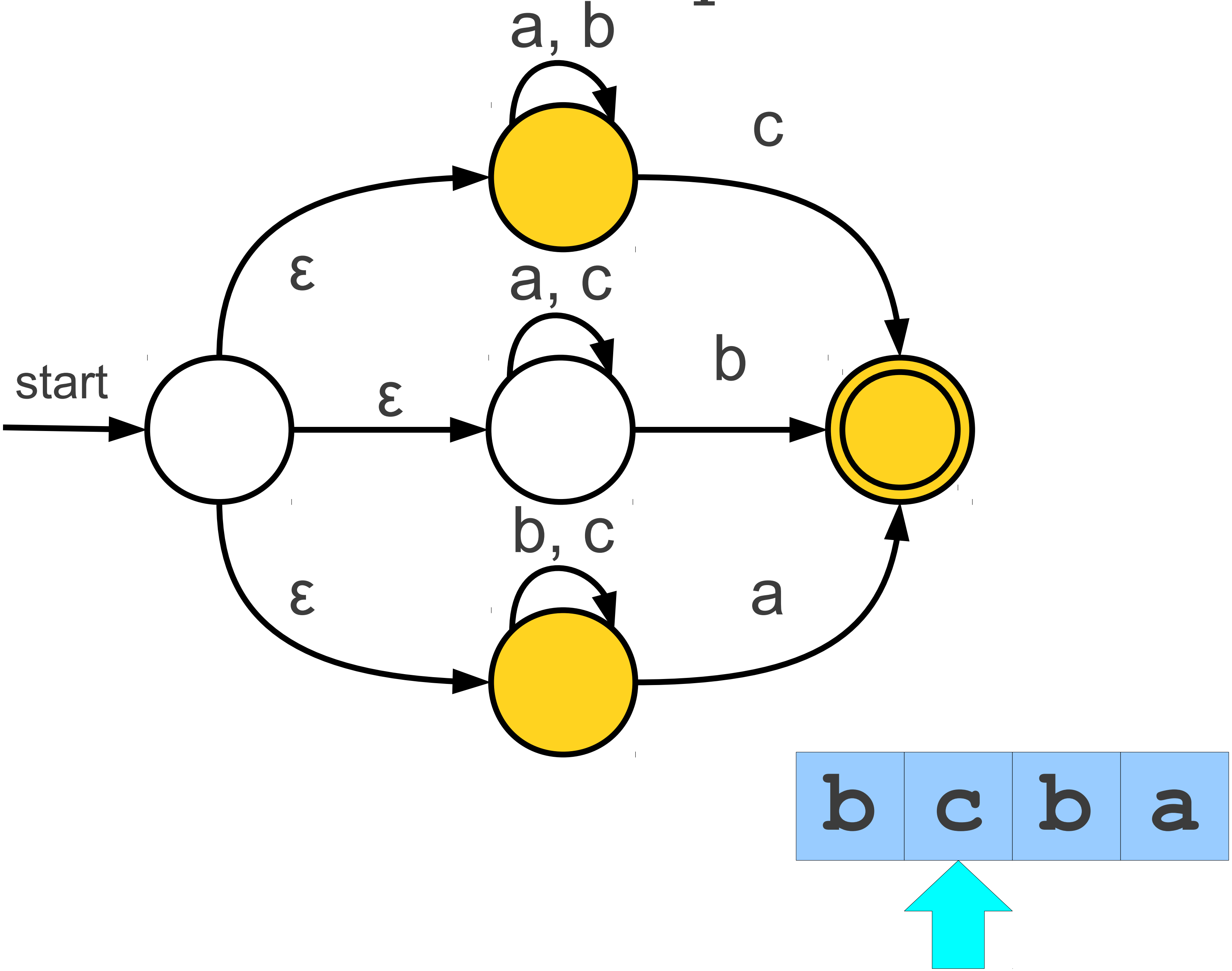
An Even More Complex Automaton



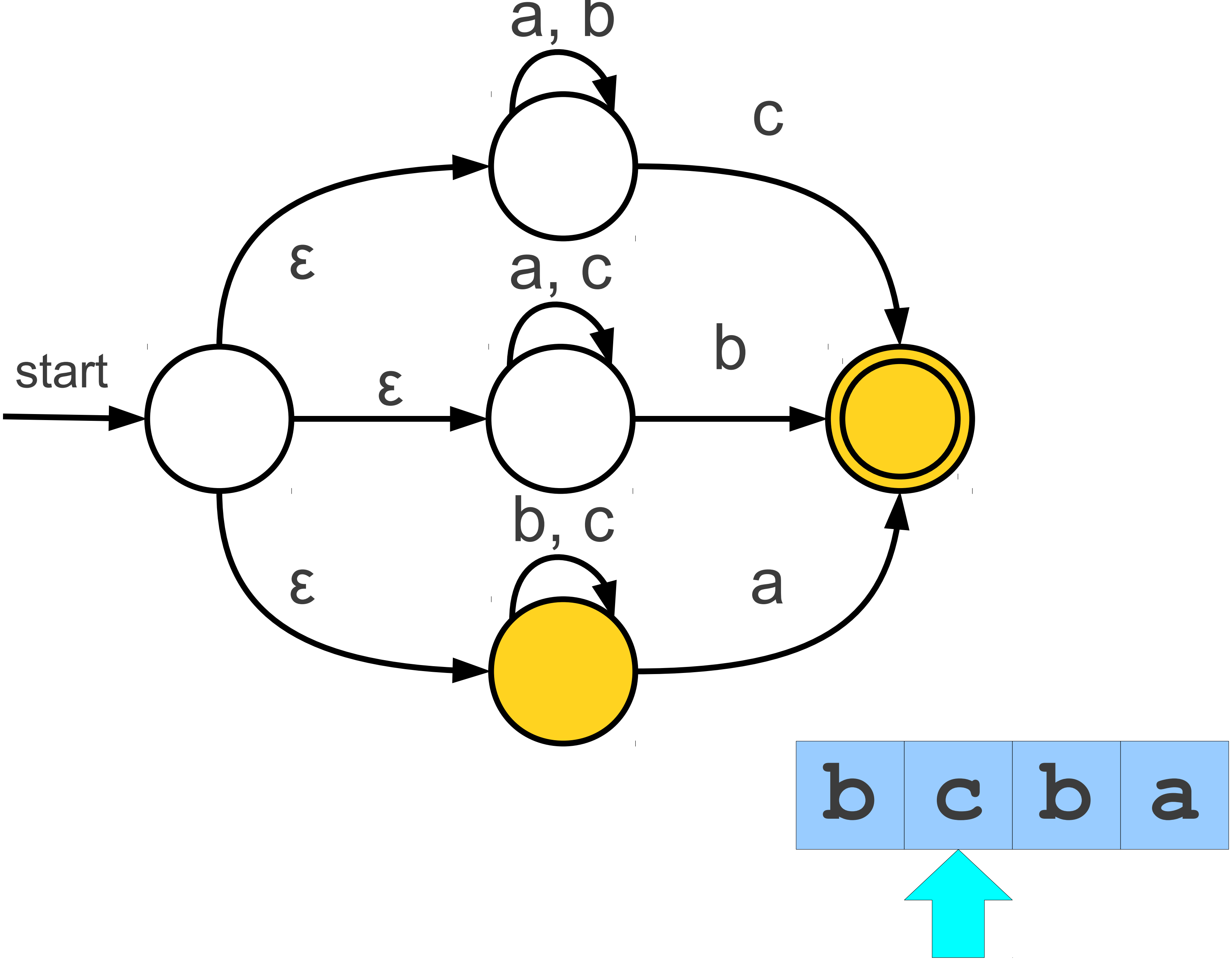
An Even More Complex Automaton



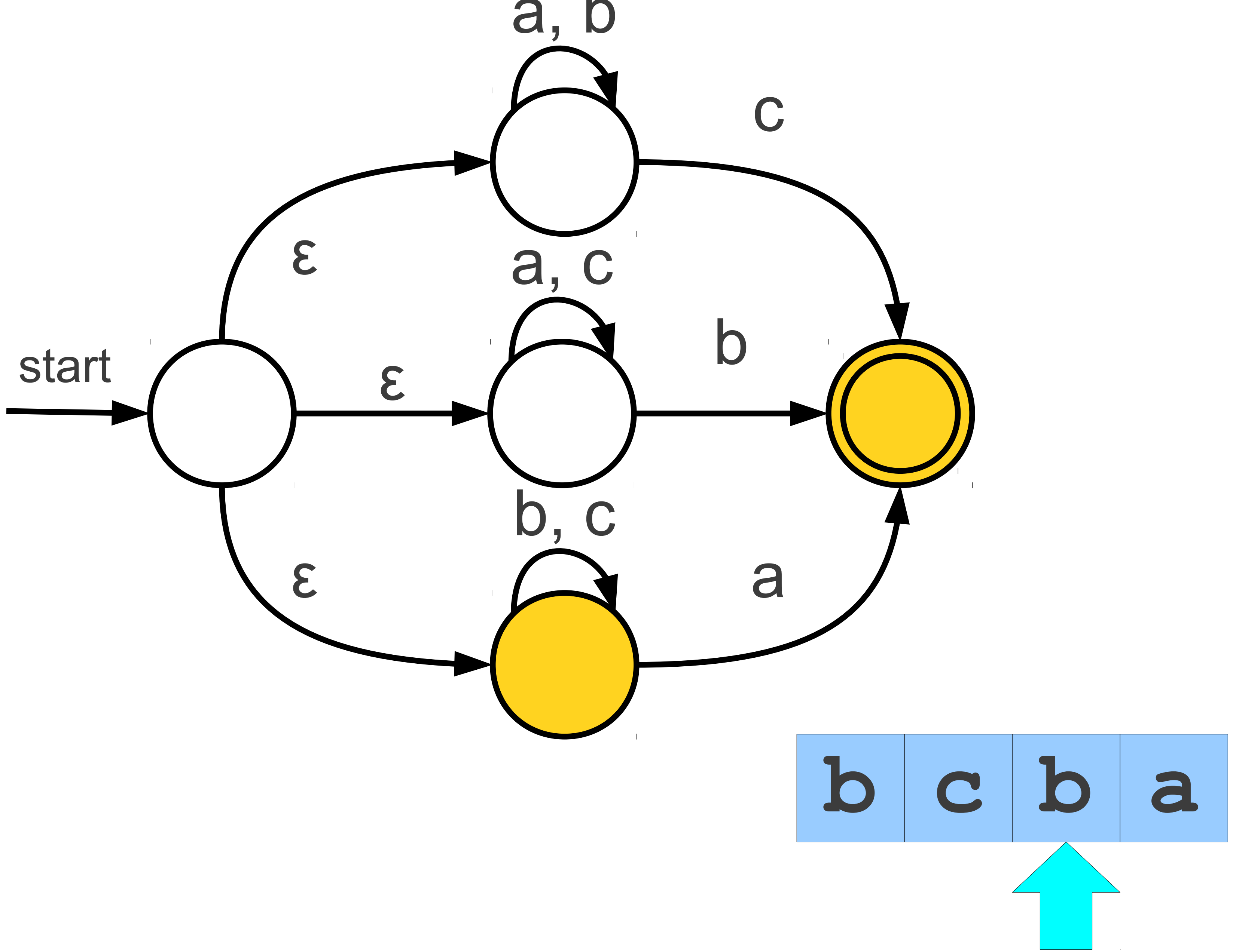
An Even More Complex Automaton



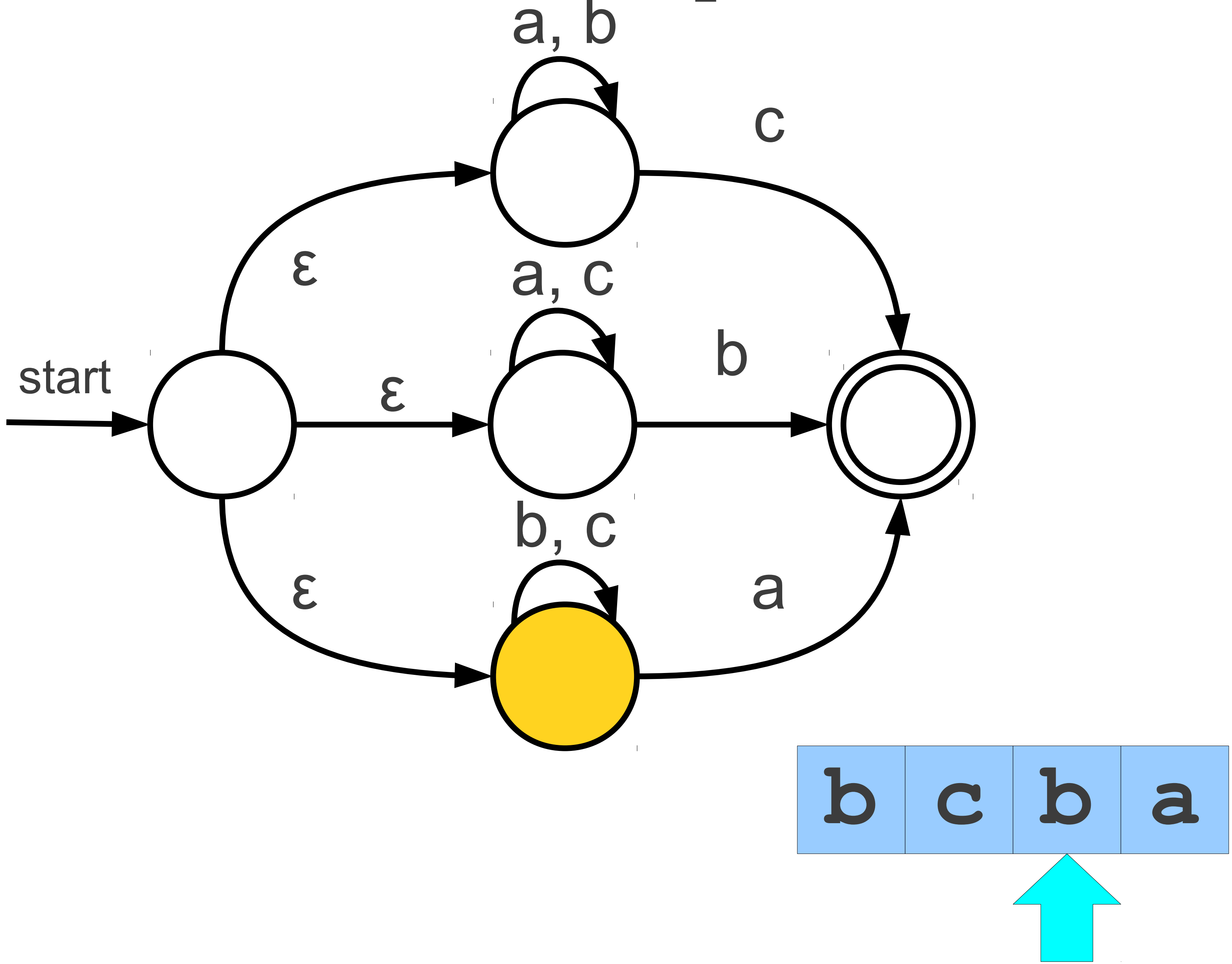
An Even More Complex Automaton



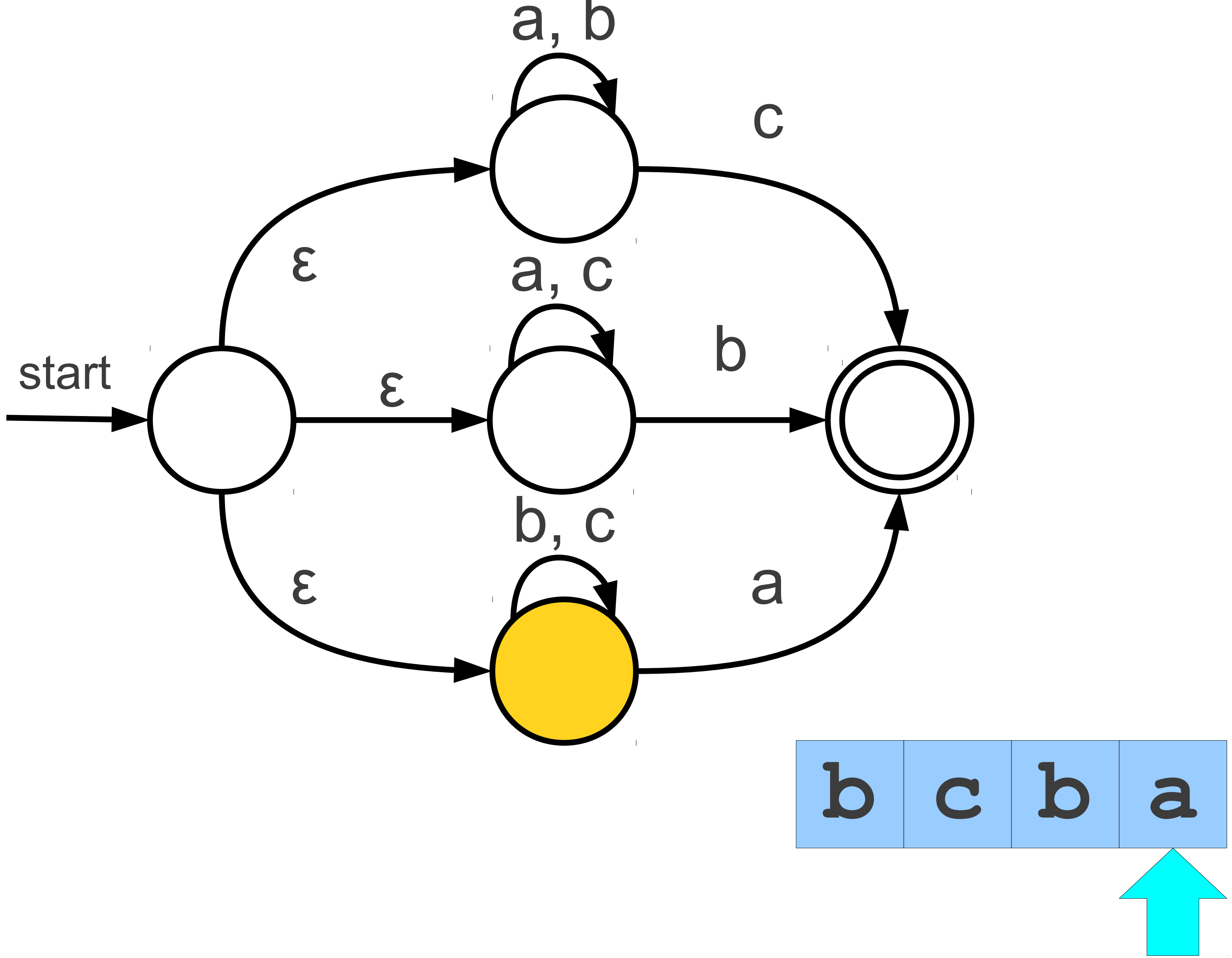
An Even More Complex Automaton



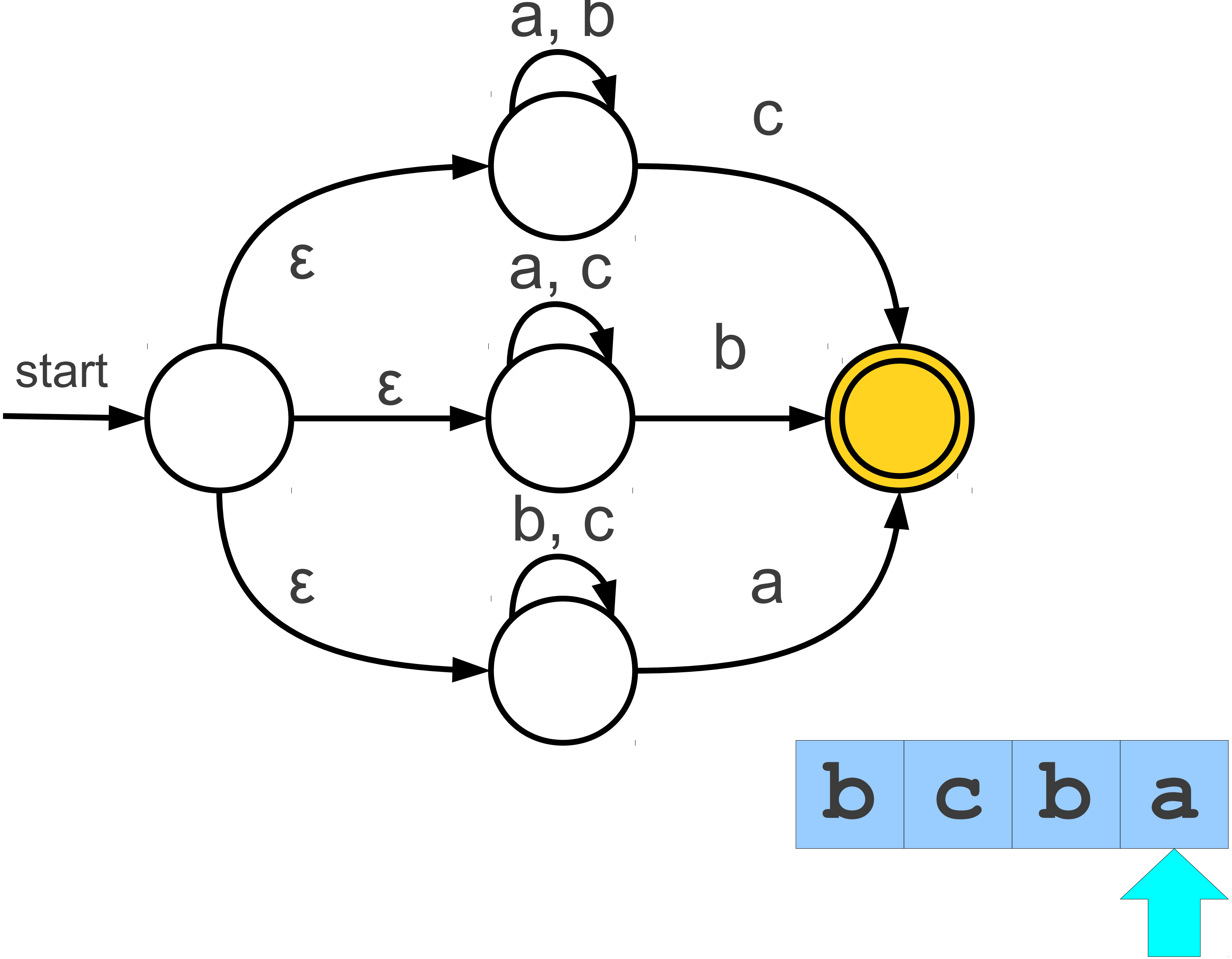
An Even More Complex Automaton



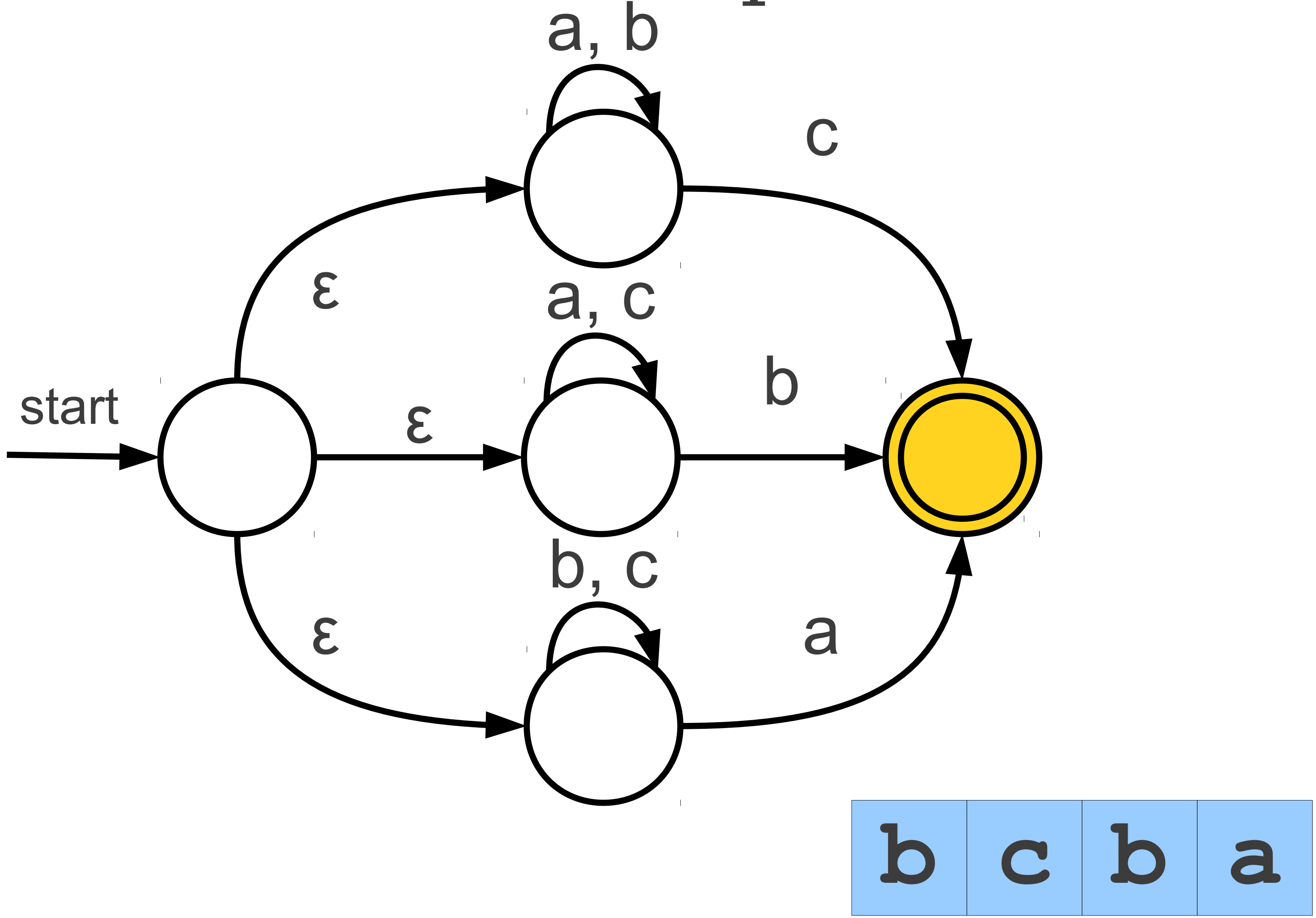
An Even More Complex Automaton



An Even More Complex Automaton



An Even More Complex Automaton

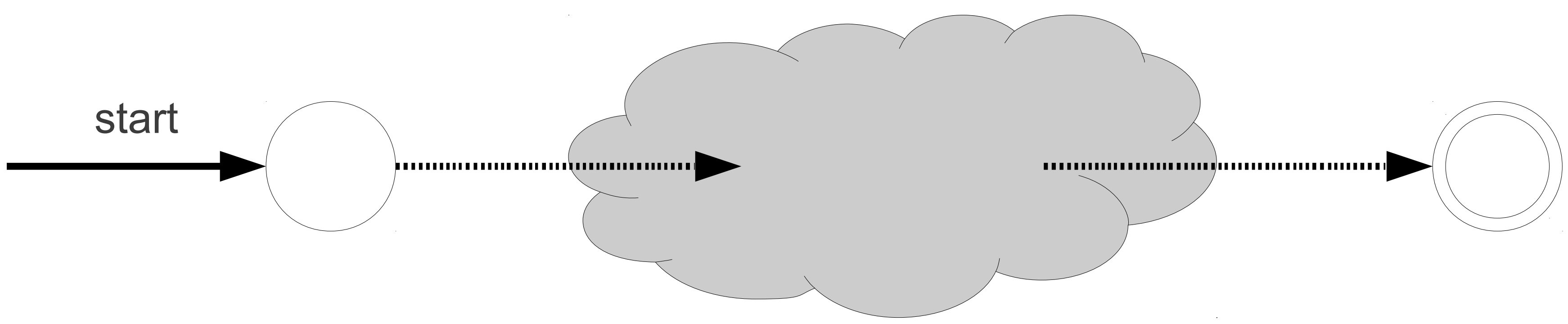


Simulating an NFA

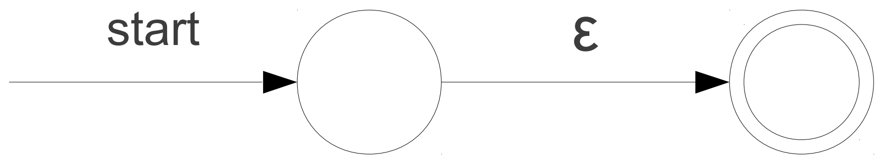
- Keep track of a set of states, initially the start state and everything reachable by ε-moves.
- For each character in the input:
  - Maintain a set of next states, initially empty.
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the 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

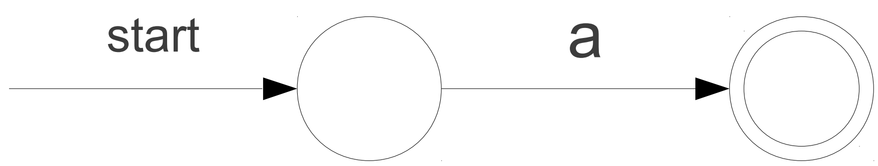
- There is a (beautiful!) procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting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



Base Cases



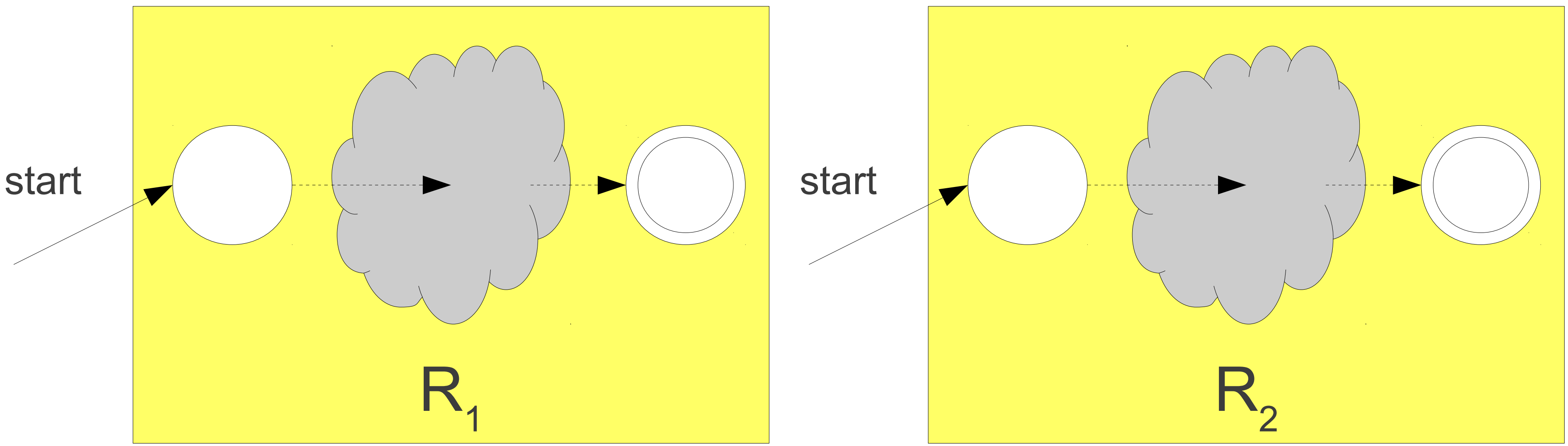
Automaton for ε



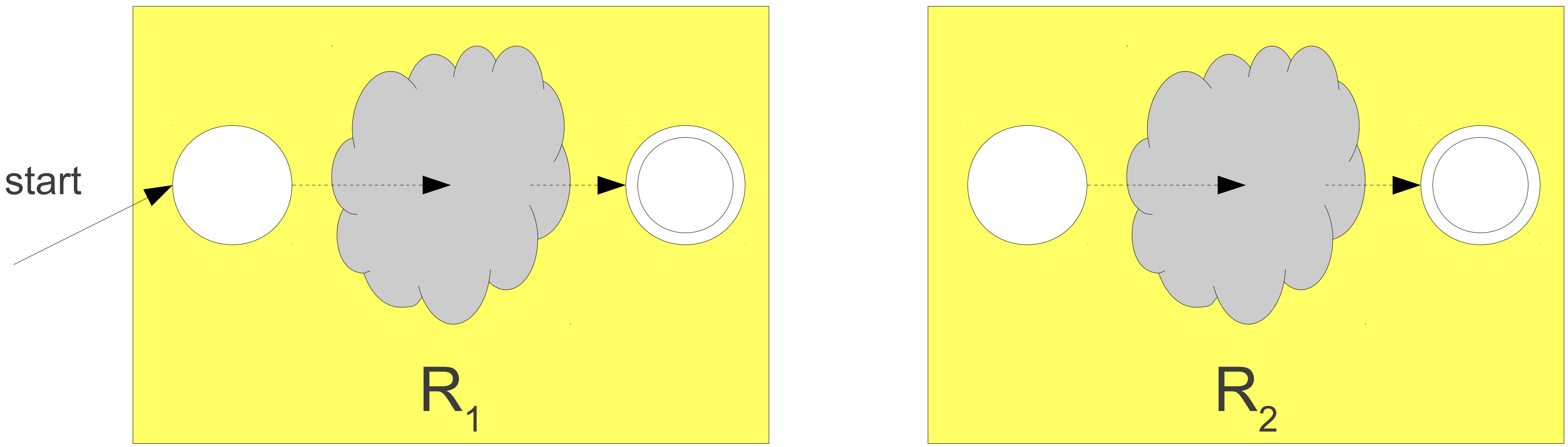
Automaton for single character **a**

Construction for R_1R_2

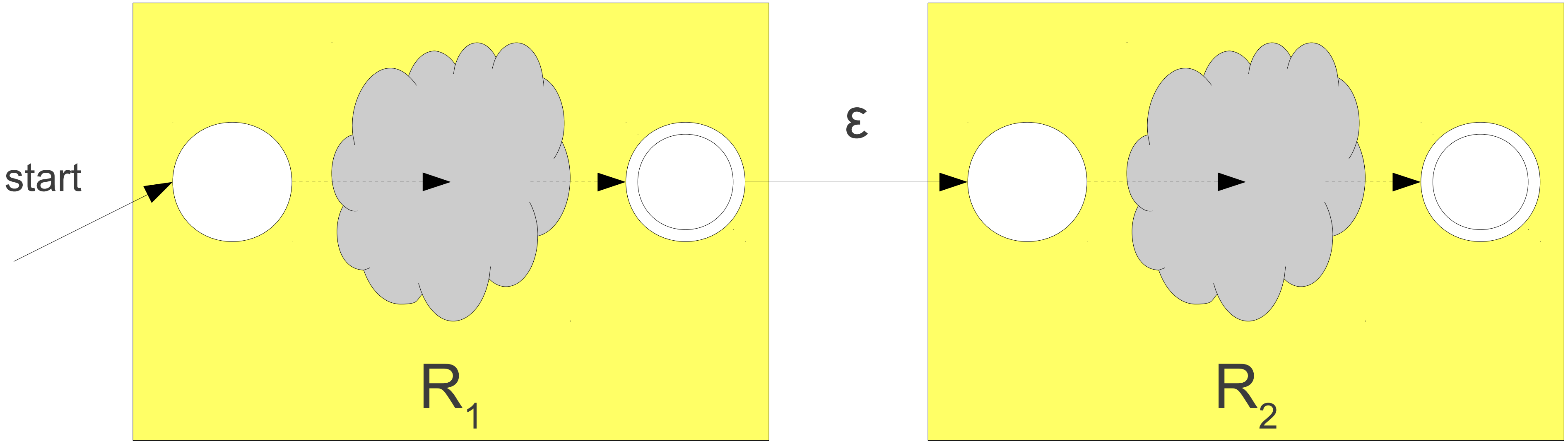
Construction for R_1R_2



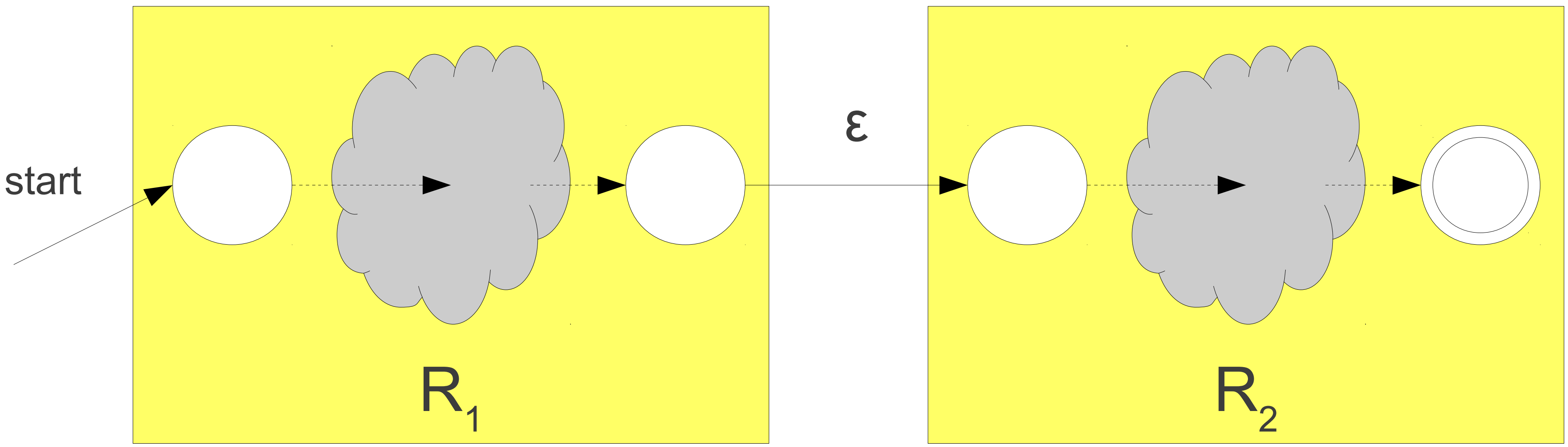
Construction for $R_1 R_2$



Construction for $R_1 R_2$

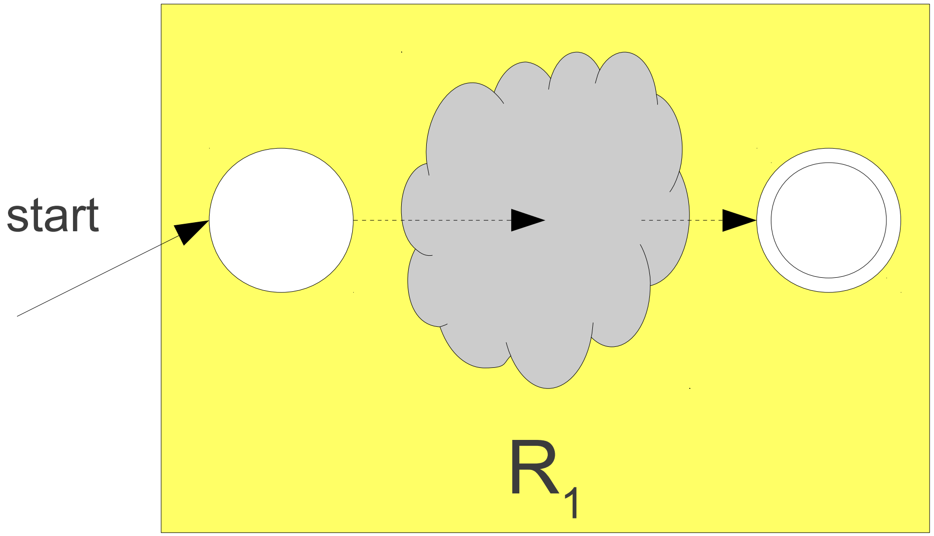


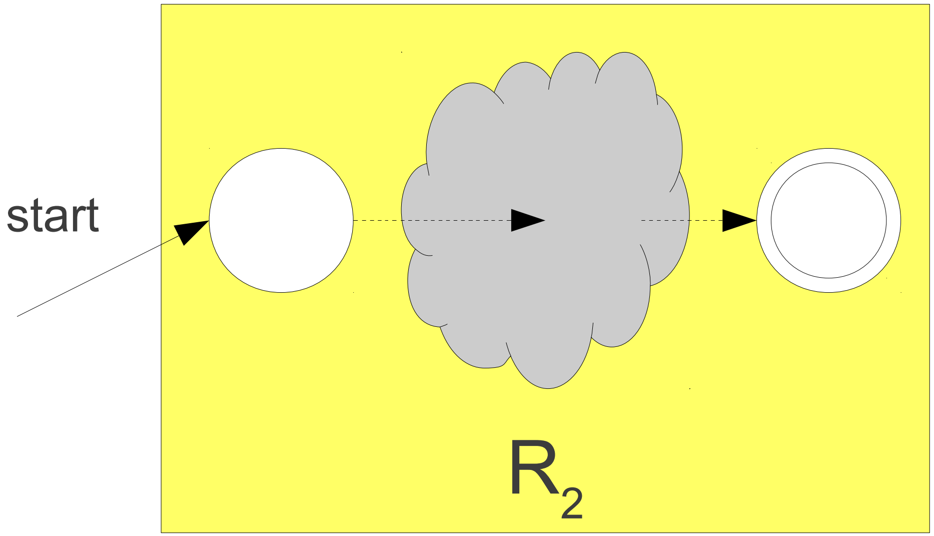
Construction for R_1R_2



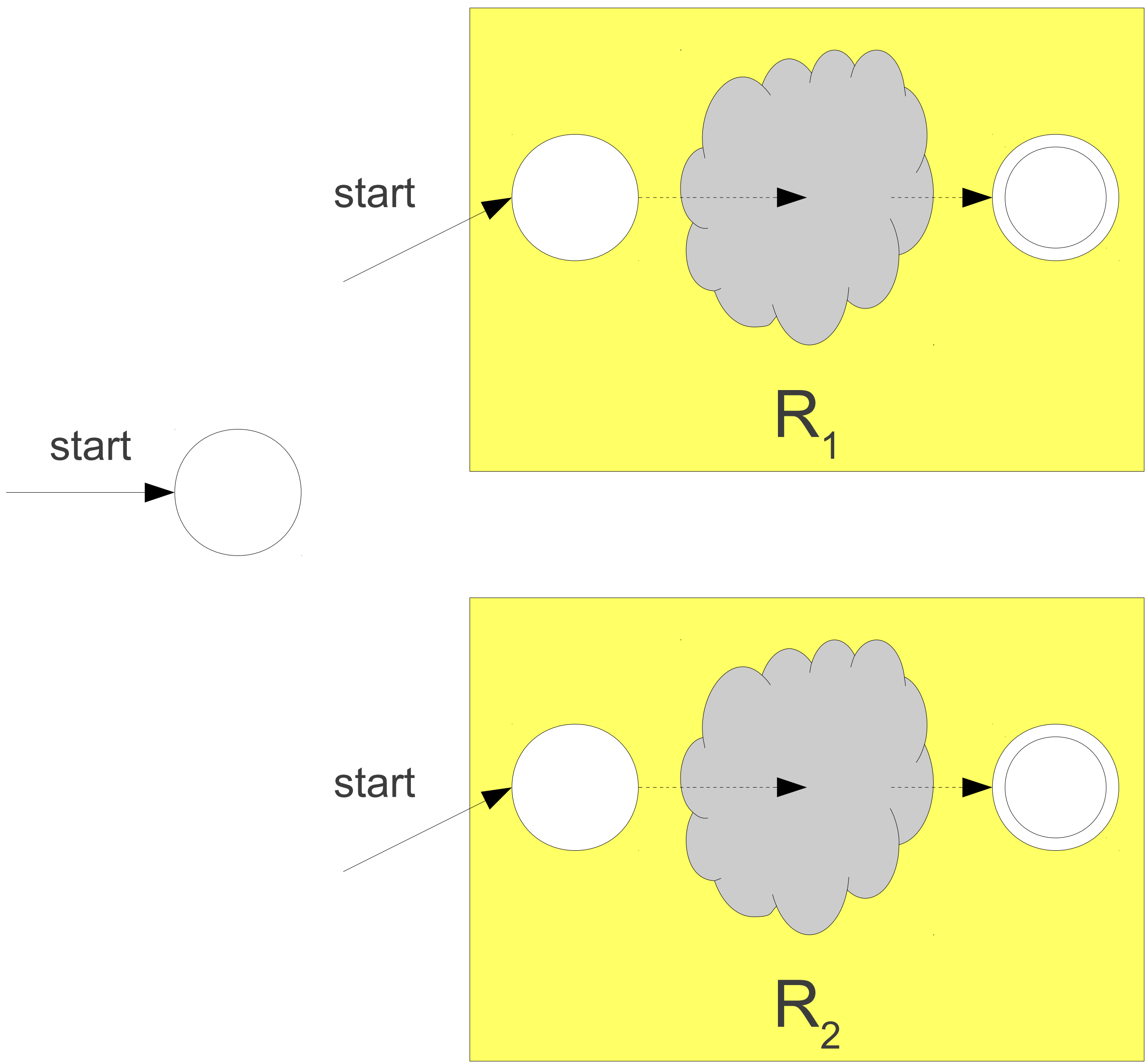
Construction for $R_1 \mid R_2$

Construction for $R_1 \mid R_2$

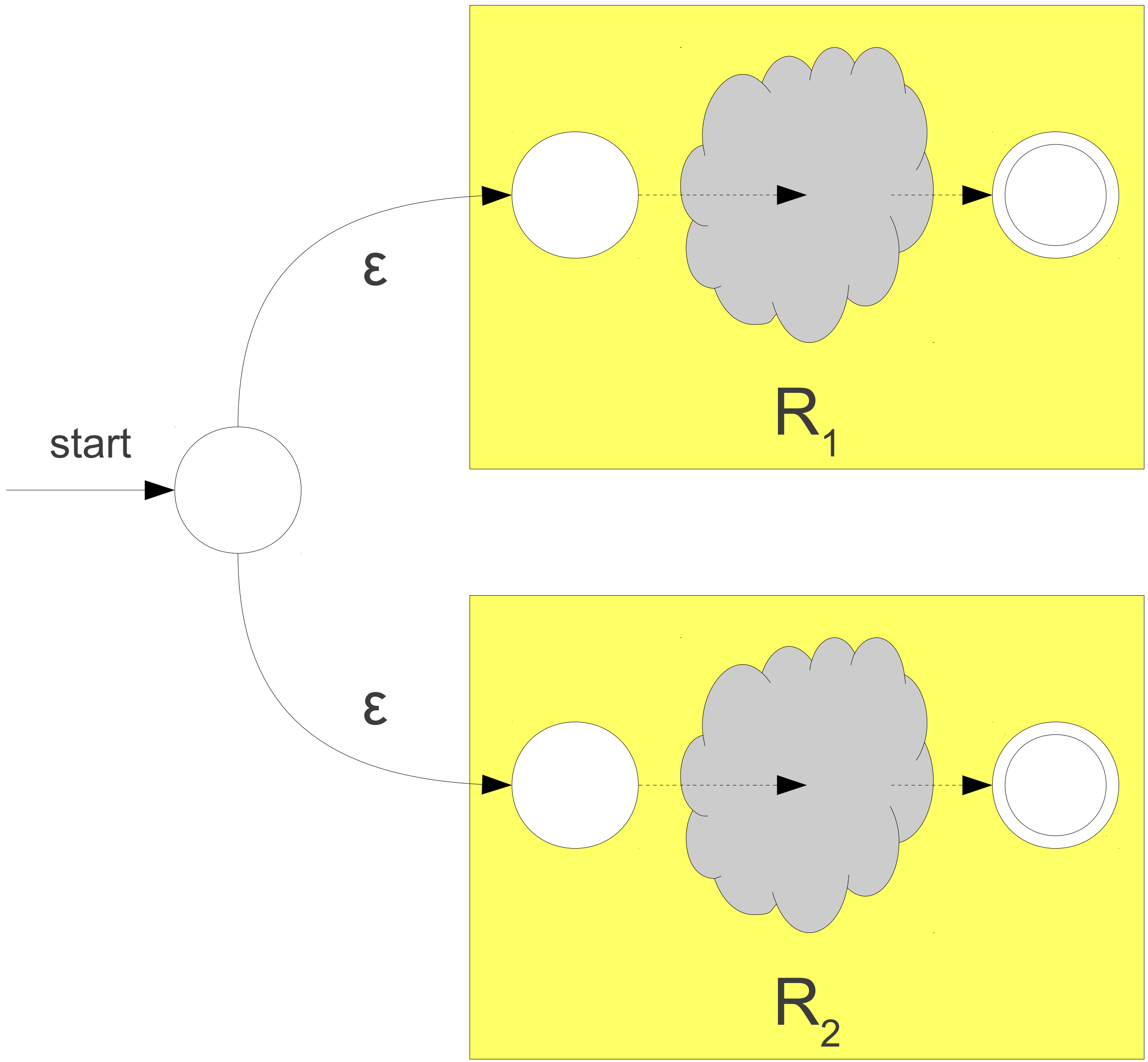




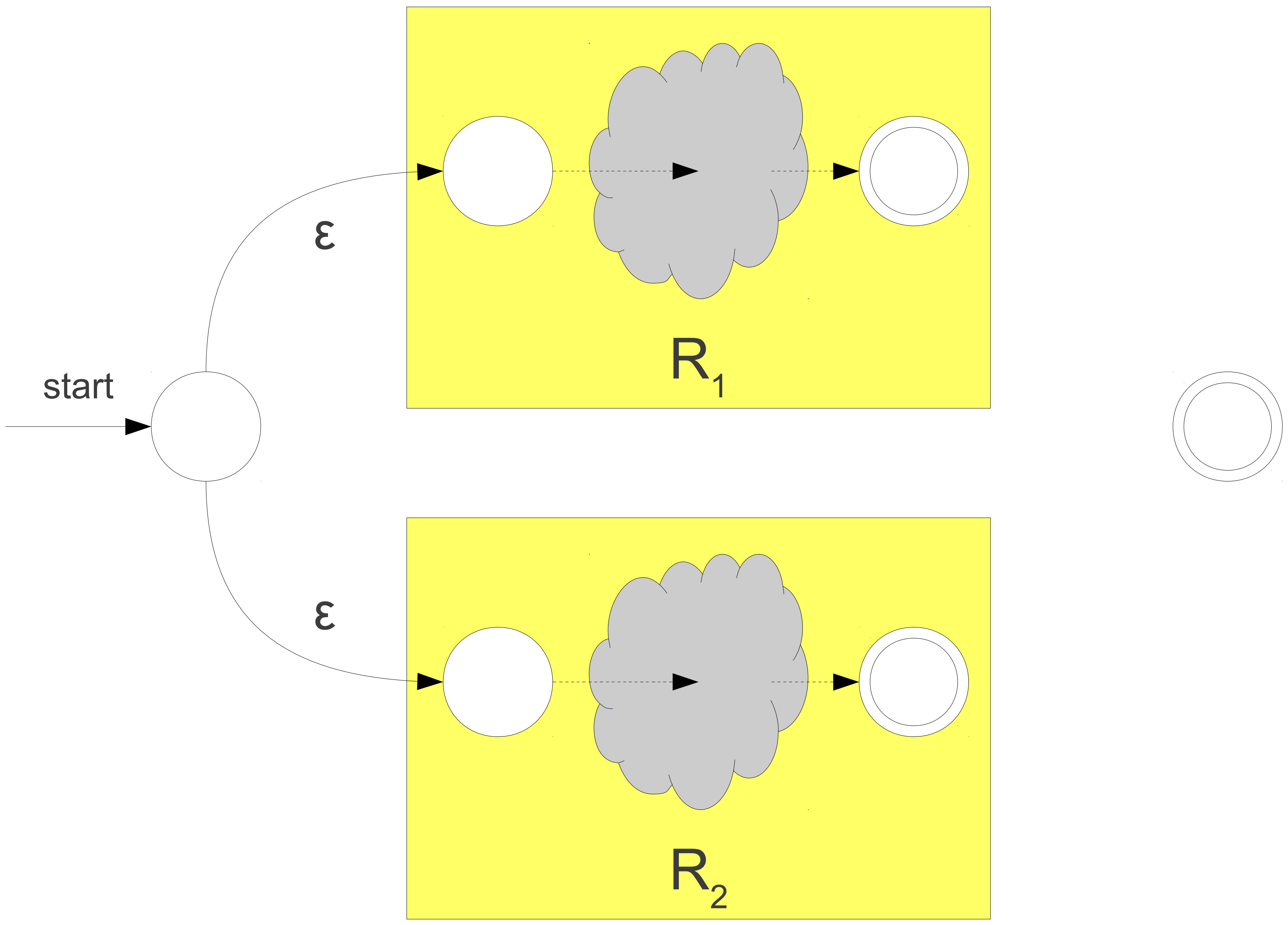
Construction for $R_1 \mid R_2$



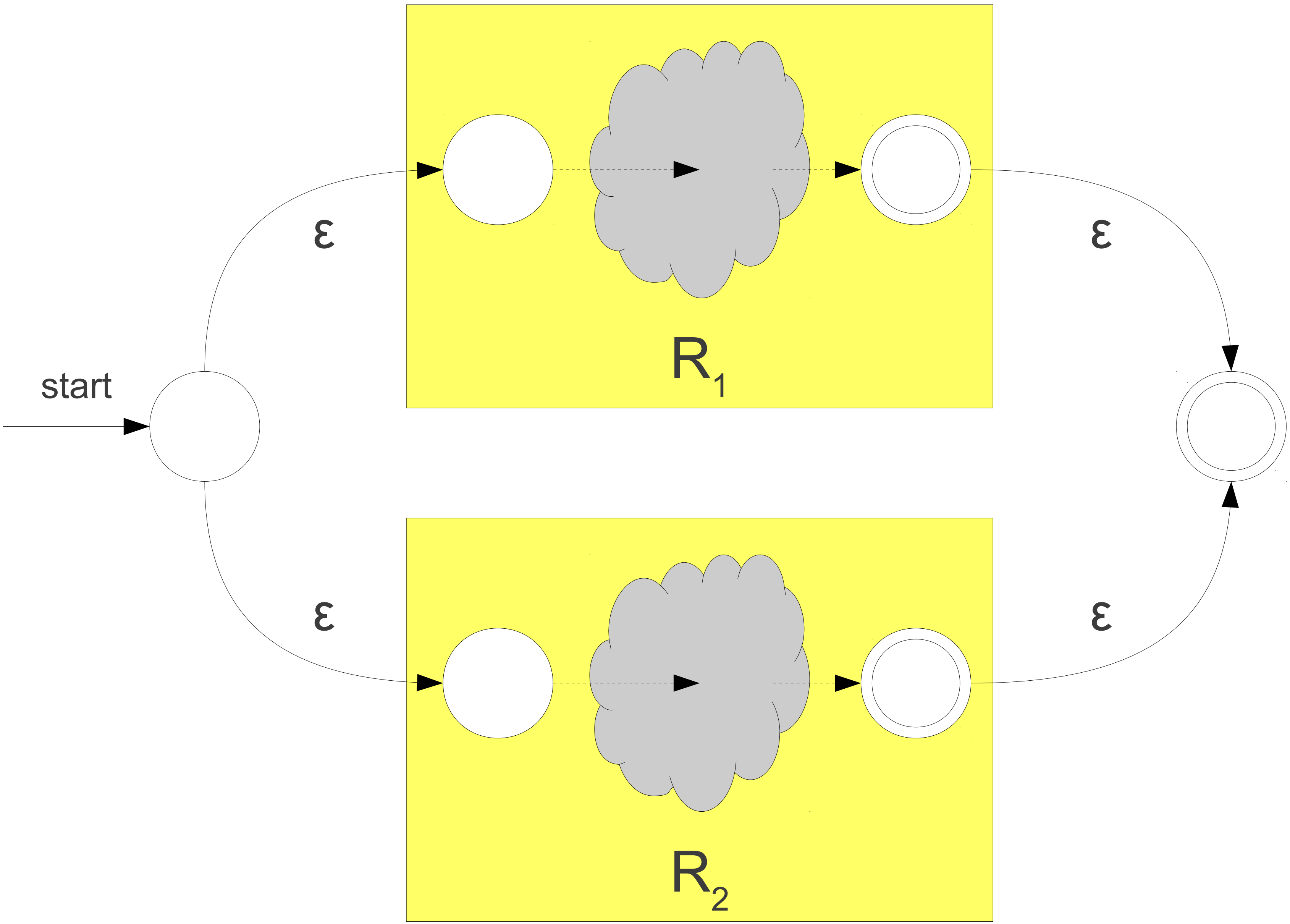
Construction for $R_1 \mid R_2$



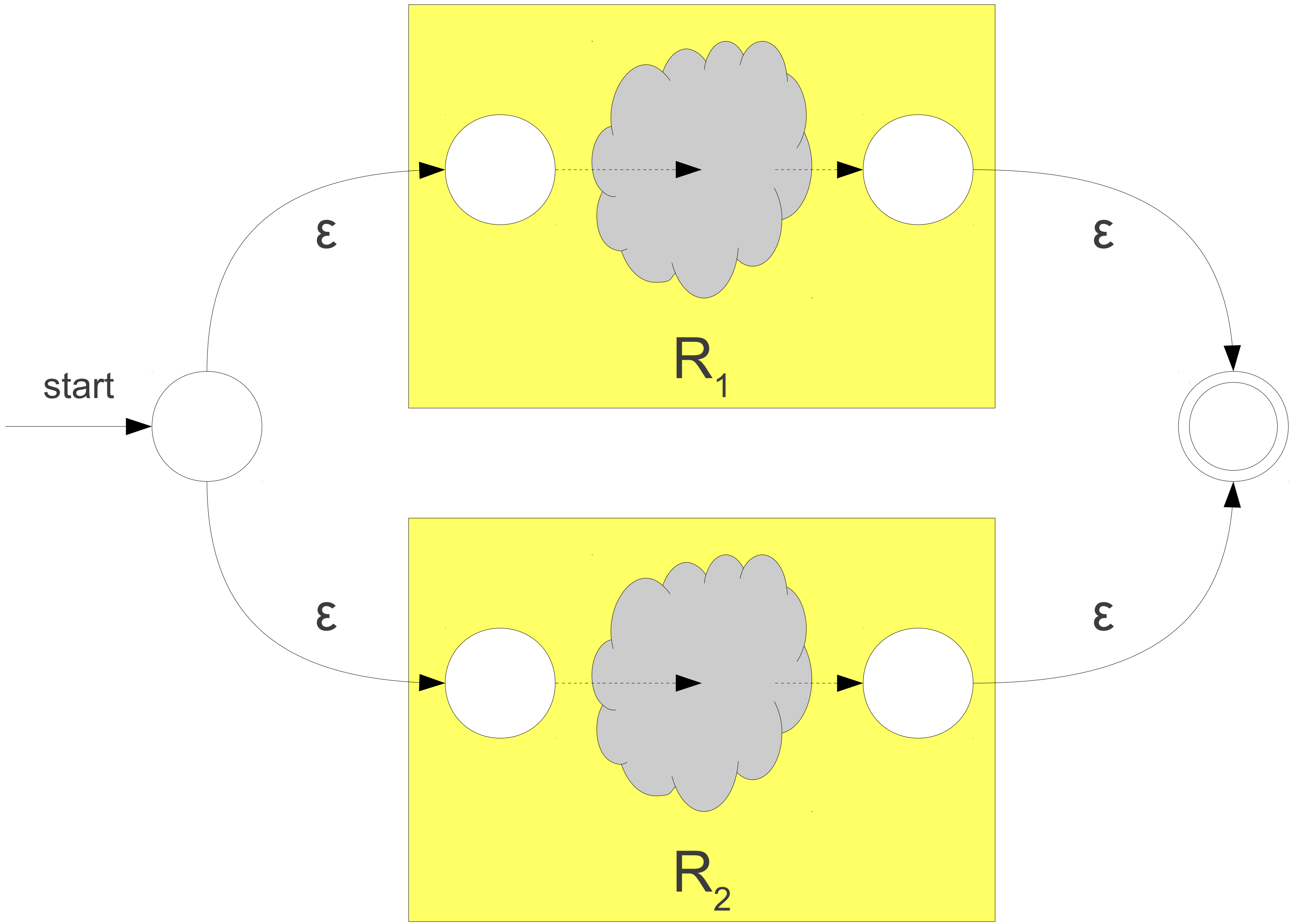
Construction for $R_1 \mid R_2$



Construction for $R_1 \mid R_2$

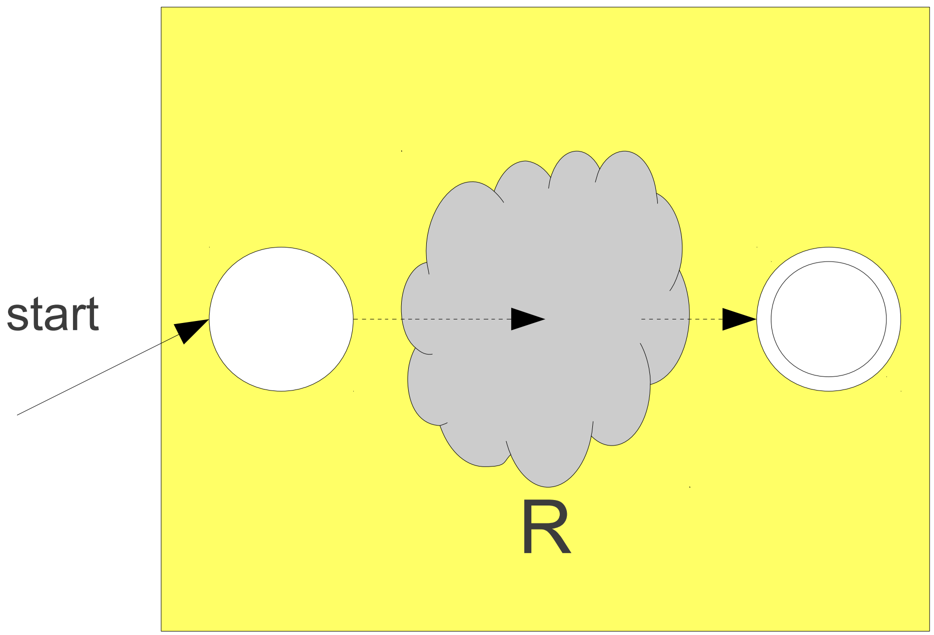


Construction for $R_1 \mid R_2$

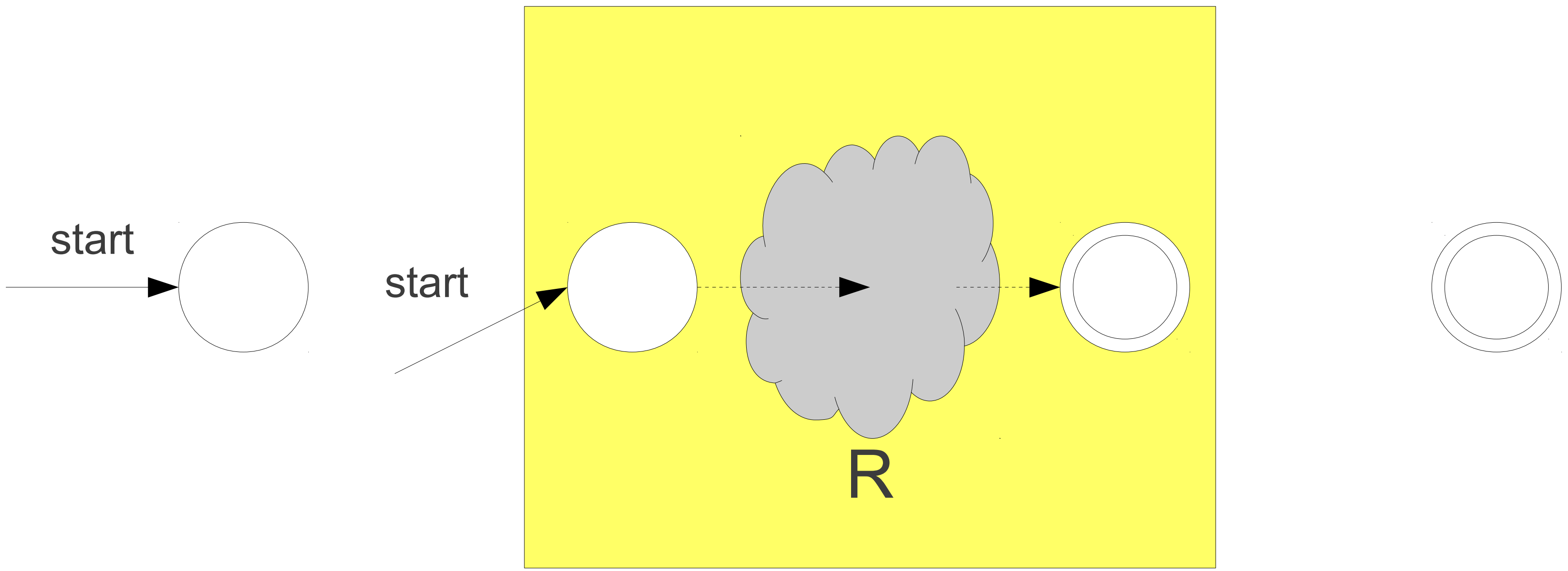


Construction for R*

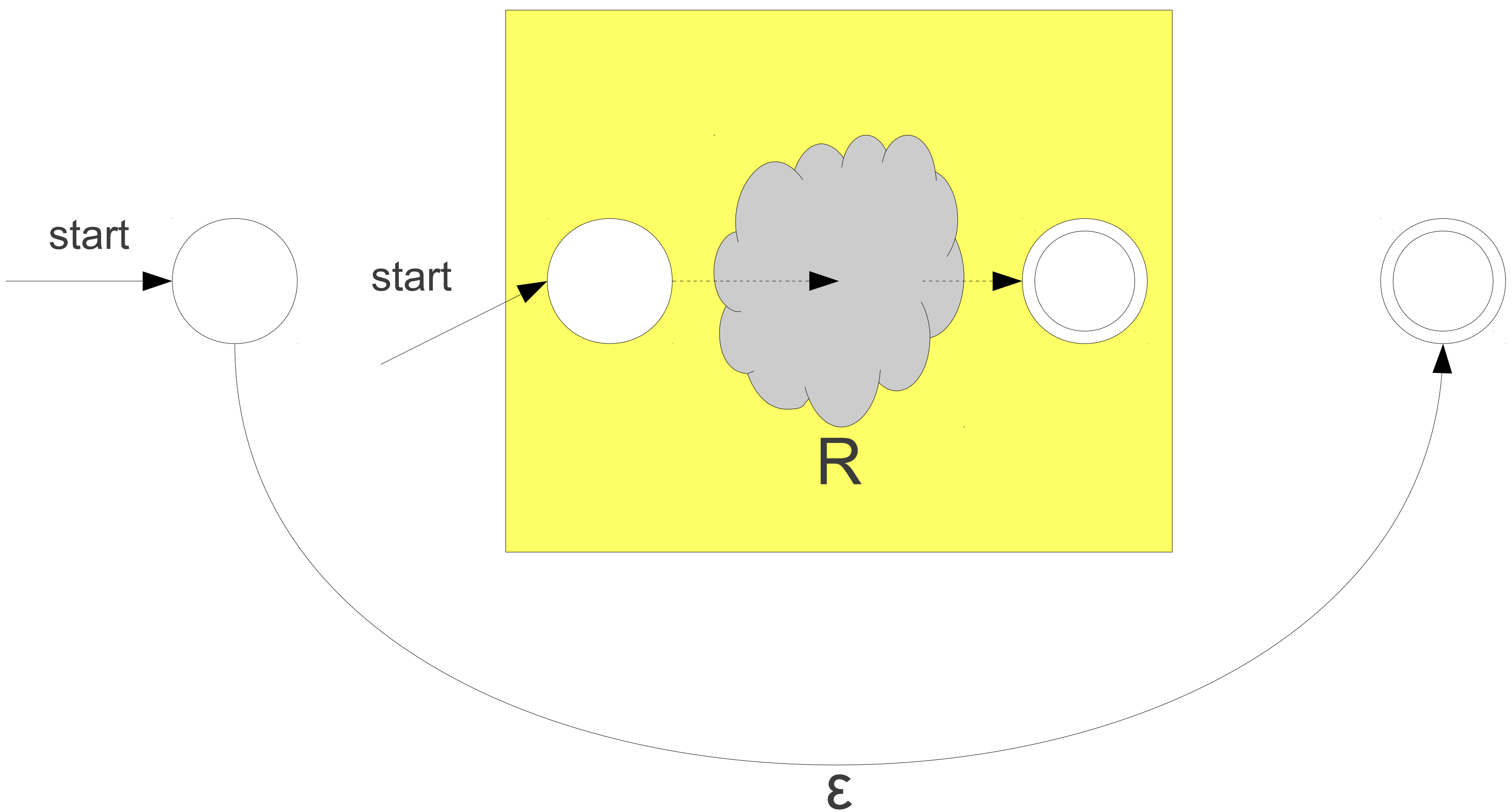
Construction for R*



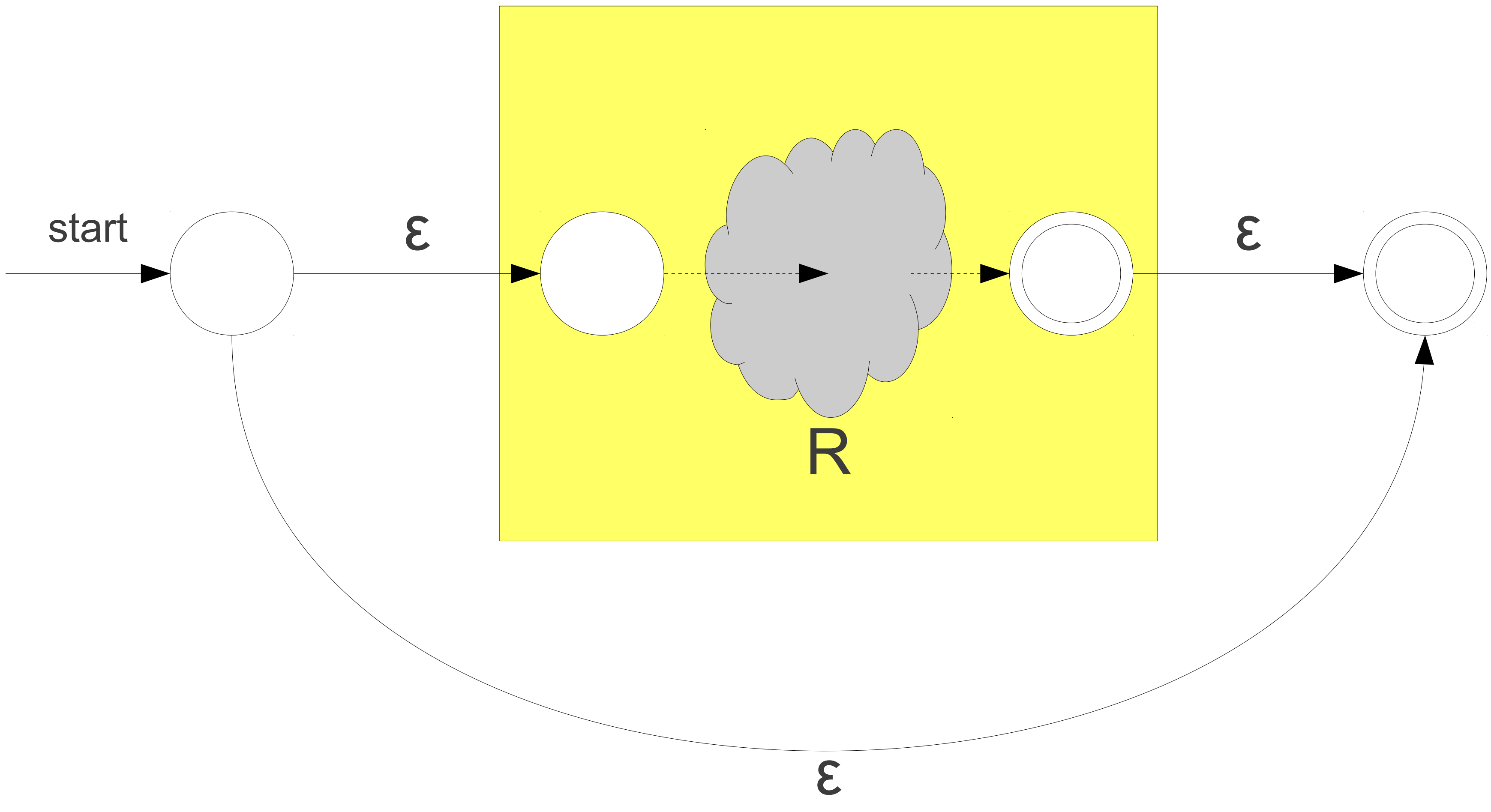
Construction for R*



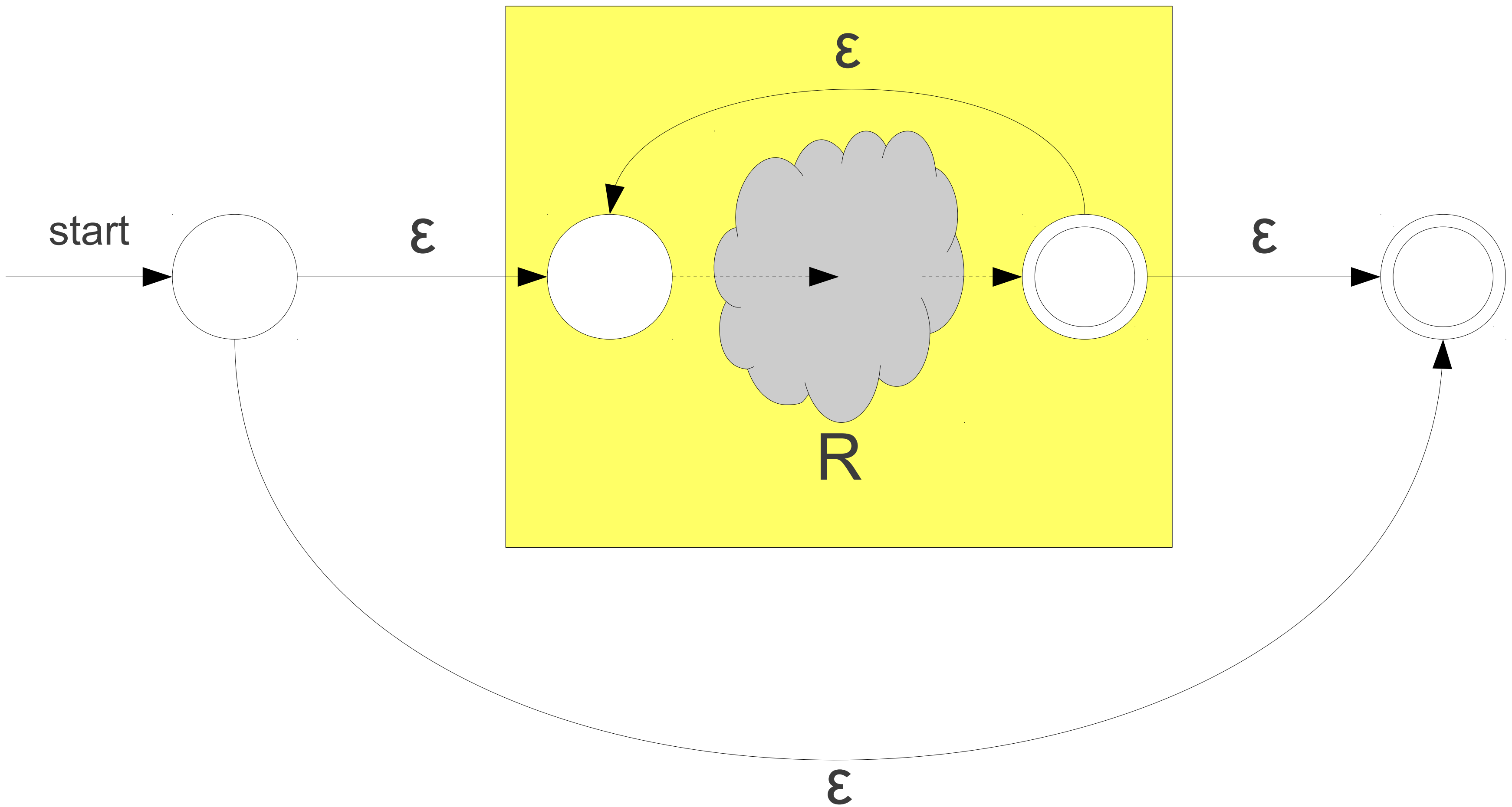
Construction for R*



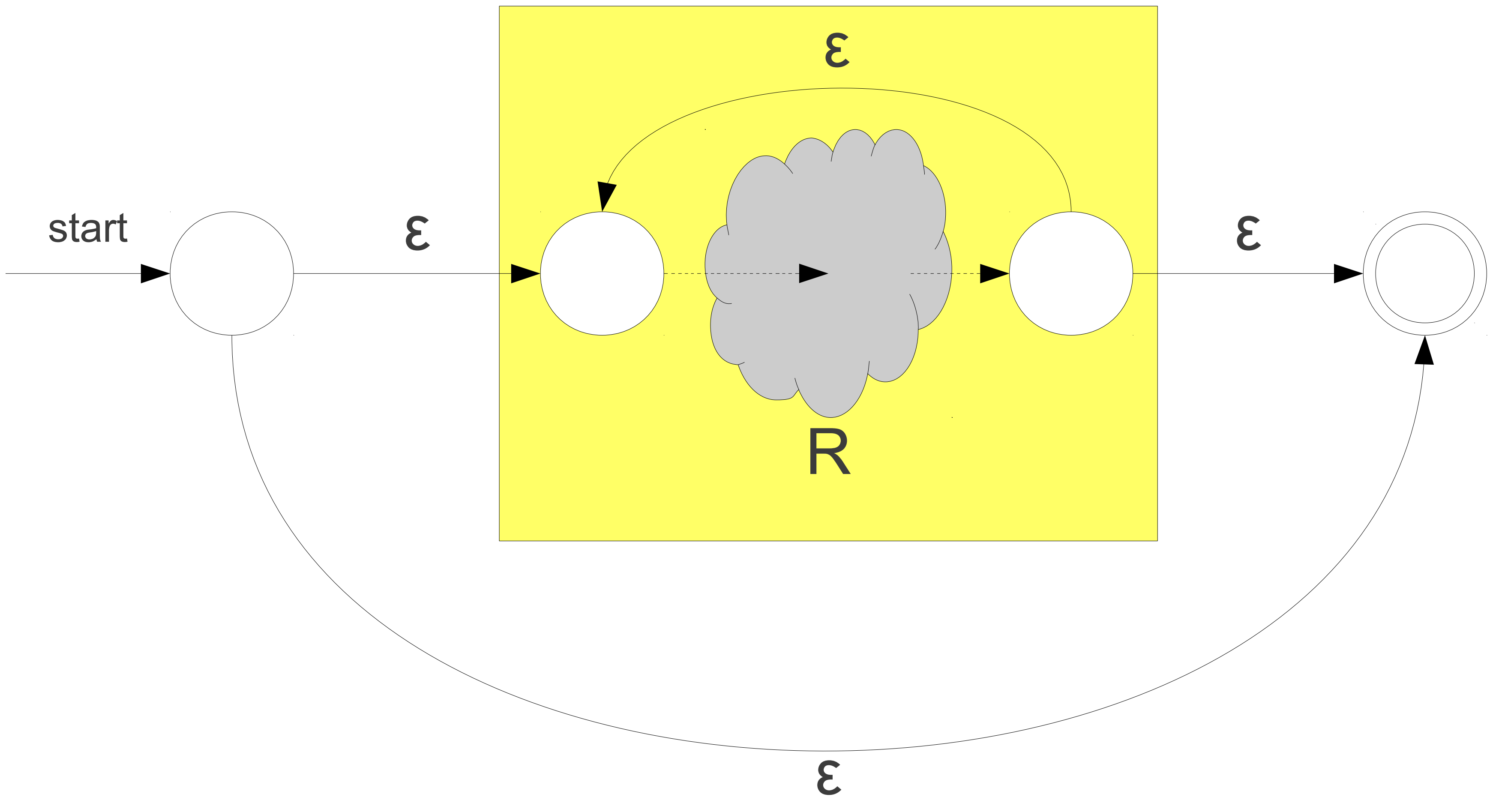
Construction for R*



Construction for R*



Construction for R*



Overall Result

- Any regular expression of length n can be converted into an NFA with $O(n)$ states.
- Can determine whether a string of length m matches a regular expression of length n in time $O(mn^2)$.
- We'll see how to make this $O(m)$ later (this is independent of the complexity of the regular expression!)

A Quick Diversion…

I am having some difficulty compiling a C++ program that I've written.

This program is very simple and, to the best of my knowledge, conforms to all the rules set forth in the C++ Standard. [...]

The program is as follows:

Source:
http://stackoverflow.com/questions/5508110/why-is-this-program-erroneously-rejected-by-three-c-compilers

I am having some difficulty compiling a C++ program that I've written.

This program is very simple and, to the best of my knowledge, conforms to all the rules set forth in the C++ Standard. [...]

The program is as follows:

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Source:
http://stackoverflow.com/questions/5508110/why-is-this-program-erroneously-rejected-by-three-c-compilers

I am having some difficulty compiling a C++ program that I've written.

This program is very simple and, to the best of my knowledge, conforms to all the rules set forth in the C++ Standard. [...]

The program is as follows:

```
#include <iostream>

int main(int argc, char** argv)
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
> g++ helloworld.png
helloworld.png: file not recognized: File format not recognized
collect2: ld returned 1 exit status
```

Source:
http://stackoverflow.com/questions/5508110/why-is-this-program-erroneously-rejected-by-three-c-compilers

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_]*
```

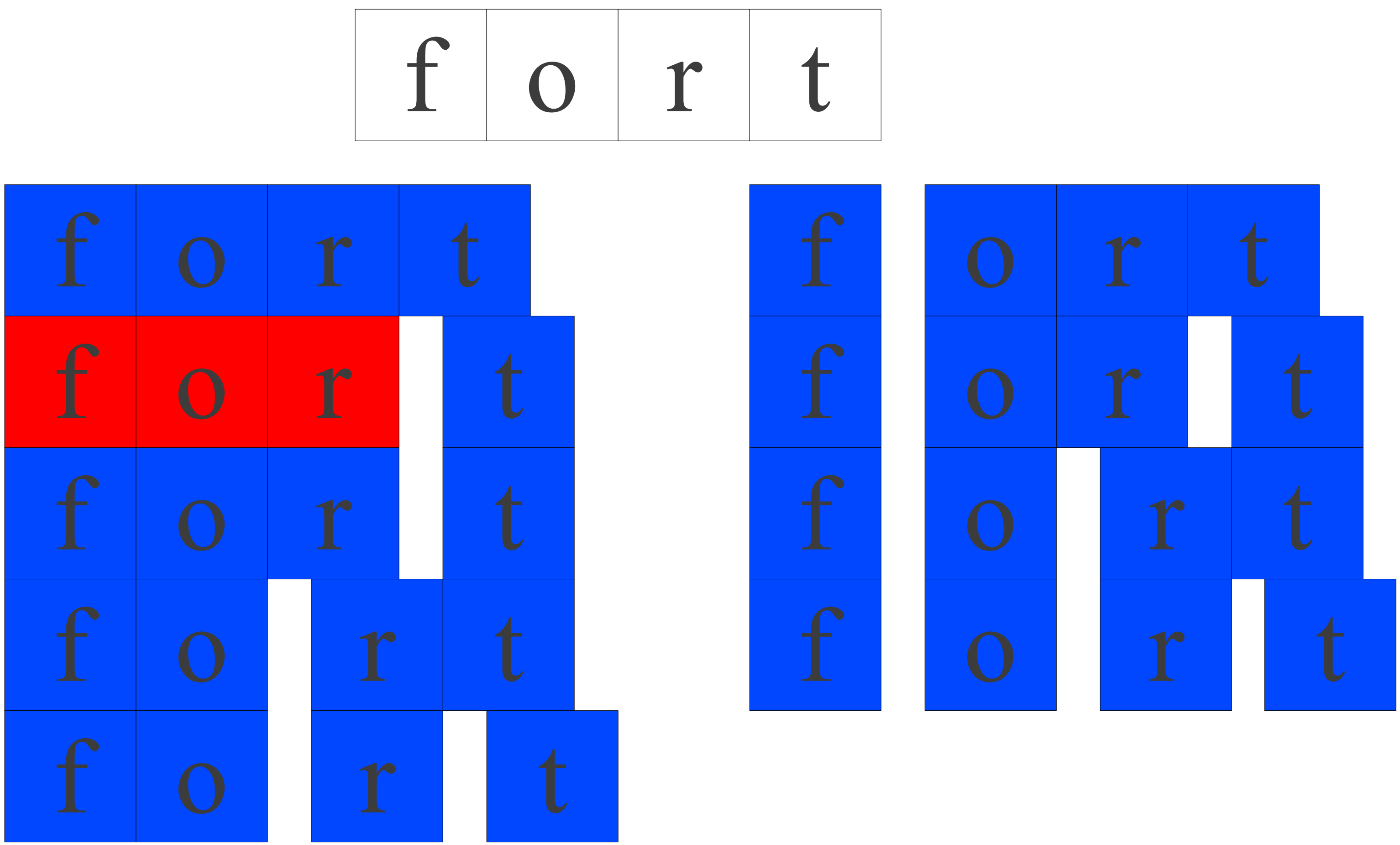
Lexing Ambiguities

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_]*
```

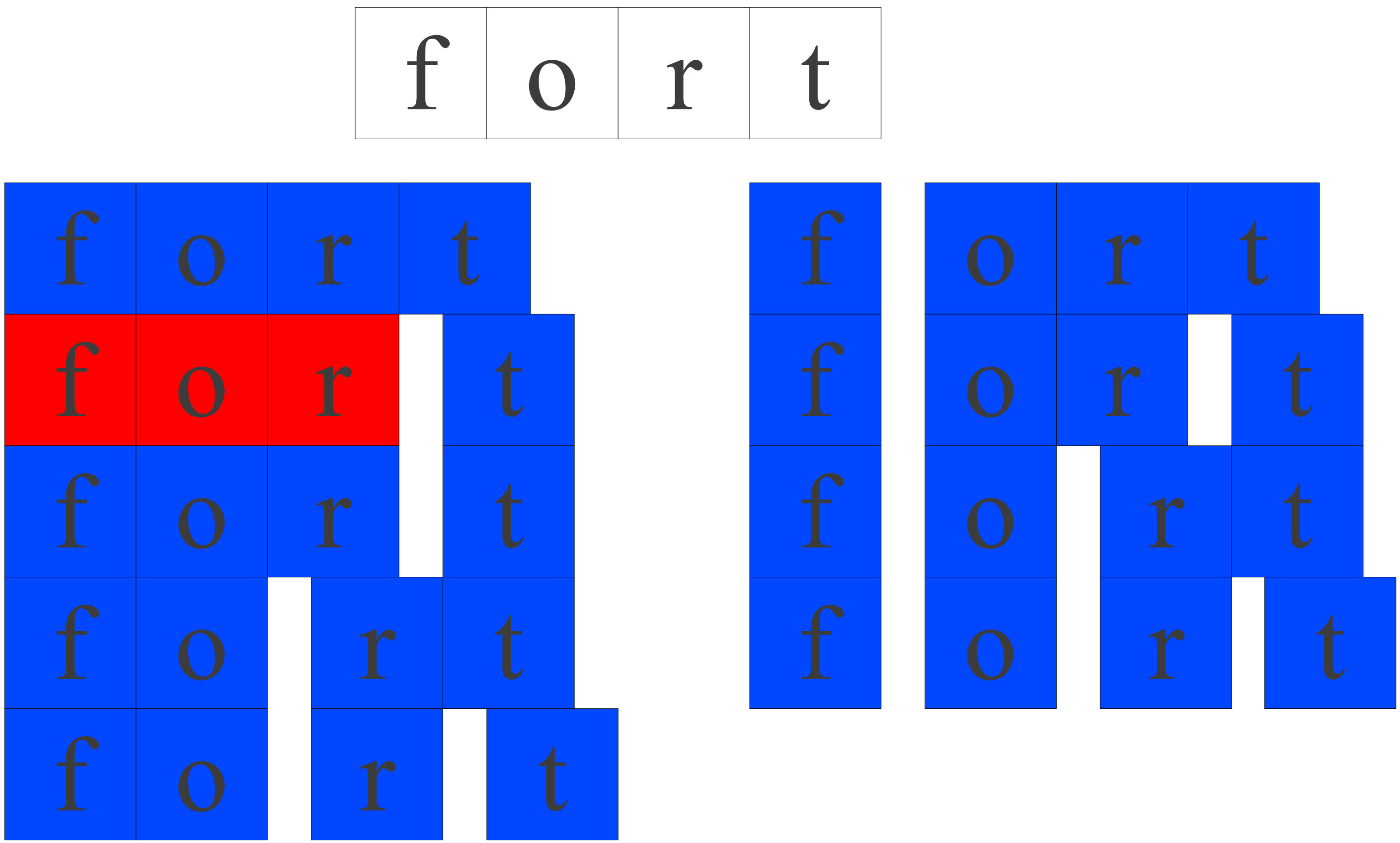


Conflict Resolution

- 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_]*
```



Lexing Ambiguities

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

f	o	r	t

f	o	r	t

Implementing Maximal Munch

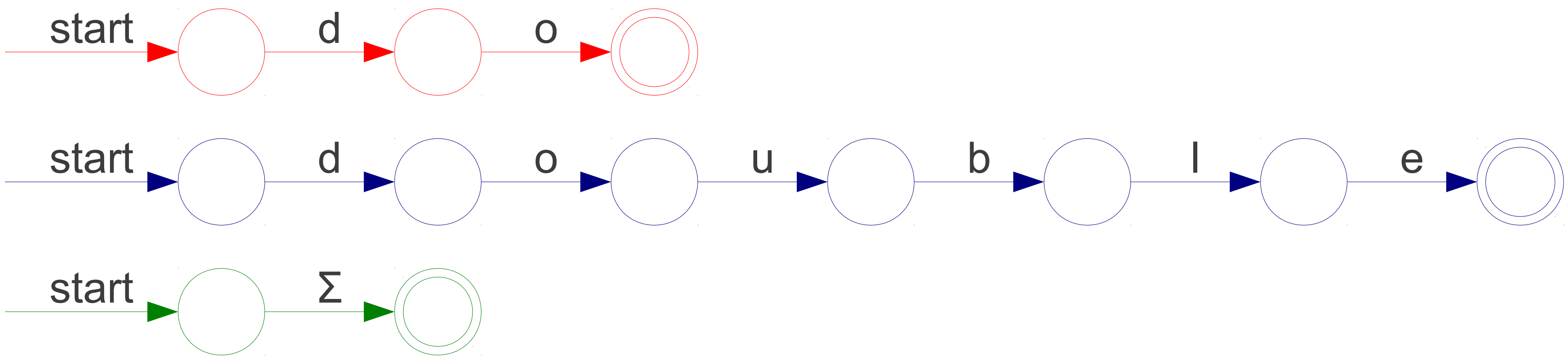
- Given a set of regular expressions, how can we use them to implement maximum munch?
- Idea:
  - Convert expressions to NFAs.
  - Run all NFAs in parallel, keeping track of the last match.
  - When all automata get stuck, report the last match and restart the search at that point.

Implementing Maximal Munch

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

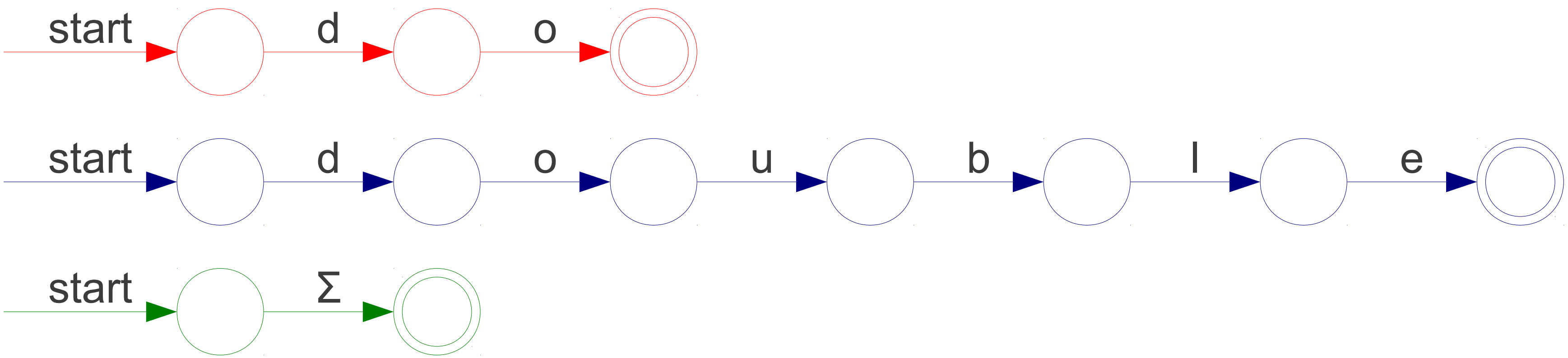
Implementing Maximal Munch

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



Implementing Maximal Munch

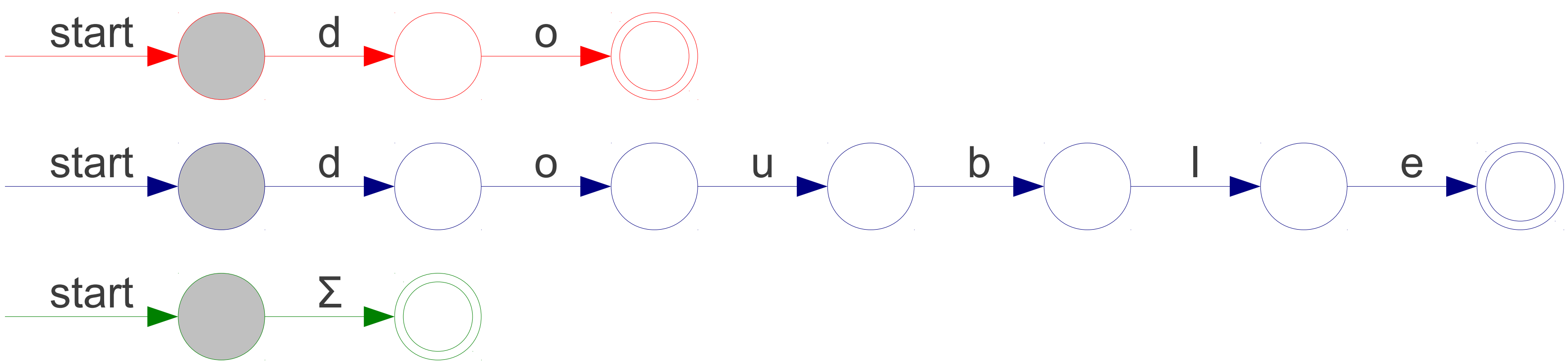
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

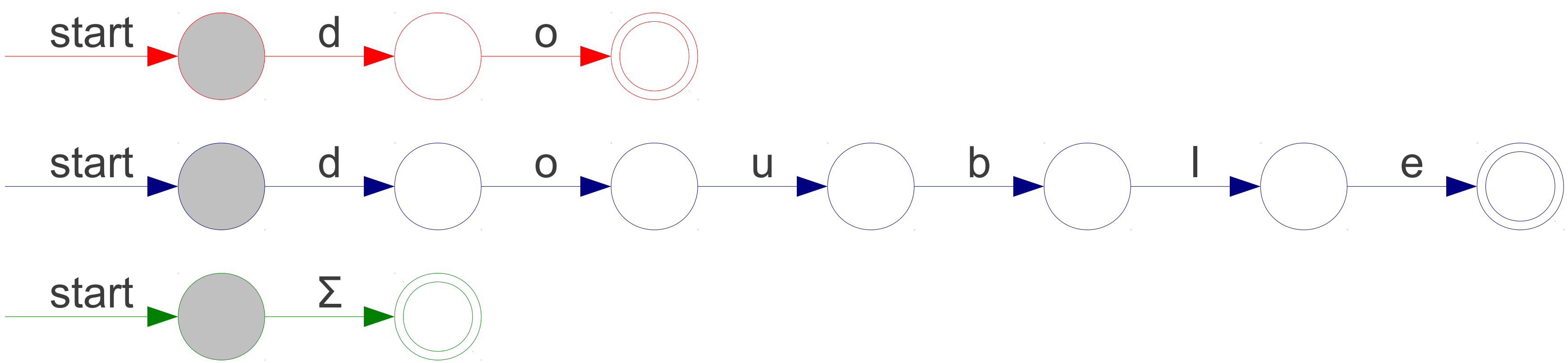
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```

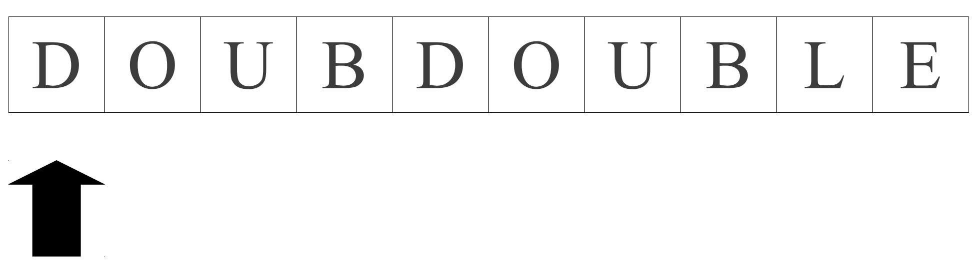


D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

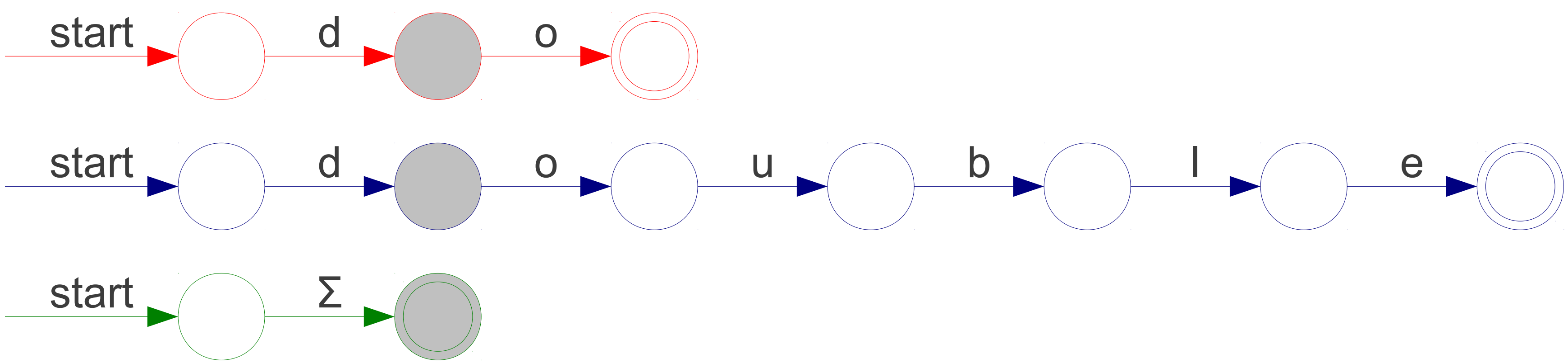
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





Implementing Maximal Munch

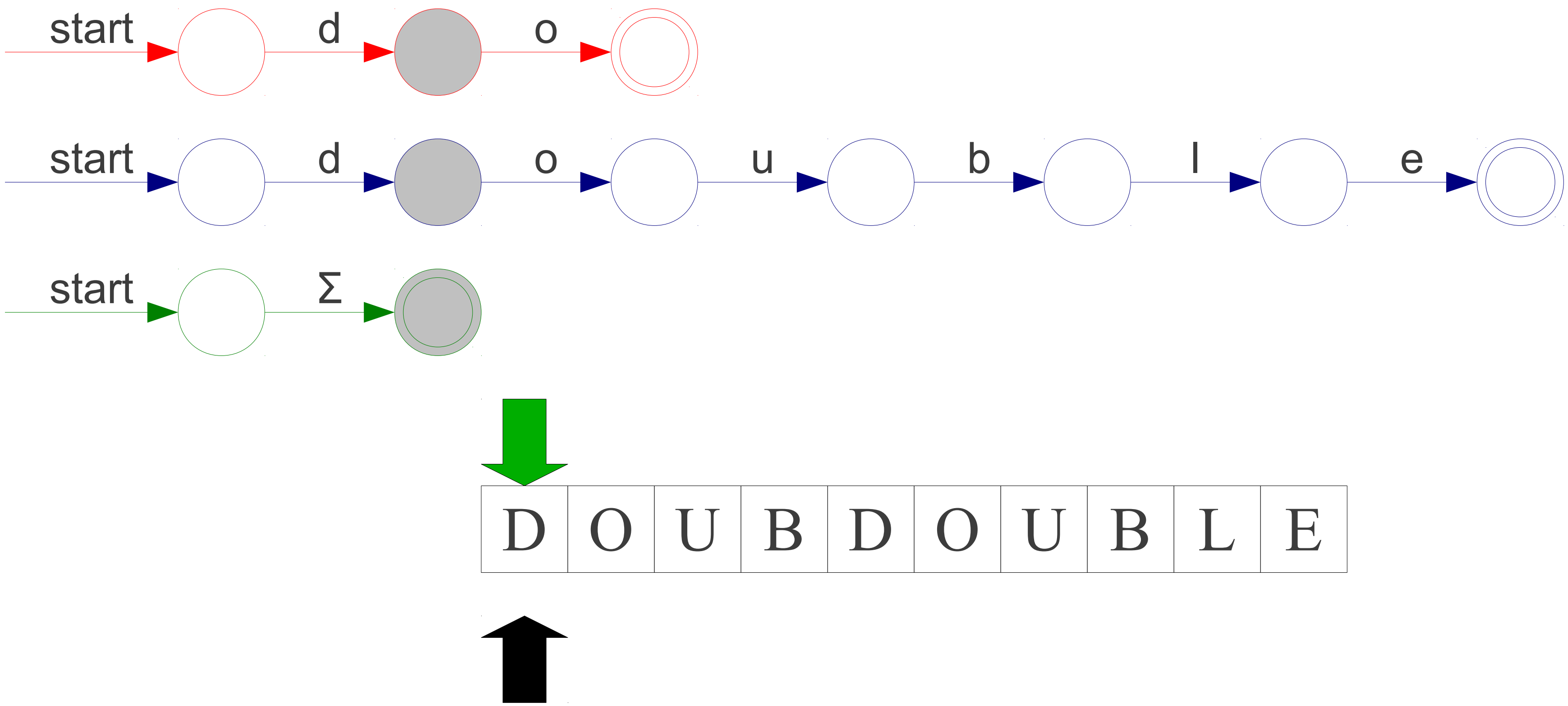
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

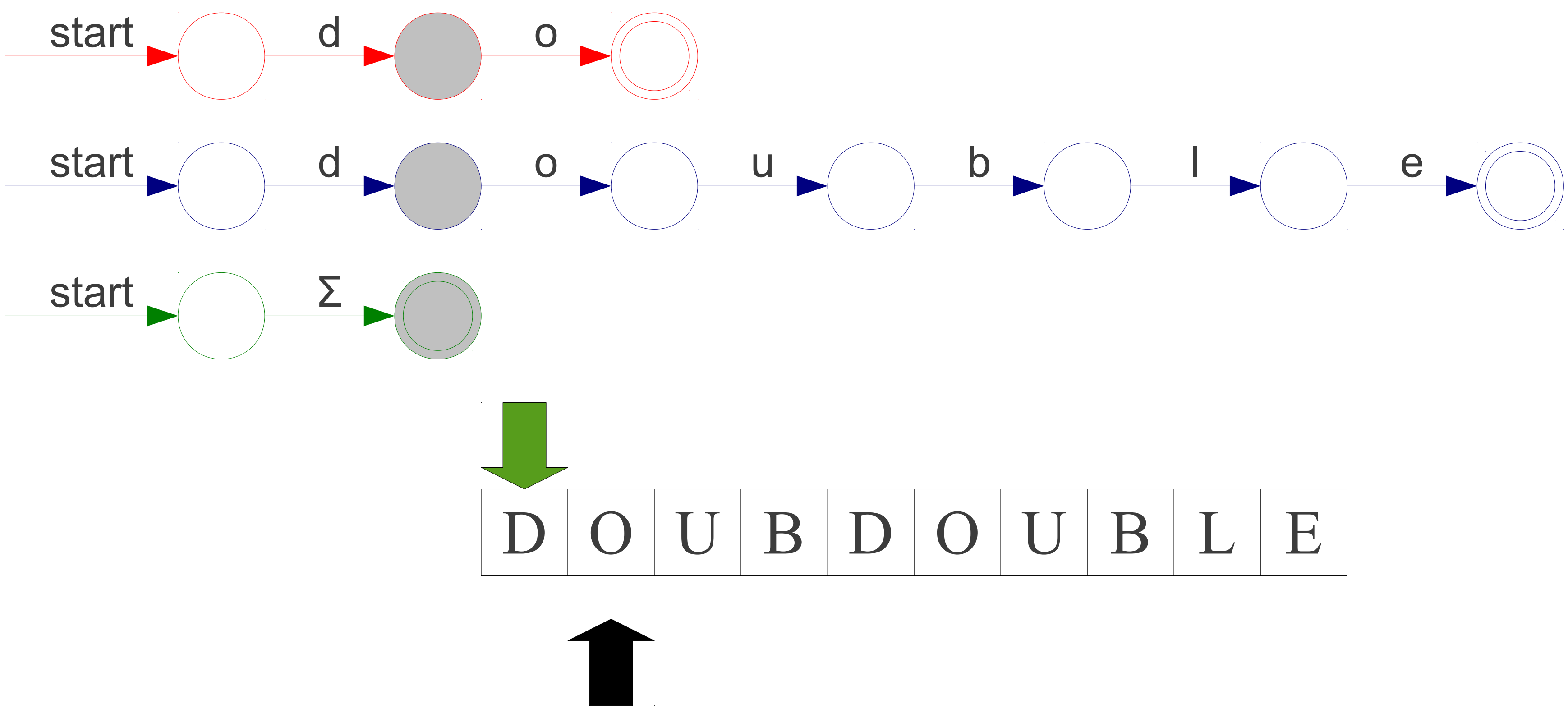
Implementing Maximal Munch

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



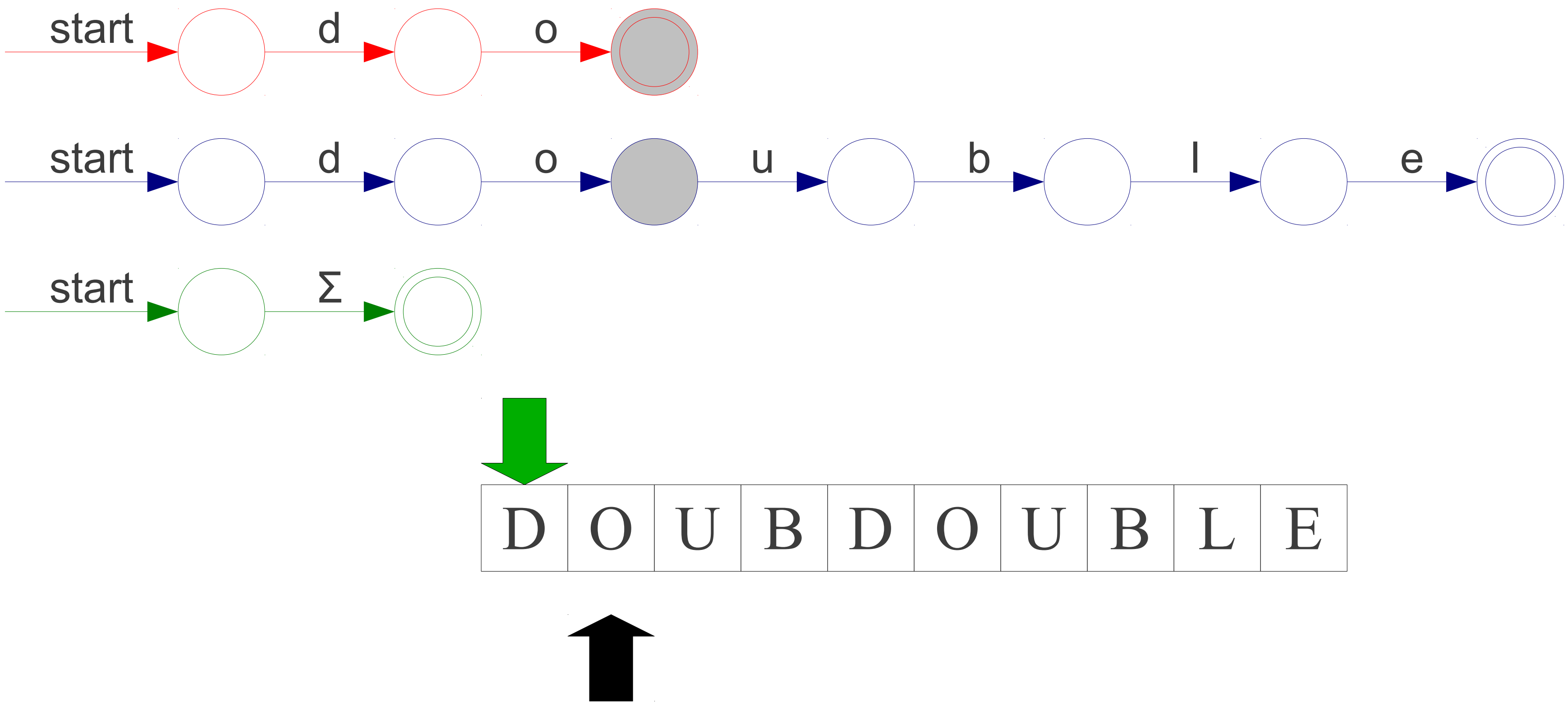
Implementing Maximal Munch

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



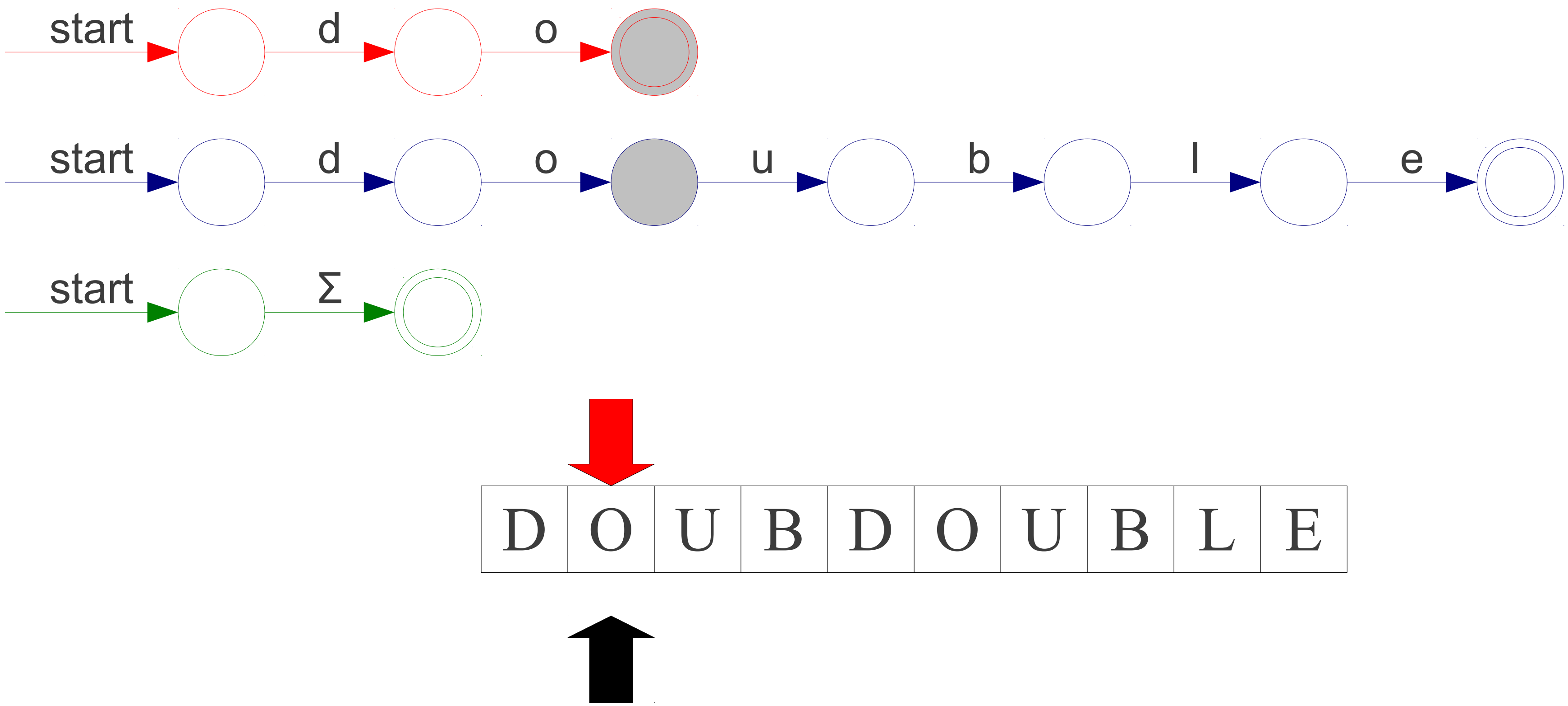
Implementing Maximal Munch

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



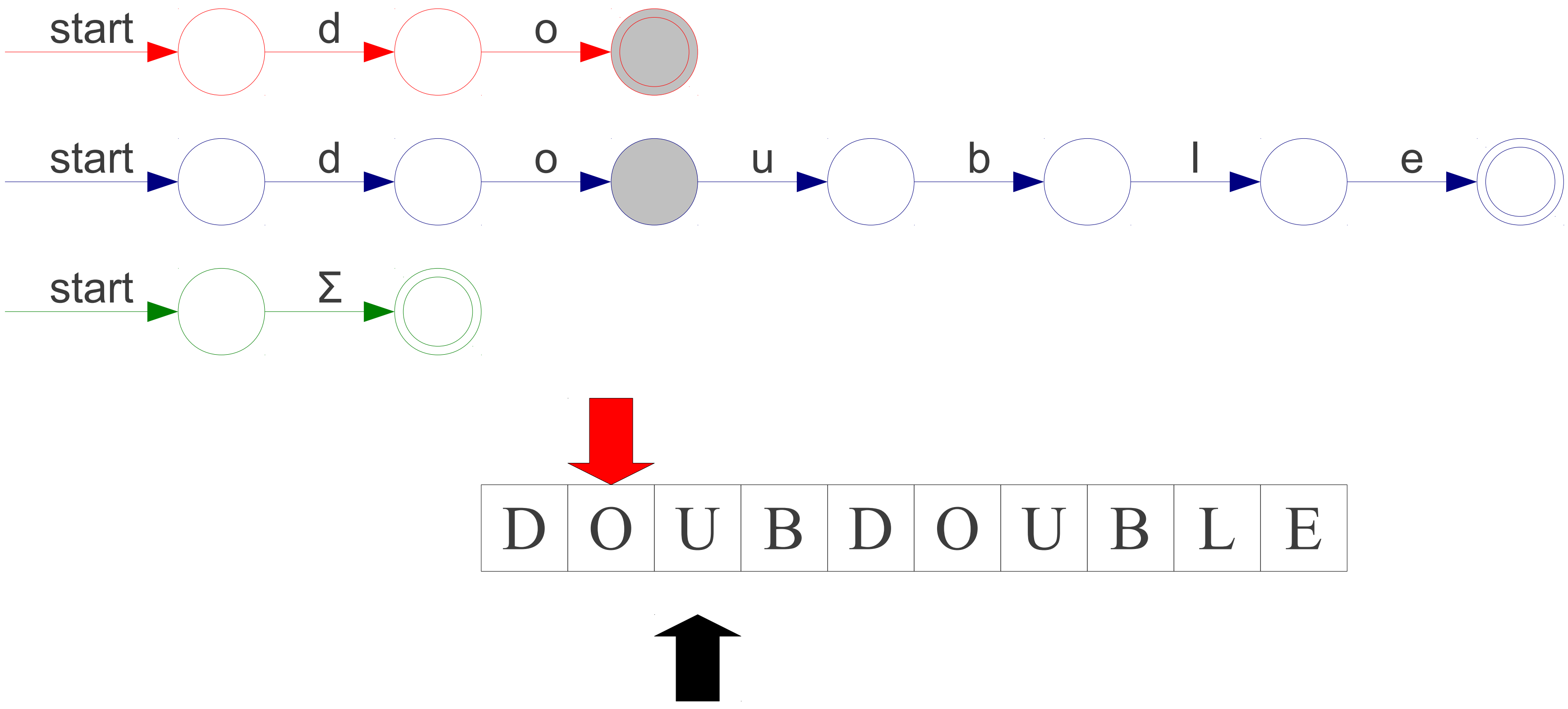
Implementing Maximal Munch

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



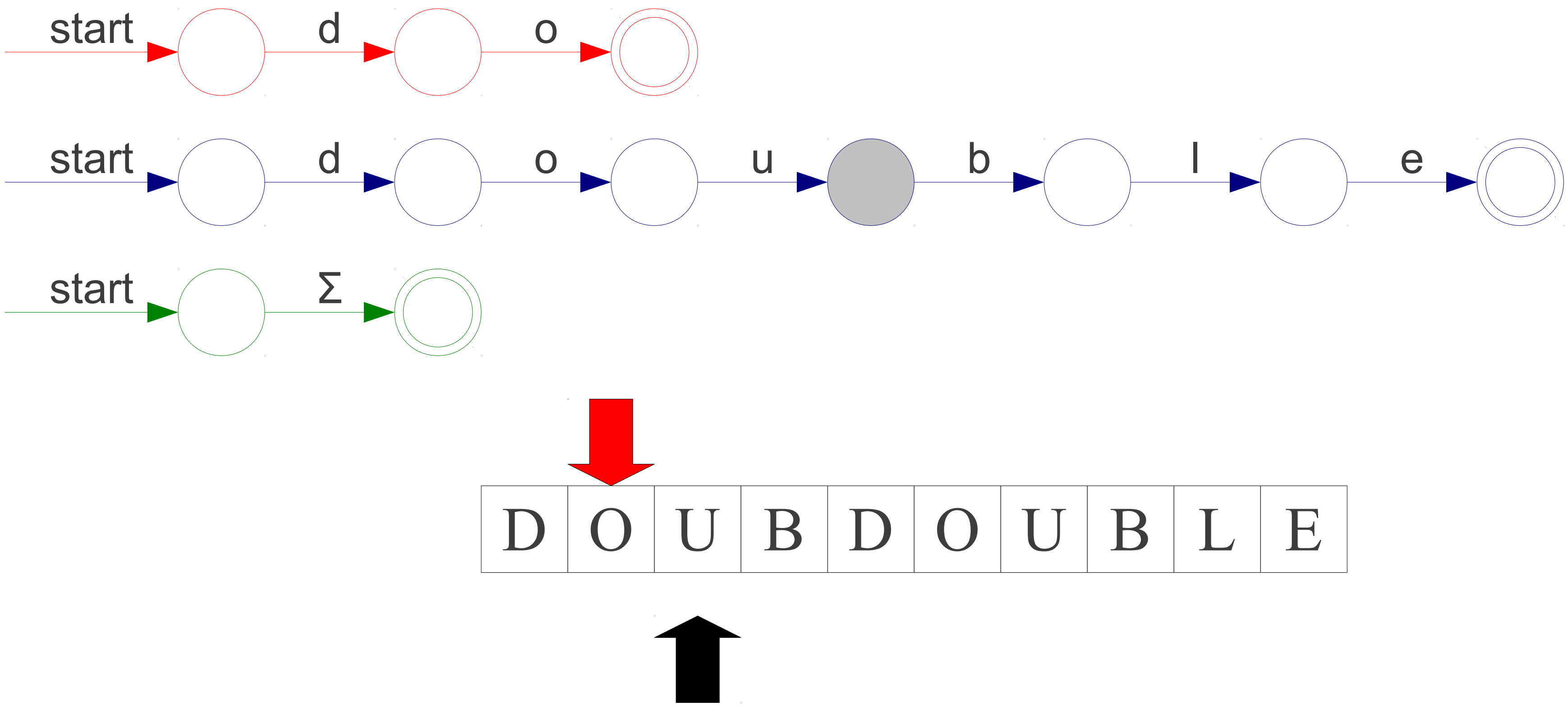
Implementing Maximal Munch

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



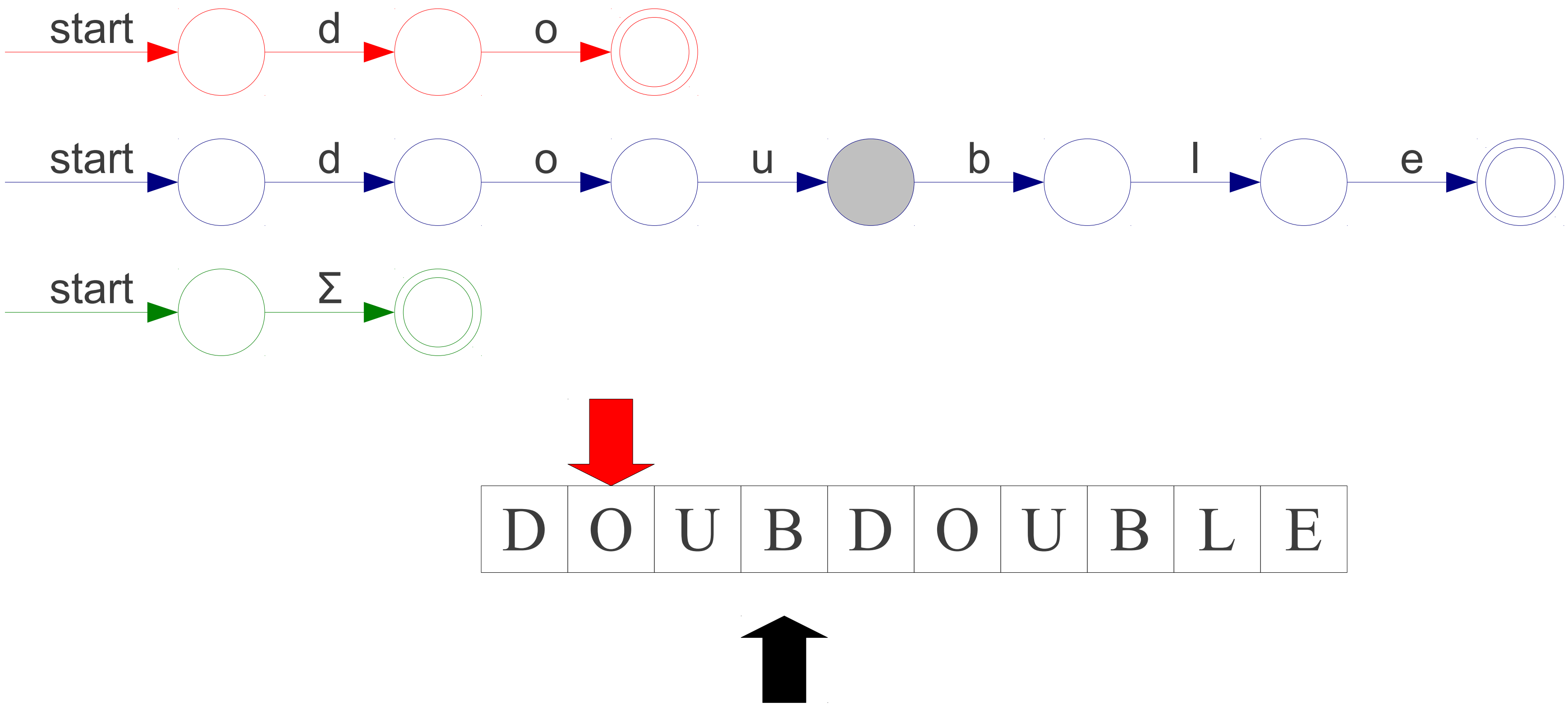
Implementing Maximal Munch

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



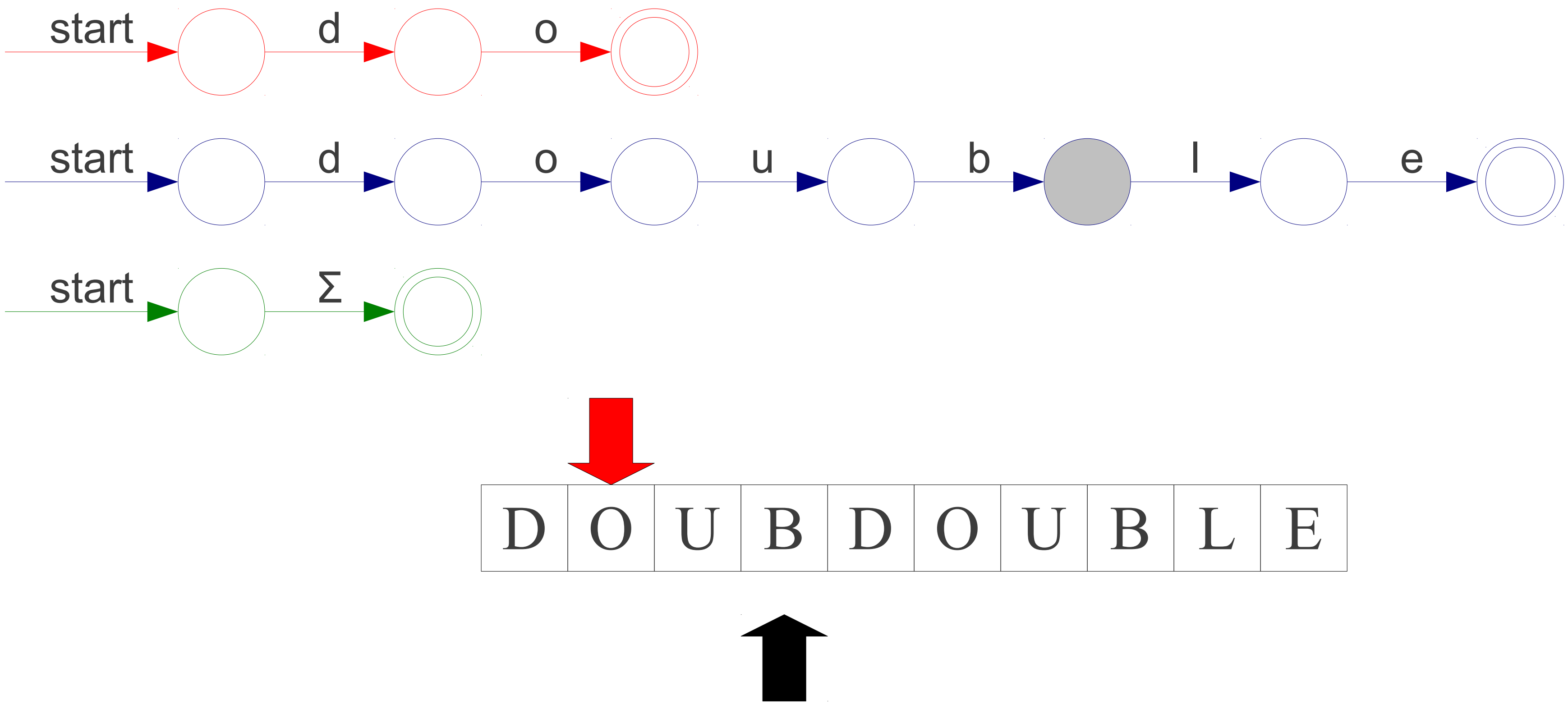
Implementing Maximal Munch

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



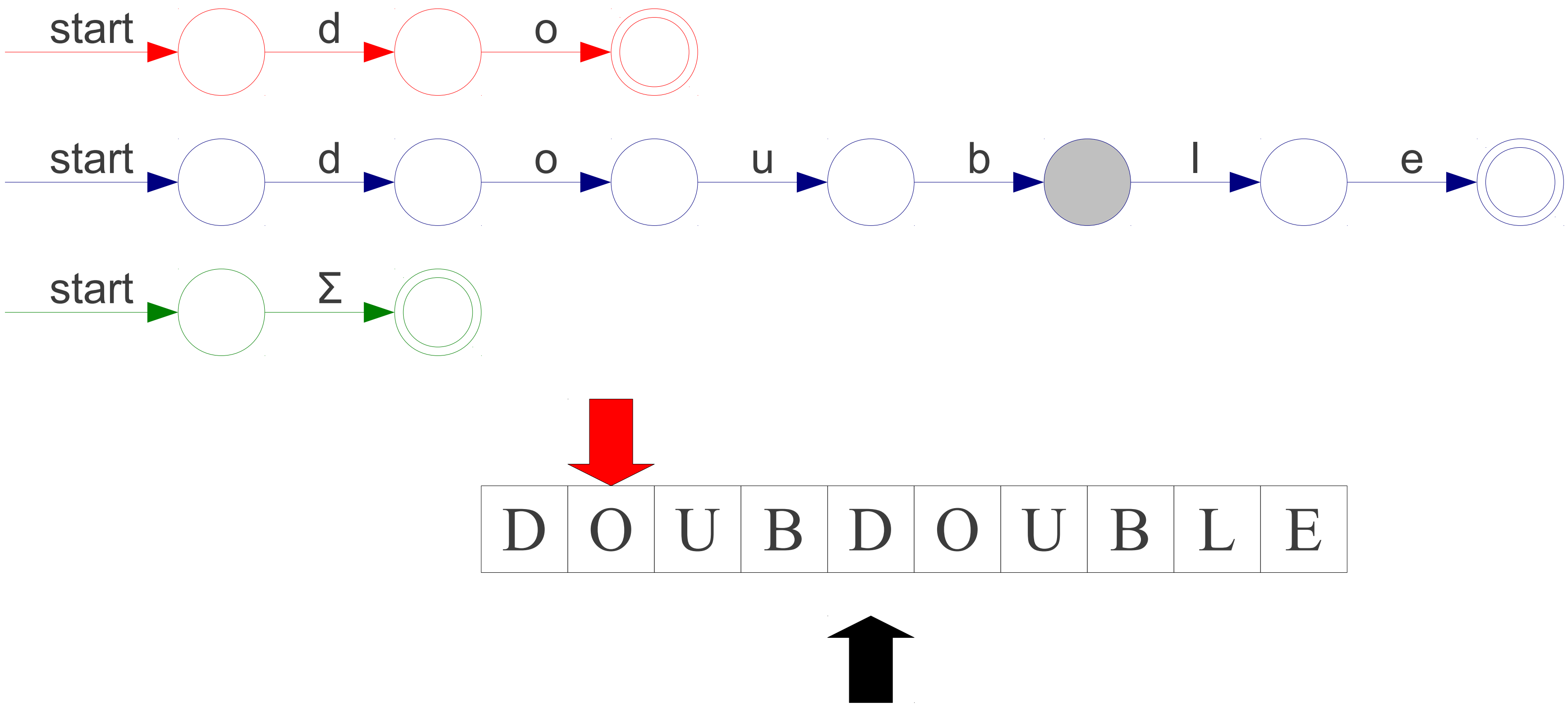
Implementing Maximal Munch

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



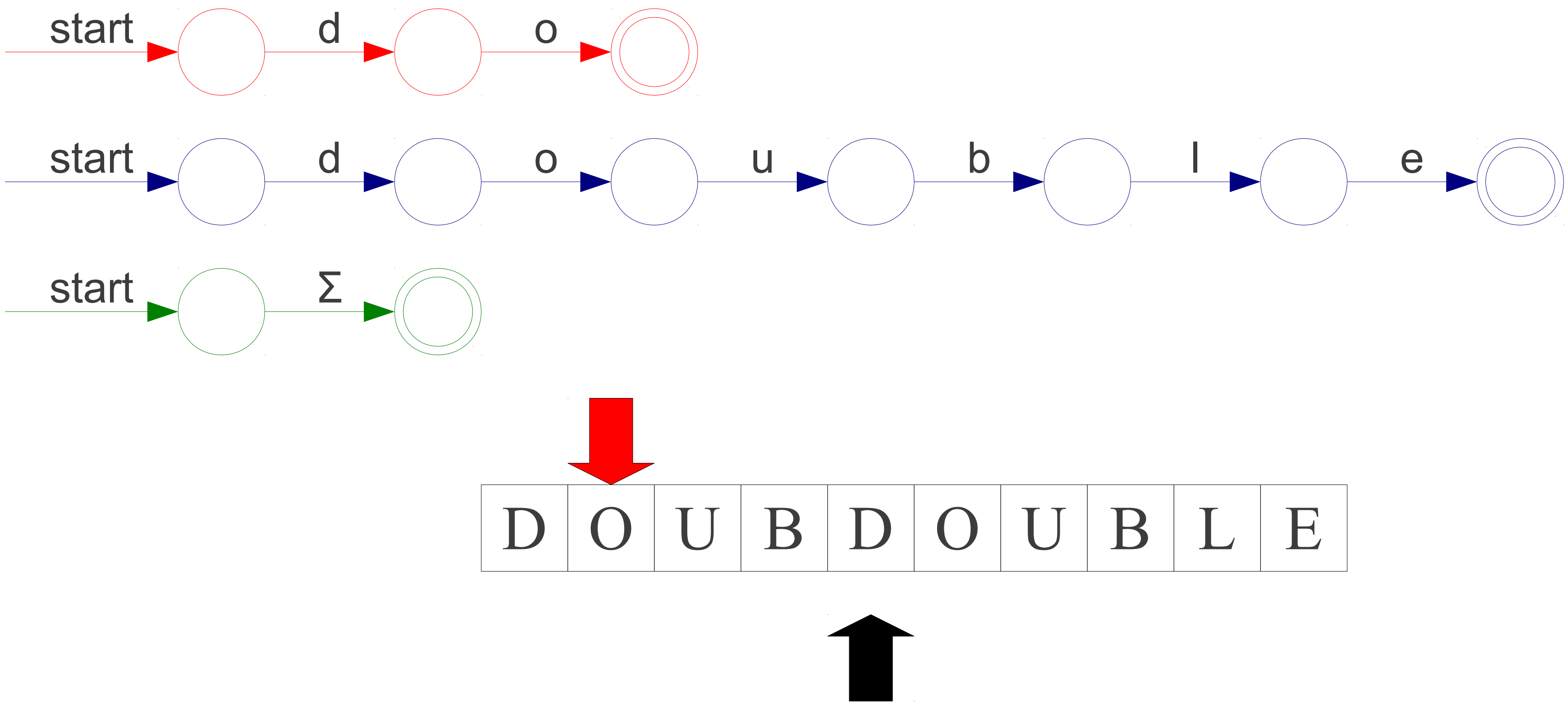
Implementing Maximal Munch

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



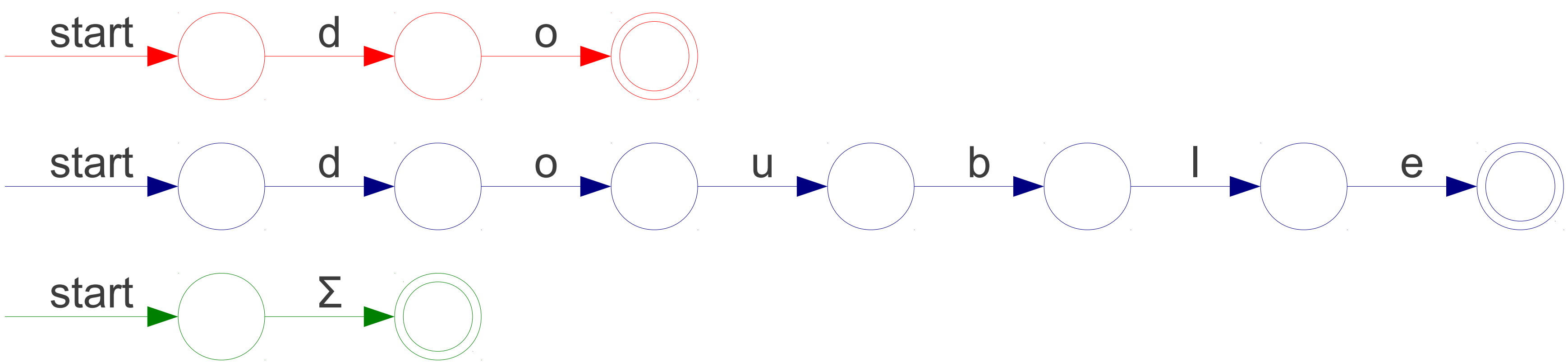
Implementing Maximal Munch

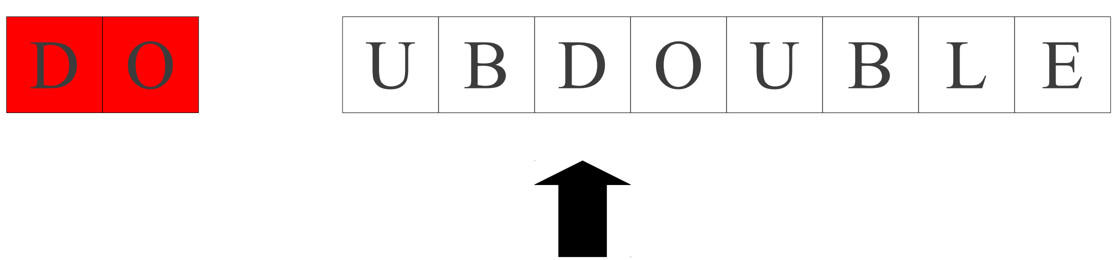
```
T_Do         do
T_Double     double
T_Mystery    [A-Za-z]
```



Implementing Maximal Munch

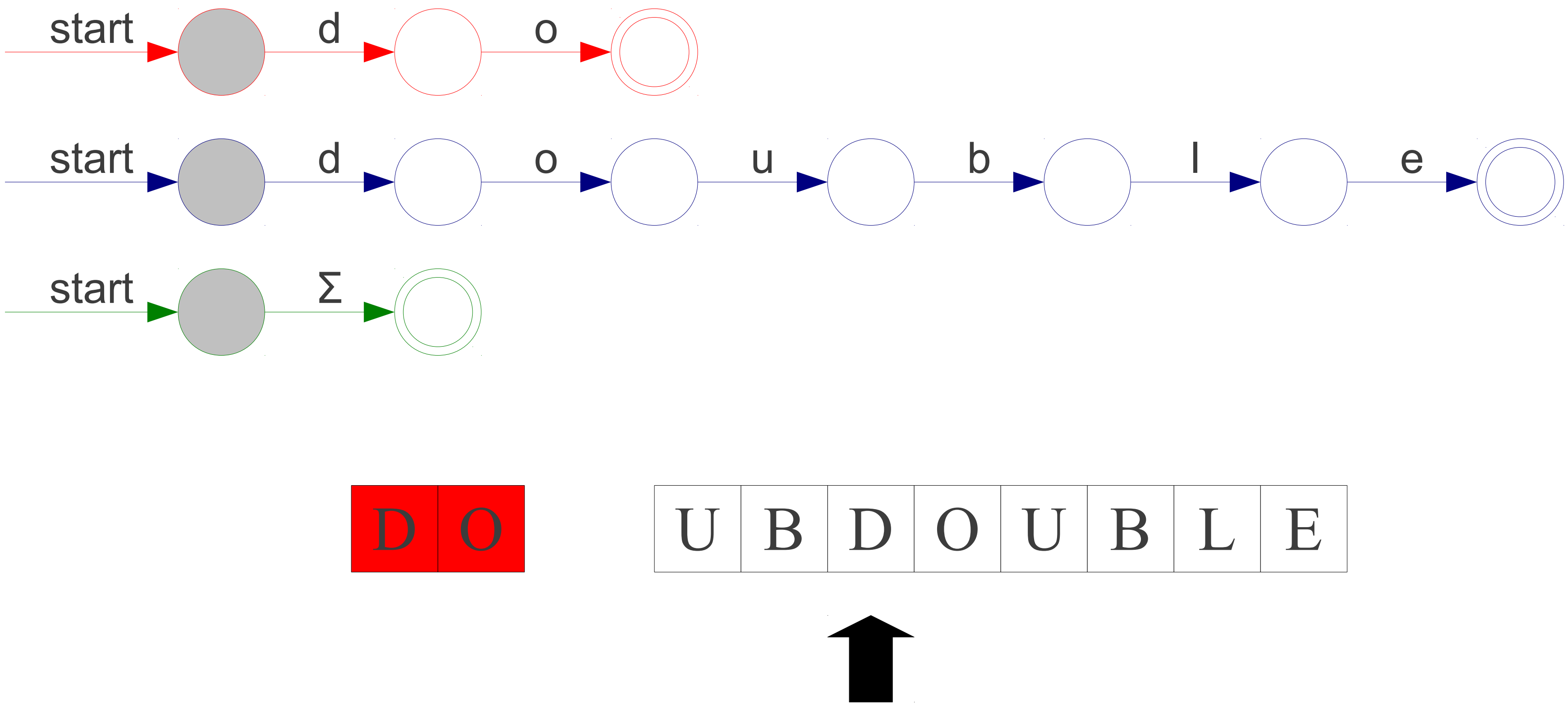
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





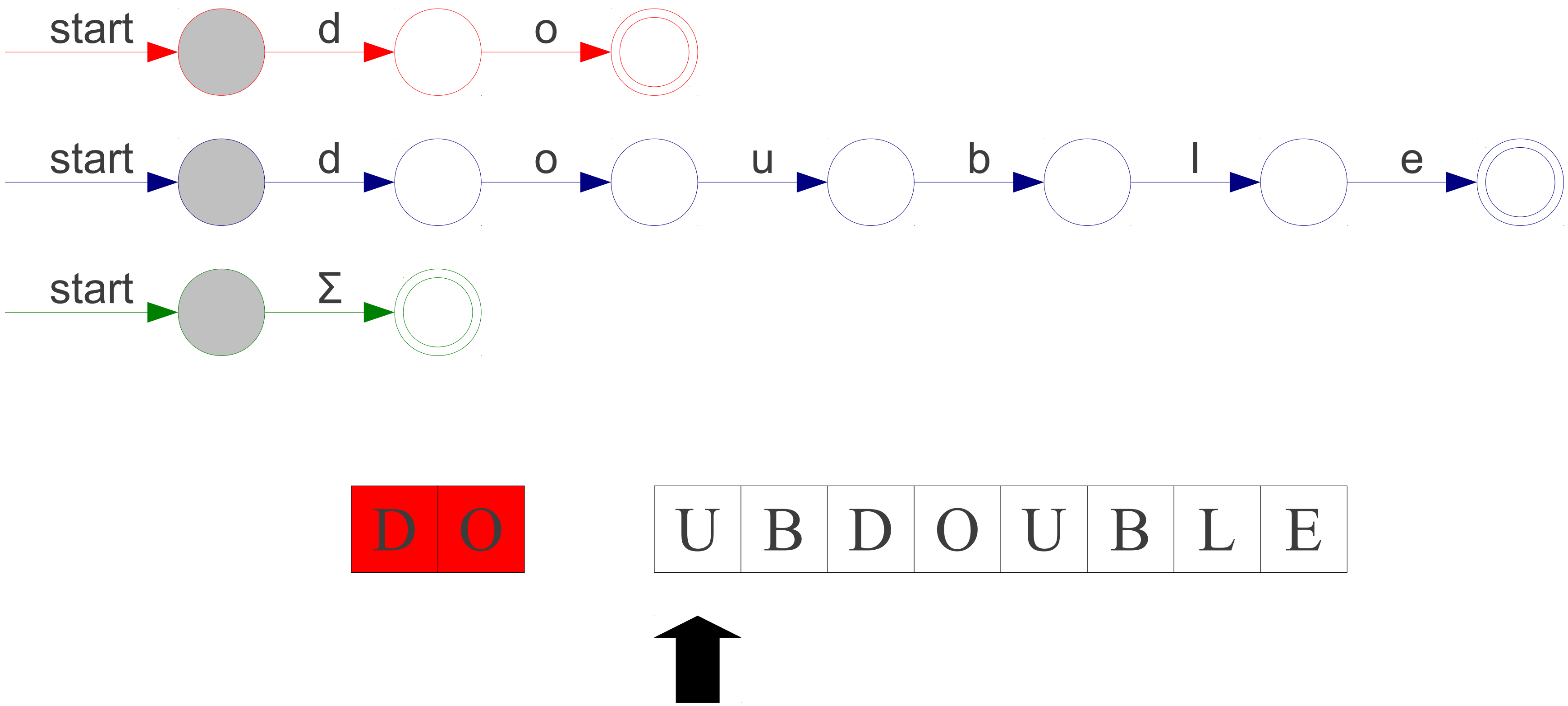
Implementing Maximal Munch

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



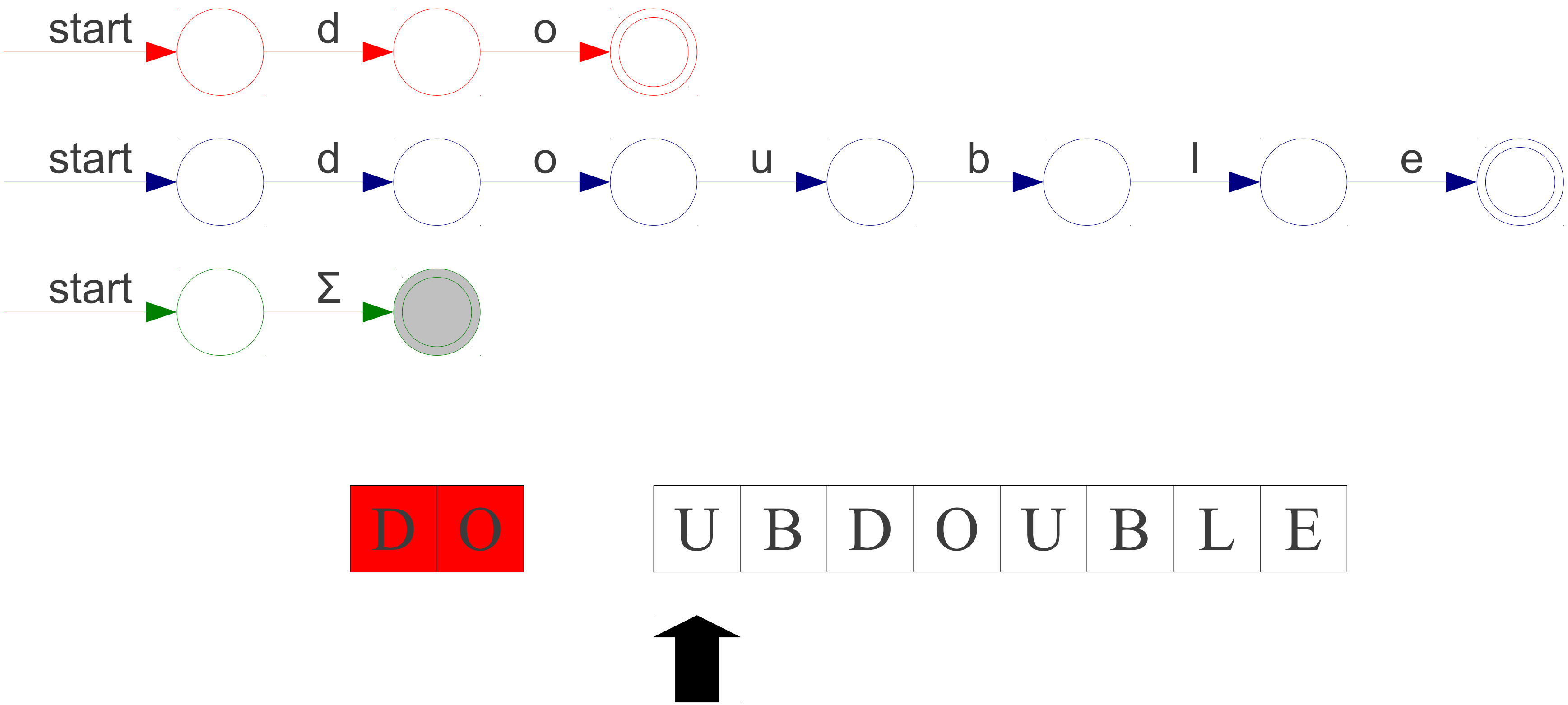
Implementing Maximal Munch

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



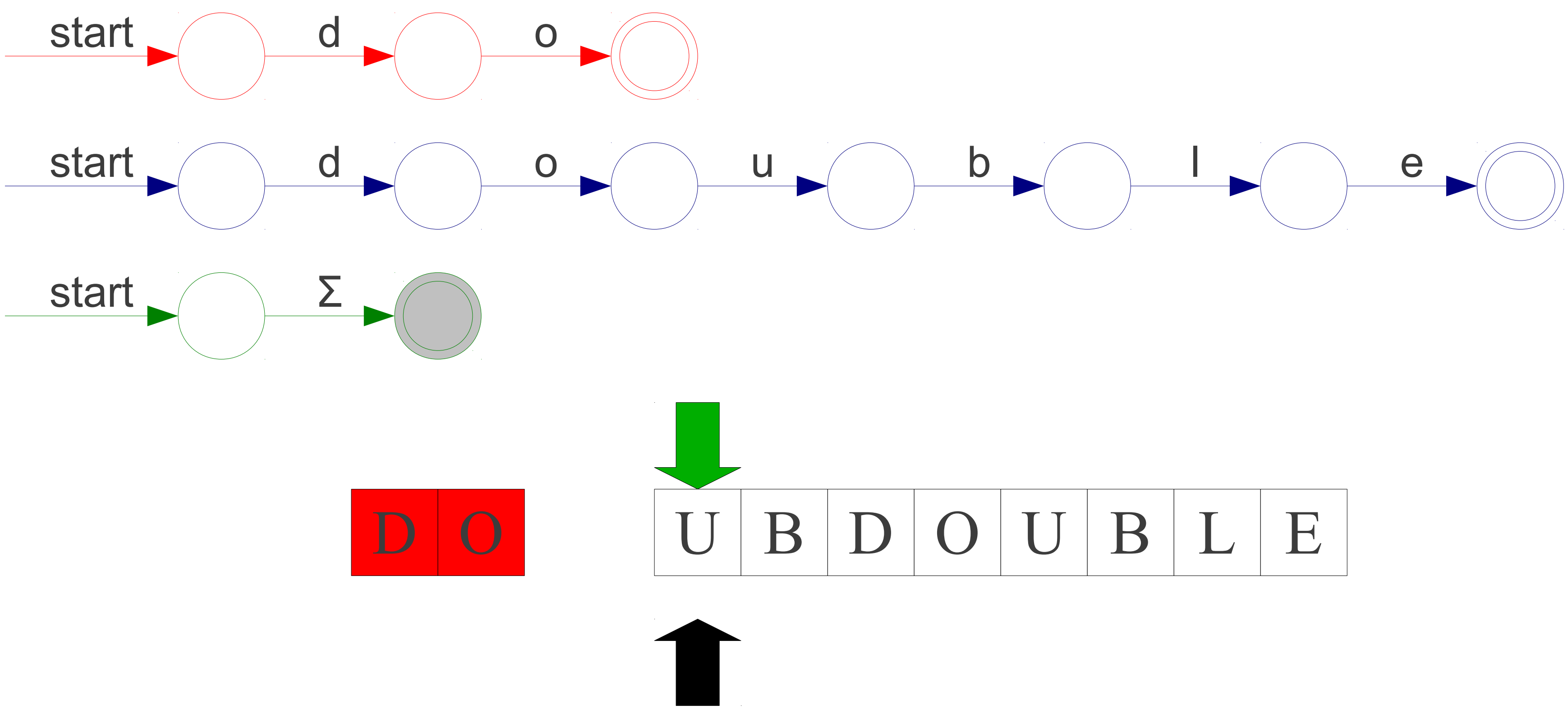
Implementing Maximal Munch

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



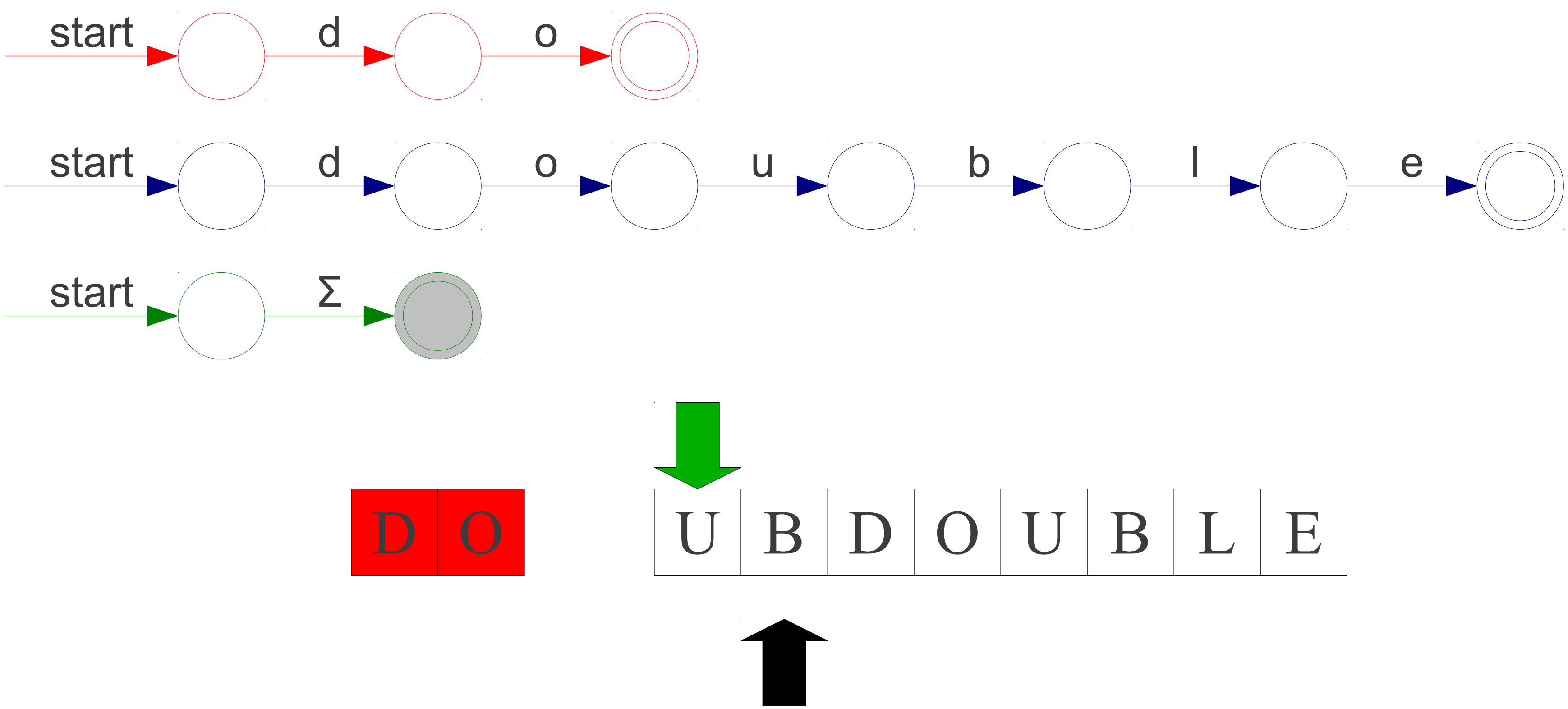
Implementing Maximal Munch

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



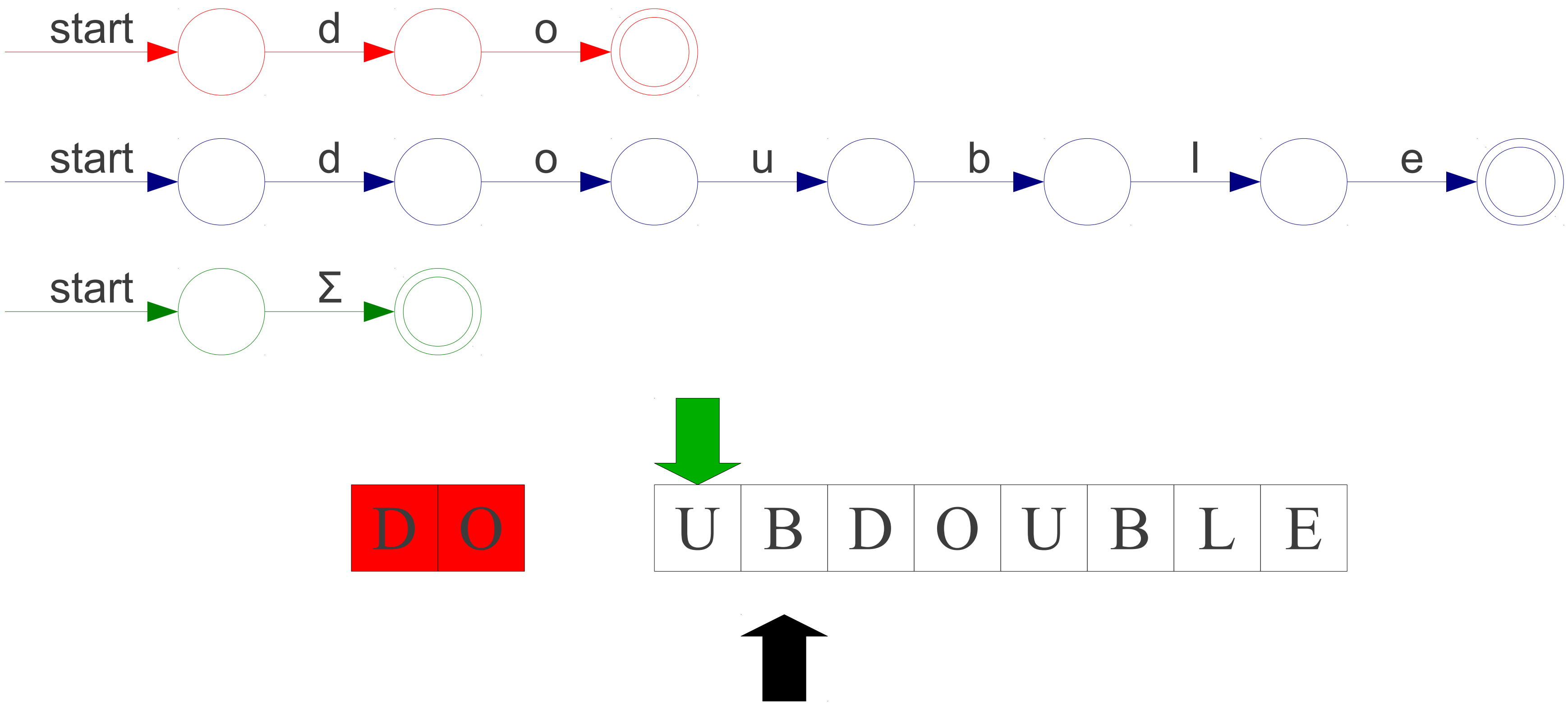
Implementing Maximal Munch

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



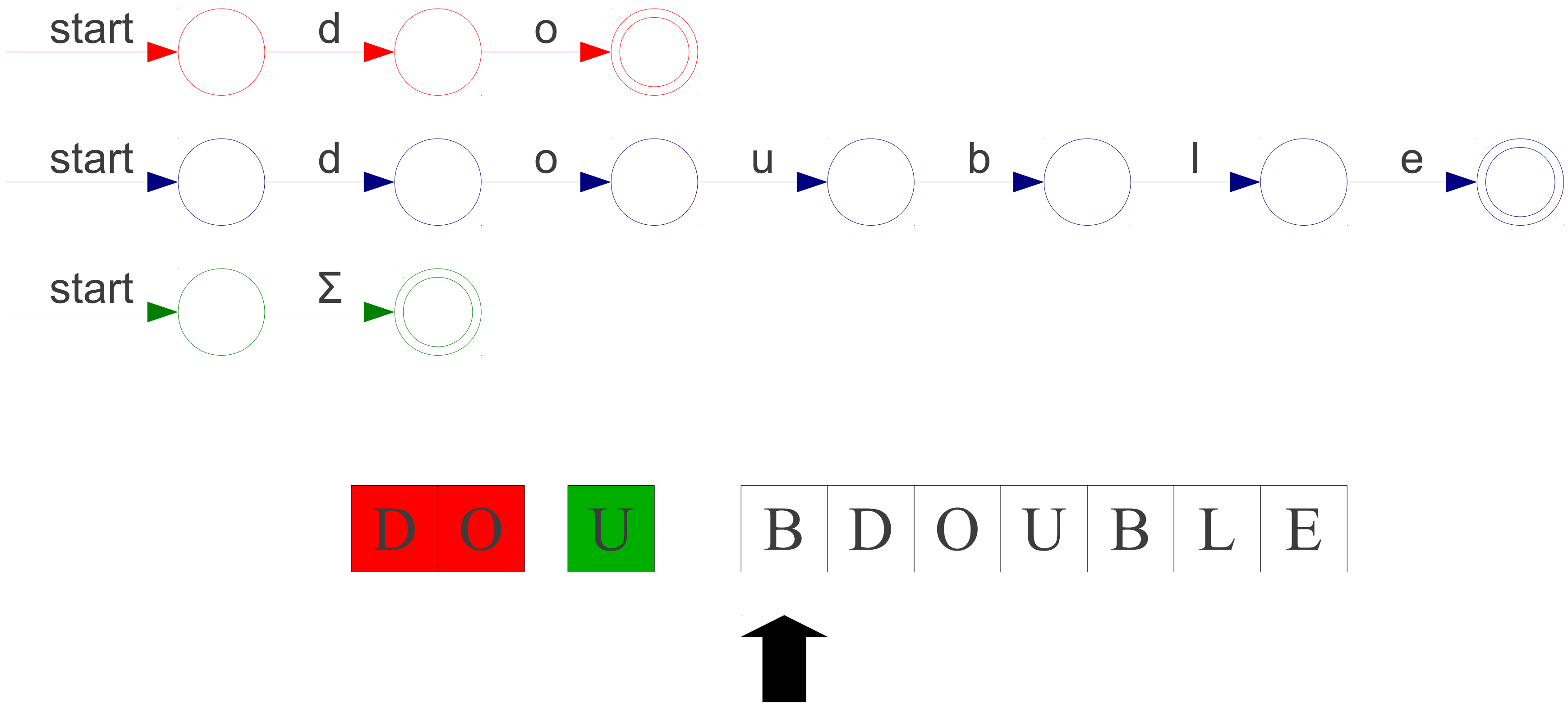
Implementing Maximal Munch

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



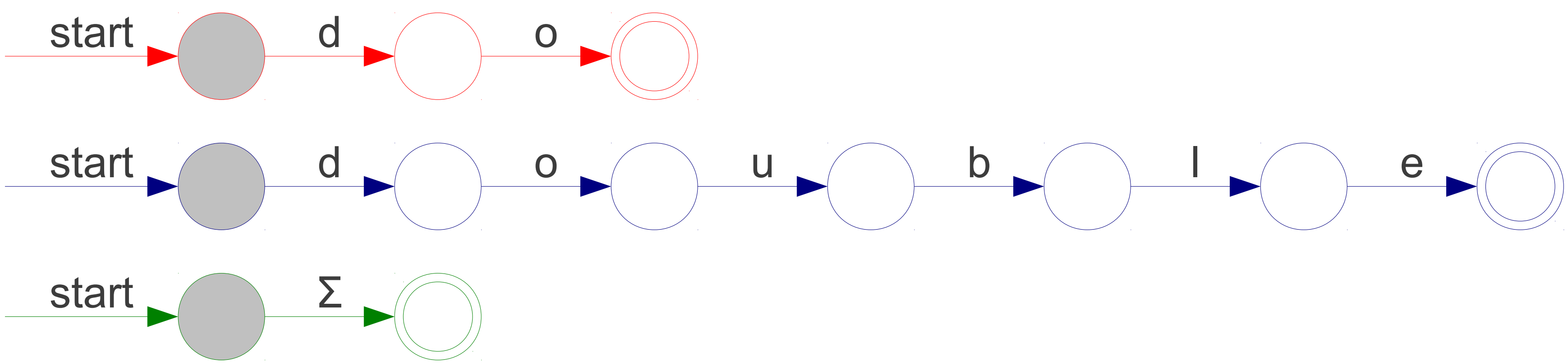
Implementing Maximal Munch

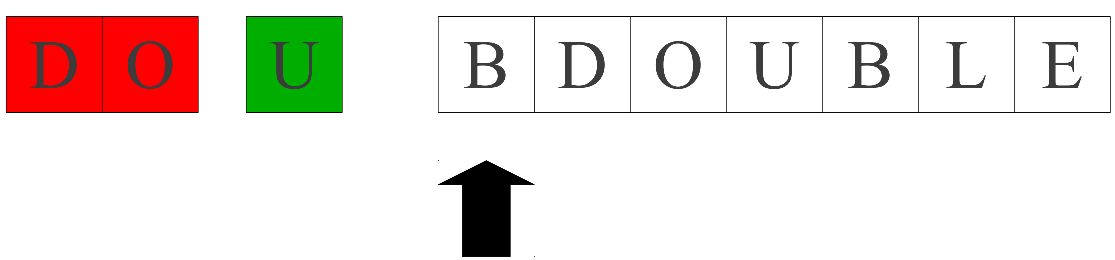
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

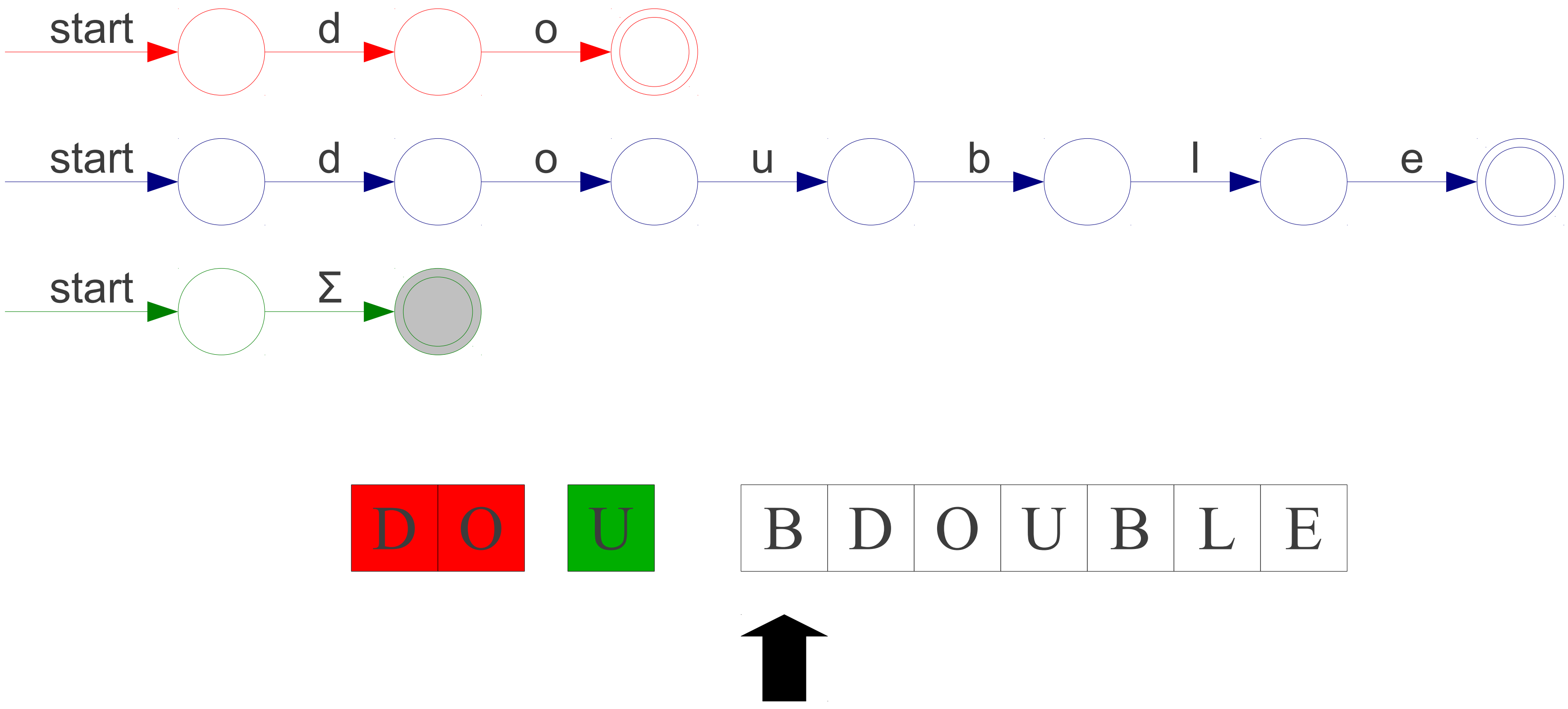
```
T_Do         do
T_Double     double
T_Mystery    [A-Za-z]
```





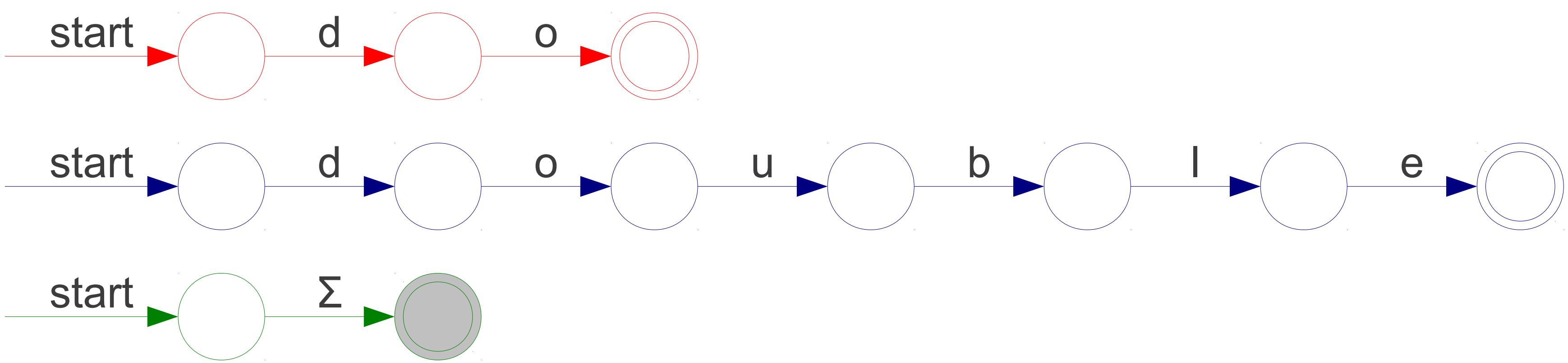
Implementing Maximal Munch

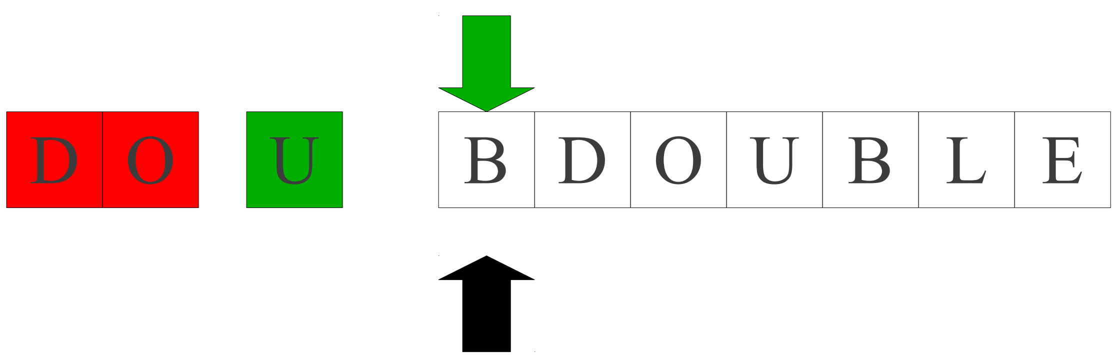
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

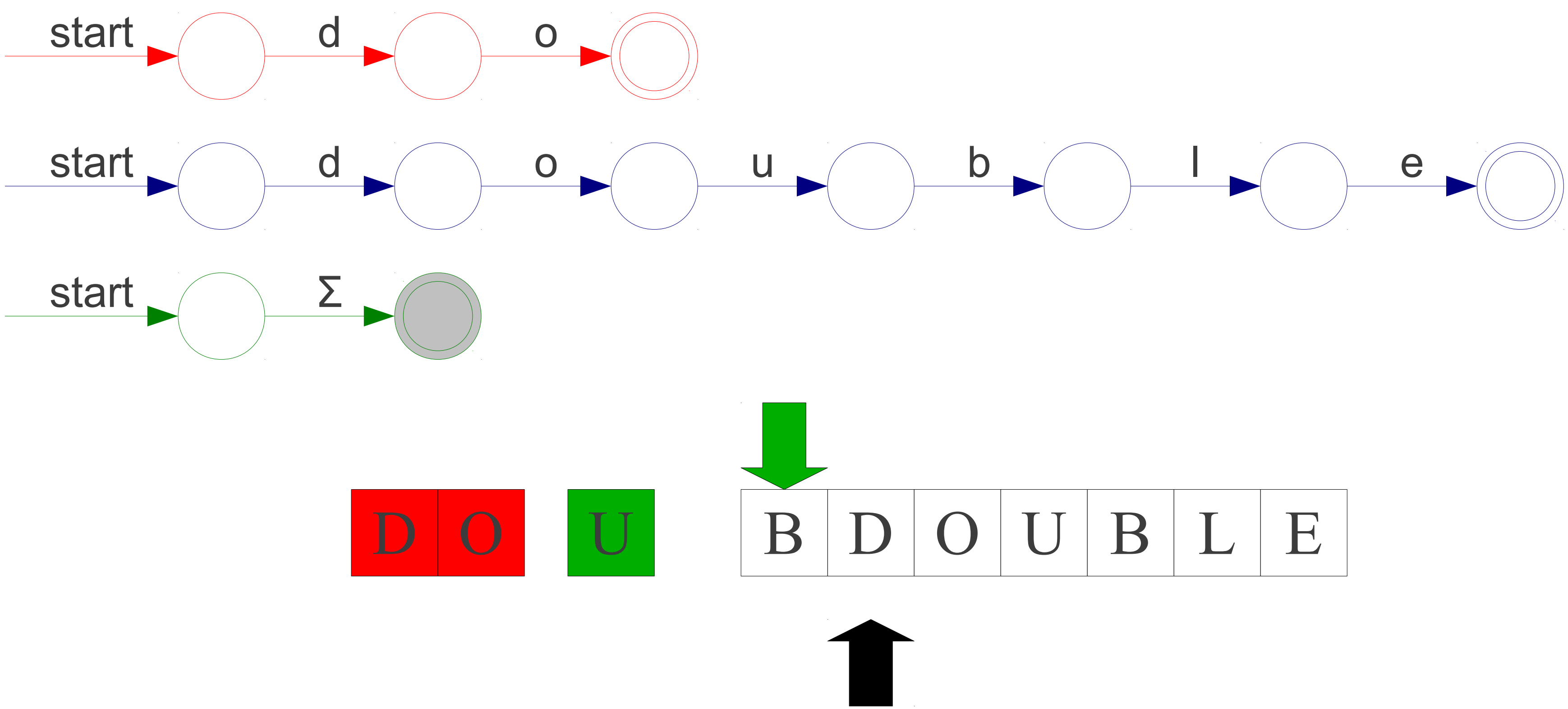
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





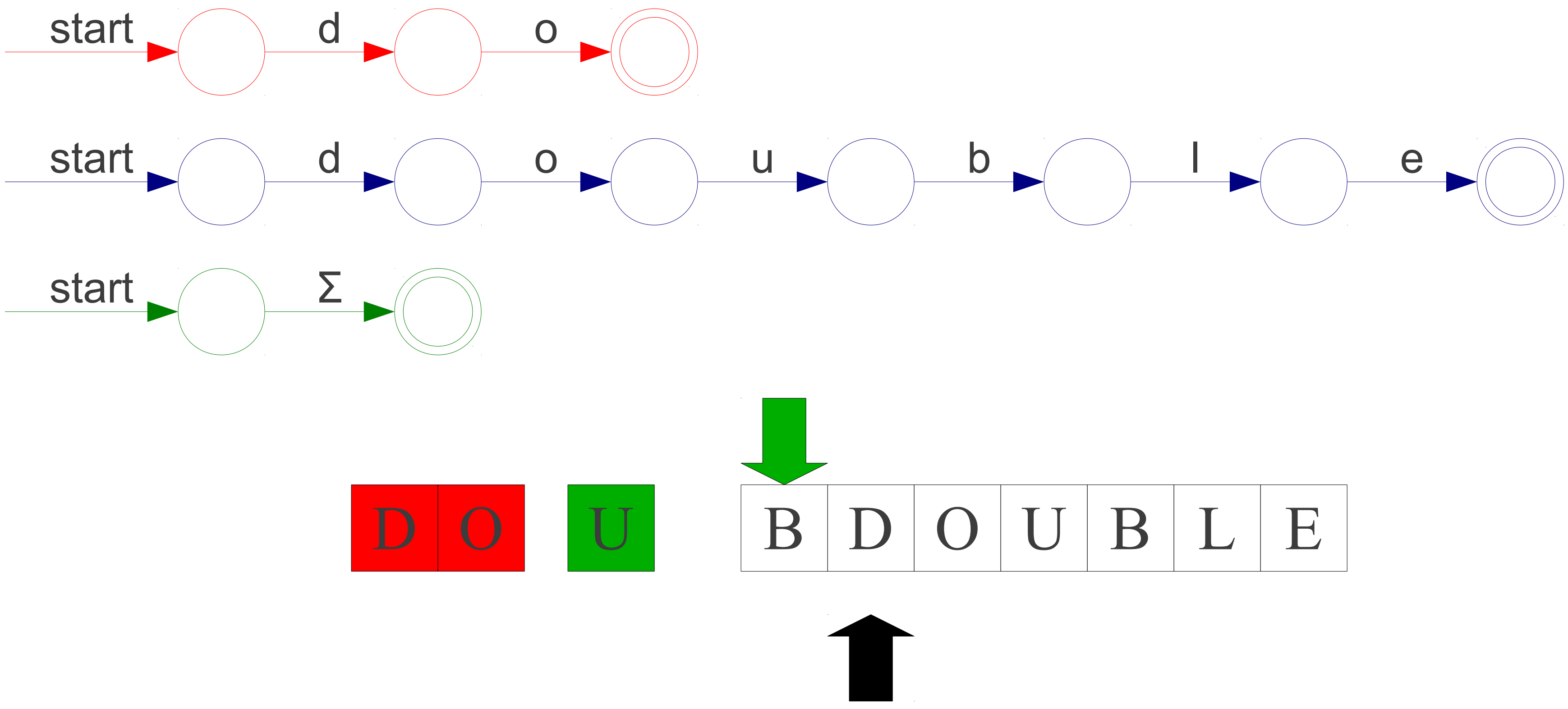
Implementing Maximal Munch

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



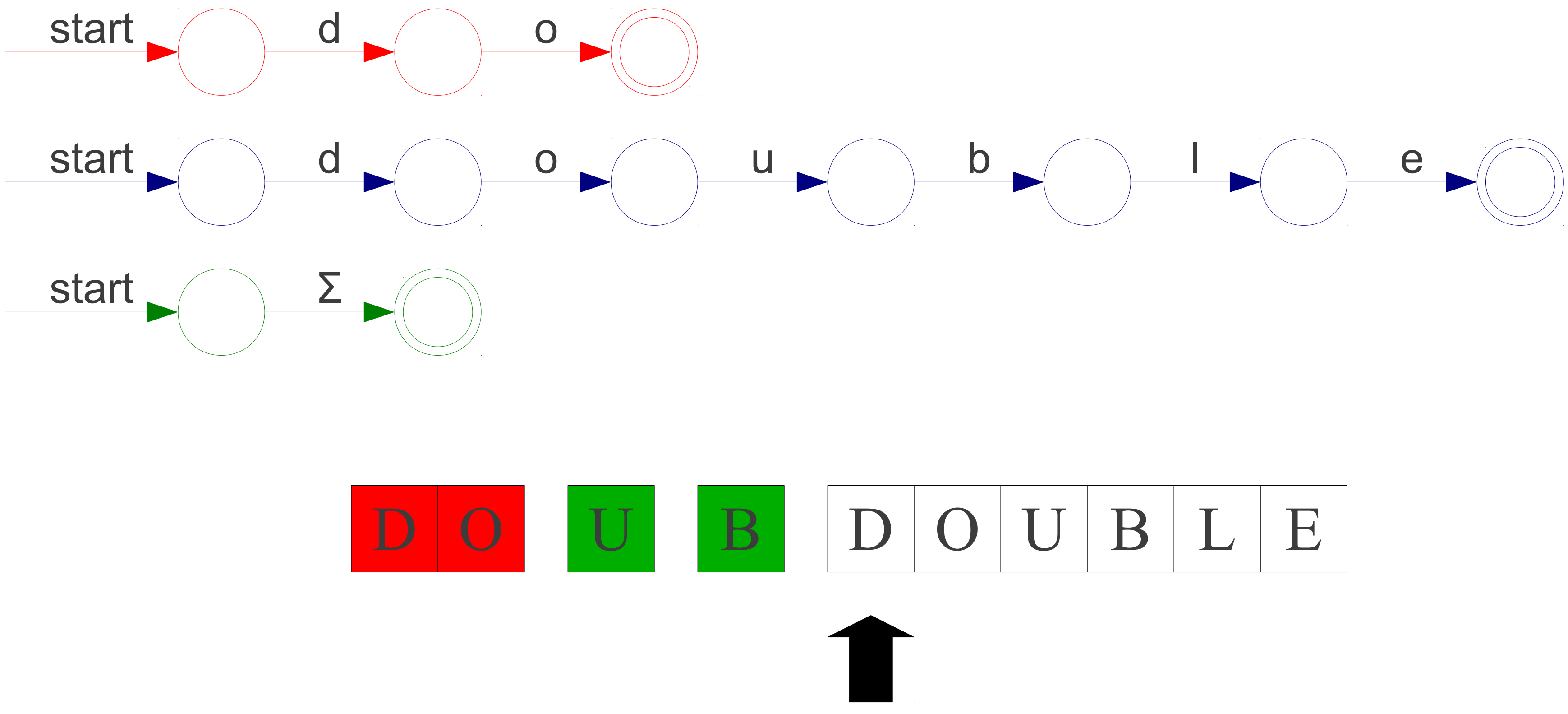
Implementing Maximal Munch

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



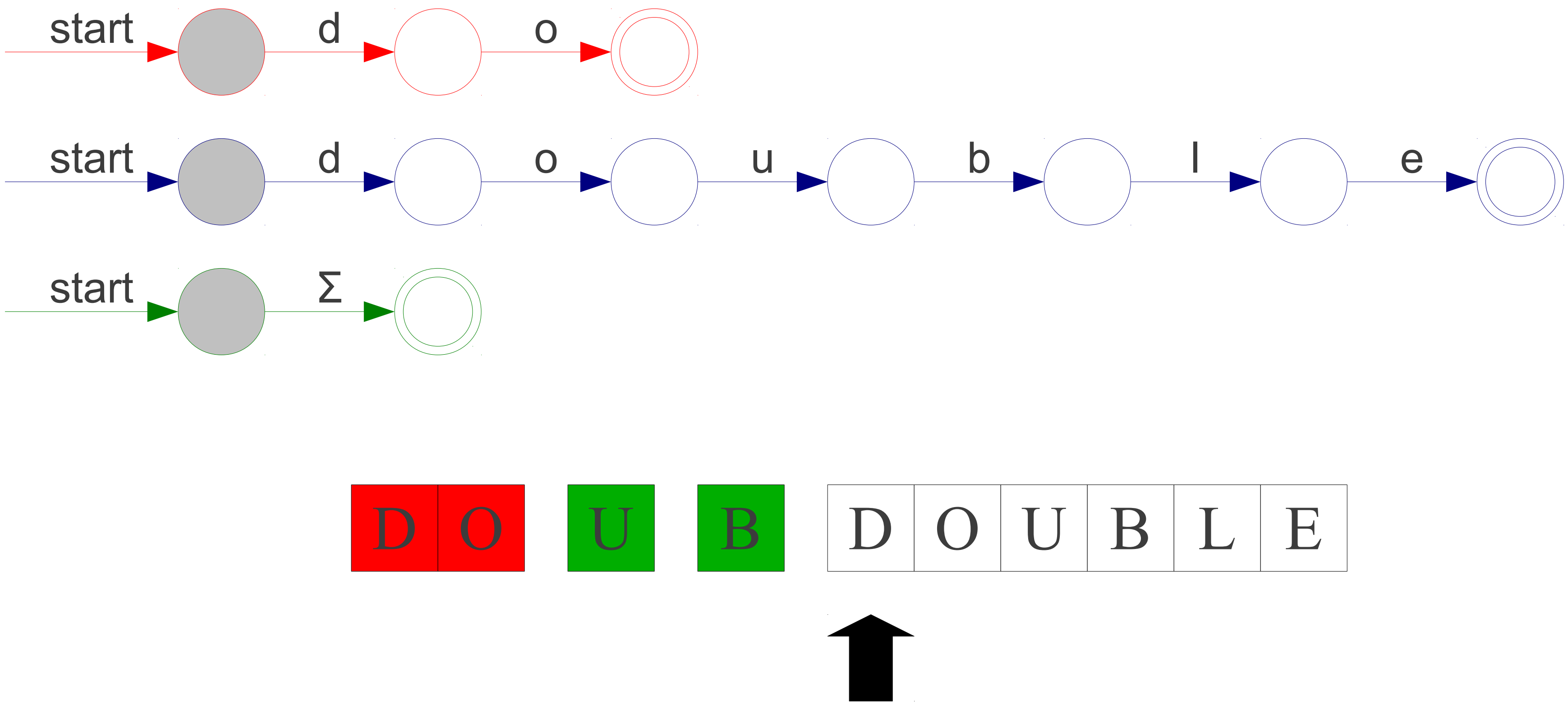
Implementing Maximal Munch

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



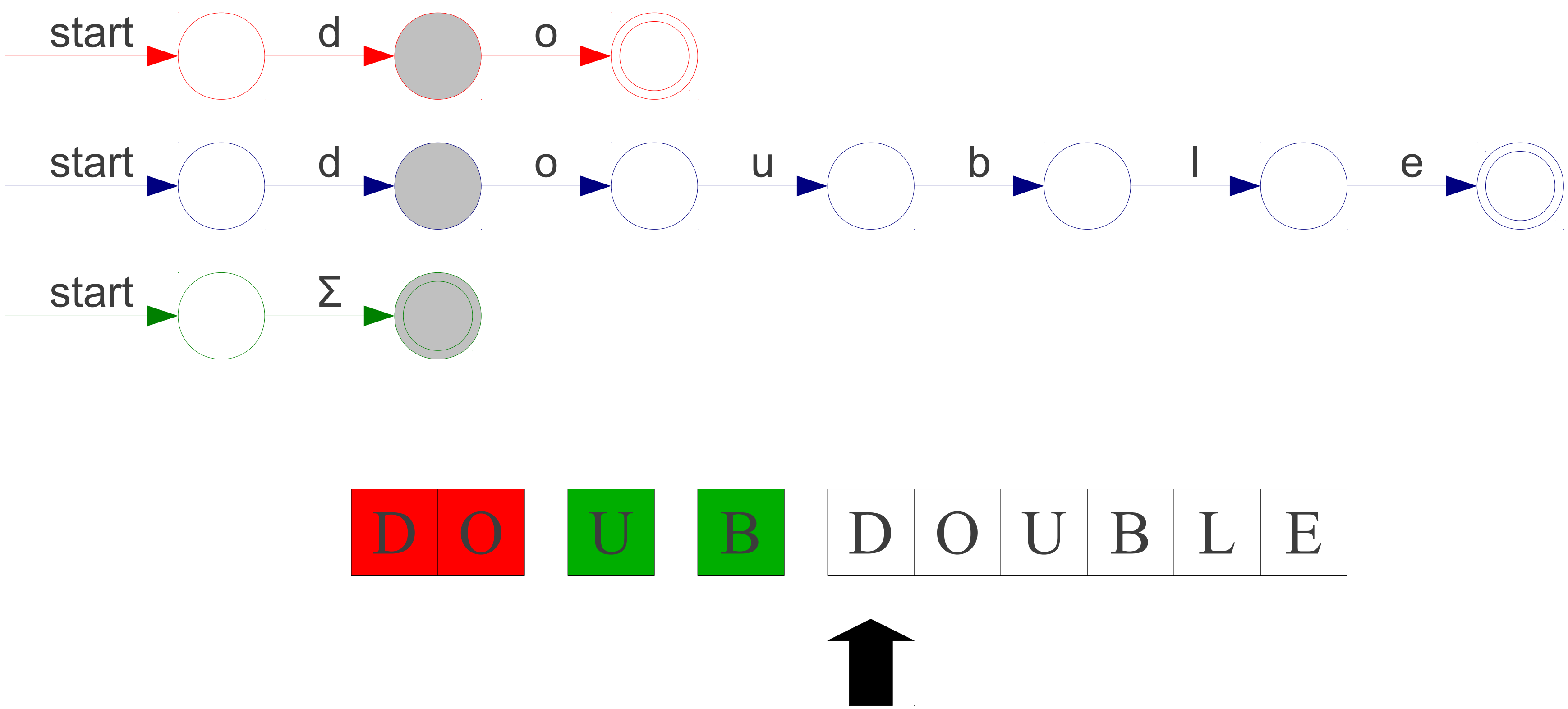
Implementing Maximal Munch

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



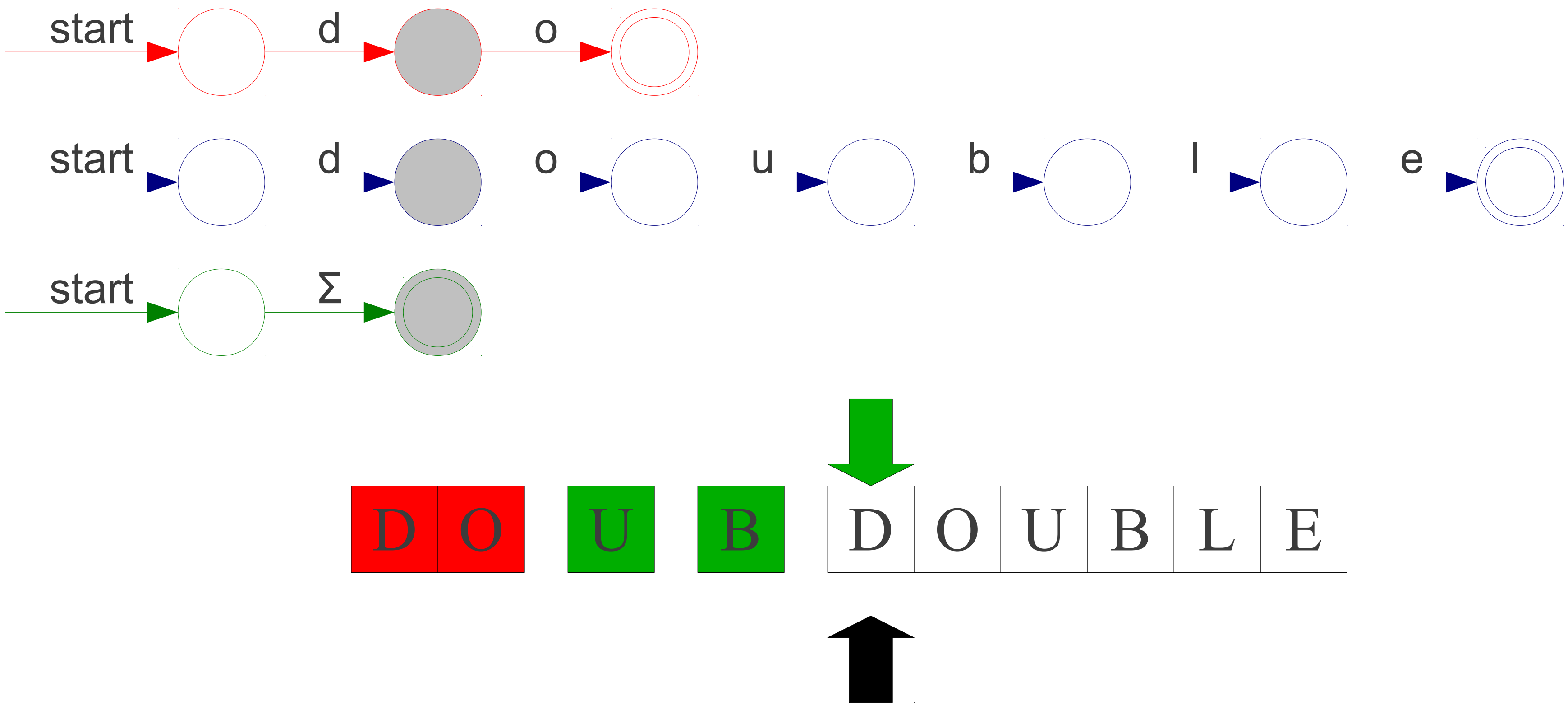
Implementing Maximal Munch

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



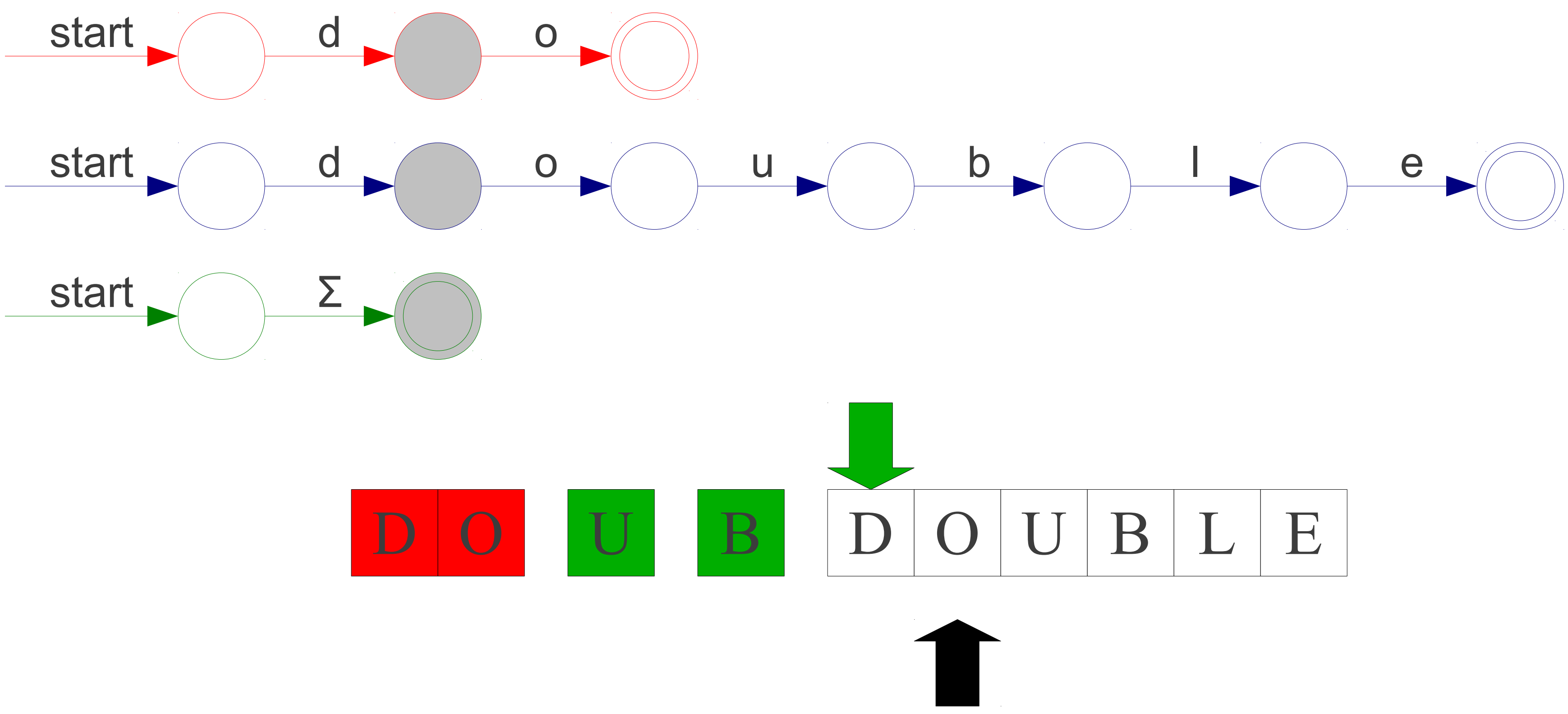
Implementing Maximal Munch

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



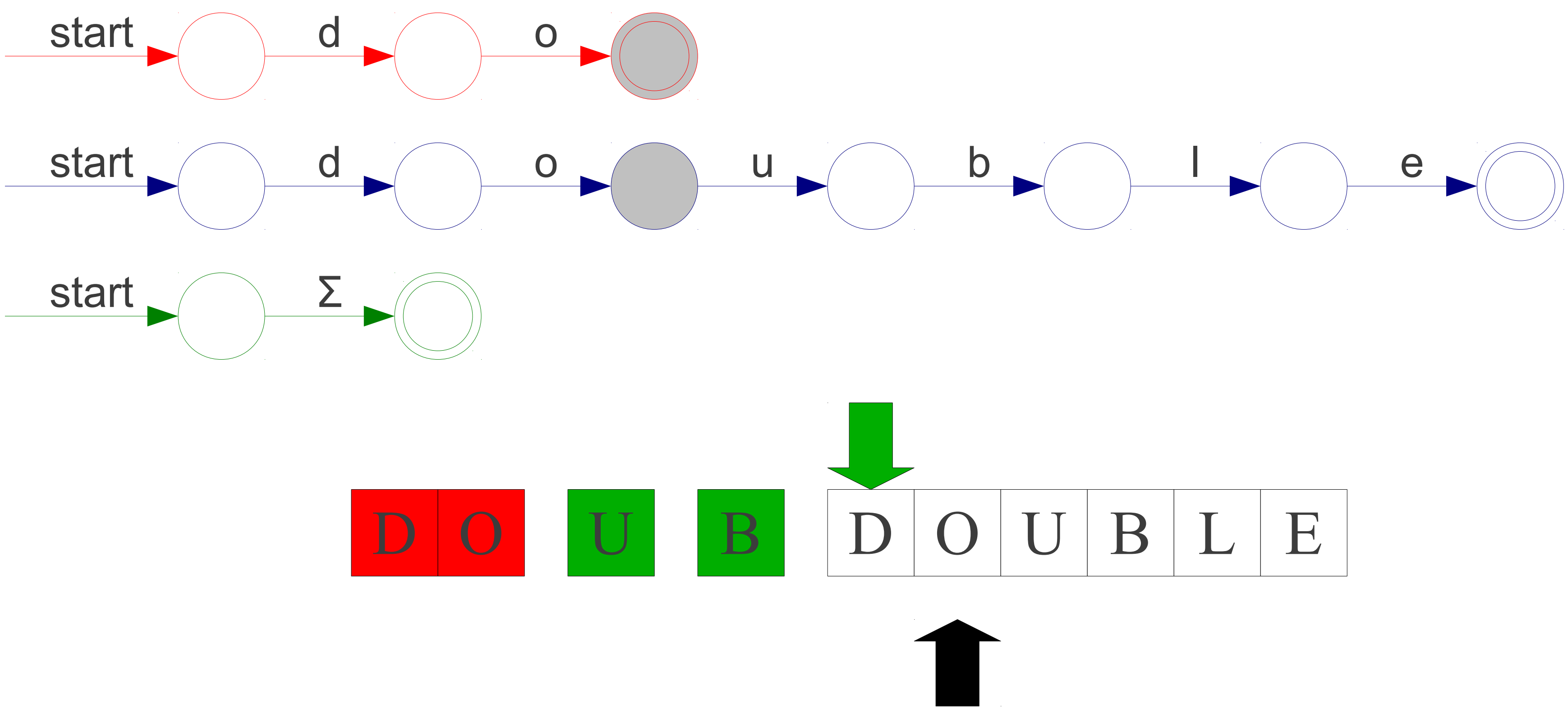
Implementing Maximal Munch

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



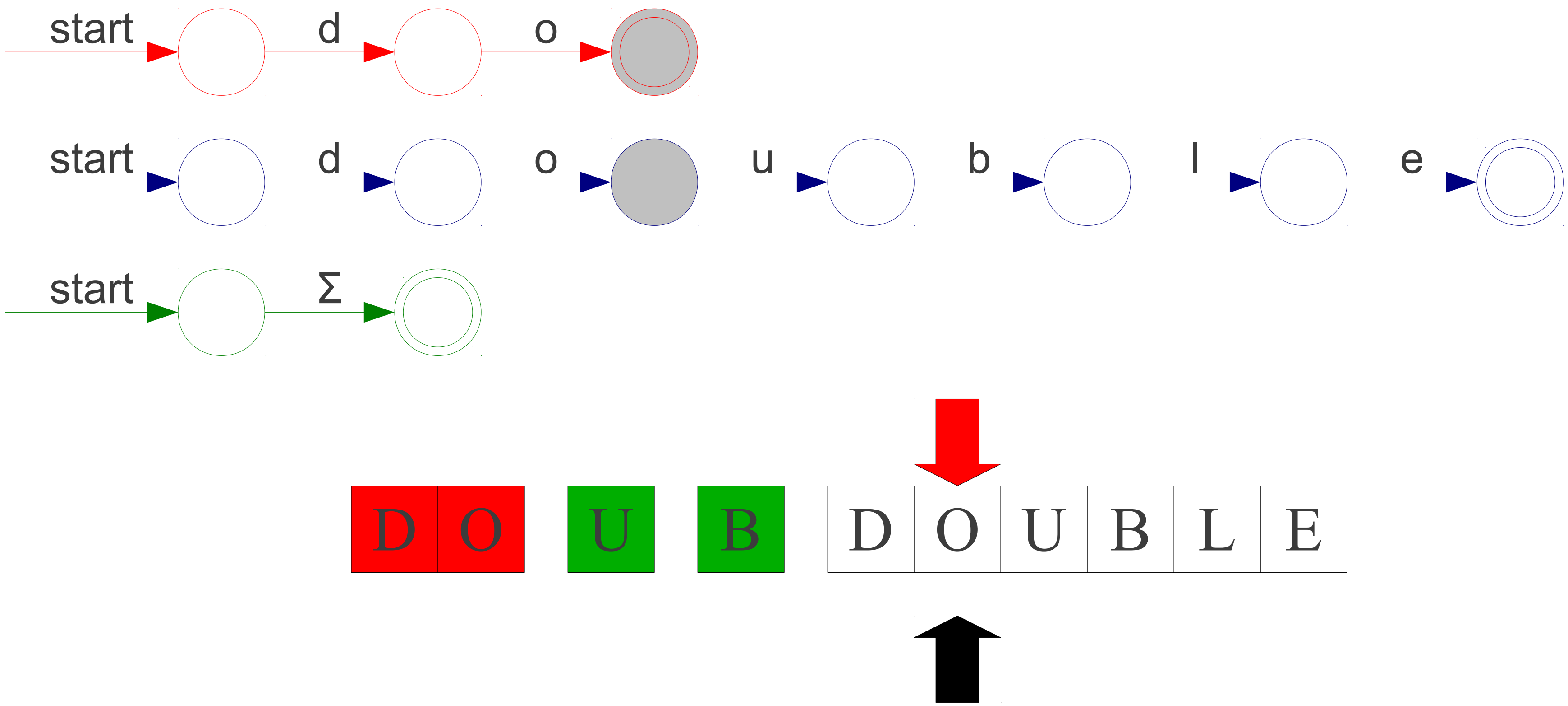
Implementing Maximal Munch

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



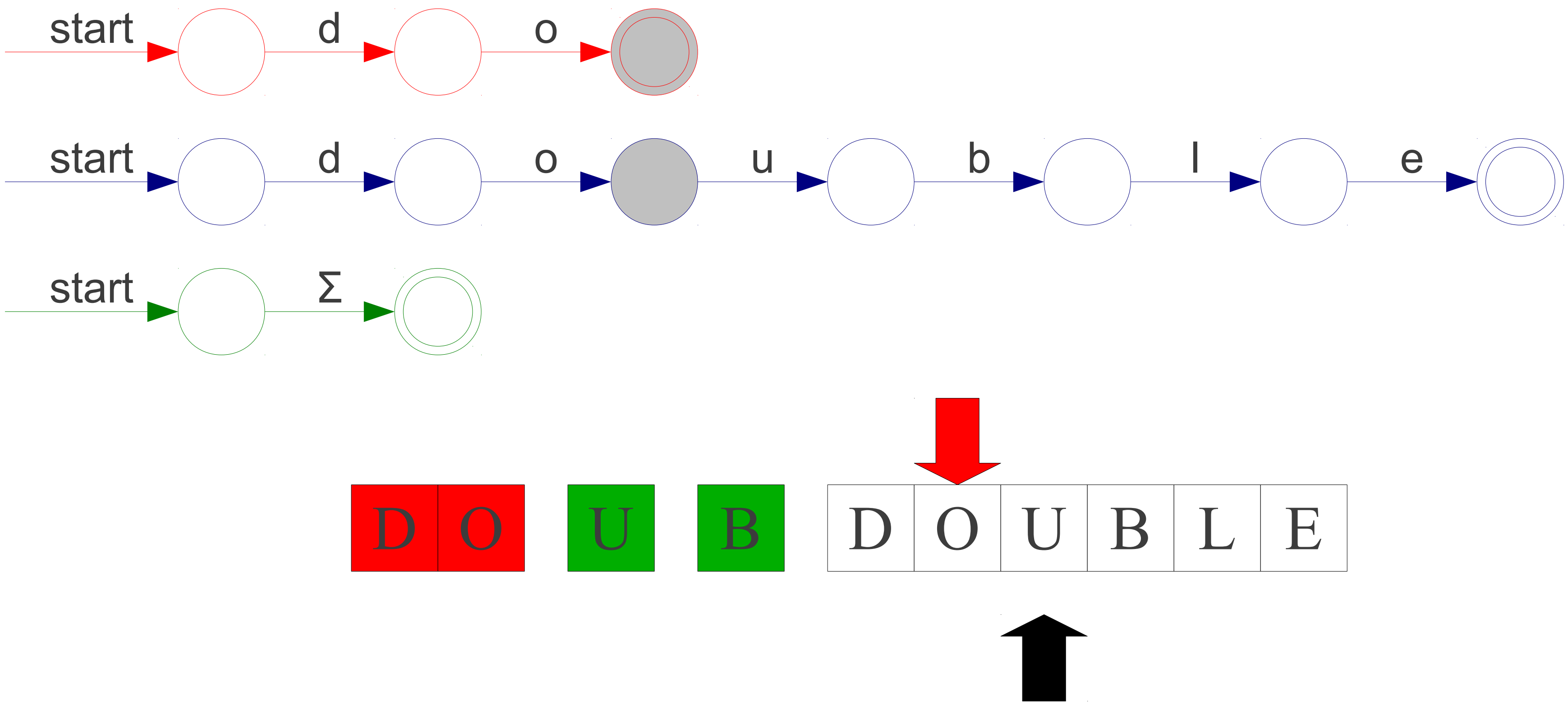
Implementing Maximal Munch

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



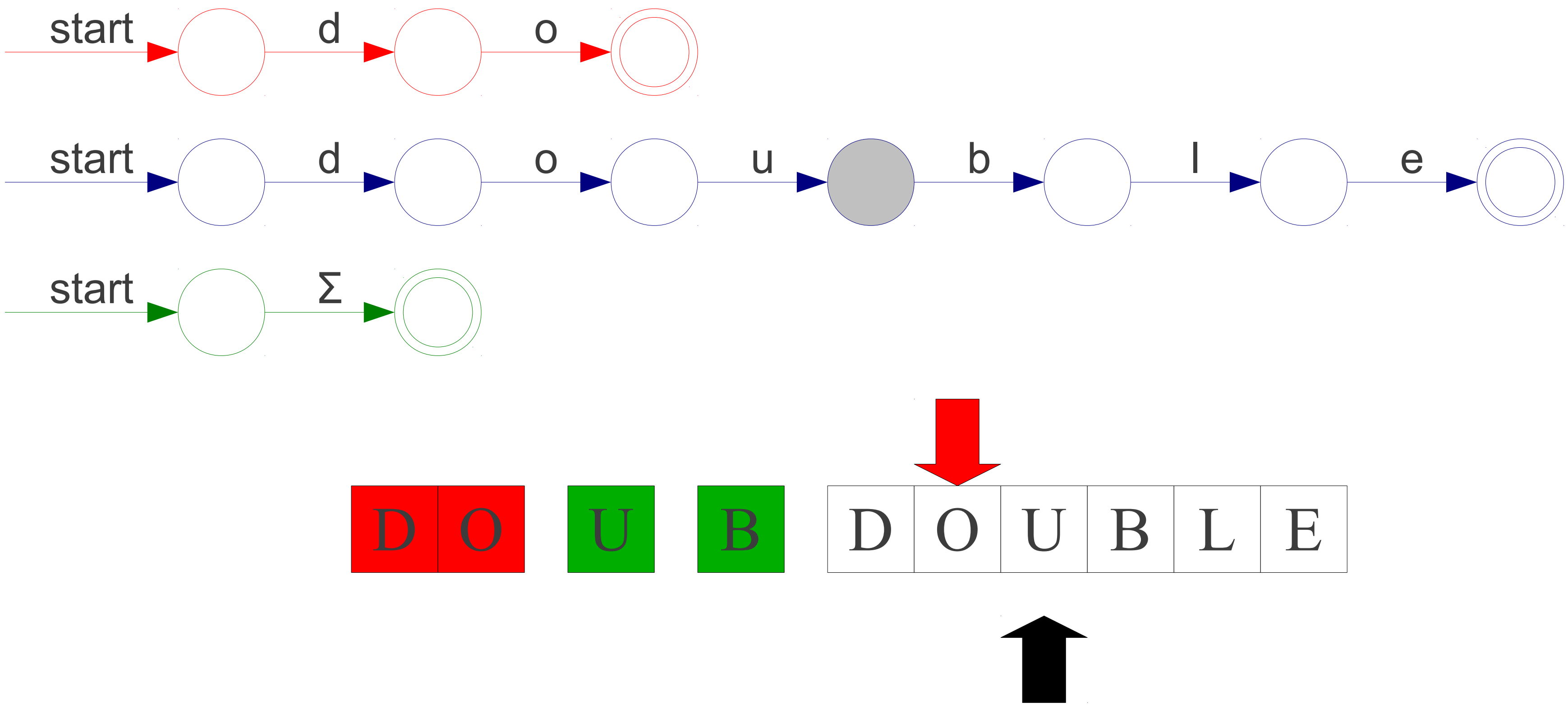
Implementing Maximal Munch

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



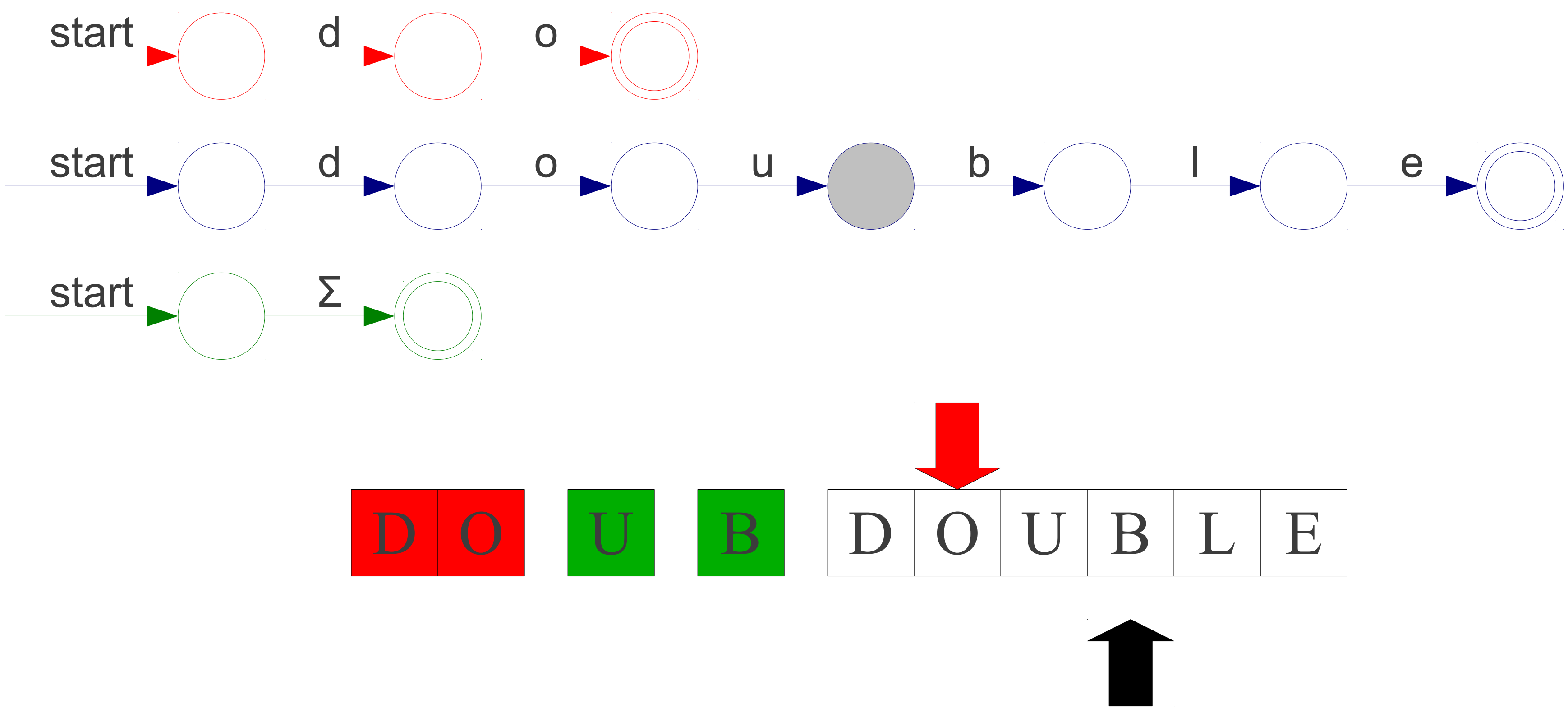
Implementing Maximal Munch

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



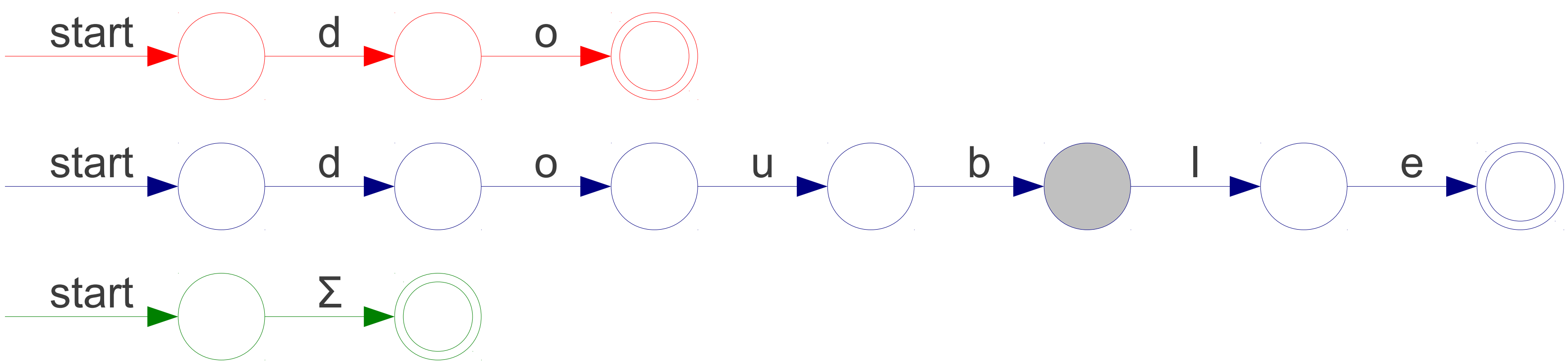
Implementing Maximal Munch

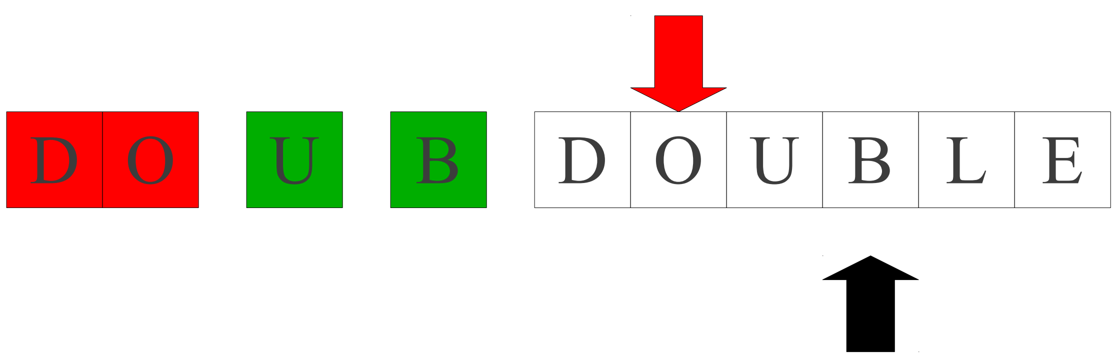
T_Do do
T_Double double
T_Mystery [A-Za-z]



Implementing Maximal Munch

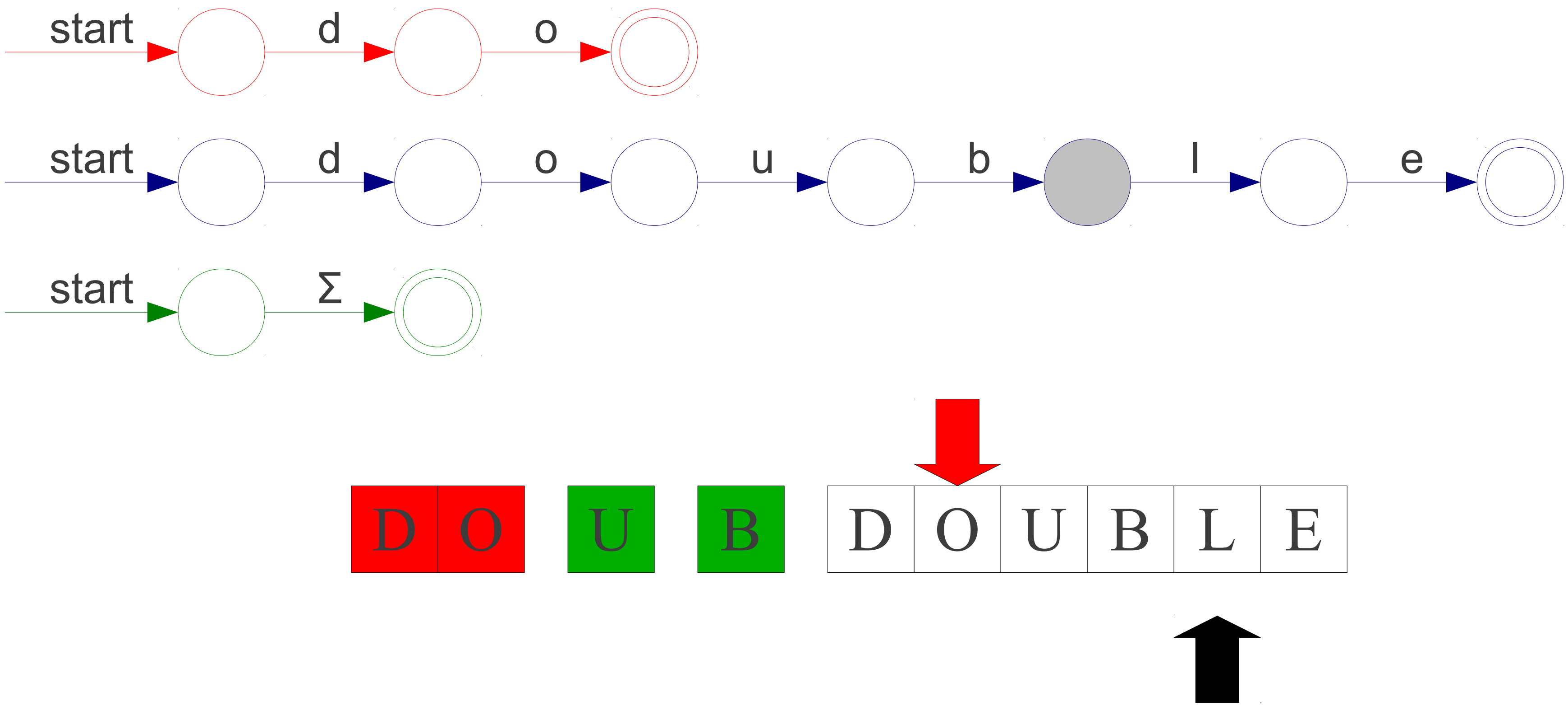
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```





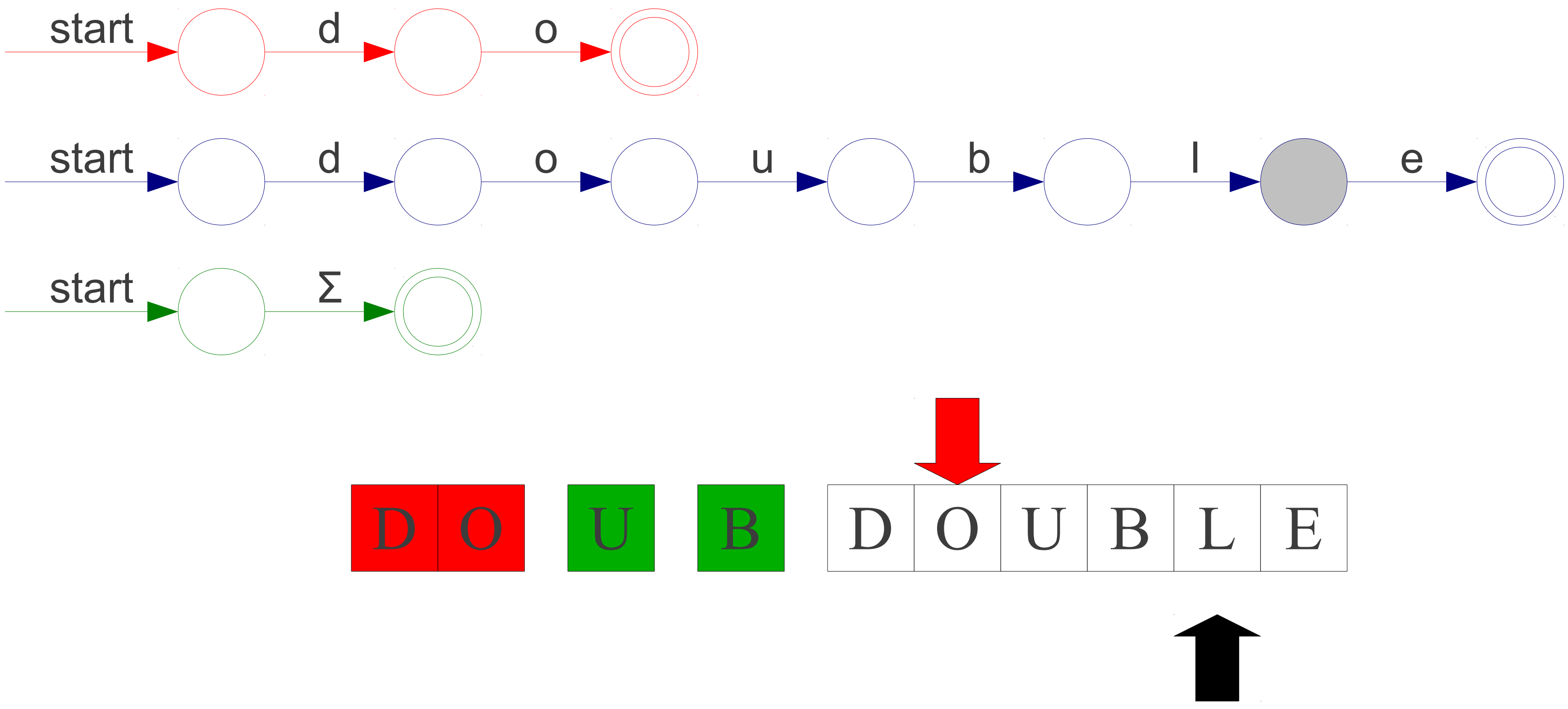
Implementing Maximal Munch

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



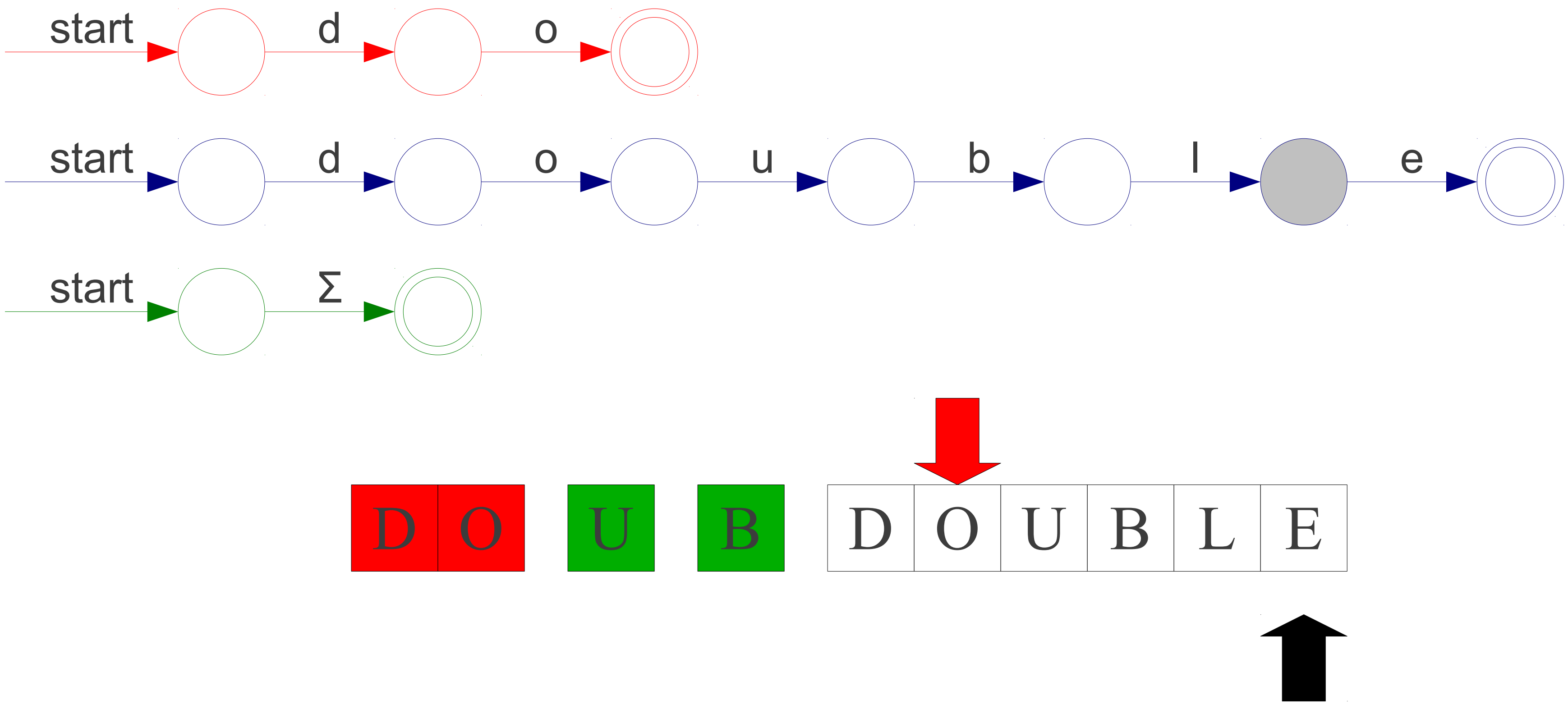
Implementing Maximal Munch

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



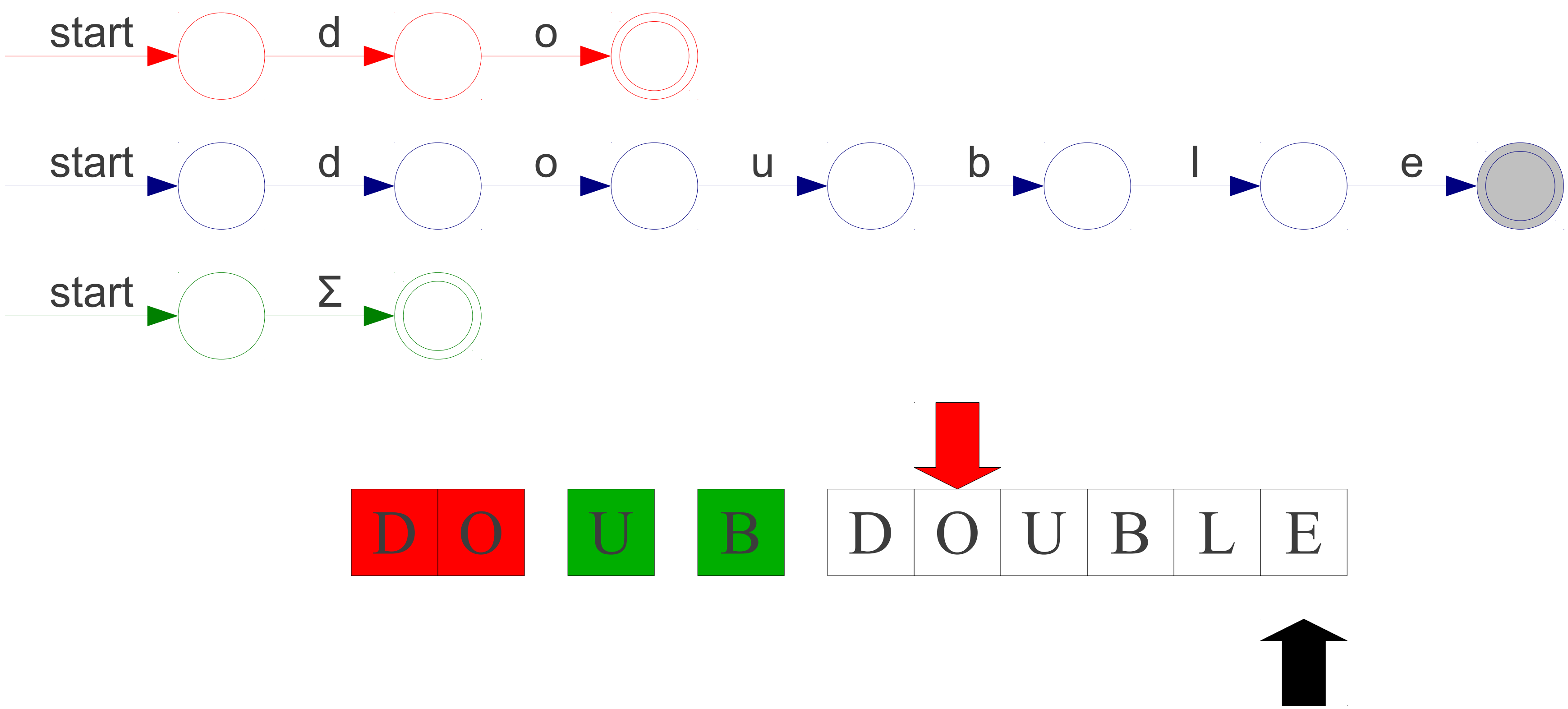
Implementing Maximal Munch

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



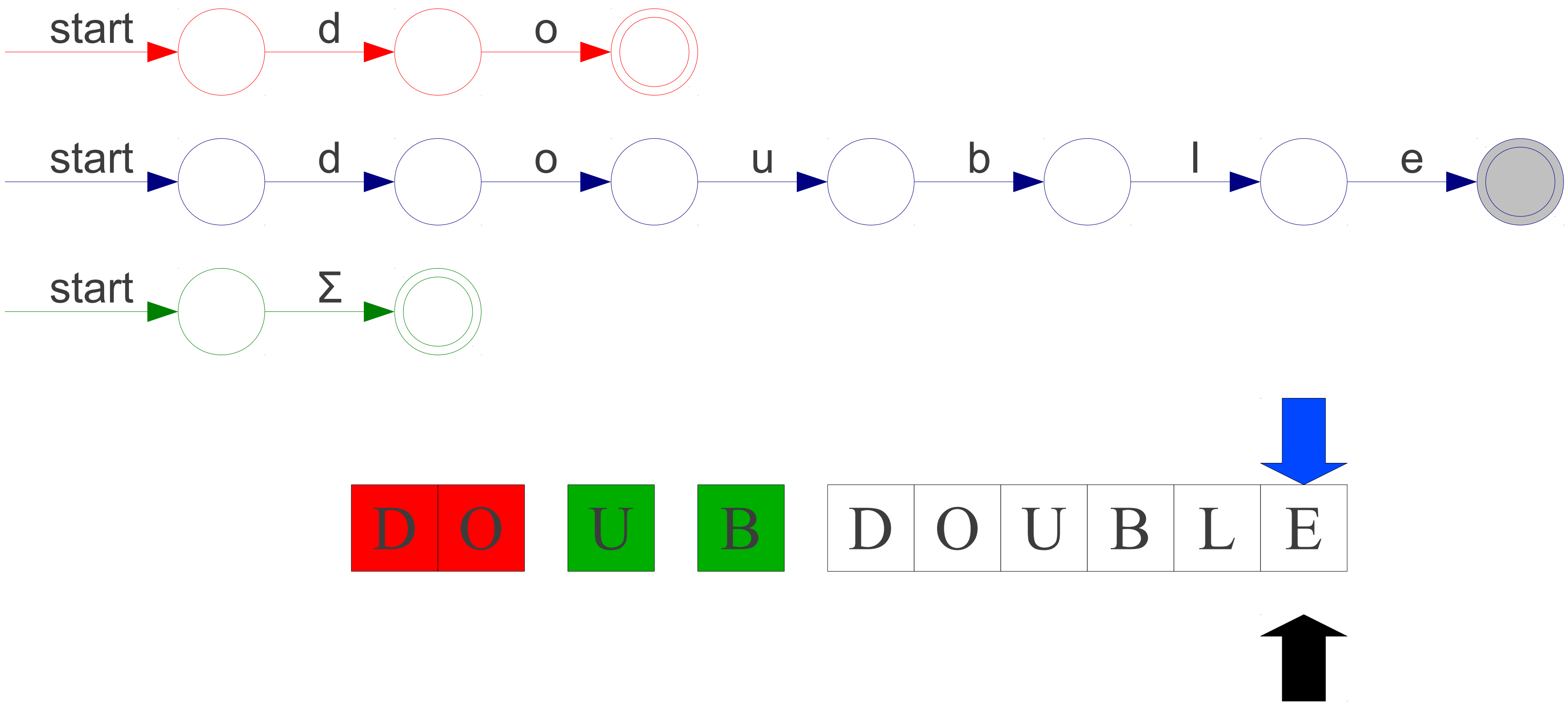
Implementing Maximal Munch

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



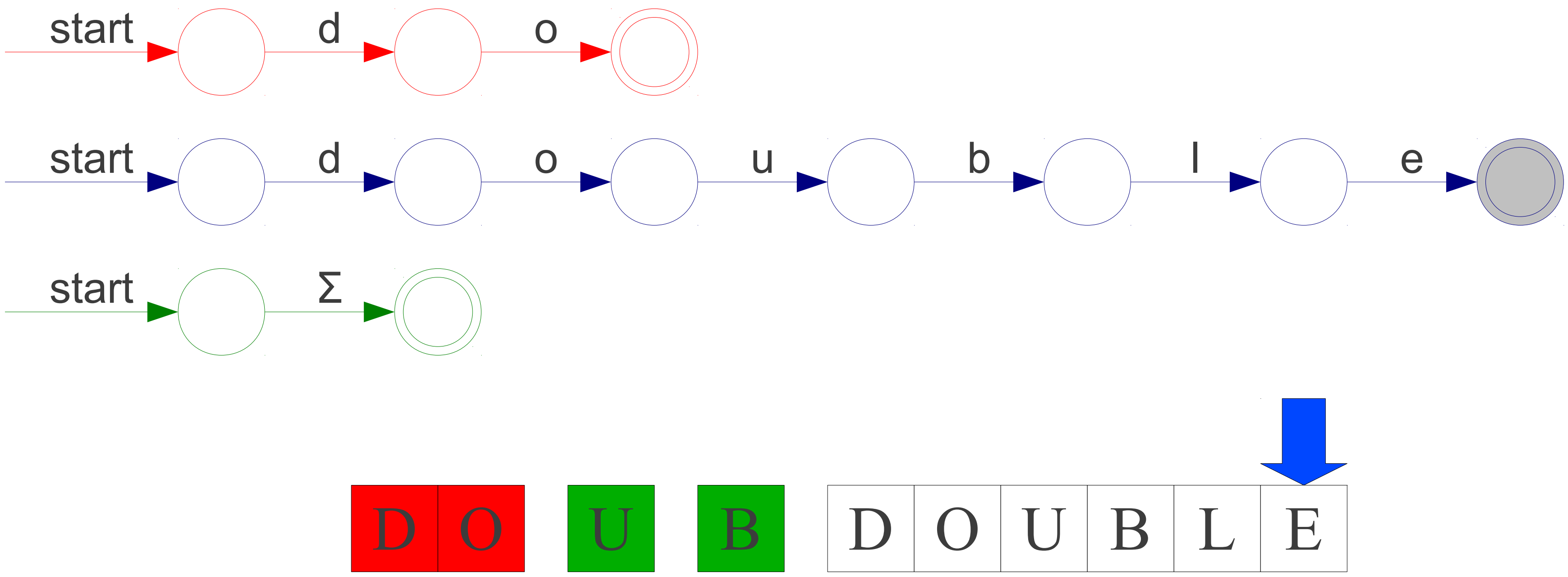
Implementing Maximal Munch

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



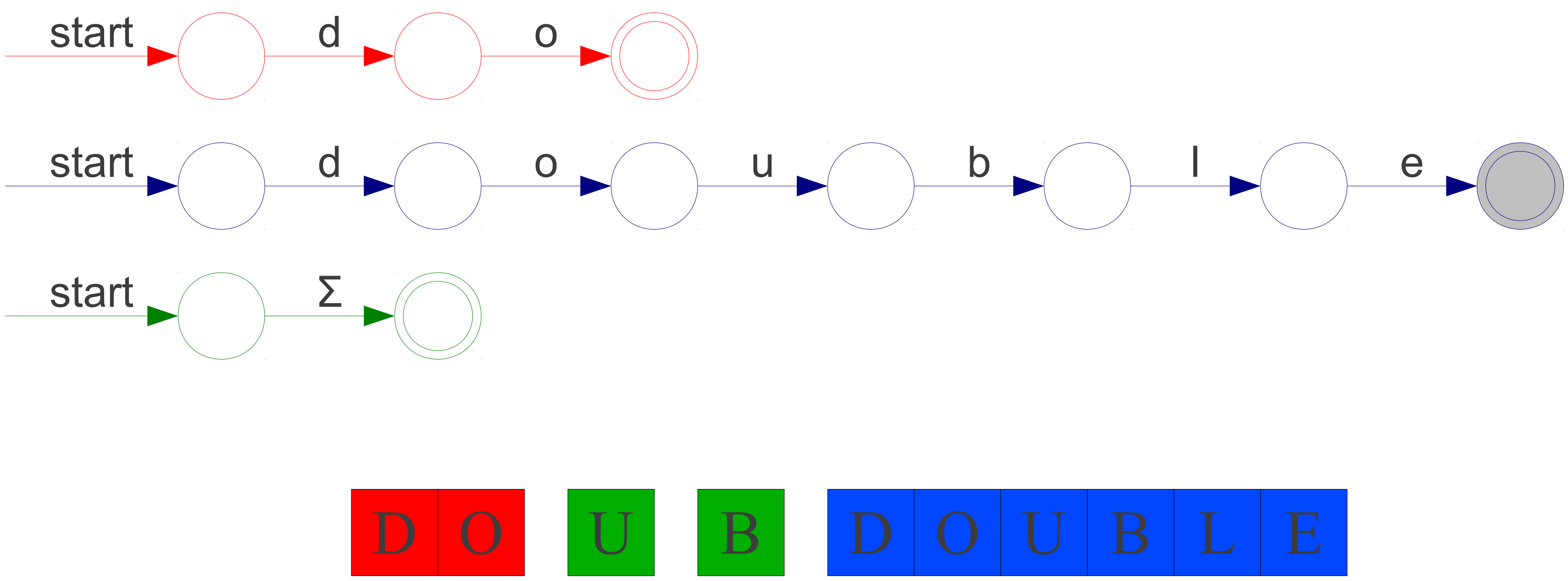
Implementing Maximal Munch

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



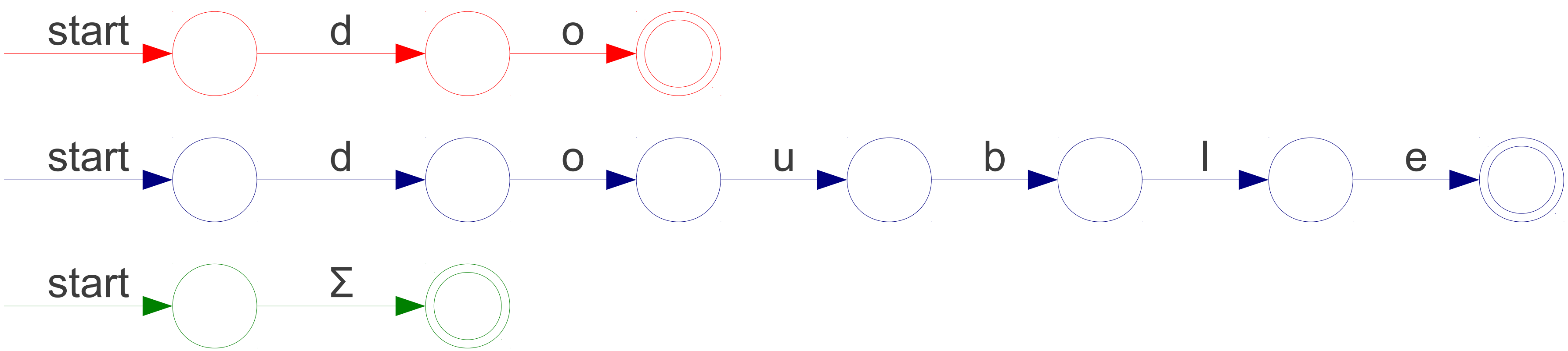
Implementing Maximal Munch

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

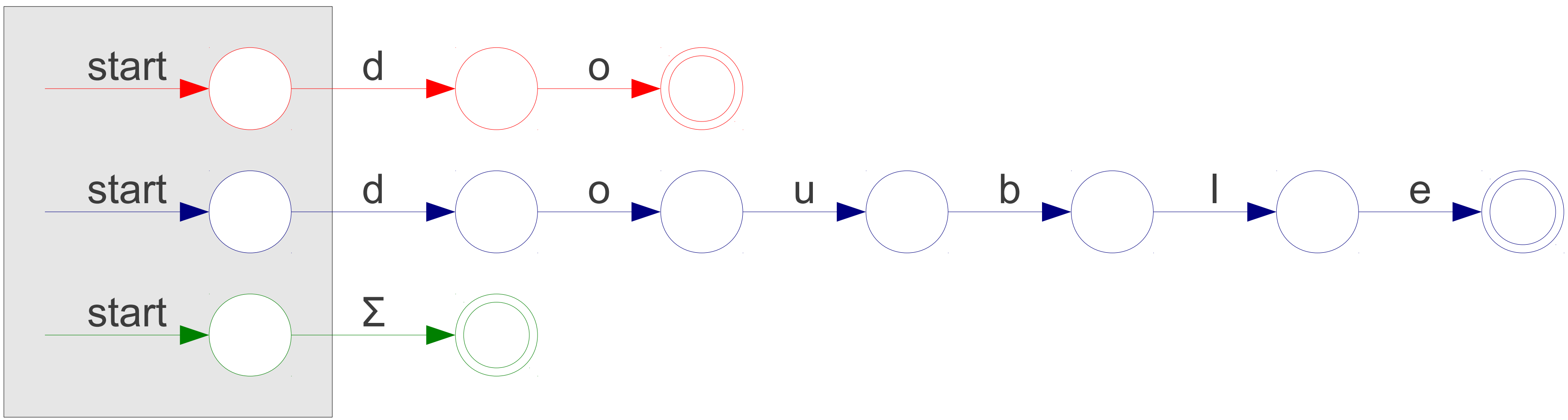


A Minor Simplification

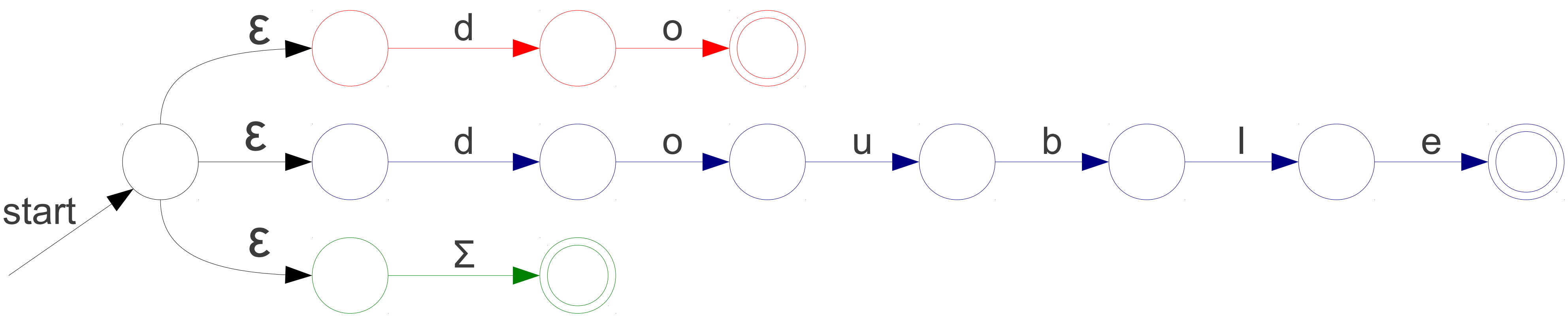
A Minor Simplification



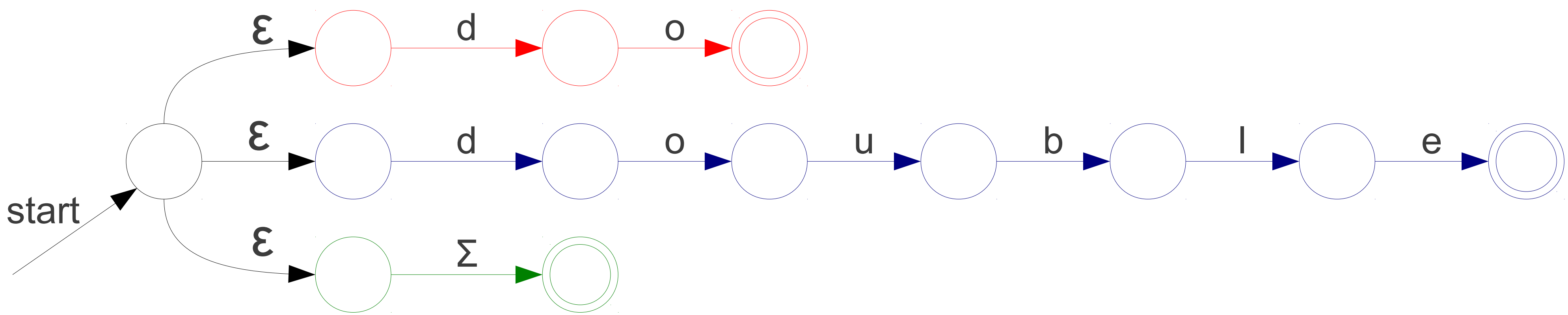
A Minor Simplification



A Minor Simplification

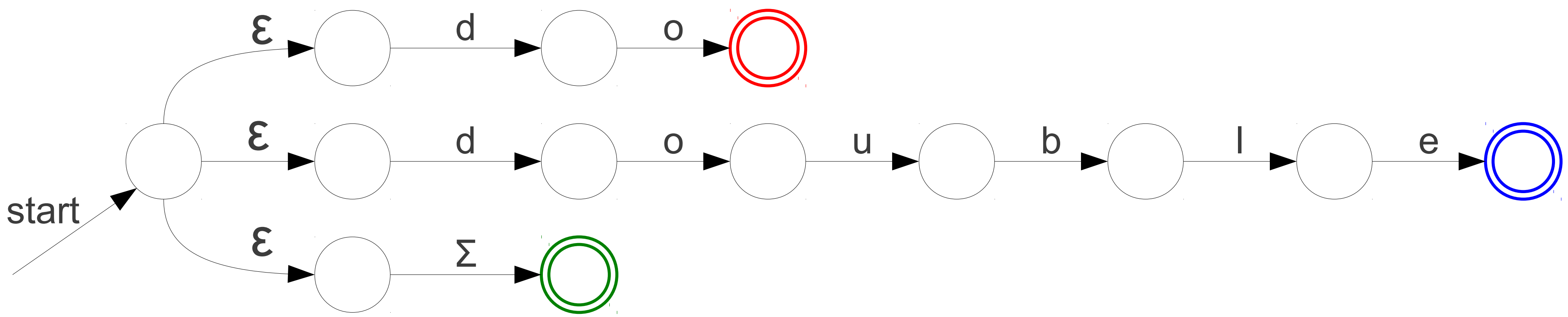


A Minor Simplification

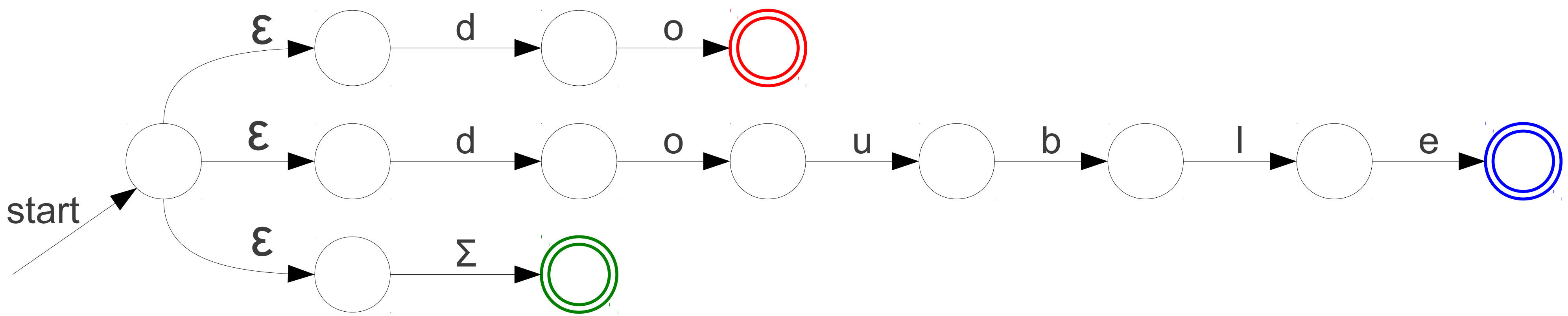


Build a single automaton that runs all the matching automata in parallel.

A Minor Simplification



A Minor Simplification



Annotate each accepting state with which automaton it came from.

Other Conflicts

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

Other Conflicts

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

d	o	u	b	l	e

Other Conflicts

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

d	o	u	b	l	e

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

More Tiebreaking

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

Other Conflicts

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

d	o	u	b	l	e

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

Other Conflicts

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

d	o	u	b	l	e

d	o	u	b	l	e

Other Conflicts

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

d	o	u	b	l	e

d	o	u	b	l	e

Why isn't this a problem?

One Last Detail...

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

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

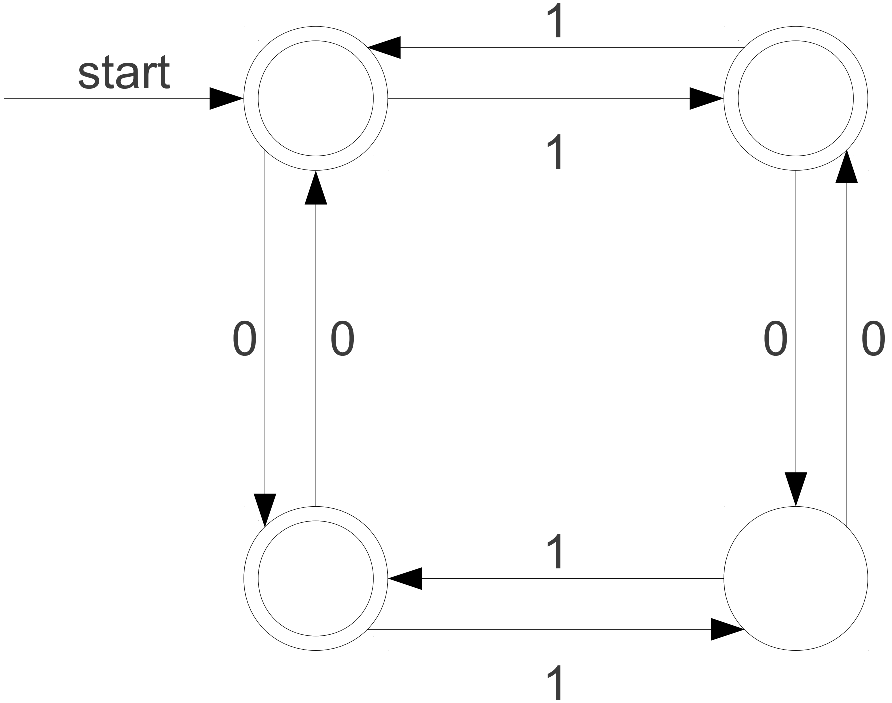
- How do we determine which lexemes are associated with each token?
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

DFAs

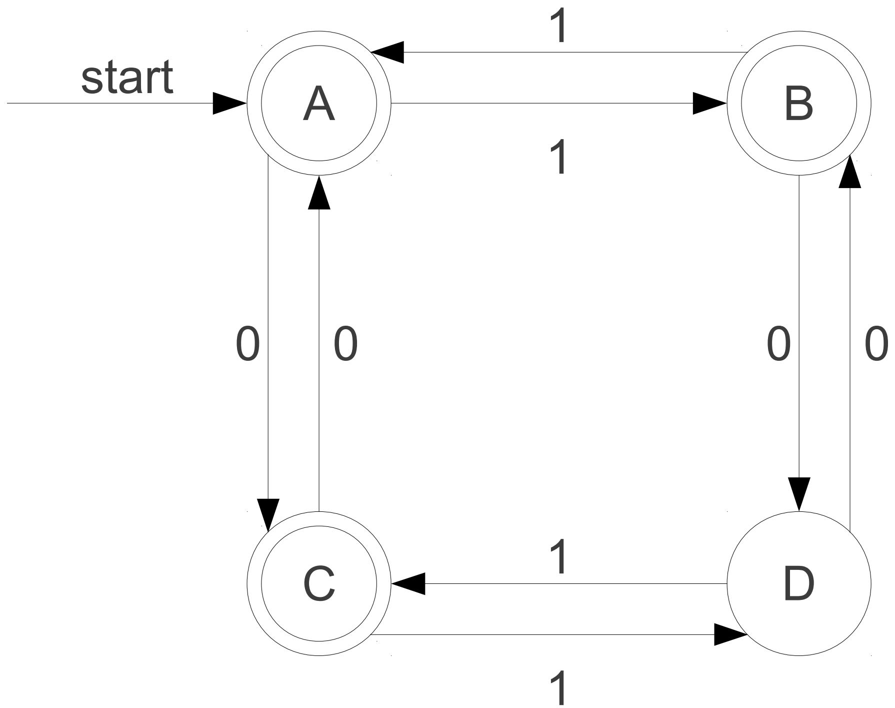
- The automata we've seen so far have all been NFAs.
- A **DFA** is like an NFA, but with tighter restrictions:
  - Every state must have **exactly one** transition defined for every letter.
  - ε-moves are not allowed.

A Sample DFA

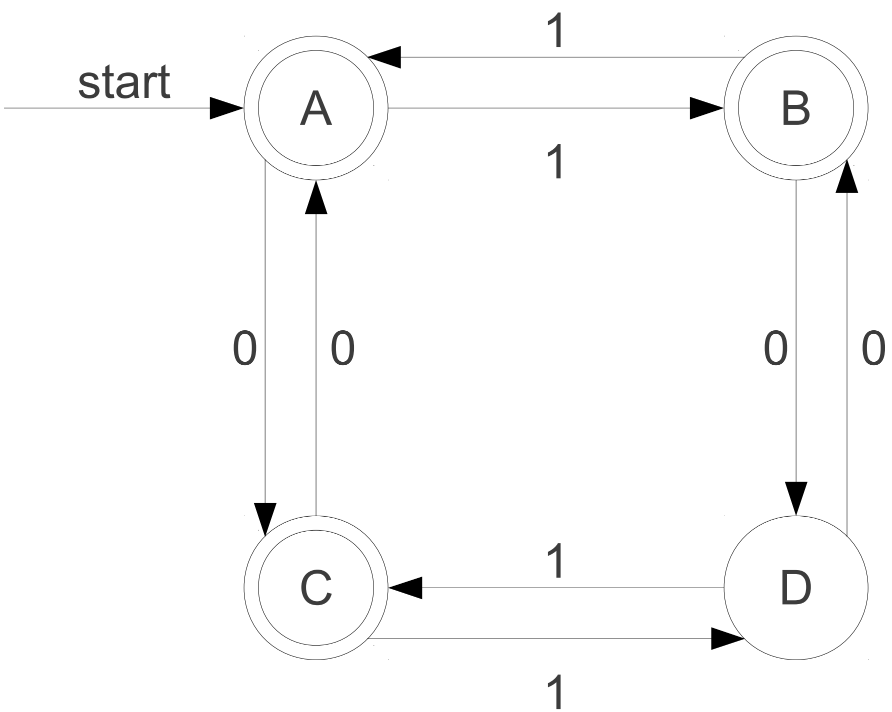
A Sample DFA



A Sample DFA



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];
}
```

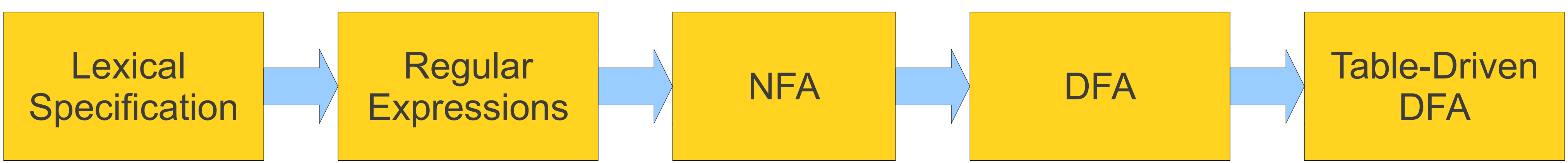
Code for DFAs

```
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.
- DFAs, on the other hand, take only $O(m)$.
- There is another (beautiful!) algorithm to convert NFAs to DFAs.

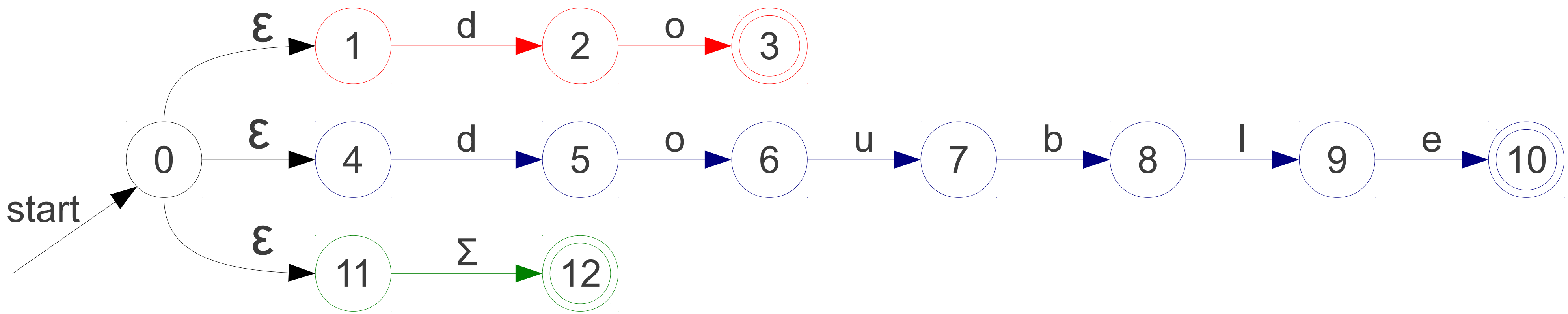


Subset Construction

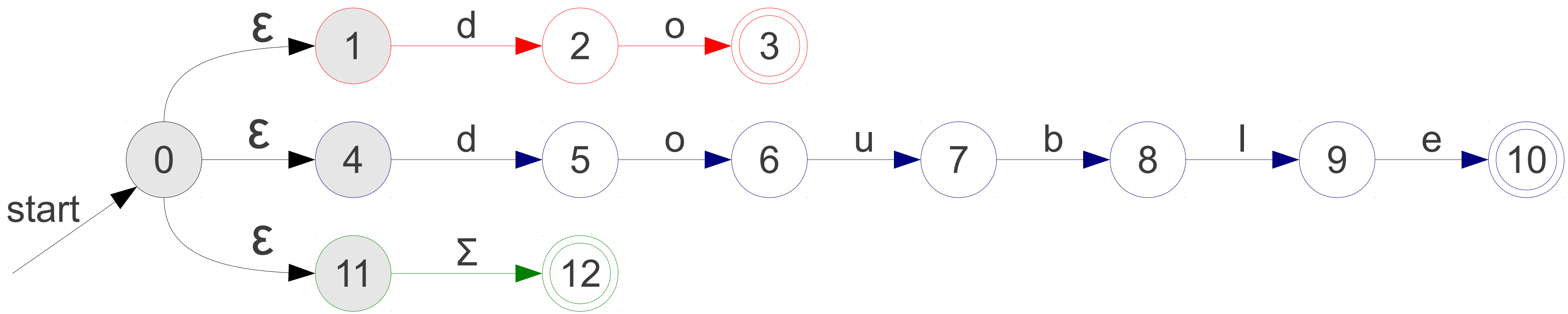
- NFAs can be in many states at once, while DFAs can only be in a single state at a time.
- Key idea: **Make the DFA simulate the NFA**.
- Have the states of the DFA correspond to the *sets of states* of the NFA.
- Transitions between states of DFA correspond to transitions between *sets of states* in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

From NFA to DFA

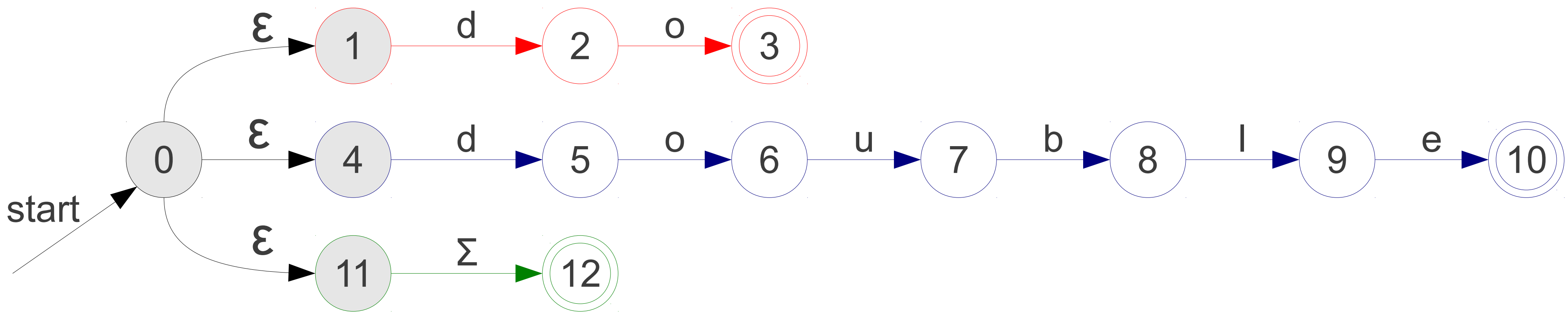
From NFA to DFA



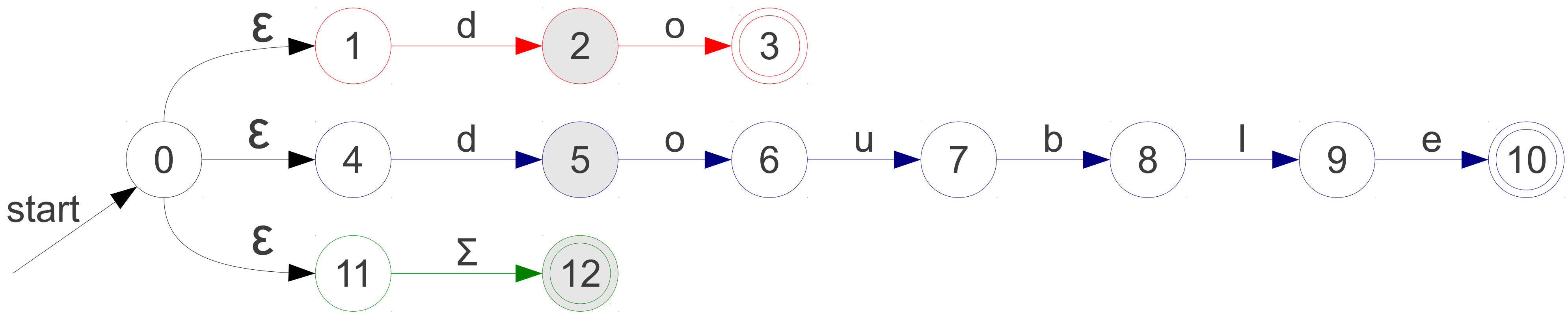
From NFA to DFA



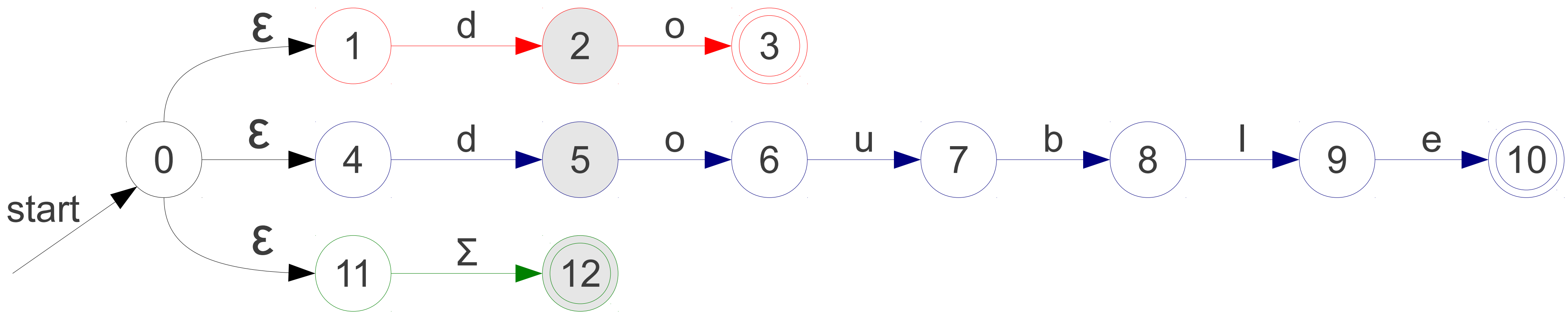
From NFA to DFA

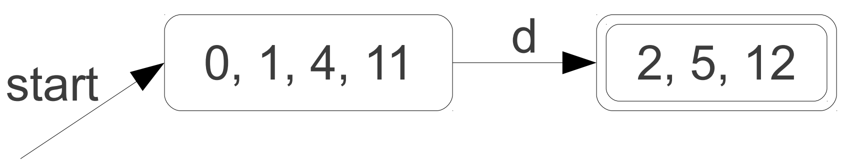


From NFA to DFA

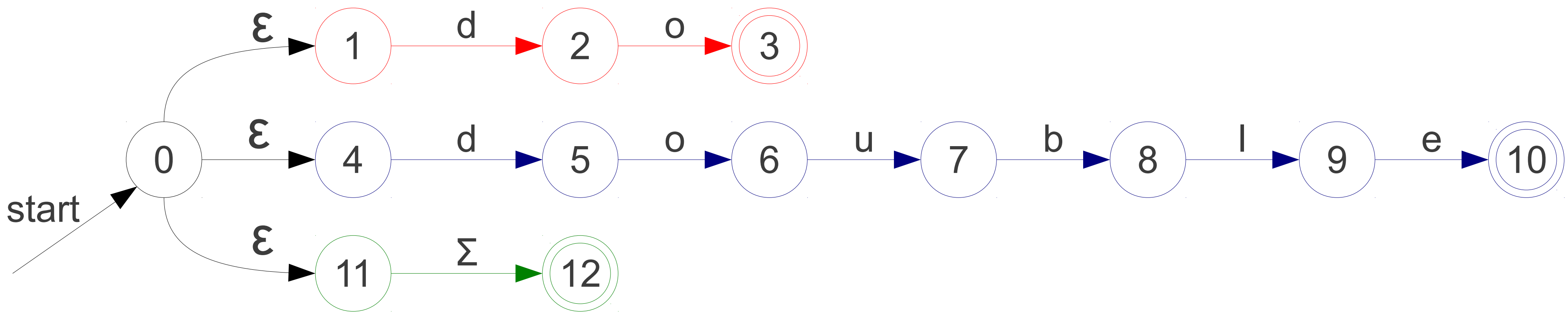


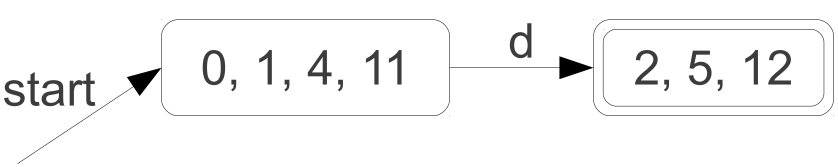
From NFA to DFA



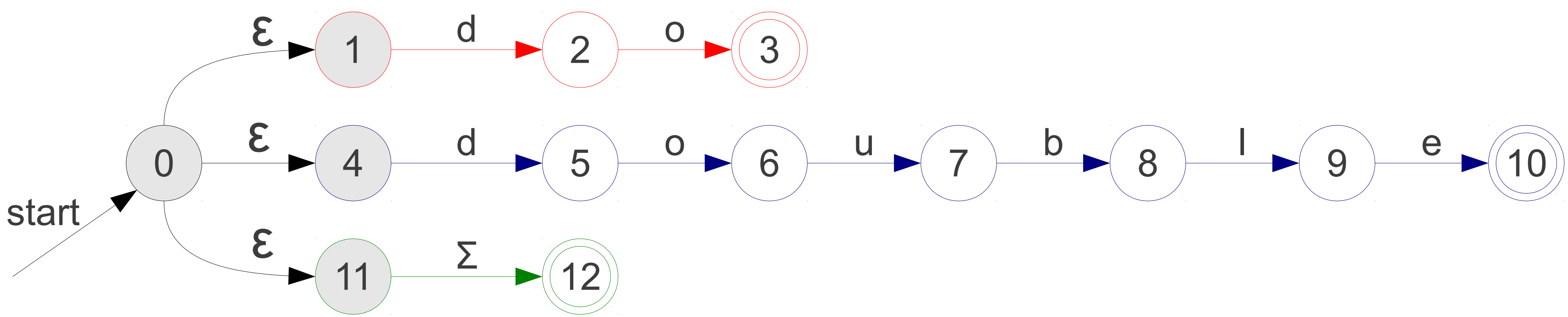


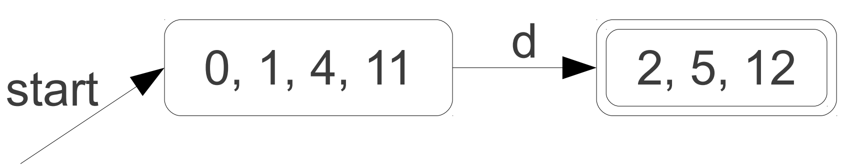
From NFA to DFA



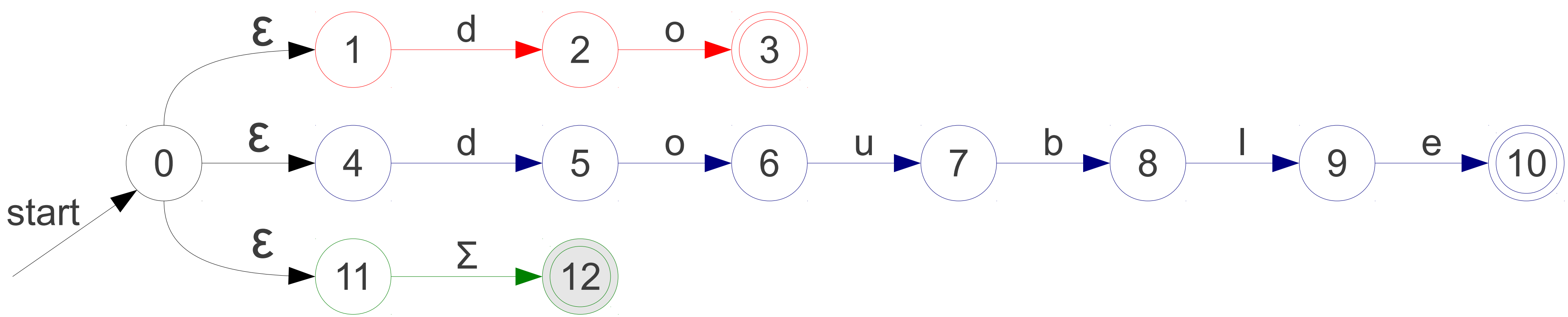


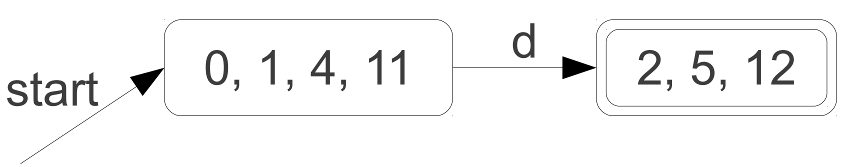
From NFA to DFA



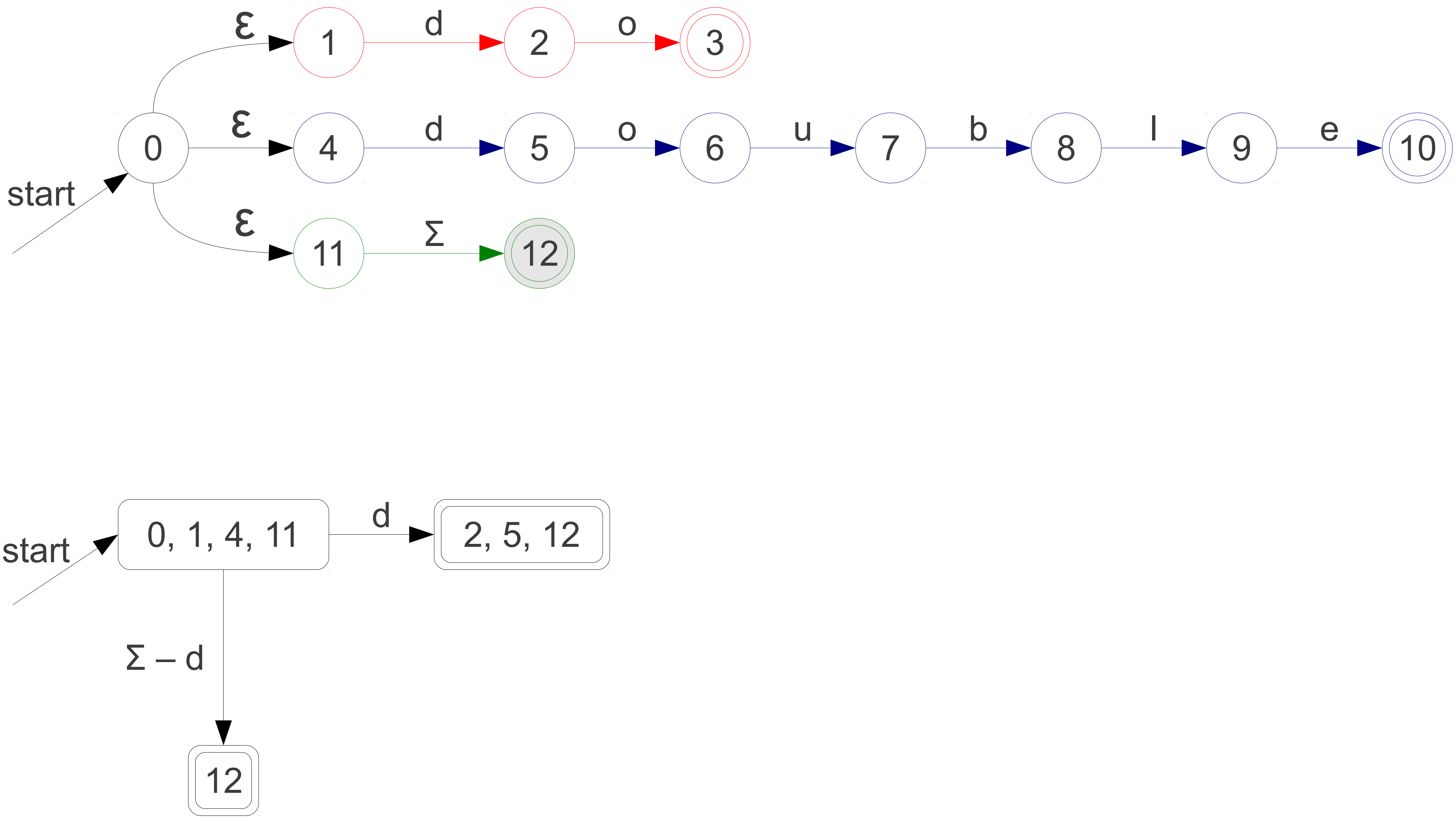


From NFA to DFA

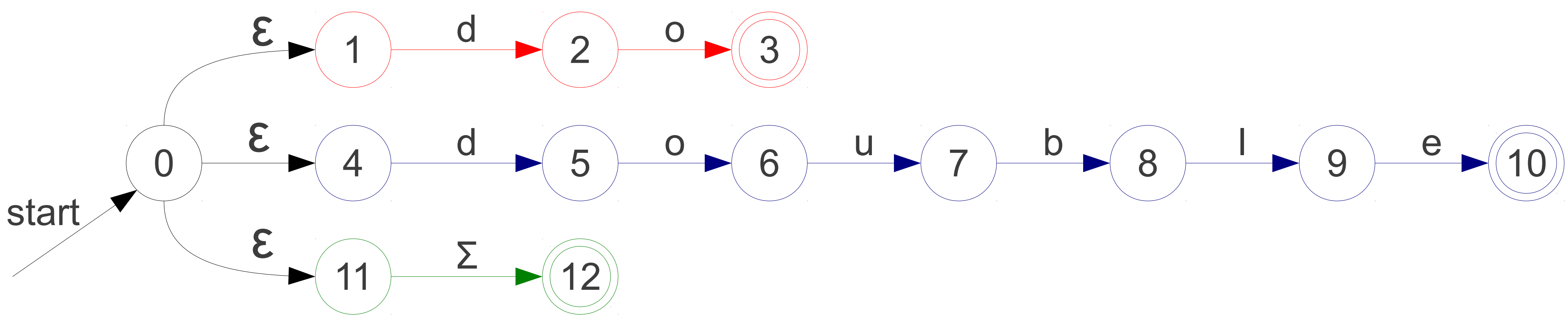


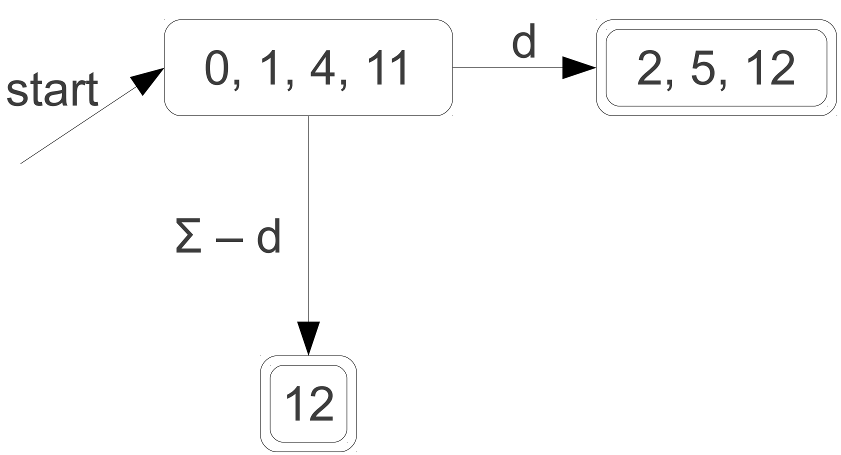


From NFA to DFA

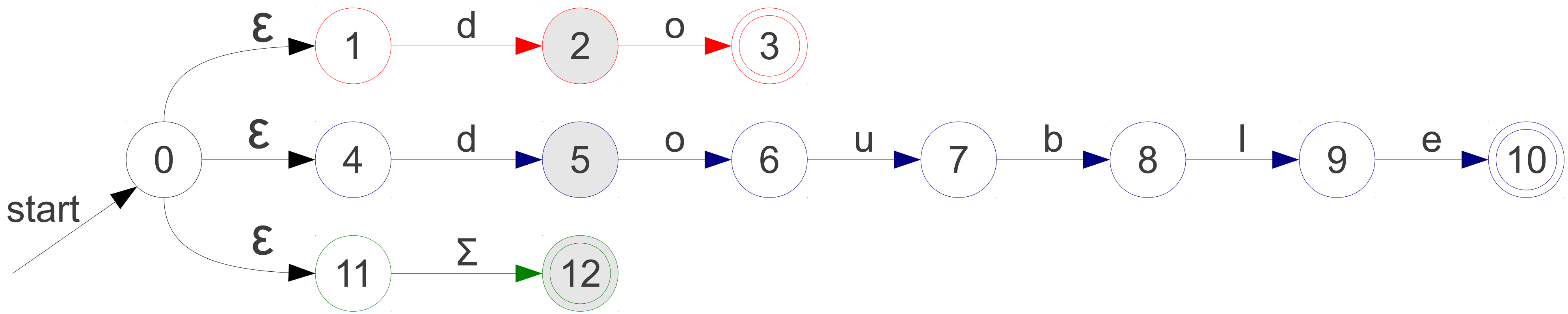


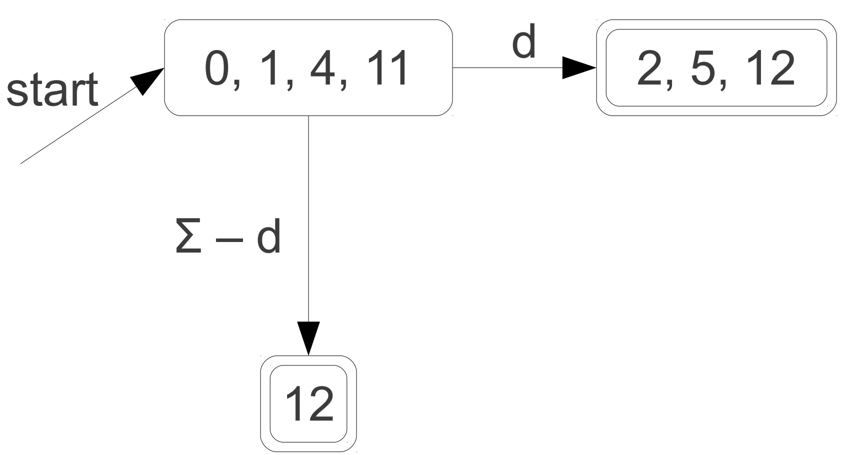
From NFA to DFA



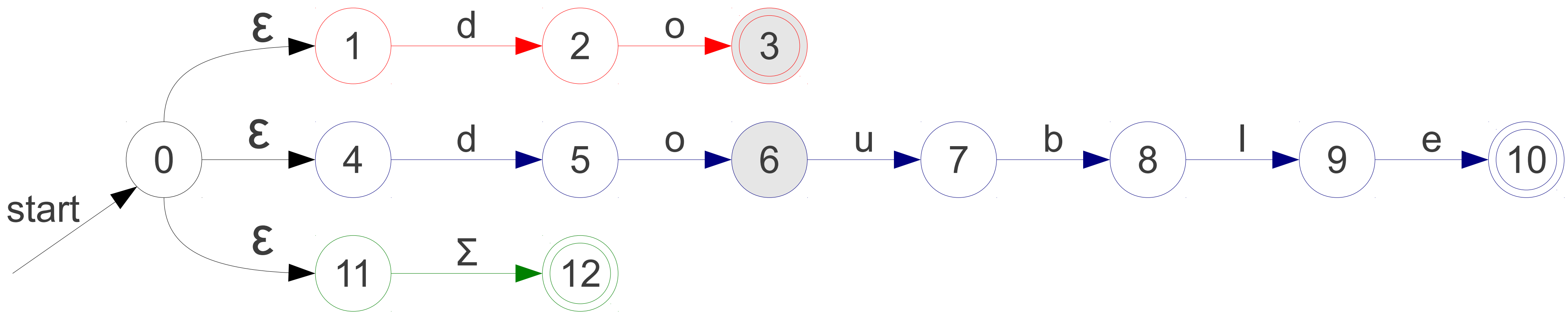


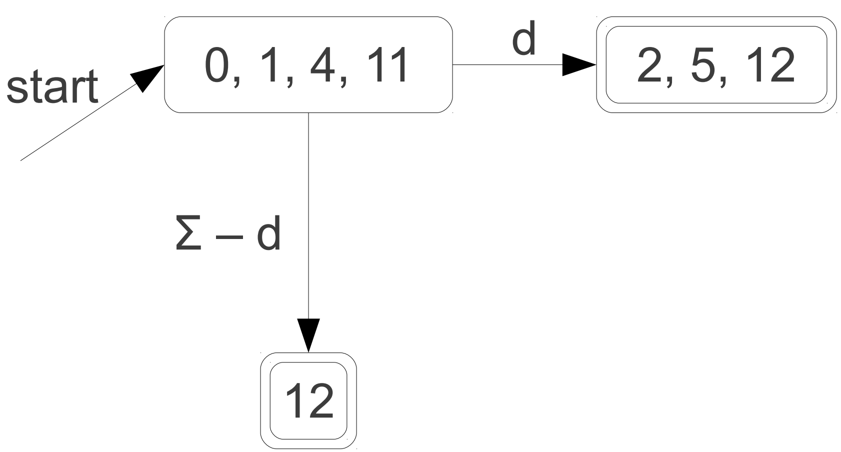
From NFA to DFA



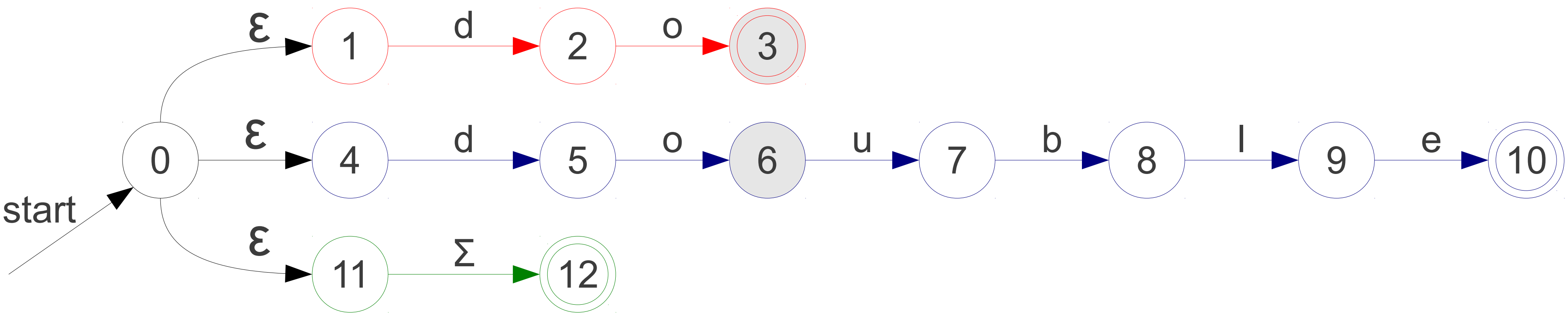


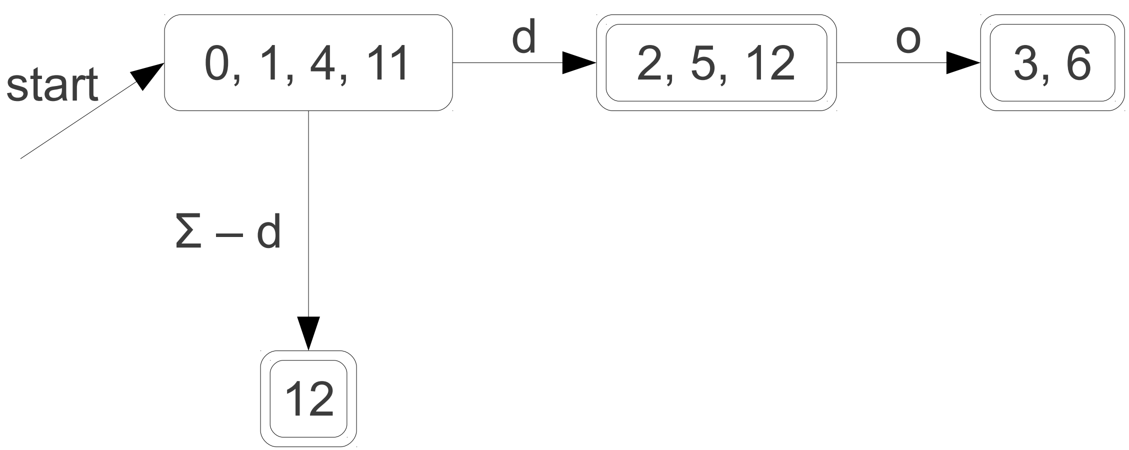
From NFA to DFA



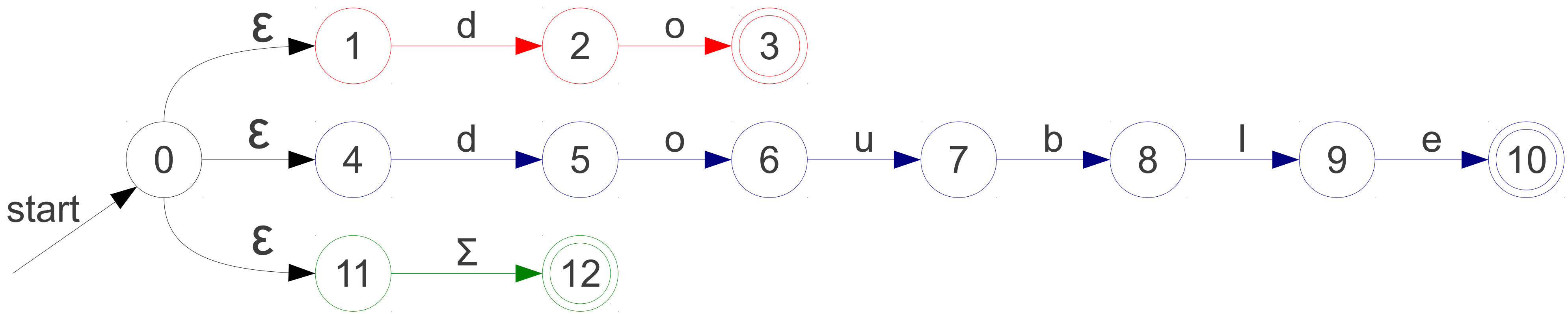


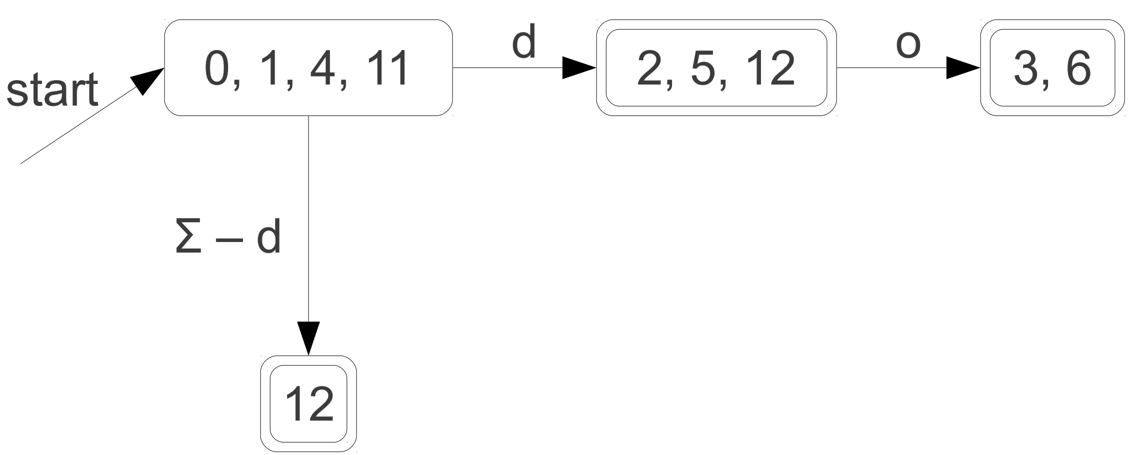
From NFA to DFA



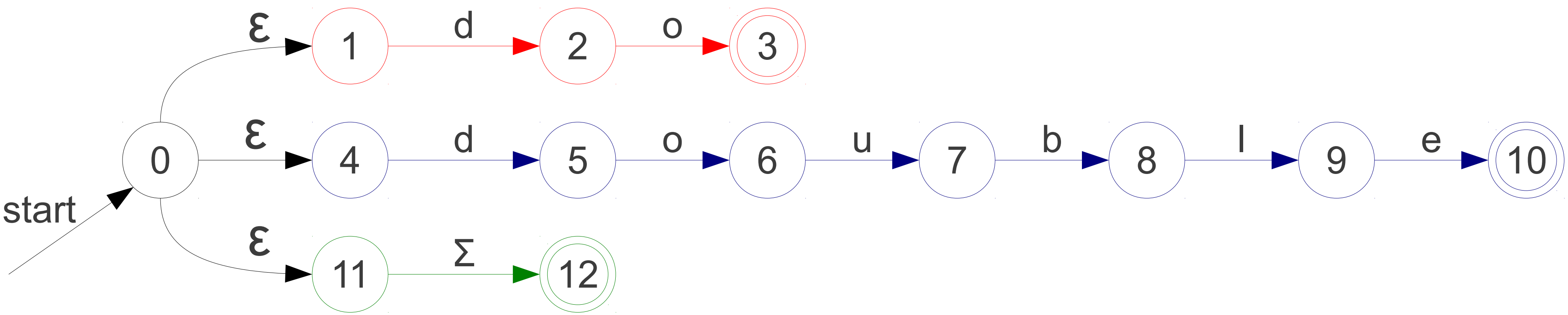


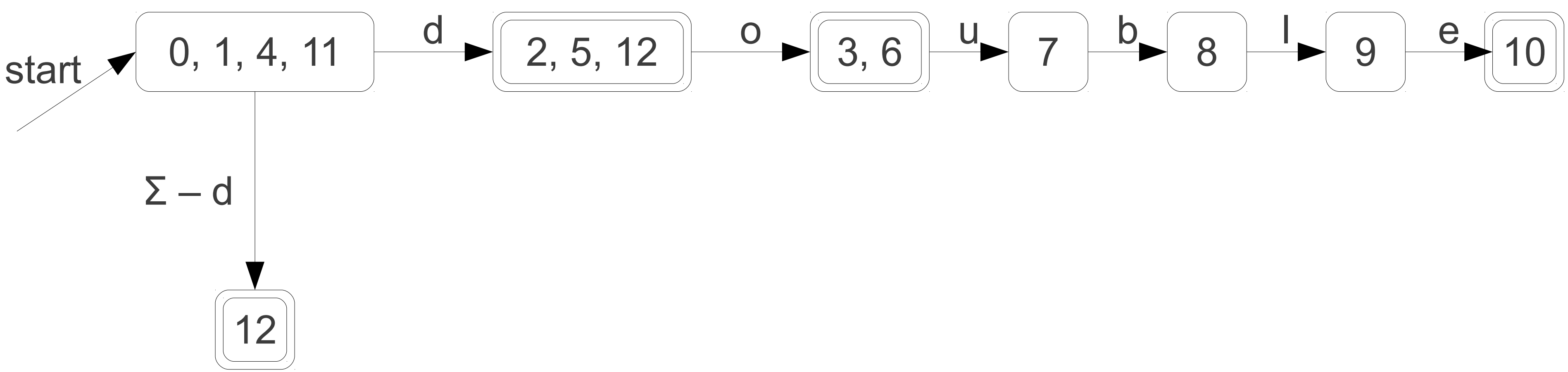
From NFA to DFA



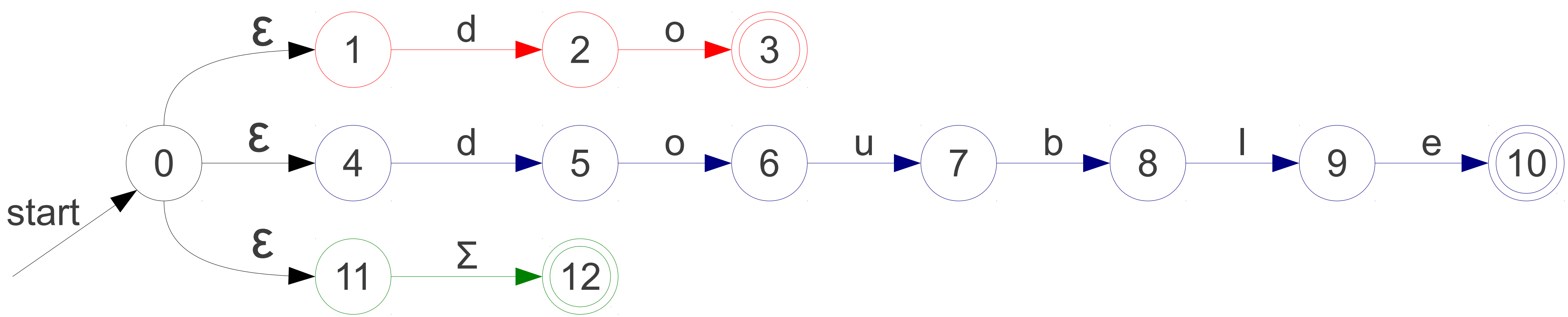


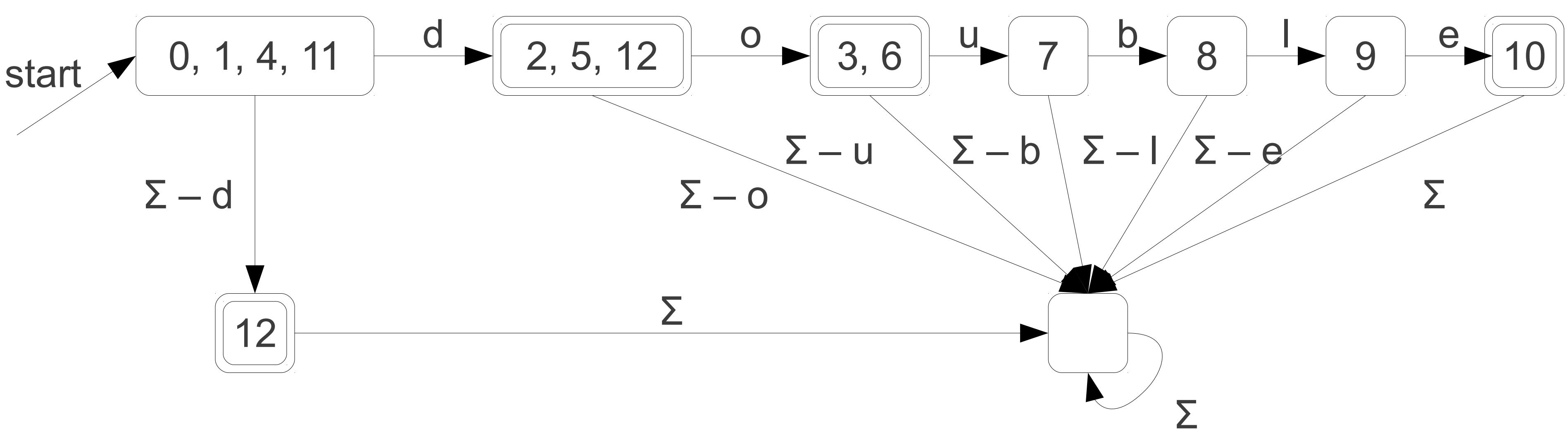
From NFA to DFA



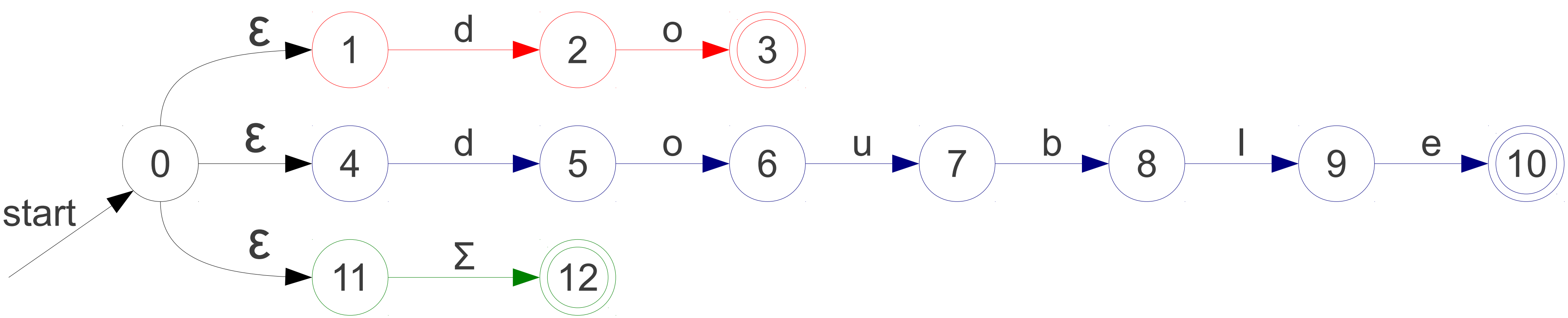


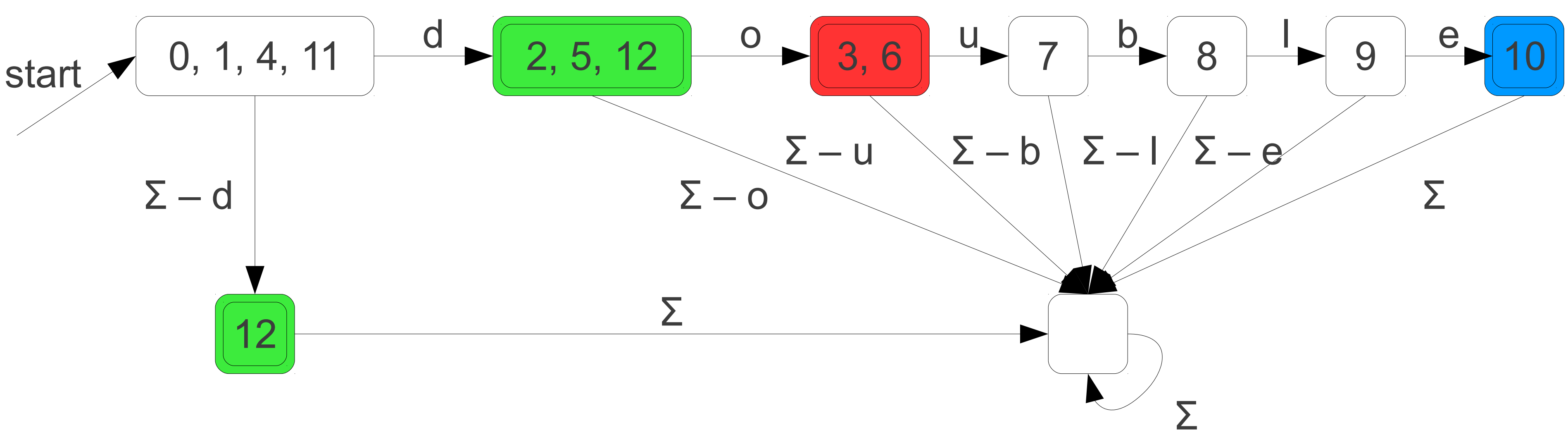
From NFA to DFA





From NFA to DFA





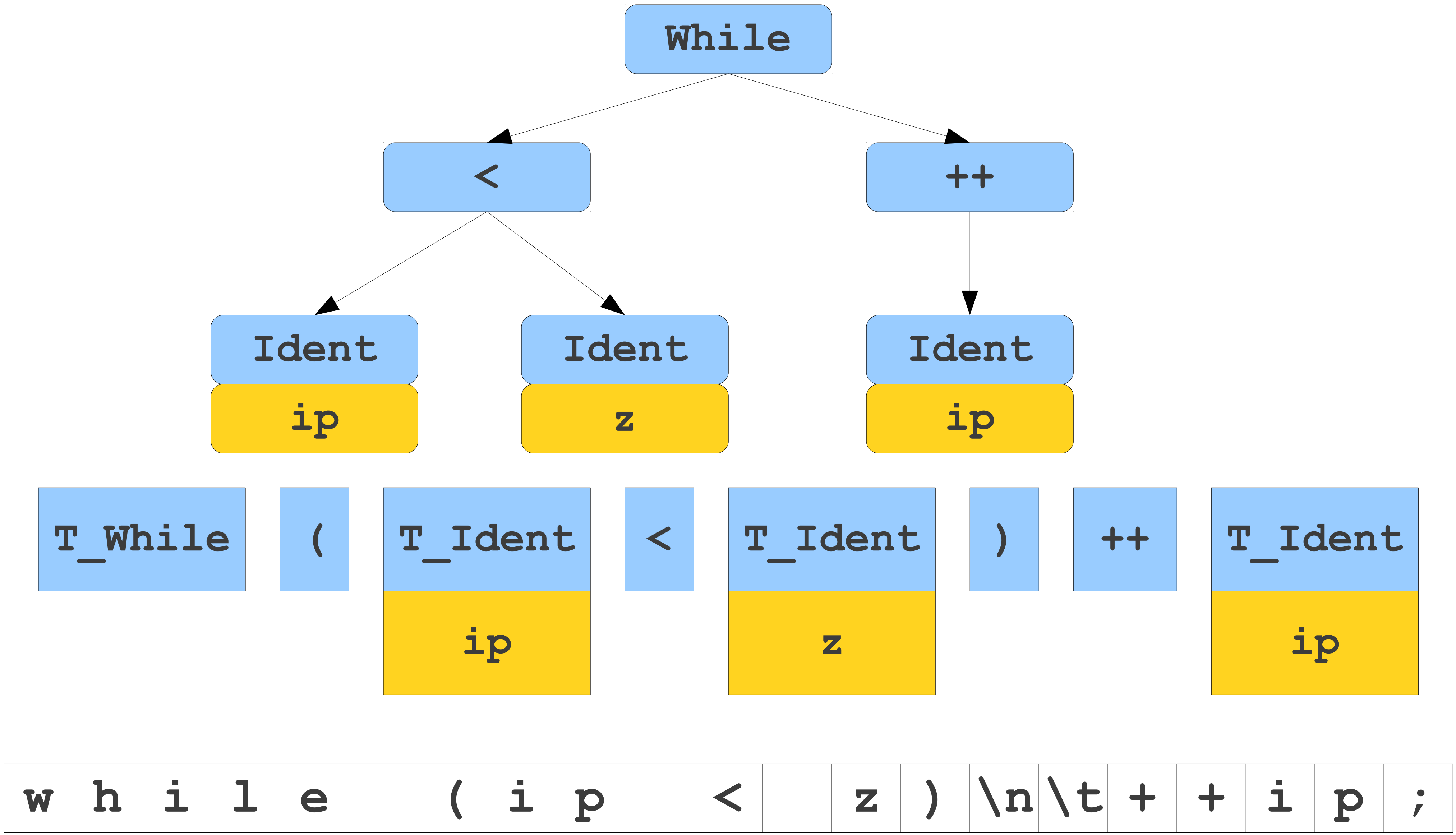
Modified Subset Construction

- Instead of marking whether a state is accepting, remember *which token type* it matches.
- Break ties with priorities.
- When using DFA as a scanner, consider the DFA “stuck” if it enters the state corresponding to the empty set.

Performance Concerns

- The NFA-to-DFA construction can introduce *exponentially* many states.
- Time/memory tradeoff:
  - Low-memory NFA has higher scan time.
  - High-memory DFA has lower scan time.
- Could use a hybrid approach by simplifying NFA before generating code.

Real-World Scanning: **Python**


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

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

Python Blocks

- Scoping handled by whitespace:

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

- What does that mean for the scanner?

Whitespace Tokens

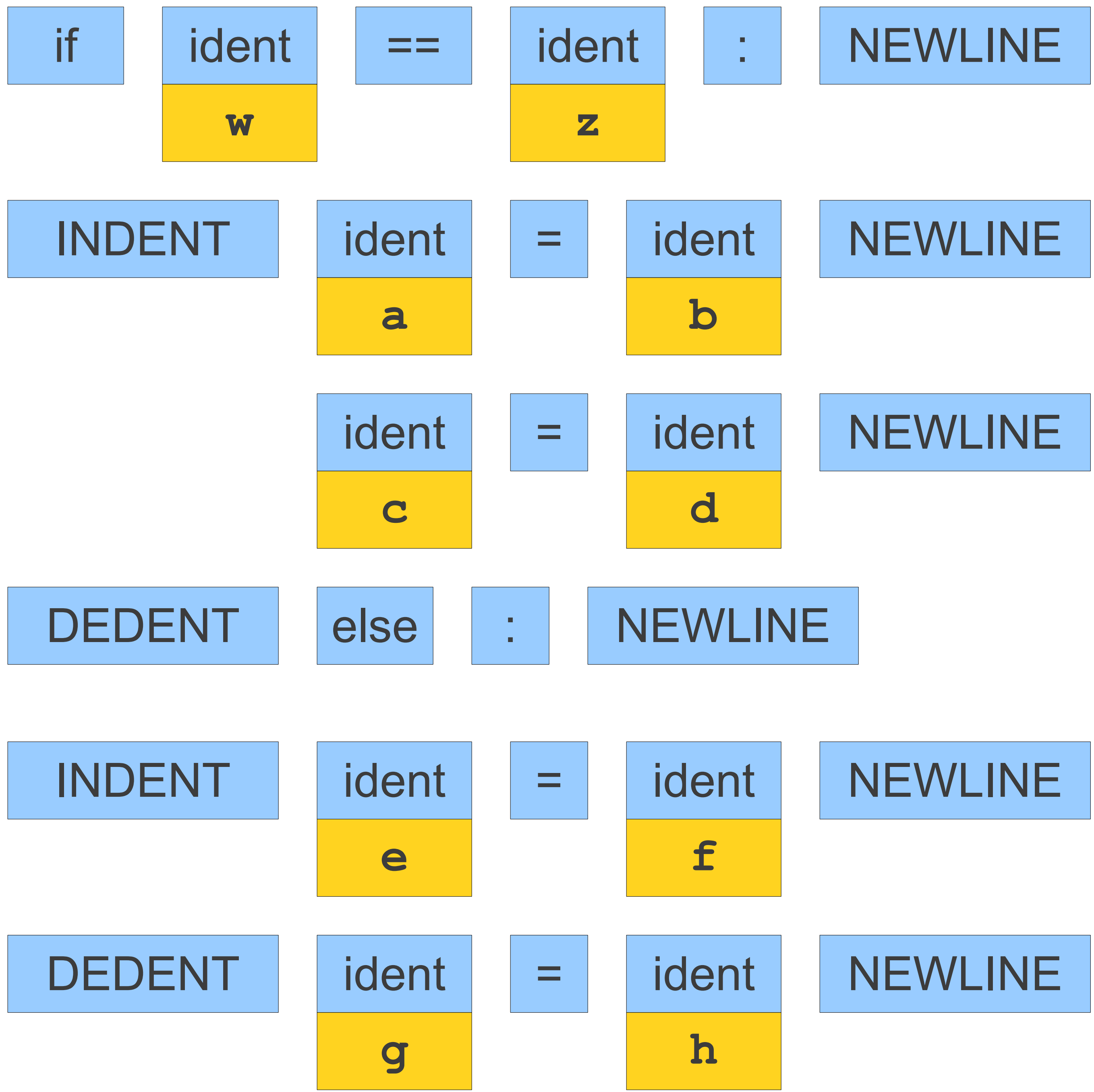
- Special tokens inserted to indicate changes in levels of indentation.
- **NEWLINE** marks the end of a line.
- **INDENT** indicates an increase in indentation.
- **DEDENT** indicates a decrease in indentation.
- Note that INDENT and DEDENT encode *change* in indentation, not the total amount of indentation.

Scanning Python

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

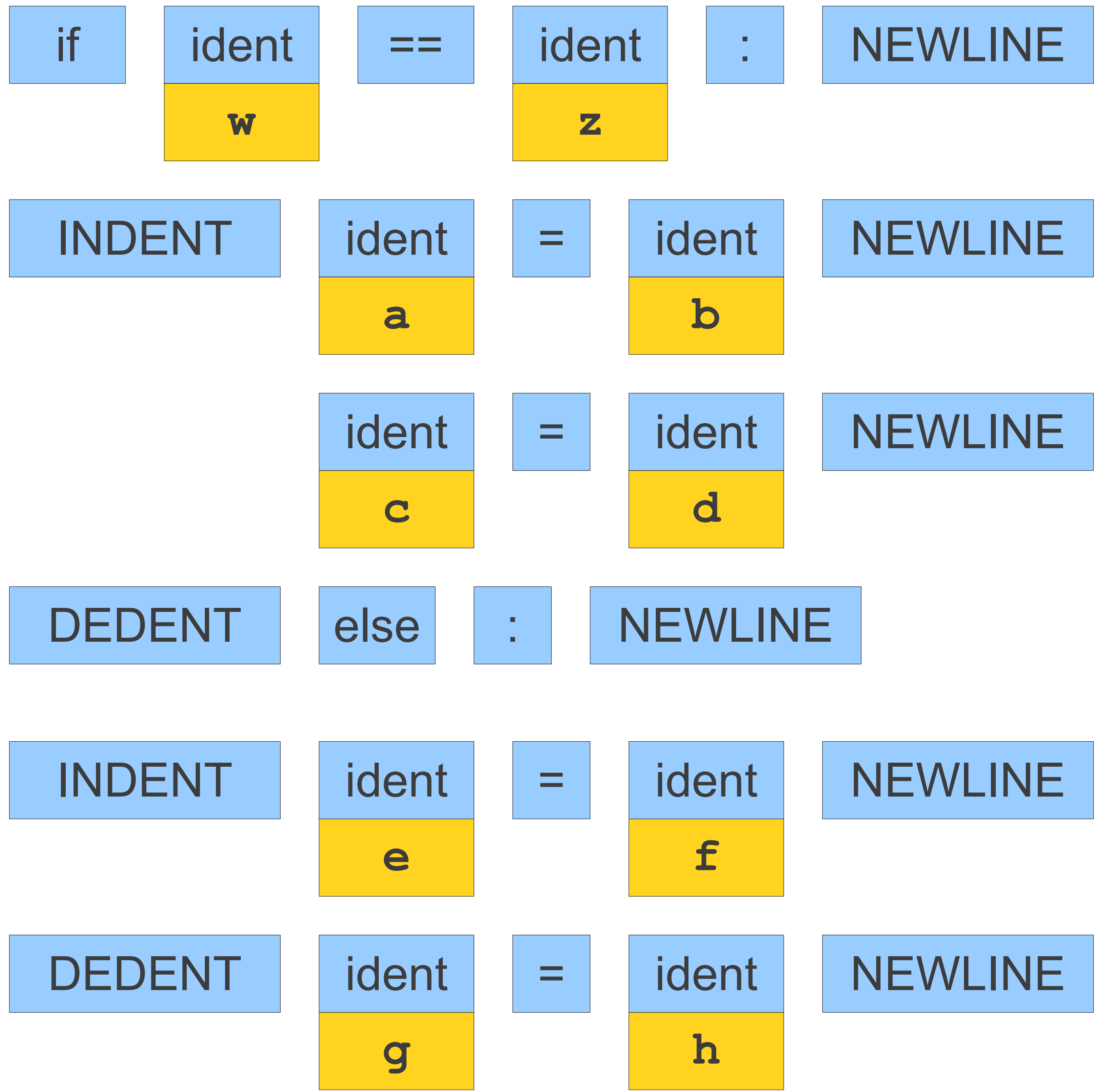
Scanning Python

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



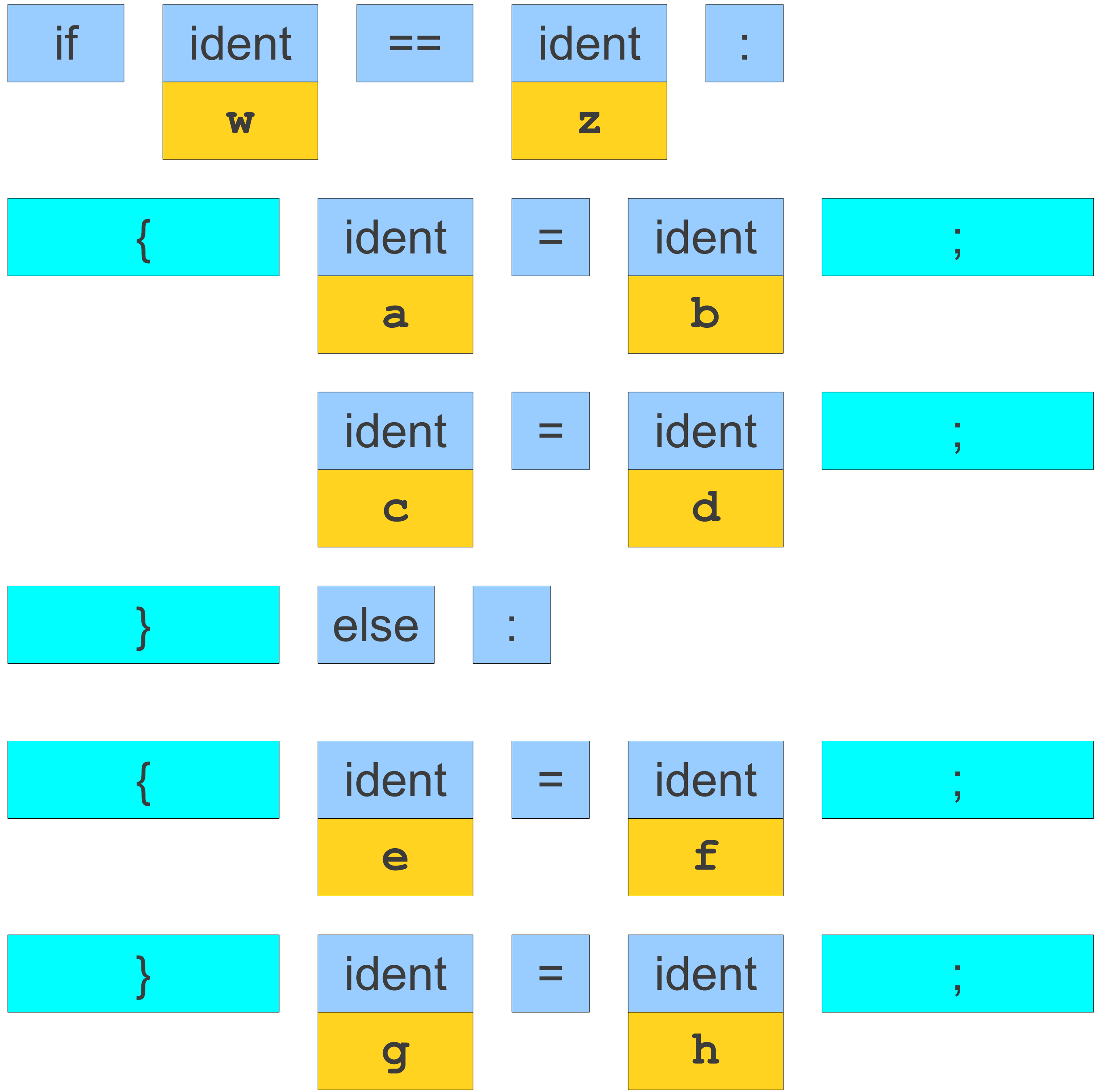
Scanning Python

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



Scanning Python

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



Where to INDENT/DEDENT?

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

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

Interesting Observation

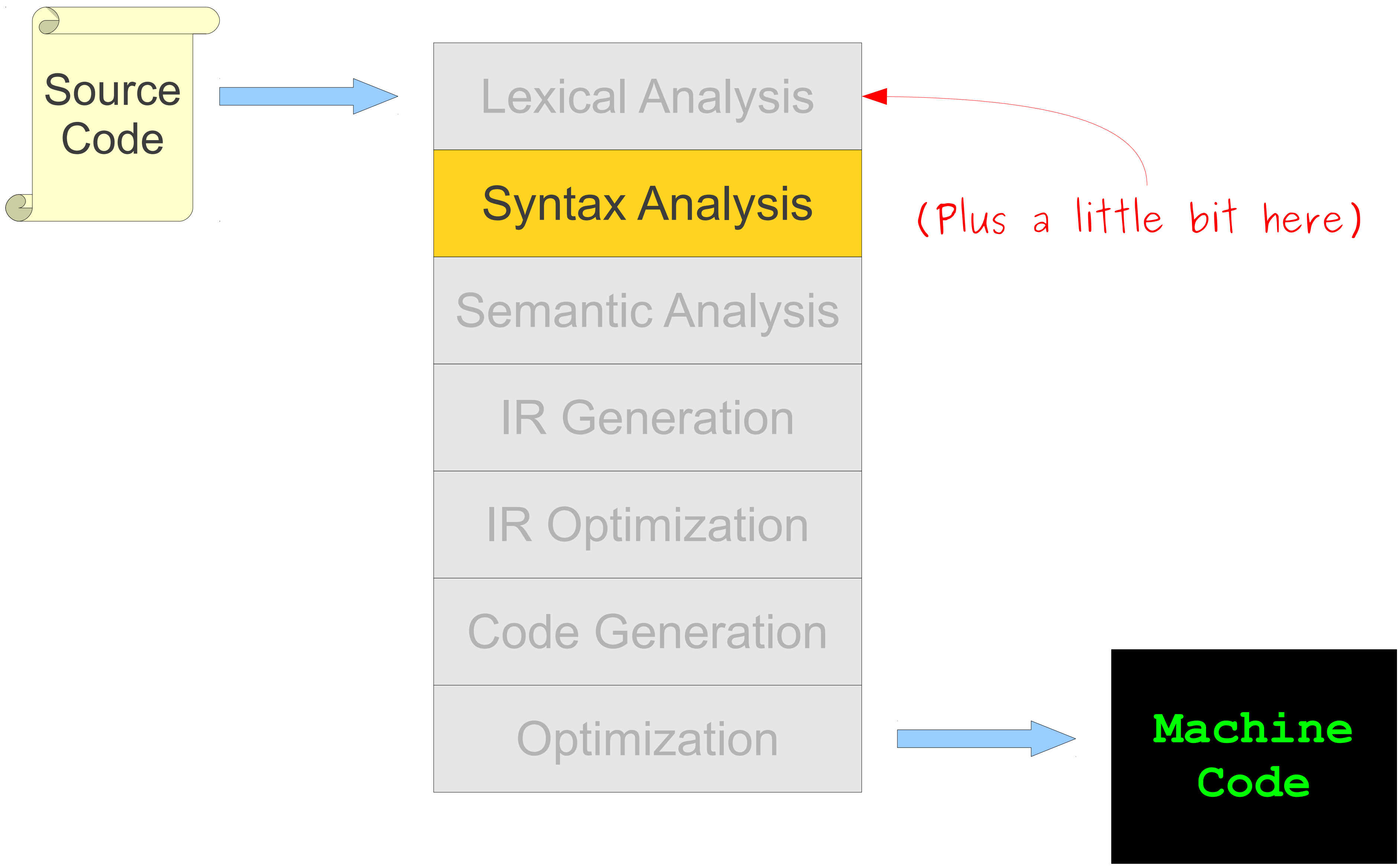
- Normally, more text on a line translates into more tokens.
- With DEDENT, *less* text on a line often means more tokens:

```
if cond1:
    if cond2:
        if cond3:
            if cond4:
                if cond5:
                    statement1
statement2
```

Summary

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

Next Time


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