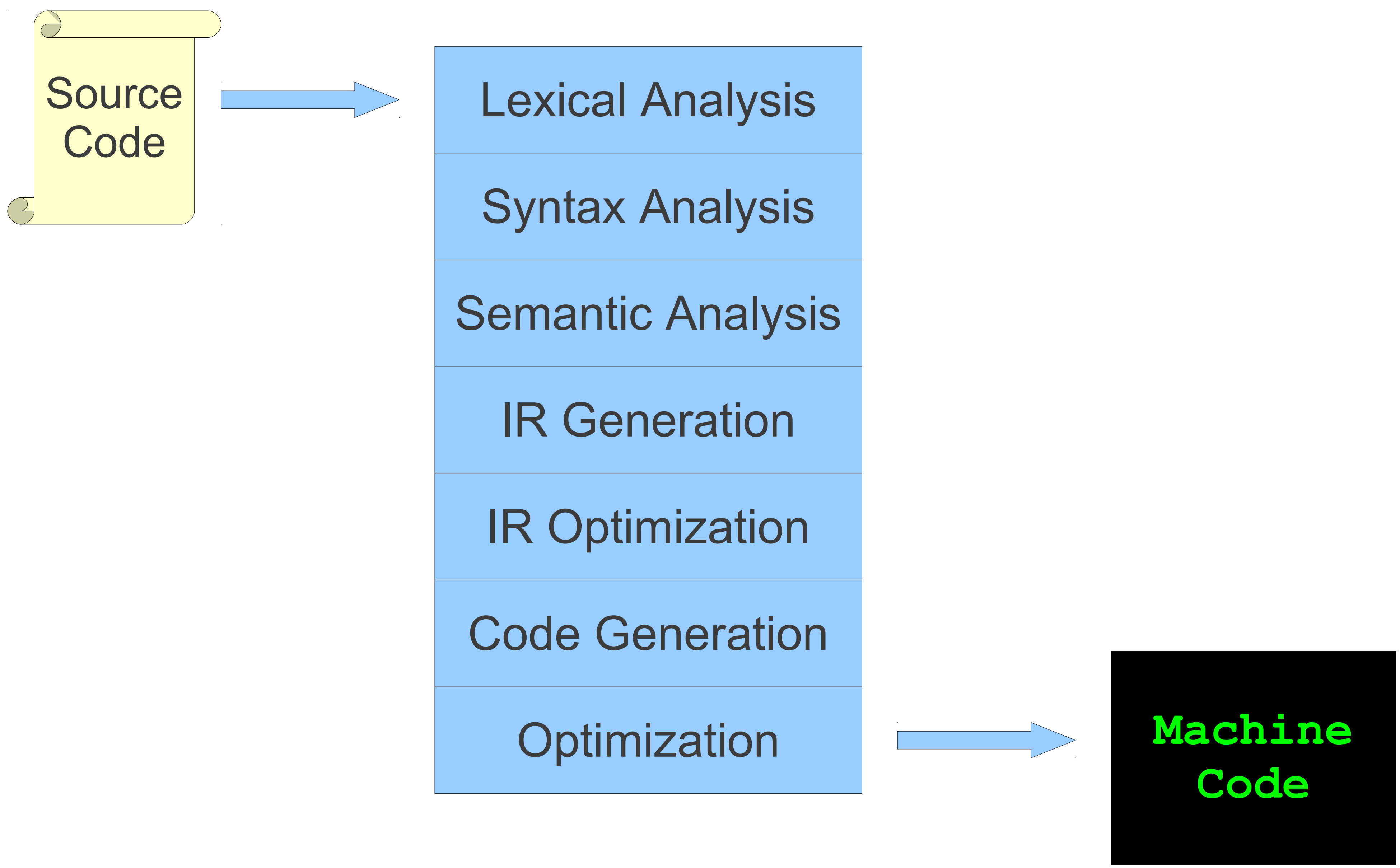
Lexical Analysis

Dec 6, 2021

Previously on EECS 483...

Structure of a modern compiler



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

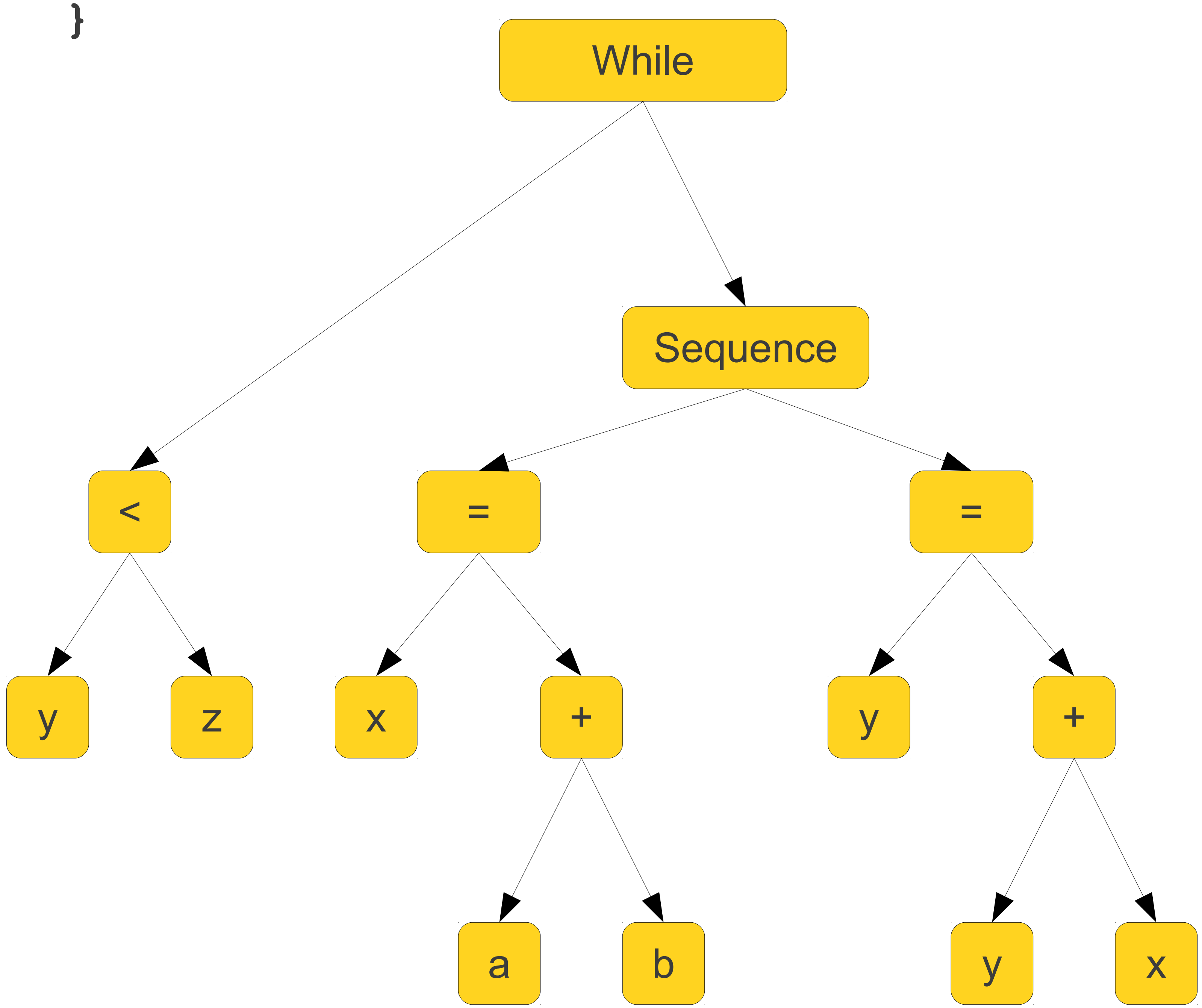
Optimization

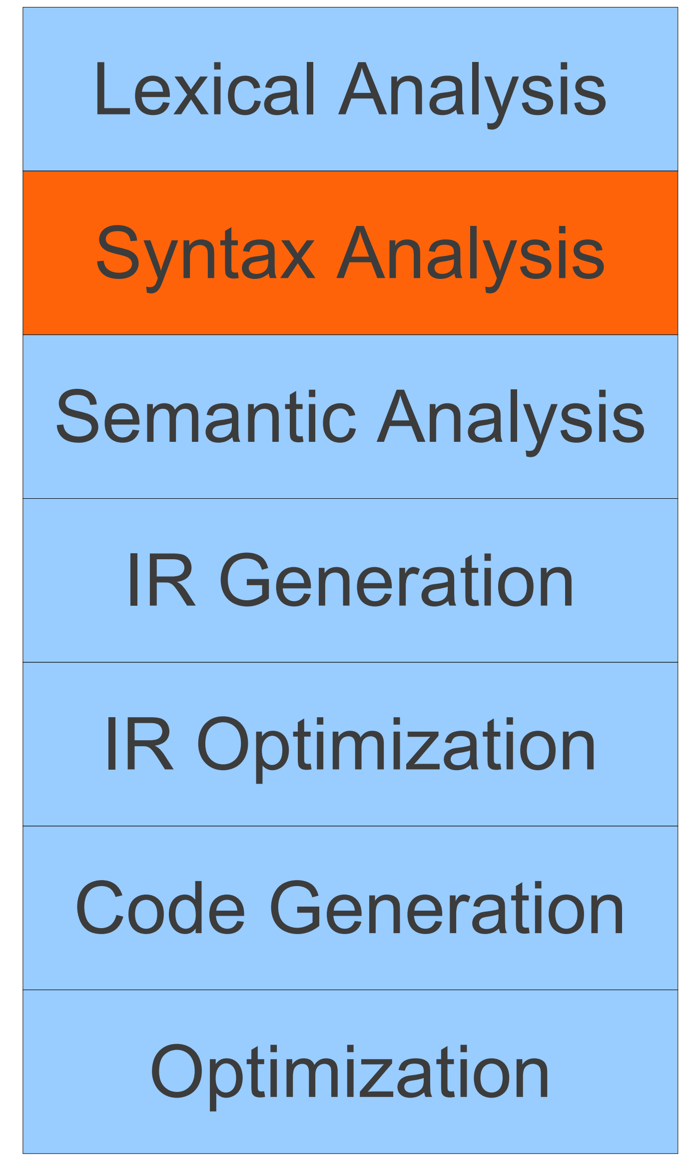
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

Lexical analysis (Scanning): Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)

```
while (y < z) {
    int x = a + b;
    y += x;
}
```





Syntax analysis (Parsing): Convert a linear structure – sequence of tokens – to a hierarchical tree-like structure - abstract syntax tree (AST)

Goal of Lexical Analysis

Breaking the program down into words or “tokens”

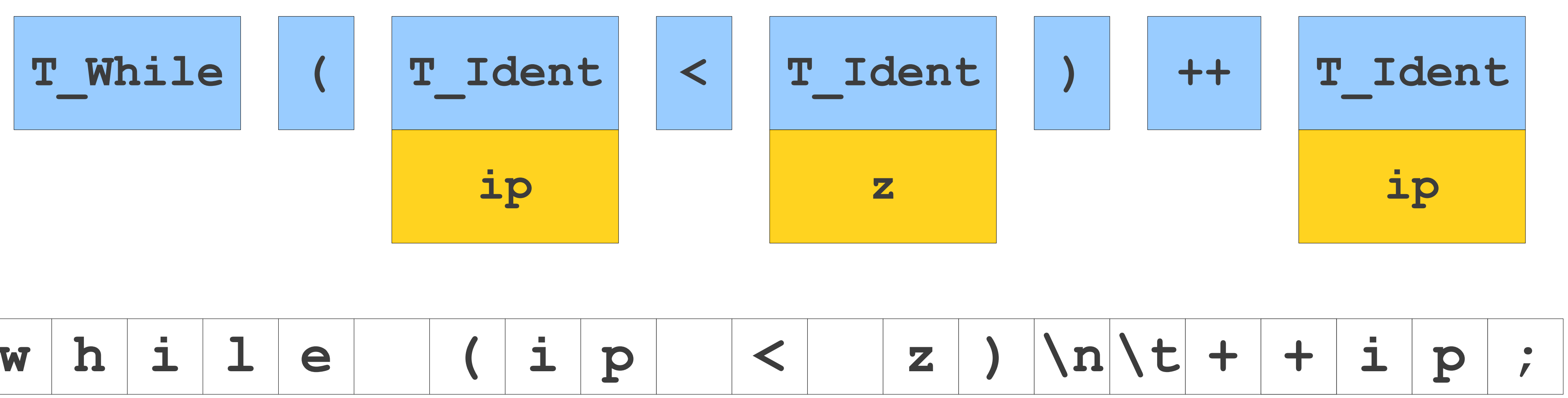
Input: code (character stream)

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

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

Goal of Lexical Analysis

Output: Token Stream




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

What's a token?

- What's a lexical unit of code?

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

What is my name ?

Token Type

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

- Keyword: for int if else while
- Punctuation: ( ) { } ;
- Operand: + - ++
- Relation: < > =
- Identifier: (variable name, function name) foo foo_2
- Integer, float point, string: 2345 2.0 “hello world”
- Whitespace, comment /* this code is awesome */

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

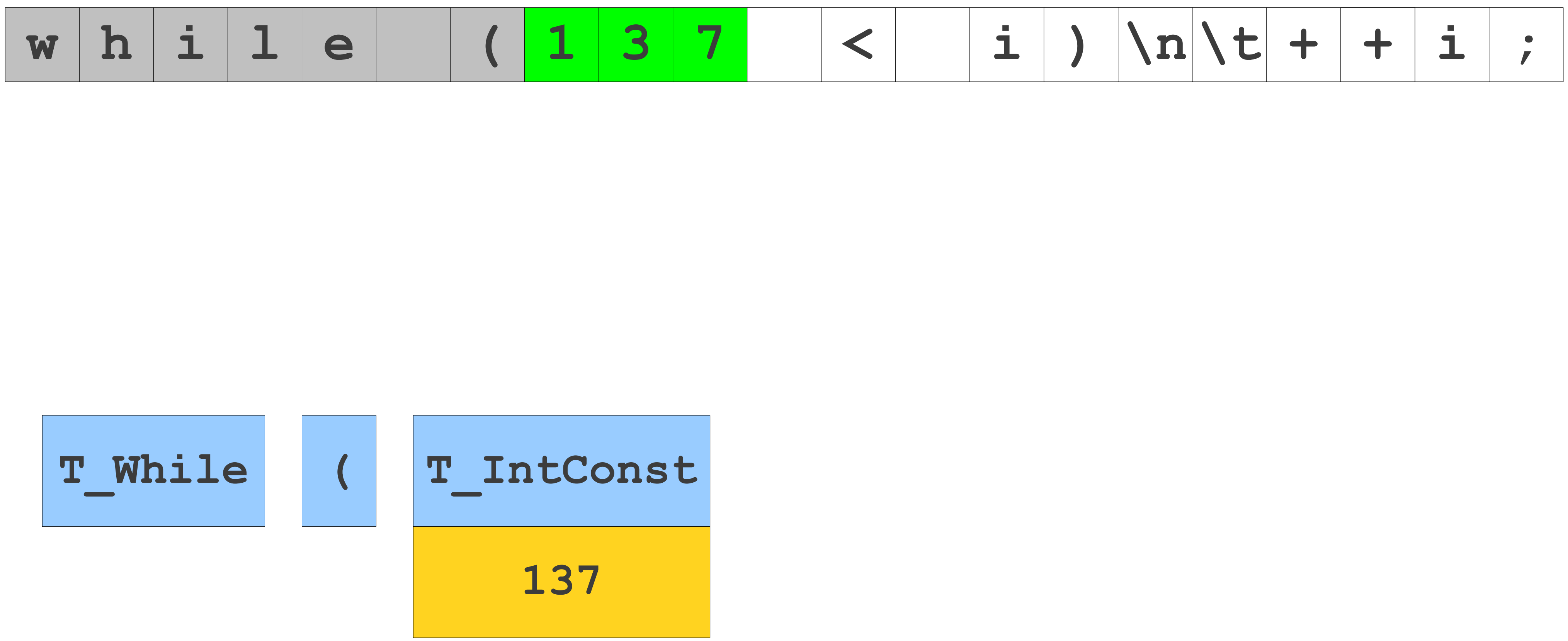
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Lexeme: the piece of the original program from which we made the token

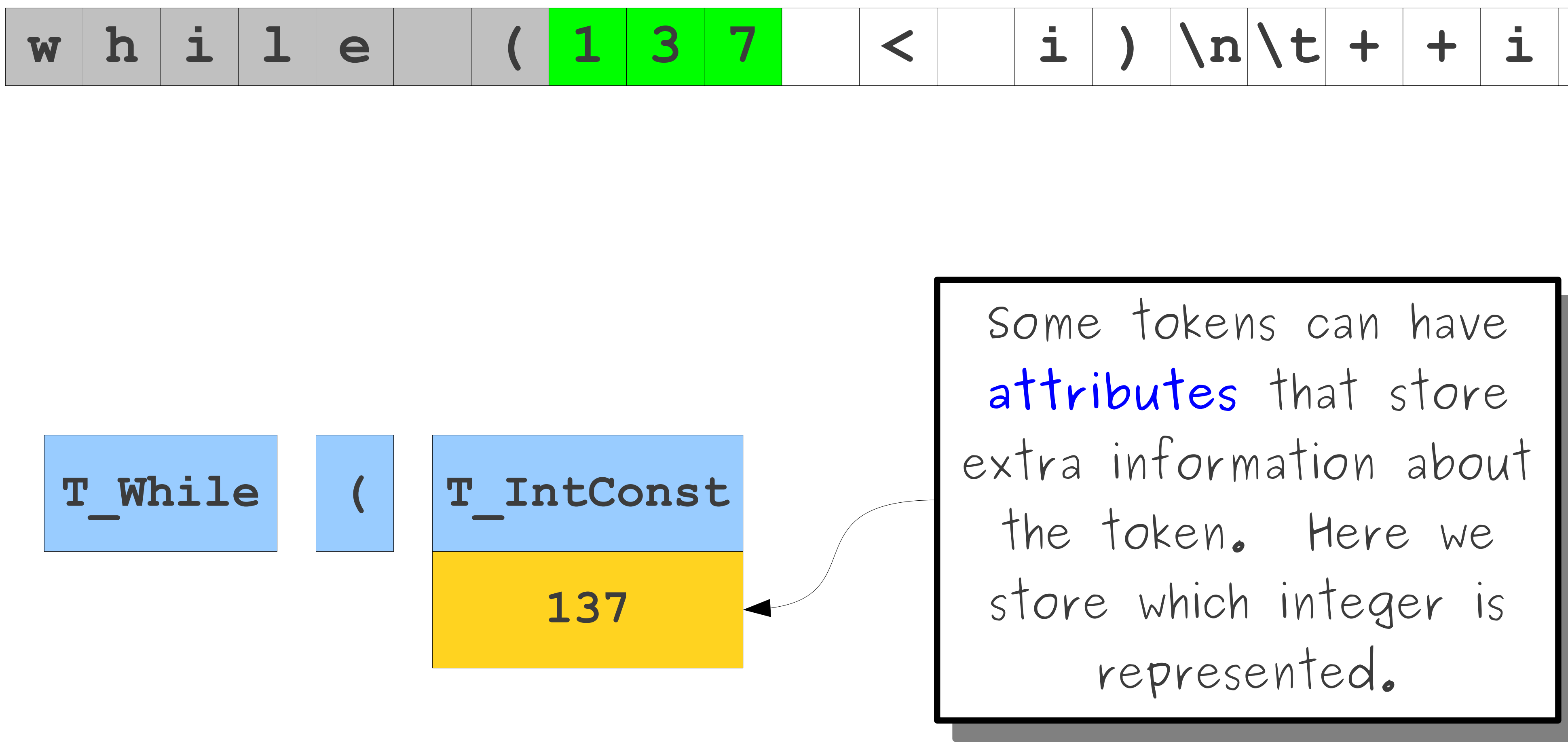
`T_While`

Token

Scanning a Source File



Scanning a Source File



Lexical Analyzer

- Recognize substrings that correspond to tokens: **lexemes**
  - **Lexeme:** actual text of the token
- For each lexeme, identify token type
  - < Token type, attribute>
  - attribute: optional, extra information, often numeric value

Challenges for Lexical Analyzer

- 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?
  - if
  - ifc
- How do we address these concerns efficiently?

Associate Lexemes to Tokens

- Tokens: categorize lexemes by what information they provide.
- Associate lexemes to token: Pattern matching
- How to describe patterns??

Token: Lexemes

- Keyword: for int if else while
- Punctuation: ( ) { } ;
- Operand: + - ++
- Relation: < > =
- Identifier: (variable name,function name) foo foo_2
- Integer, float point, string: 2345 2.0 “hello world”
- Whitespace, comment /* this code is awesome */

Finite possible lexemes

Infinite possible lexemes

- 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.

- What type of formal language should we use to describe tokens?

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

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.

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

Applied Regular Expressions

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

?

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

- More examples
  - Whitespace: [\t\n]+
  - Integers: [+\-]?[0-9]+
  - Hex numbers: 0x[0-9a-f]+
  - identifier
    - [A-Za-z]([A-Za-z]|[0-9])*

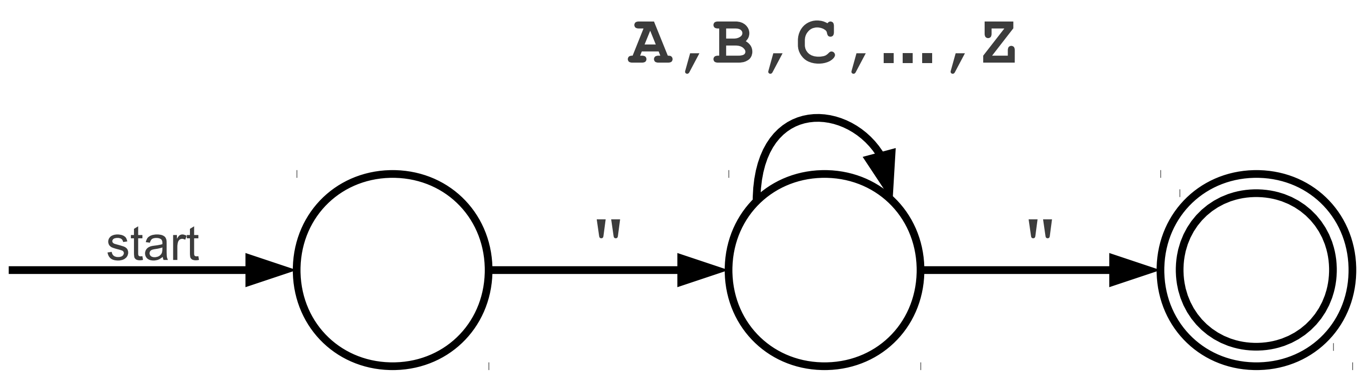
- Use regular expressions to describe token types
- How do we match regular expressions?

Recognizing Regular Language

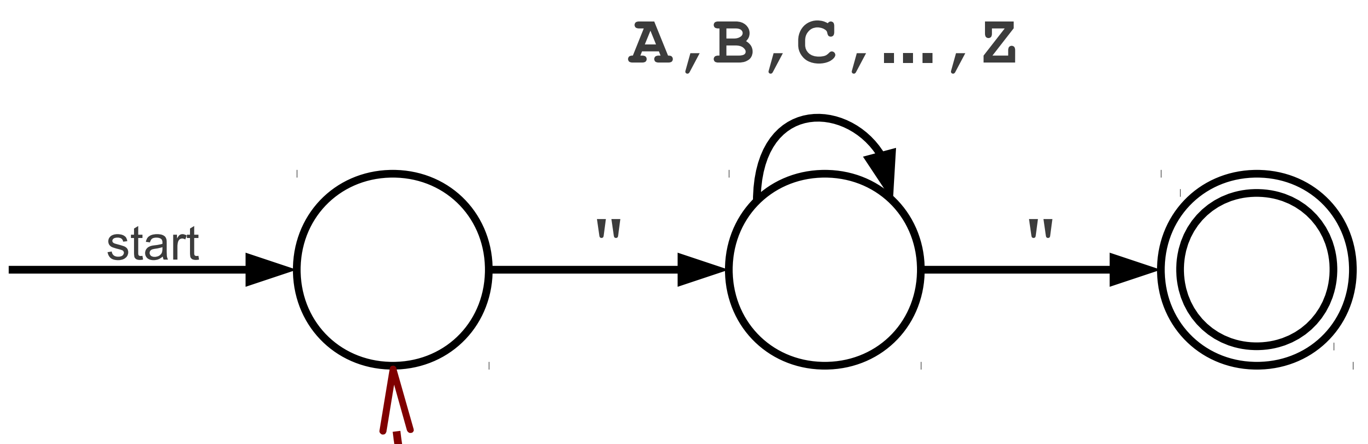
What is the machine that recognize regular language??

- Finite Automata
- DFA (Deterministic Finite Automata)
- NFA (Non-deterministic Finite Automata)

A Simple Automaton

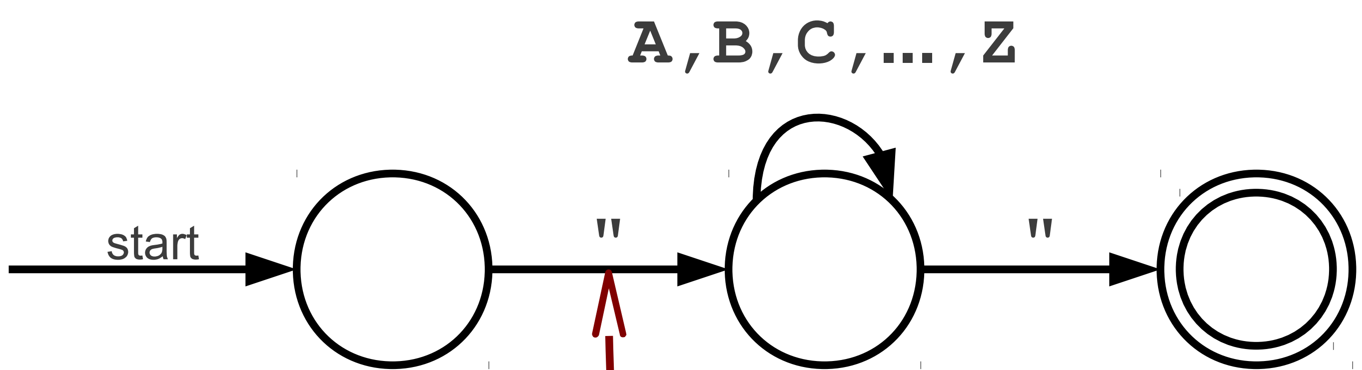


A Simple Automaton



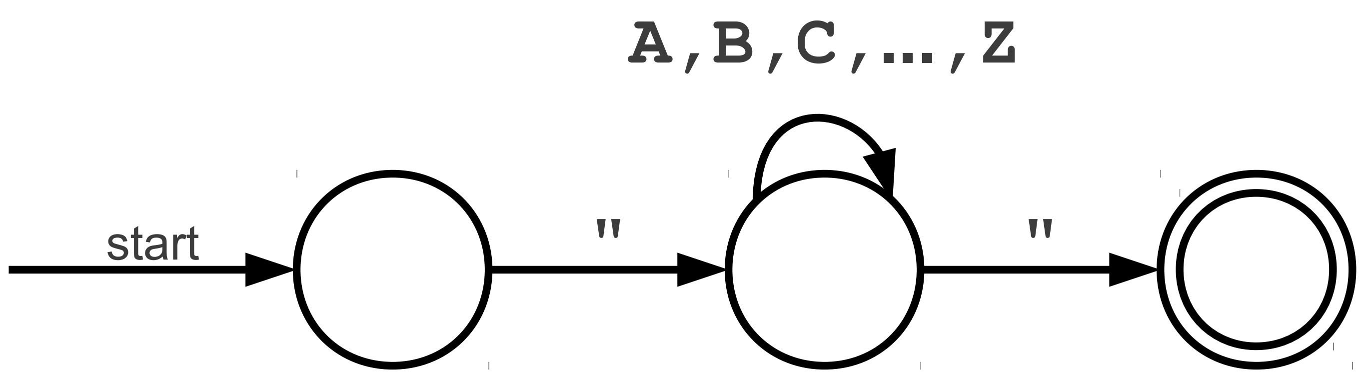
Each circle is a state of the automaton. The automaton's configuration is determined by what state(s) it is in.

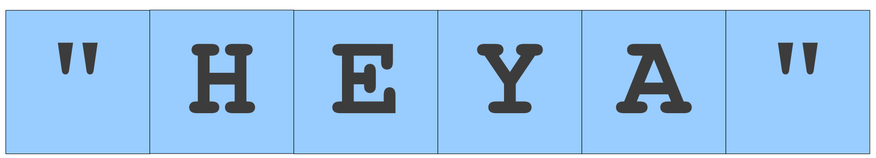
A Simple Automaton



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

A Simple Automaton

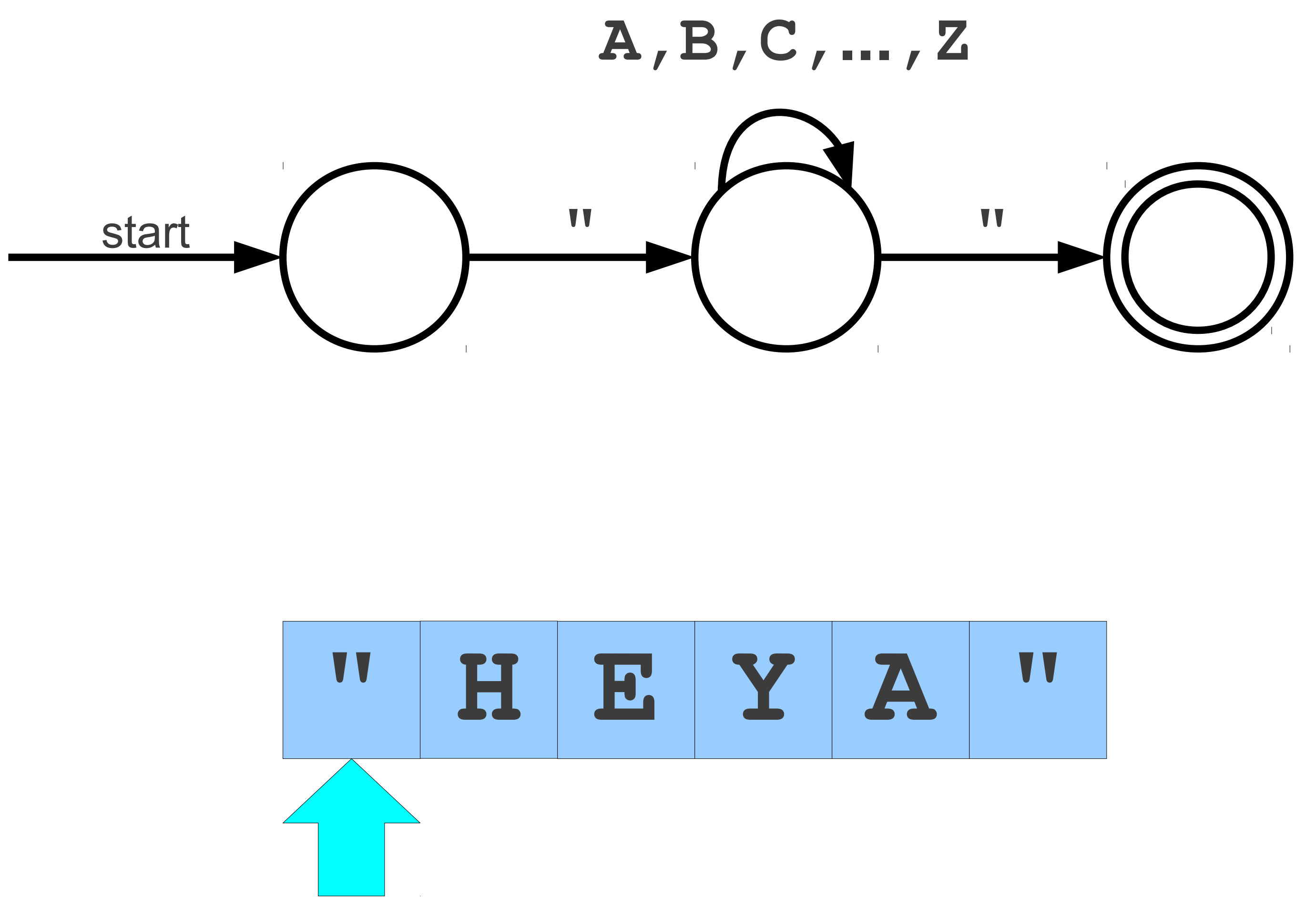




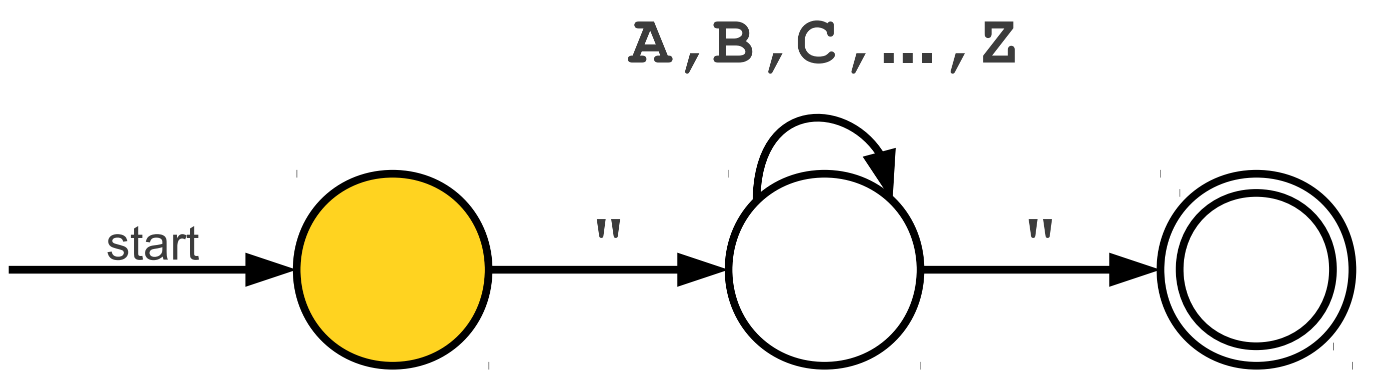
Finite Automata: Takes an input string and determines whether it's a valid sentence of a language

accept or reject

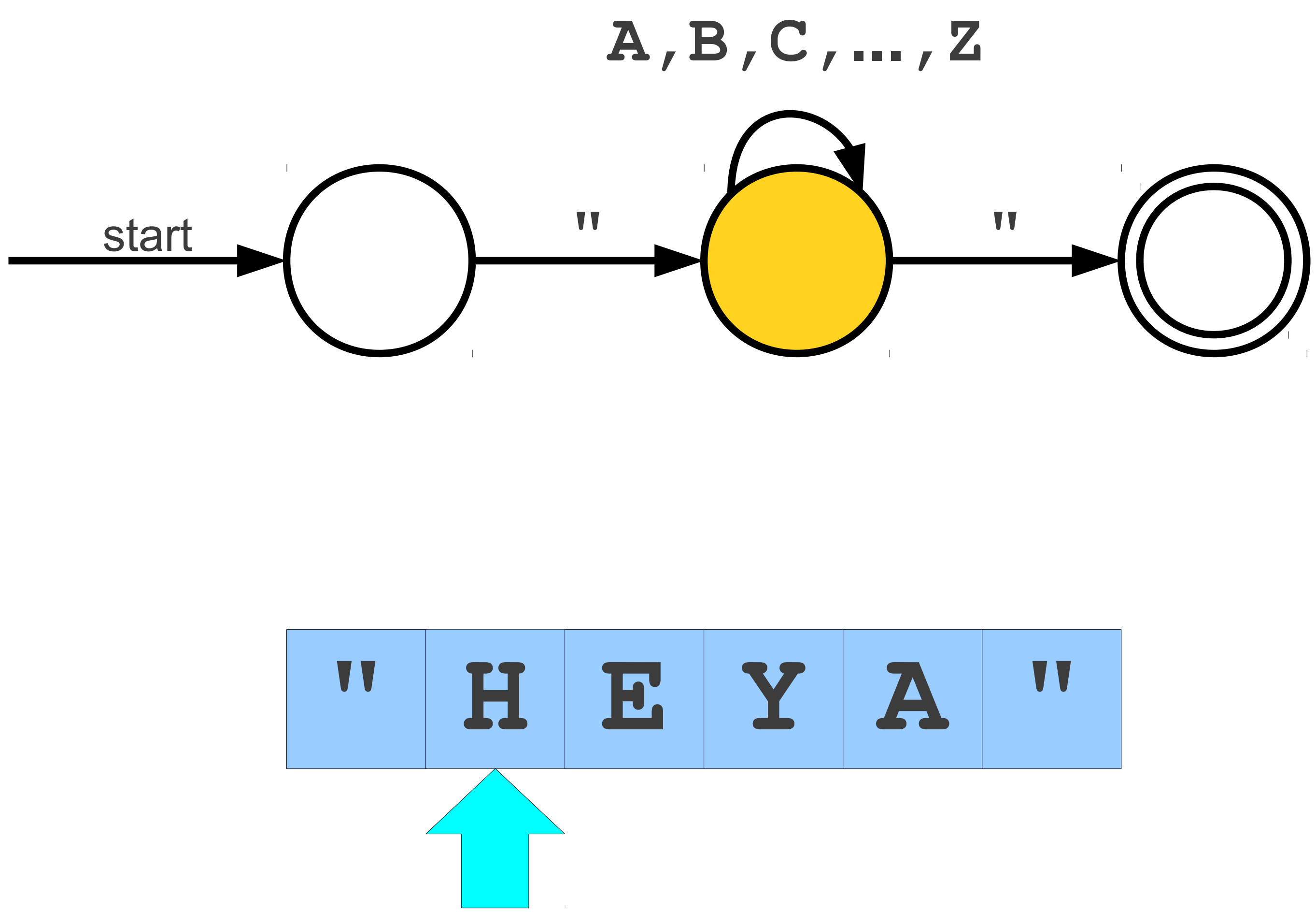
A Simple Automaton



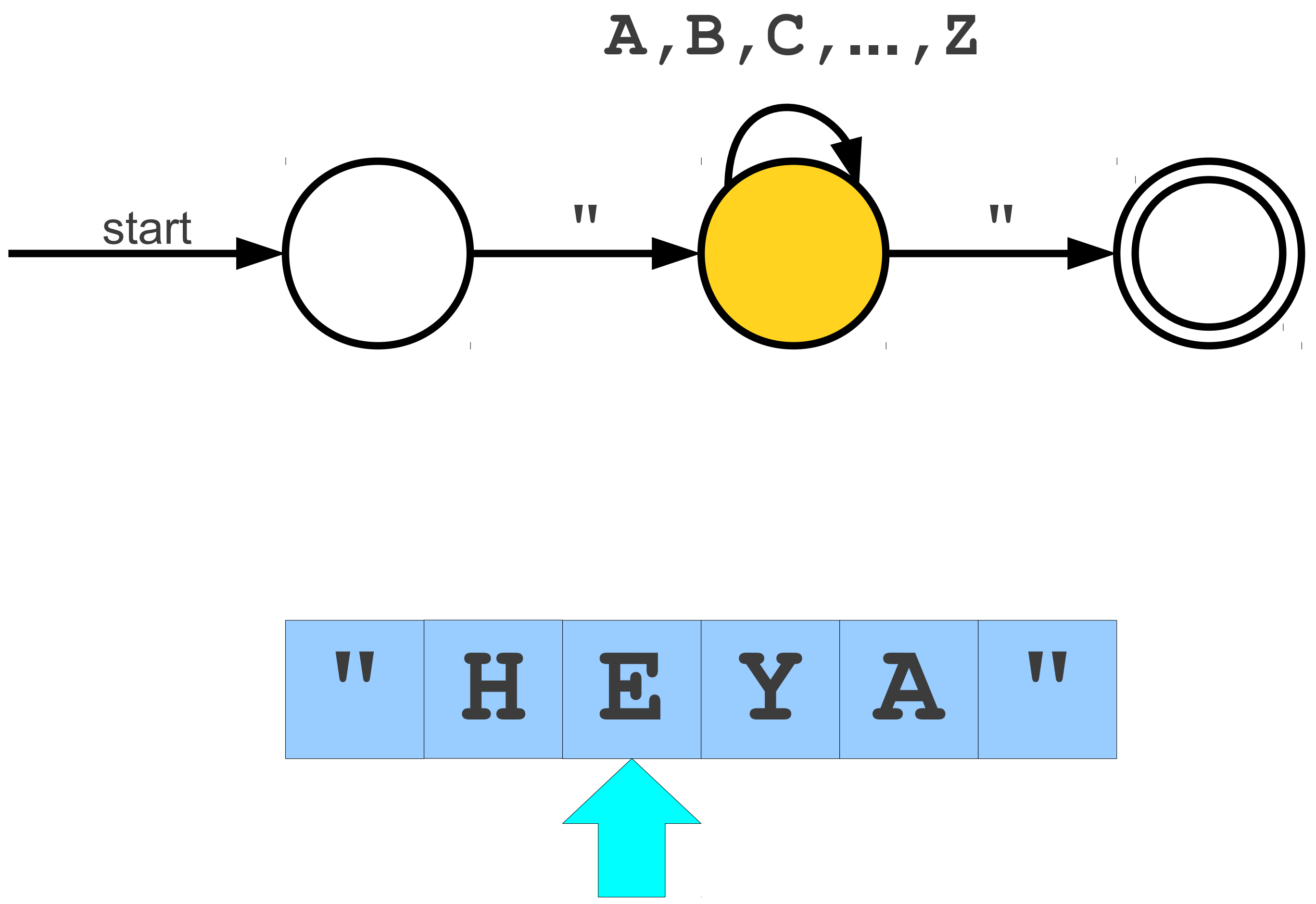
A Simple Automaton



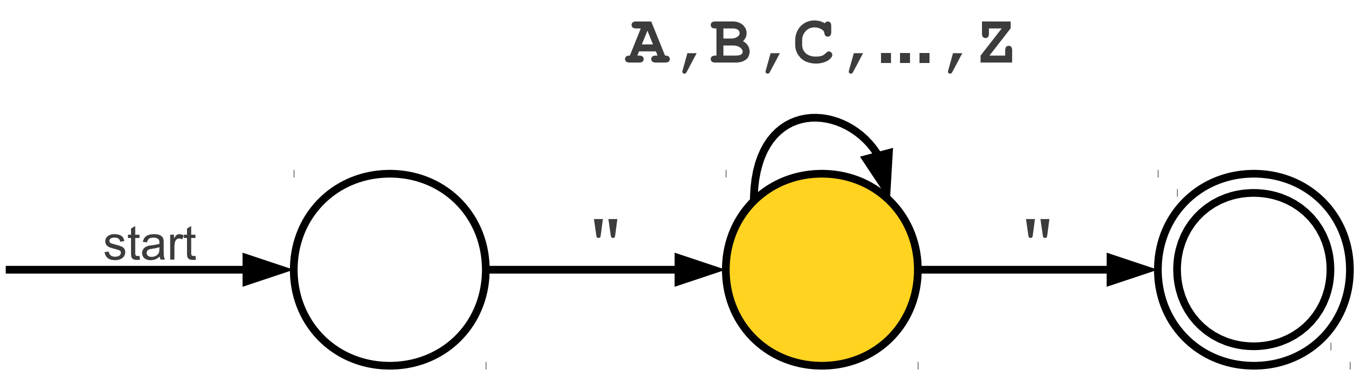
A Simple Automaton

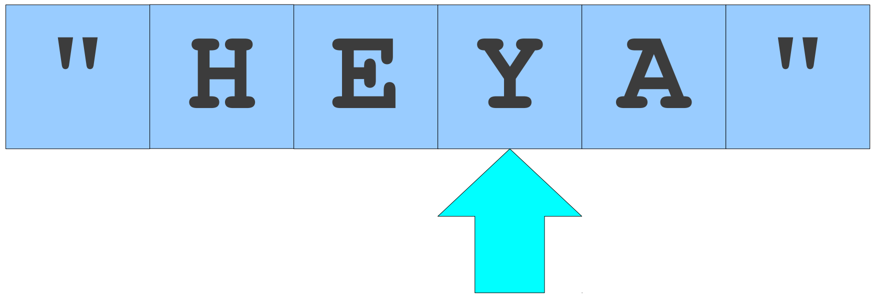


A Simple Automaton

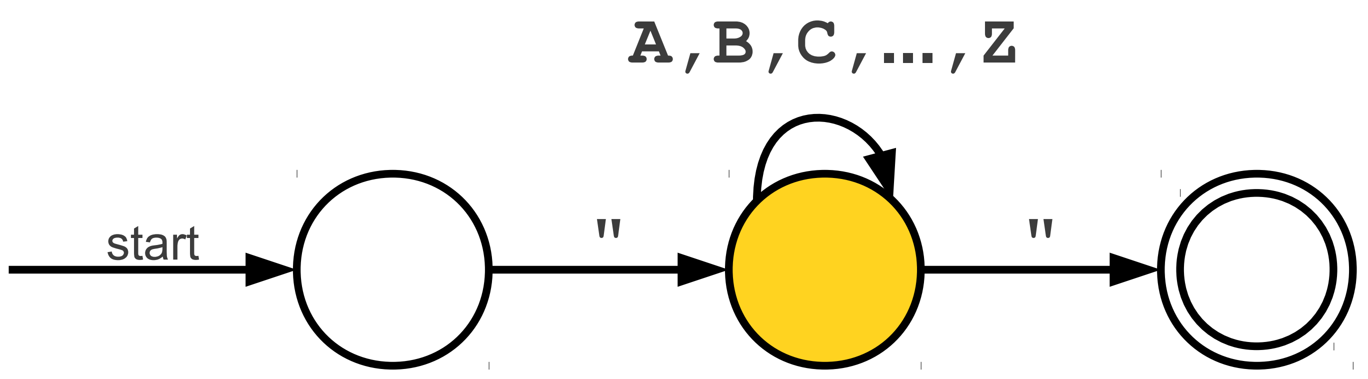


A Simple Automaton

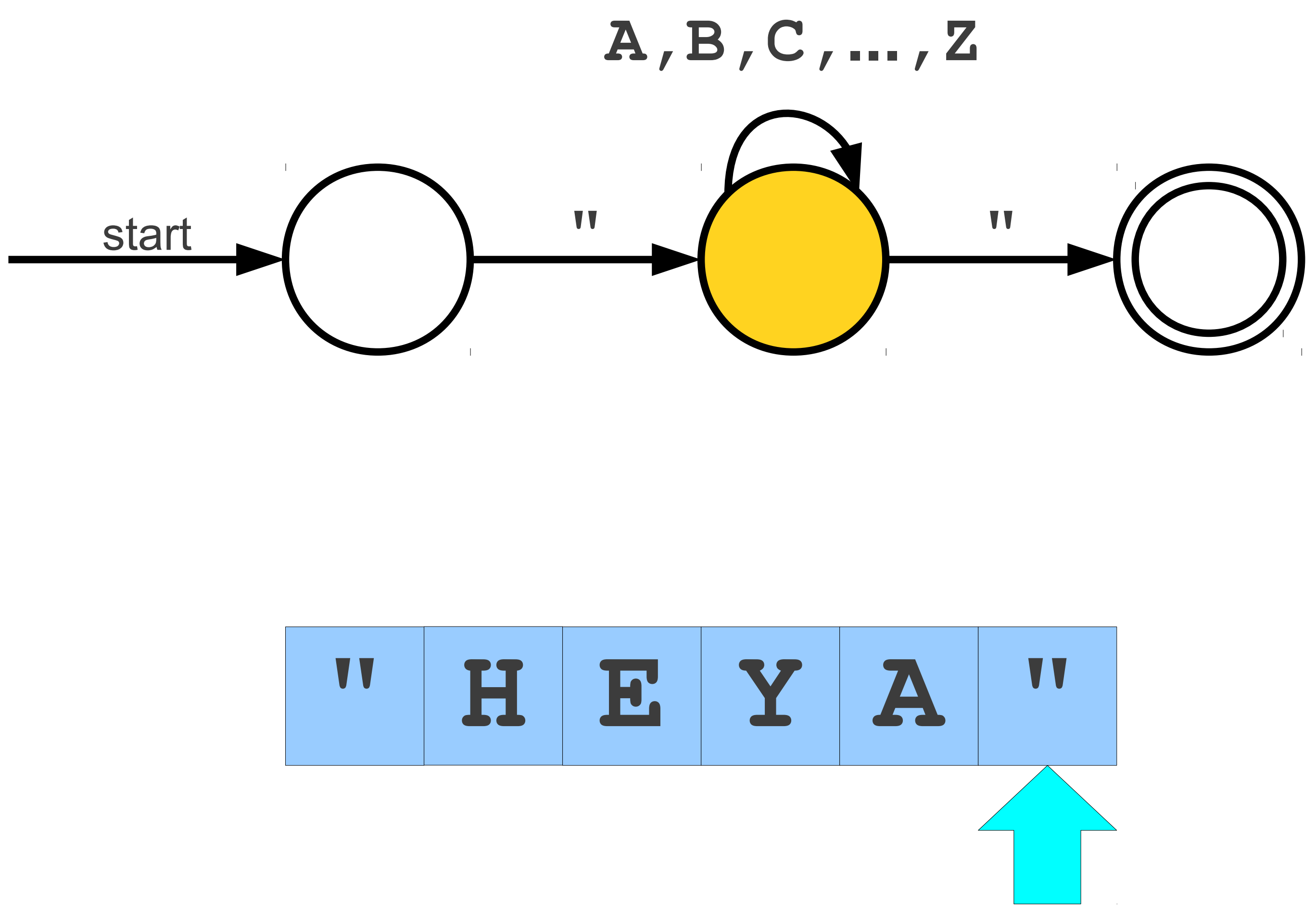




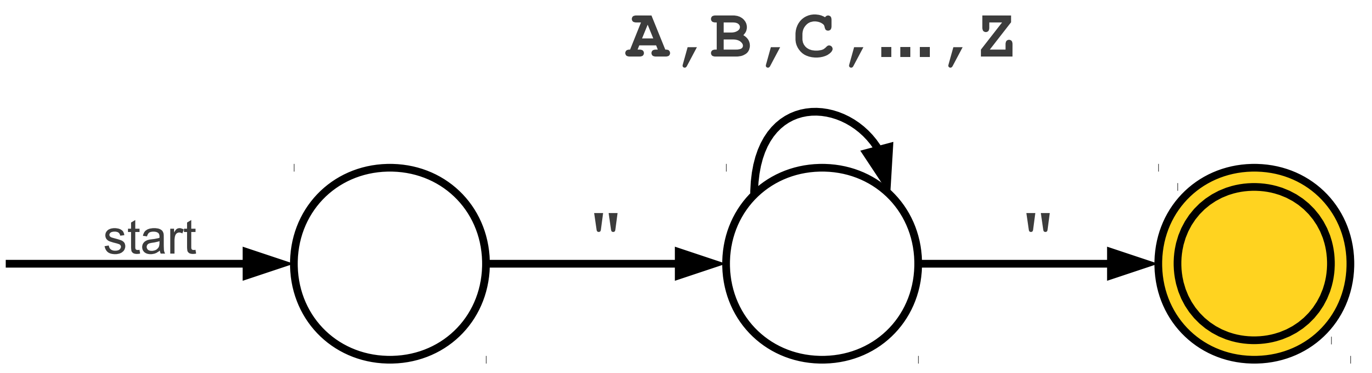
A Simple Automaton



A Simple Automaton

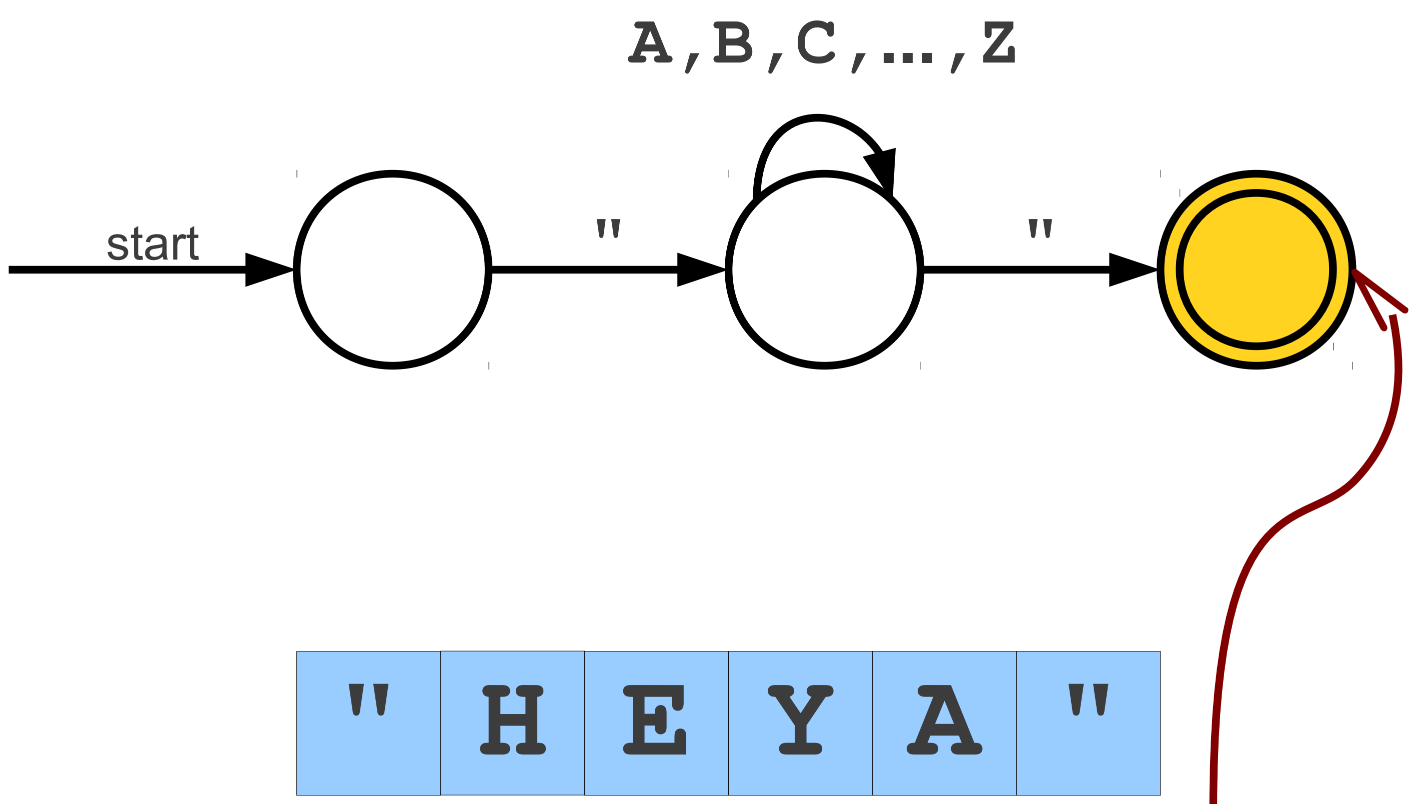


A Simple Automaton



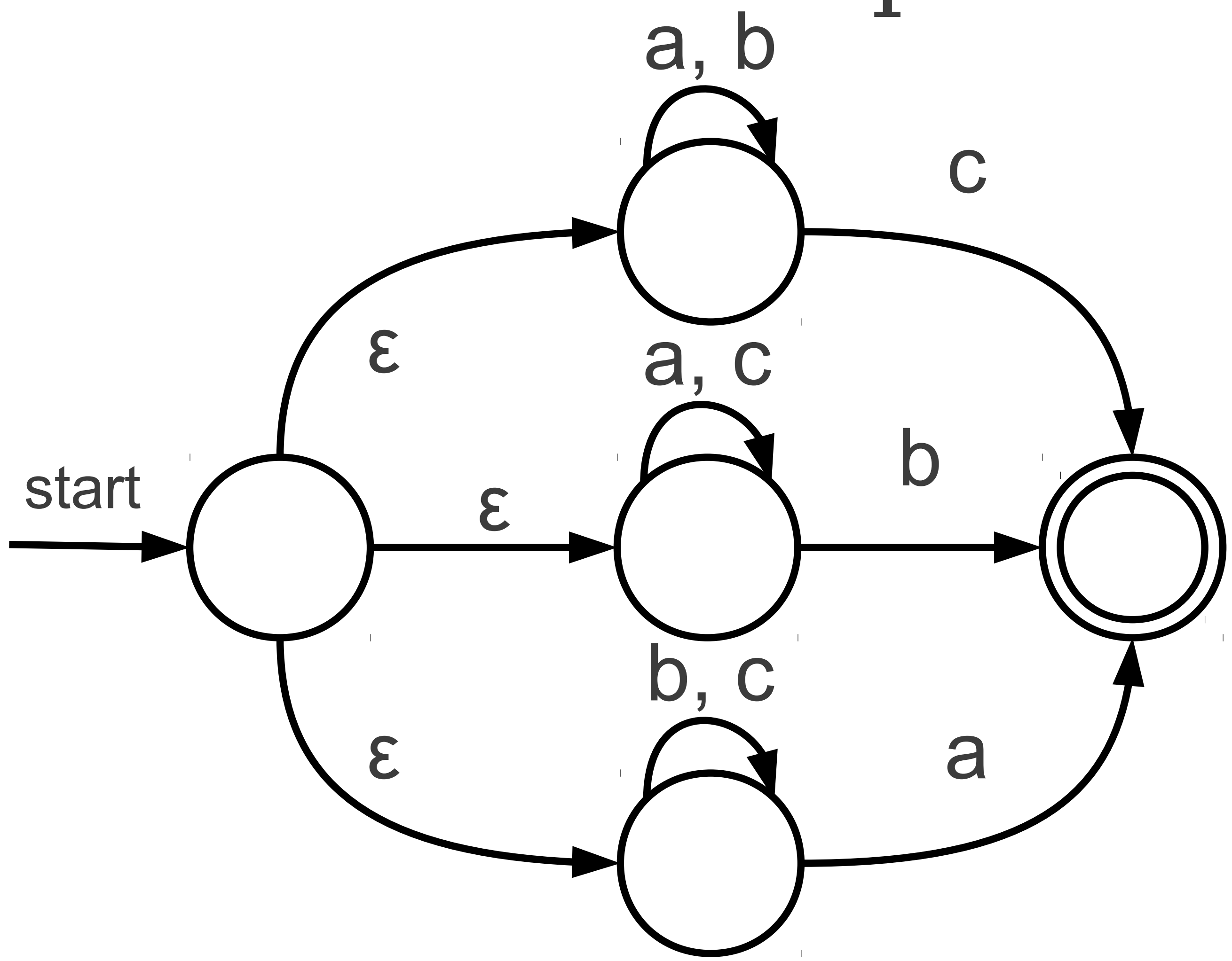
"	H	E	Y	A	"

A Simple Automaton

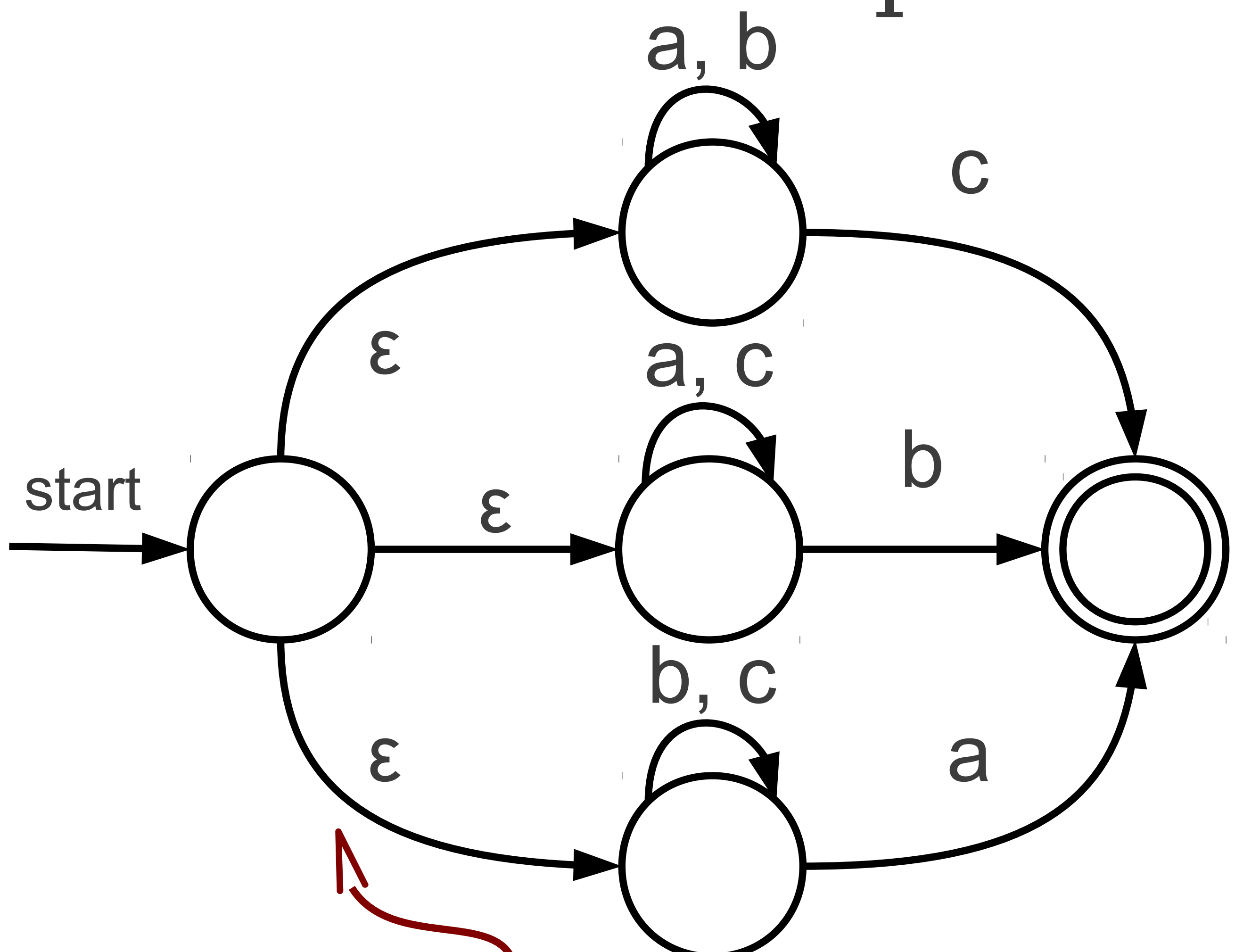


The double circle indicates that this state is an accepting state. The automaton accepts the string if it ends in an accepting state.

An Even More Complex Automaton

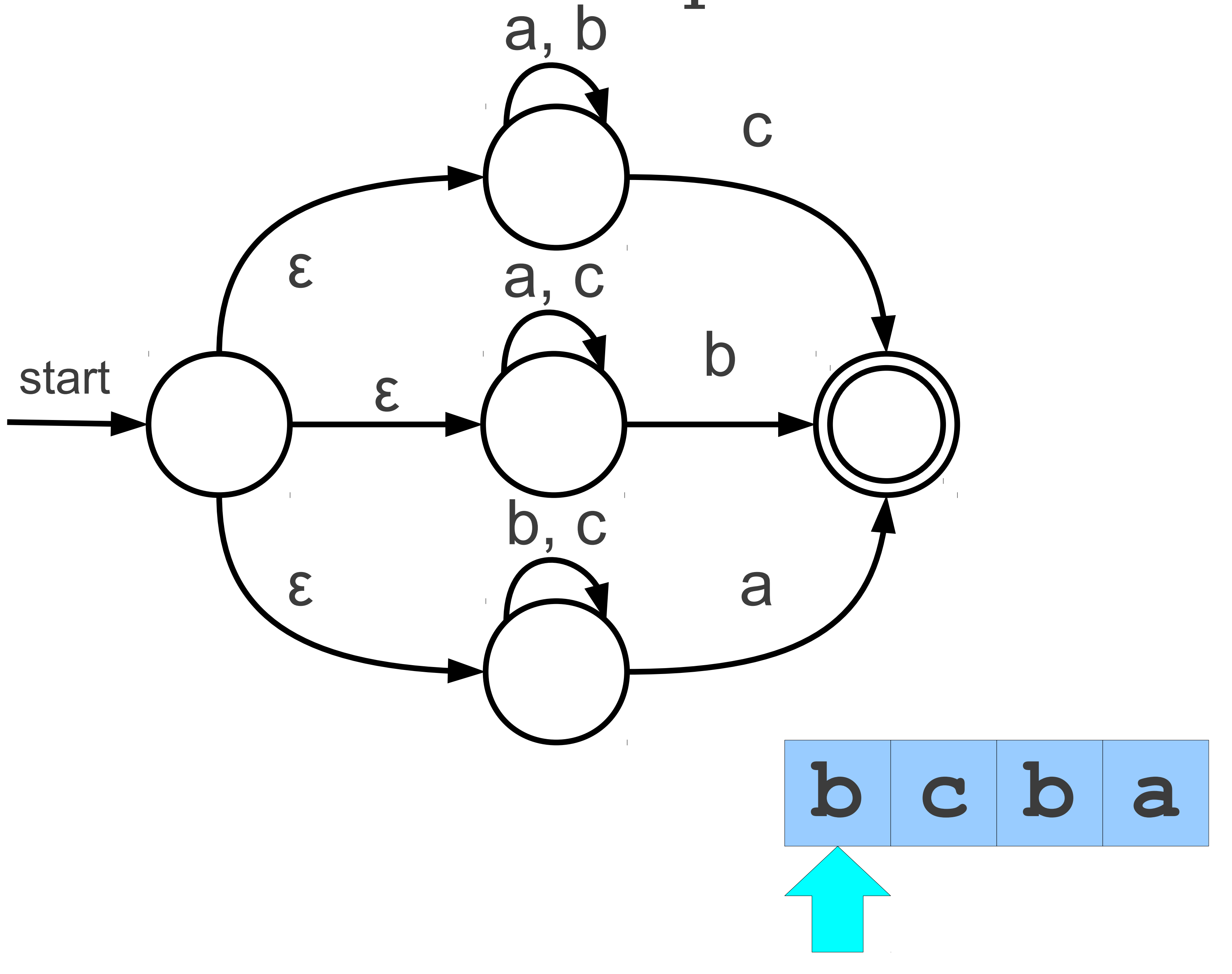


An Even More Complex Automaton

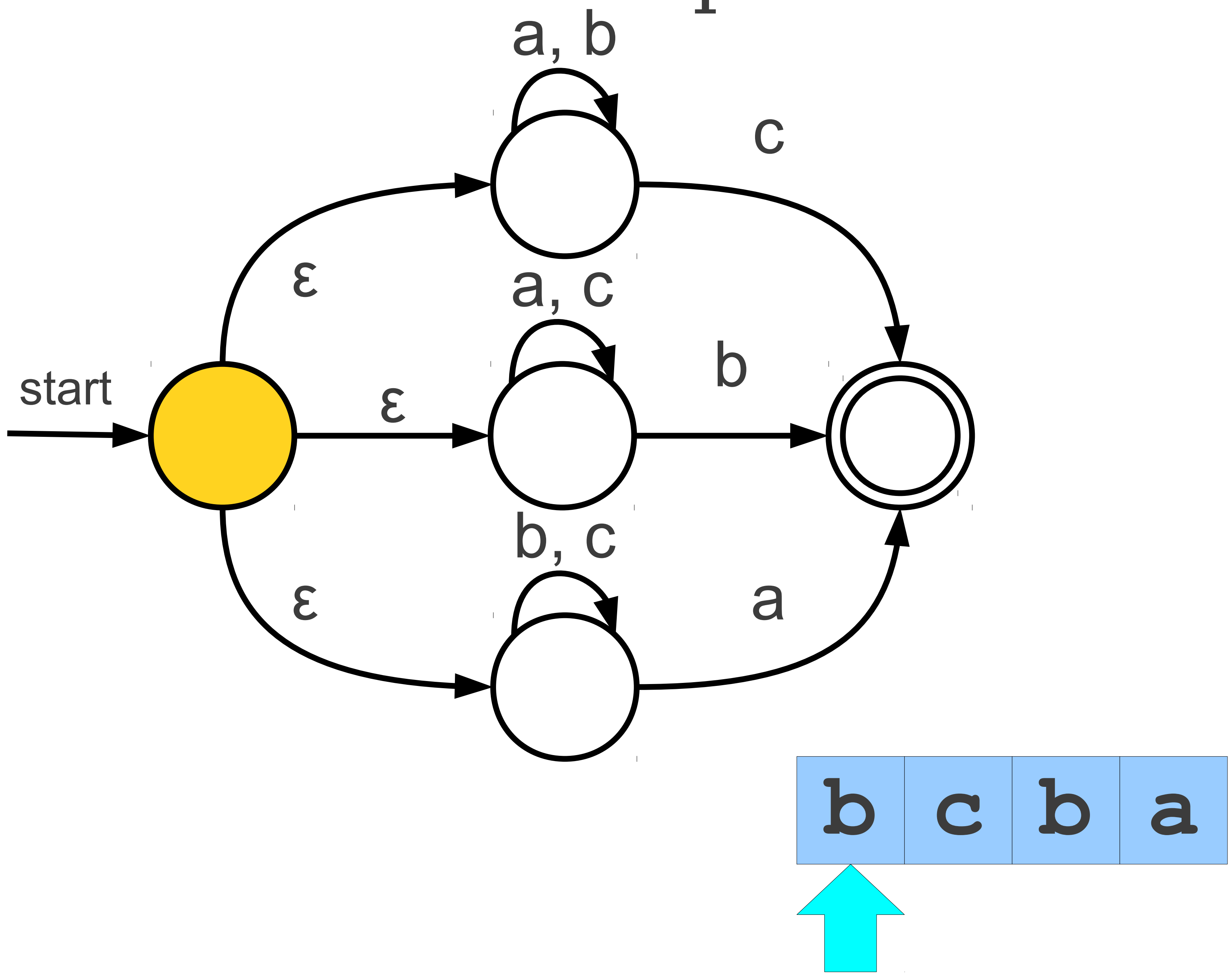


These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

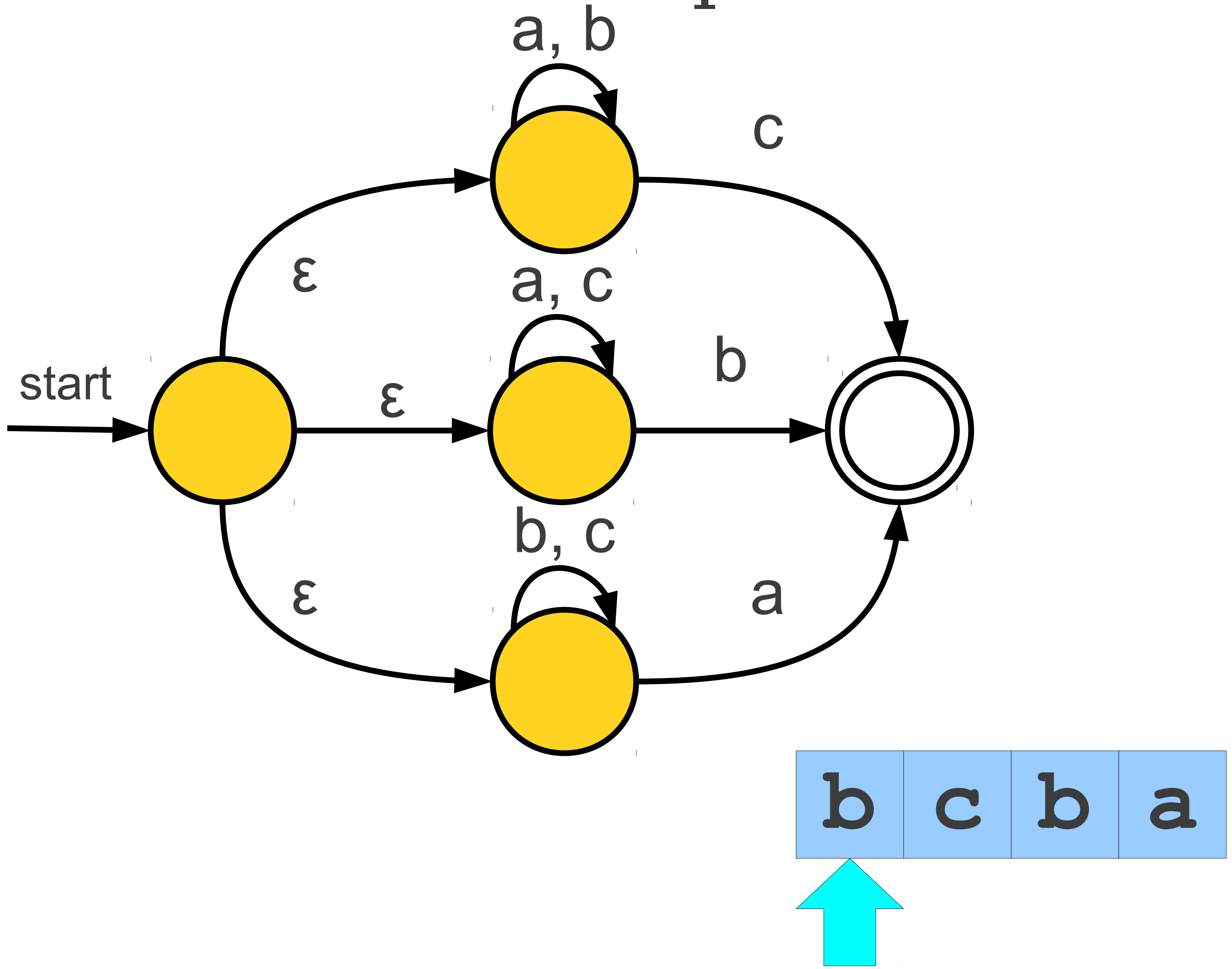
An Even More Complex Automaton



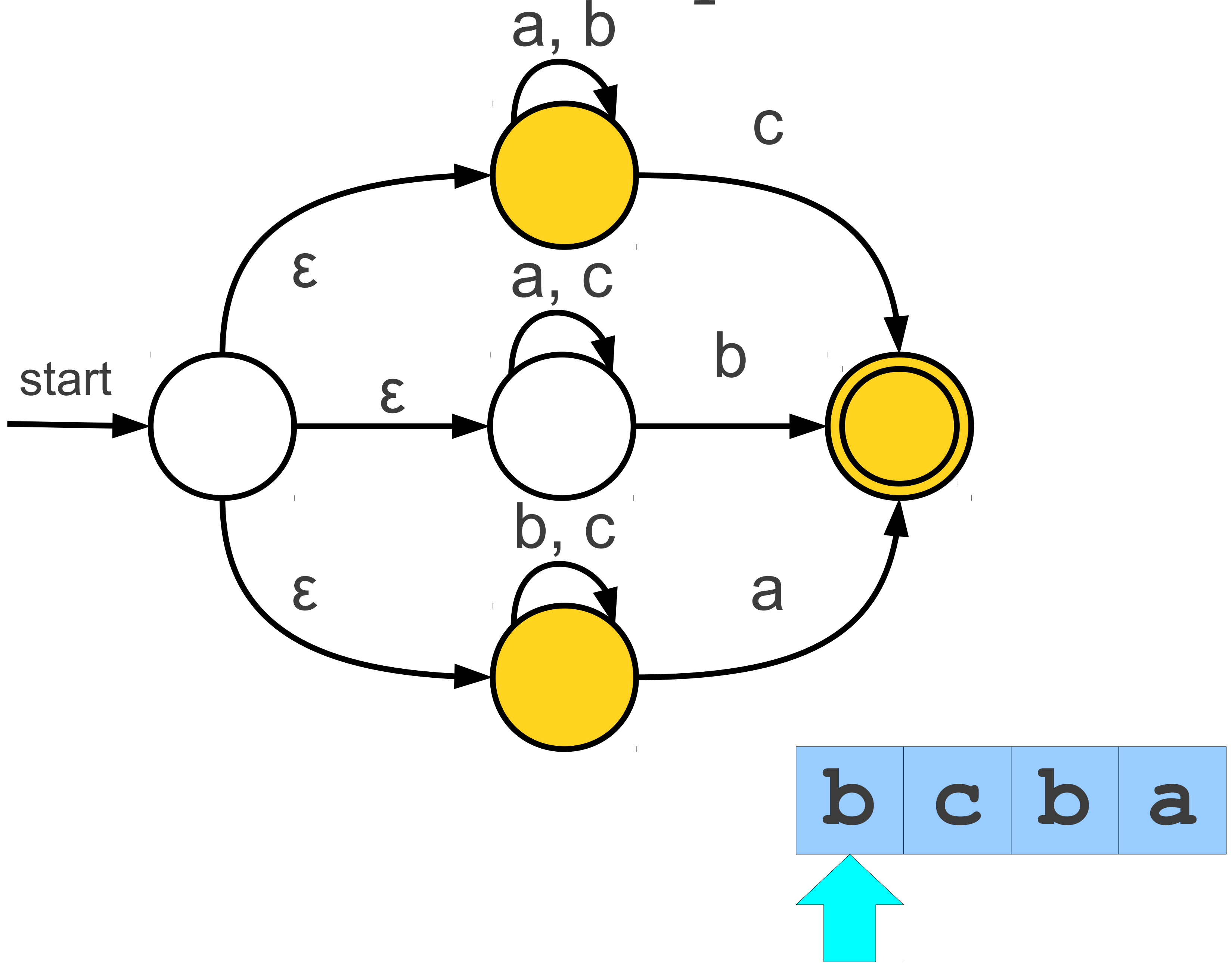
An Even More Complex Automaton



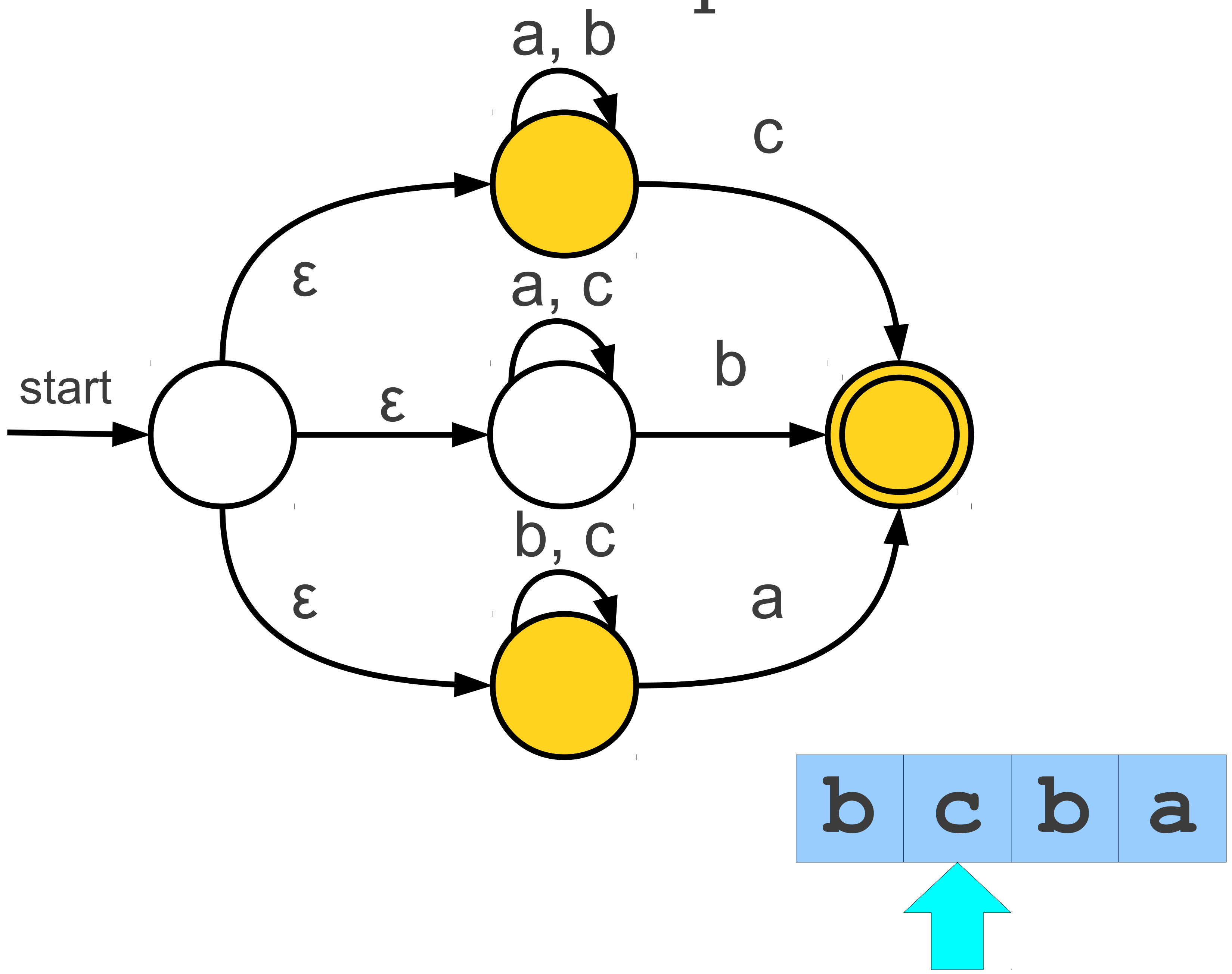
An Even More Complex Automaton



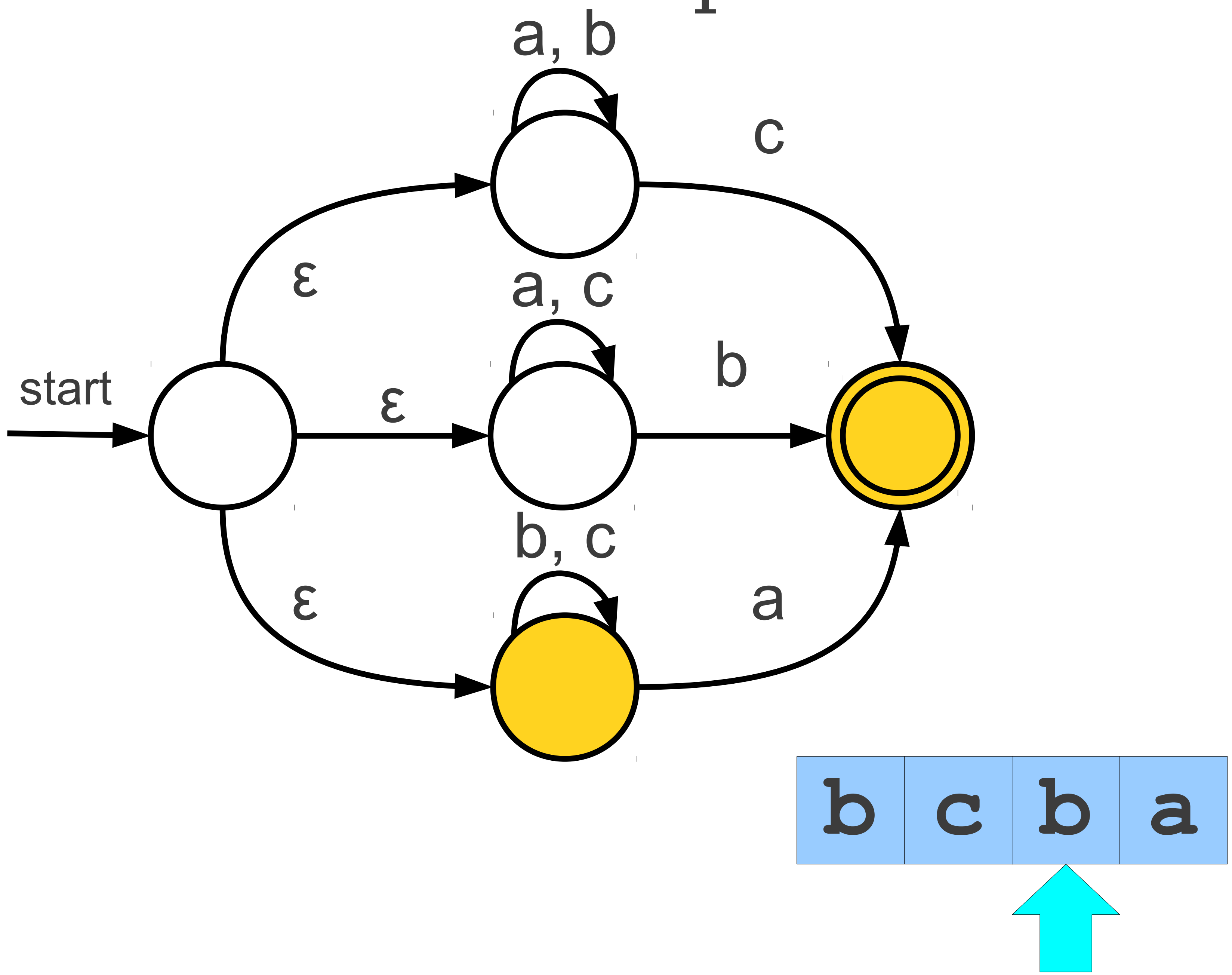
An Even More Complex Automaton



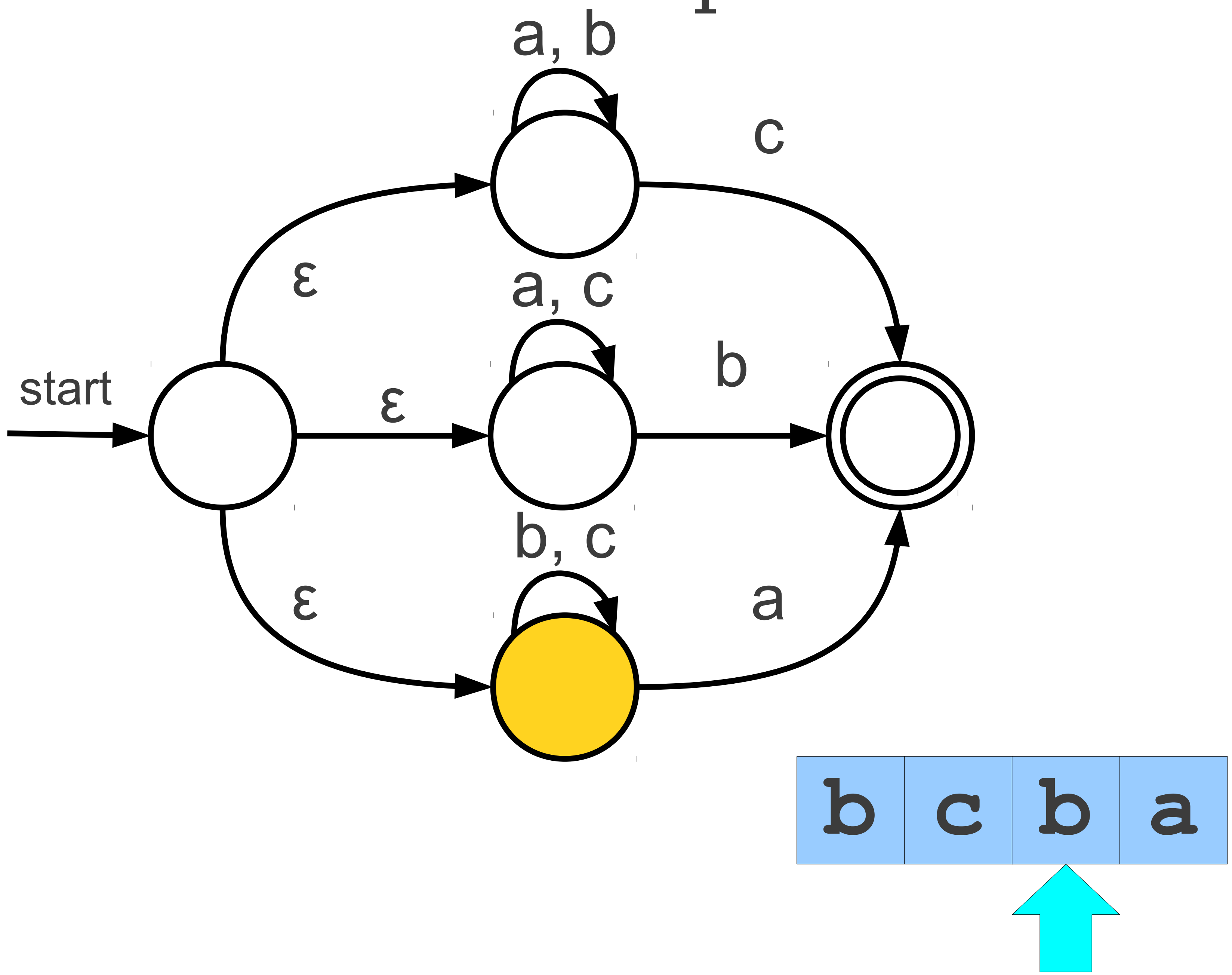
An Even More Complex Automaton



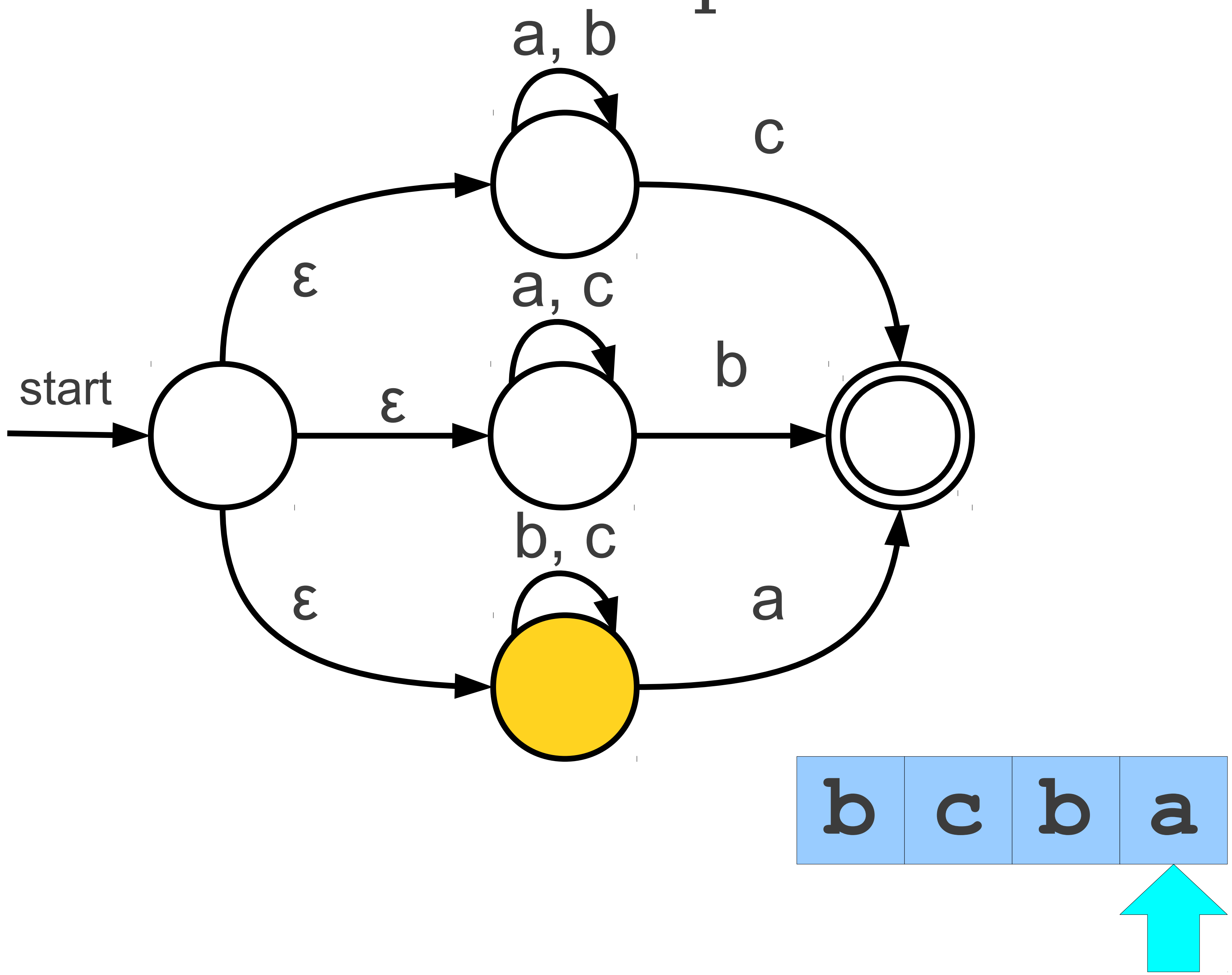
An Even More Complex Automaton



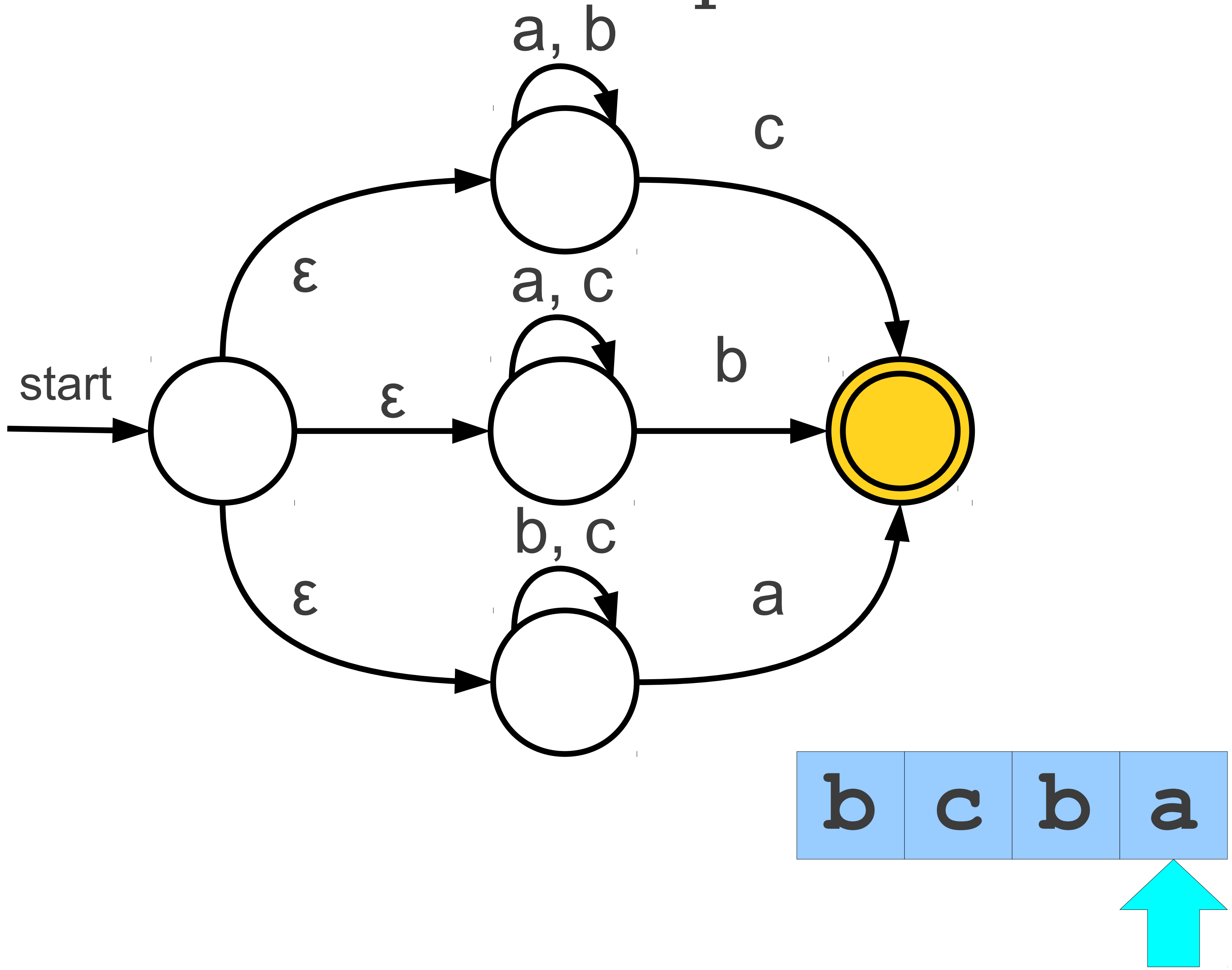
An Even More Complex Automaton



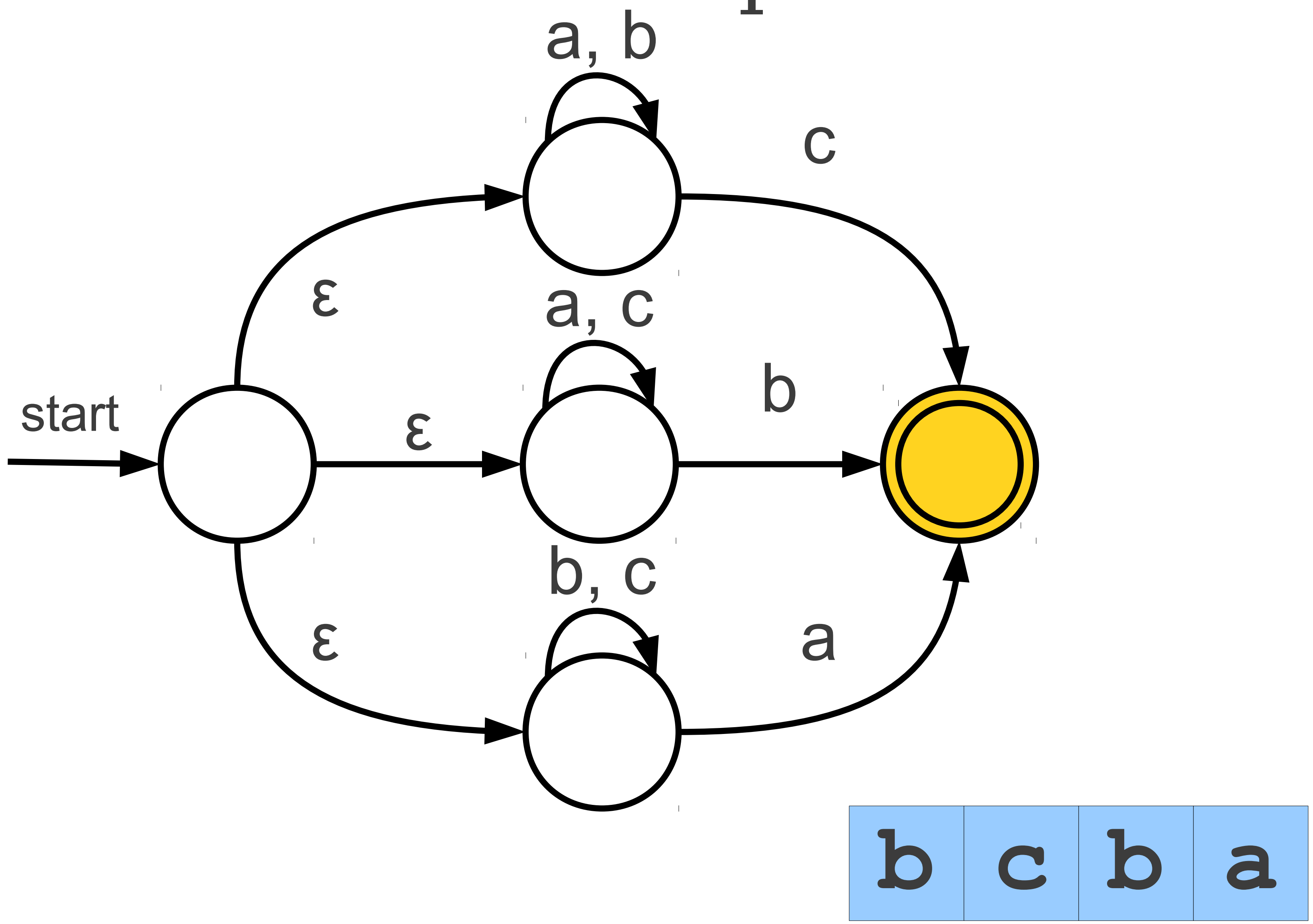
An Even More Complex Automaton



An Even More Complex Automaton



An Even More Complex Automaton



Lexer Generator

- Given regular expressions to describe the language (token types),
  - Step 1: Generates NFA that can recognize the regular language defined
    - existing algorithms
  - Step 2: Transforms NFA to DFA
    - existing algorithms
  - Tools: **lex, flex**

Challenges for Lexical Analyzer

- How do we determine which lexemes are associated with each token?
    - Regular expression to describe token type
- 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_]*
```

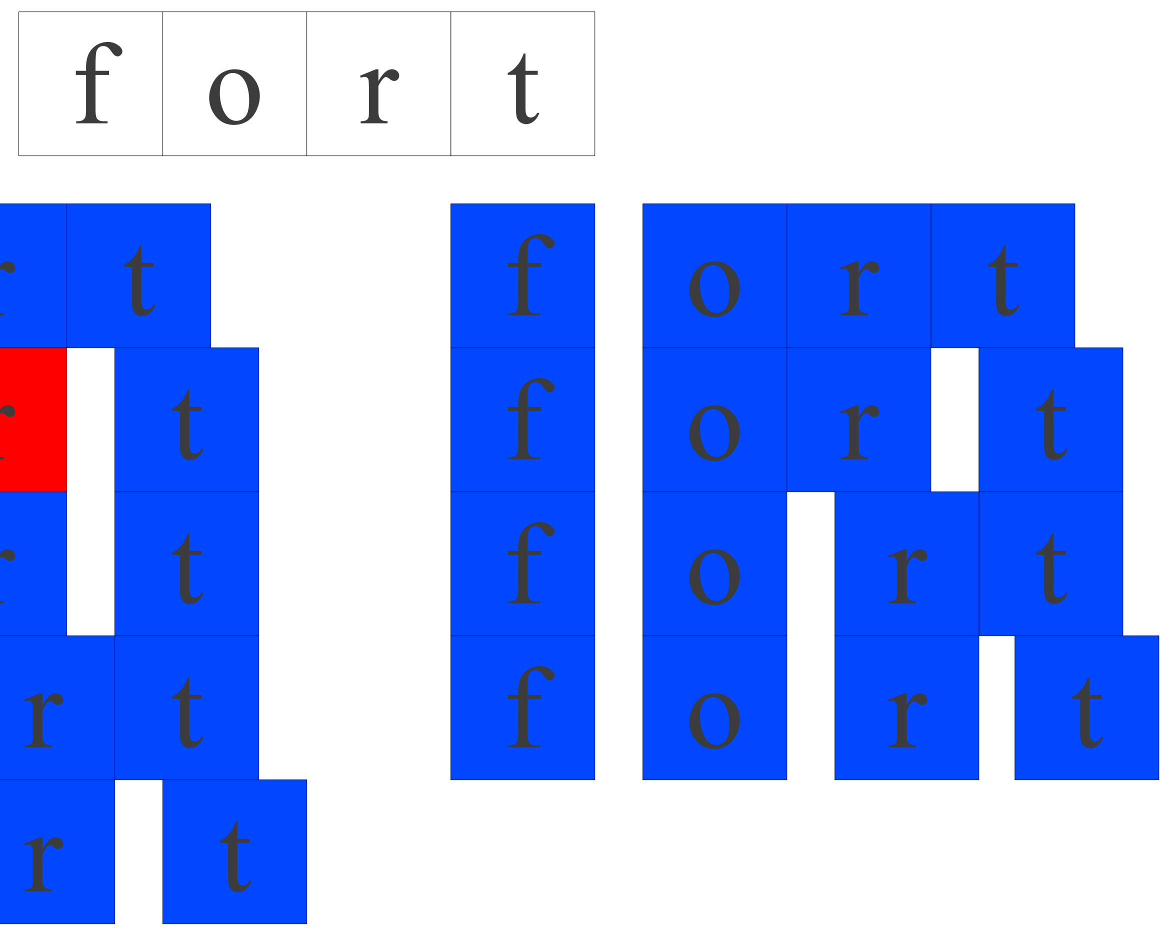
Lexing Ambiguities

```
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_]*
```

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?

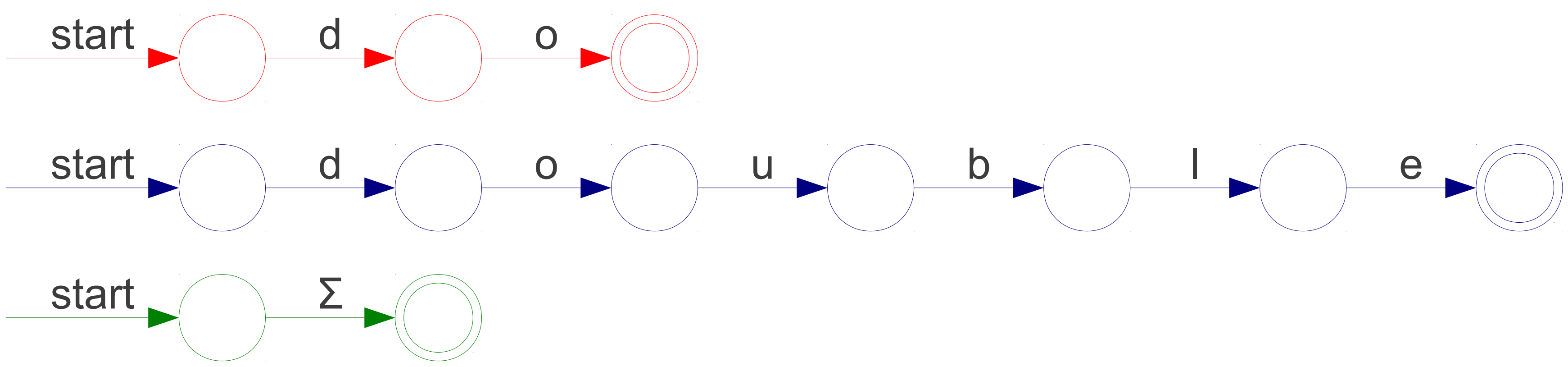
- Example

Implementing Maximal Munch

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

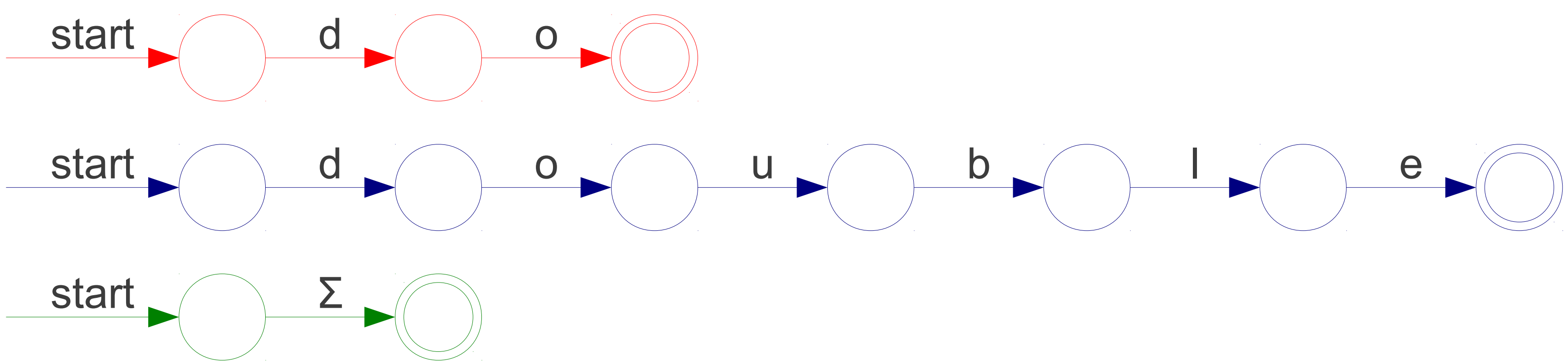
Implementing Maximal Munch

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



Implementing Maximal Munch

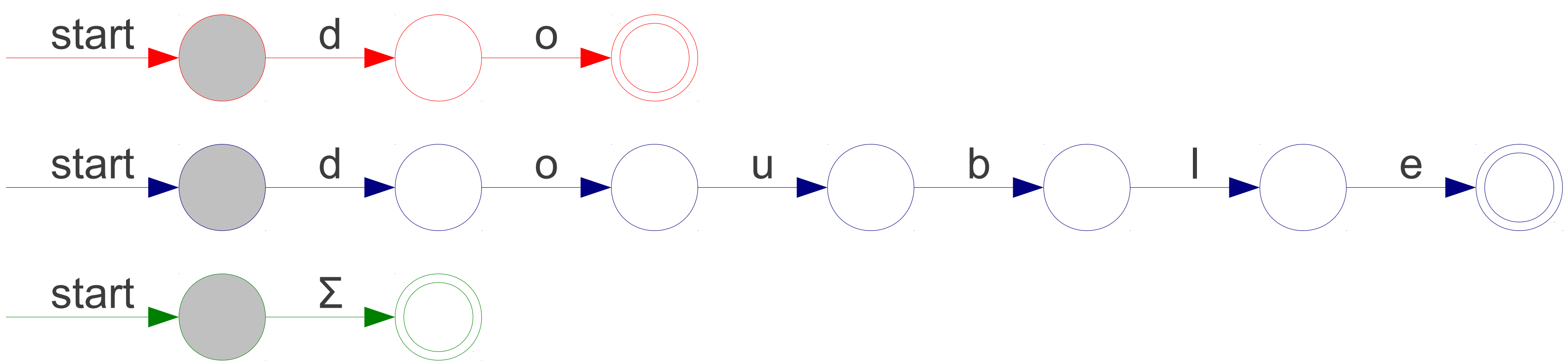
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

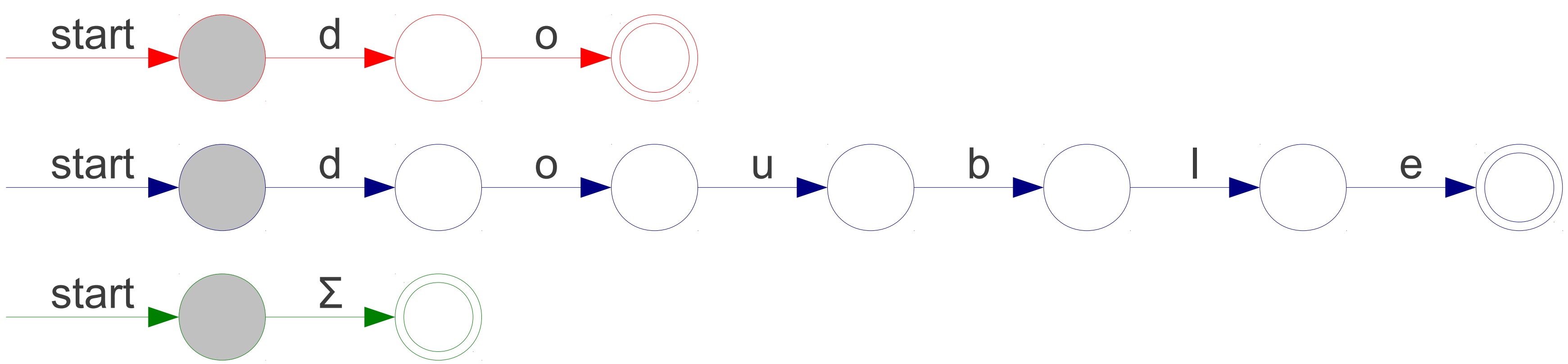
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```

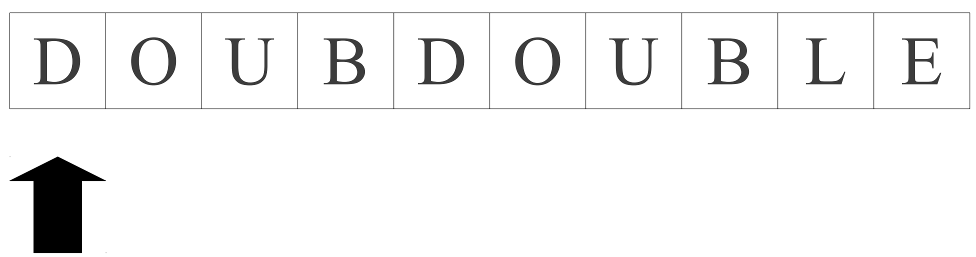


D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

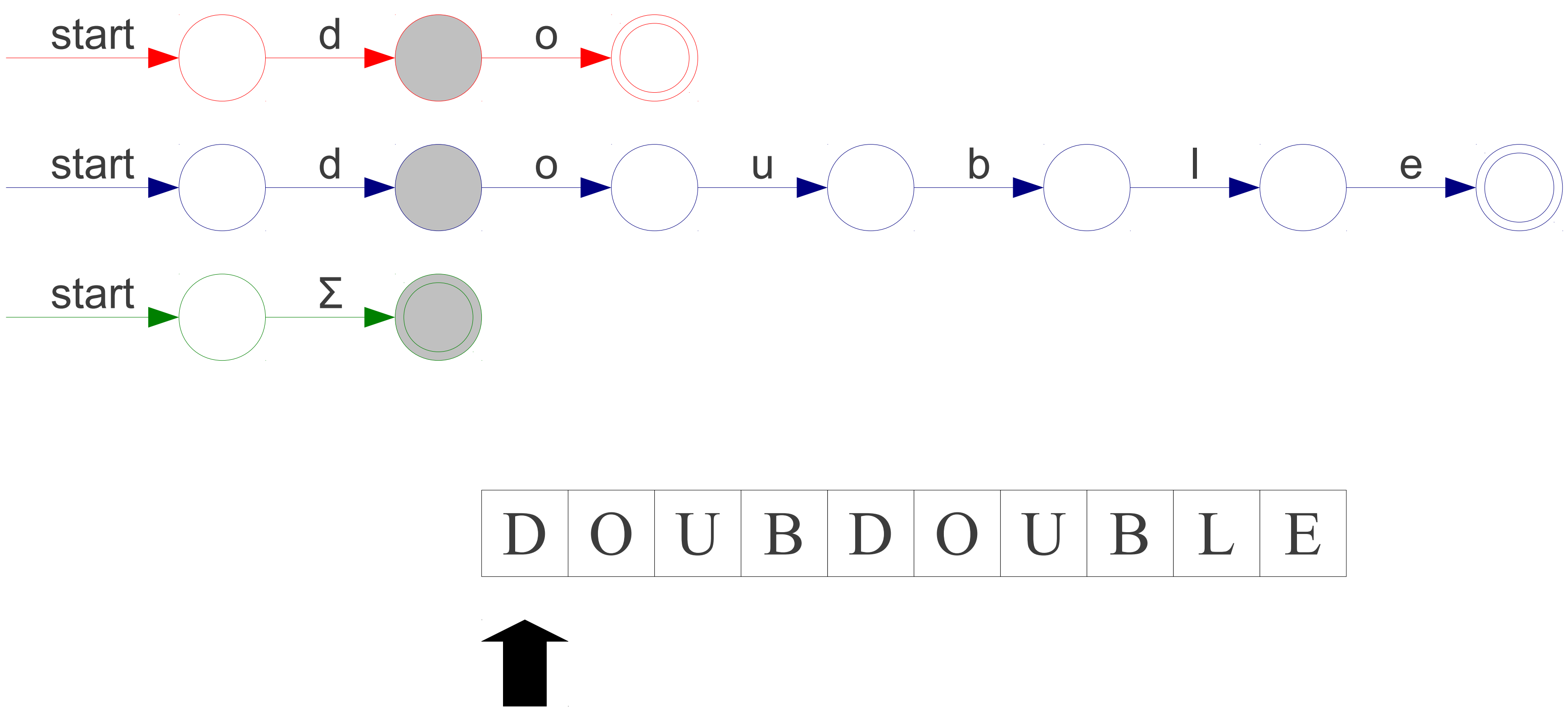
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





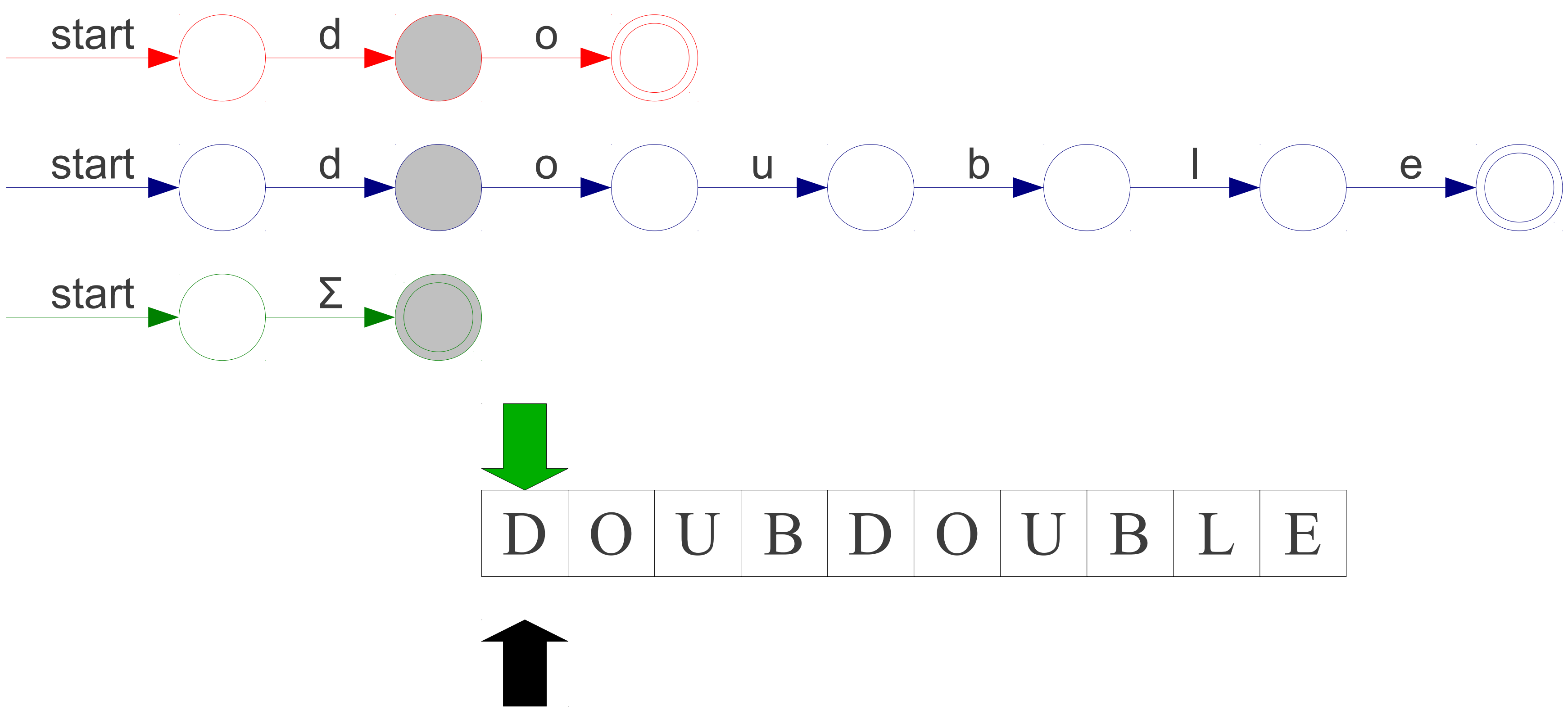
Implementing Maximal Munch

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



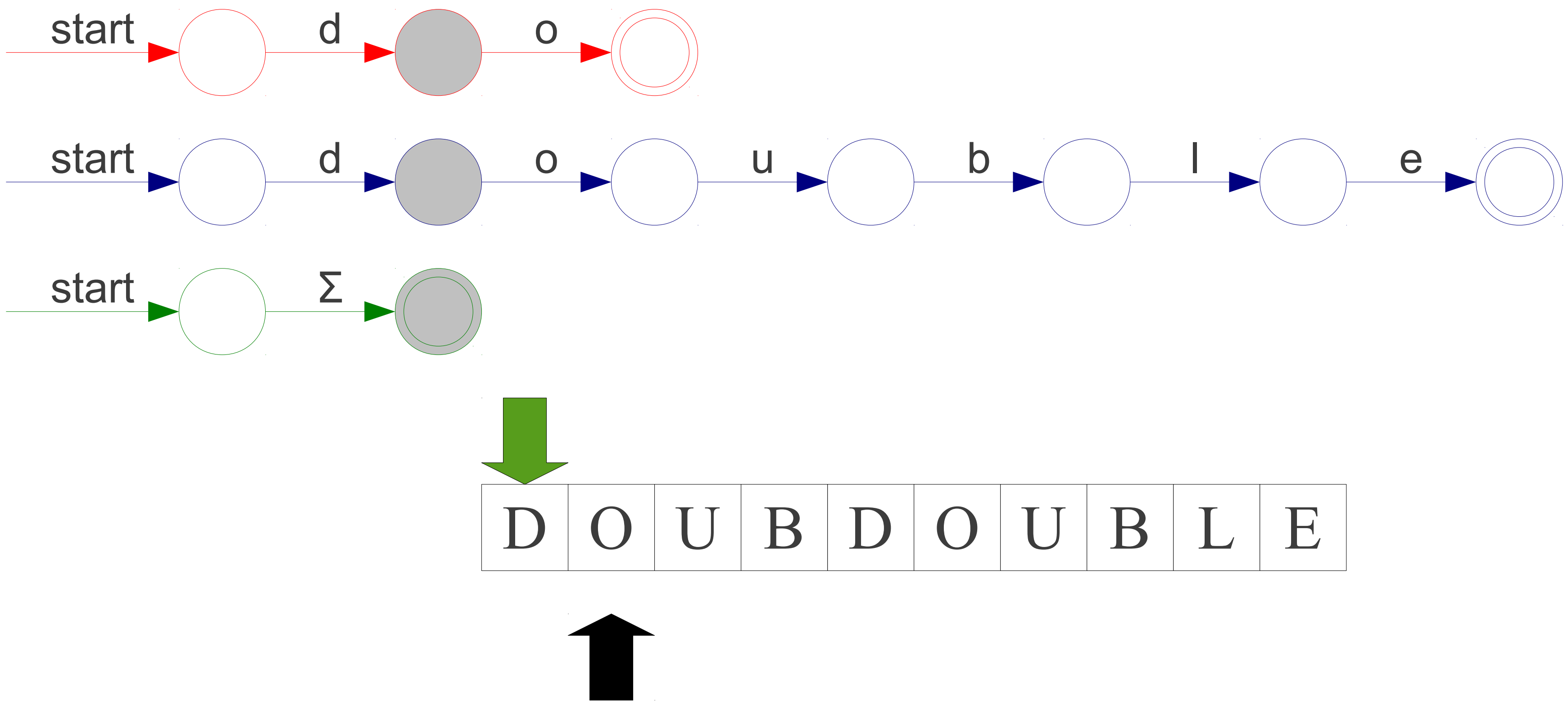
Implementing Maximal Munch

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



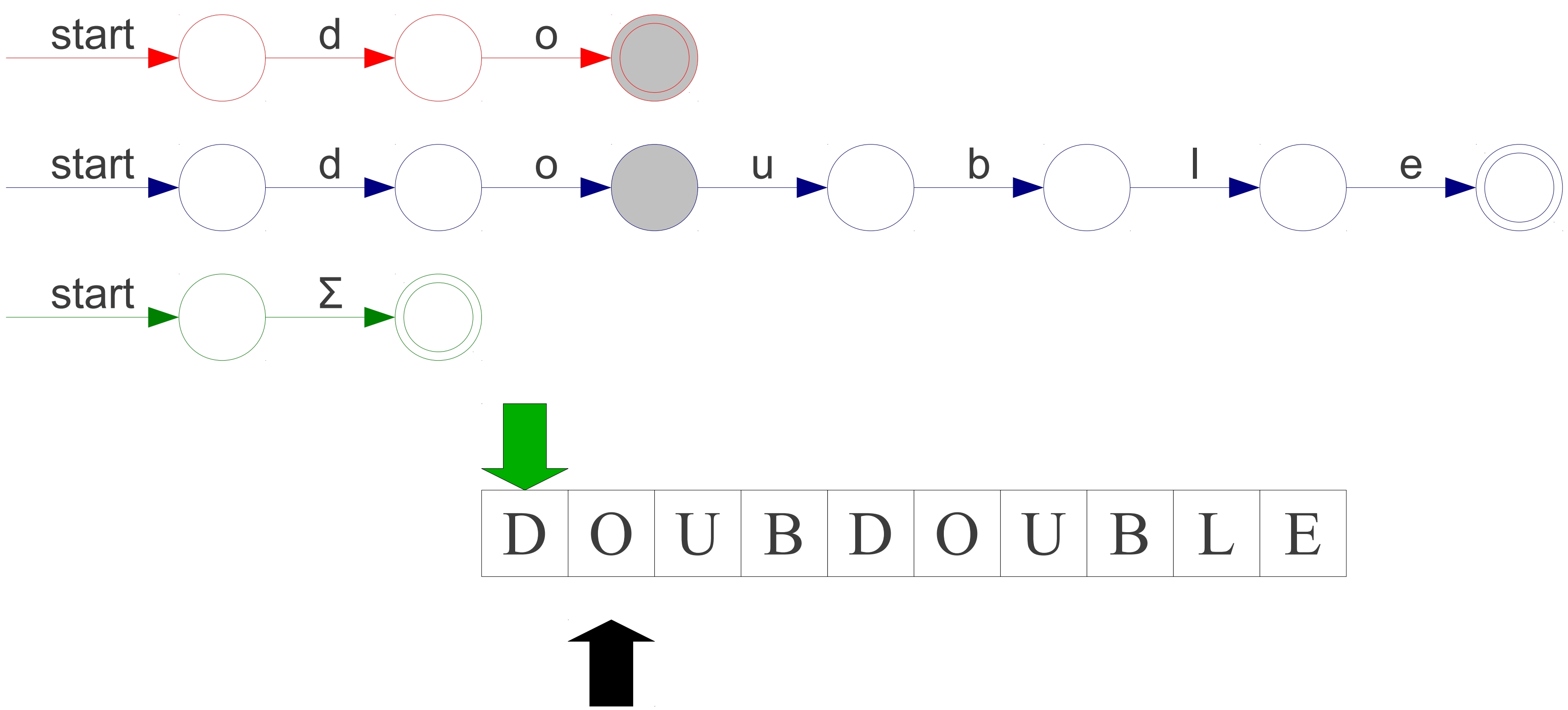
Implementing Maximal Munch

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



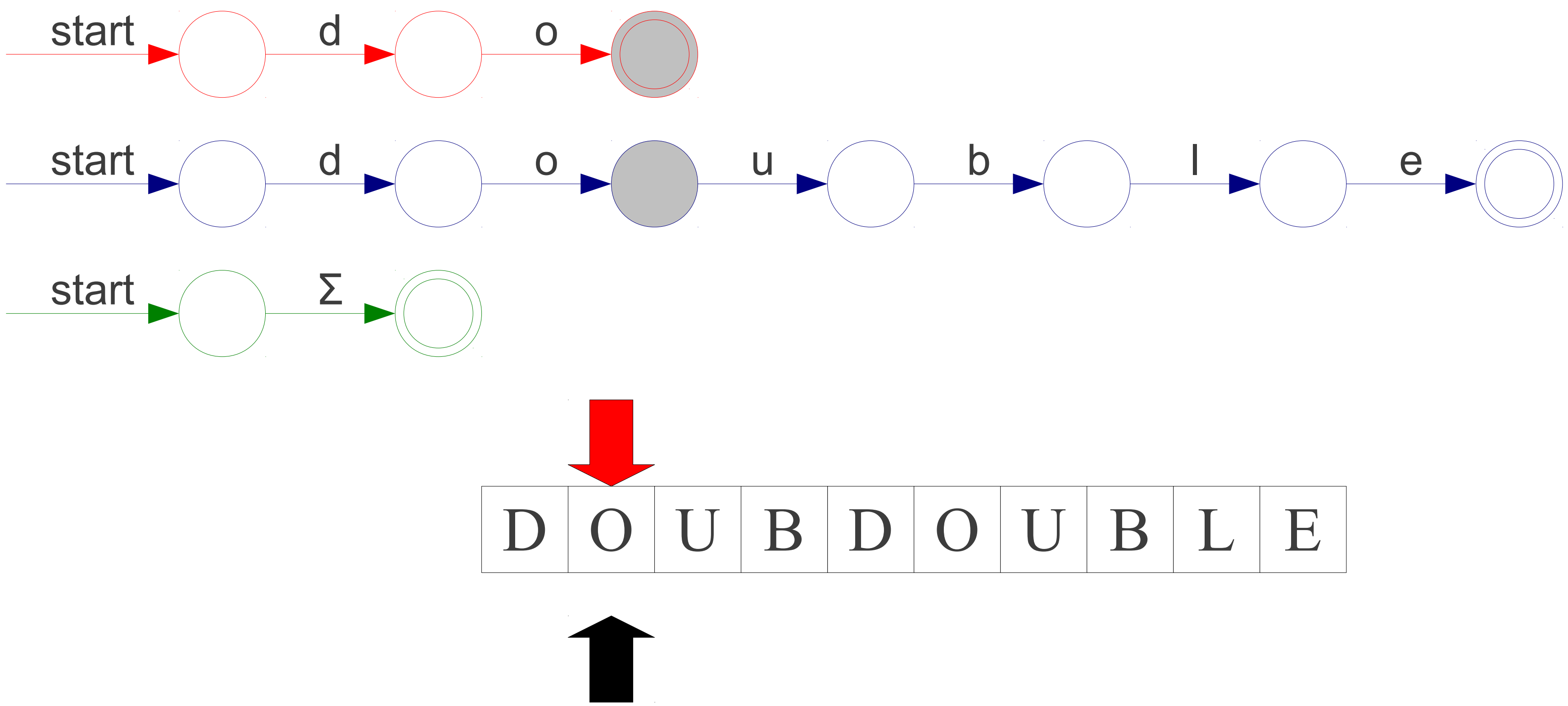
Implementing Maximal Munch

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



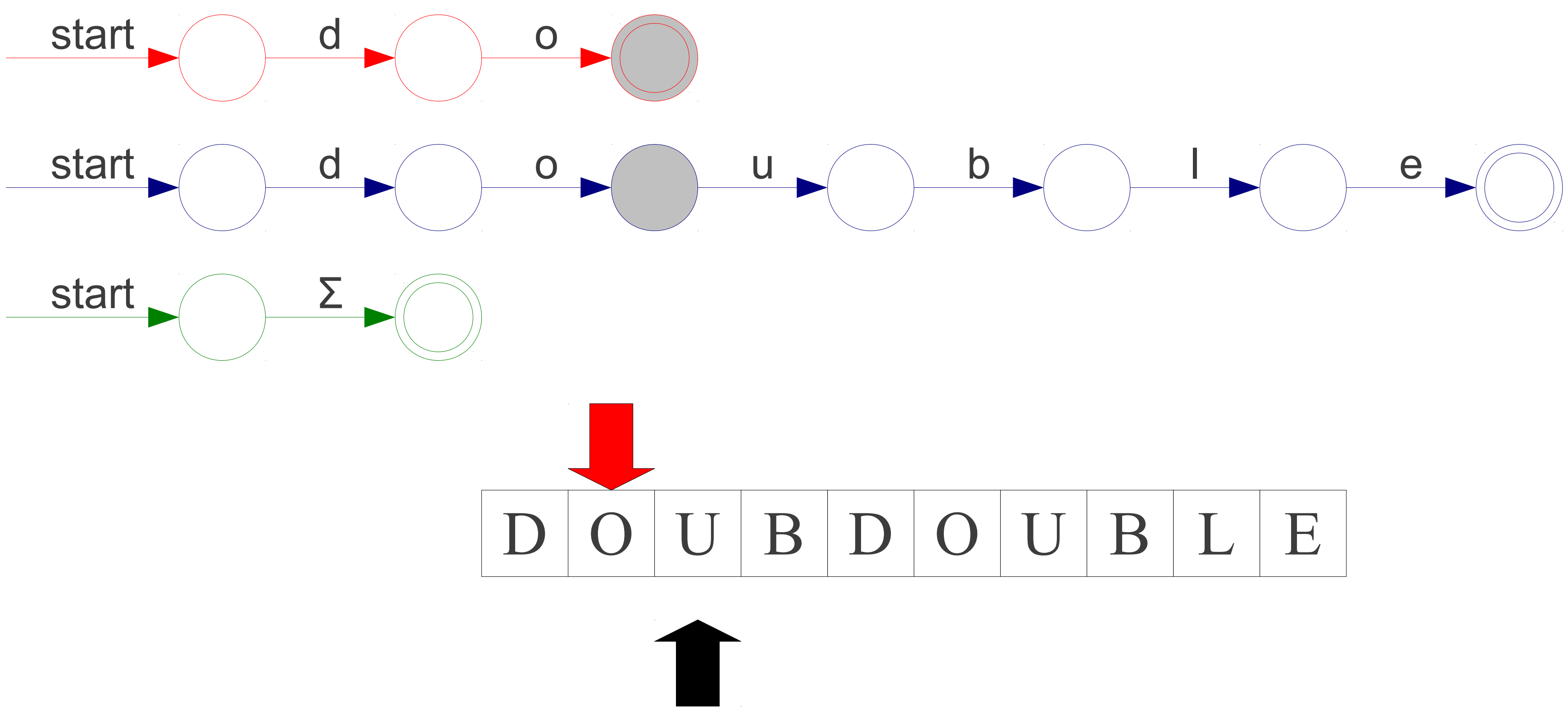
Implementing Maximal Munch

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



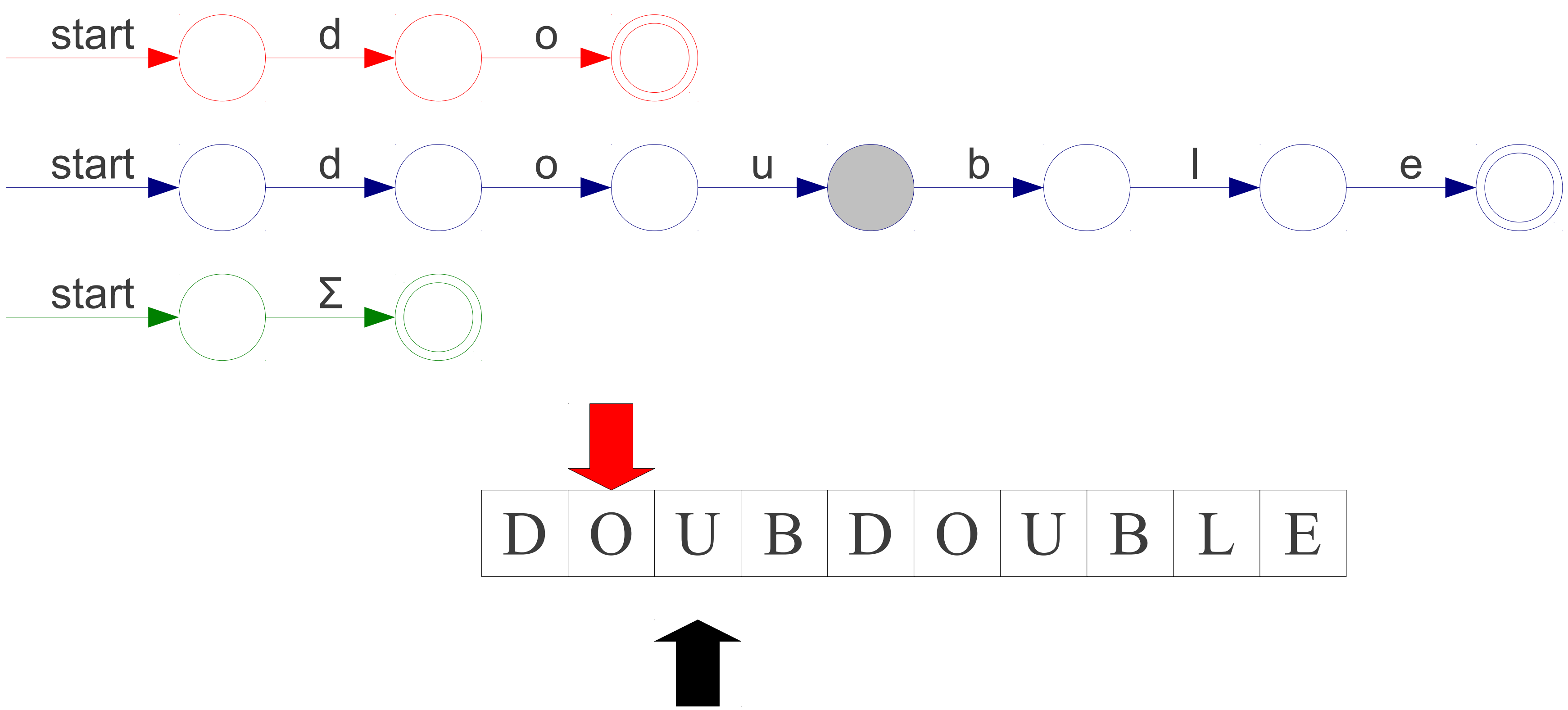
Implementing Maximal Munch

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



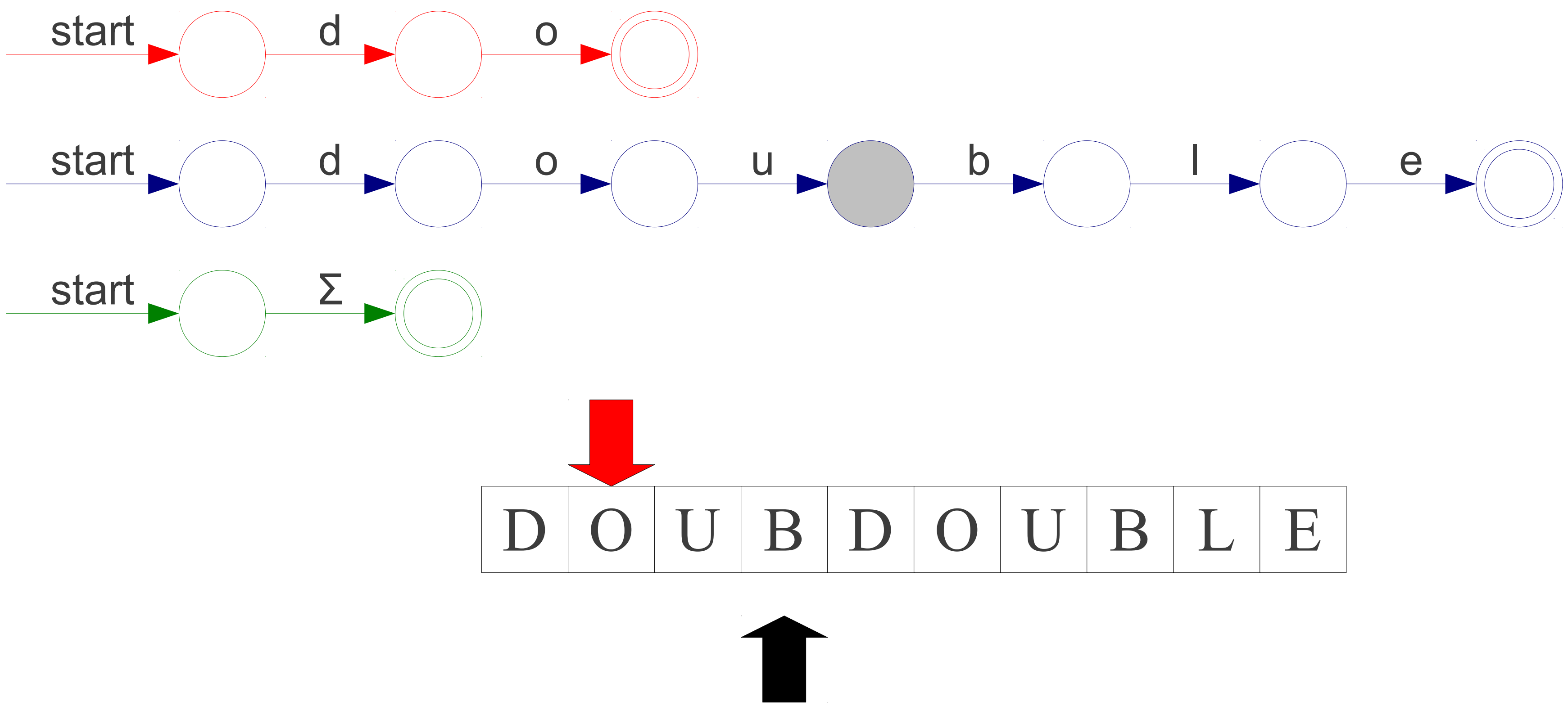
Implementing Maximal Munch

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



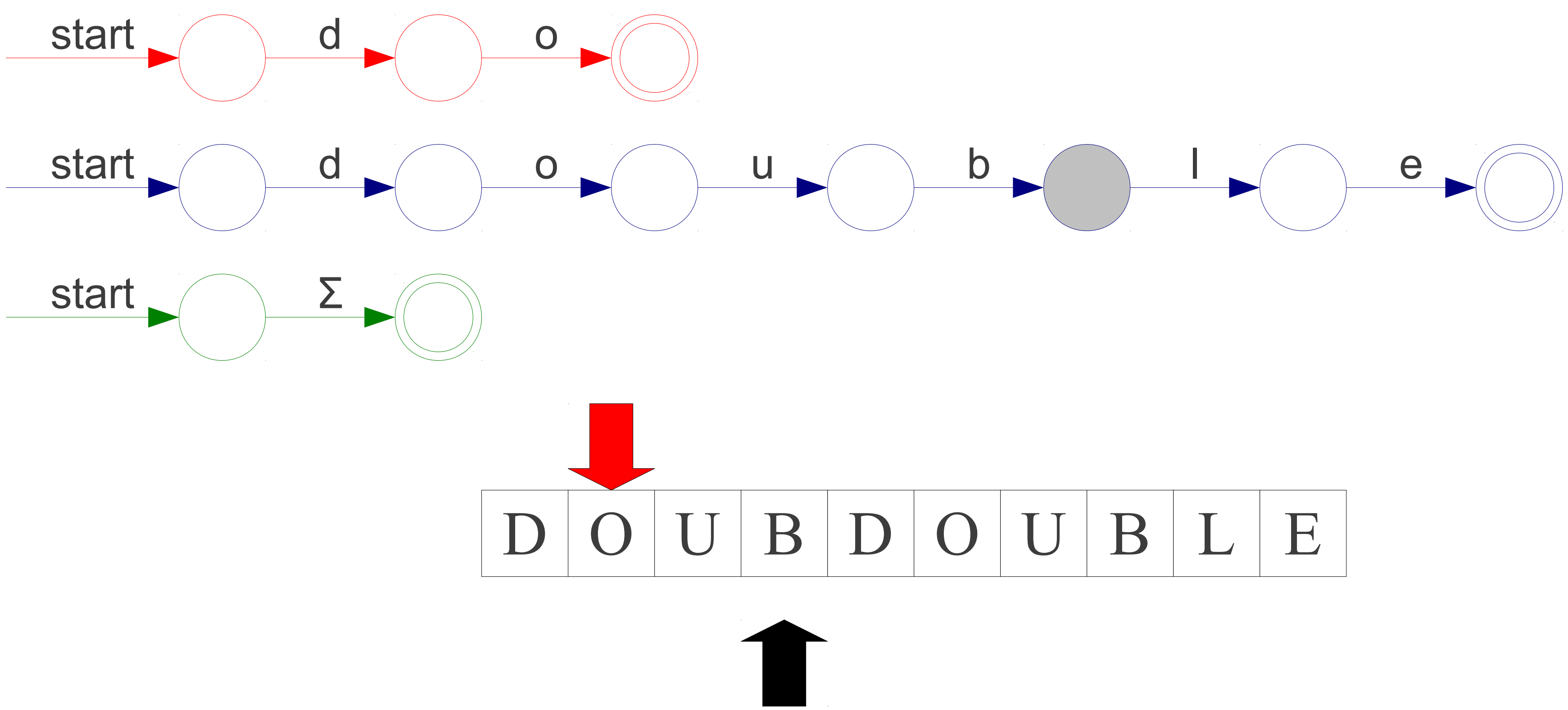
Implementing Maximal Munch

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



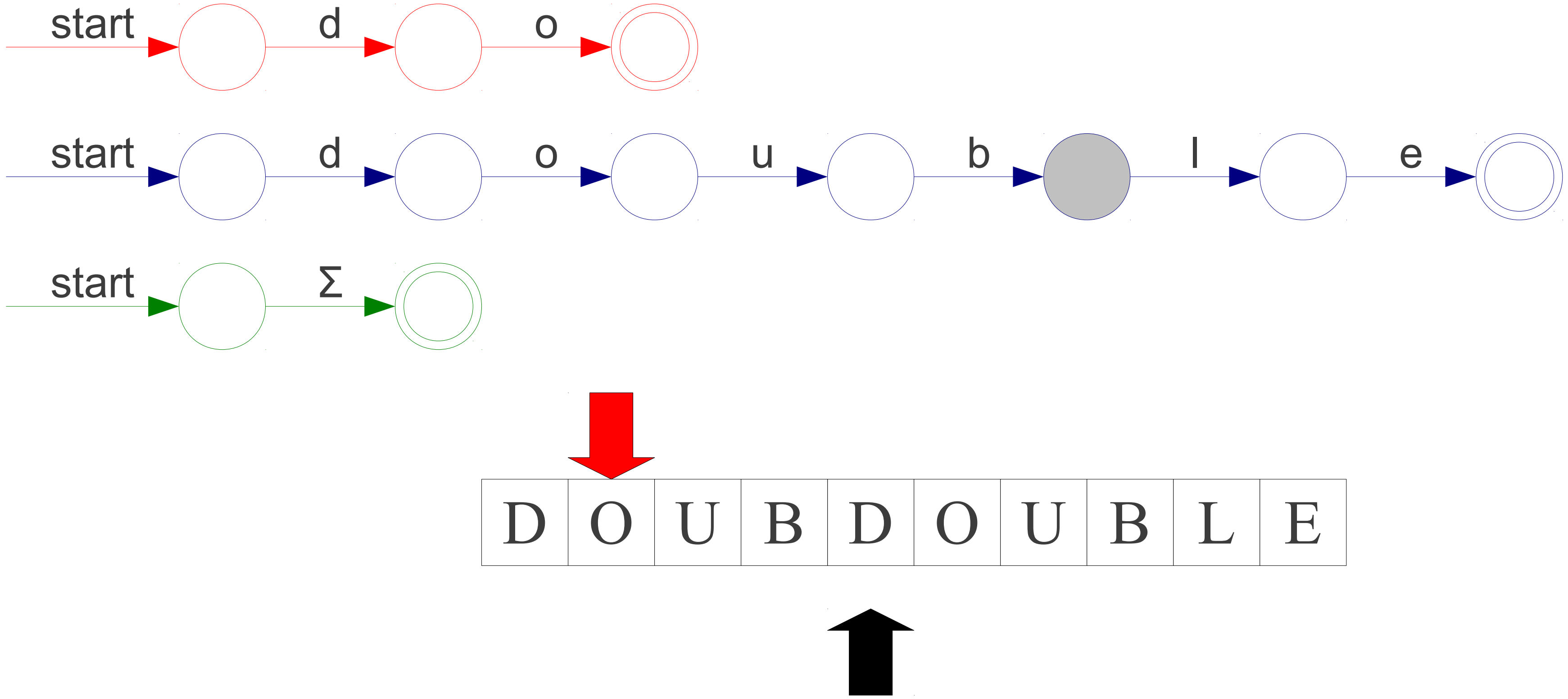
Implementing Maximal Munch

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



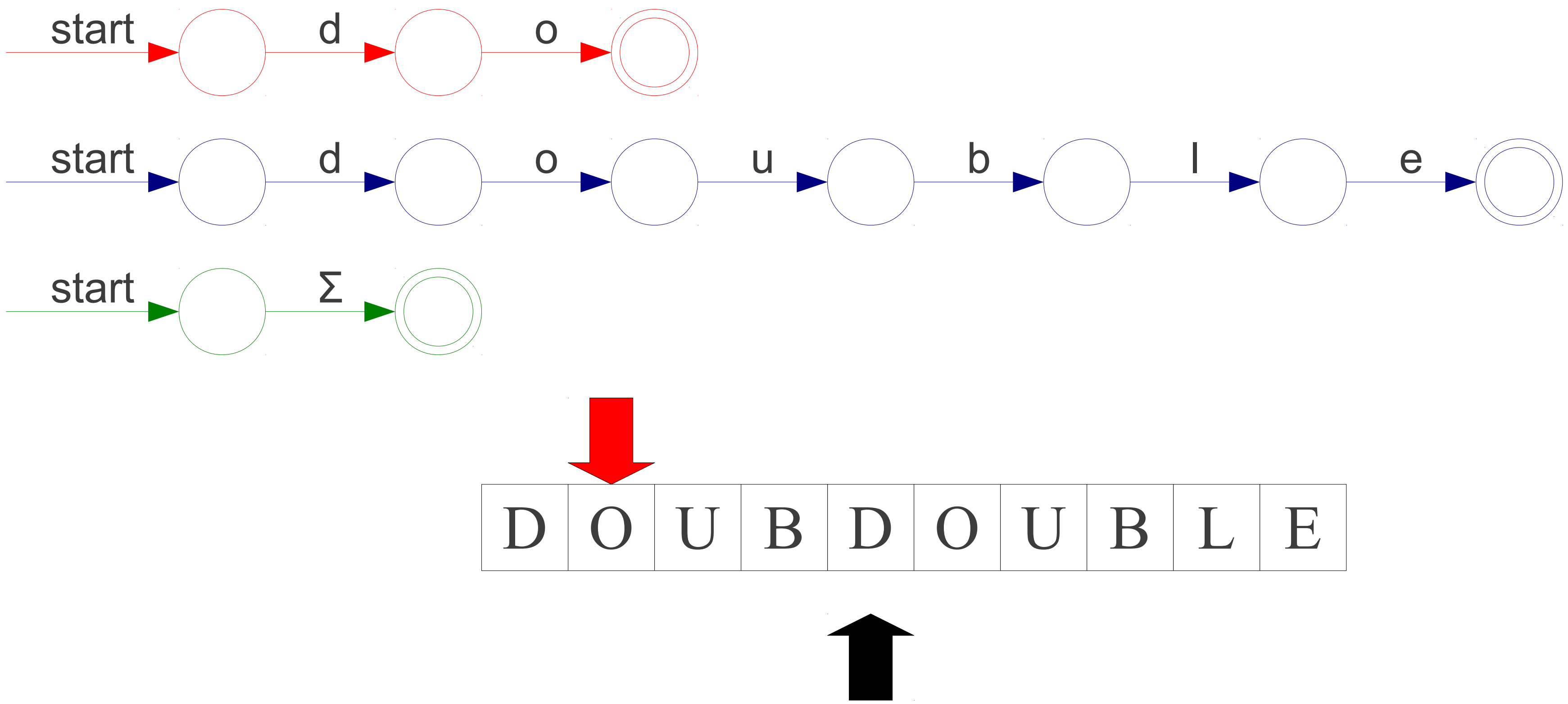
Implementing Maximal Munch

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



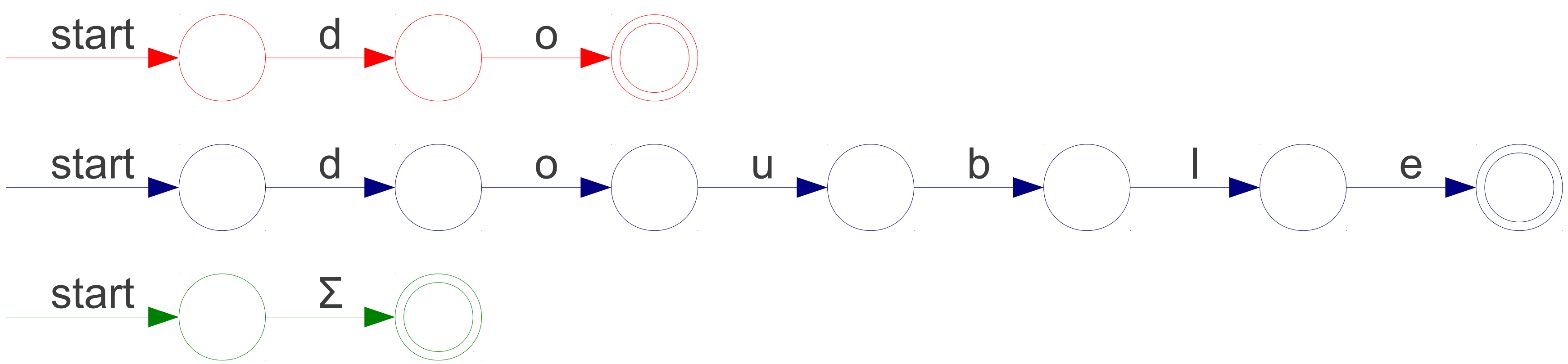
Implementing Maximal Munch

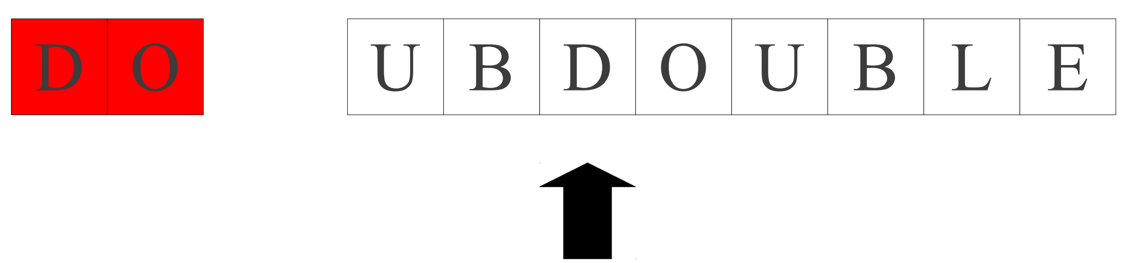
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

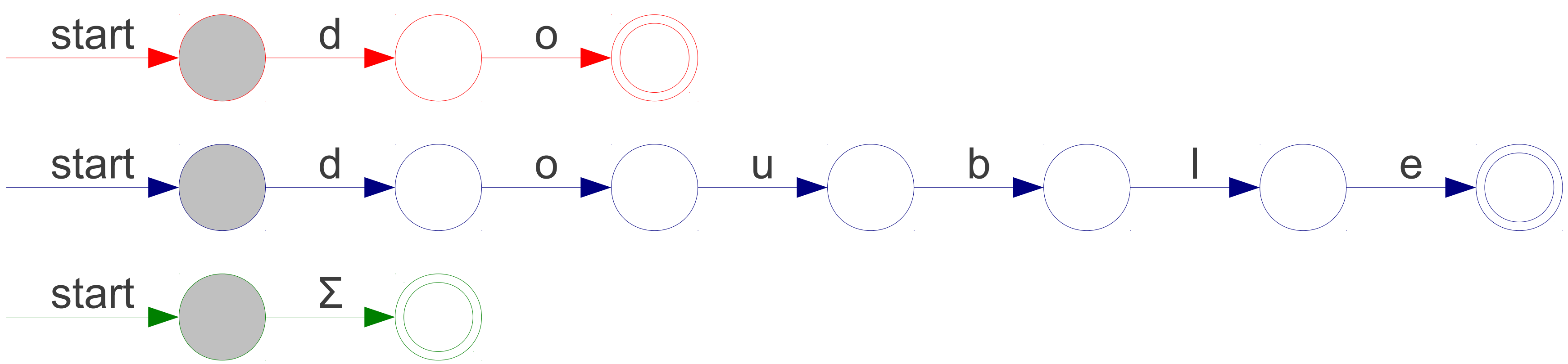
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

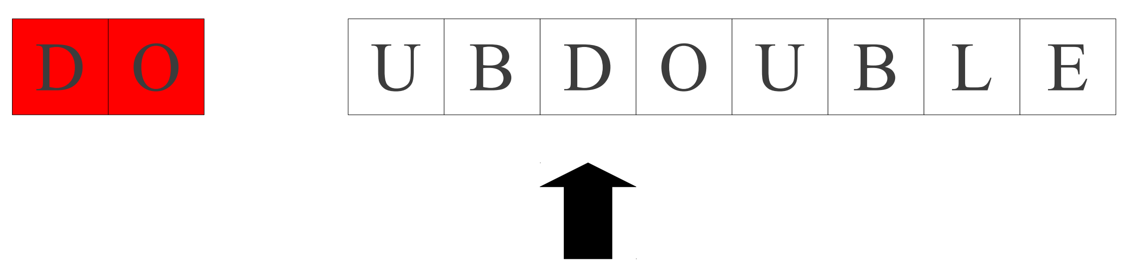




Implementing Maximal Munch

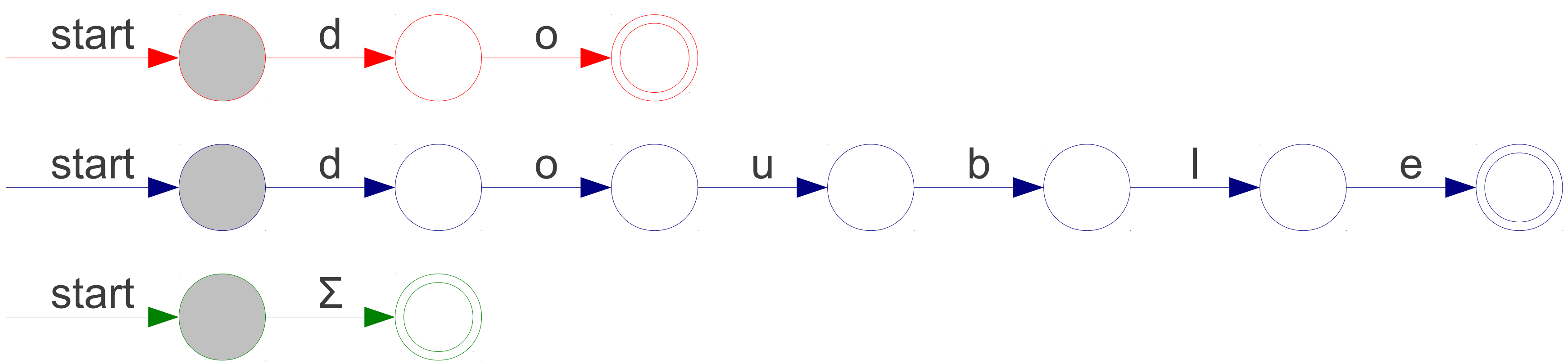
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```

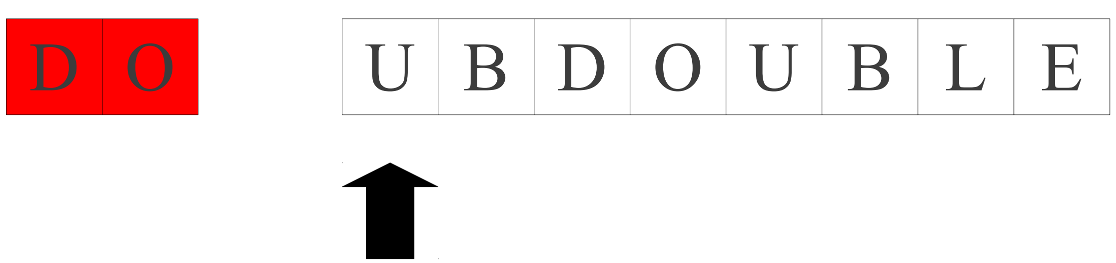




Implementing Maximal Munch

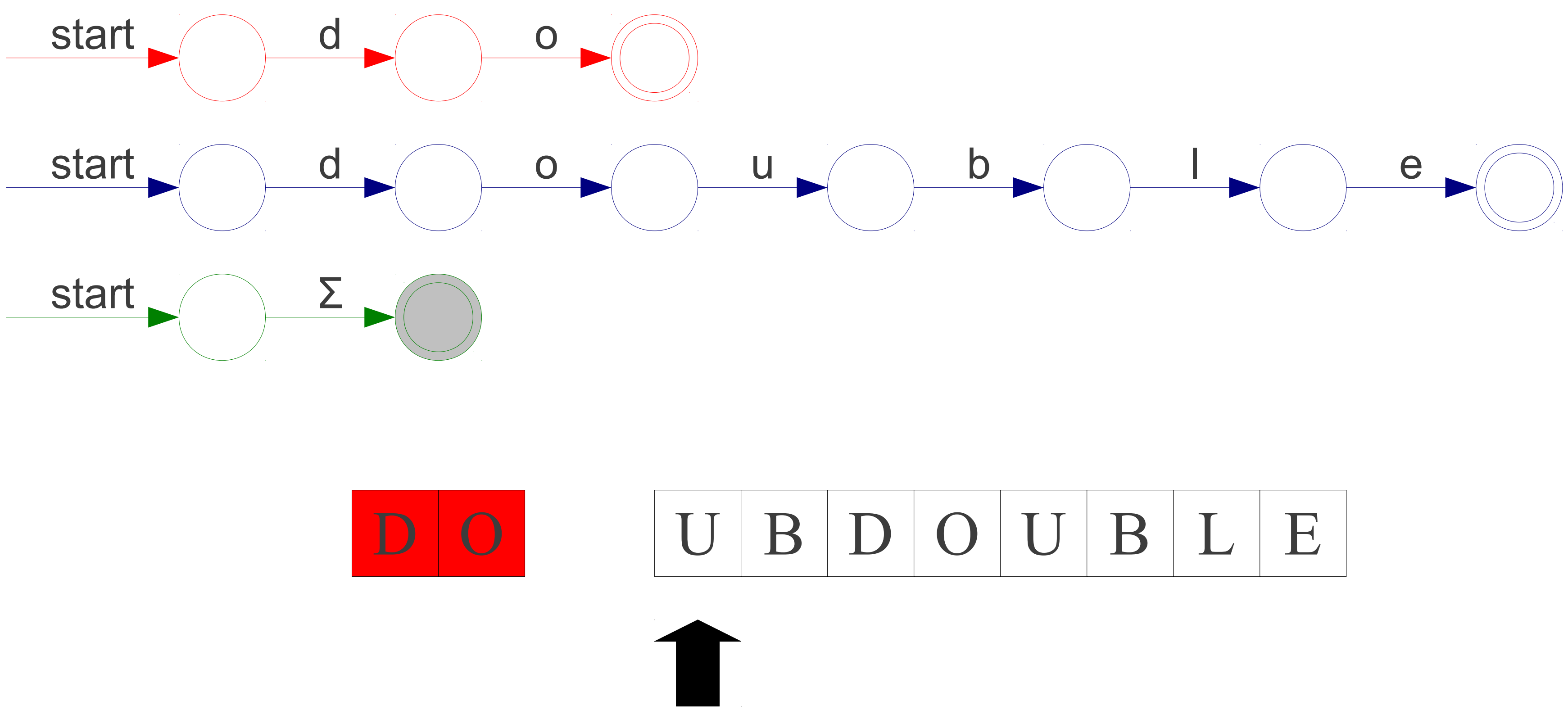
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





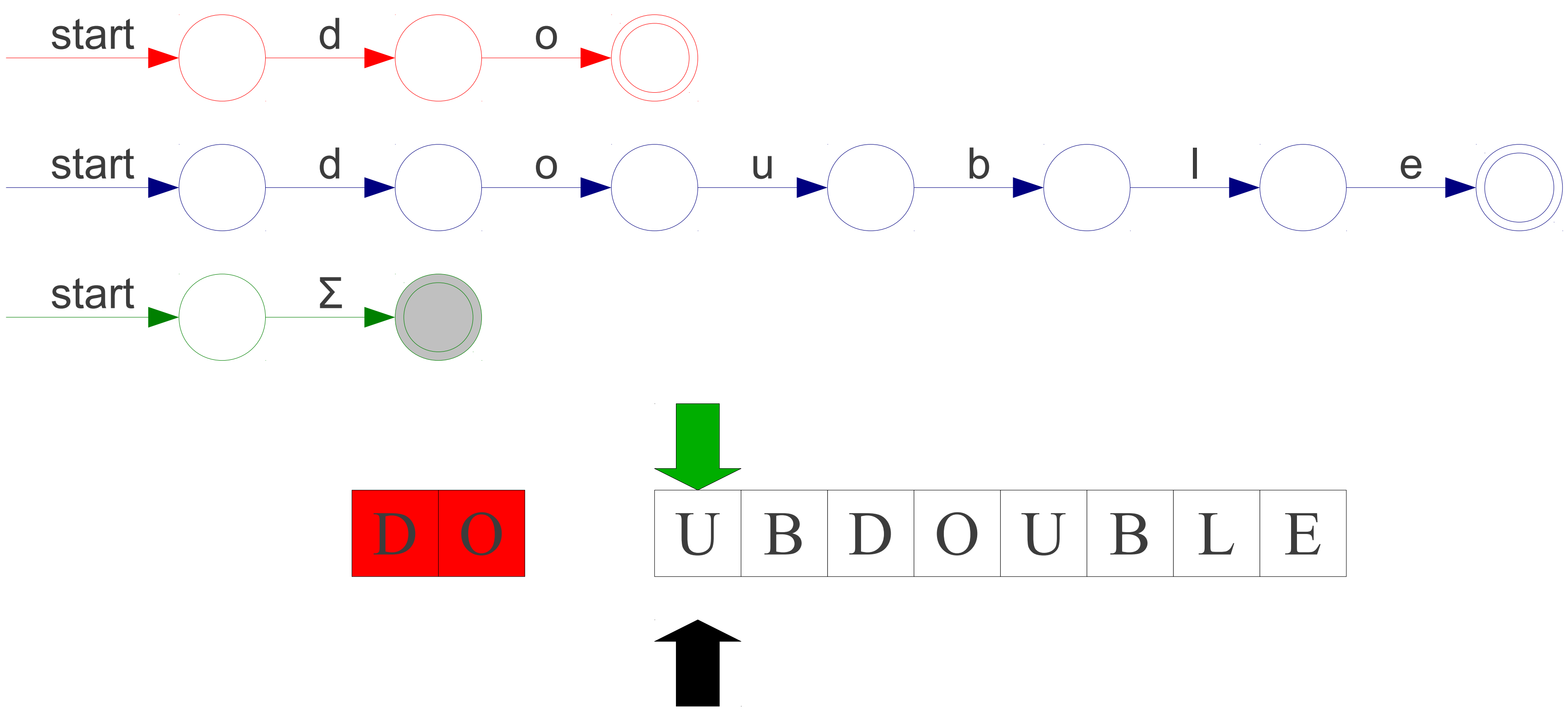
Implementing Maximal Munch

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



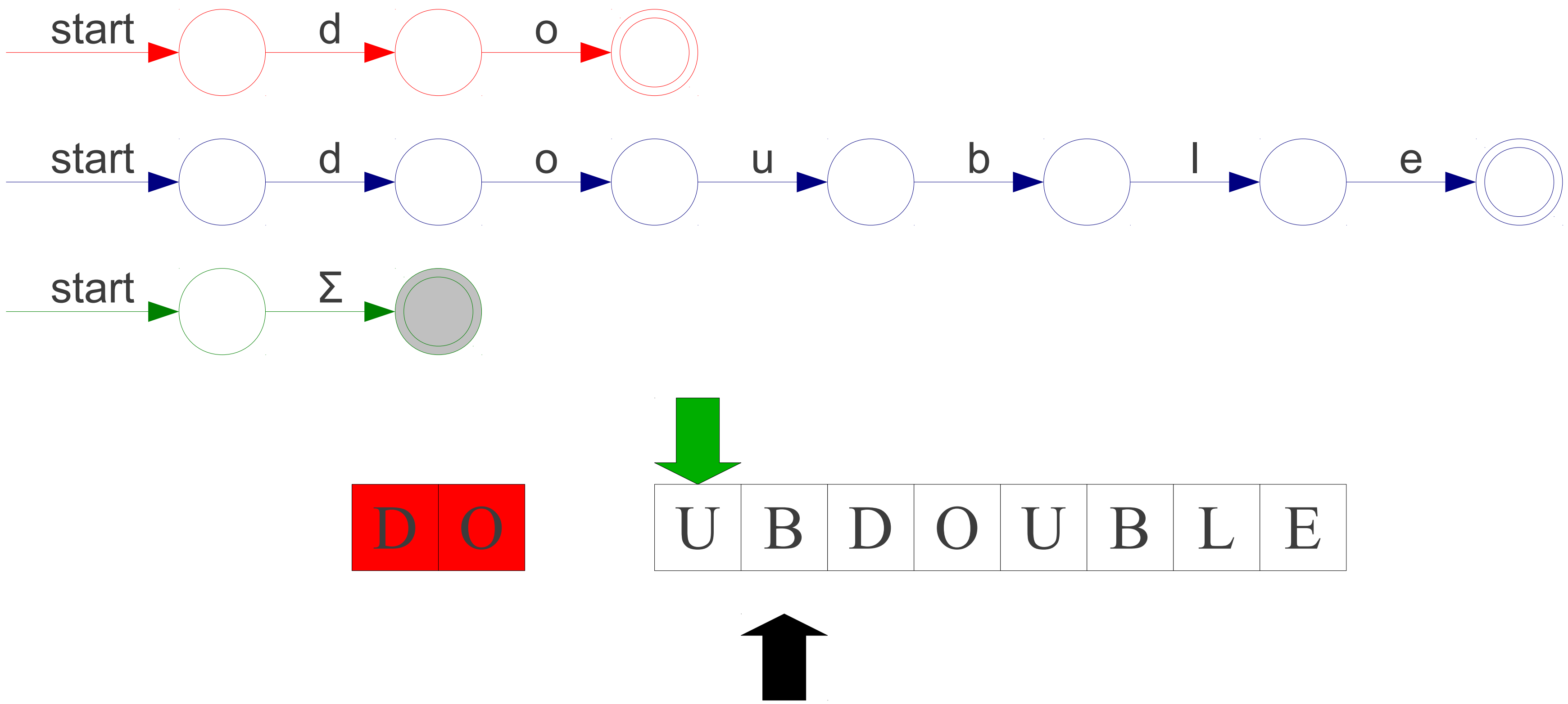
Implementing Maximal Munch

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



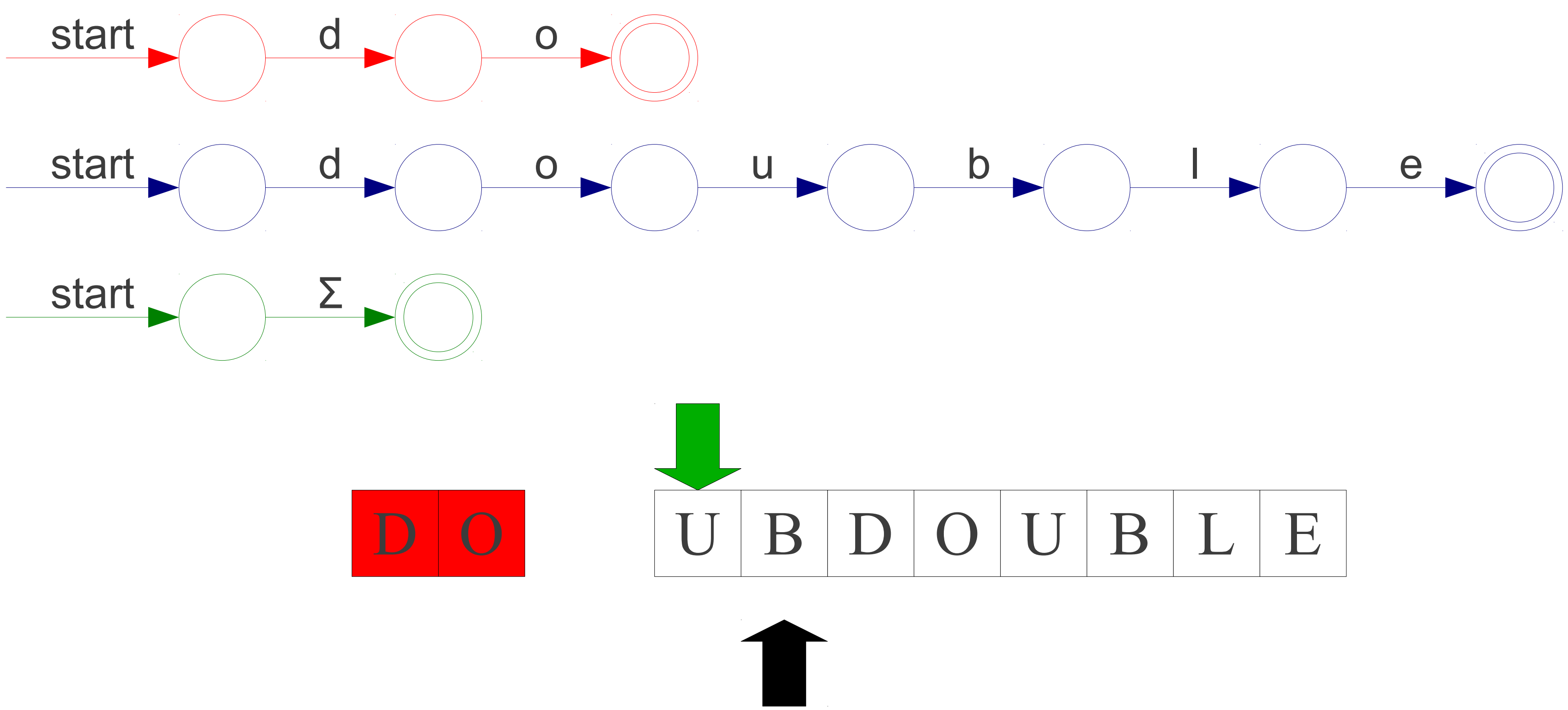
Implementing Maximal Munch

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



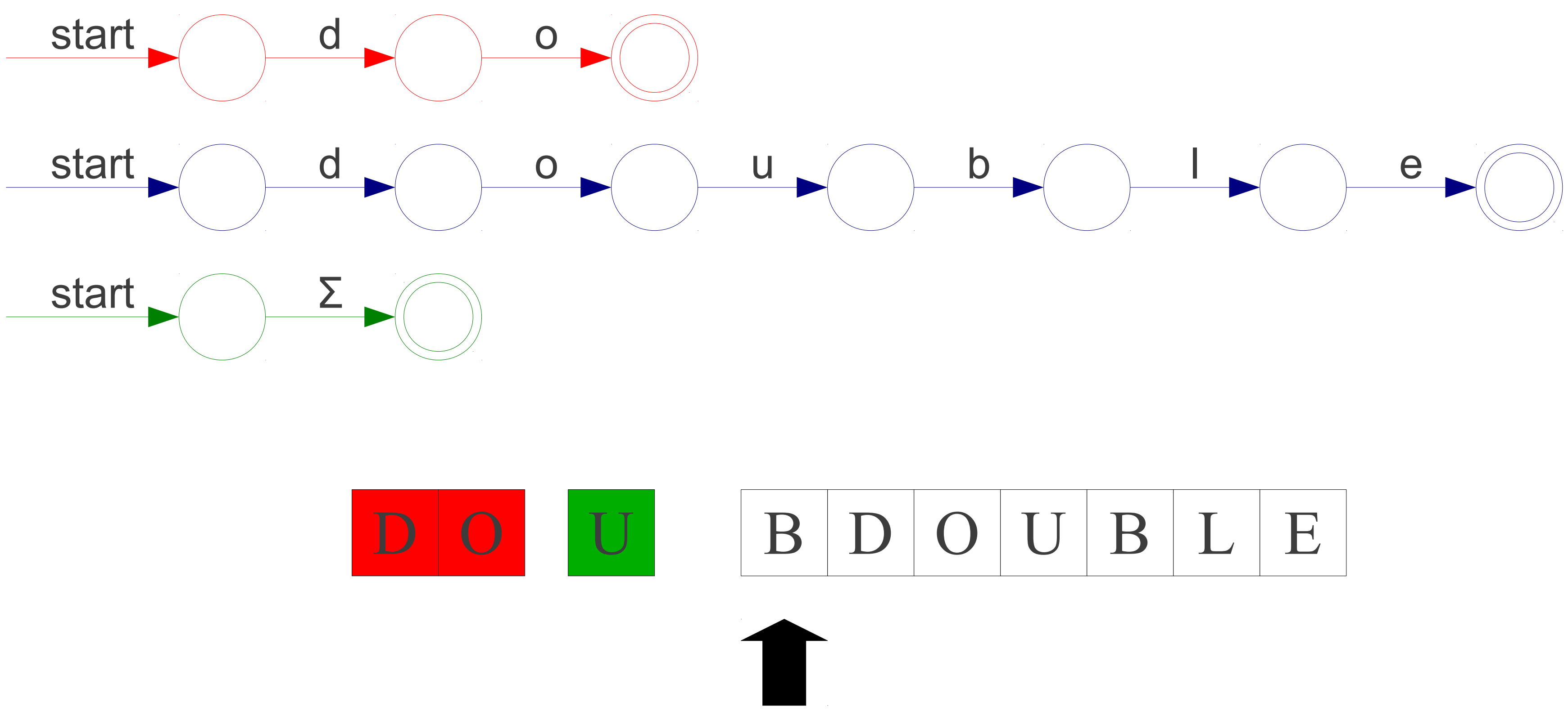
Implementing Maximal Munch

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



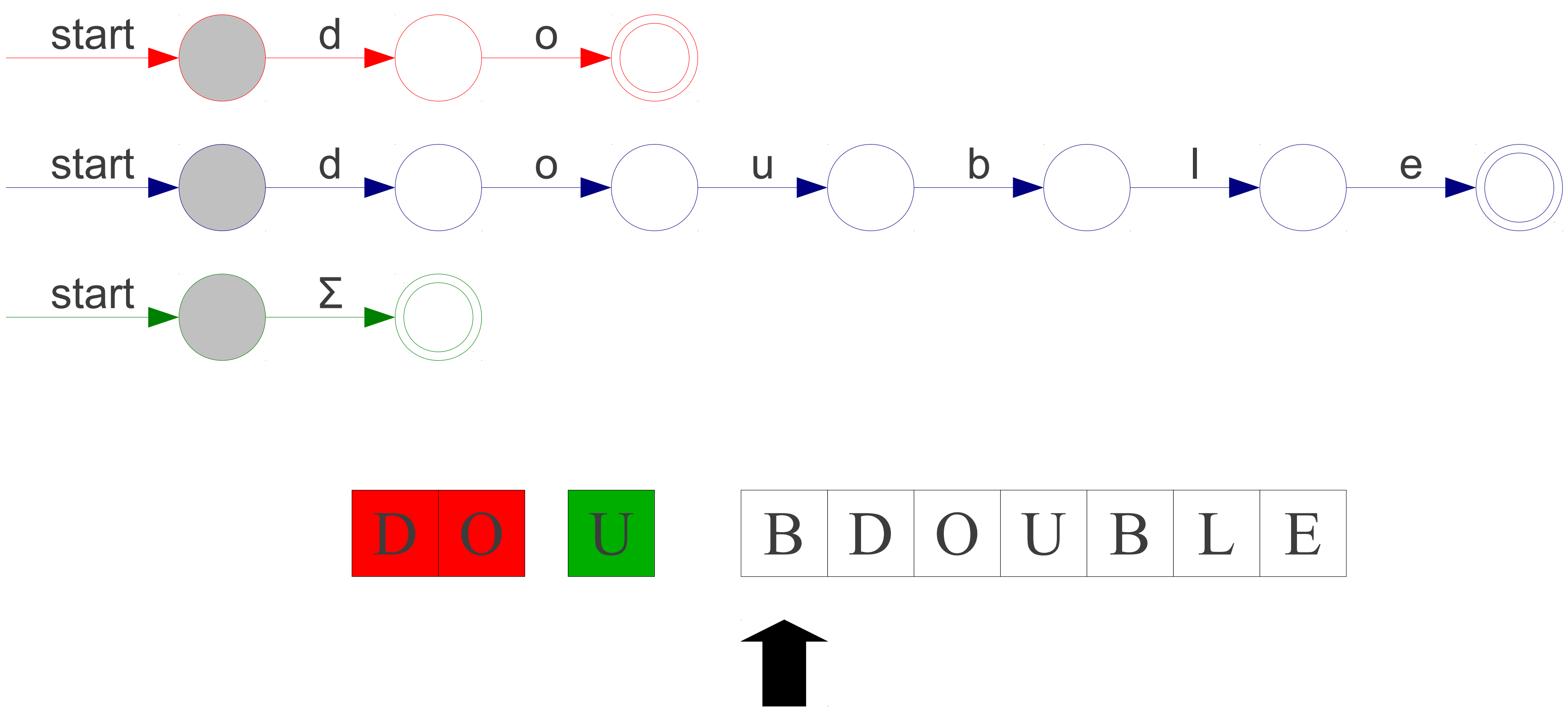
Implementing Maximal Munch

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



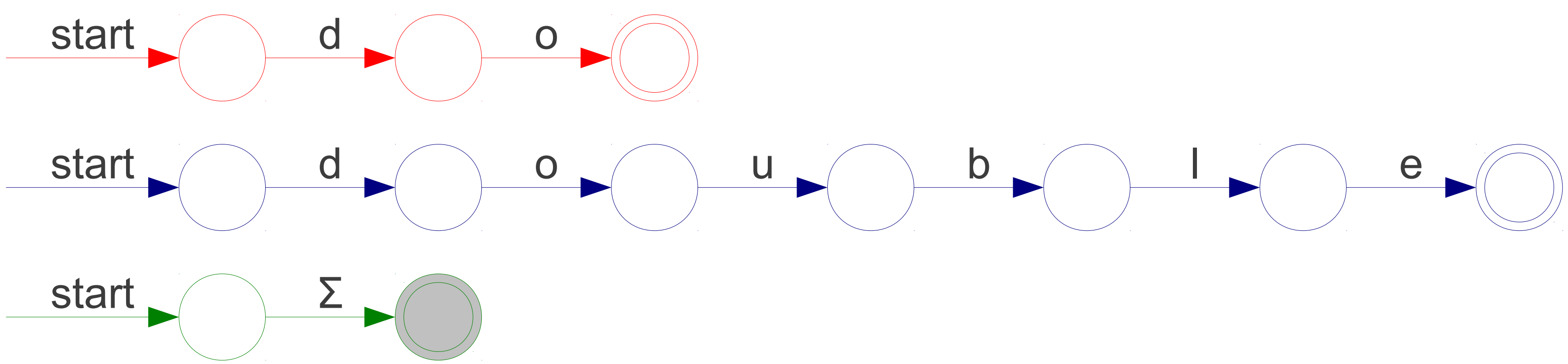
Implementing Maximal Munch

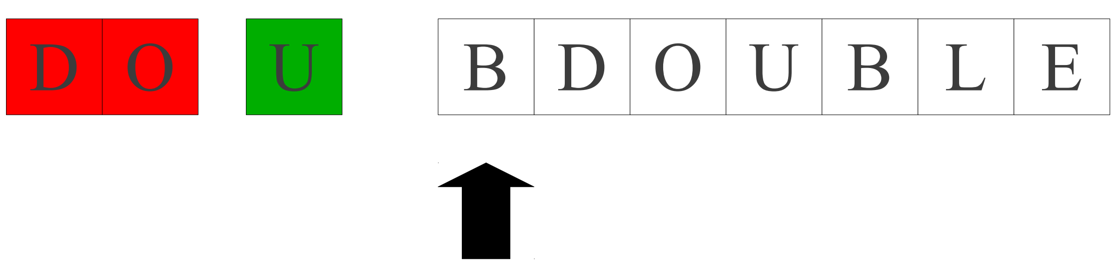
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

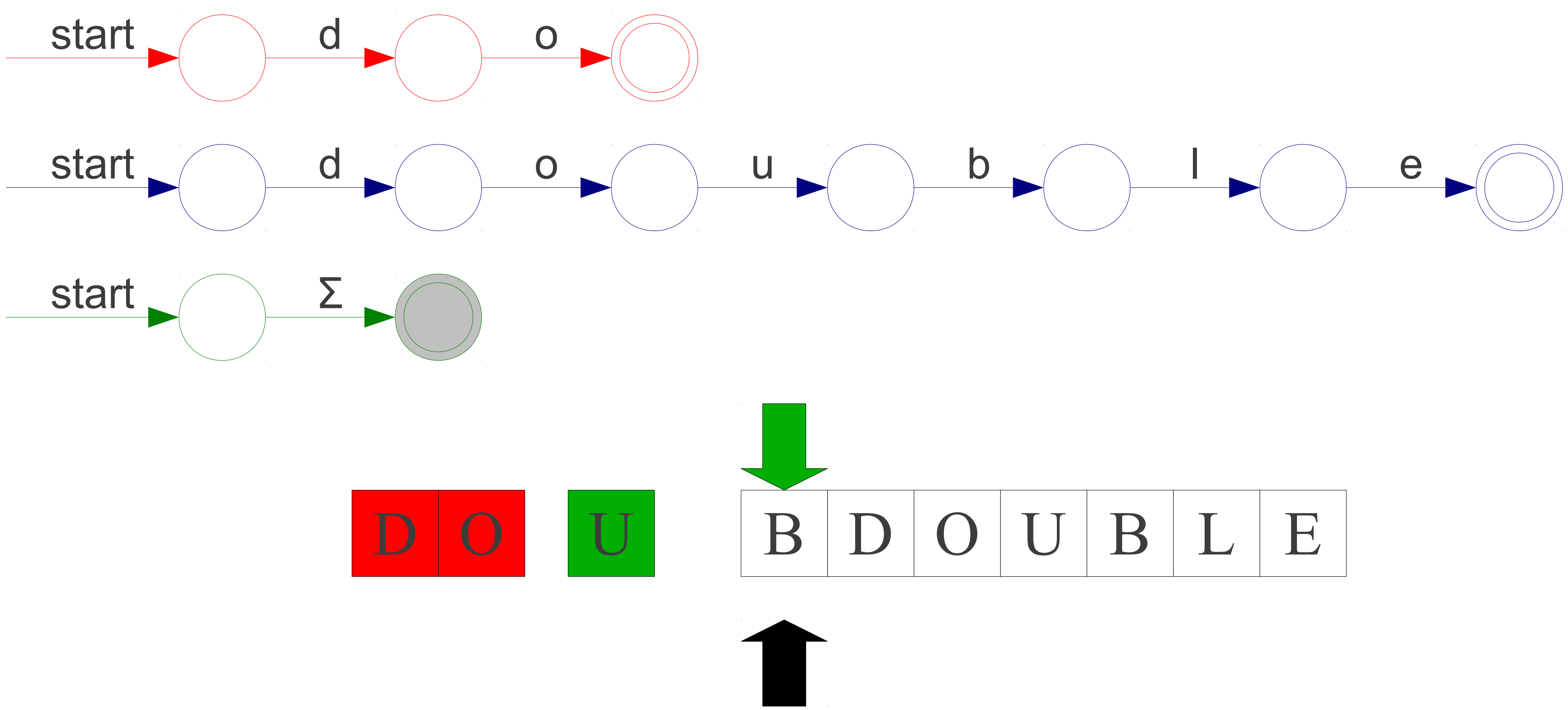
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```





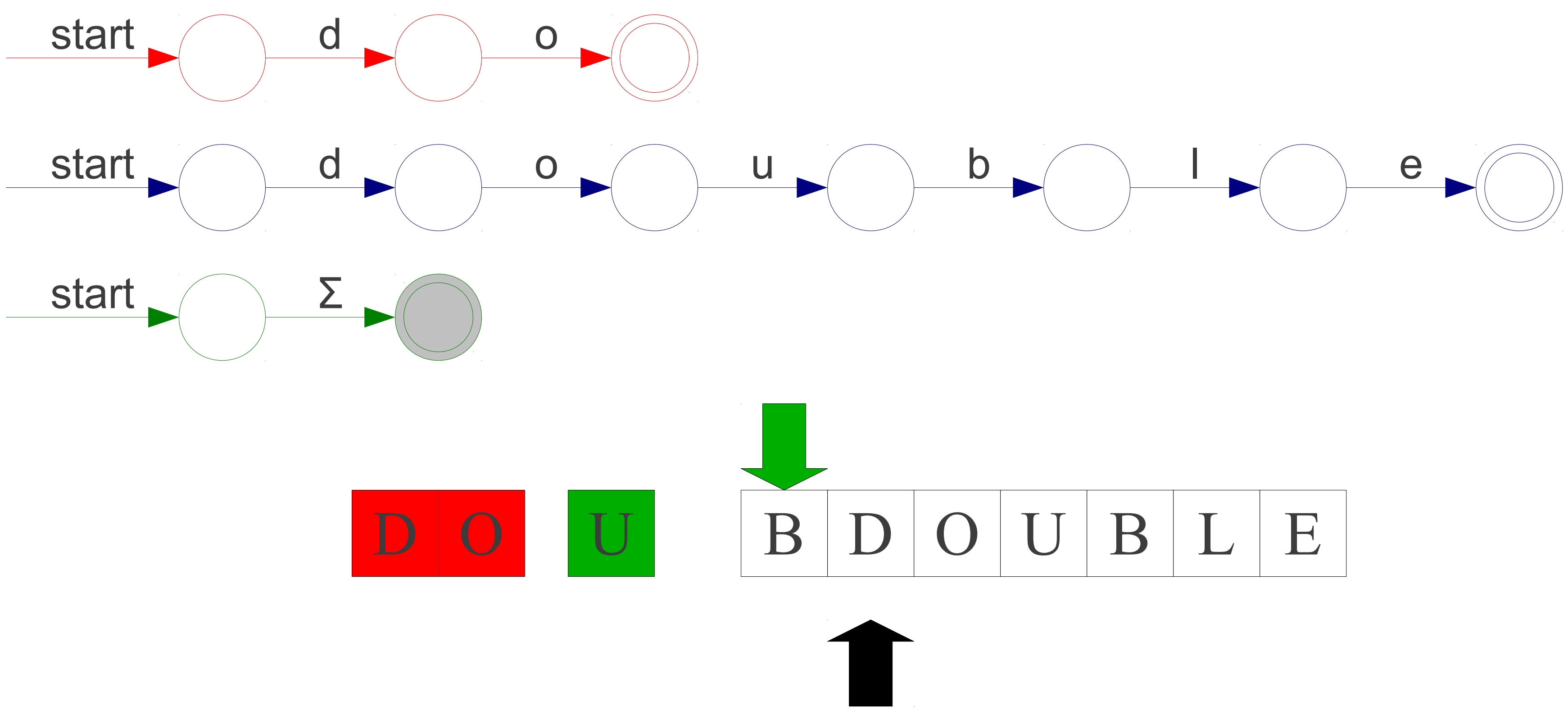
Implementing Maximal Munch

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



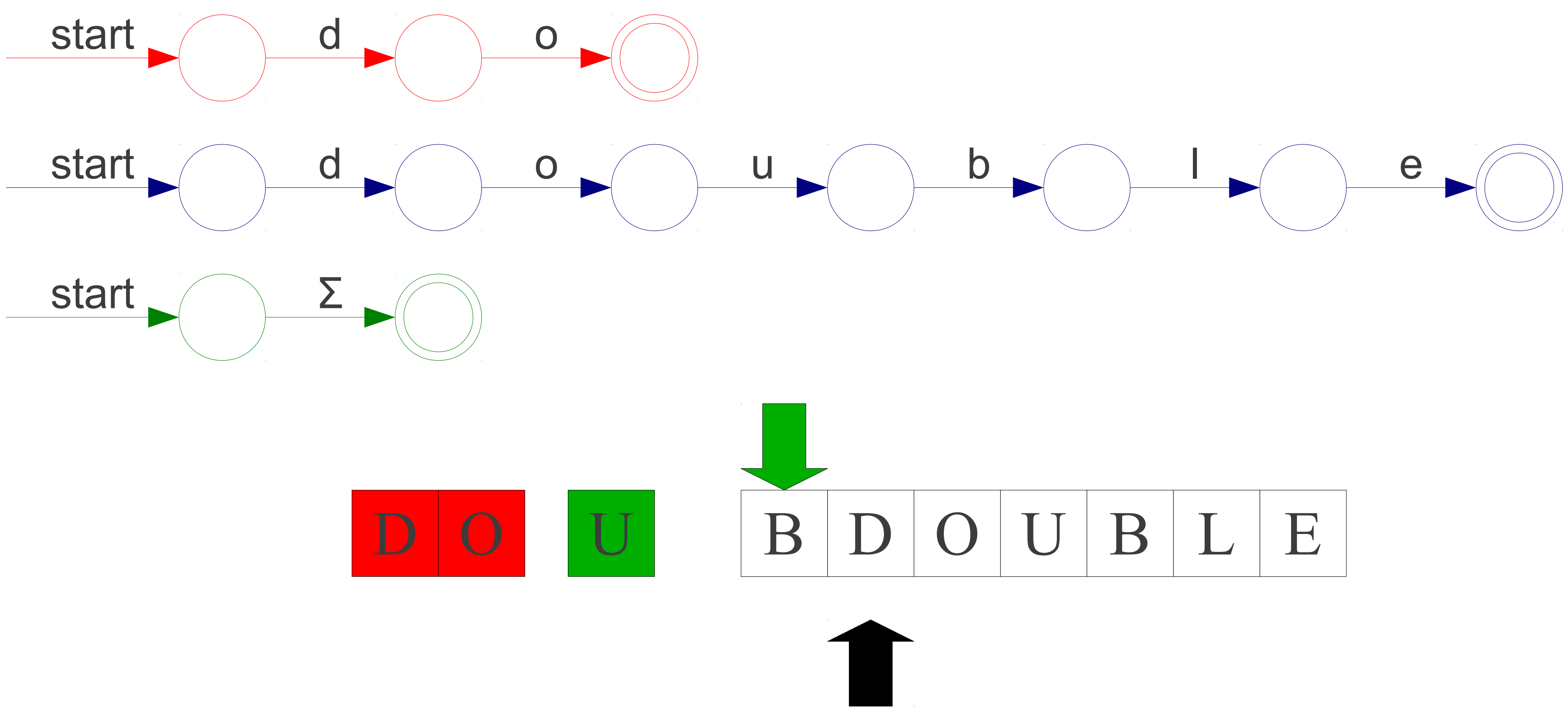
Implementing Maximal Munch

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



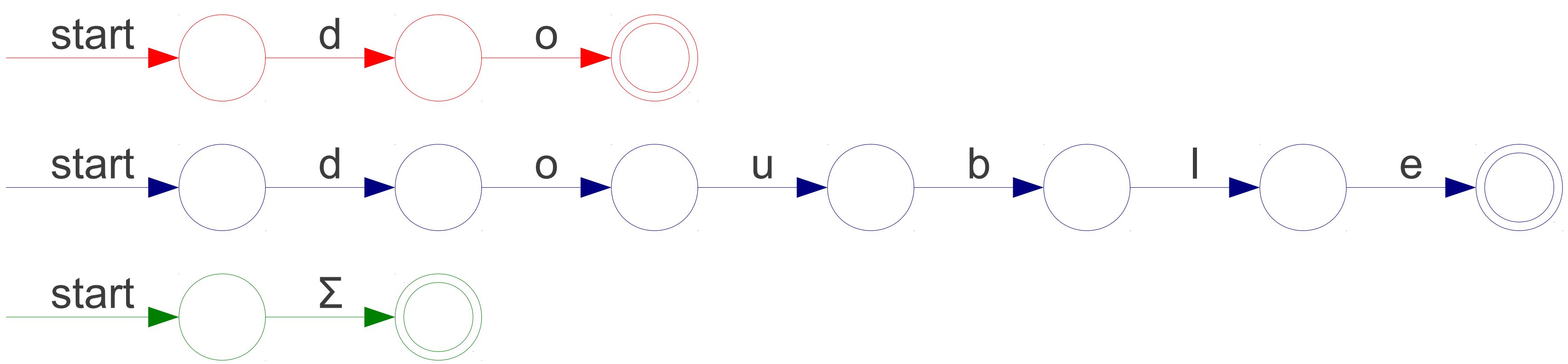
Implementing Maximal Munch

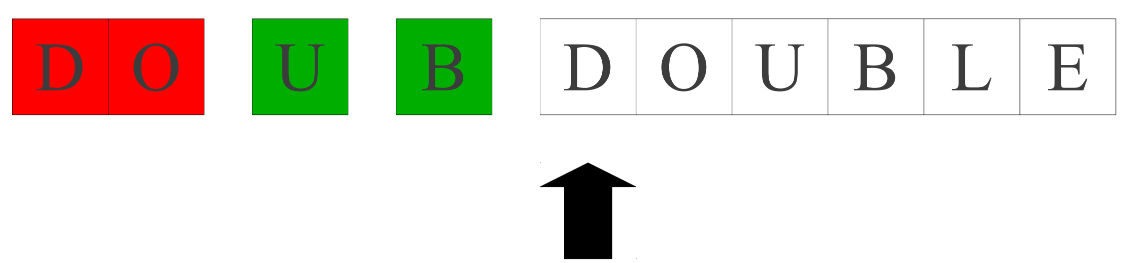
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

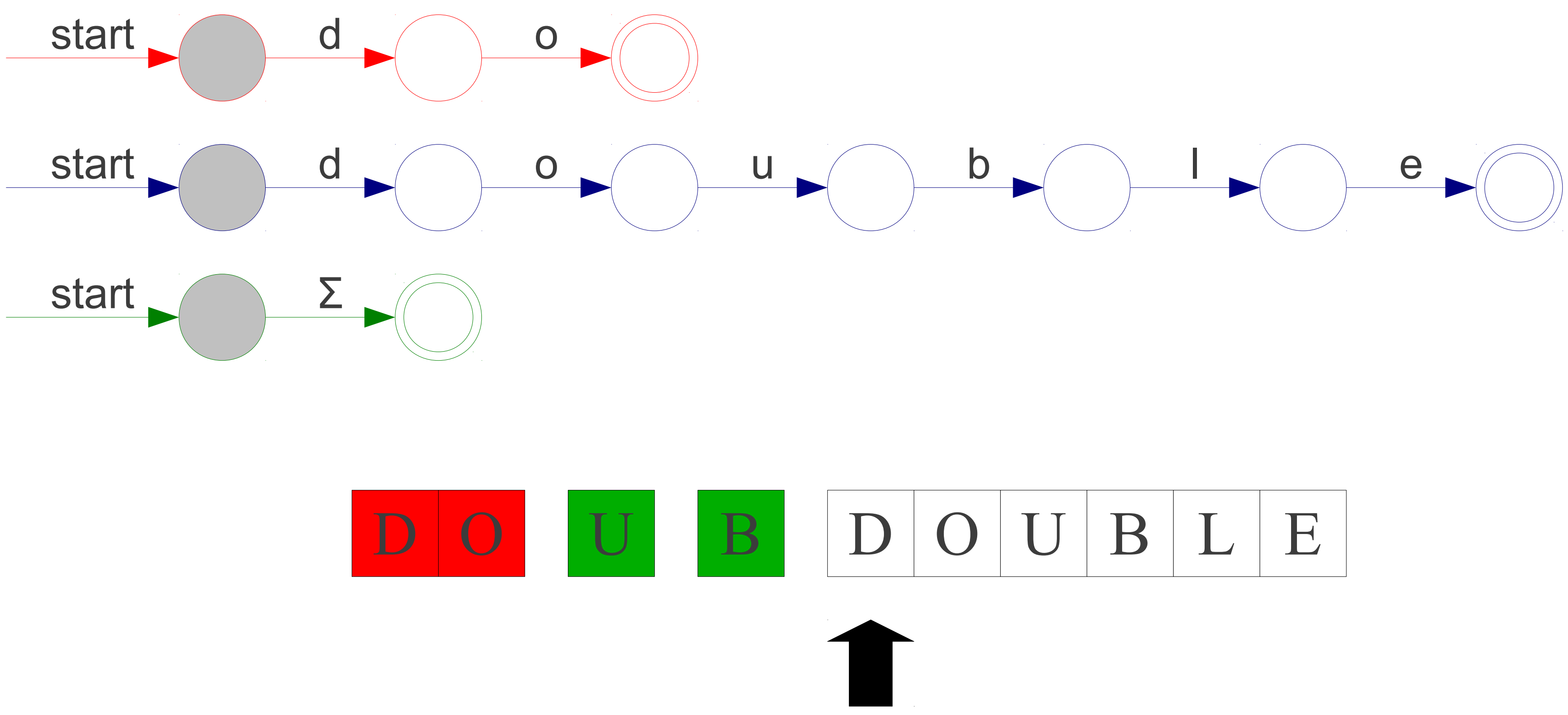
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```





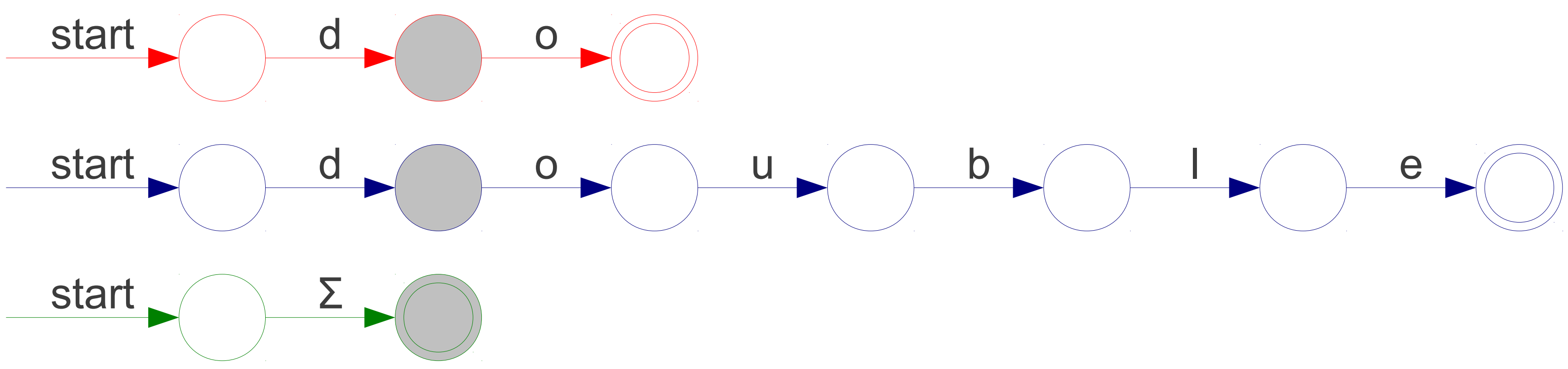
Implementing Maximal Munch

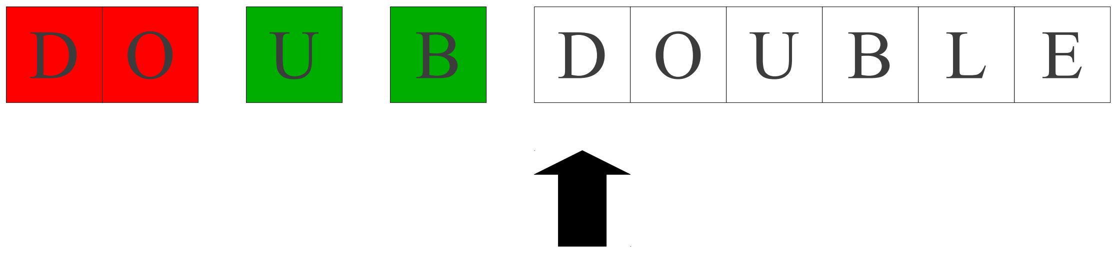
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

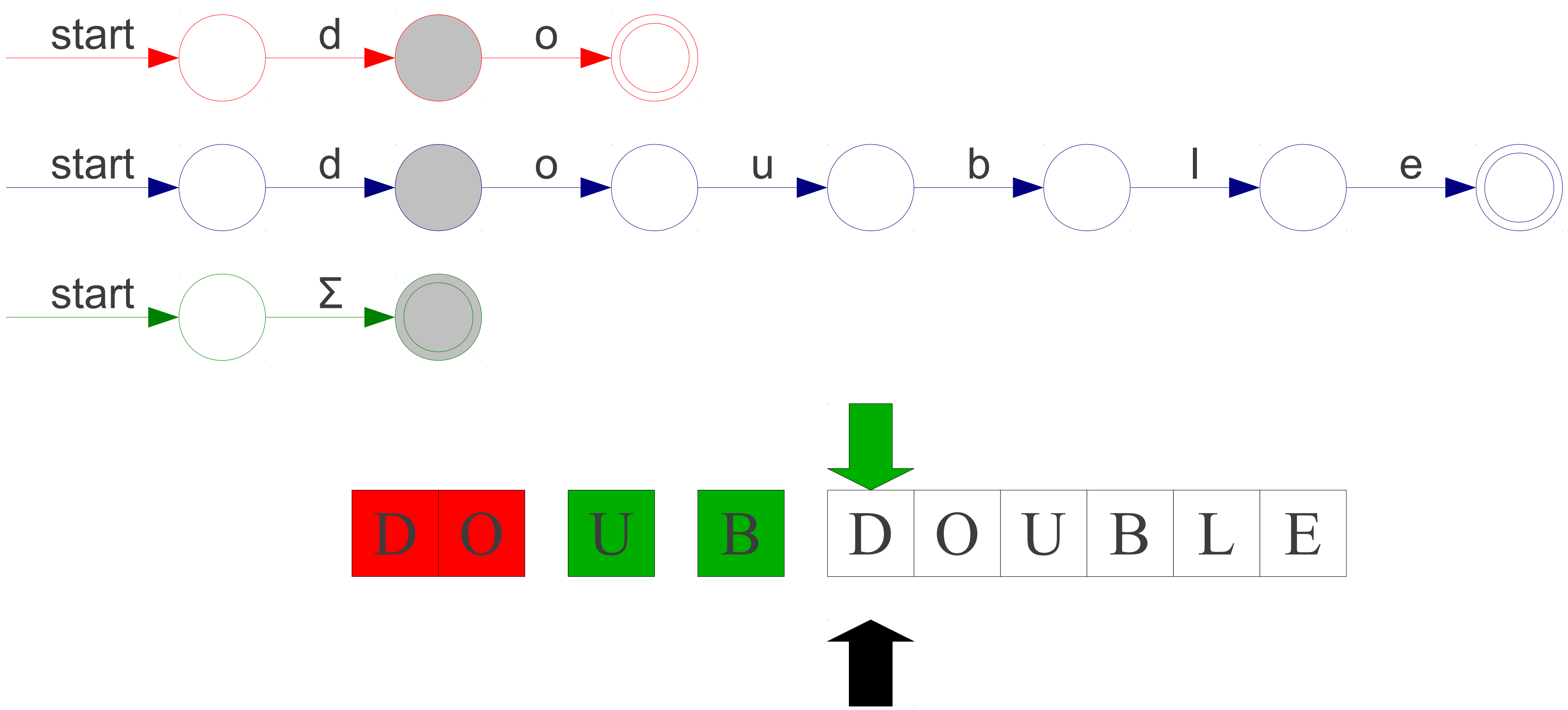
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```





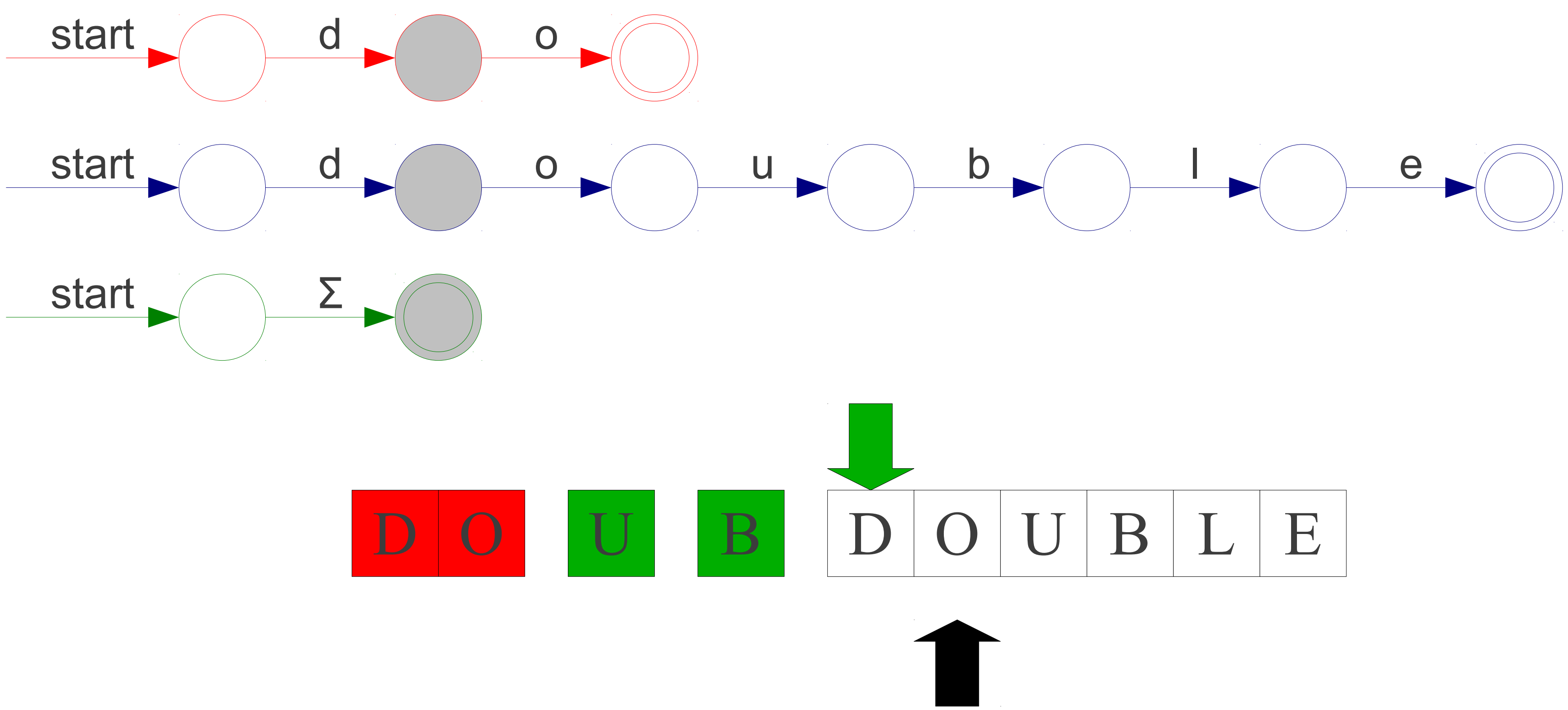
Implementing Maximal Munch

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



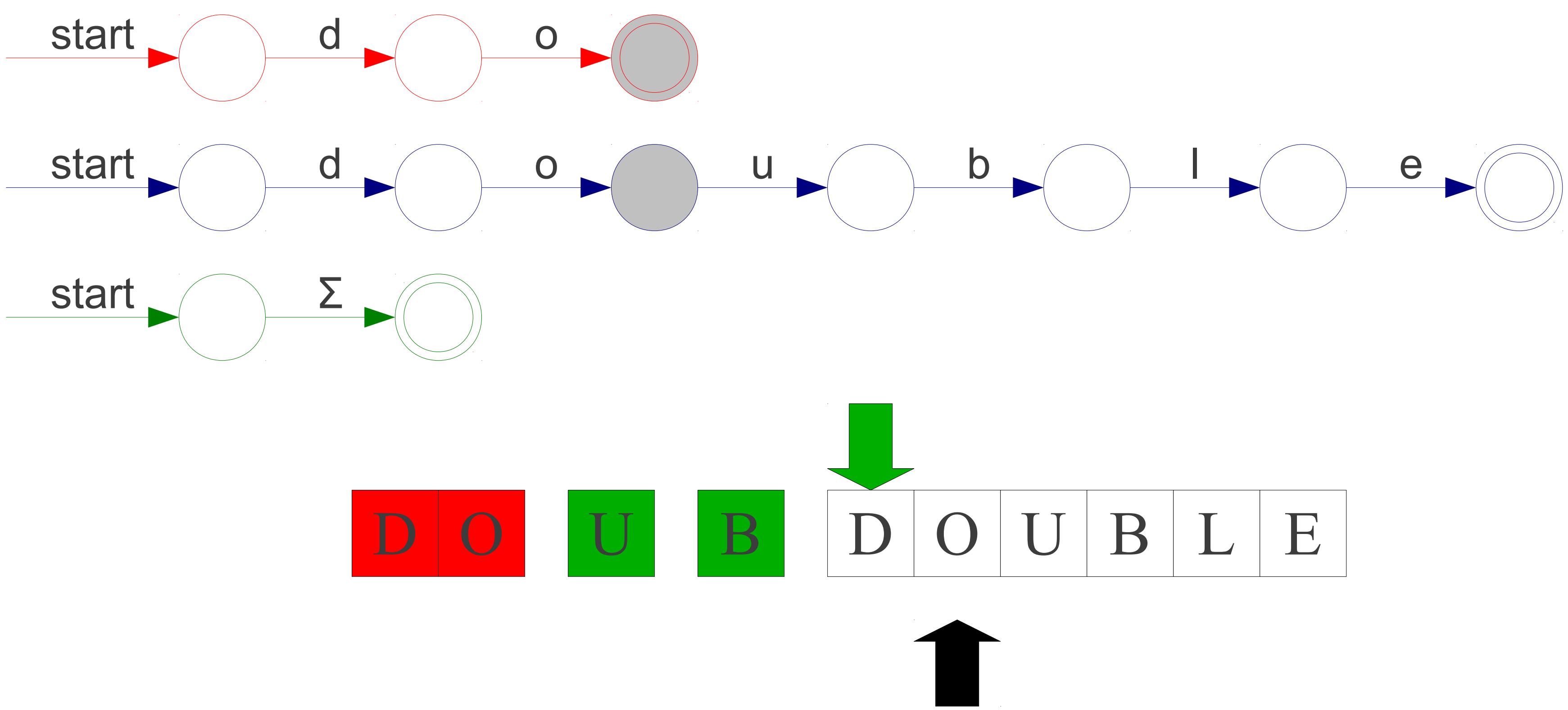
Implementing Maximal Munch

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



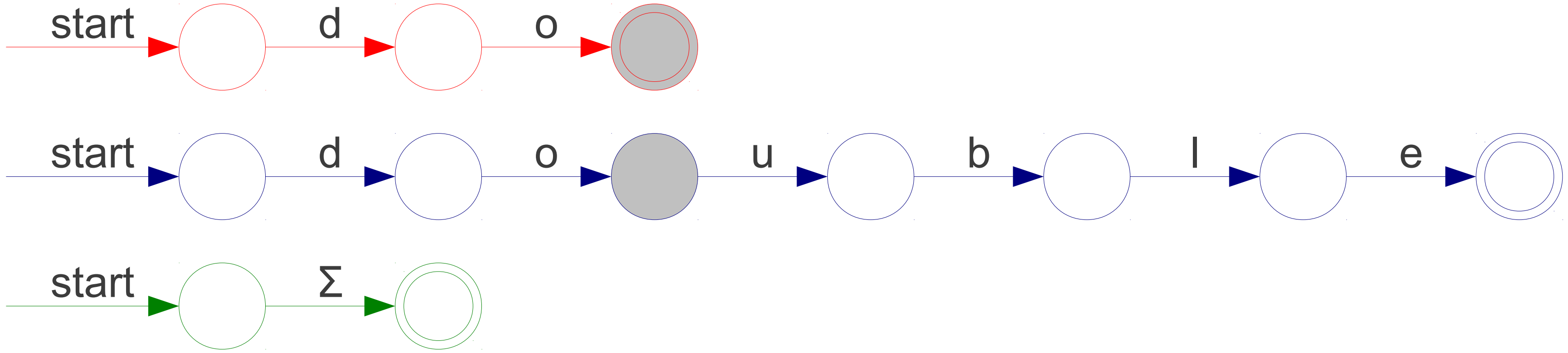
Implementing Maximal Munch

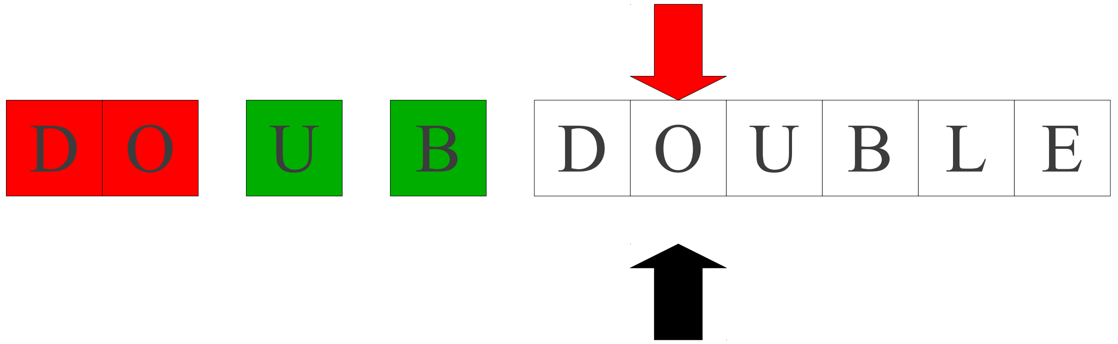
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

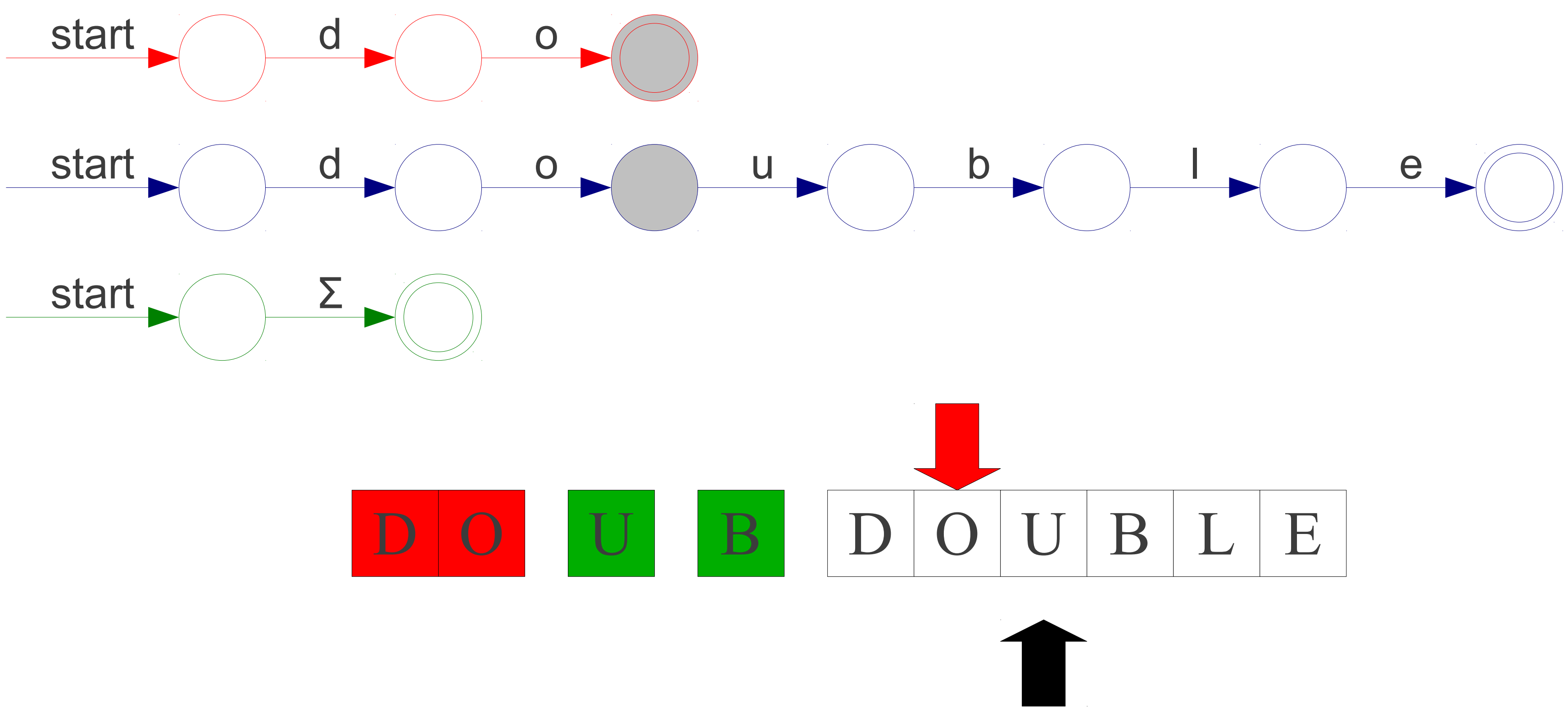
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





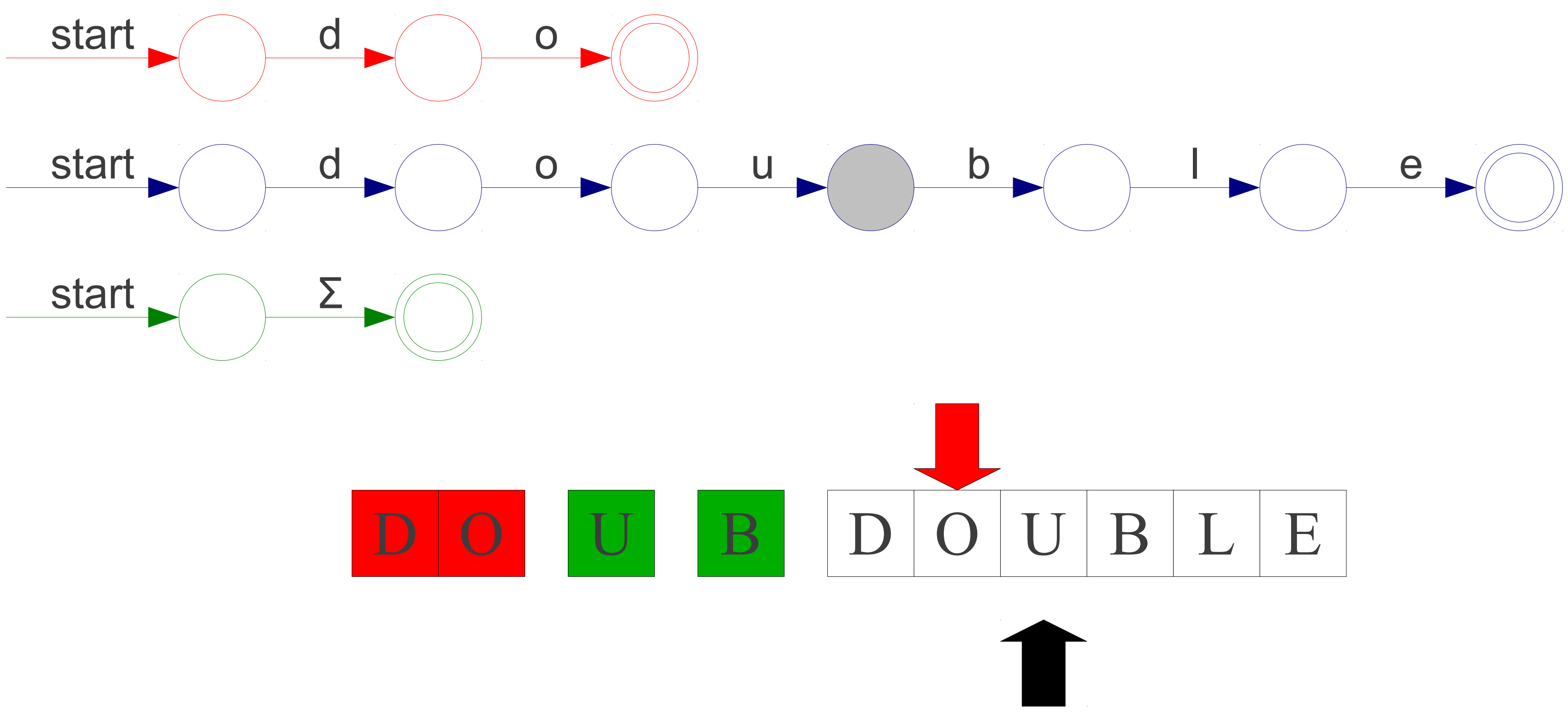
Implementing Maximal Munch

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



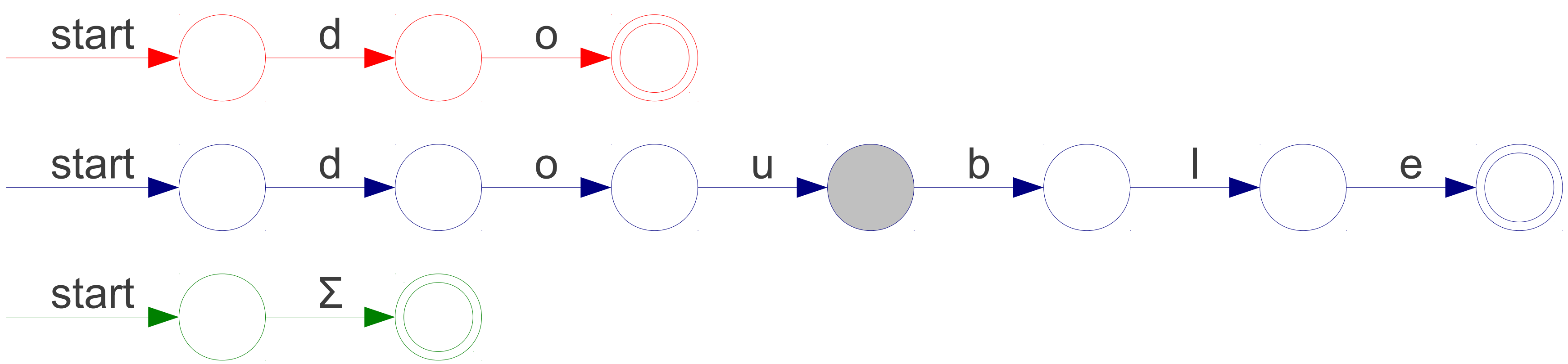
Implementing Maximal Munch

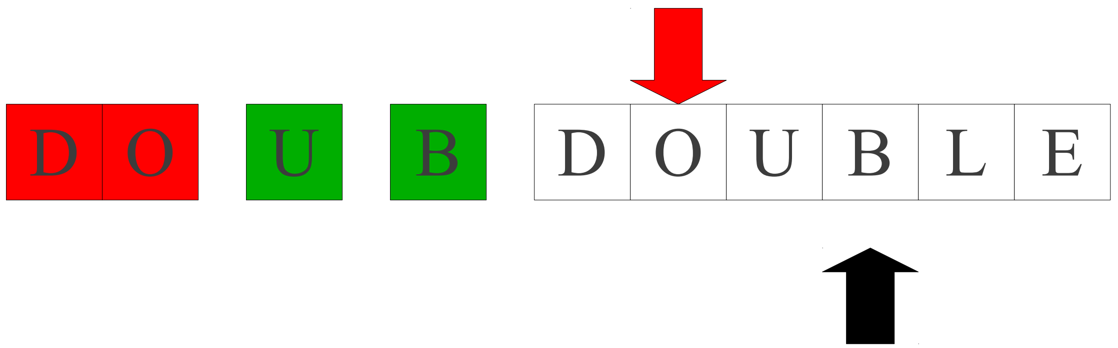
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```



Implementing Maximal Munch

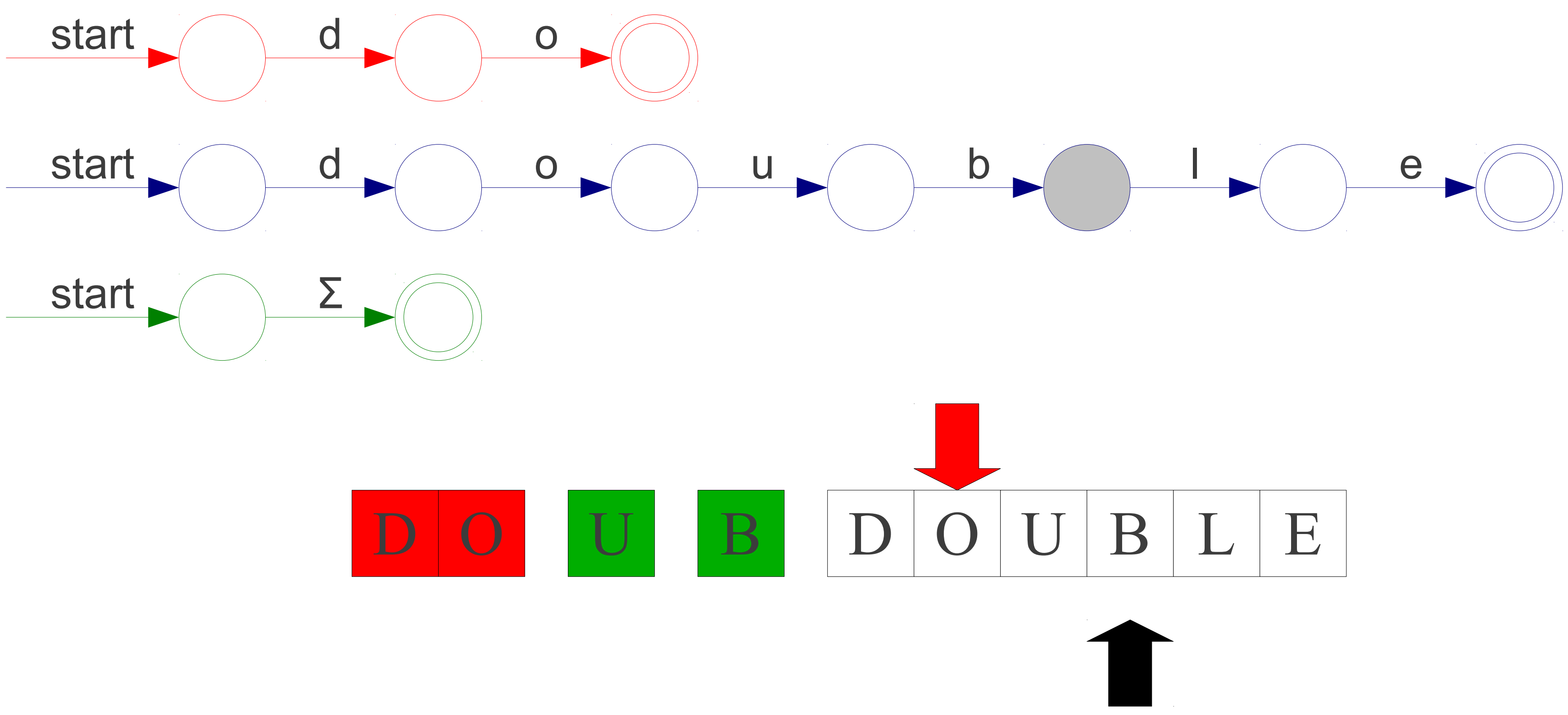
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```





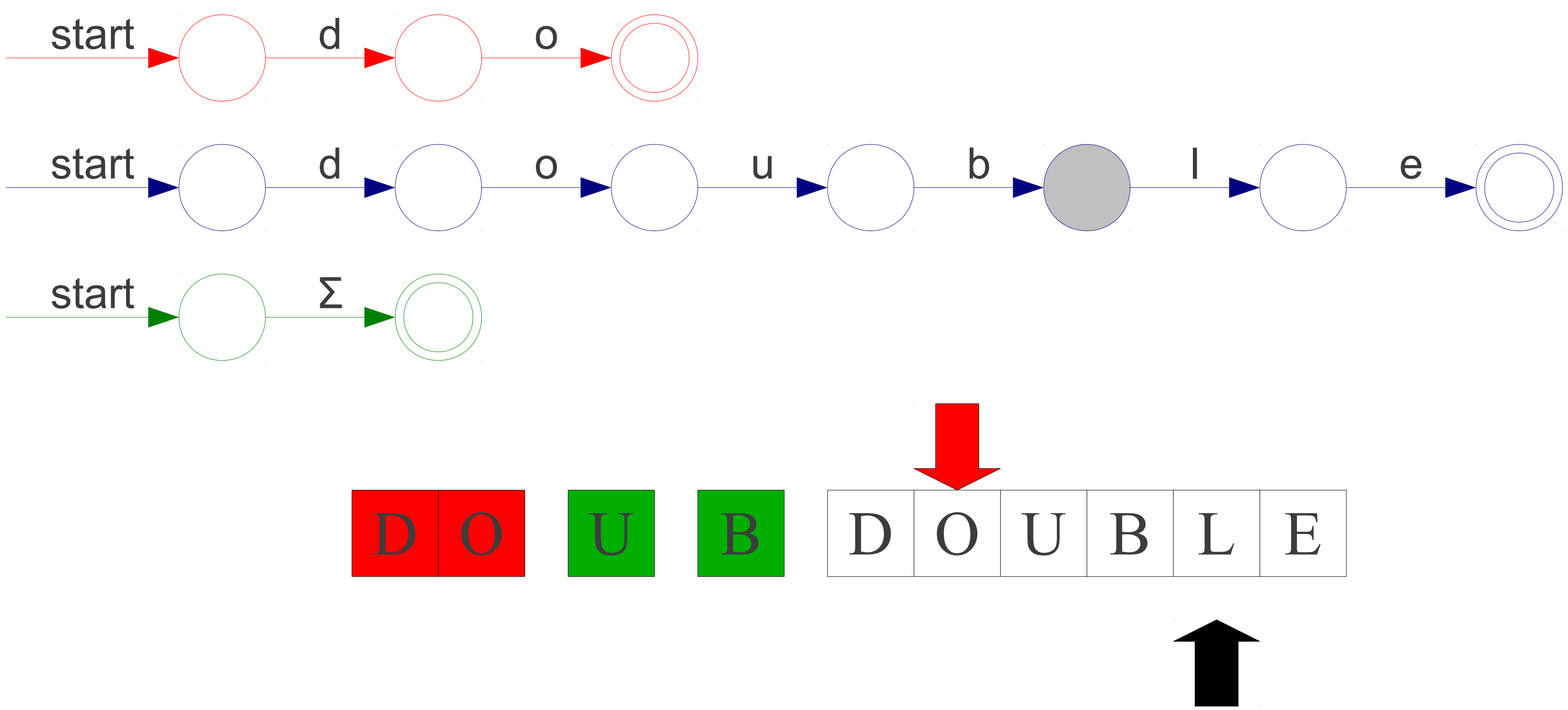
Implementing Maximal Munch

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



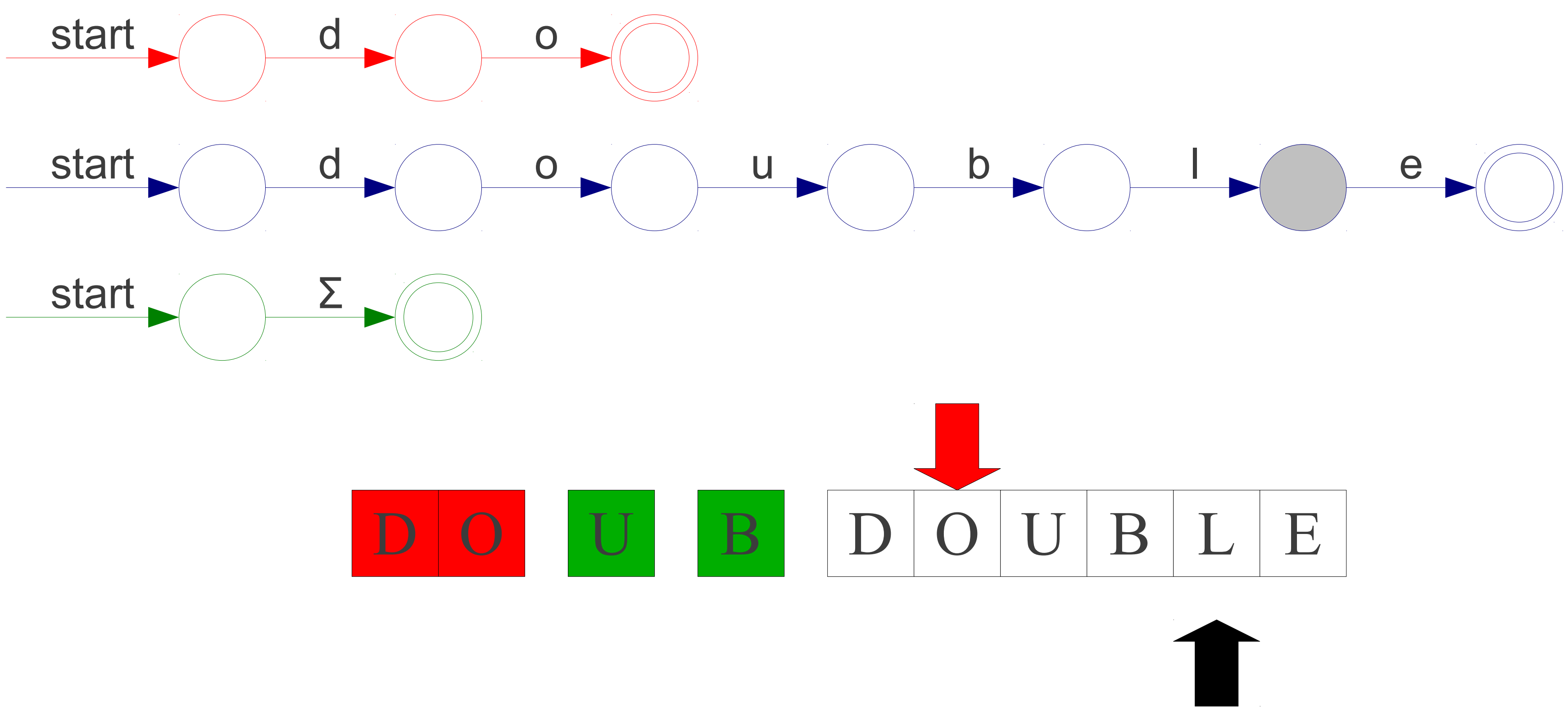
Implementing Maximal Munch

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



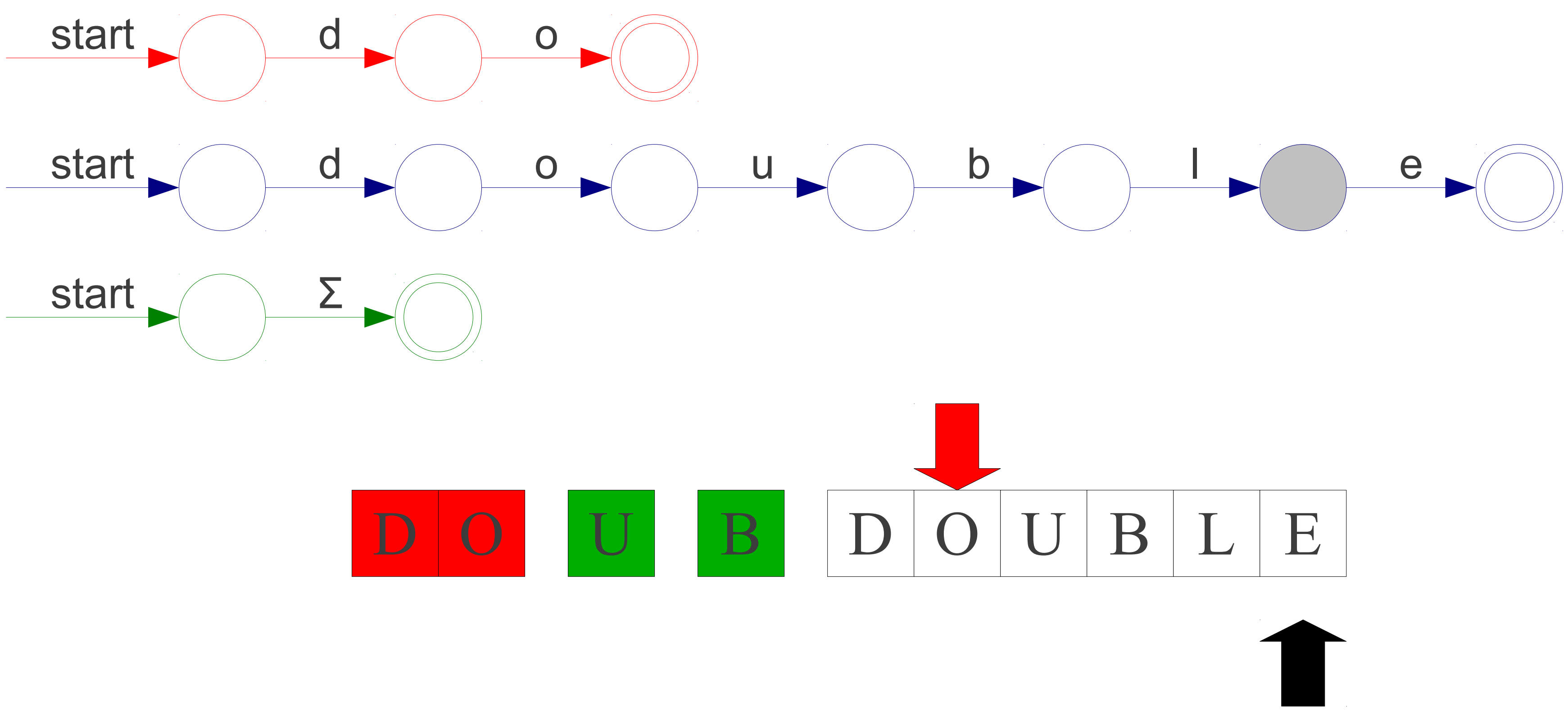
Implementing Maximal Munch

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



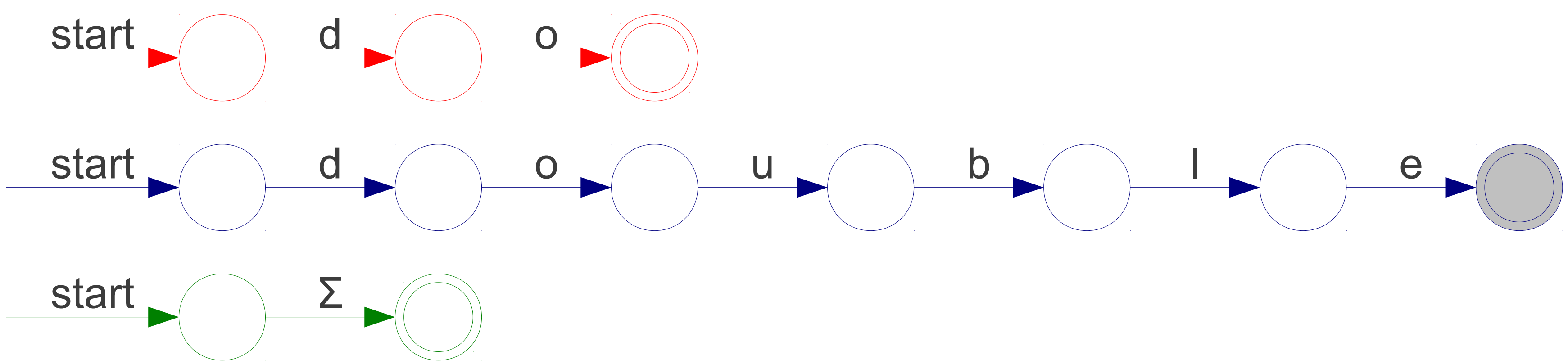
Implementing Maximal Munch

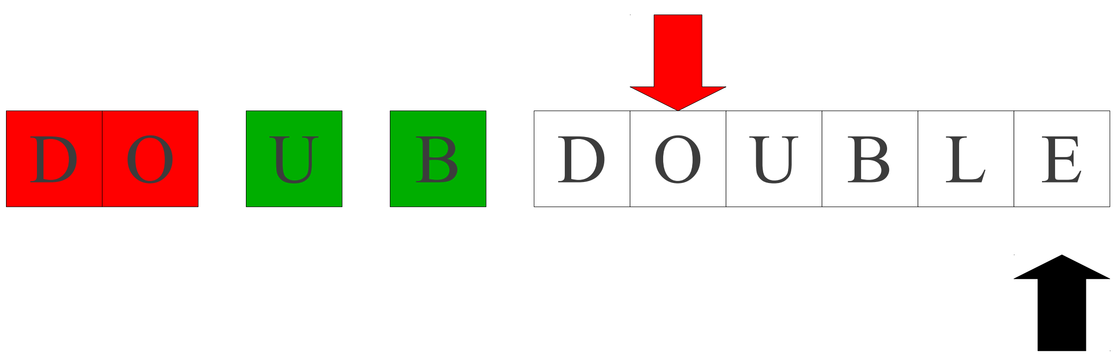
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



Implementing Maximal Munch

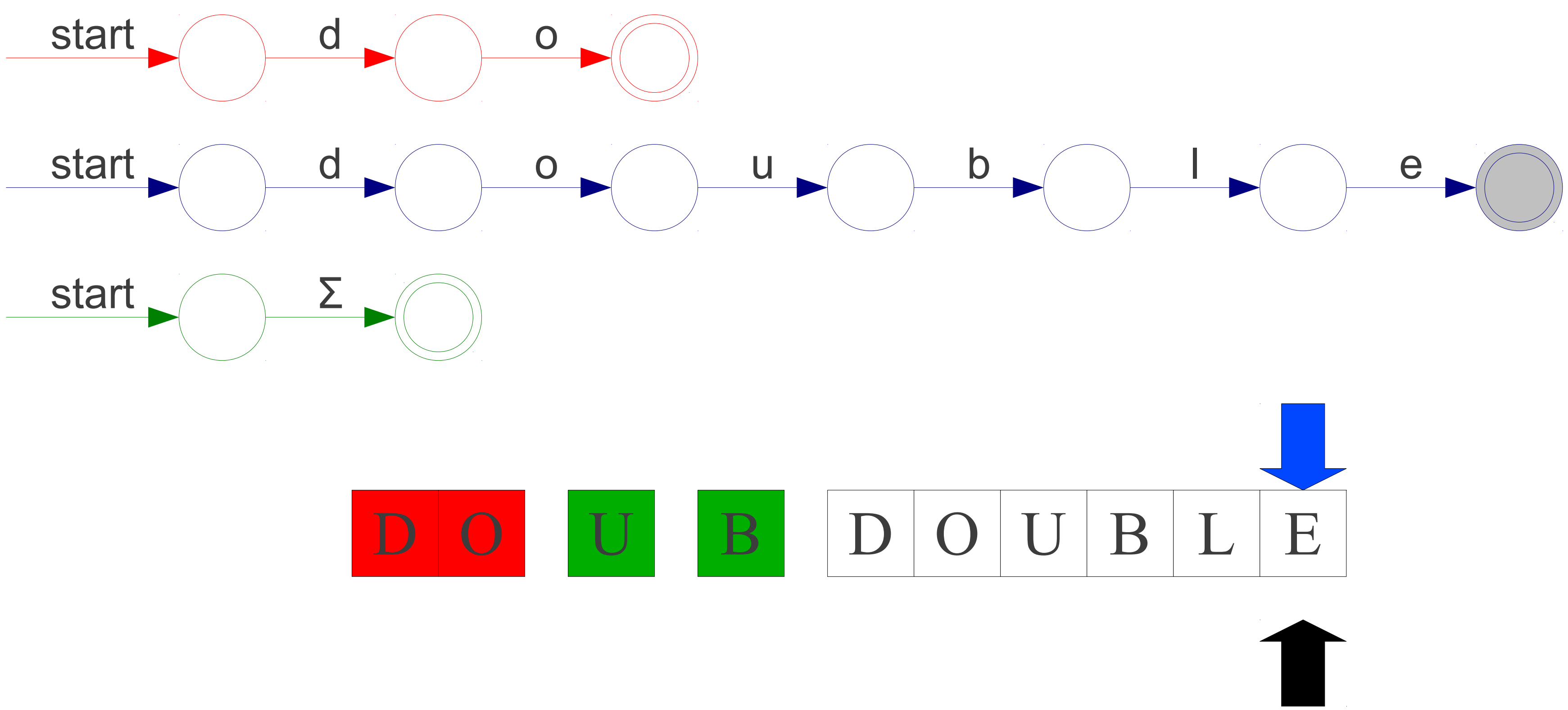
```
T_Do         do
T_Double     double
T_Mystery    [A-Za-z]
```





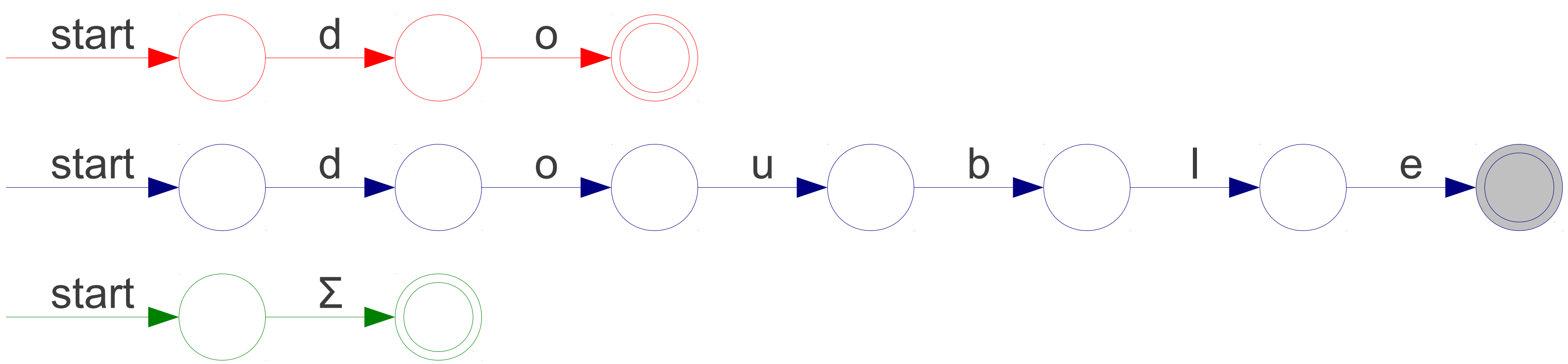
Implementing Maximal Munch

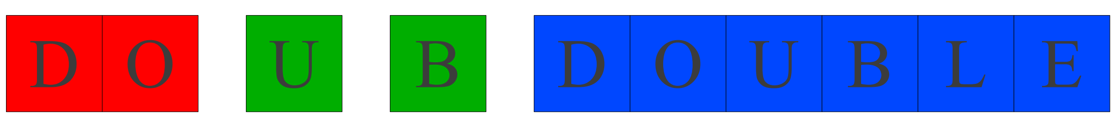
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



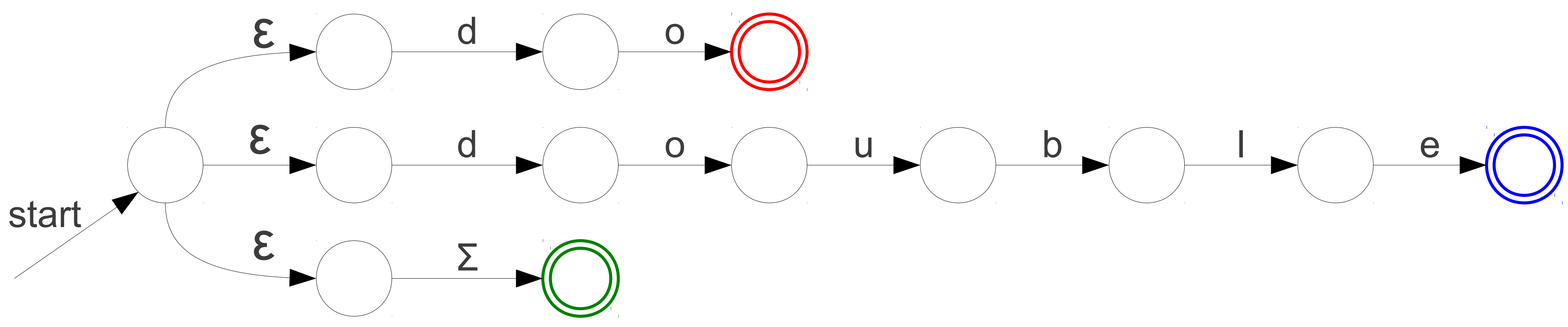
Implementing Maximal Munch

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





A Minor Simplification



Other Conflicts

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

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

Implement a lexical analyzer

- Step 1: Use regular expressions to describe token types (keyword, identifier, integer constant..)

  Number = digit + …
  Keyword = ‘if’ + ‘else’ + …
  Identifier = letter (letter + digit)*
  OpenPar = ‘(‘
  …

  Then construct Regular language R, matching all lexemes for all tokens

  R = Keyword + Identifier + Number + …
  = R1 + R2 + …

- Step 2: Use DFA/NFA to recognize the regular language

- But...good news. you don’t need to implement the algorithms to transform your regular expressions to DFA/NFA to recognize it

  - **flex**: given regular expressions -> output c code that does lexical analysis (it internally generates DFA)

Lexical analyzer

REs + priorities + longest matching token rule

= definition of a lexical analyzer

DFA vs. NFA

- NFAs and DFAs recognize the same set of languages (regular languages)
  - For a given NFA, there exists a DFA, and vice versa

- DFAs are faster to execute
  - There are no choices to consider
  - Tradeoff: simplicity
    - For a given language DFA can be exponentially larger than NFA.

Automating Lexical Analyzer (scanner) Construction

To convert a specification into code:

1 Write down the RE for the input language

2 Build a big NFA

3 Build the DFA that simulates the NFA

4 Systematically shrink the DFA

5 Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood

Automating Lexical Analyzer (scanner) Construction

RE→ NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

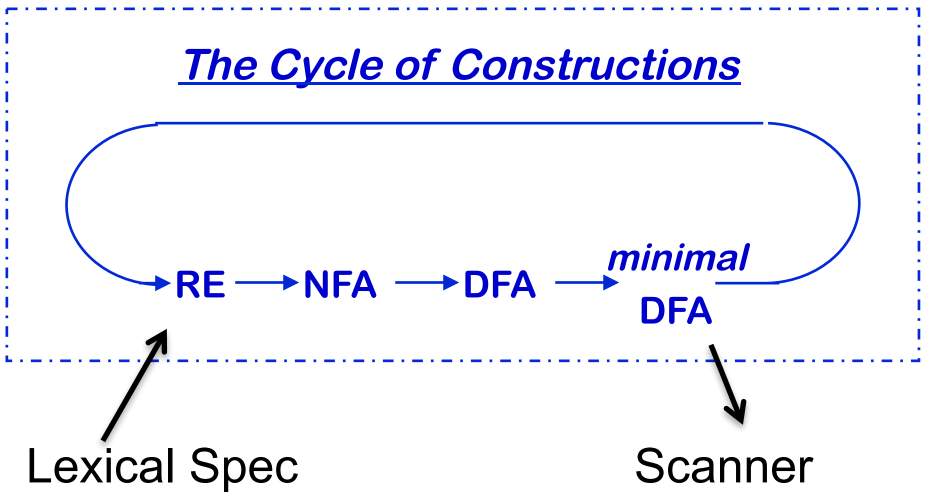
- Build the simulation

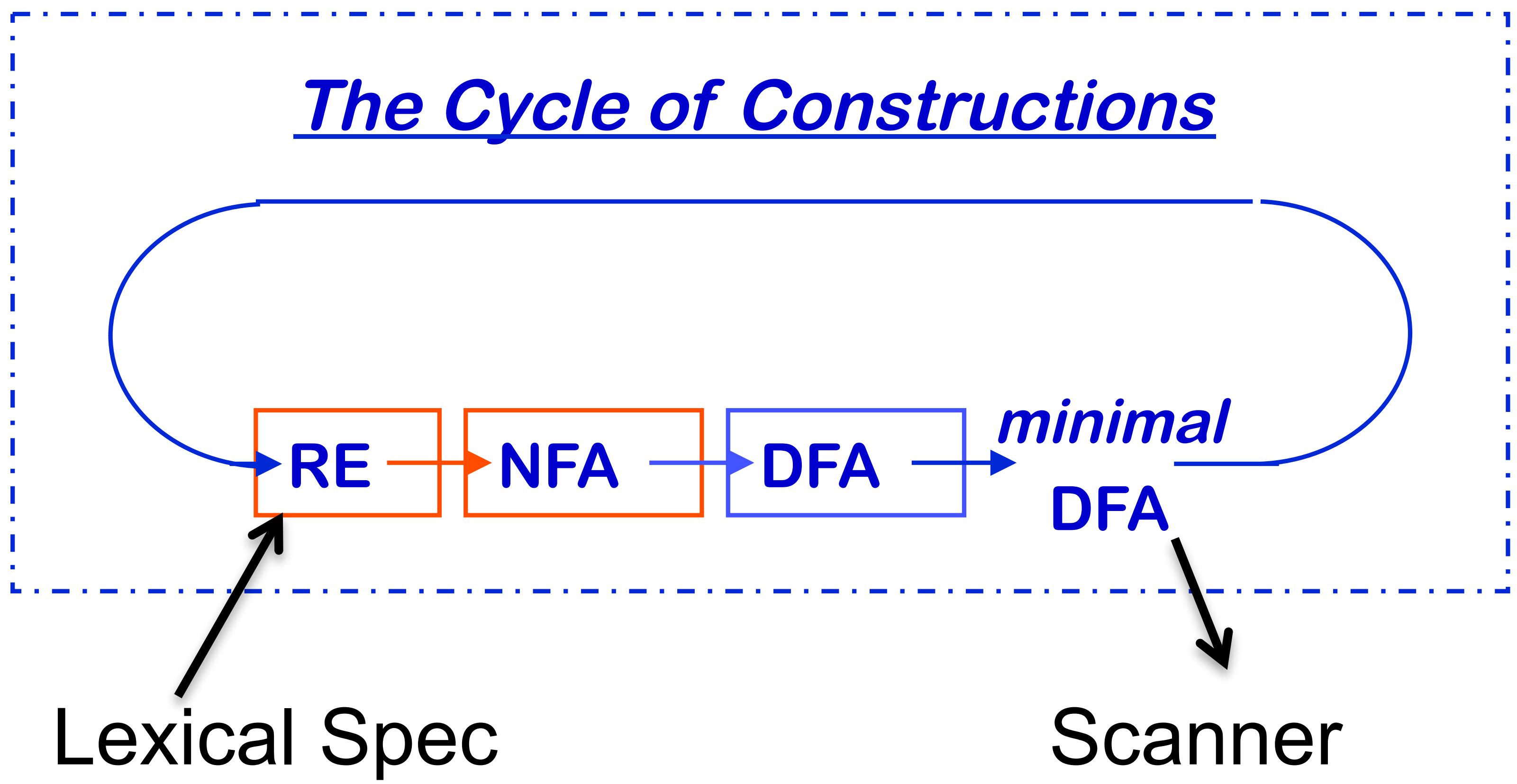
DFA → Minimal DFA

- Hopcroft's algorithm

DFA →RE *(Not part of the scanner construction)*

- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state

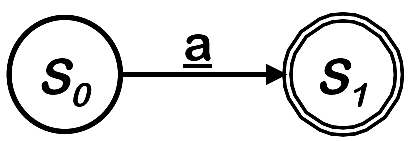



The Cycle of Constructions
RE
NFA
DFA
minimal
DFA
Lexical Spec
Scanner

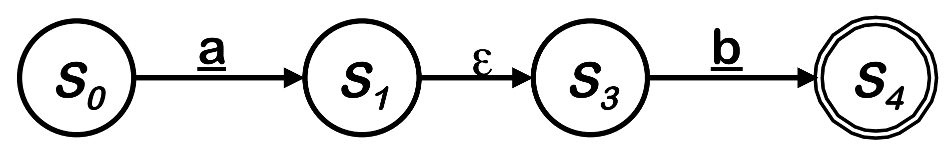
RE →NFA using Thompson's Construction

Key idea

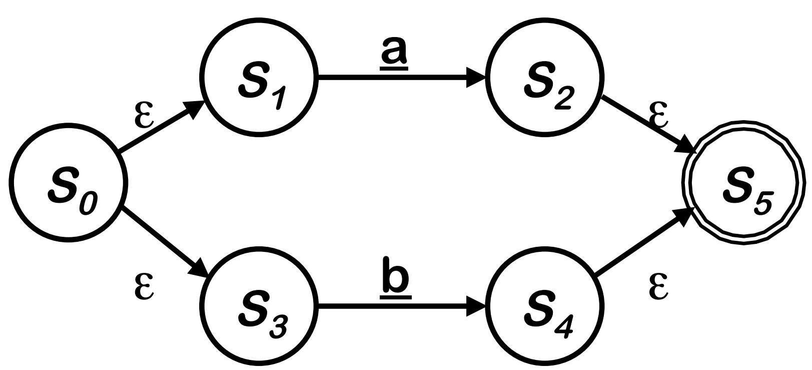
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



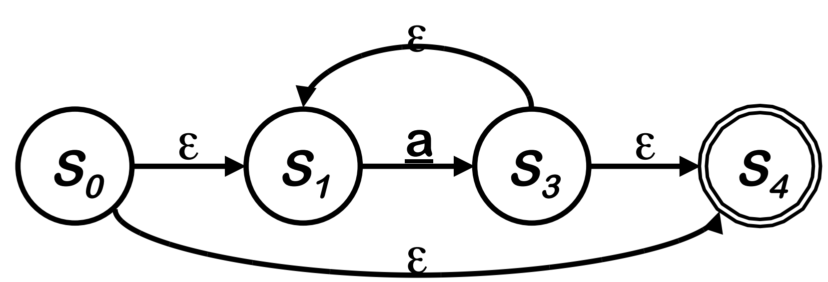
NFA for a



NFA for ab



NFA for a | b



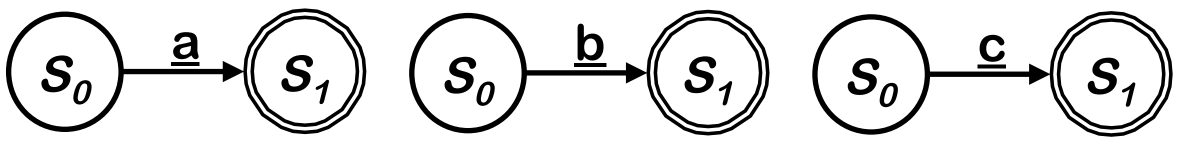
NFA for a^*

Ken Thompson, CACM, 1968

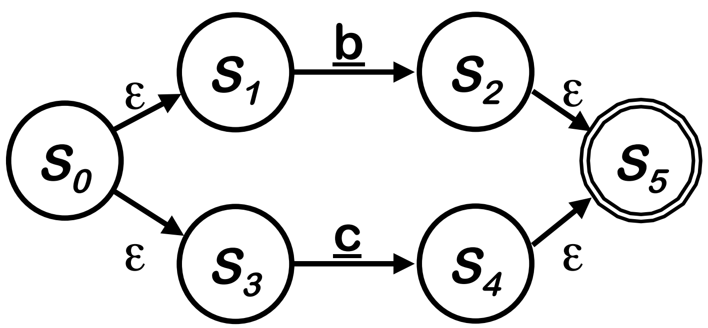
Example of Thompson's Construction

Let's try <u>a</u> (<u>b</u> | <u>c</u>)*

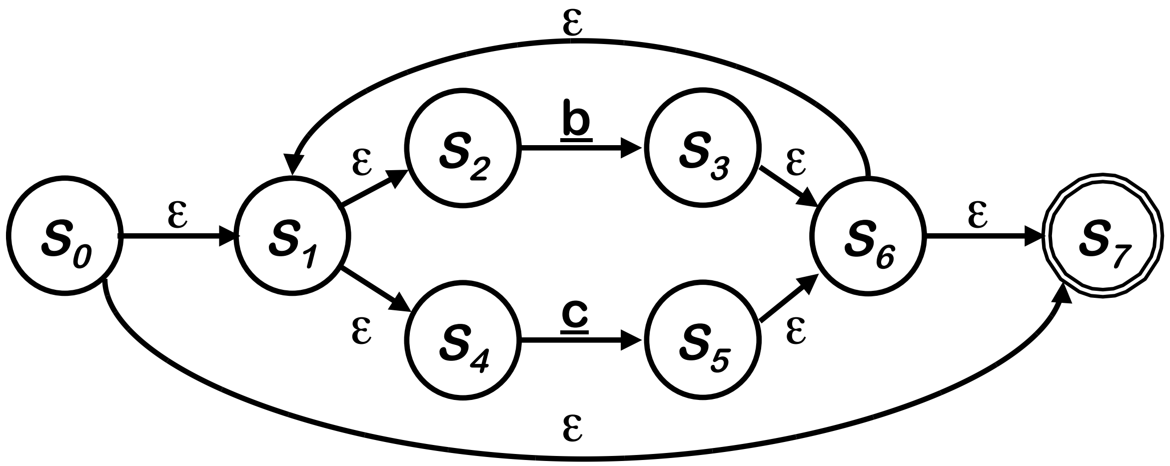
1. <u>a</u>, <u>b</u>, & <u>c</u>



2. <u>b</u> | <u>c</u>

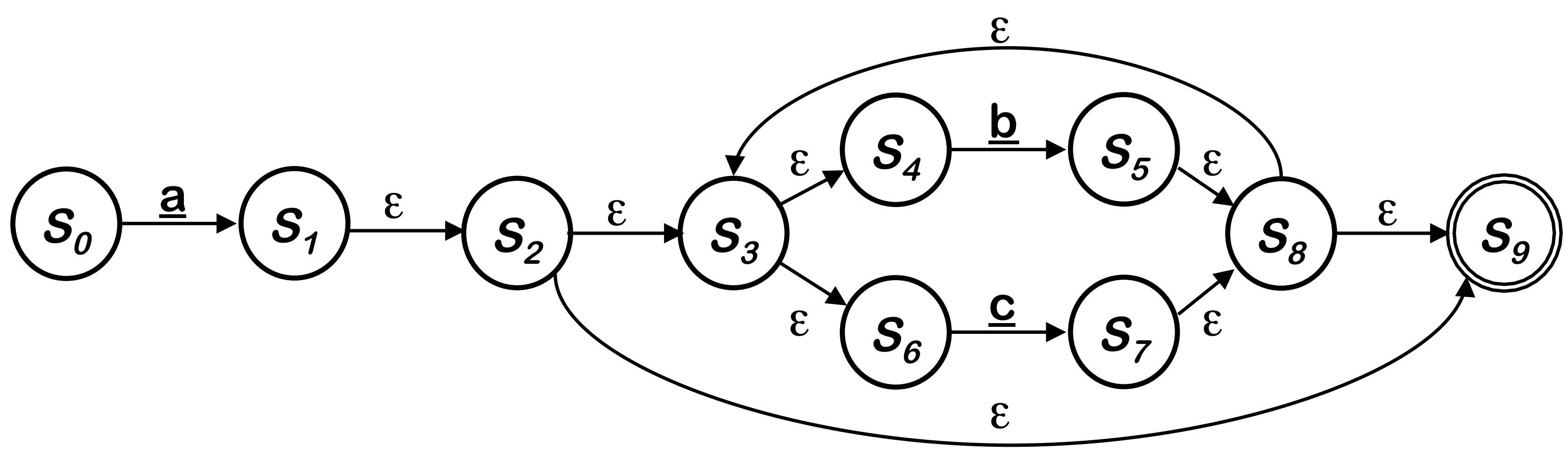


3. (<u>b</u> | <u>c</u>)*

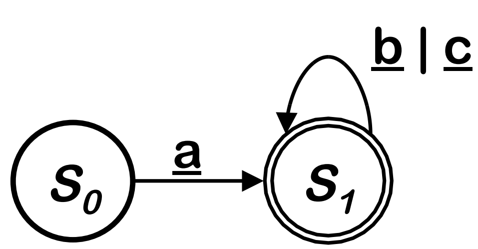


Example of Thompson's Construction *(con't)*

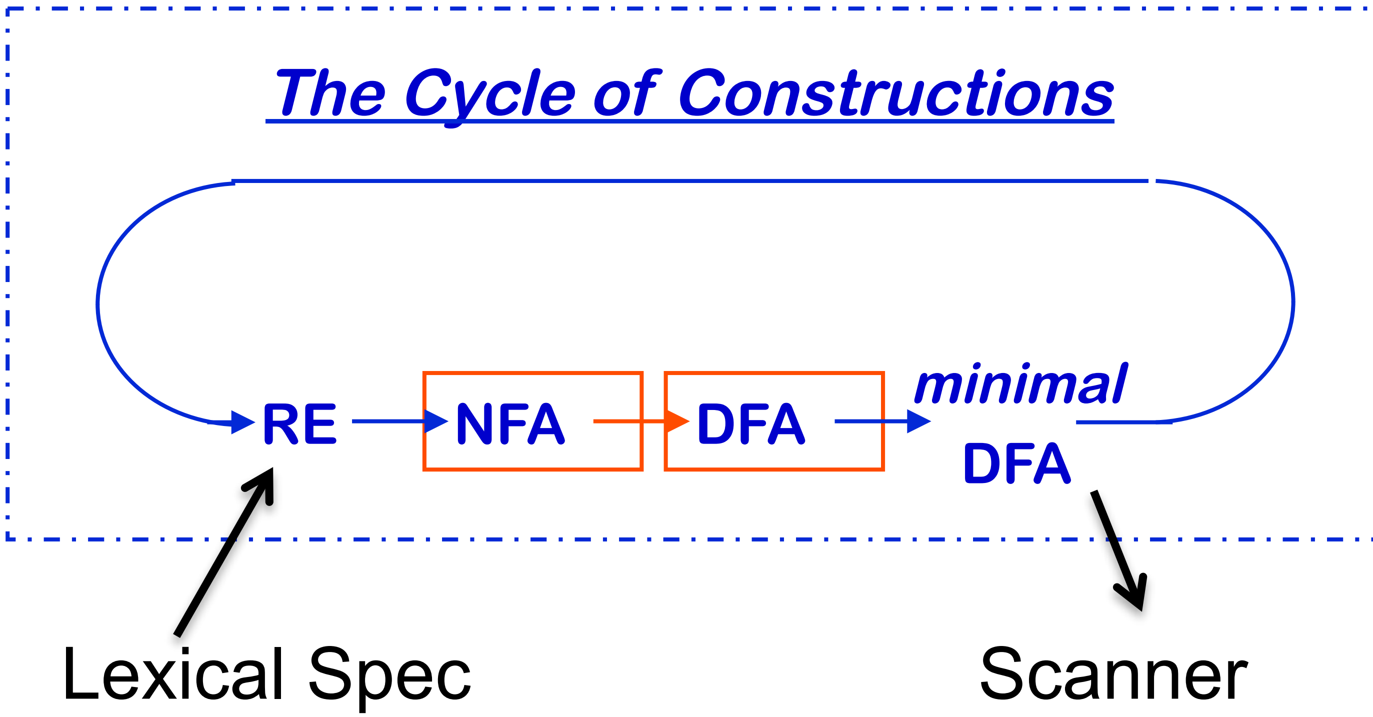
4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...



The Cycle of Constructions
RE
NFA
DFA
minimal
DFA
Lexical Spec
Scanner

NFA to DFA : Trick

- Simulate the NFA
- Each state of DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through e-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from any state in S after seeing the input a, considering ε-moves as well

NFA to DFA : cont..

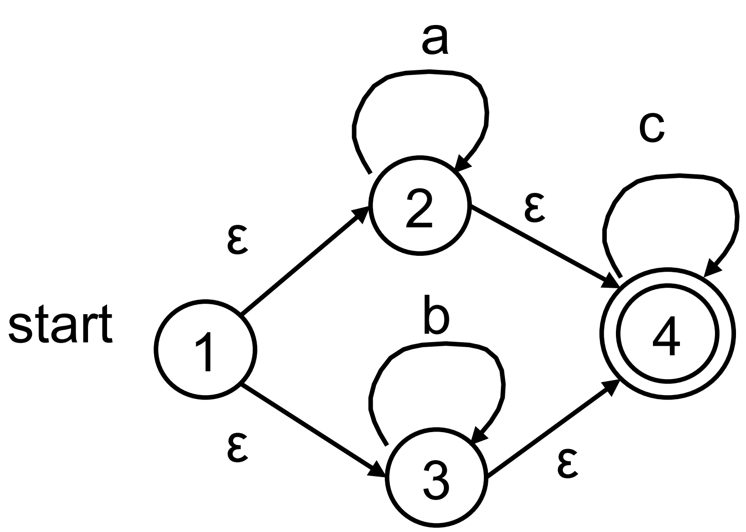
- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?

  $2^N - 1$ = finitely many

NFA to DFA

- Remove the non-determinism
  - States with multiple outgoing edges due to same input
  - ε transitions

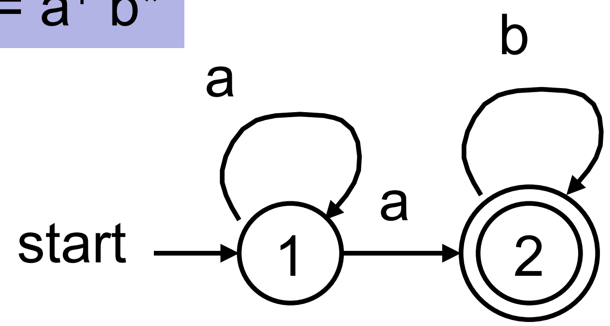
(a*| b*) c*



NFA to DFA (2)

- Multiple transitions
  - Solve by subset construction
  - Build new DFA based upon the set of states each representing a unique subset of states in NFA

$R = a^{+} b^{*}$

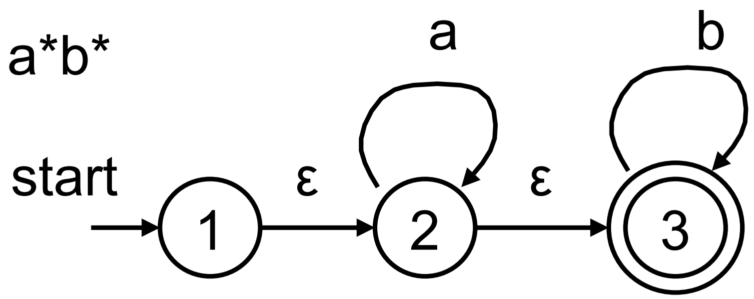


ε-closure(1) = {1} include state “1”
(1,a) → {1,2} include state “1/2”
(1,b) → ERROR

NFA to DFA (3)

- ε transitions
  - Any state reachable by an ε transition is “part of the state”
  - ε-closure - Any state reachable from S by ε transitions is in the ε-closure; treat ε-closure as 1 big state, always include ε-closure as part of the state

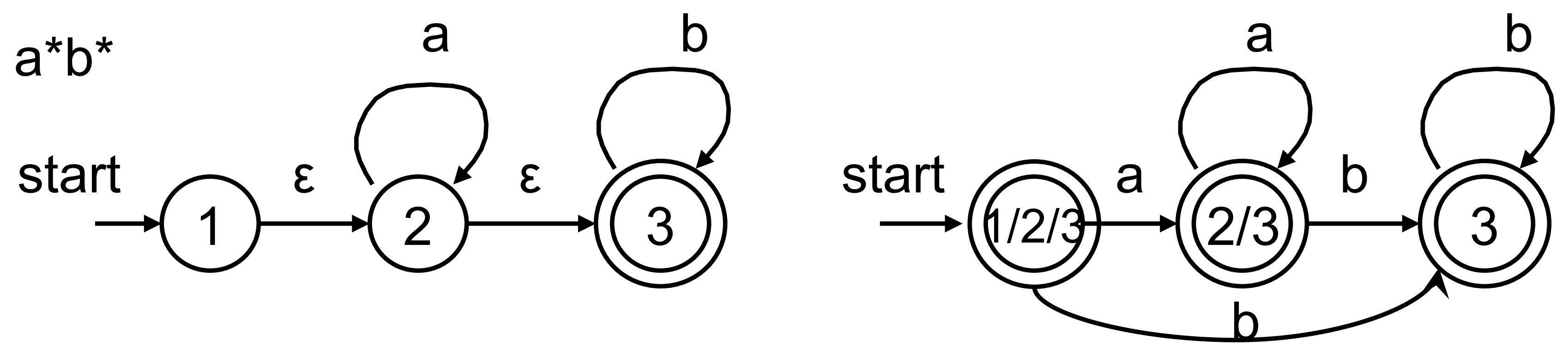
a*b*



1. ε-closure(1) = {1,2,3}; include1/2/3
2. Move(1/2/3, a) = {2, 3} + ε-closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} + ε-closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} + ε-closure(2) = {2,3}
5. Move(2/3, b) = {3} + ε-closure(3) = {3}
6. Move(3, b) = {3} + ε-closure(3) = {3}

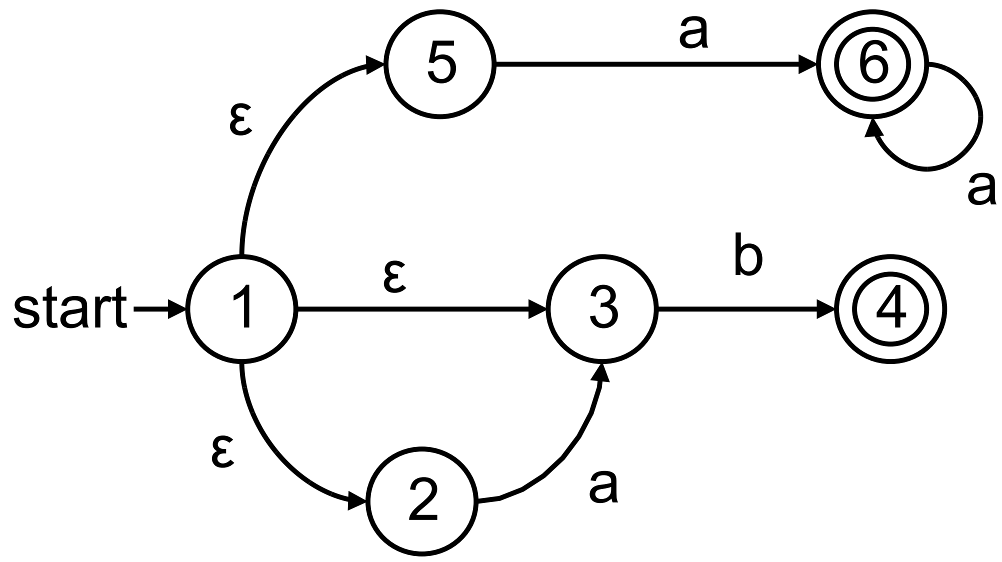
NFA to DFA (3)

- ε transitions
  - Any state reachable by an ε transition is “part of the state”
  - ε-closure - Any state reachable from S by ε transitions is in the ε-closure; treat ε-closure as 1 big state, always include ε-closure as part of the state

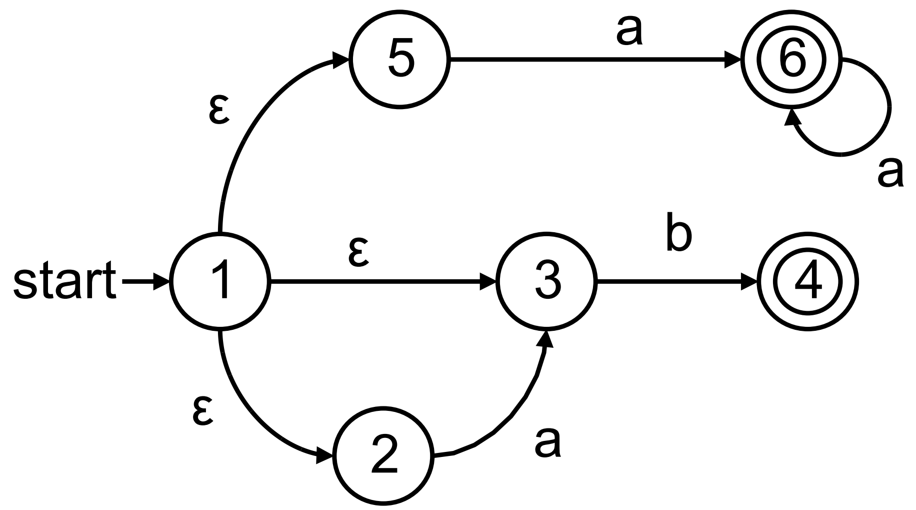


1. ε-closure(1) = {1,2,3}; include1/2/3
2. Move(1/2/3, a) = {2, 3} + ε-closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} + ε-closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} + ε-closure(2) = {2,3}
5. Move(2/3, b) = {3} + ε-closure(3) = {3}
6. Move(3, b) = {3} + ε-closure(3) = {3}

NFA to DFA - Example



NFA to DFA - Example



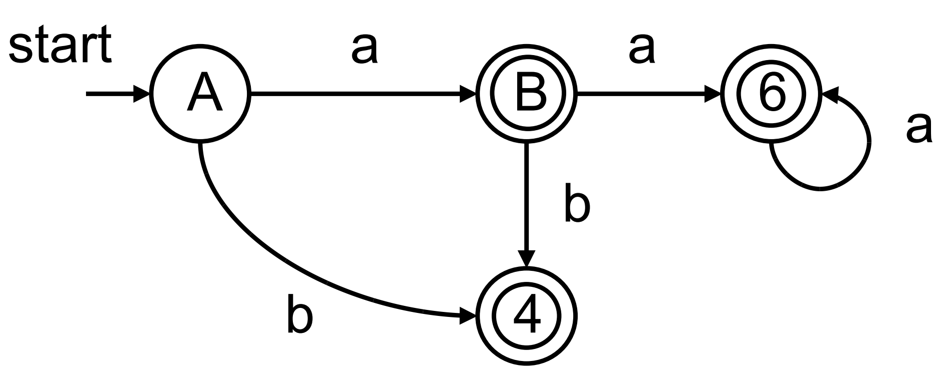
ε-closure(1) = {1, 2, 3, 5}

Create a new state A = {1, 2, 3, 5}

move(A, a) = {3, 6} + ε-closure(3,6) = {3,6}

Create B = {3,6}

move(A, b) = {4} + ε-closure(4) = {4}



move(B, a) = {6} + ε-closure(6) = {6}

move(B, b) = {4} + ε-closure(4) = {4}

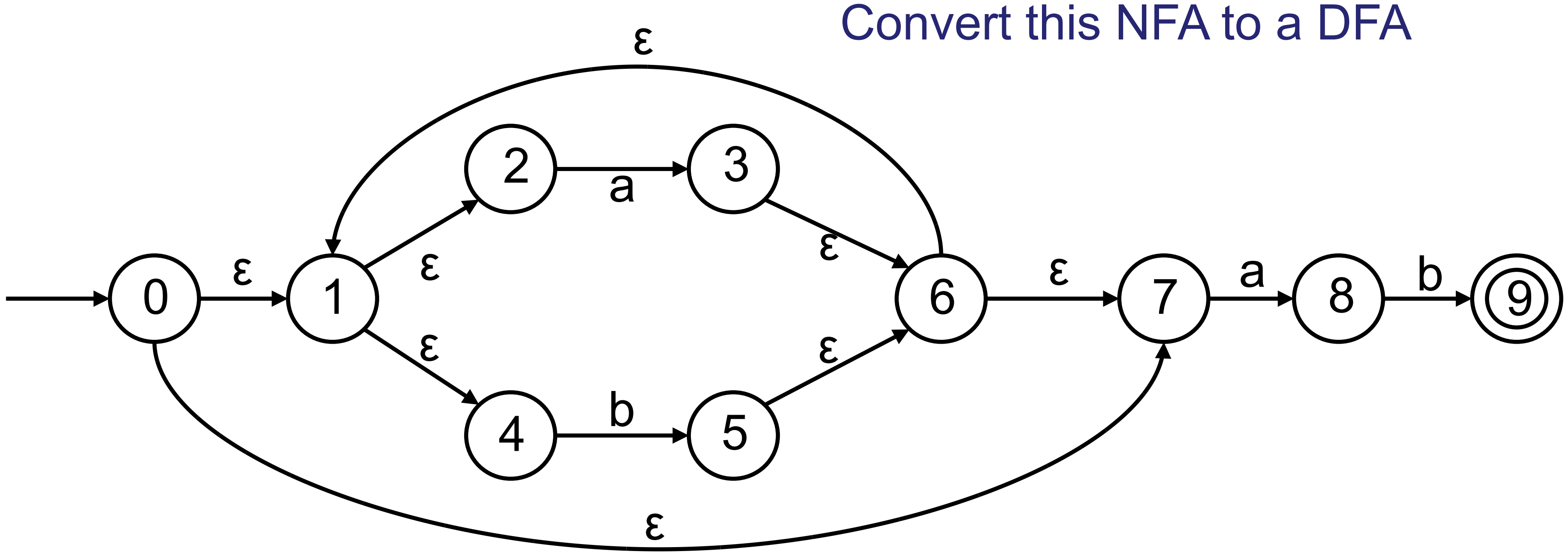
move(6, a) = {6} + ε-closure(6) = {6}

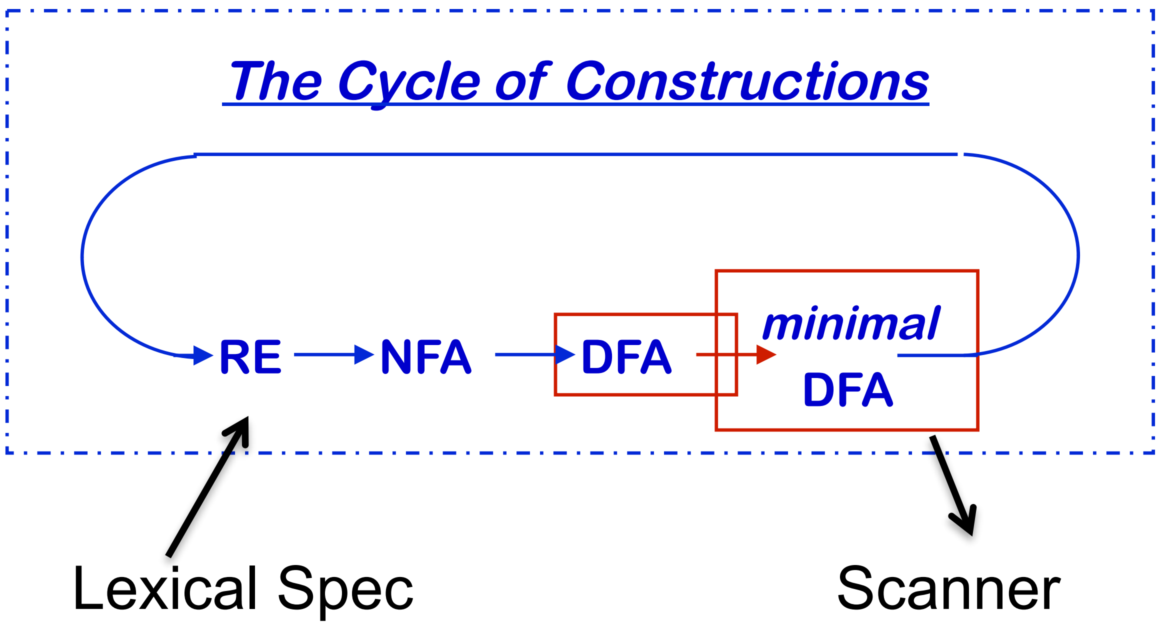
move(6, b) → ERROR

move(4, a|b) → ERROR

Class Problem

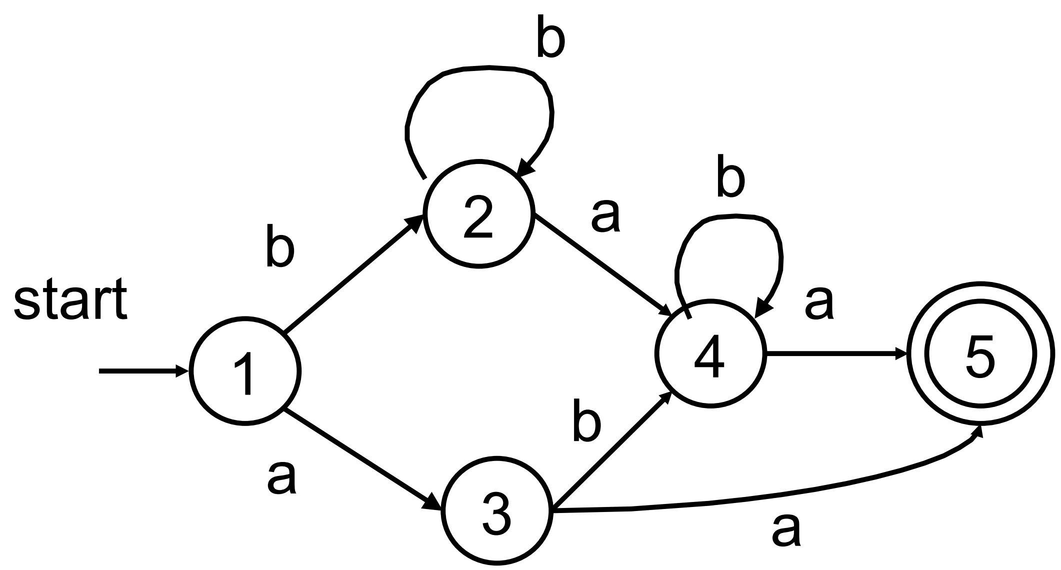
Convert this NFA to a DFA

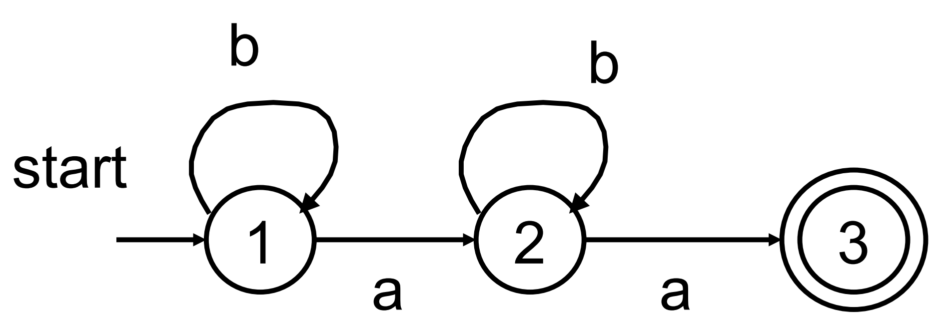



The Cycle of Constructions
RE
NFA
DFA
minimal
DFA
Lexical Spec
Scanner

State Minimization

- Resulting DFA can be quite large
  - Contains redundant or equivalent states

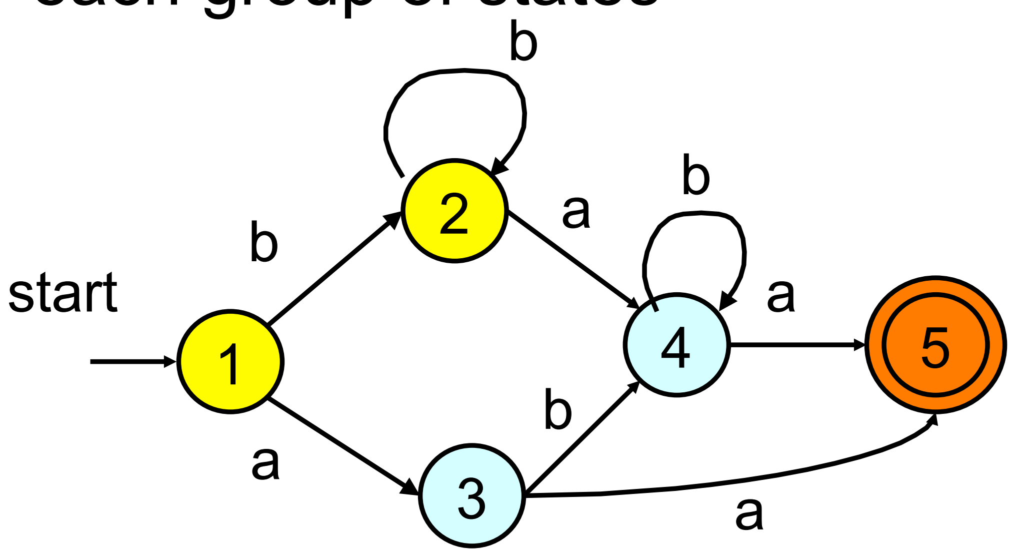




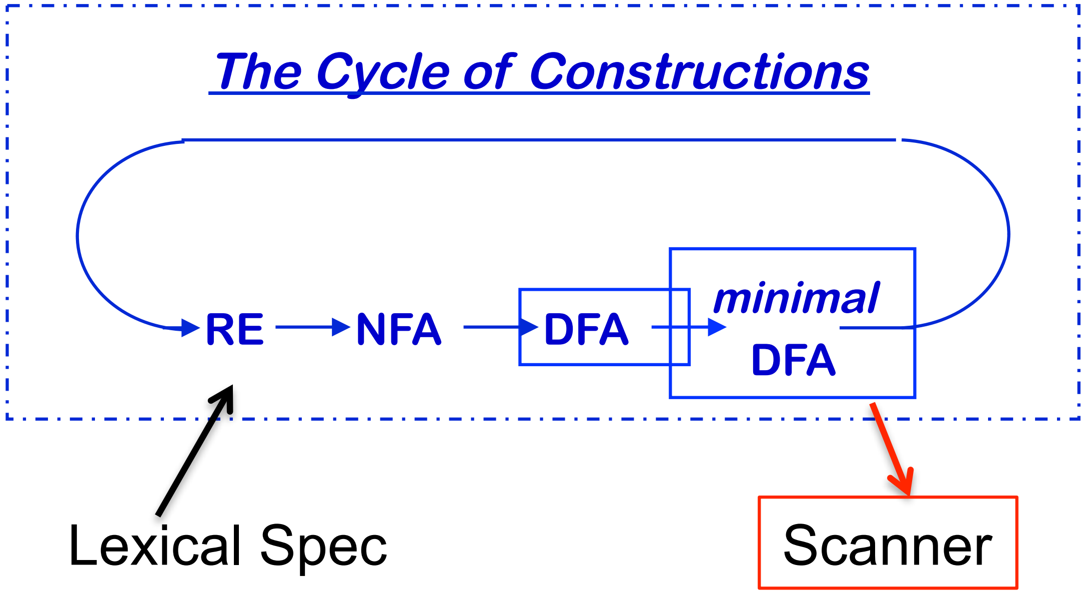
Both DFAs accept
b*ab*a

State Minimization (2)

- Idea – find groups of equivalent states and merge them
  - All transitions from states in group G1 go to states in another group G2
  - Construct minimized DFA such that there is 1 state for each group of states



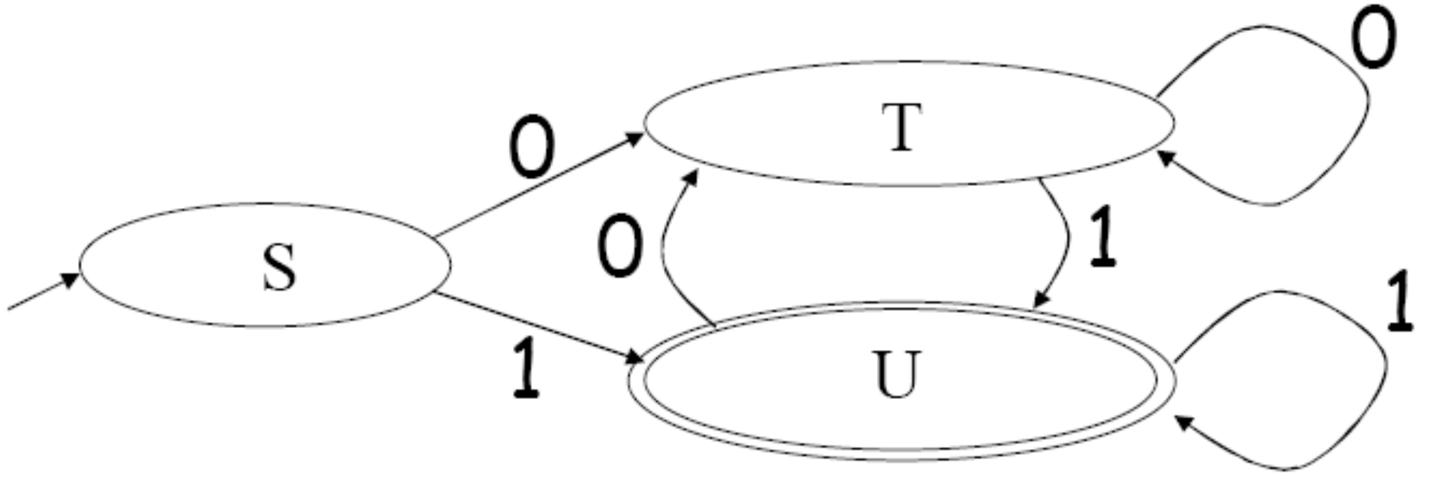
Basic strategy: identify distinguishing transitions


The Cycle of Constructions
RE
NFA
DFA
minimal DFA
Lexical Spec
Scanner

DFA Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition Si $\rightarrow^{a}$ Sk define T[i,a] = k
- DFA “execution”
  - If in state Si and input a, read T[i,a] = k and skip to state Sk
  - Very efficient

DFA Table Implementation : Example



	0	1
S	T	U
T	T	U
U	T	U

Implementation Cont ..

- NFA -> DFA conversion is at the heart of tools such as flex

- But, DFAs can be huge

- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations