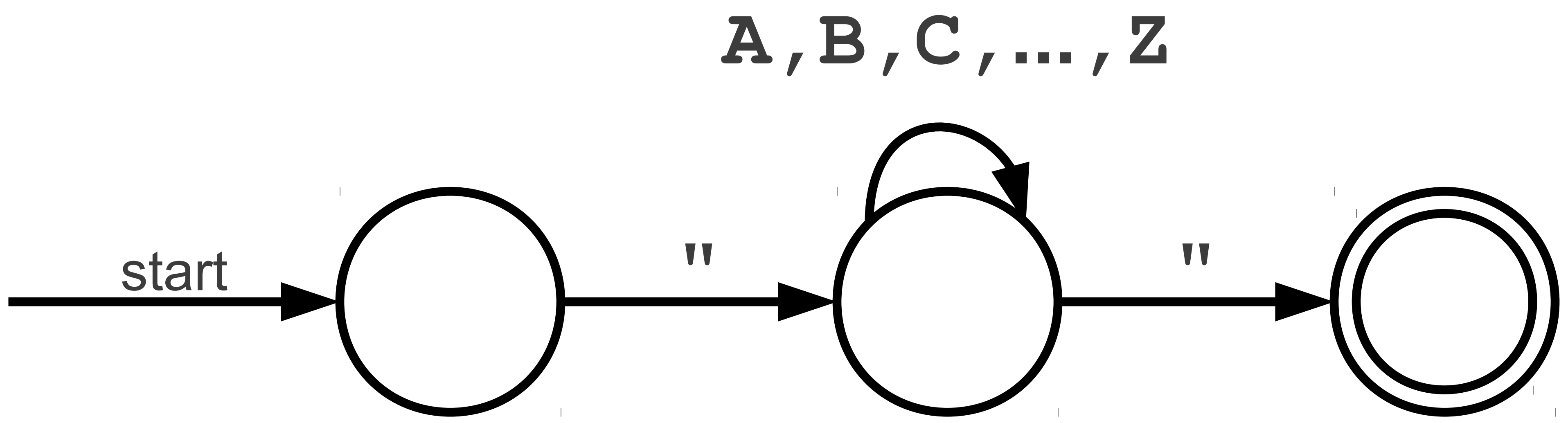
Lexical Analysis 2:
Automata and Lexer Generators

Recognizing Regular Languages

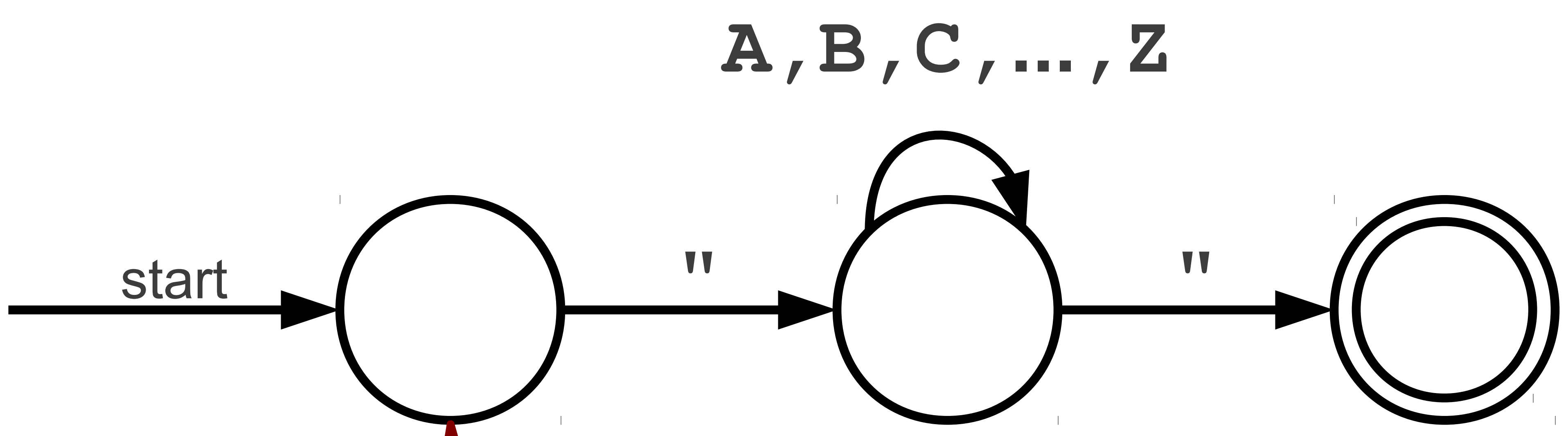
How can we efficiently implement a recognizer for a regular language?

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

A Simple Automaton

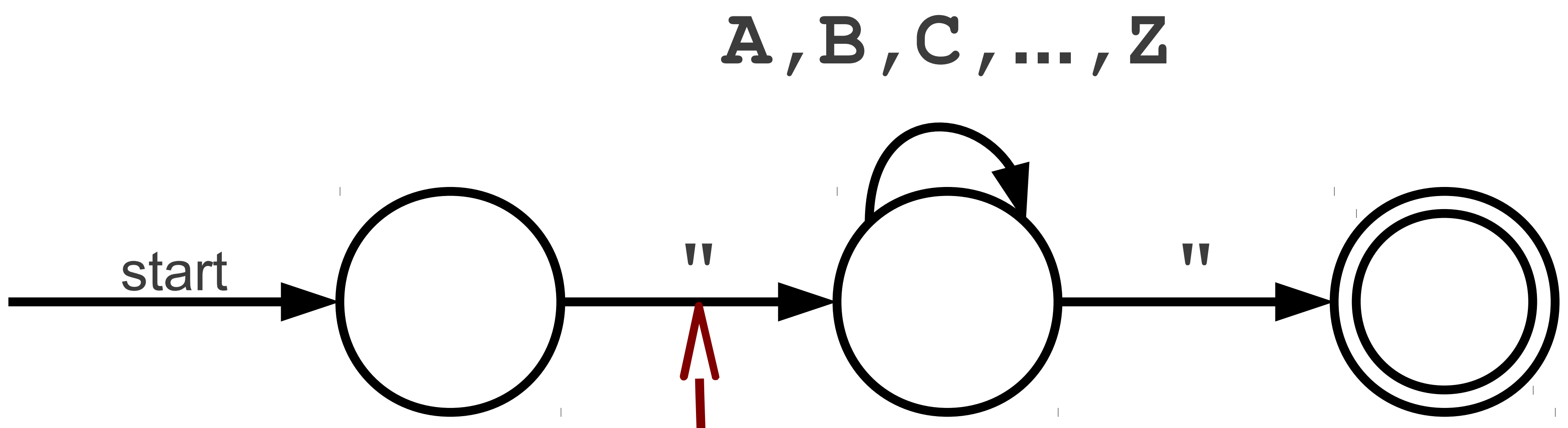


A Simple Automaton



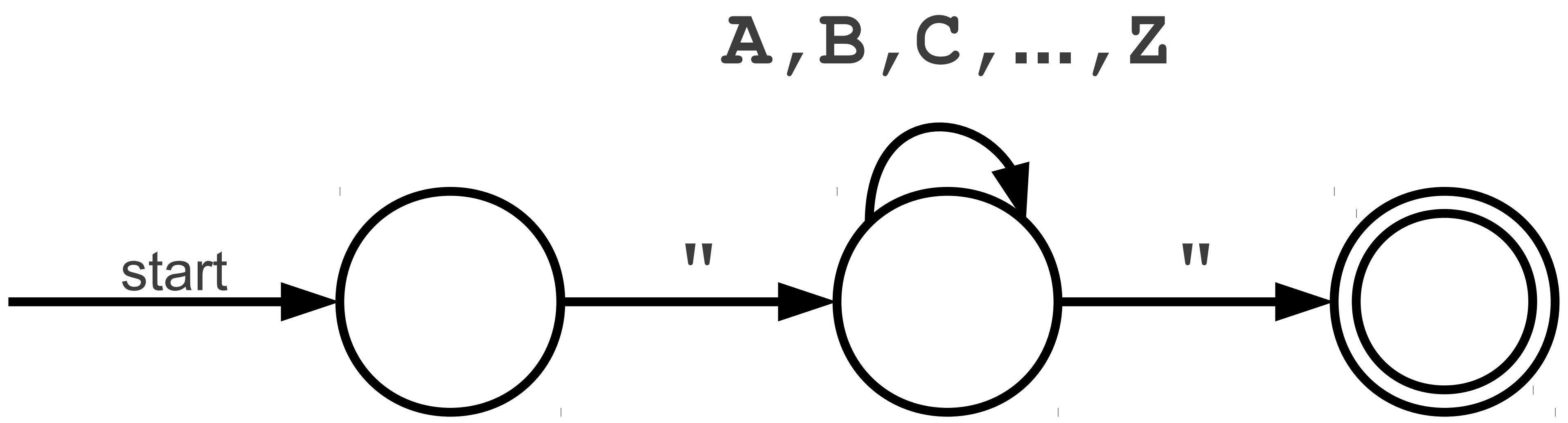
Each circle is a state of the automaton. The automaton's configuration is determined by what state(s) it is in.

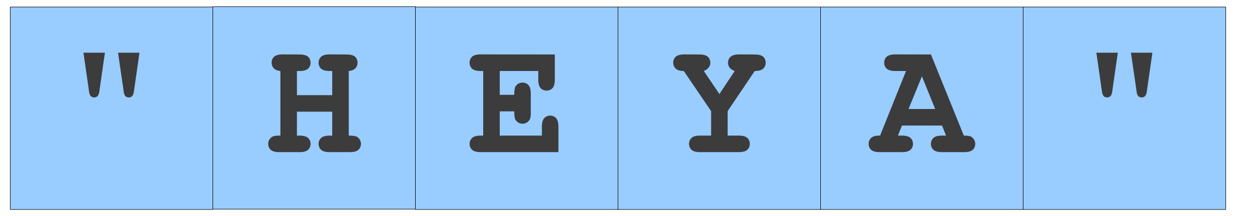
A Simple Automaton



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

A Simple Automaton

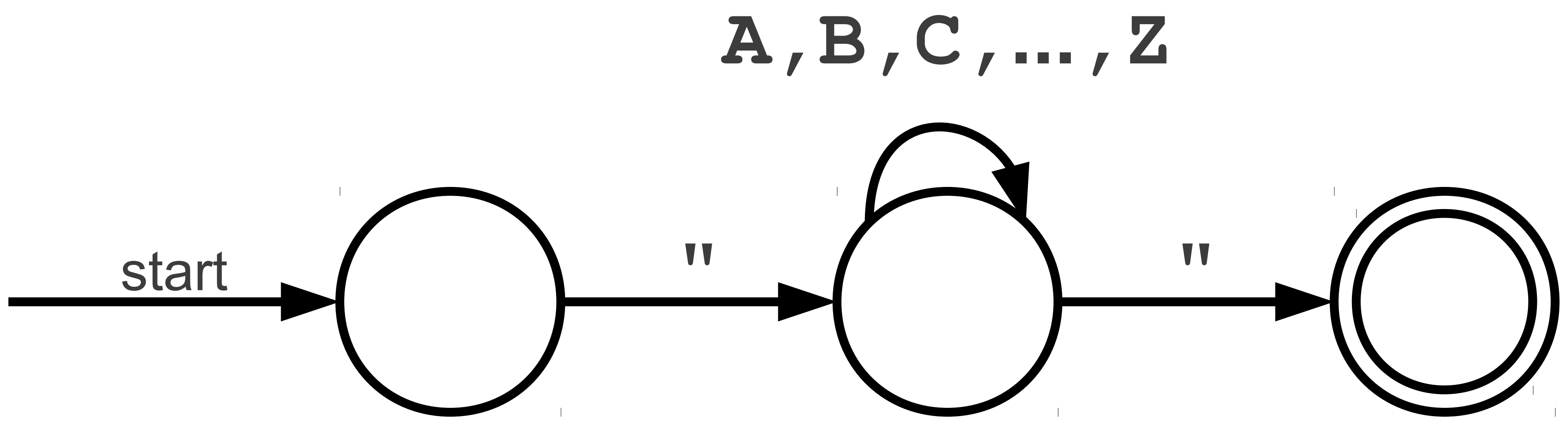




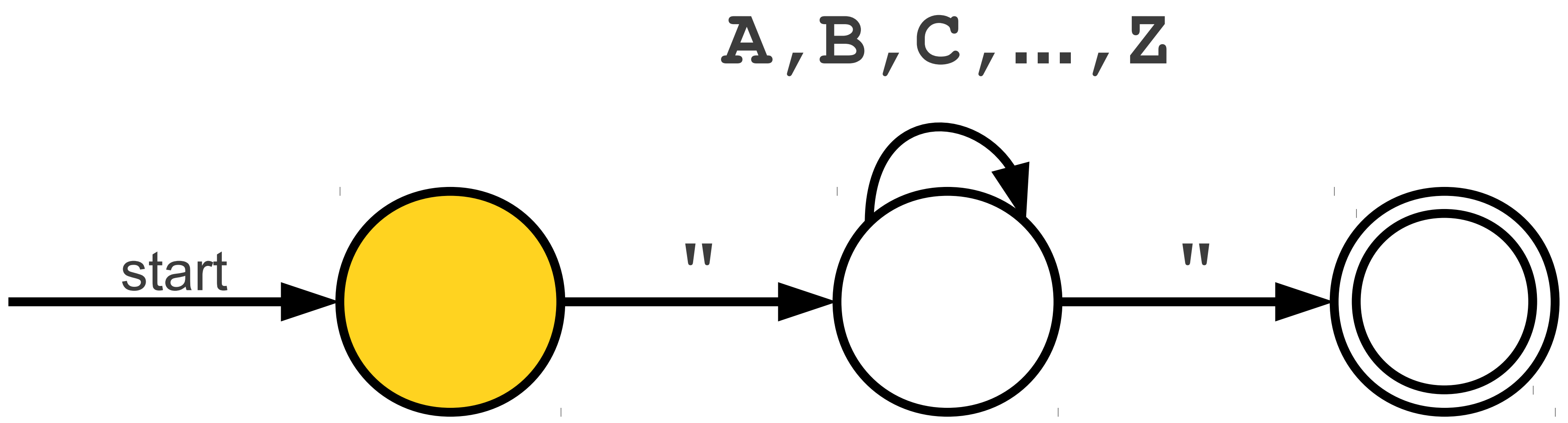
Finite Automata: Takes an input string and determines whether it's a valid sentence of a language

accept or reject

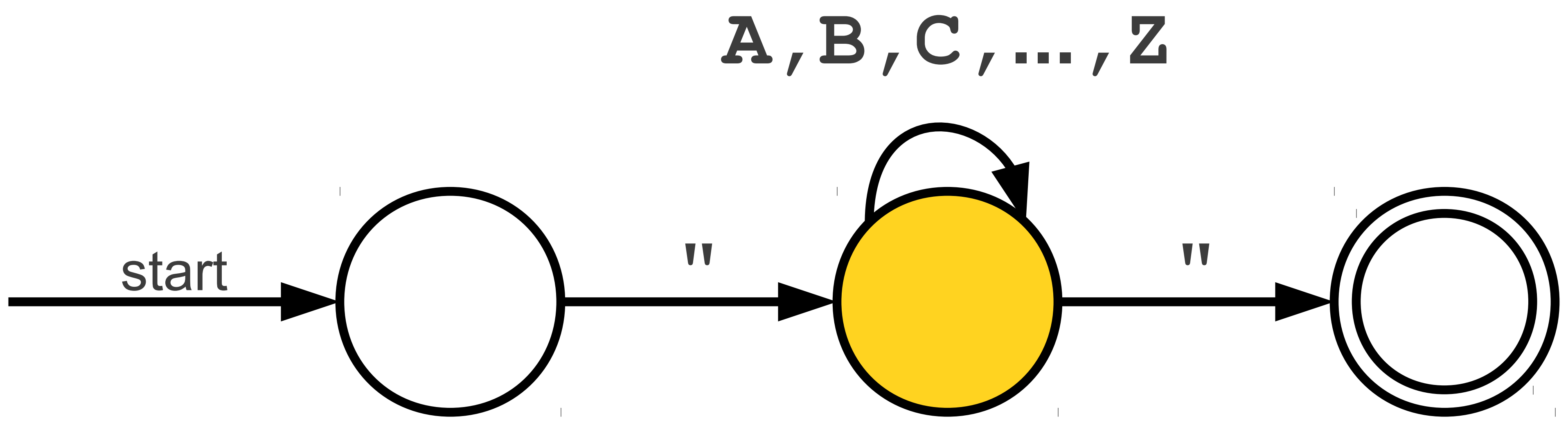
A Simple Automaton



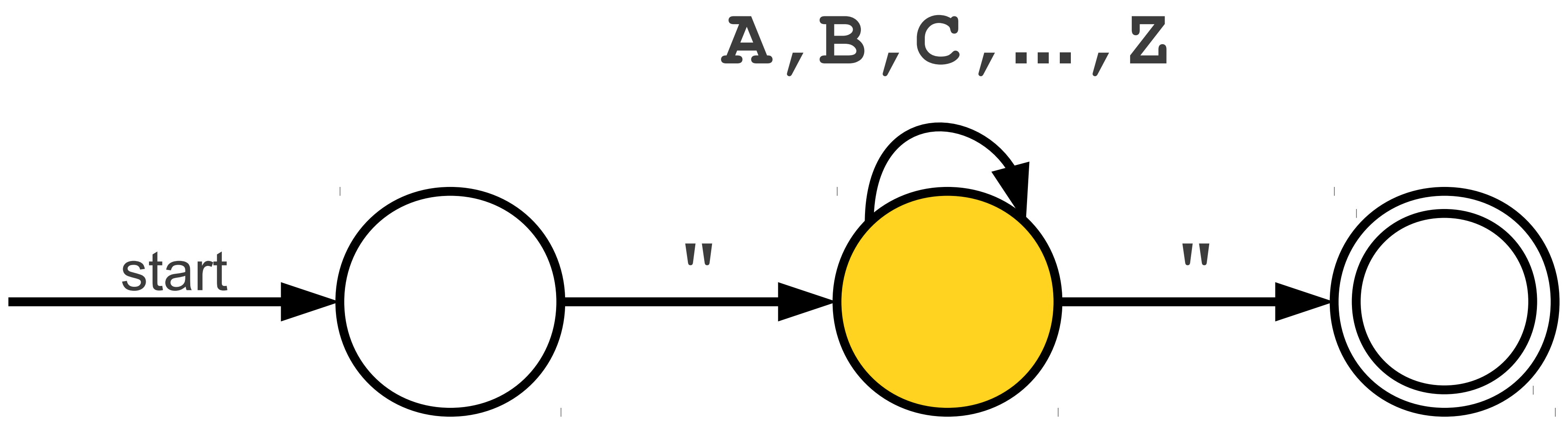
A Simple Automaton

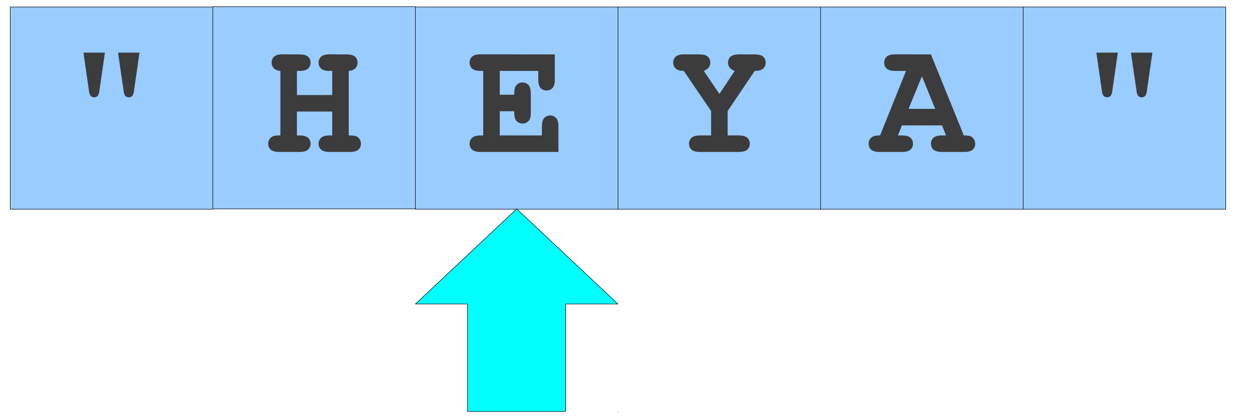


A Simple Automaton

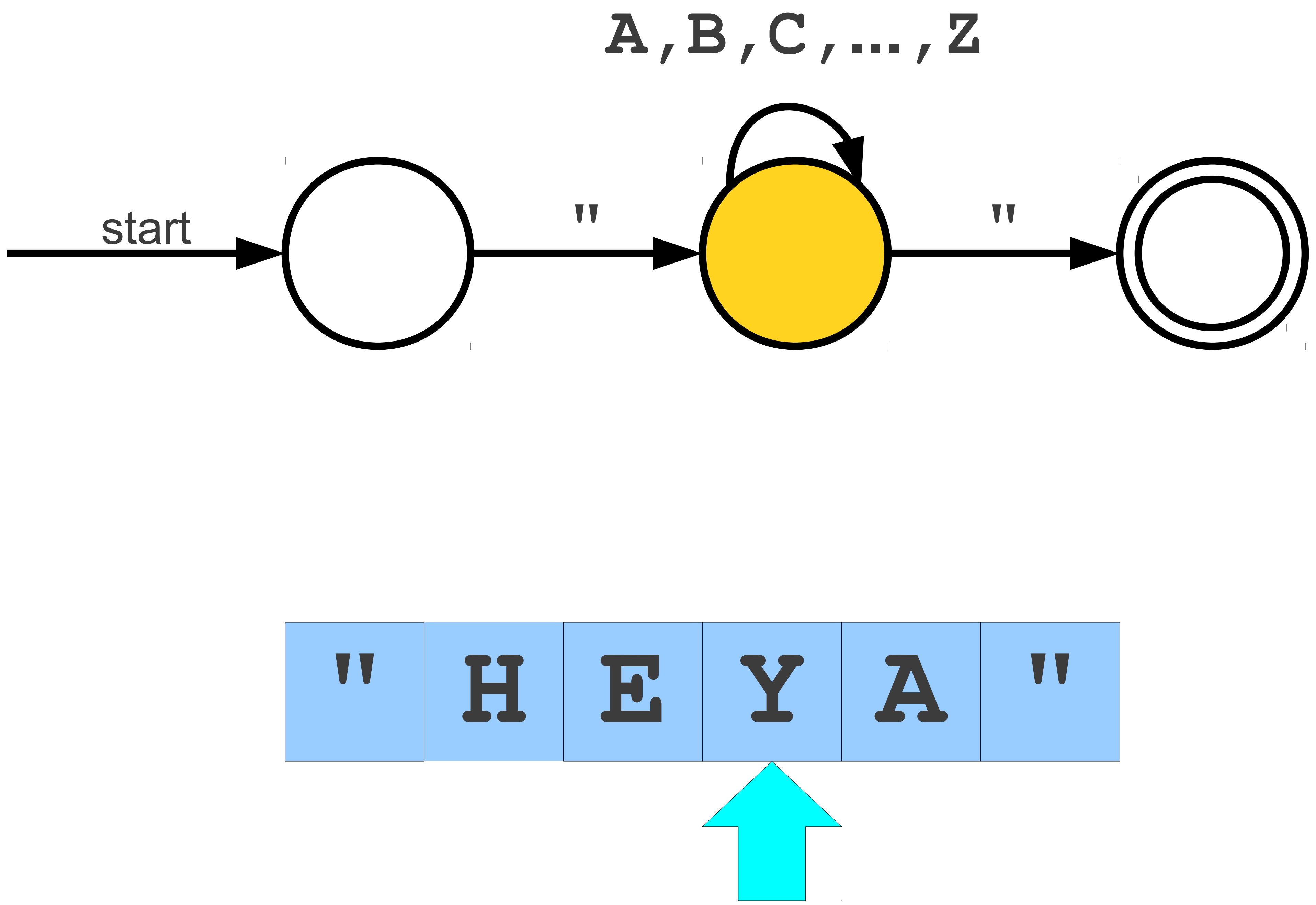


A Simple Automaton

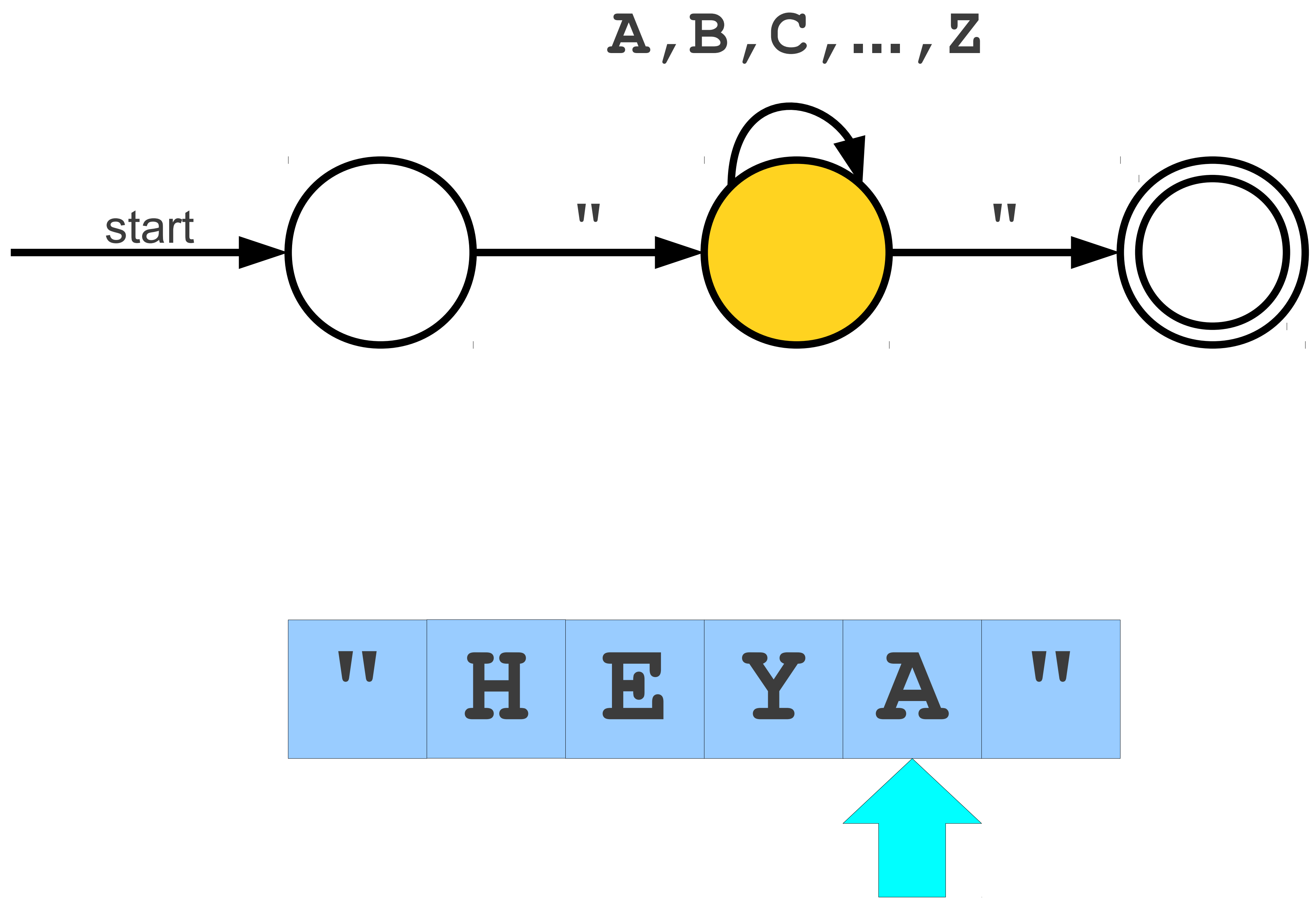




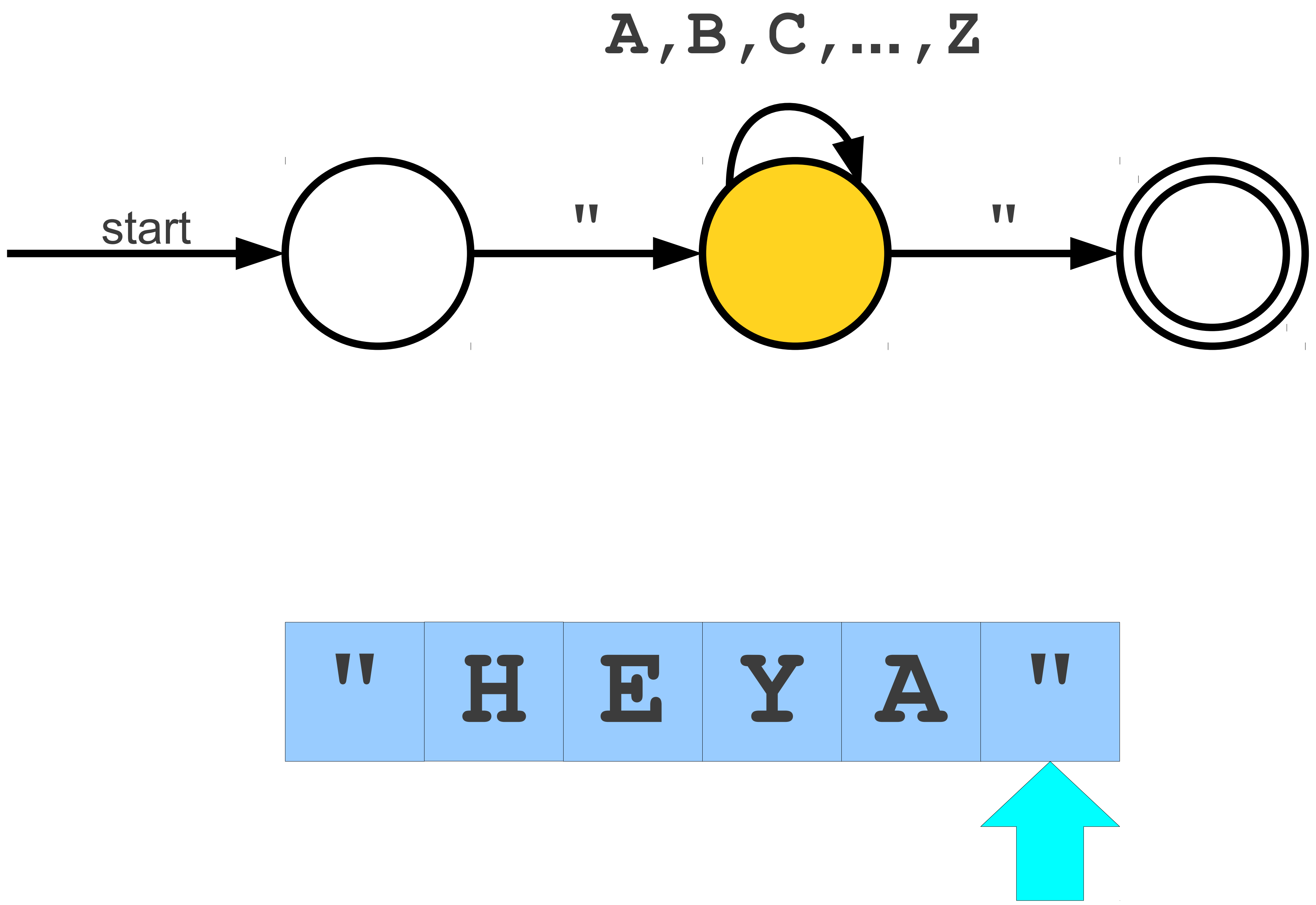
A Simple Automaton



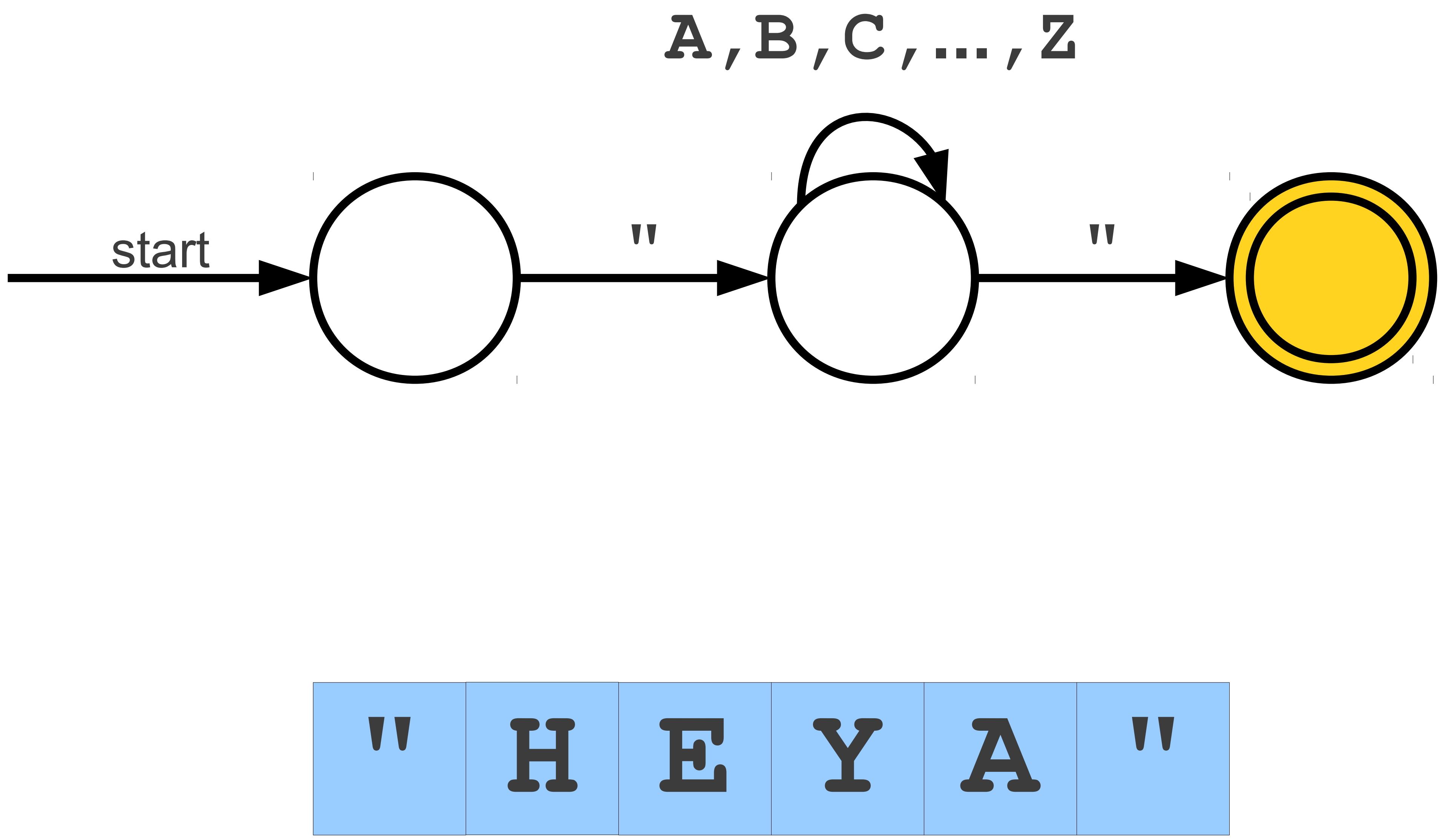
A Simple Automaton



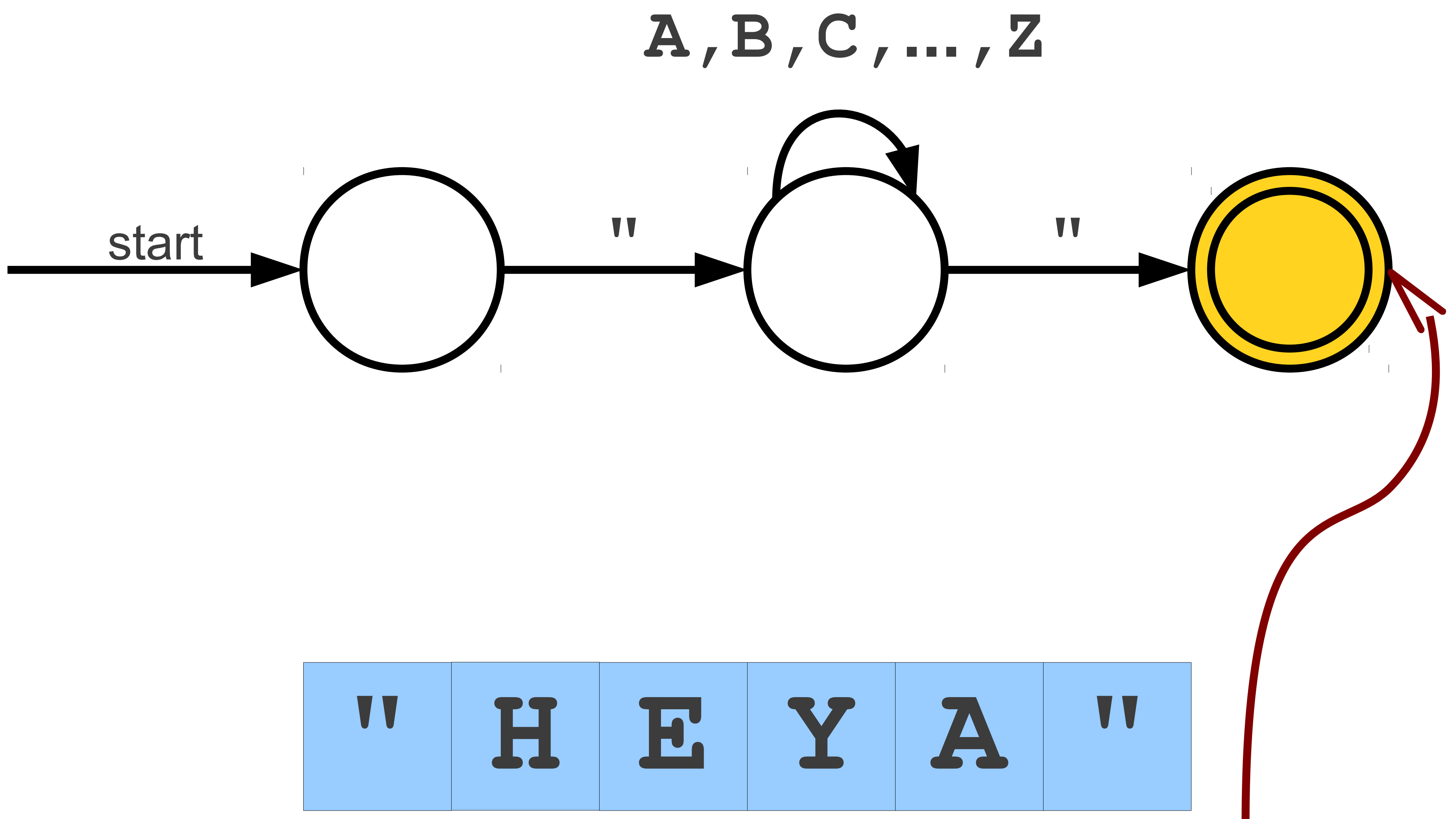
A Simple Automaton



A Simple Automaton

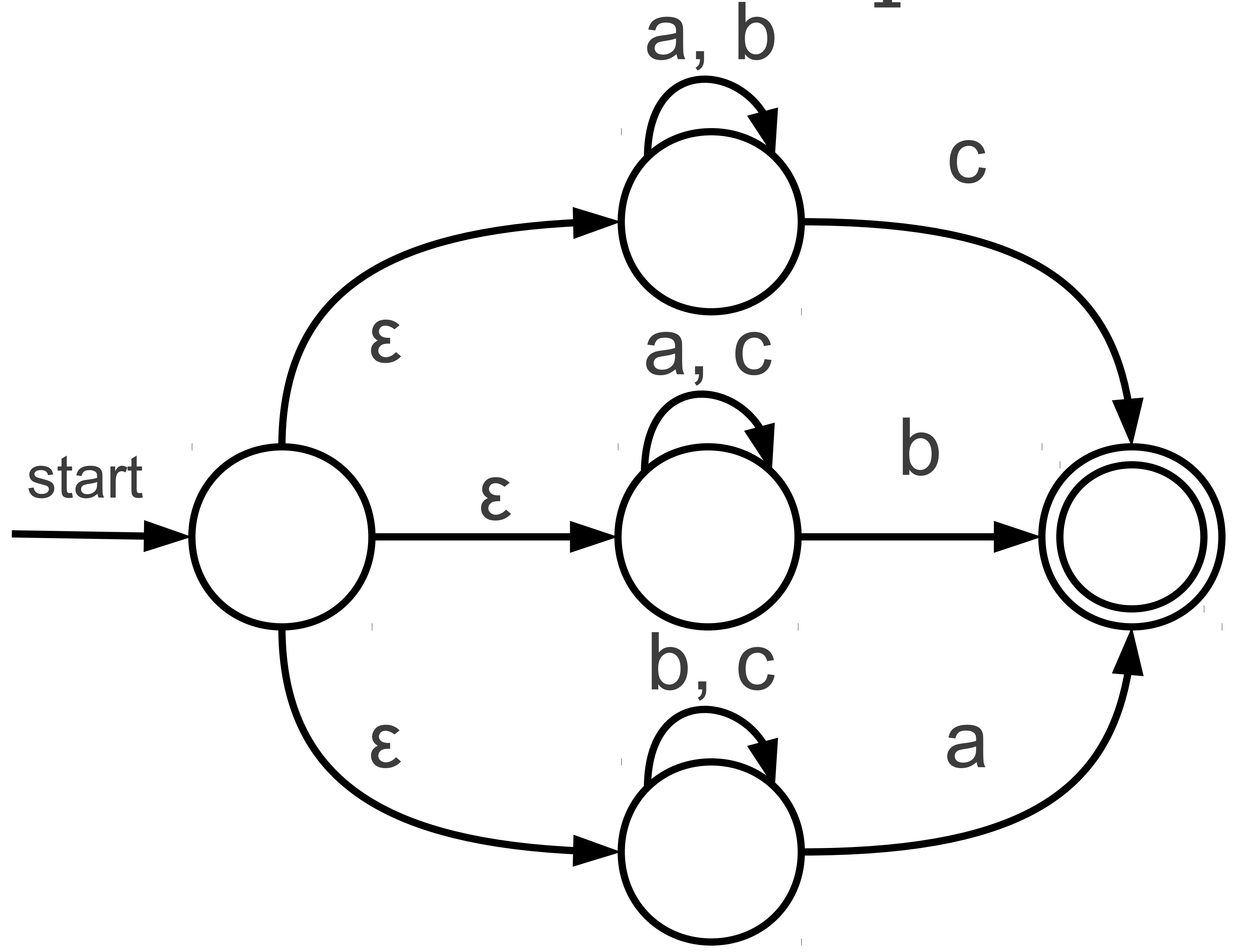


A Simple Automaton

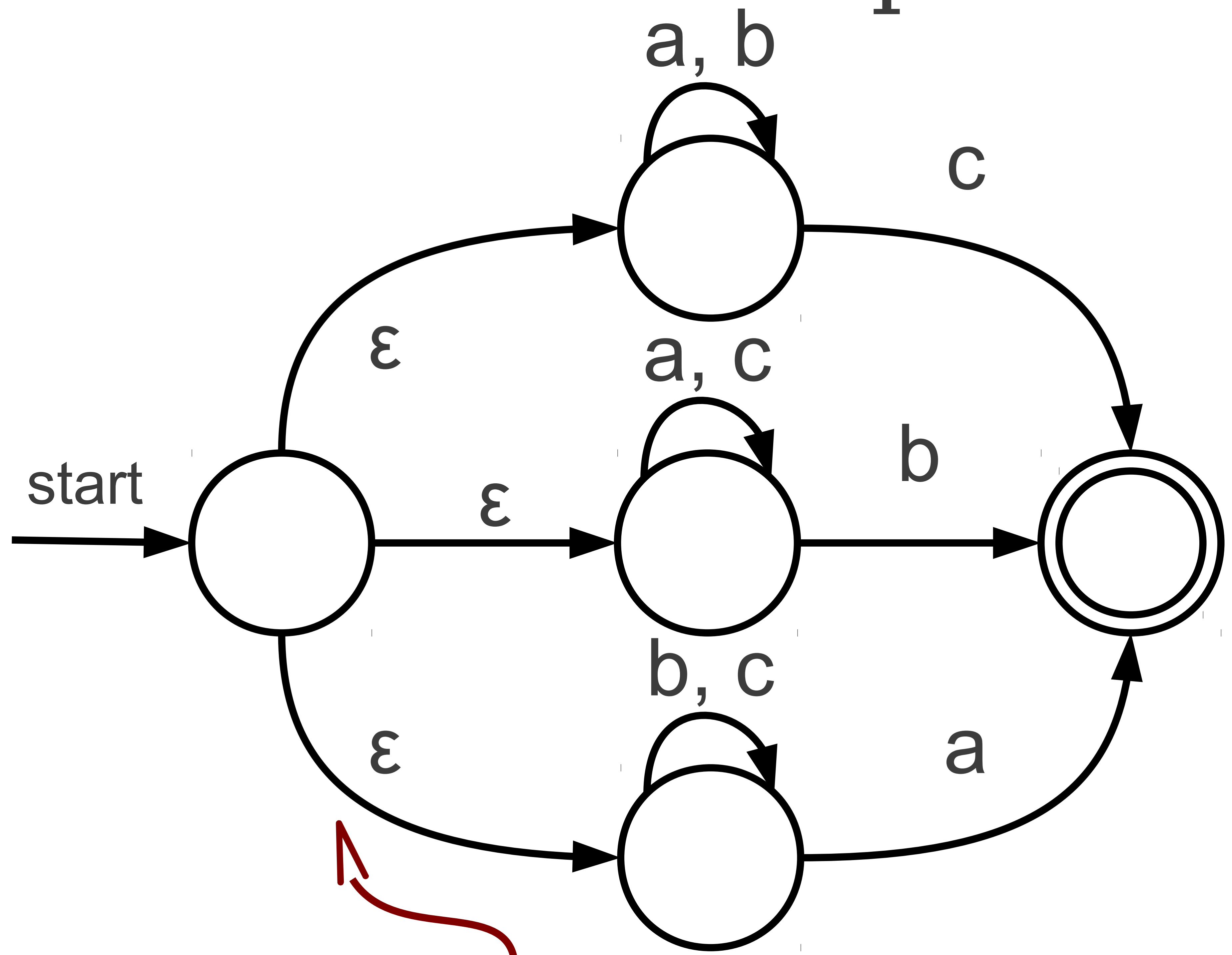


The double circle indicates that this state is an accepting state. The automaton accepts the string if it ends in an accepting state.

An Even More Complex Automaton

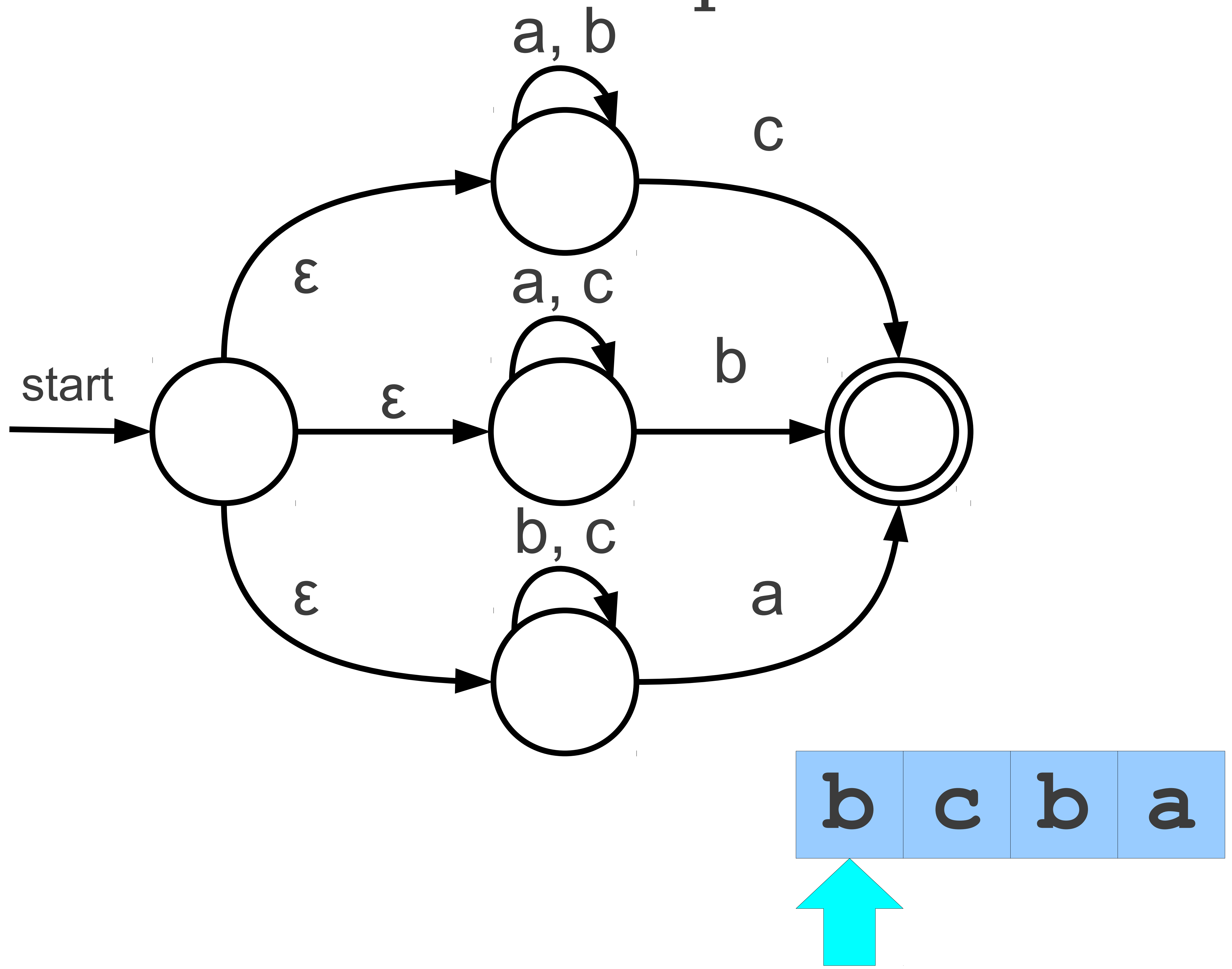


An Even More Complex Automaton

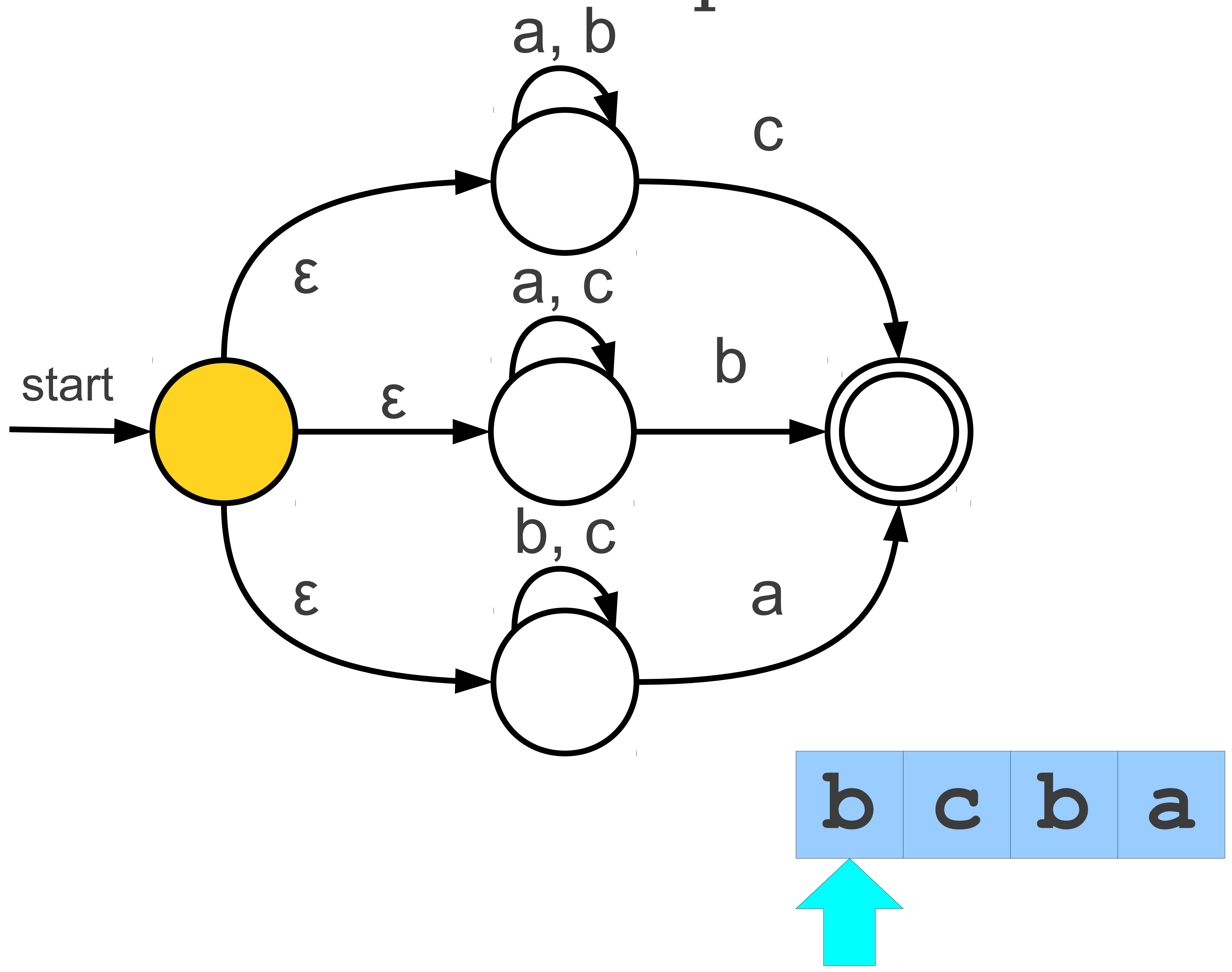


These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

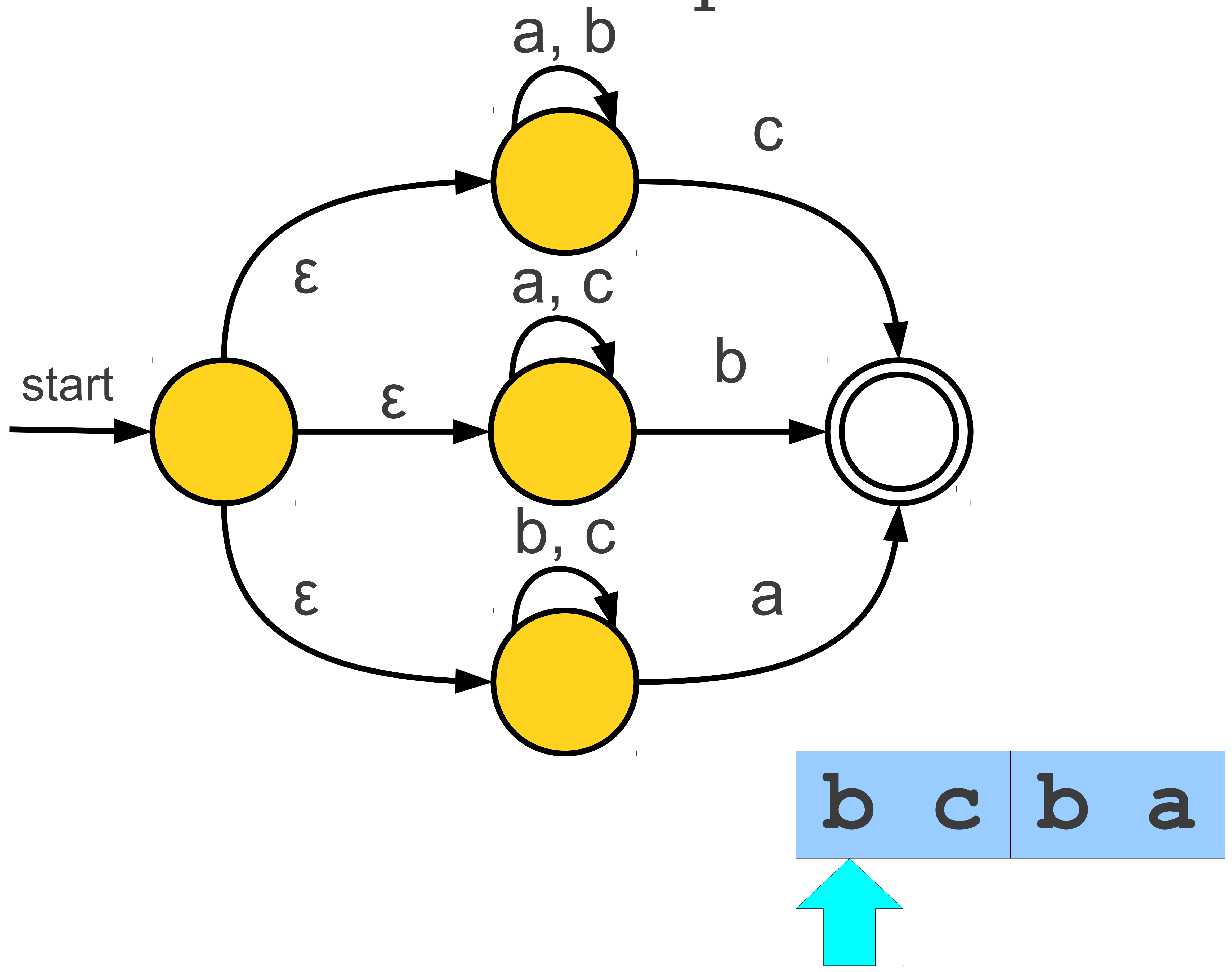
An Even More Complex Automaton



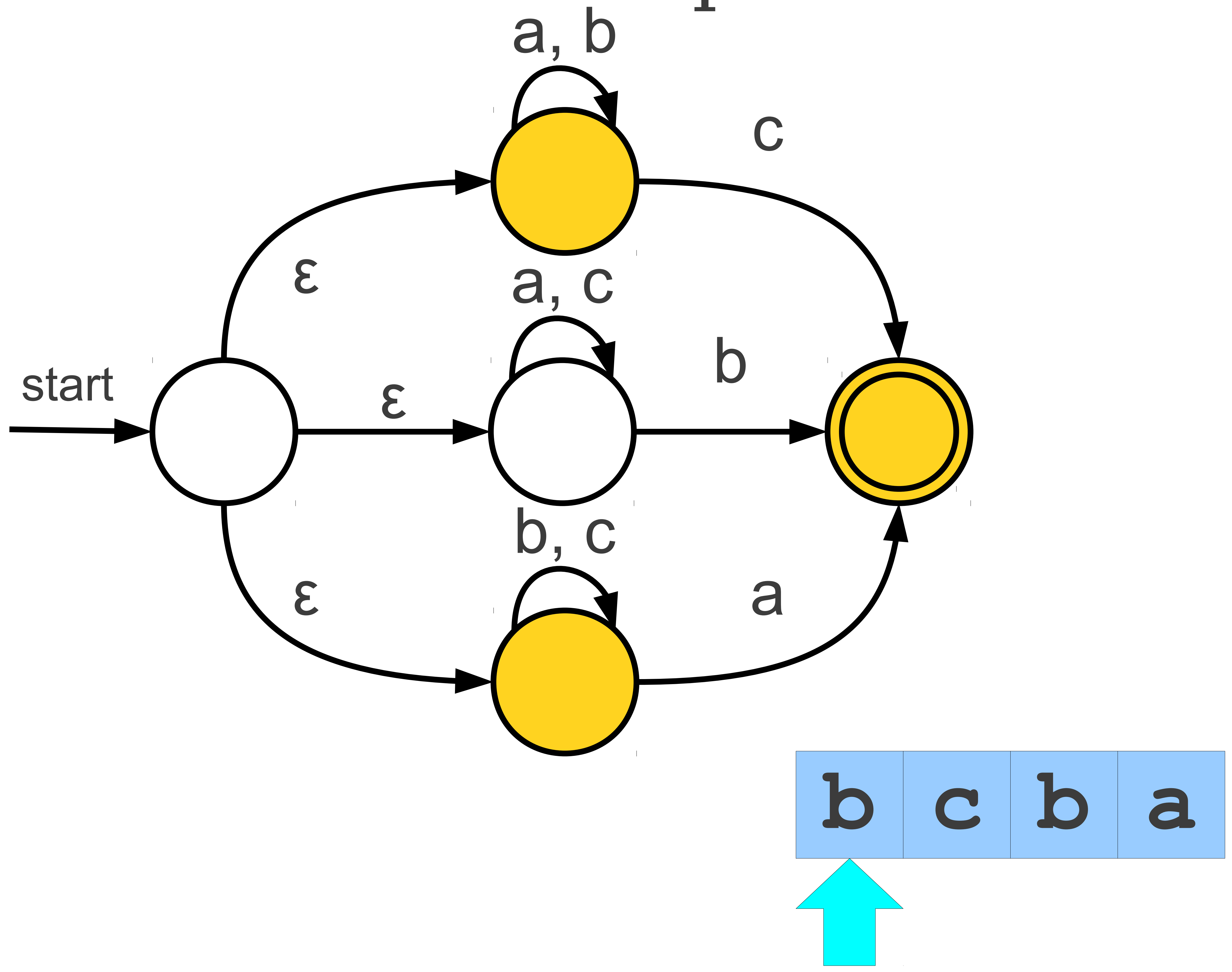
An Even More Complex Automaton



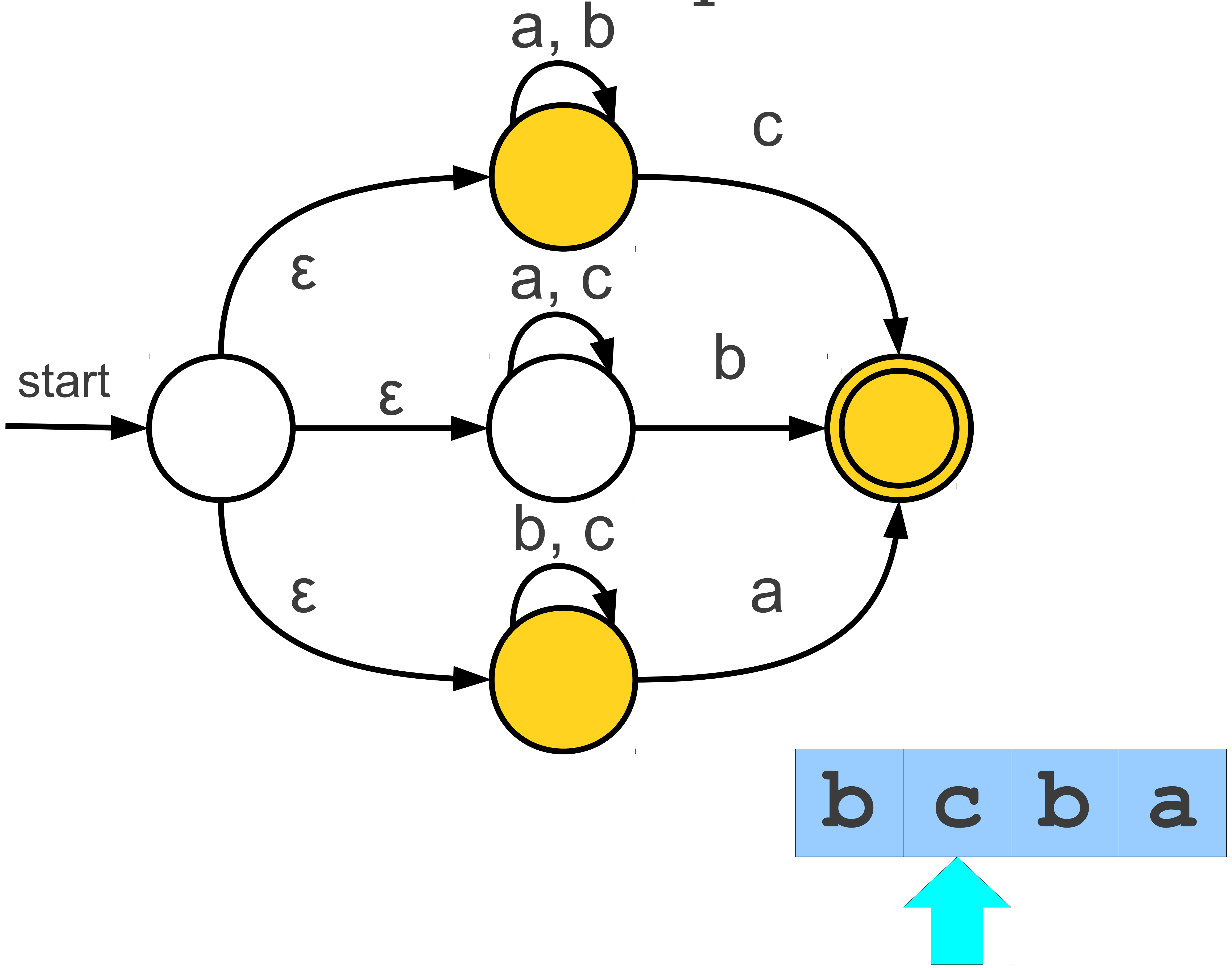
An Even More Complex Automaton



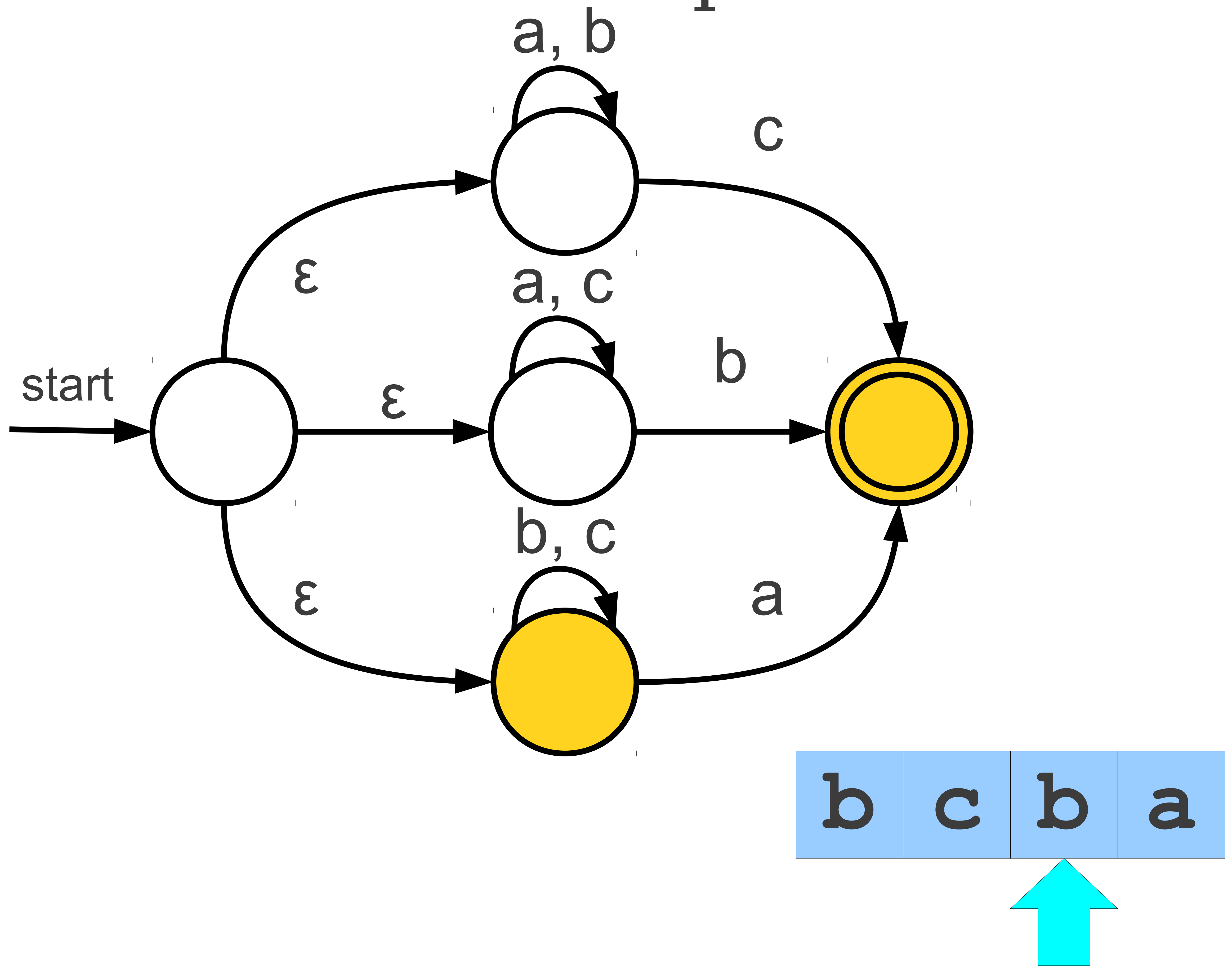
An Even More Complex Automaton



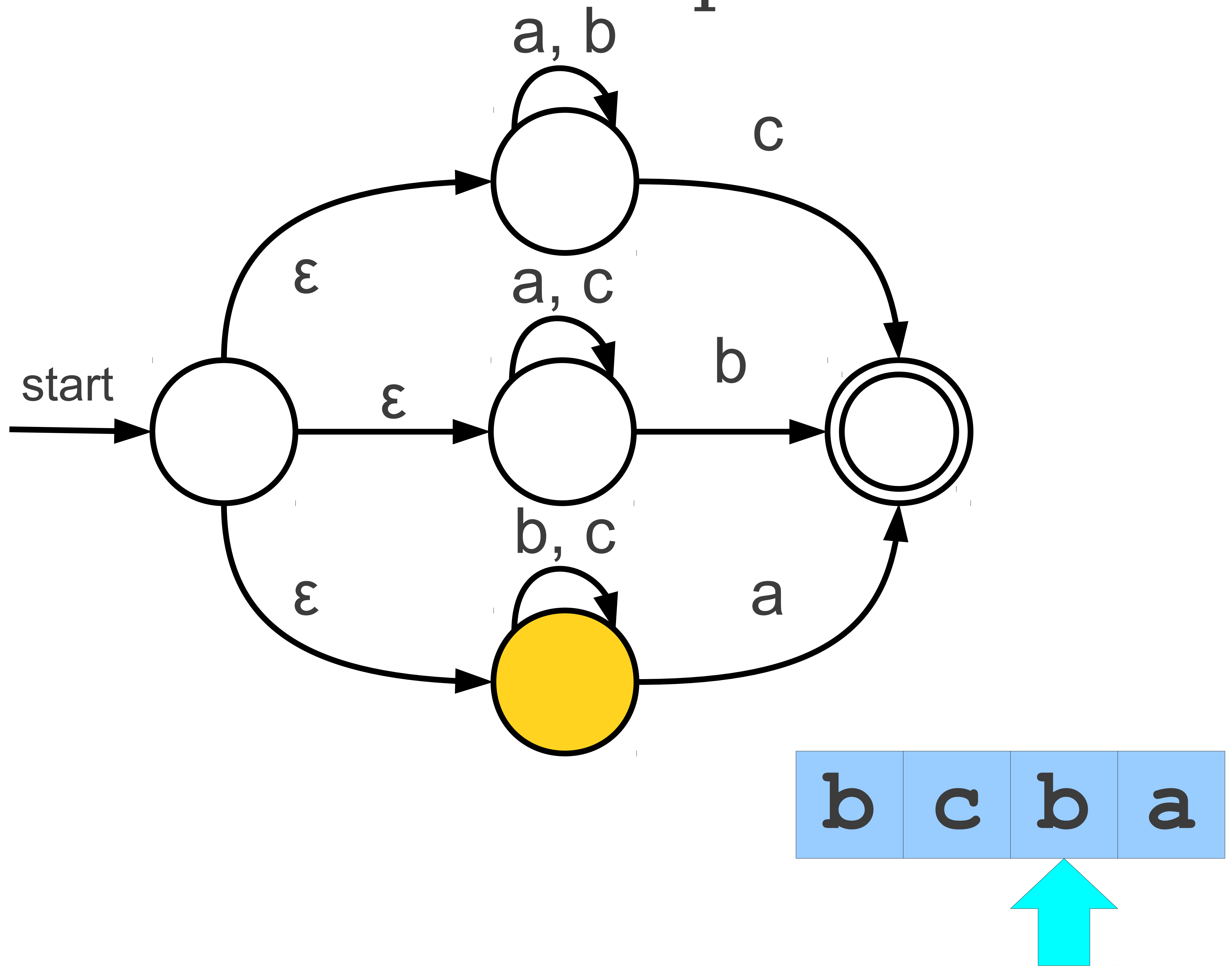
An Even More Complex Automaton



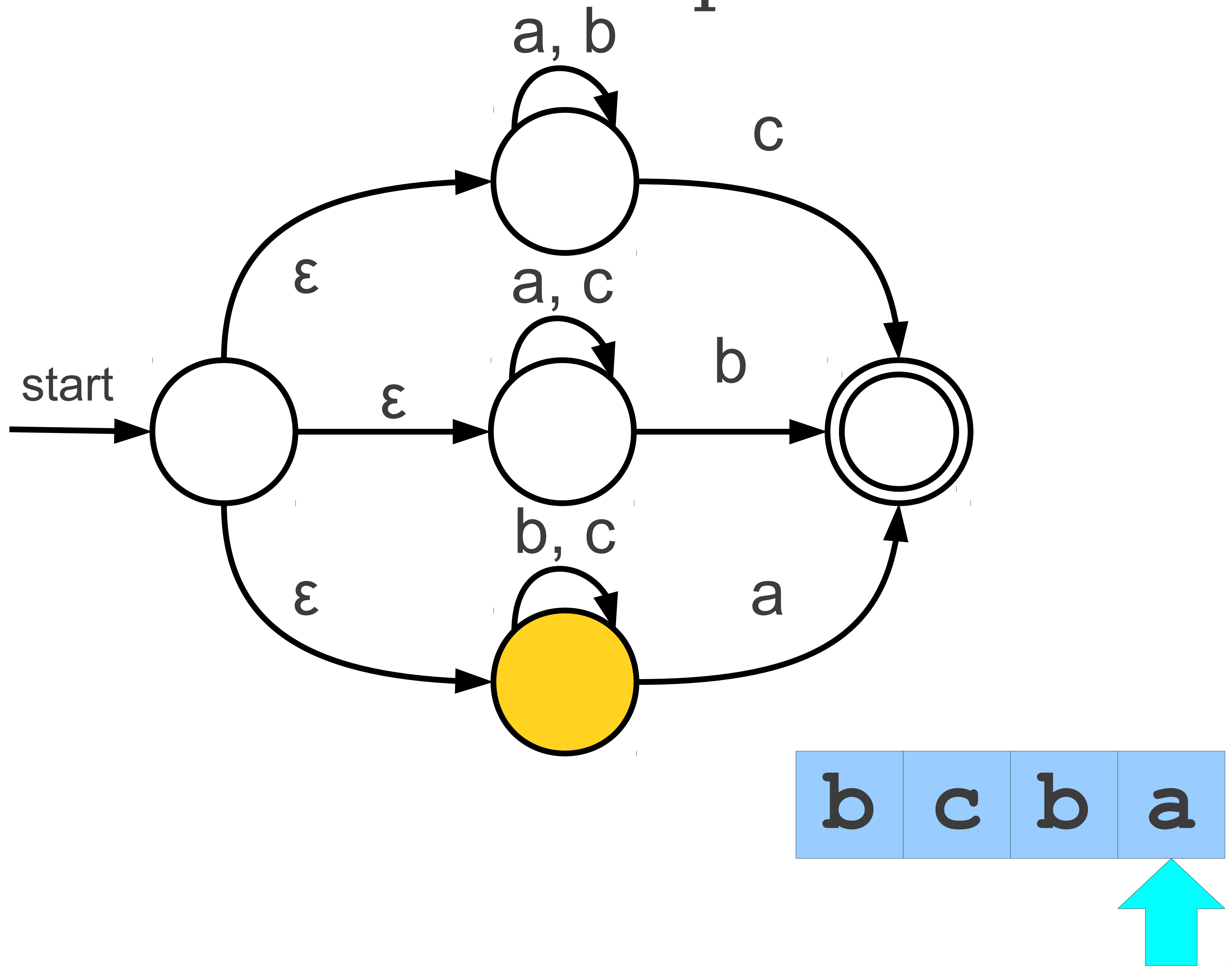
An Even More Complex Automaton



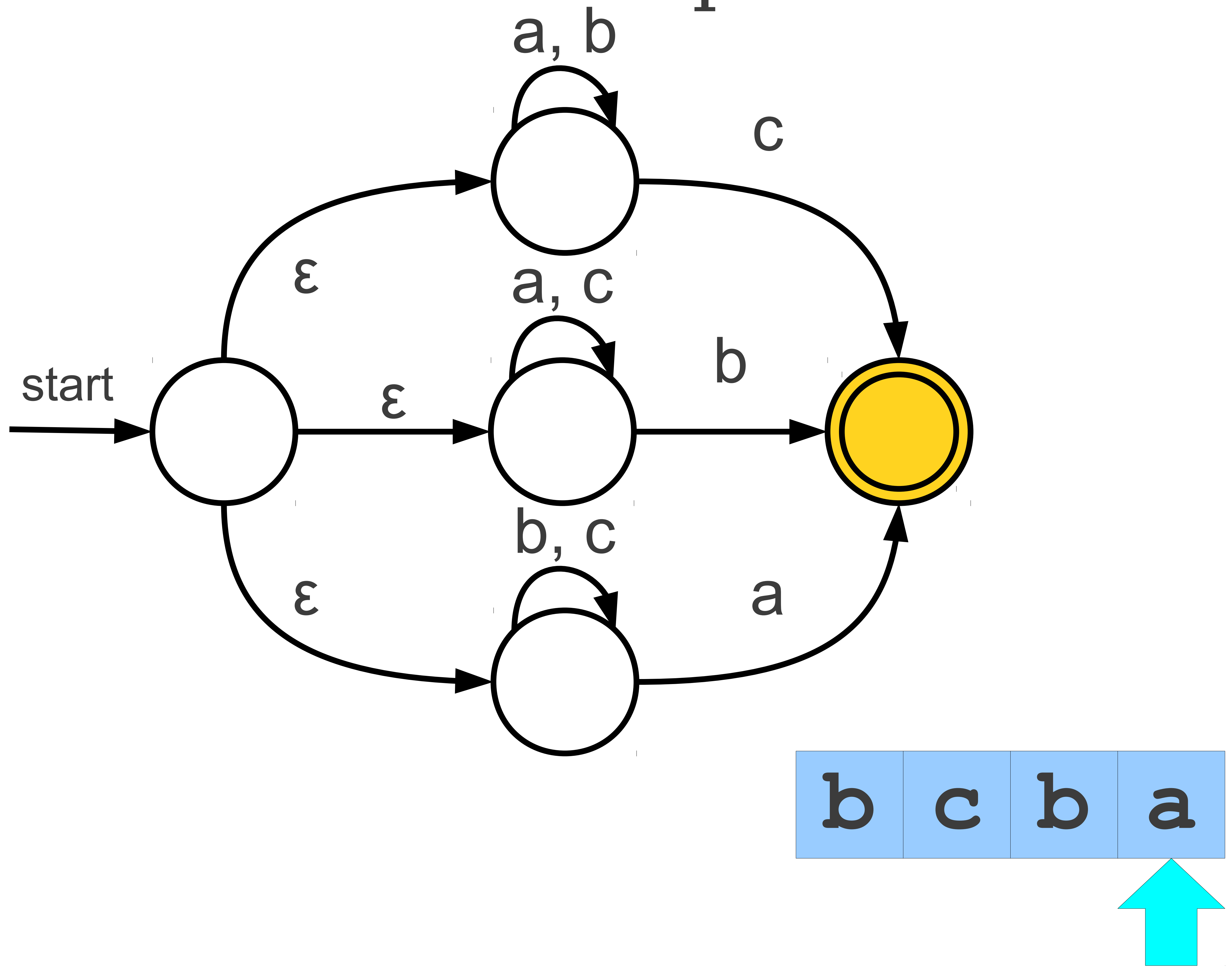
An Even More Complex Automaton



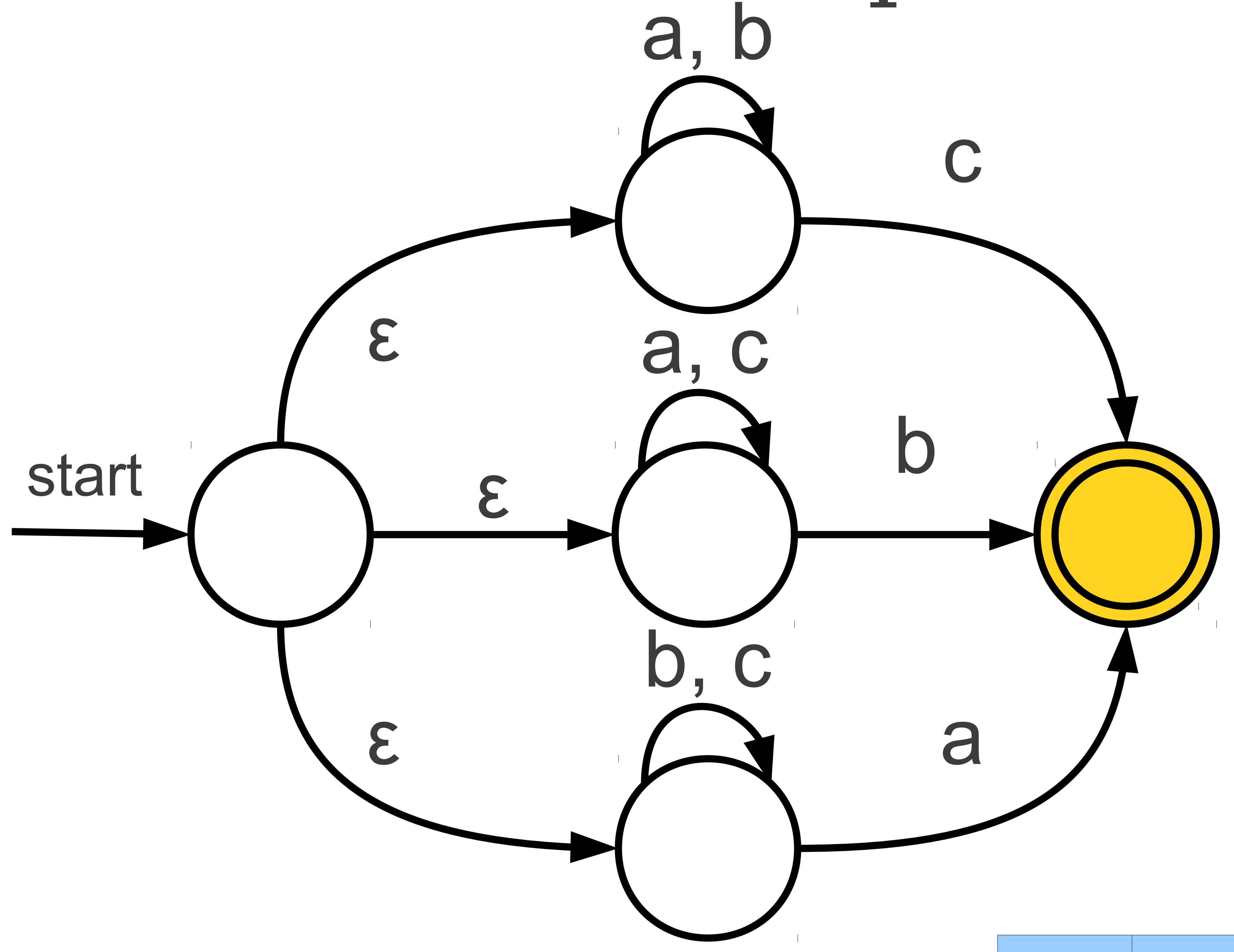
An Even More Complex Automaton



An Even More Complex Automaton



An Even More Complex Automaton



b	c	b	a

Finite State Automata

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions δ
    - $state_k$ ----> $state_j$

Finite State Automata

- Transition

  $s1 \rightarrow^{a} s2$

- A character is read

  In state s1 on input “a” go to state s2

- If end of input and in accepting state

  Accept

- Otherwise

  Reject

DFA vs. NFA

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

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

Alternative Approaches

- We'll go through the "classic" procedure above but some scanners use different approaches:
    - Brzozowski: use the "derivative" operation on languages to directly produce a DFA from a regexp
        - Advantage: simple to implement, extends easily to support regex conjunction, negation. Often used for regex interpreters
        - Disadvantage: computationally expensive to generate minimal DFAs

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)*

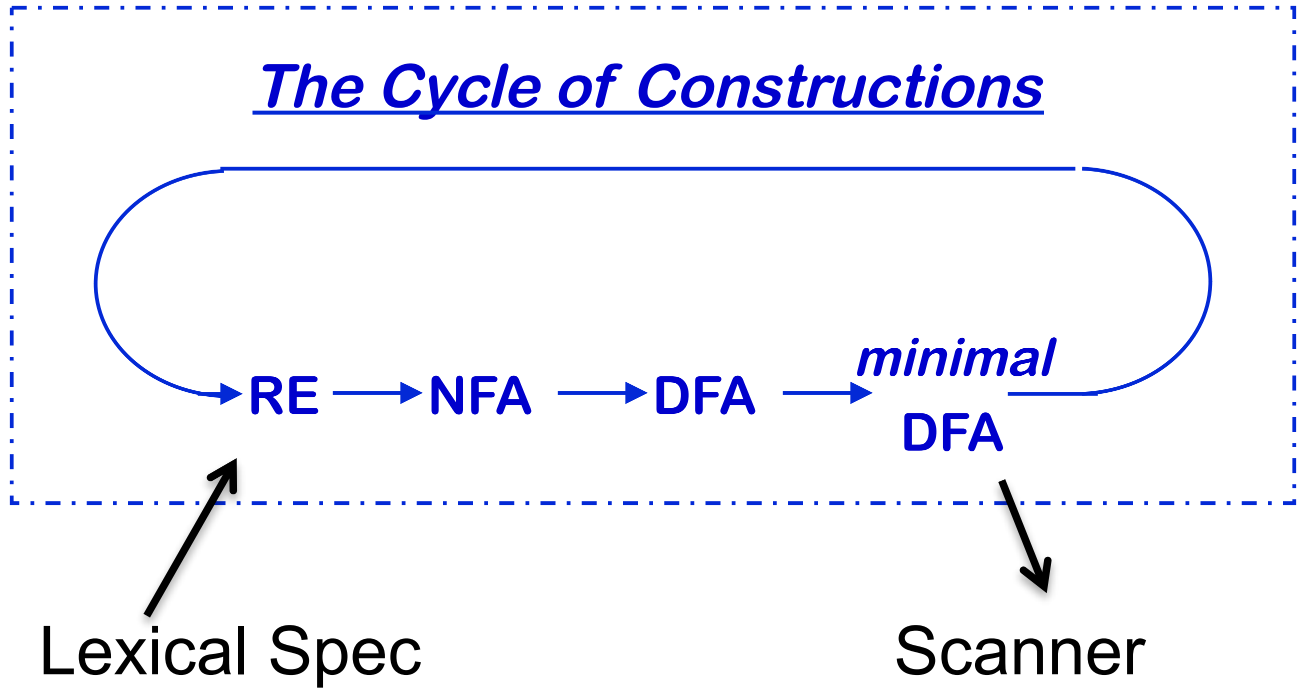
- Build the simulation

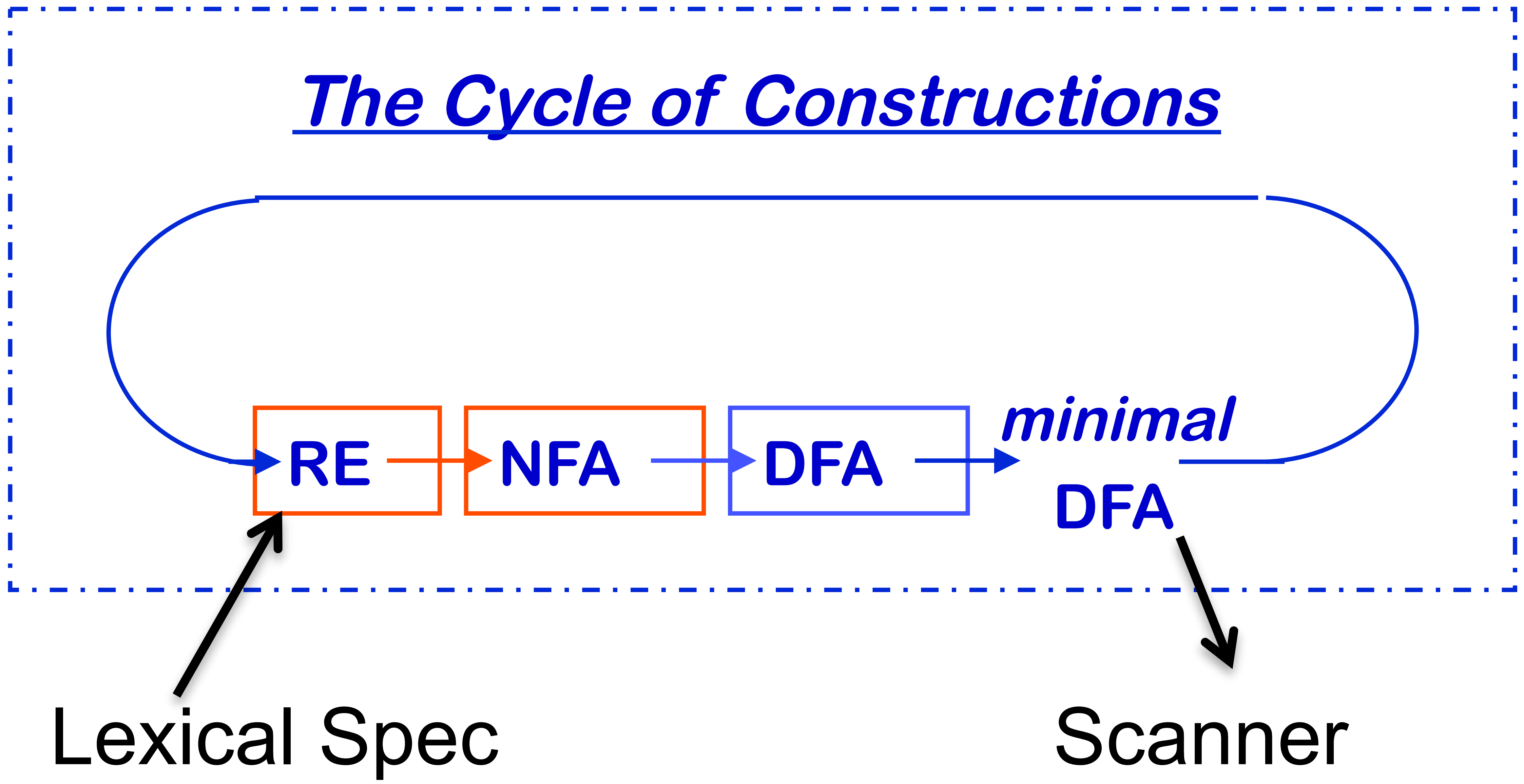
DFA → Minimal DFA

- 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

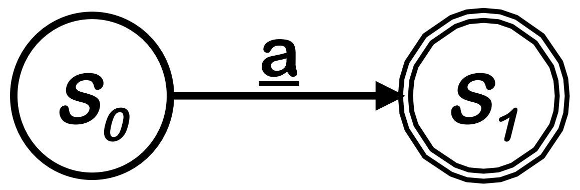



The Cycle of Constructions
RE
NFA
DFA
minimal
DFA
Lexical Spec
Scanner

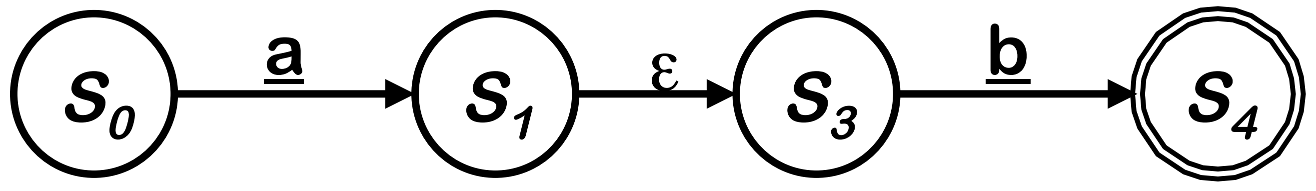
RE →NFA using Thompson's Construction

Key idea

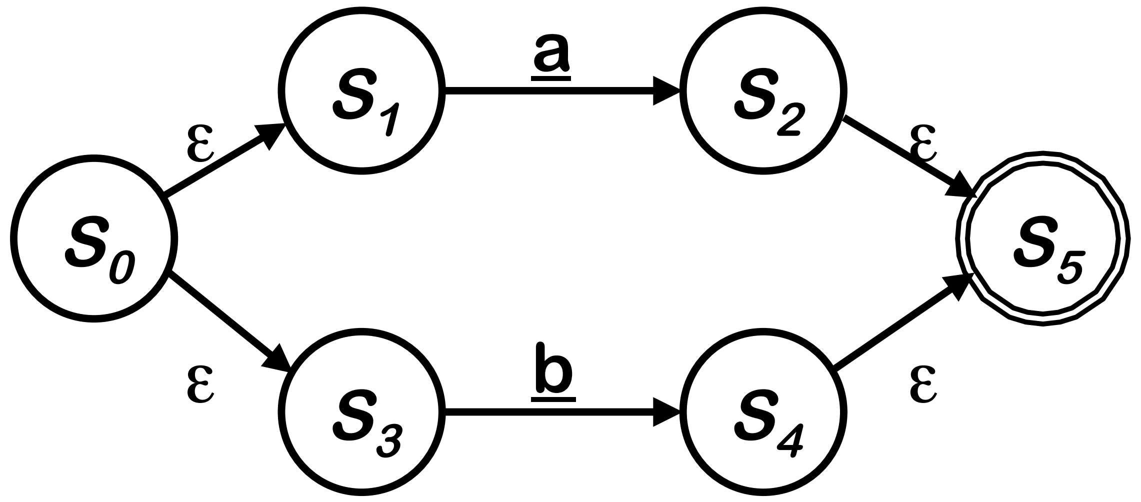
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



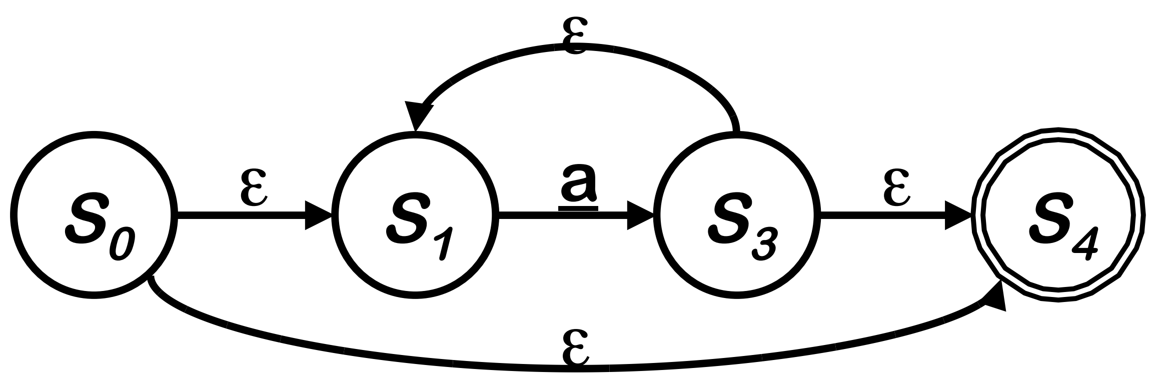
NFA for a



NFA for ab



NFA for a | b



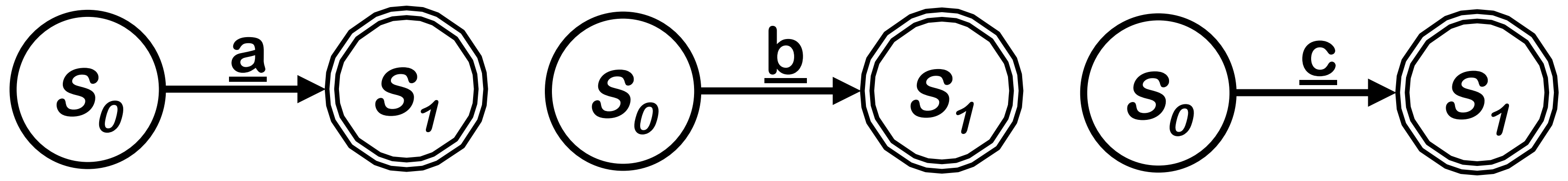
NFA for a*

Ken Thompson, CACM, 1968

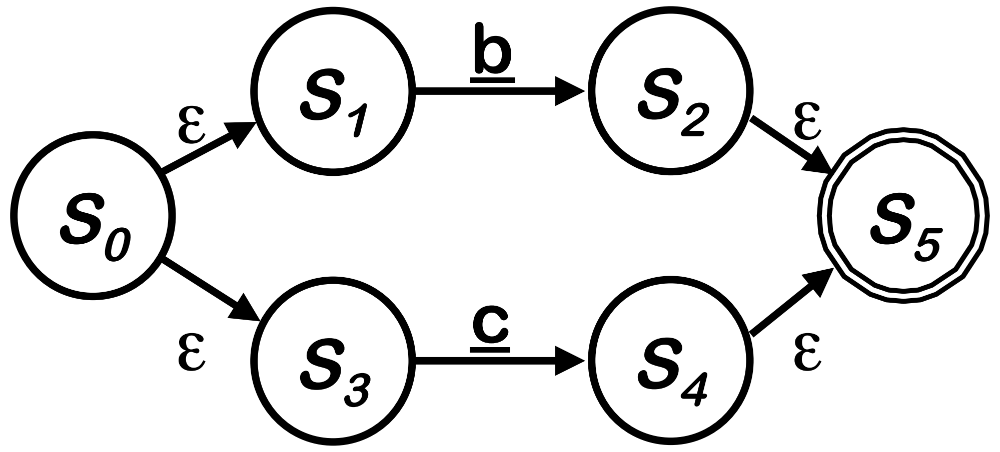
Example of Thompson’s Construction

Let’s try a (b | c)*

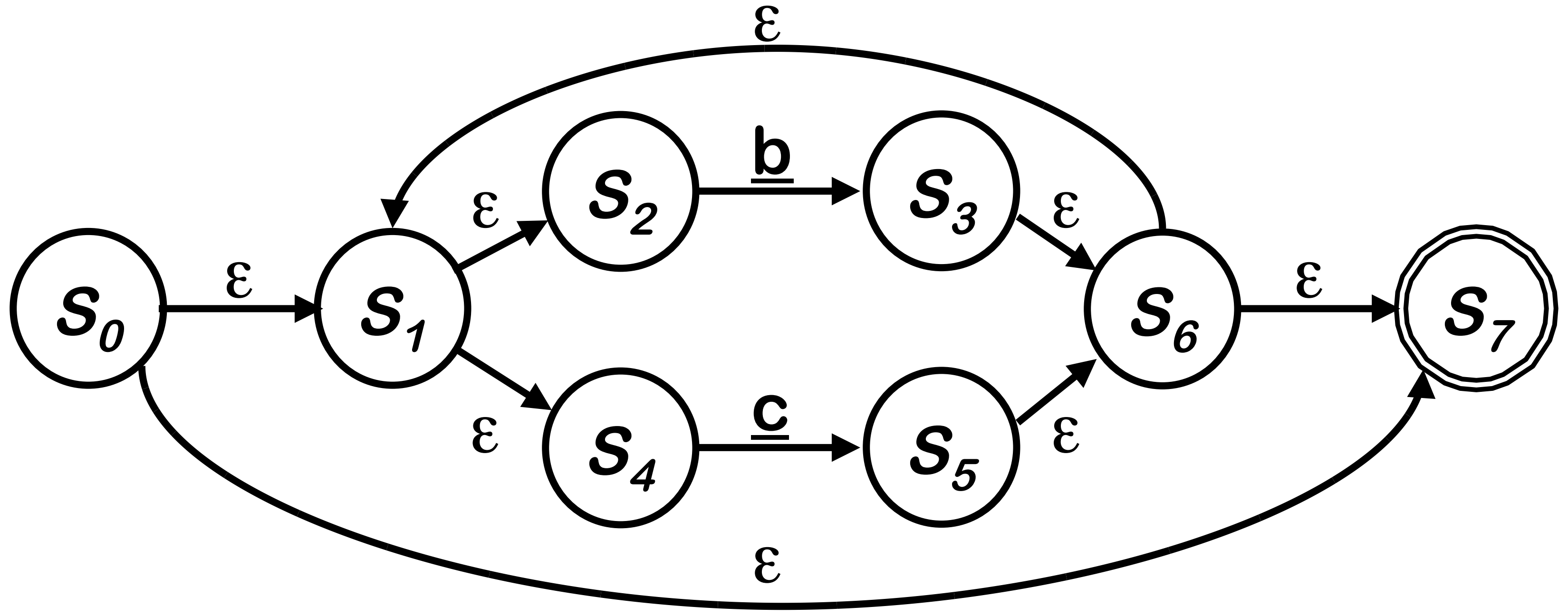
1. a, b, & c



2. b | c

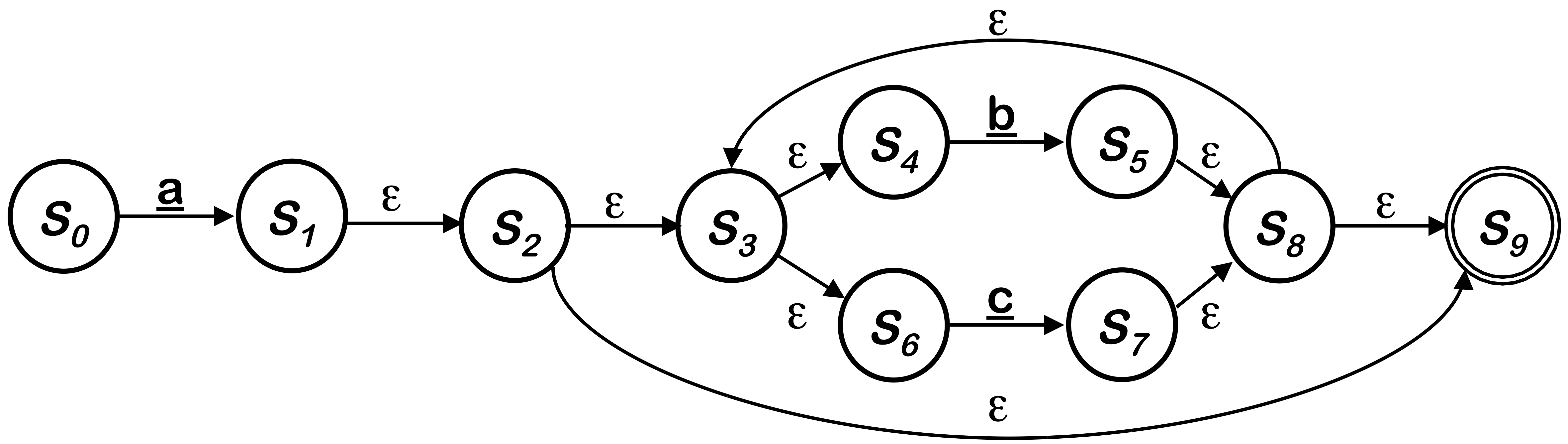


3. (b | c)*

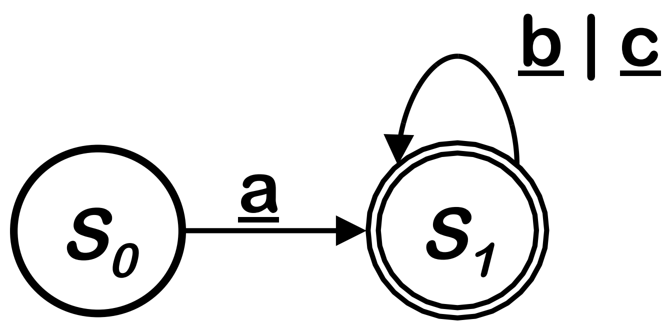


Example of Thompson's Construction *(con't)*

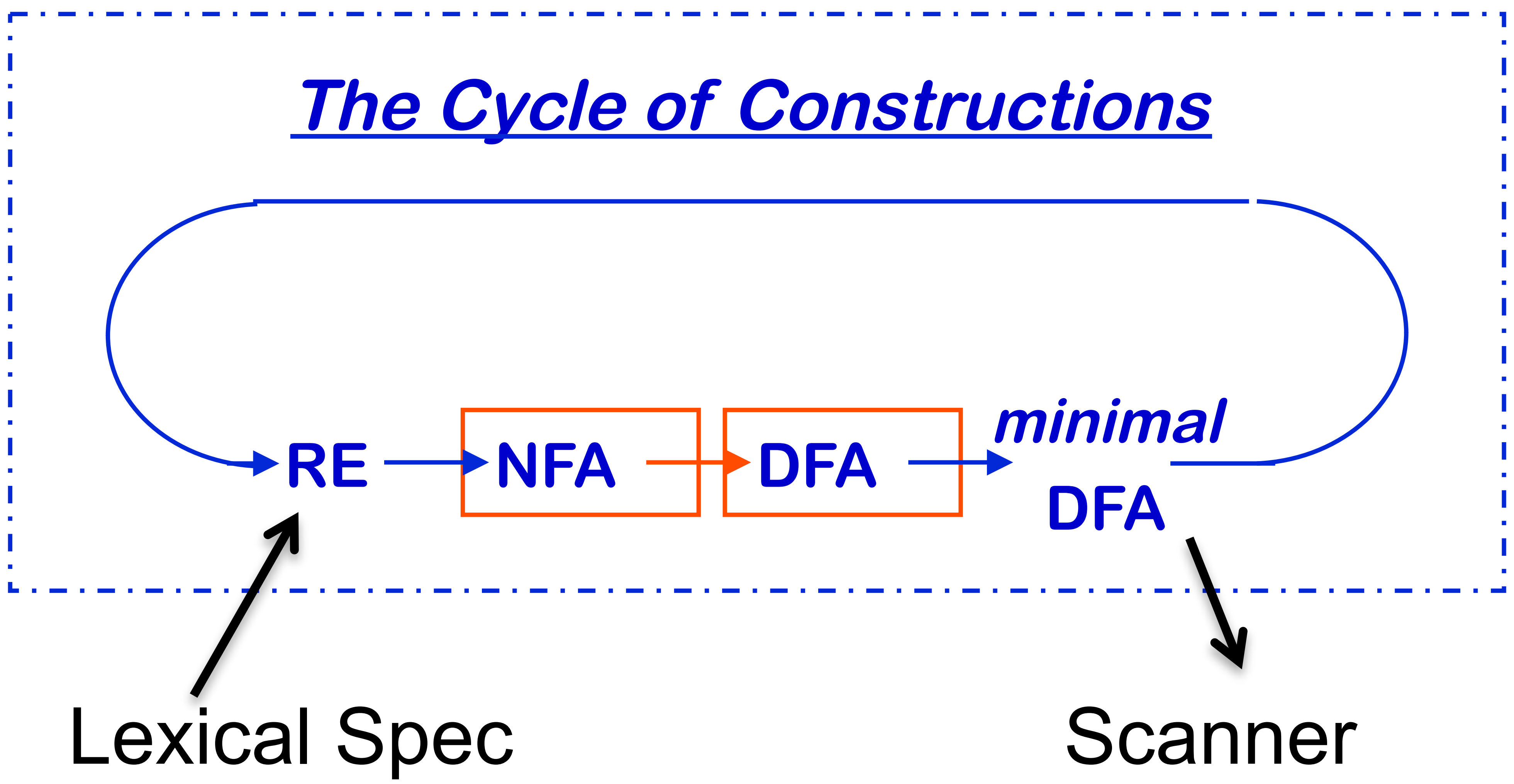
4. a (b | c)*



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

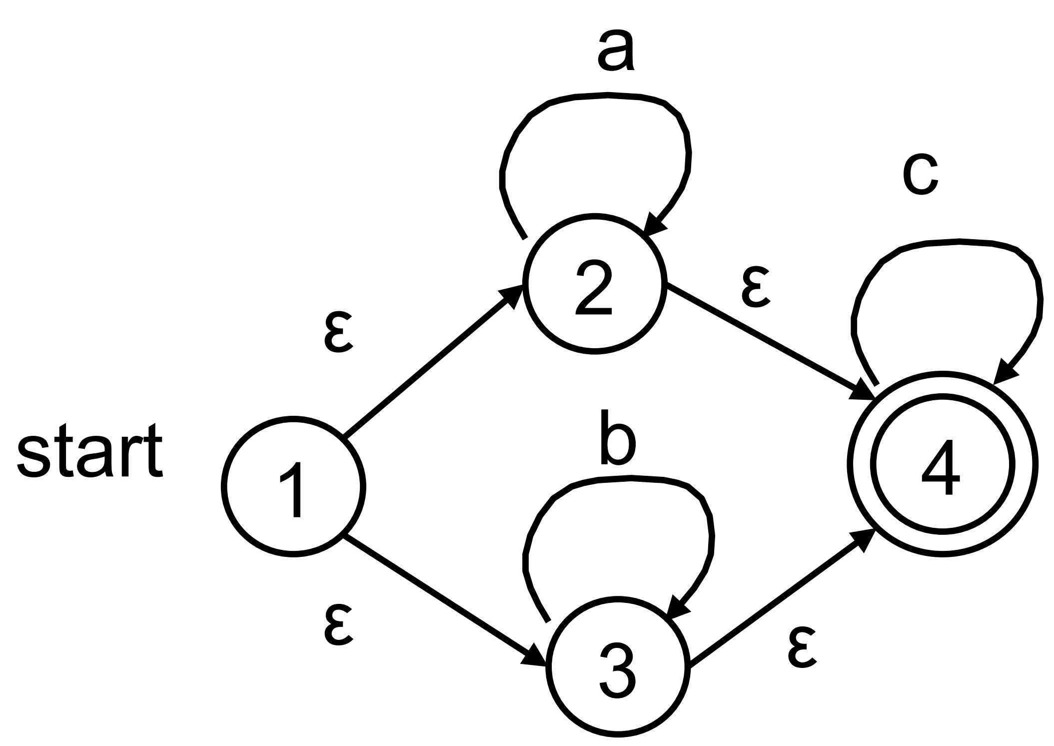
  = a non-empty subset of states of the NFA
- Start state

  = 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

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

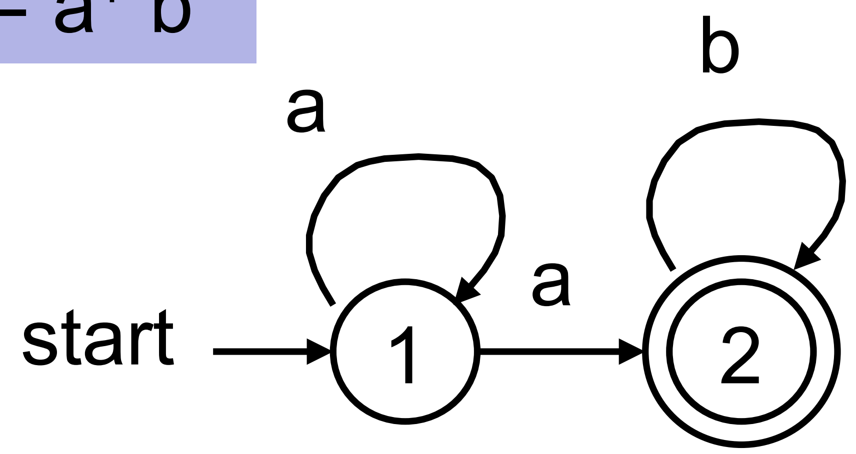
(a*| b*) c*



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

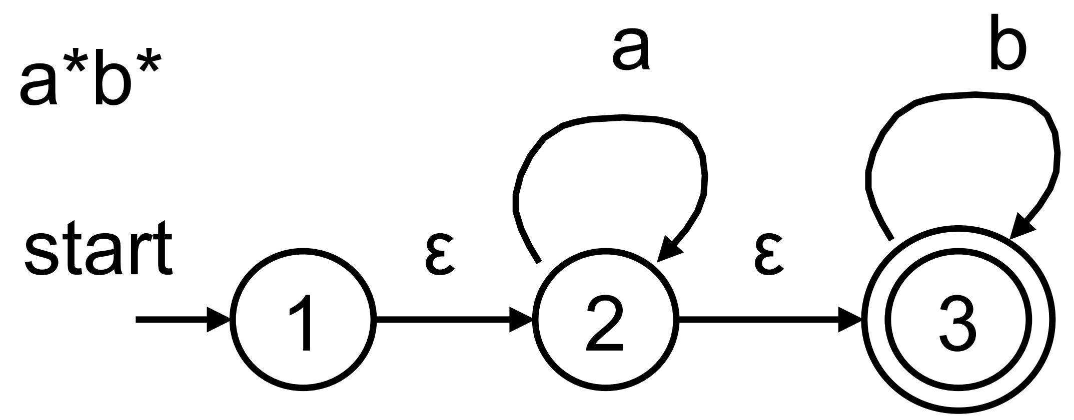
R= $a^+ b^*$



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

NFA to DFA (3)

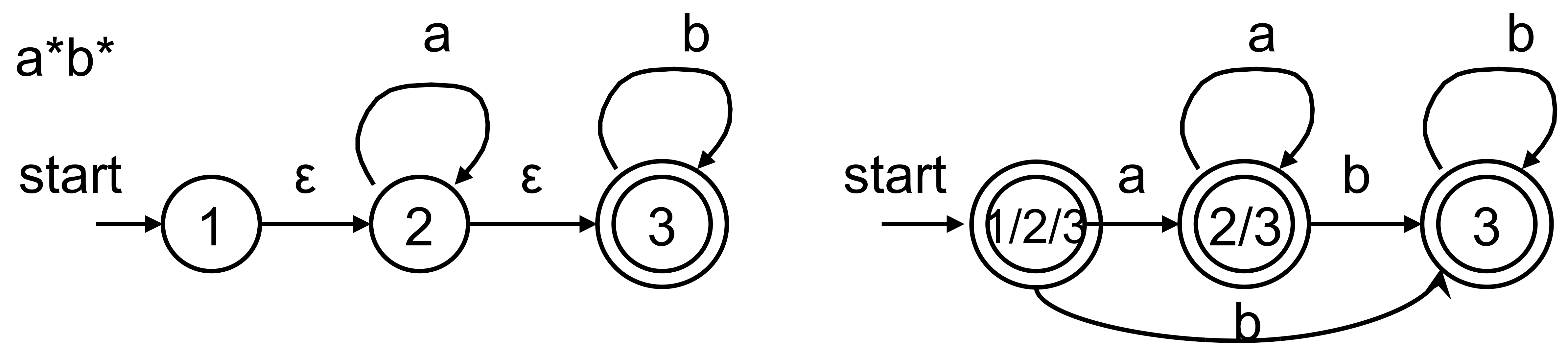
- ε transitions
  - Any state reachable by an ε transition is “part of the state”
  - ε-closure - Any state reachable from S by ε transitions is in the ε-closure; treat ε-closure as 1 big state, always include ε-closure as part of the state



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}

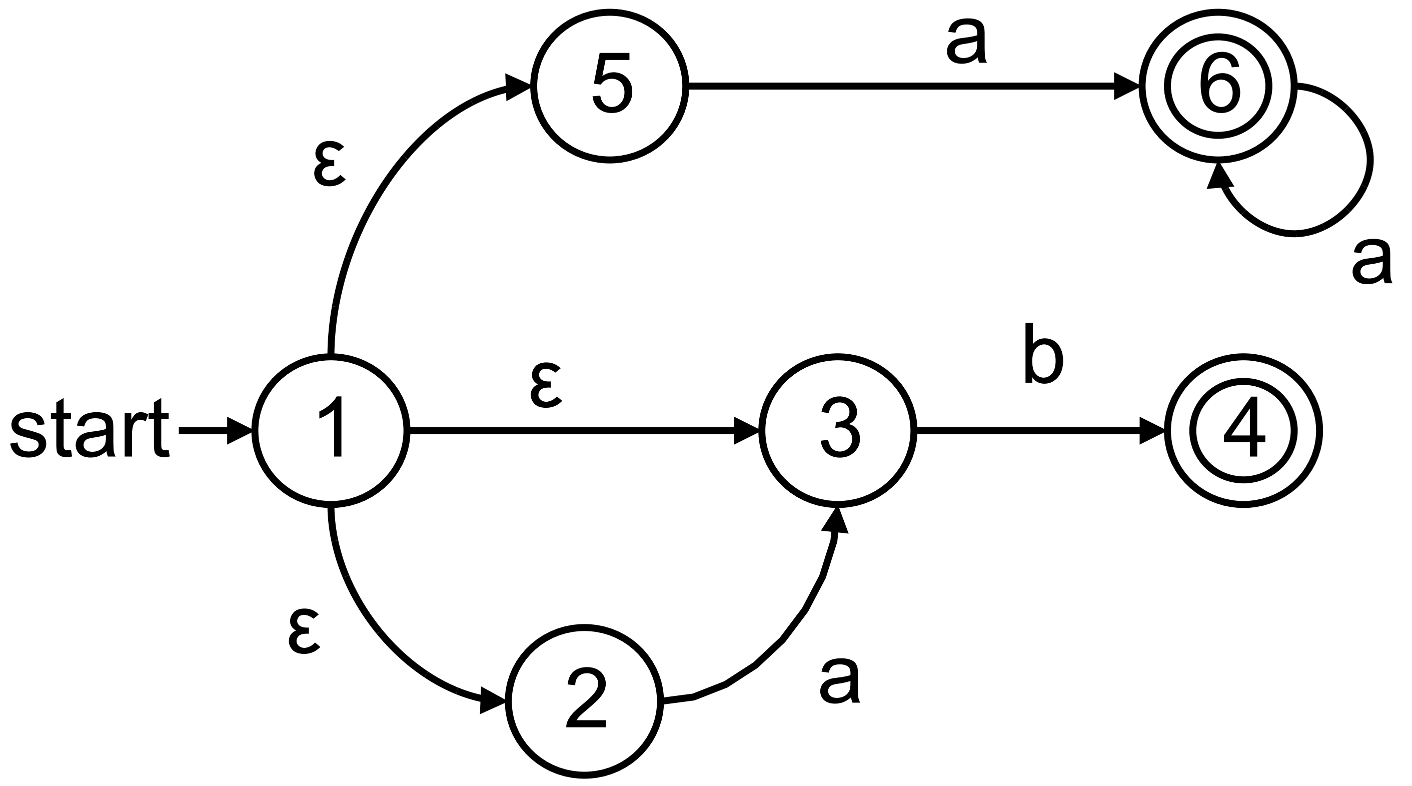
NFA to DFA (3)

- 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

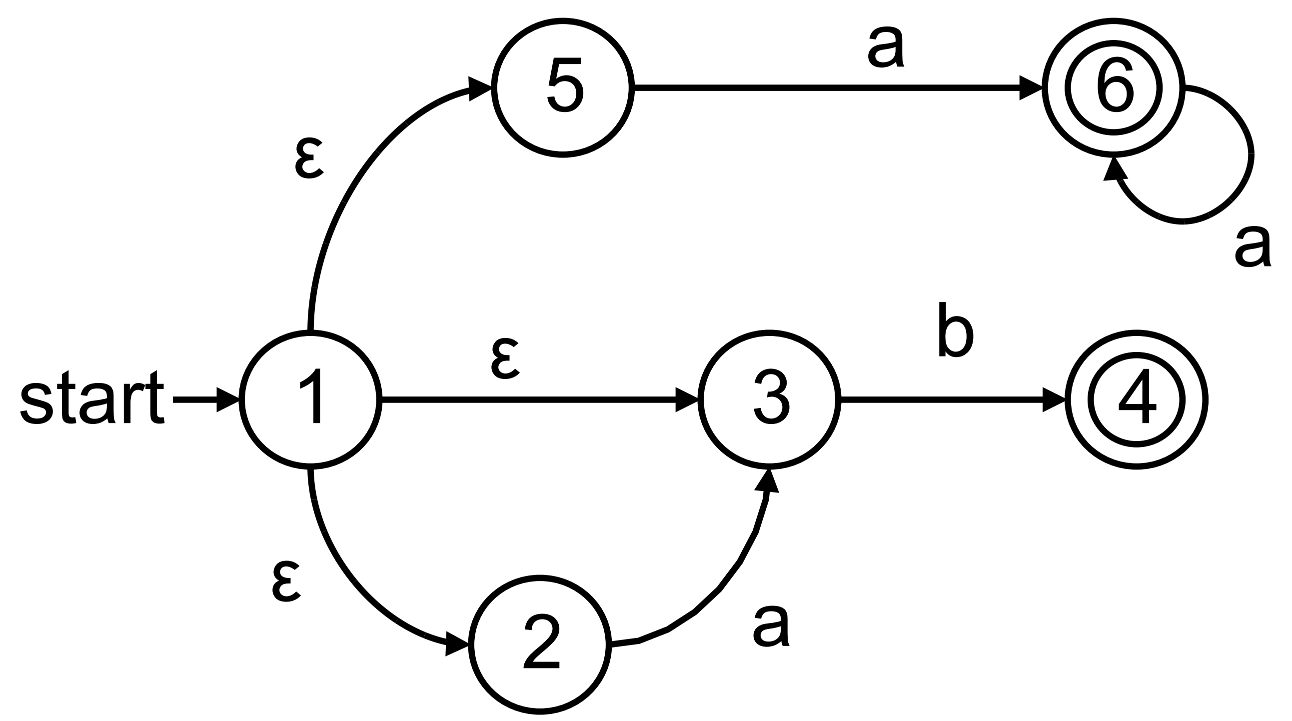


1. ε-closure(1) = {1,2,3}; include1/2/3
2. Move(1/2/3, a) = {2, 3} + ε-closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} + ε-closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} + ε-closure(2) = {2,3}
5. Move(2/3, b) = {3} + ε-closure(3) = {3}
6. Move(3, b) = {3} + ε-closure(3) = {3}

NFA to DFA - Example



NFA to DFA - Example



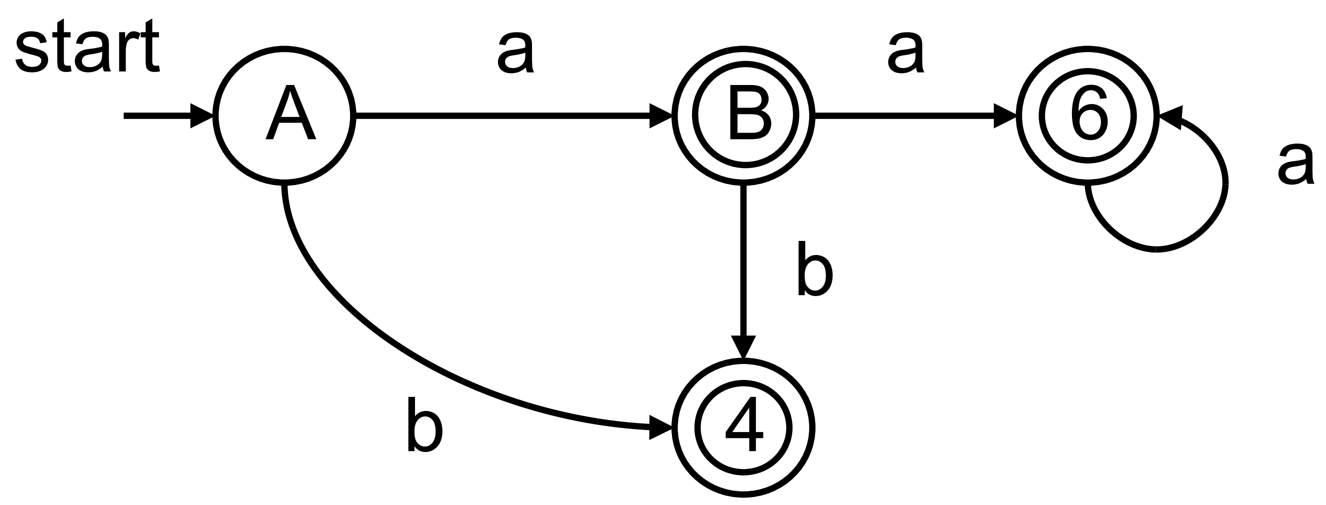
ε-closure(1) = {1, 2, 3, 5}

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}

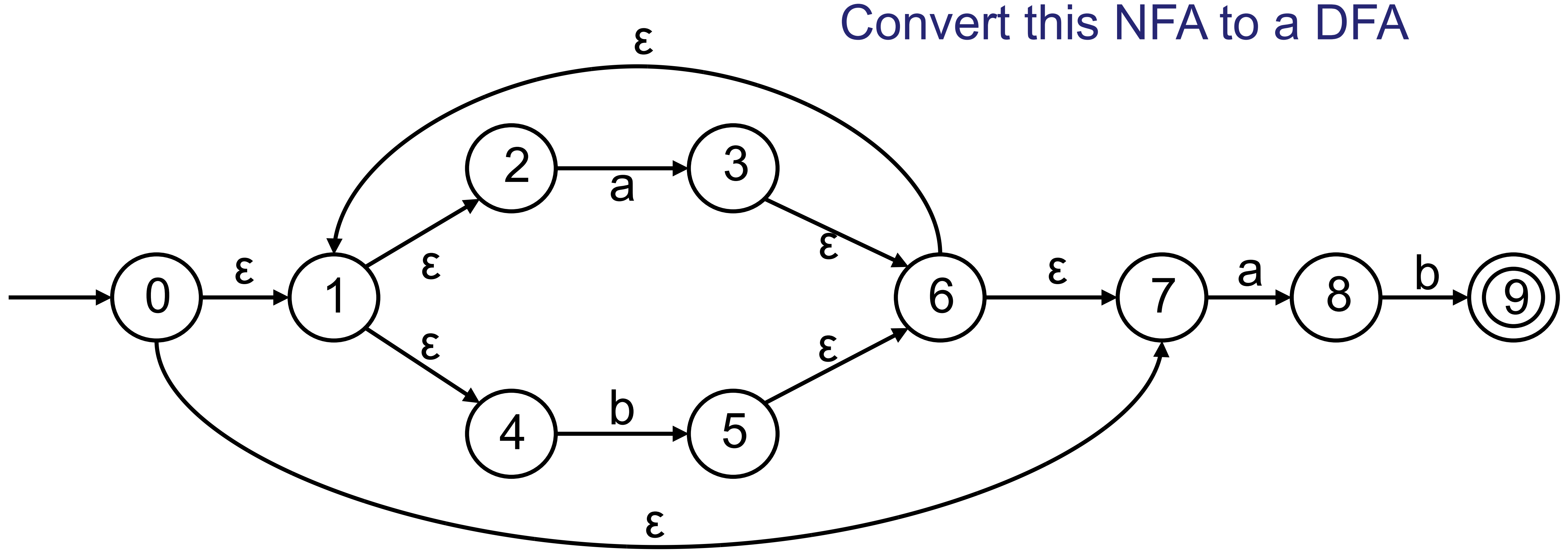
move(6, a) = {6} + ε-closure(6) = {6}

move(6, b) → ERROR

move(4, a|b) → ERROR

Class Problem

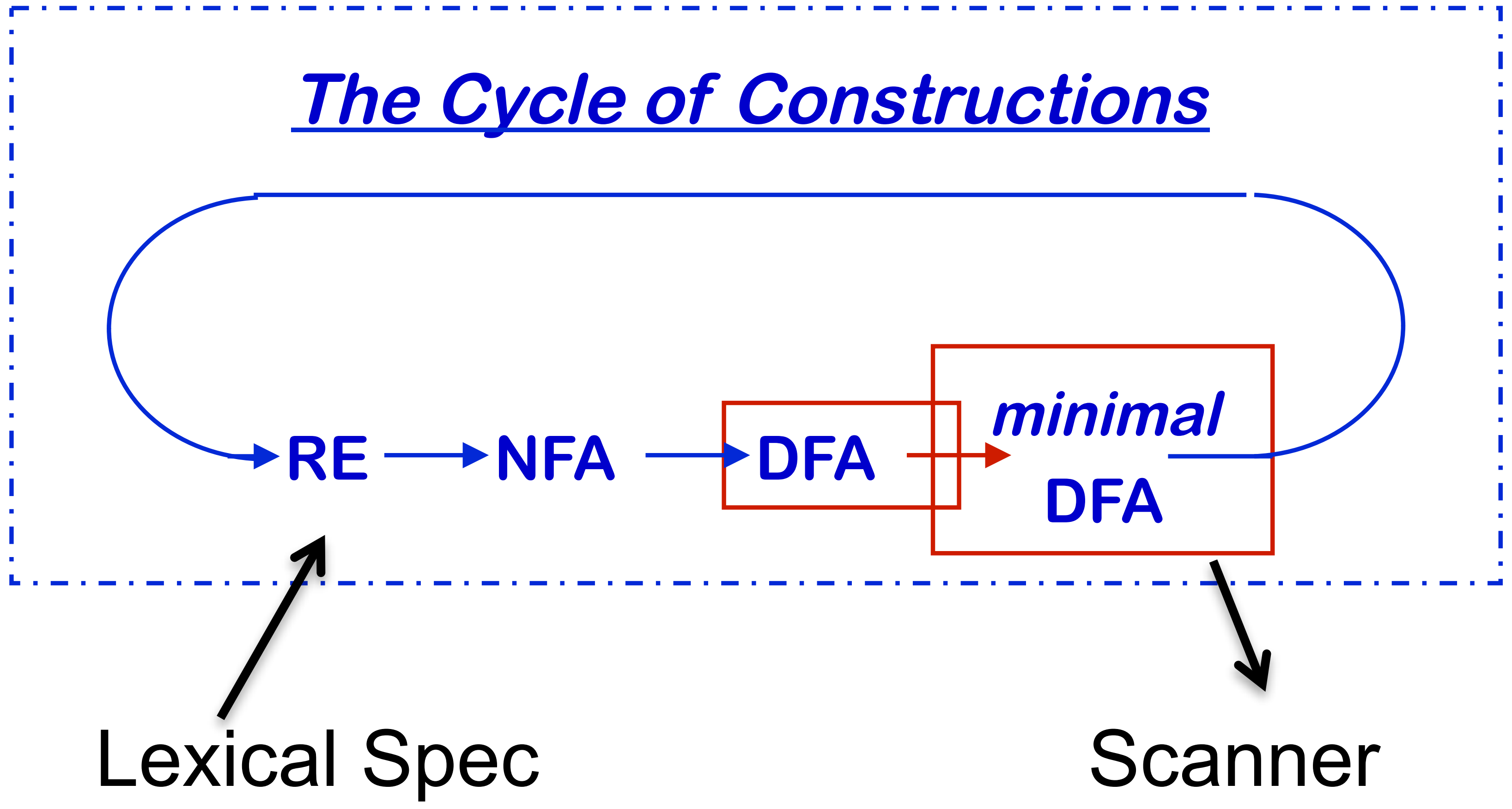
Convert this NFA to a DFA



NFA to DFA : cont..

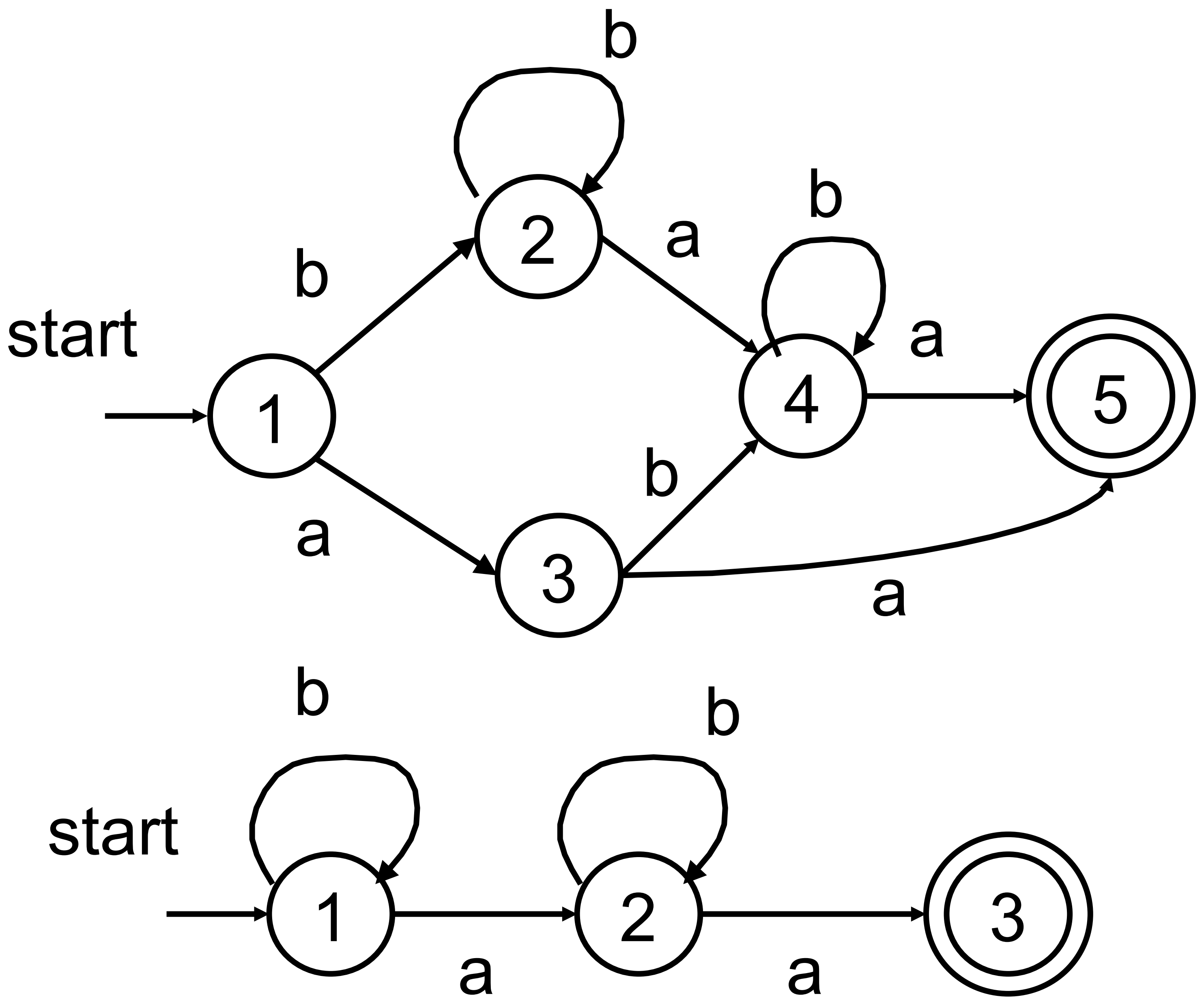
- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?

  2^N - 1 = finitely many


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

State Minimization

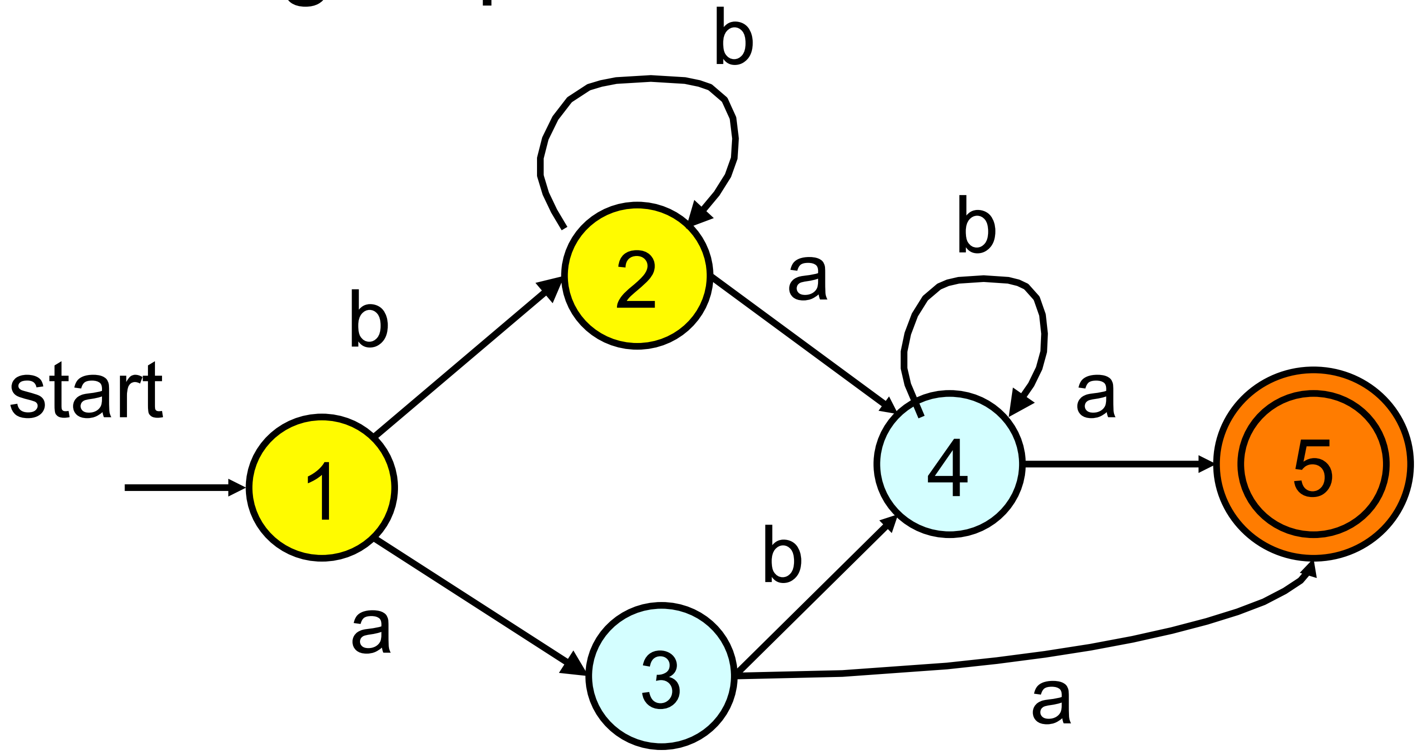
- Resulting DFA can be quite large
  - Contains redundant or equivalent states



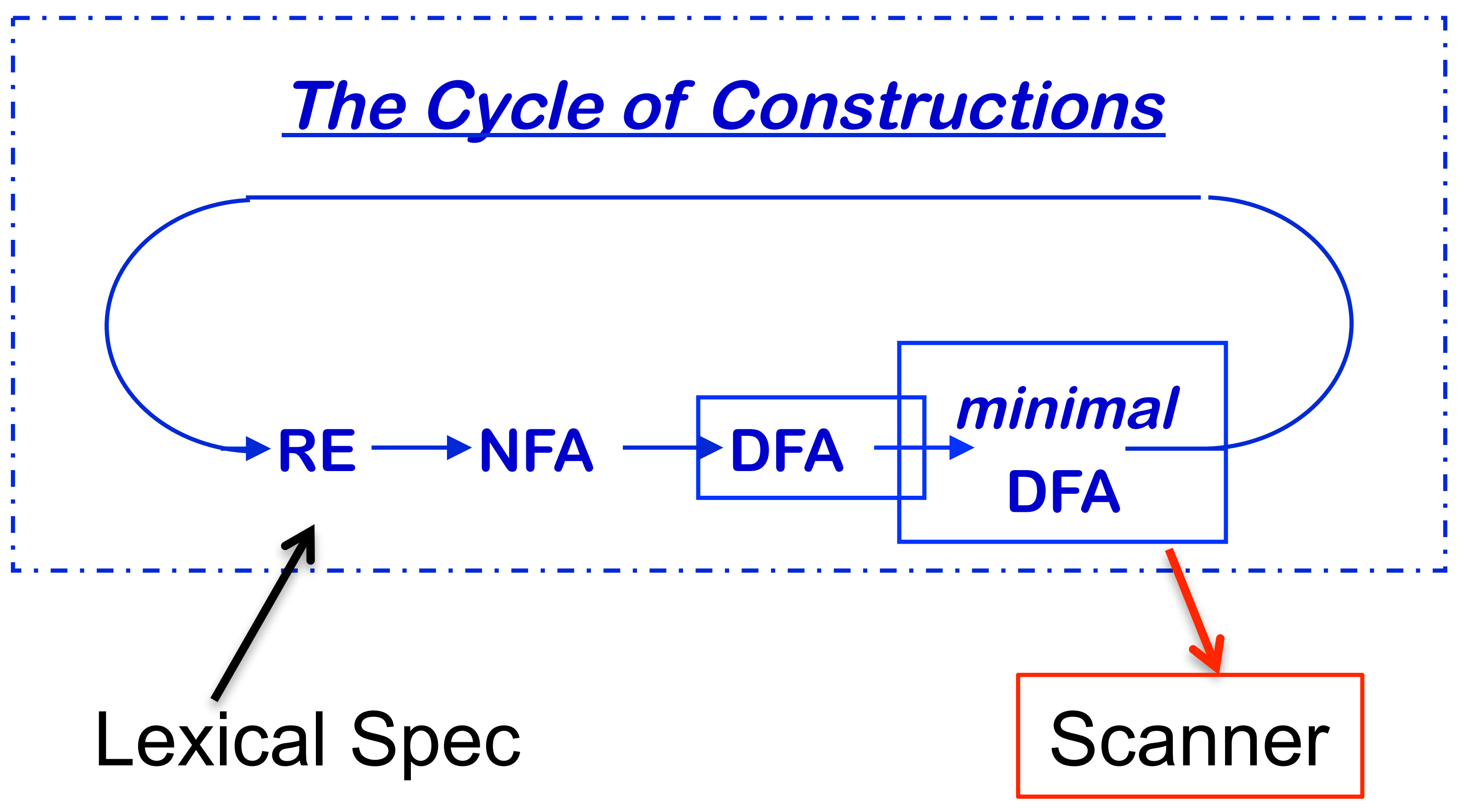
Both DFAs accept
b*ab*a

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



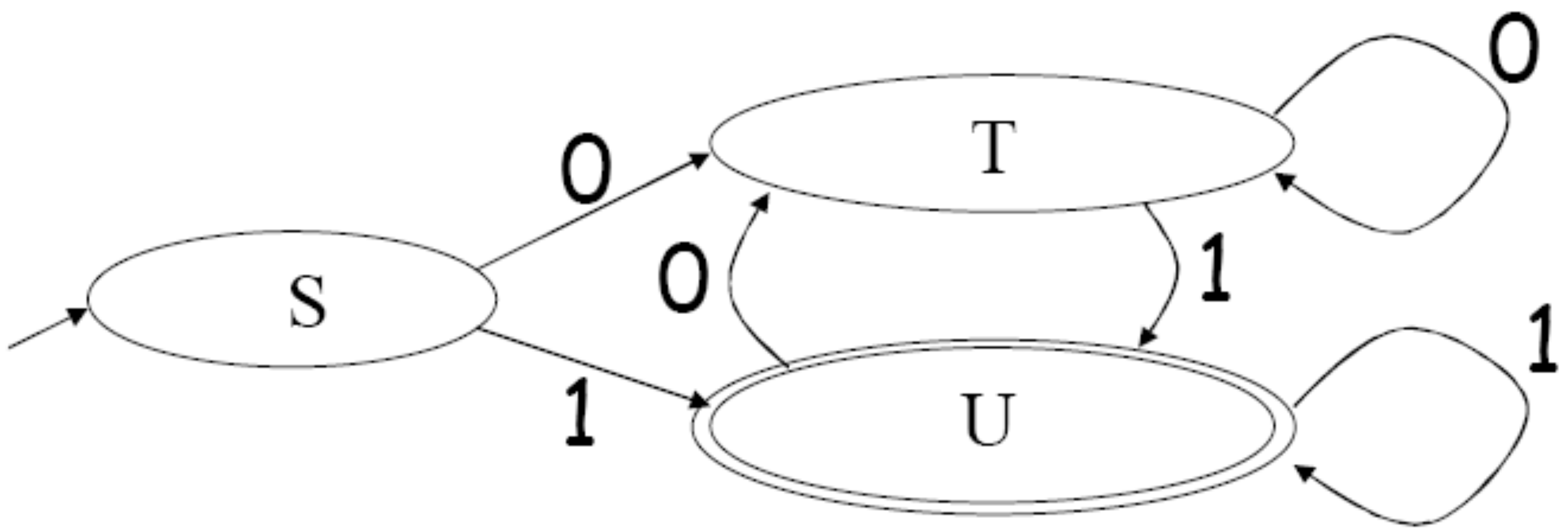
Basic strategy: identify distinguishing transitions


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

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** for C

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_]*
```

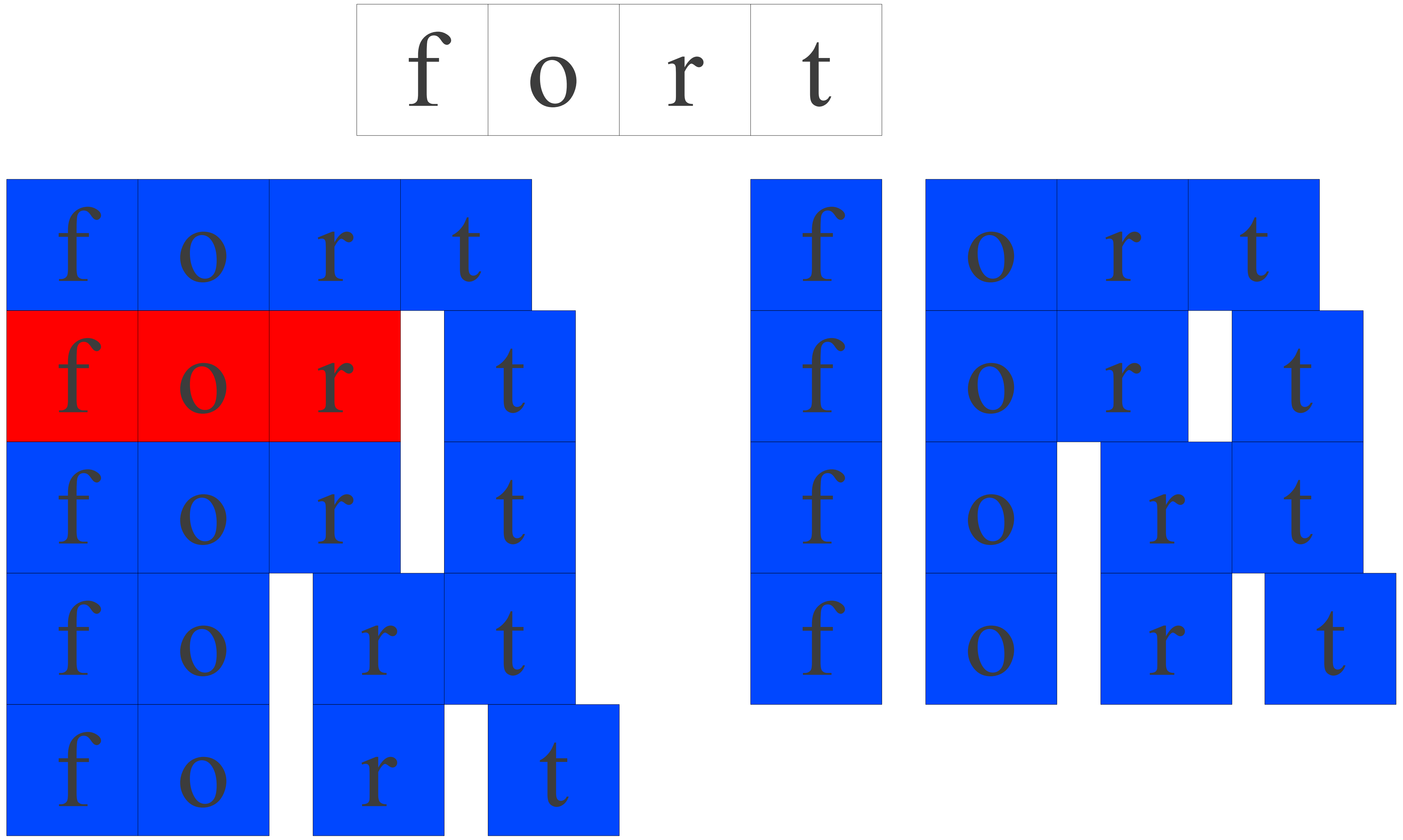
Lexing Ambiguities

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?

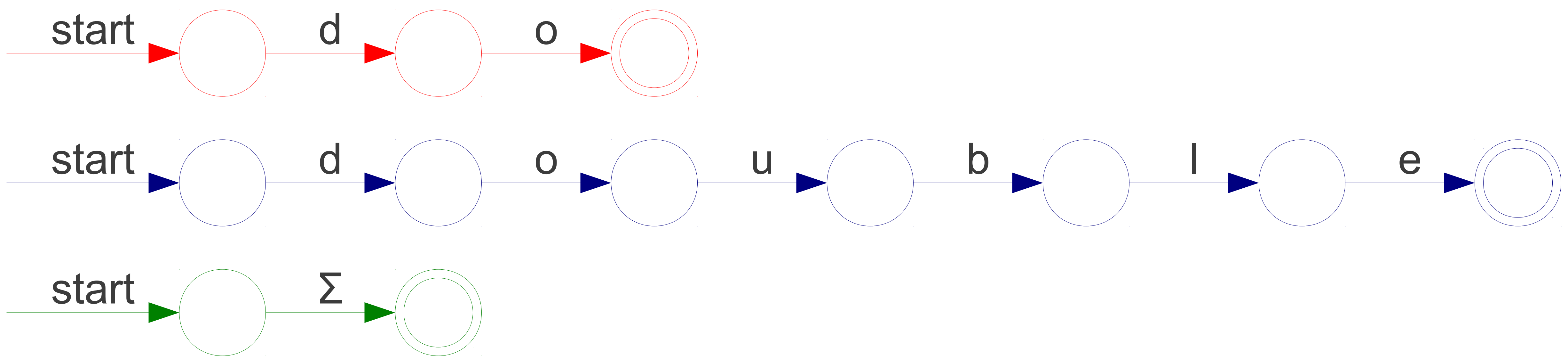
- Example

Implementing Maximal Munch

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

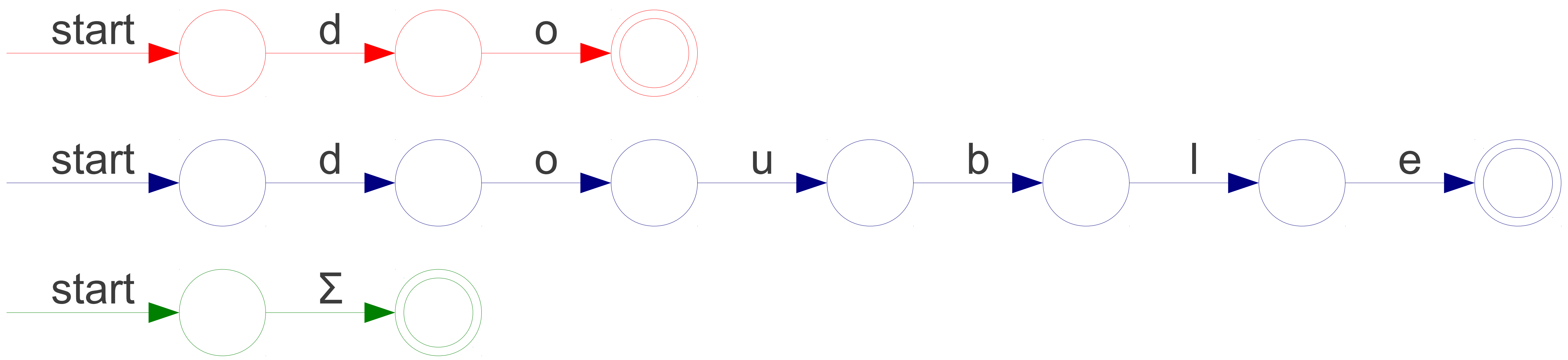
Implementing Maximal Munch

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



Implementing Maximal Munch

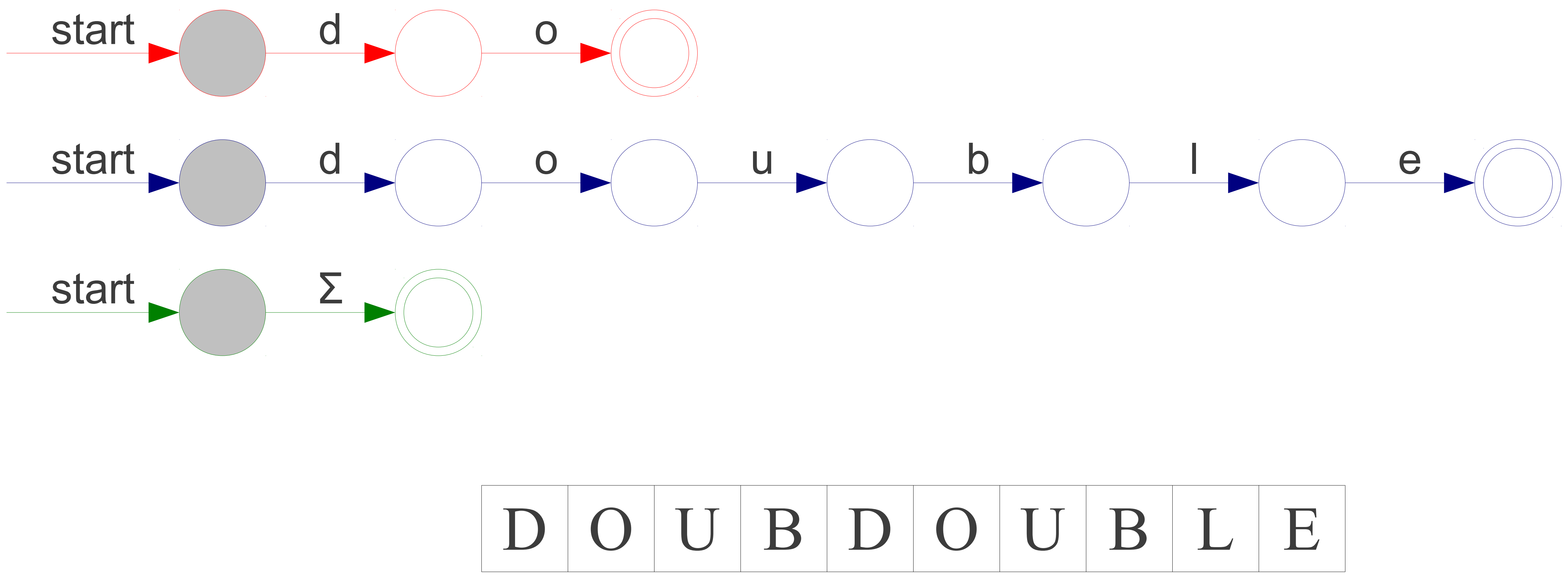
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



D	O	U	B	D	O	U	B	L	E

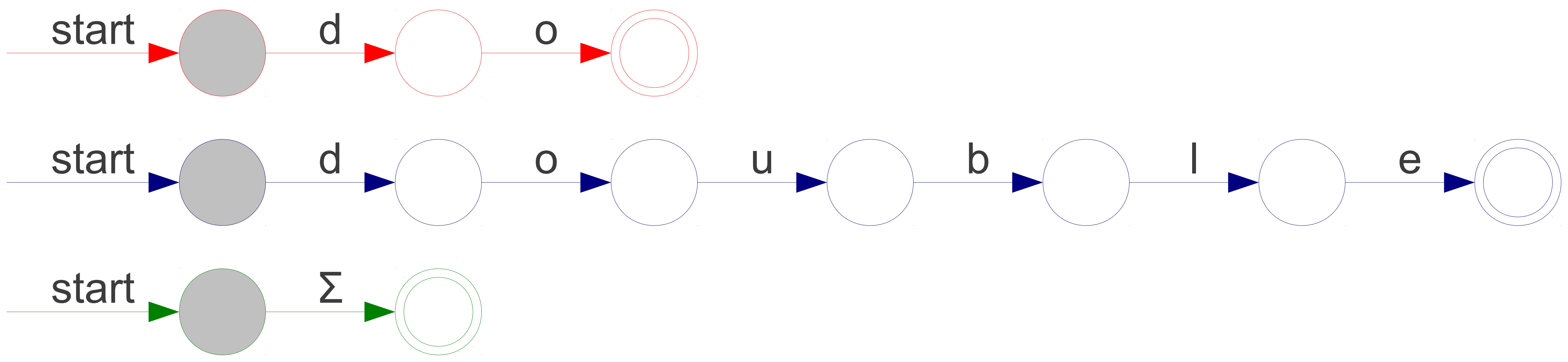
Implementing Maximal Munch

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



Implementing Maximal Munch

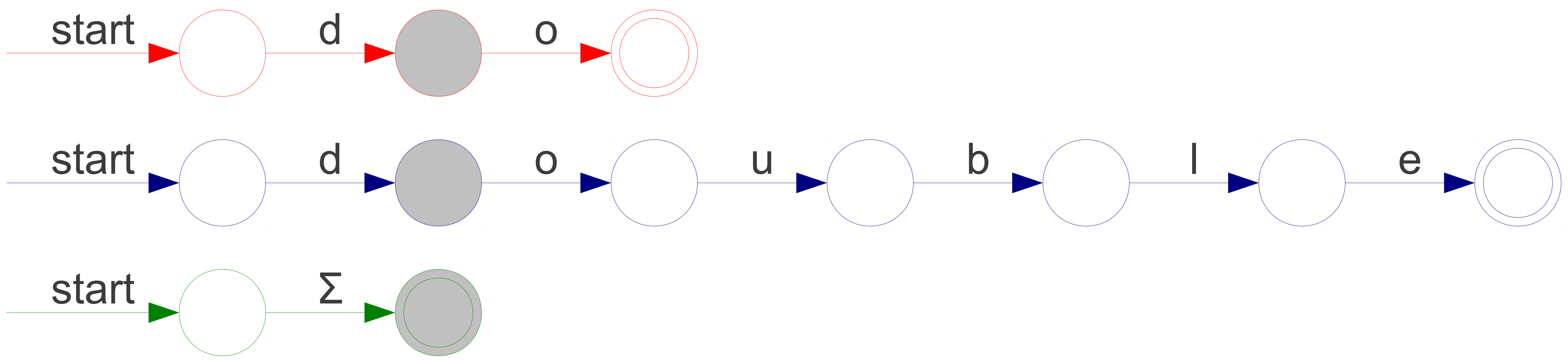
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

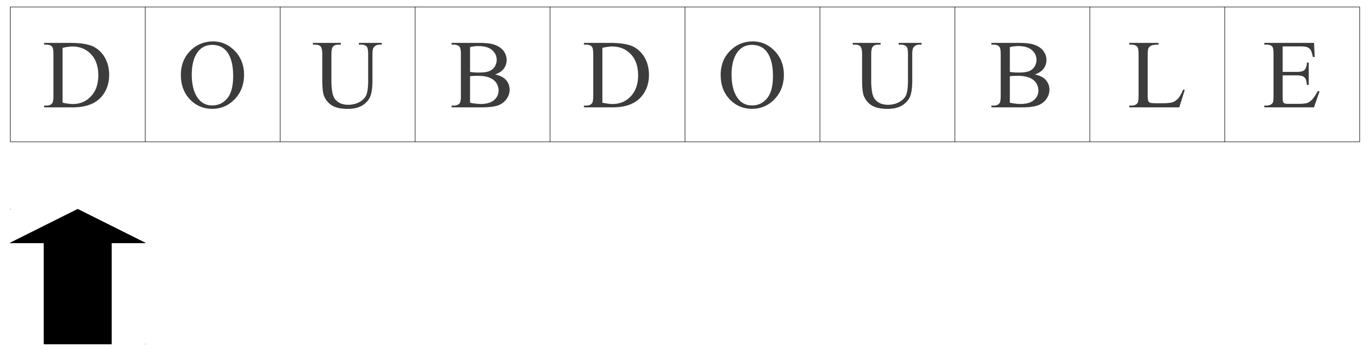


D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

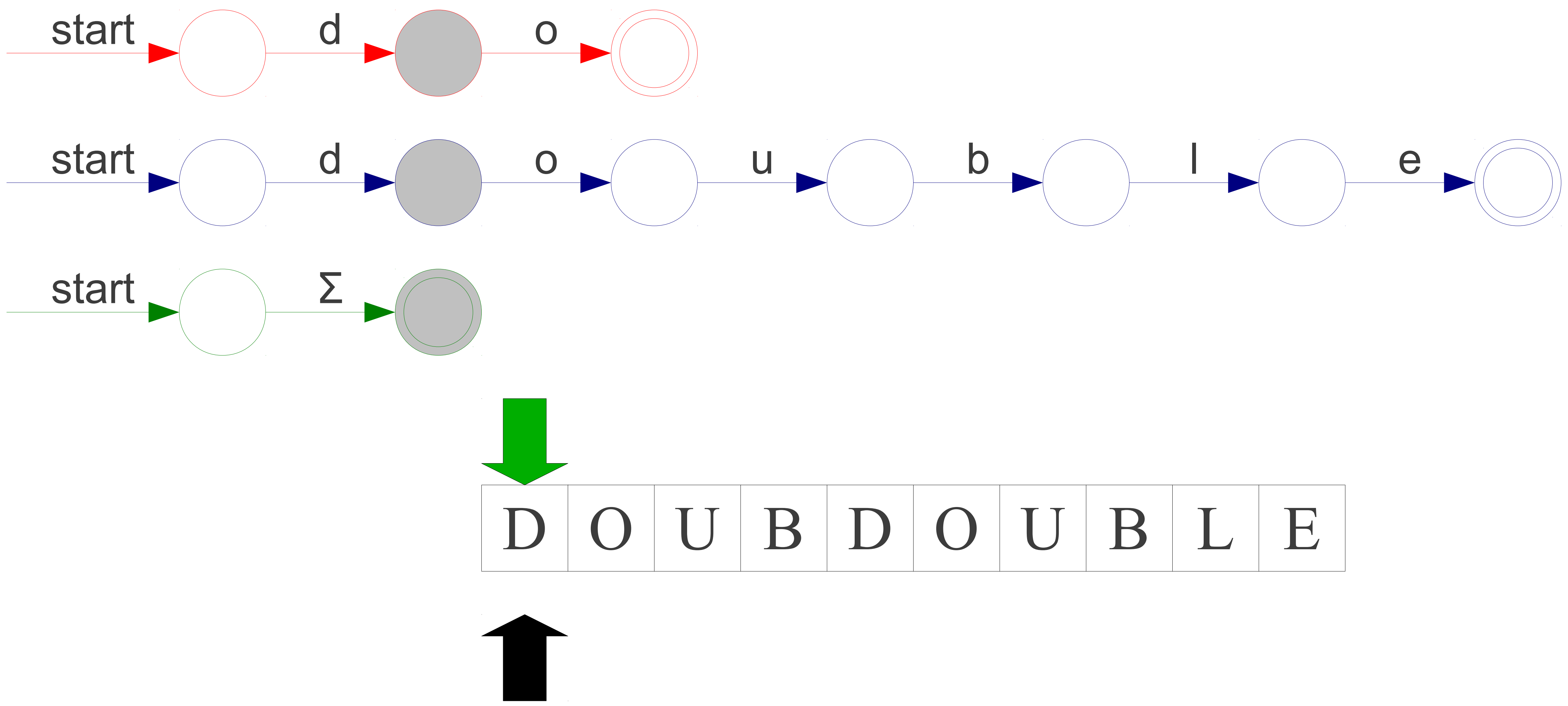
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





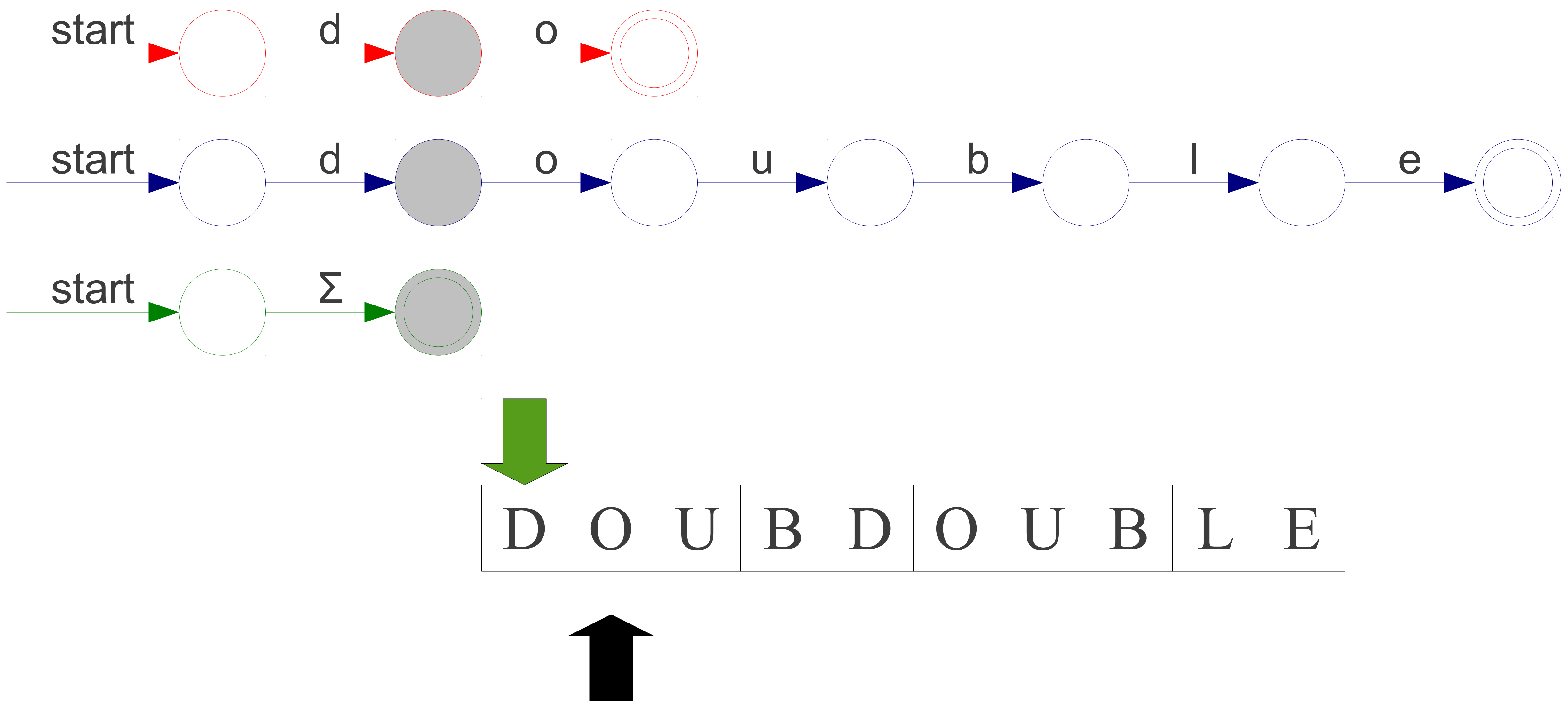
Implementing Maximal Munch

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



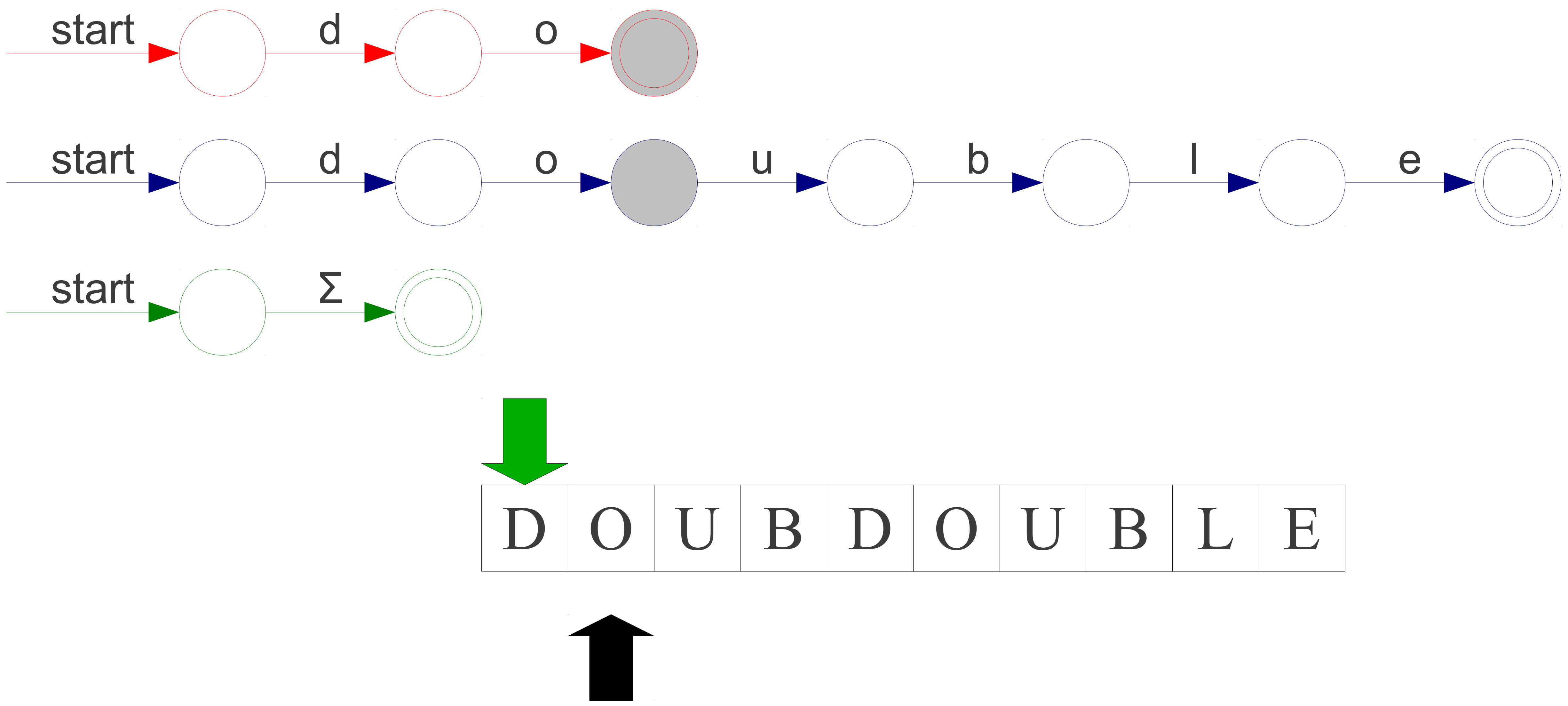
Implementing Maximal Munch

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



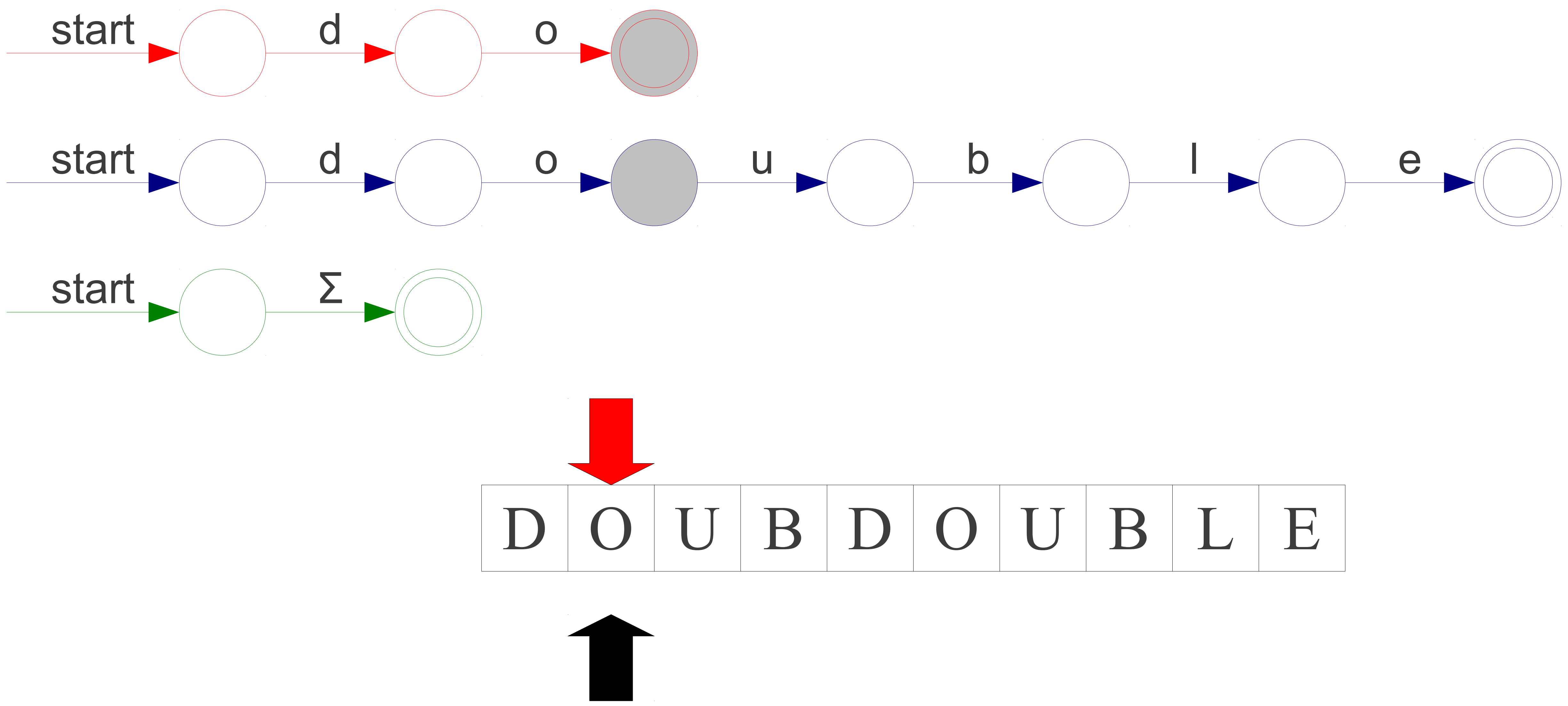
Implementing Maximal Munch

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



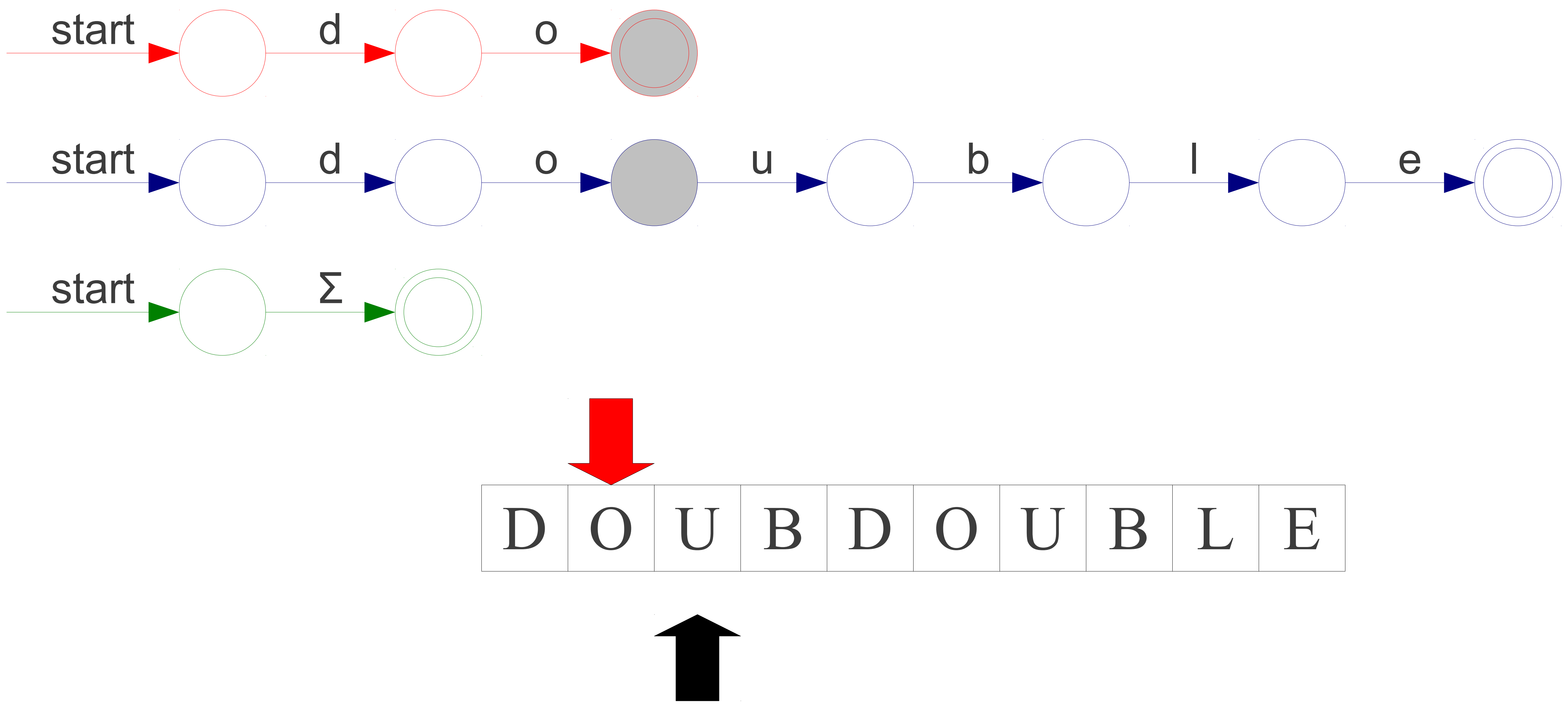
Implementing Maximal Munch

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



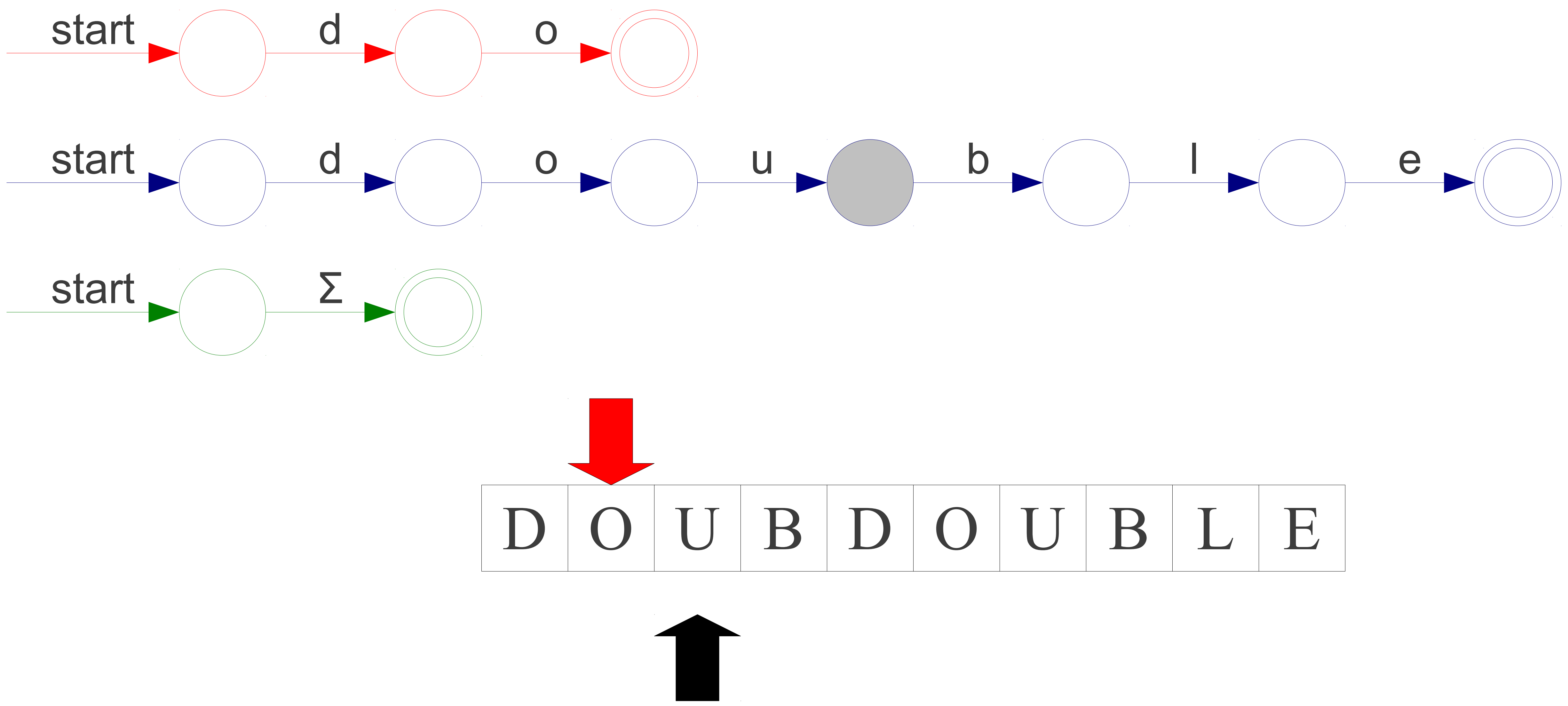
Implementing Maximal Munch

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



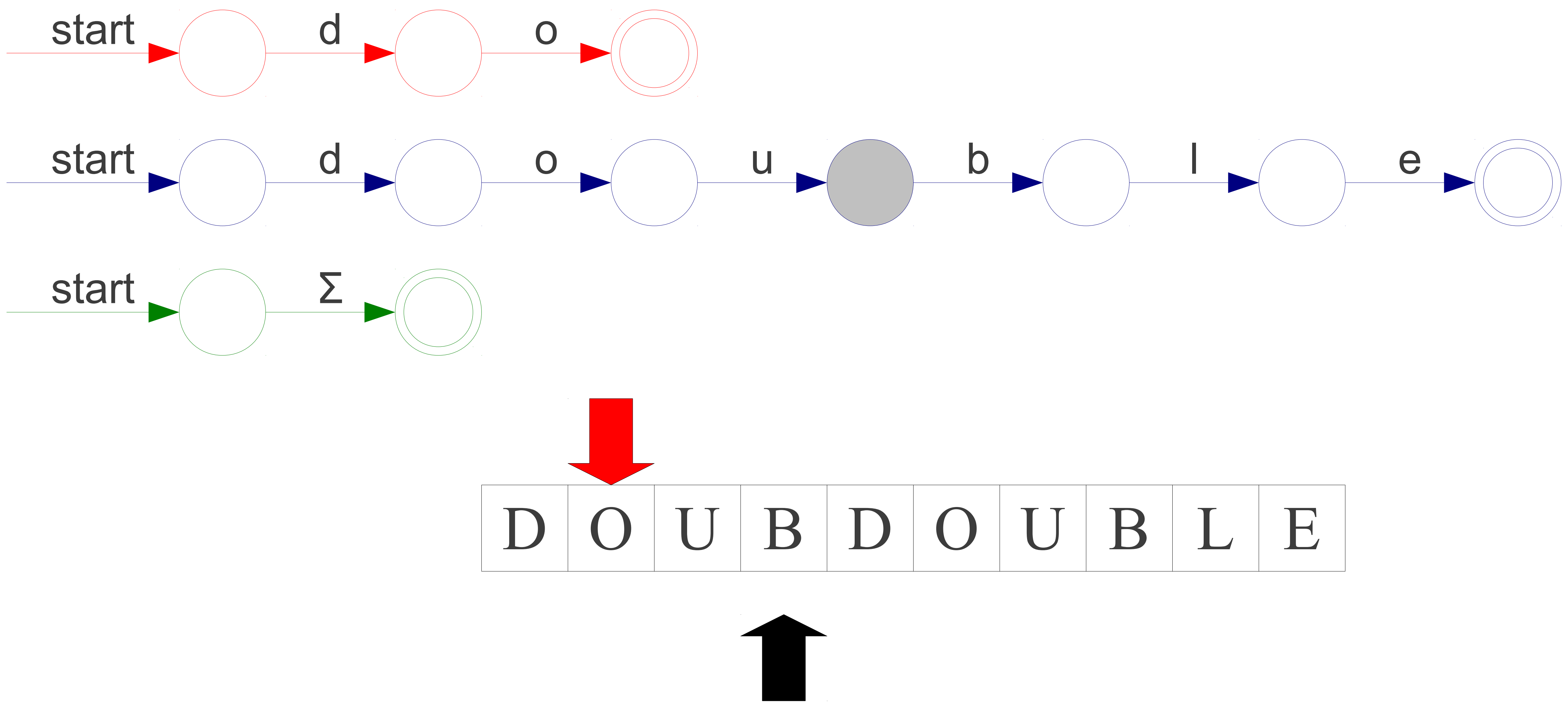
Implementing Maximal Munch

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



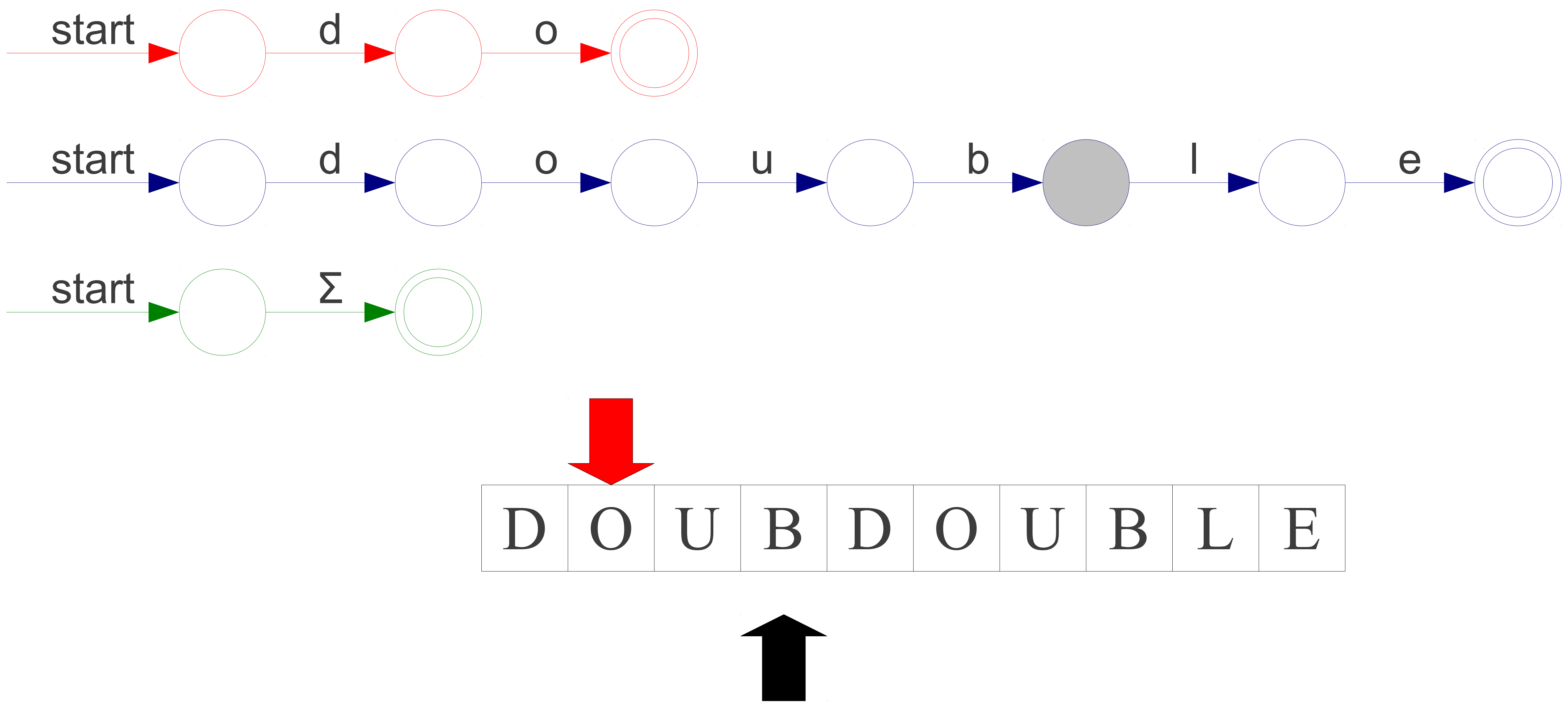
Implementing Maximal Munch

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



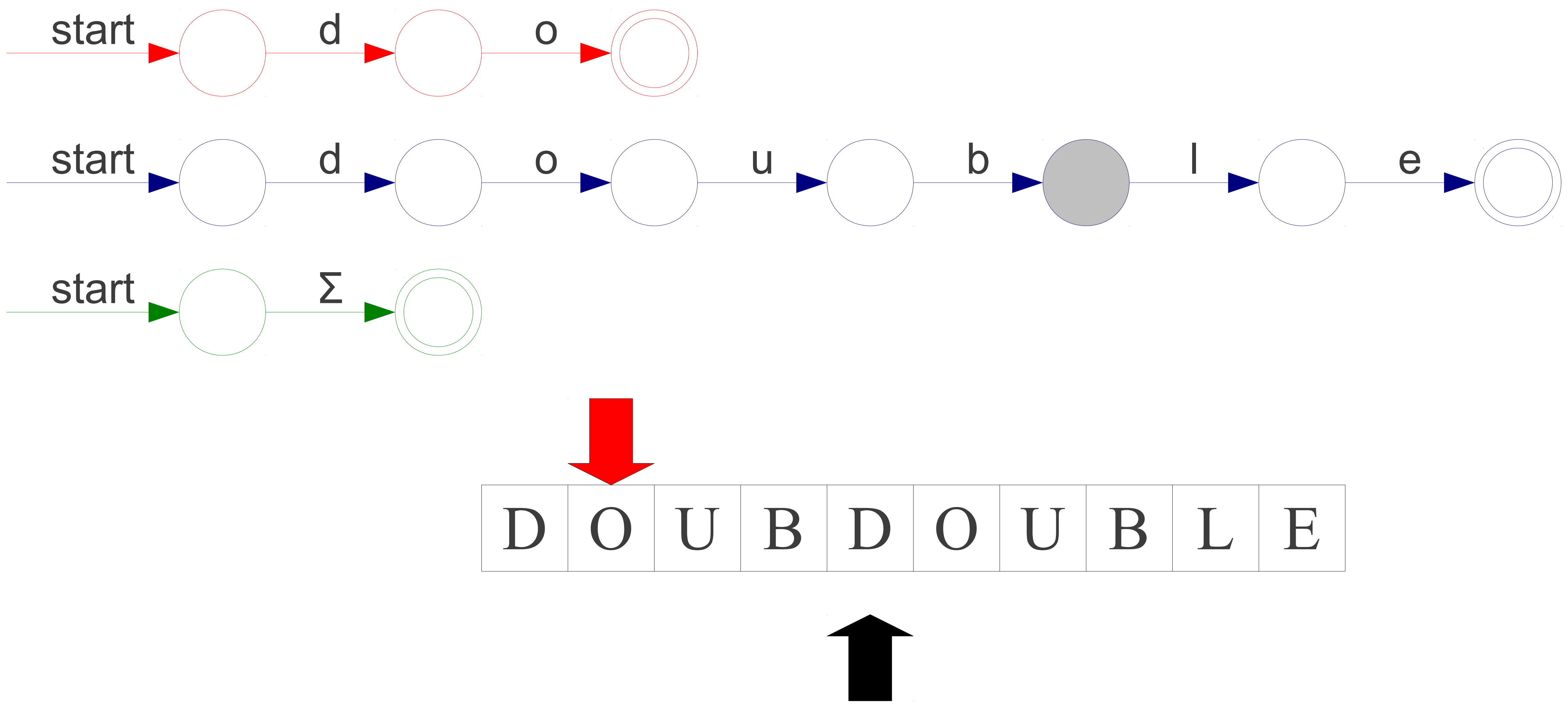
Implementing Maximal Munch

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



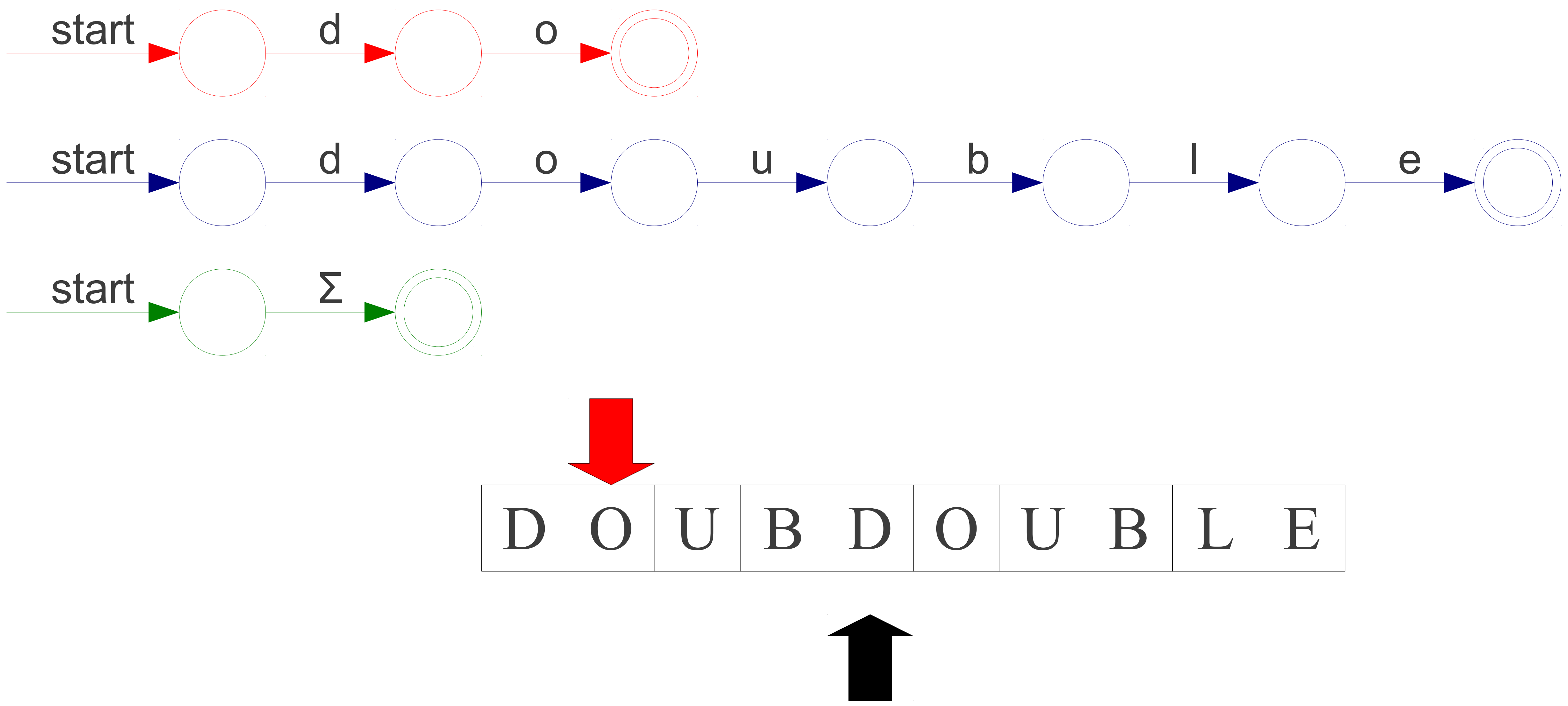
Implementing Maximal Munch

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



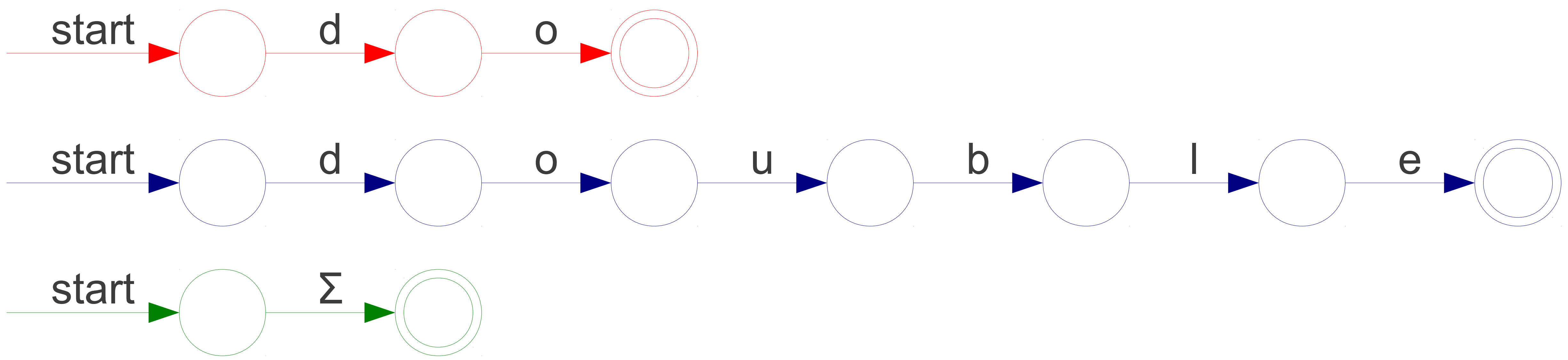
Implementing Maximal Munch

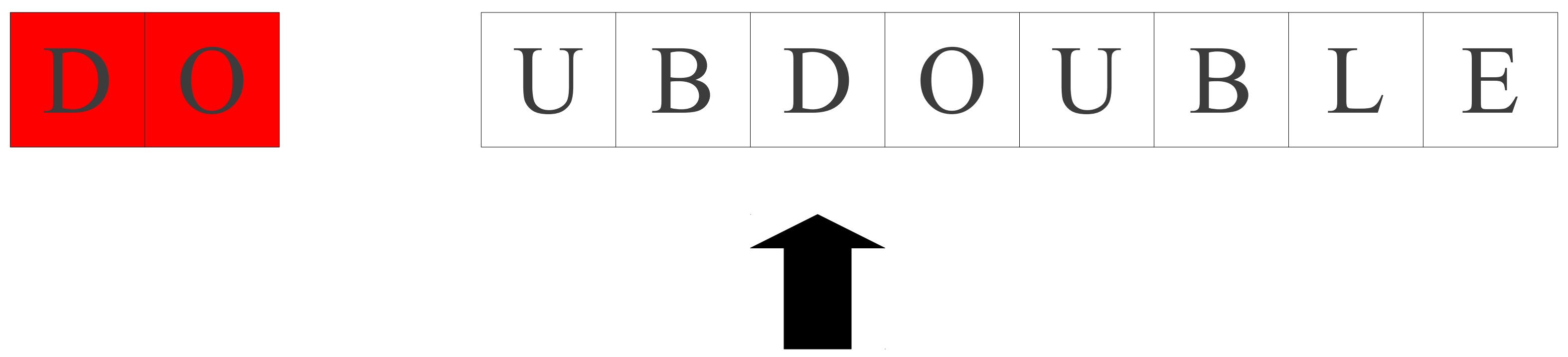
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

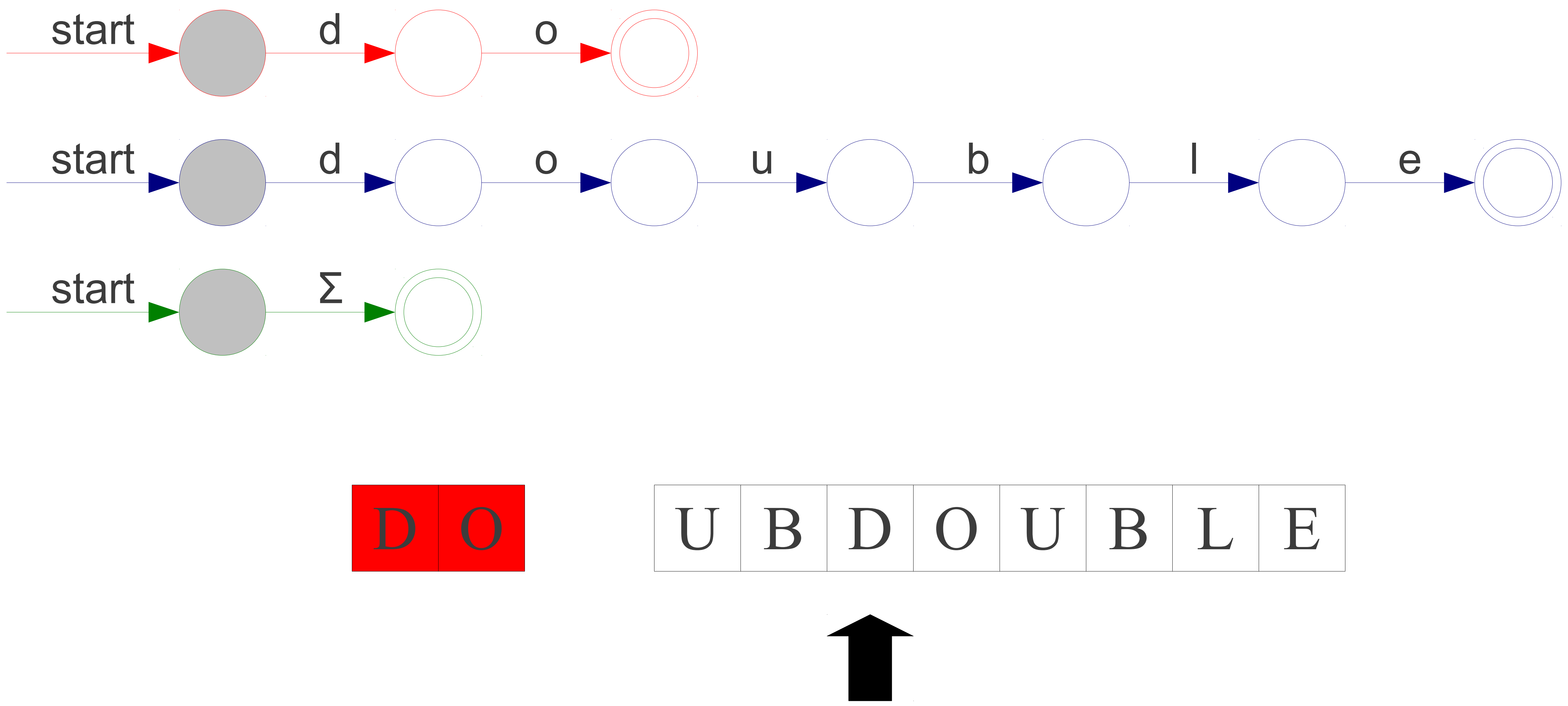
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





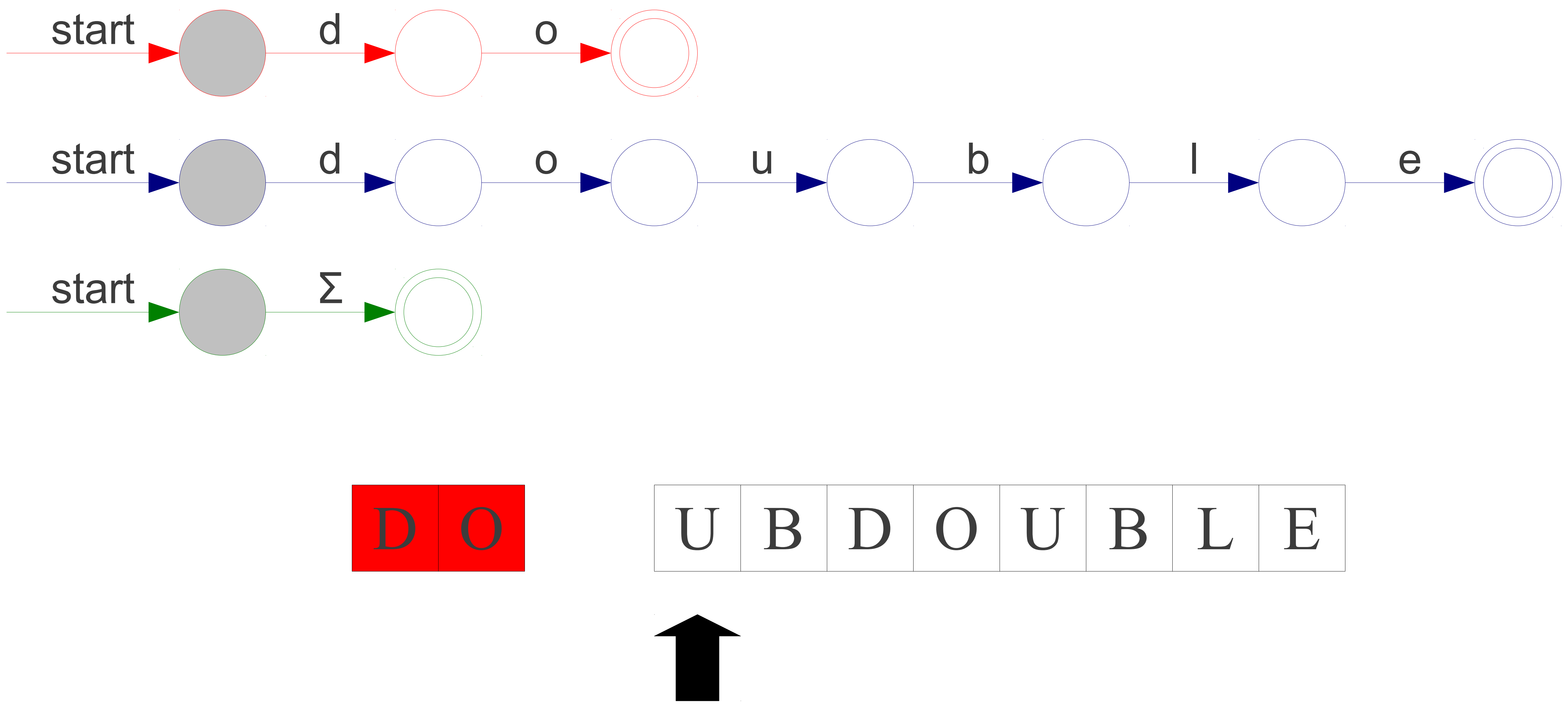
Implementing Maximal Munch

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



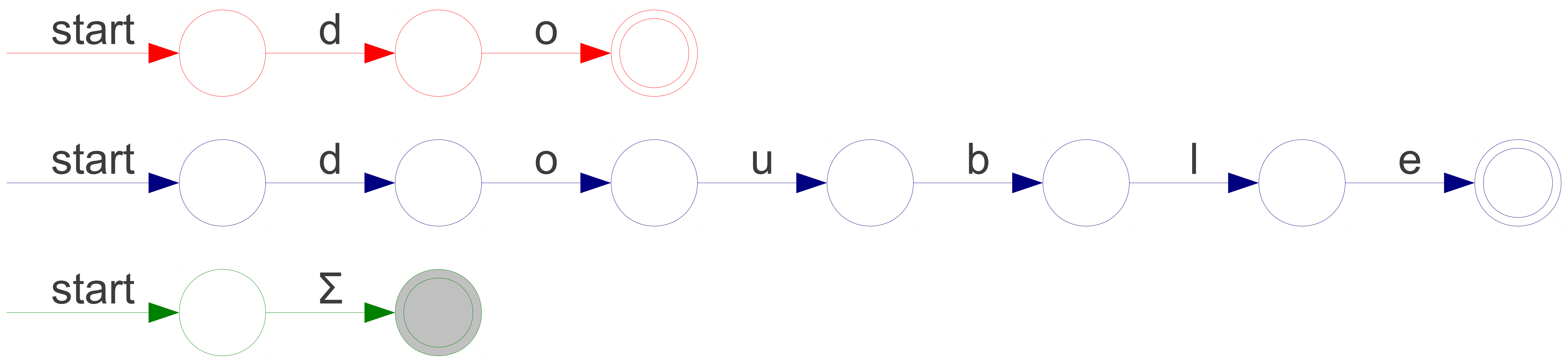
Implementing Maximal Munch

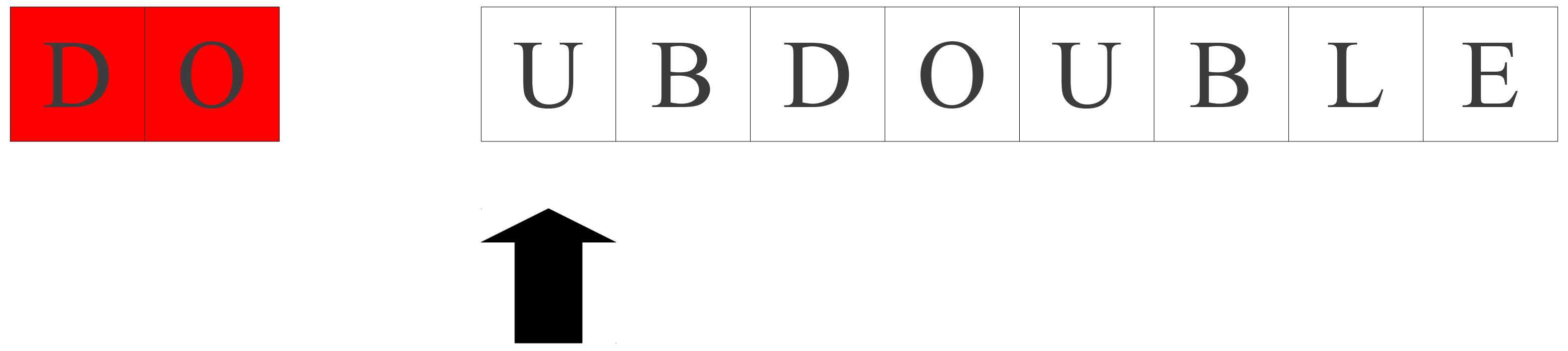
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



Implementing Maximal Munch

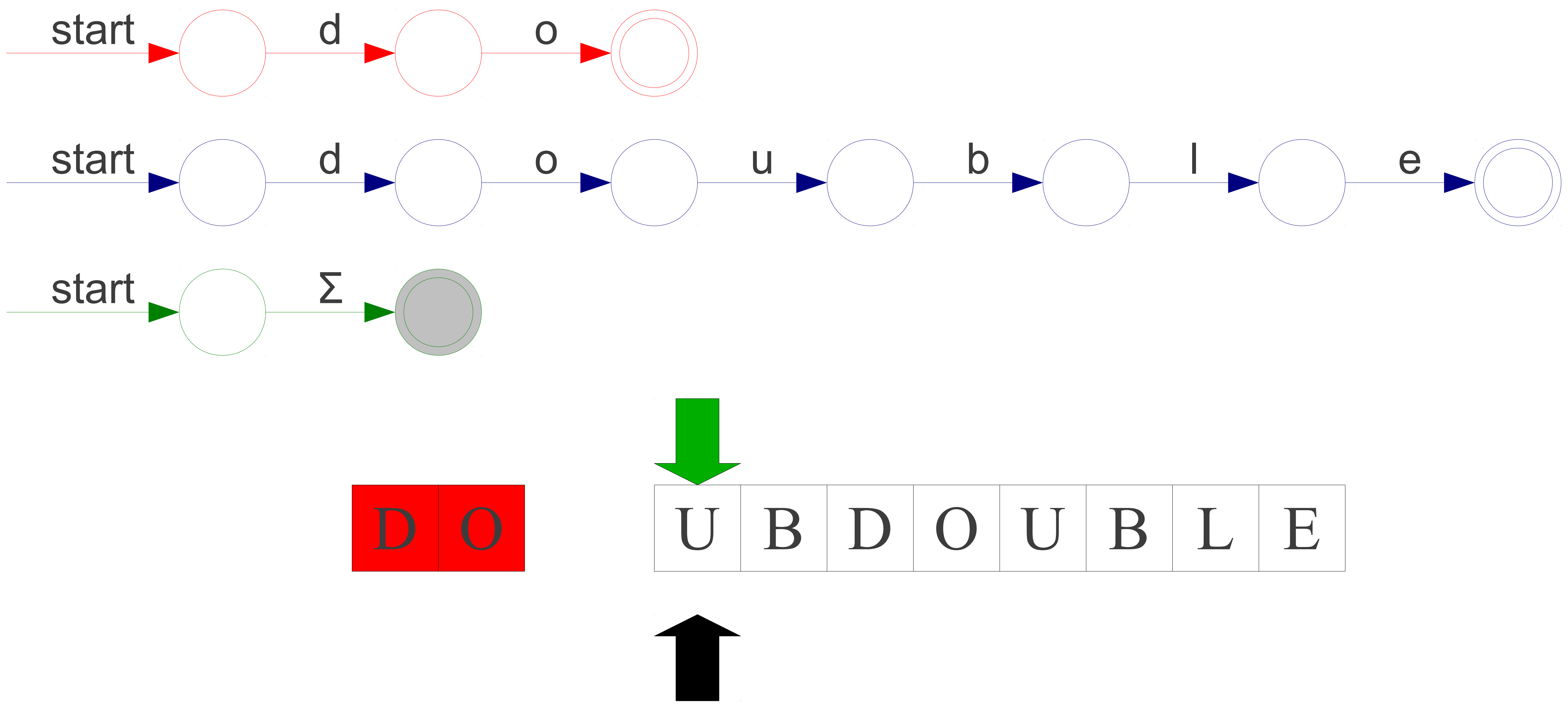
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





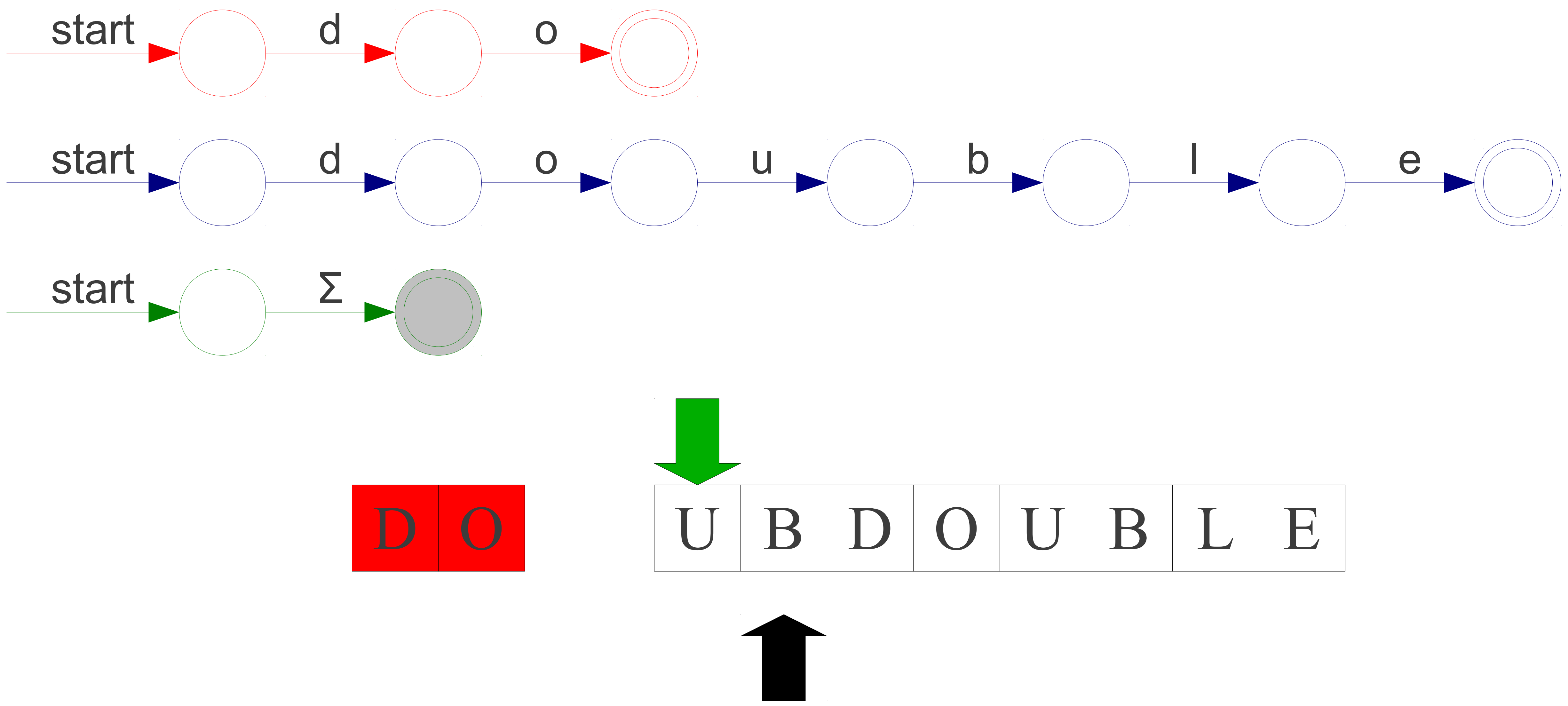
Implementing Maximal Munch

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



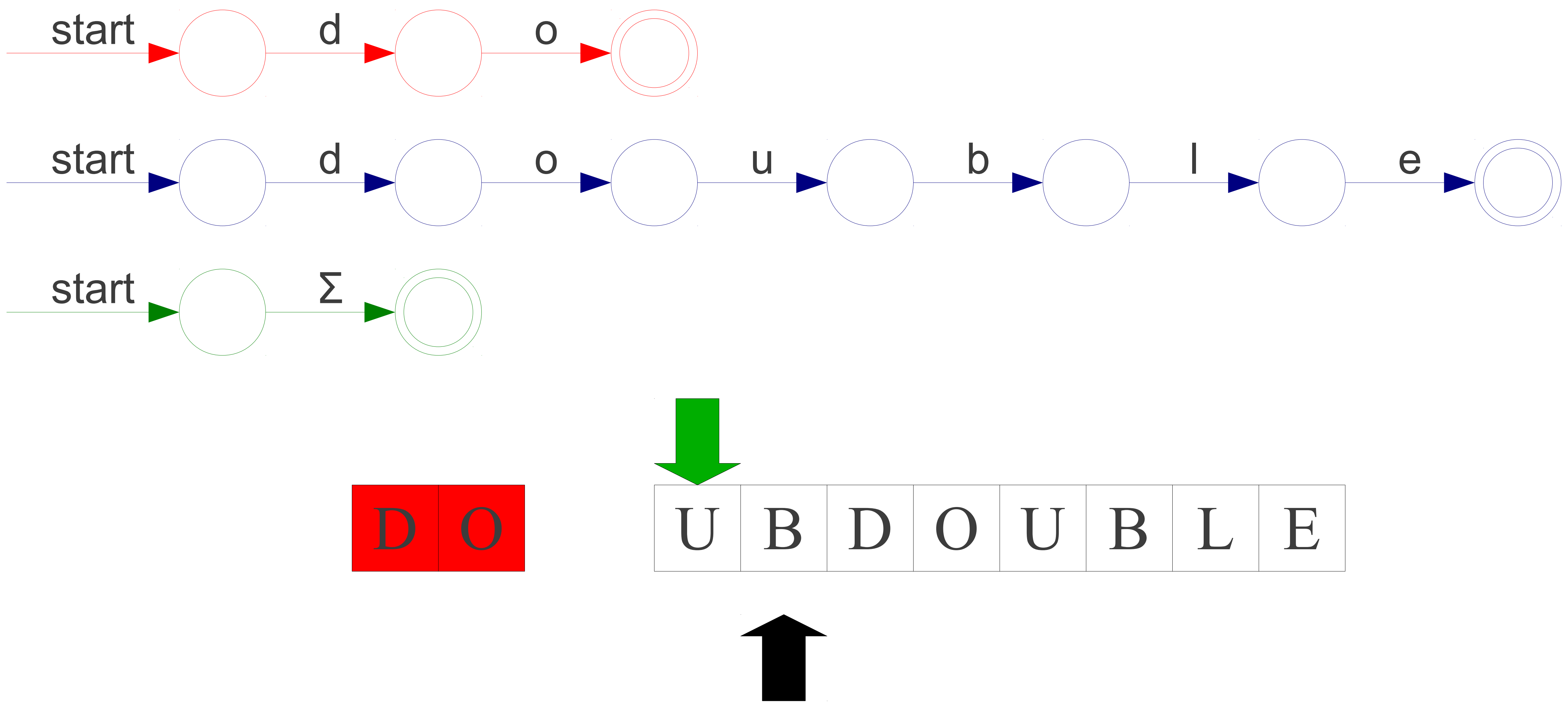
Implementing Maximal Munch

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



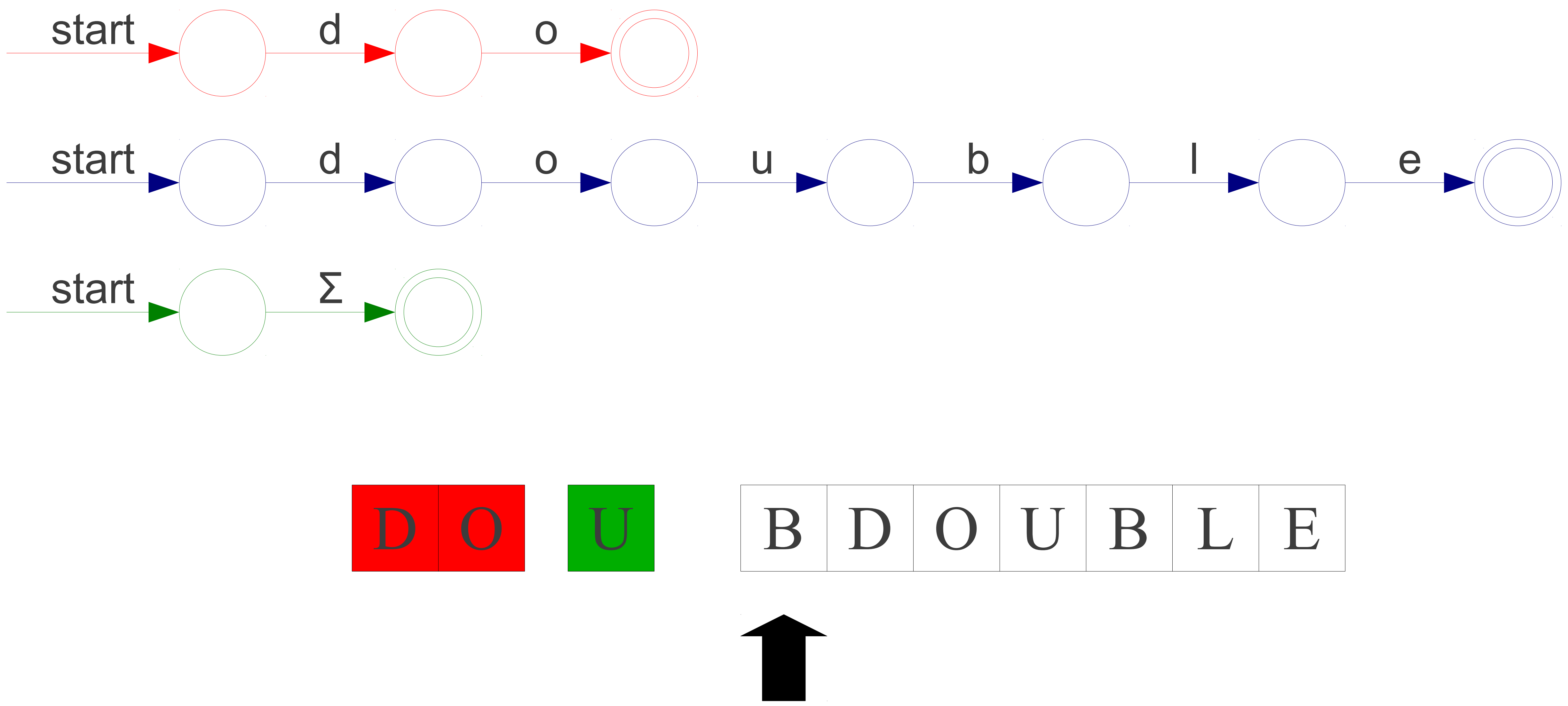
Implementing Maximal Munch

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



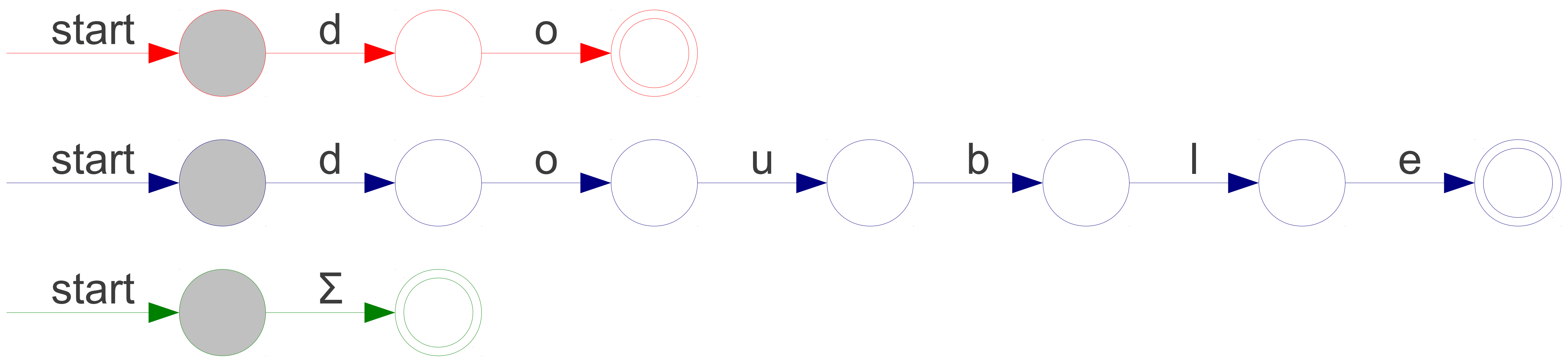
Implementing Maximal Munch

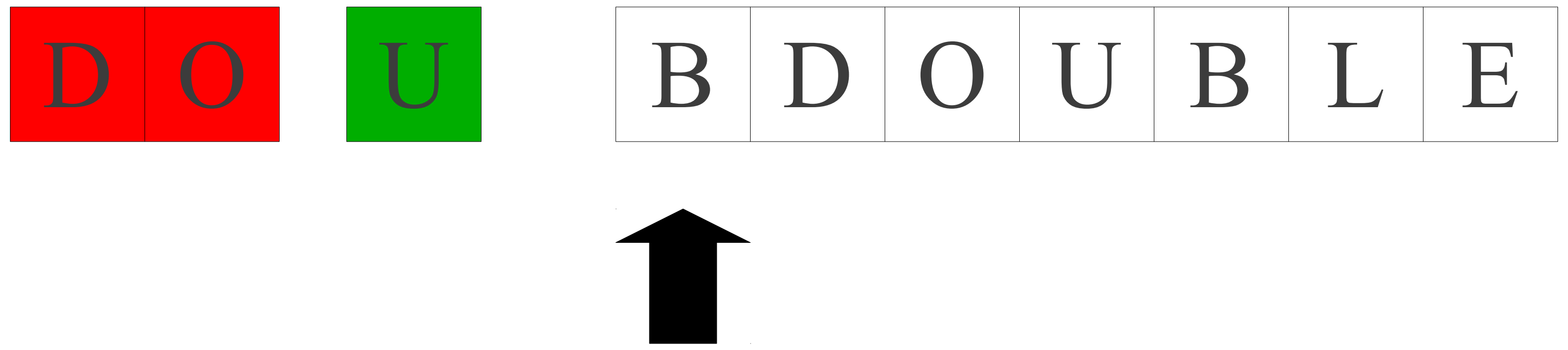
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

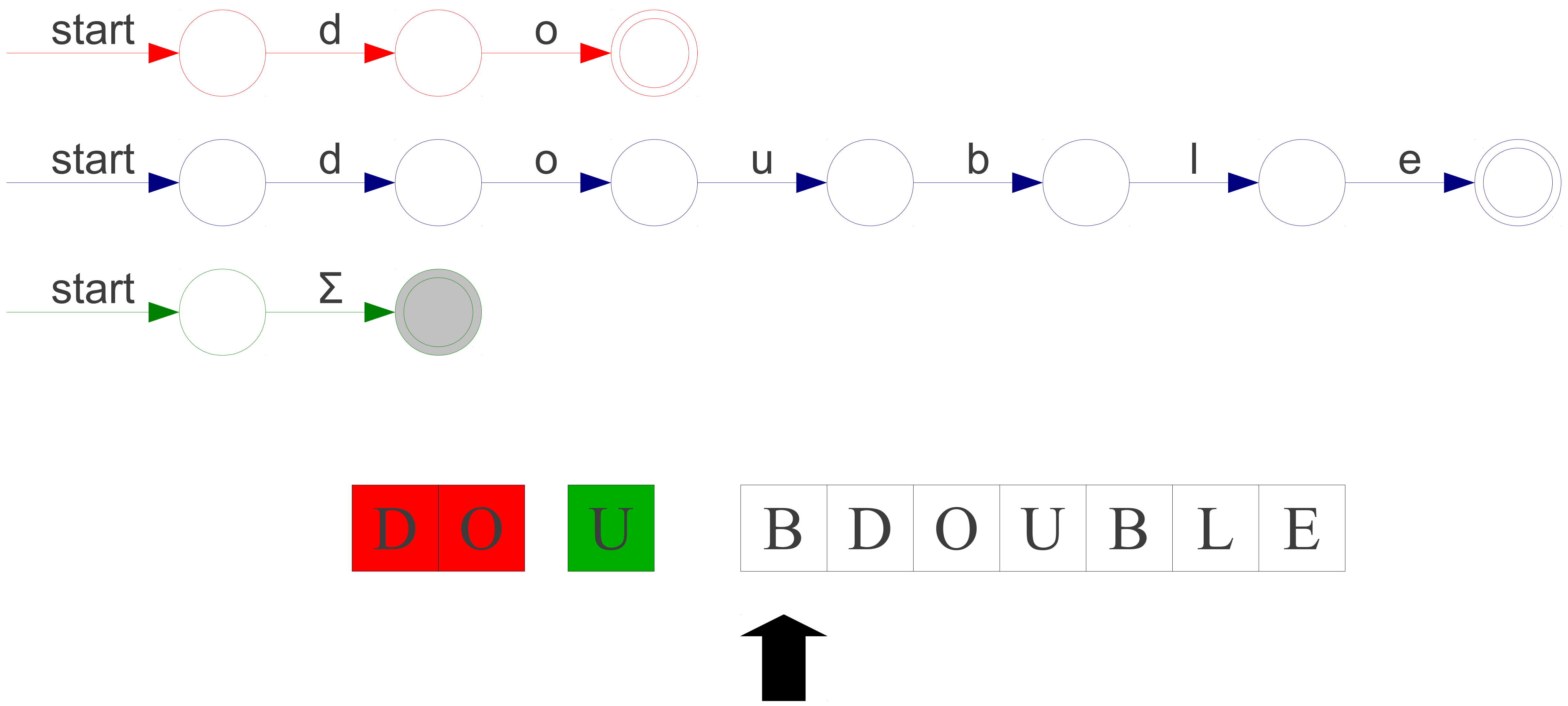
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





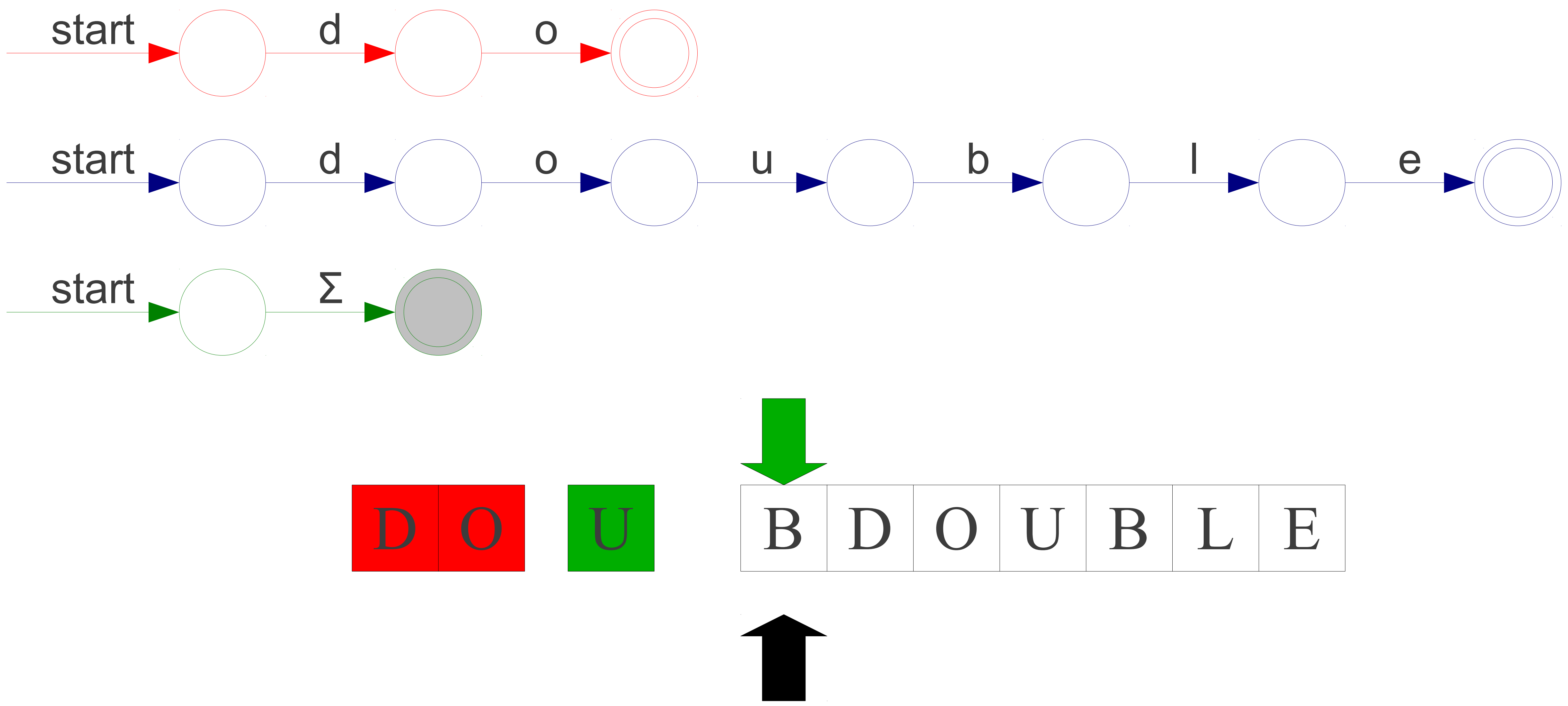
Implementing Maximal Munch

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



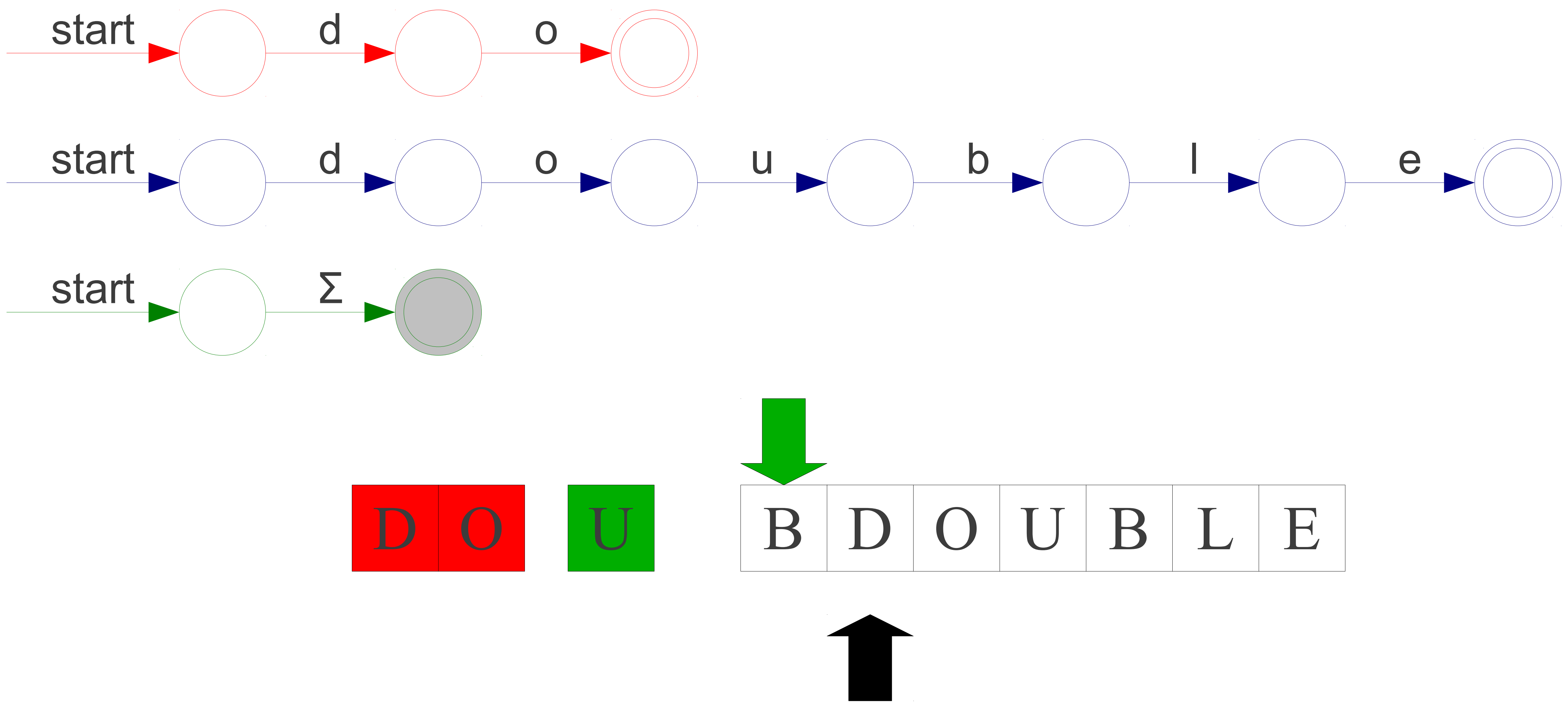
Implementing Maximal Munch

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



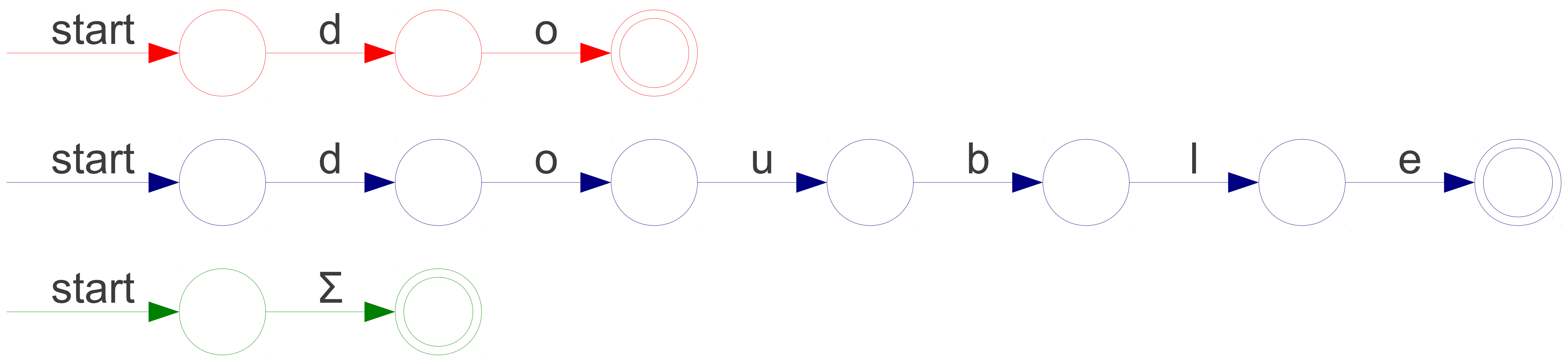
Implementing Maximal Munch

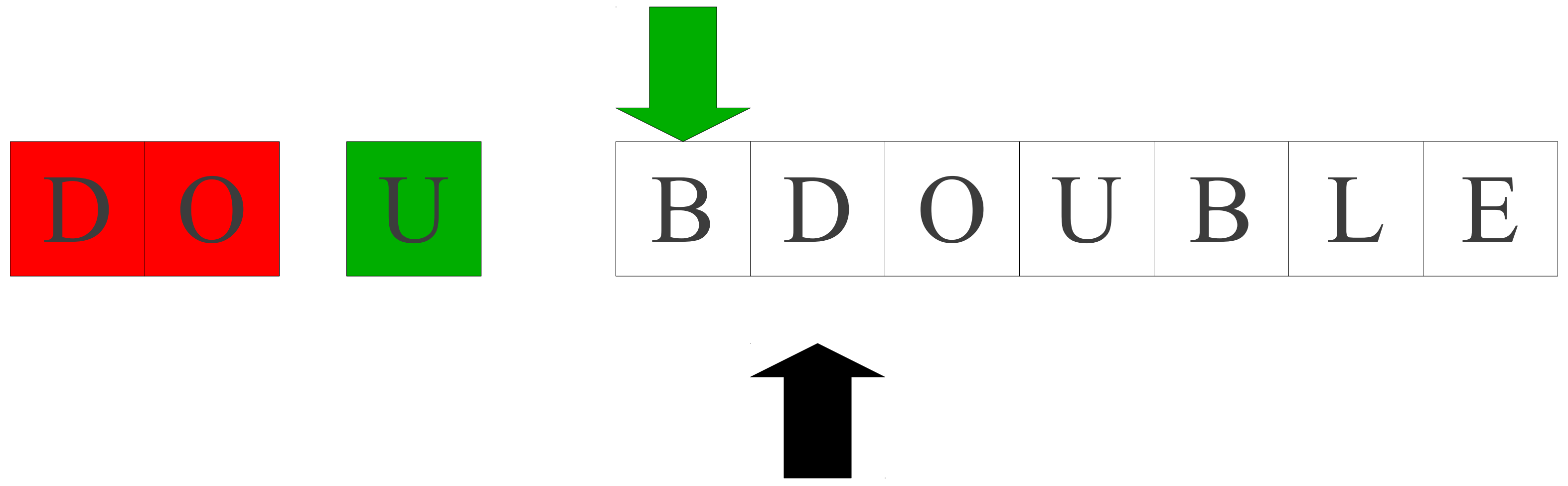
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

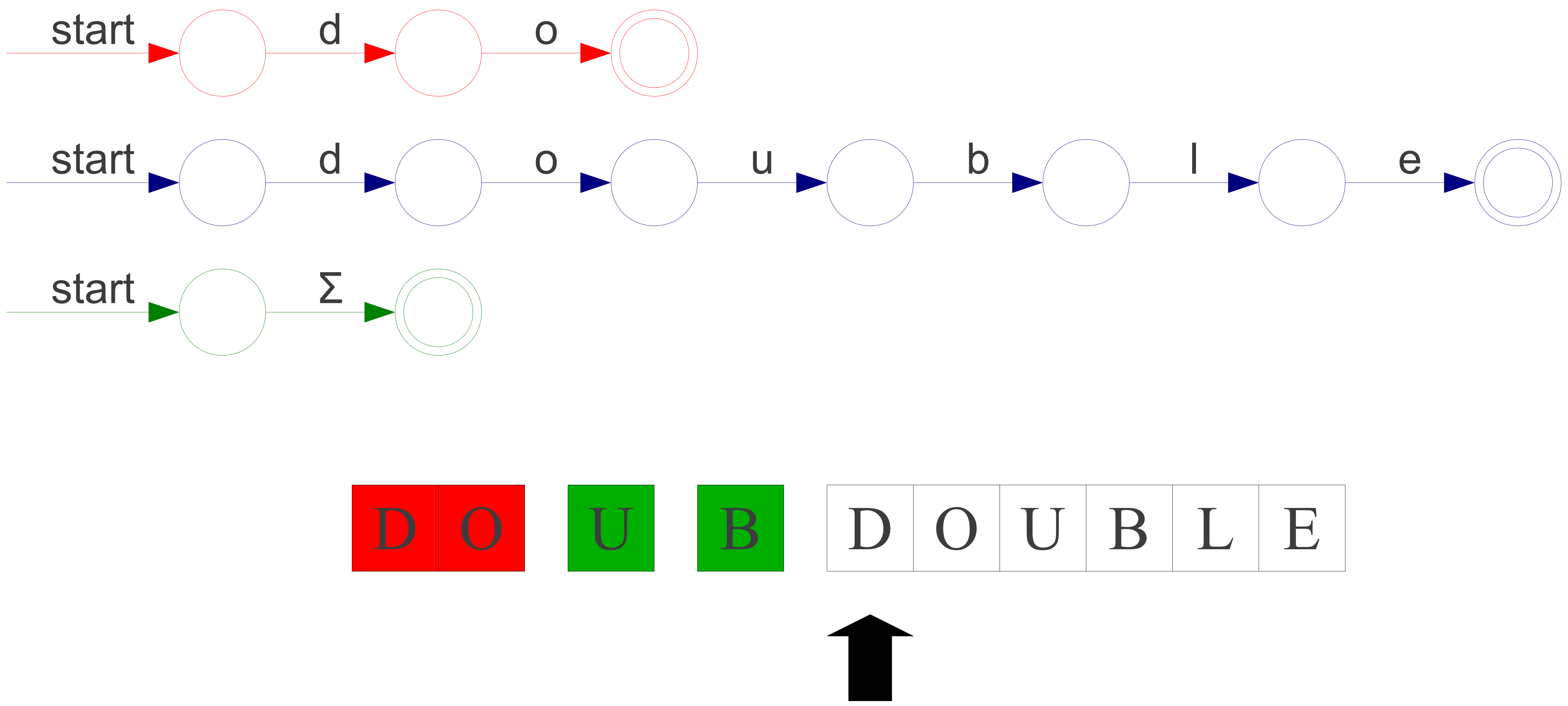
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





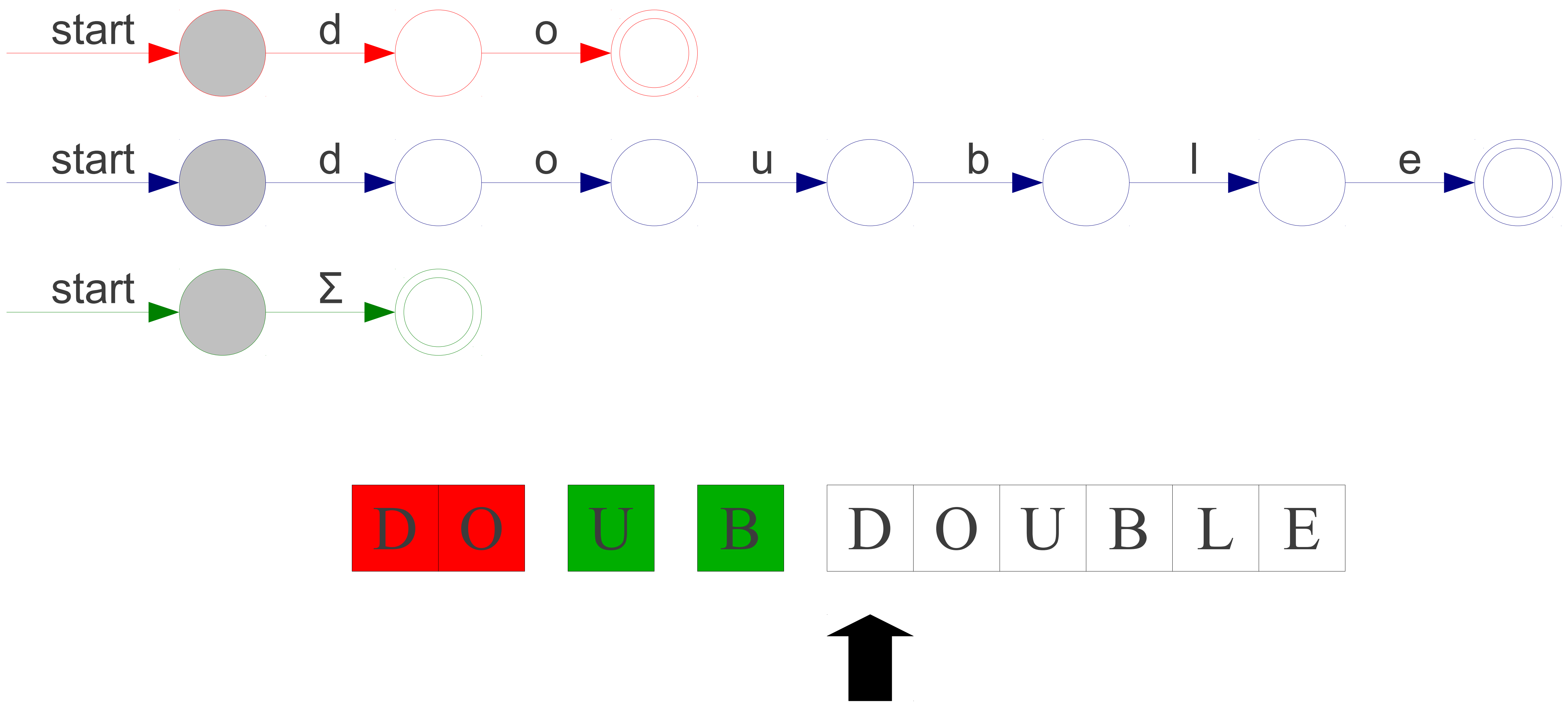
Implementing Maximal Munch

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



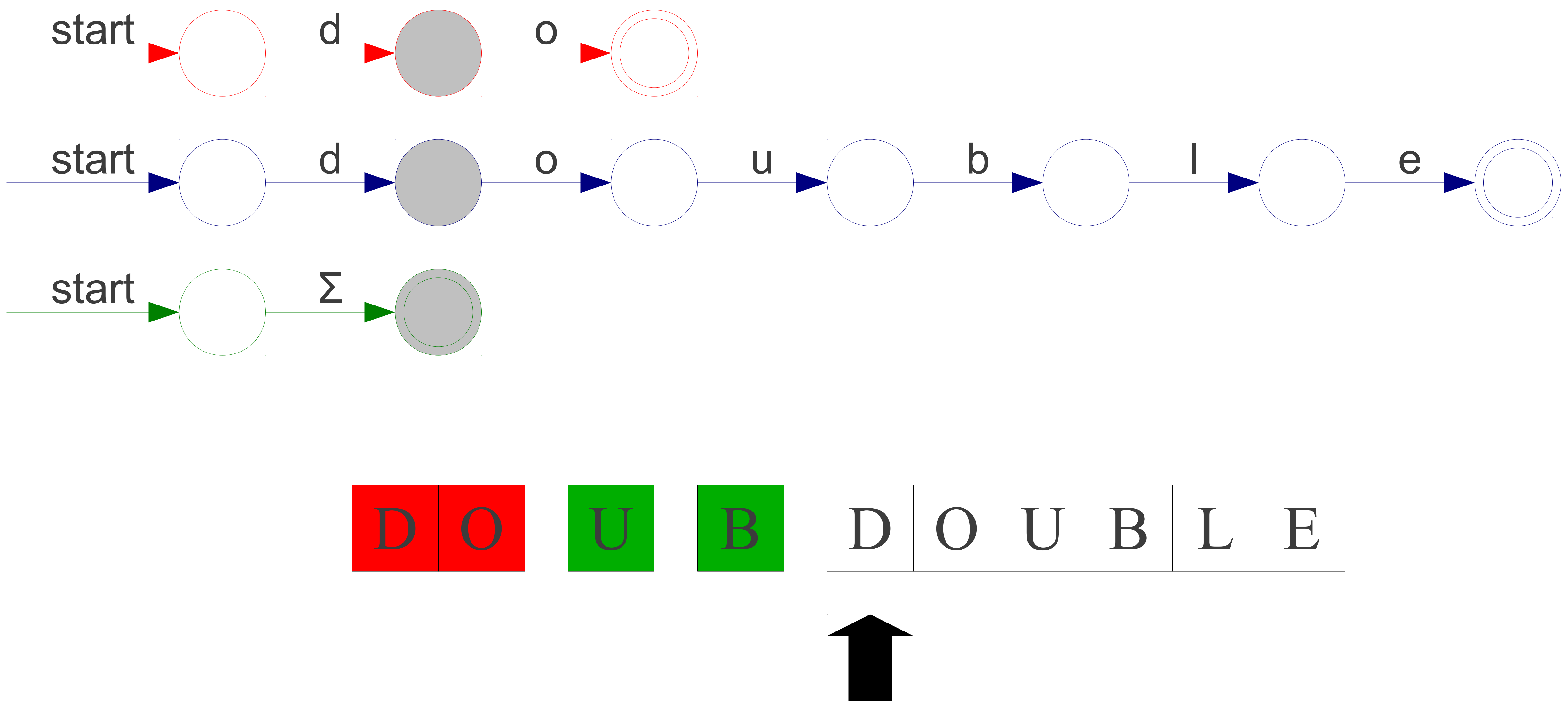
Implementing Maximal Munch

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



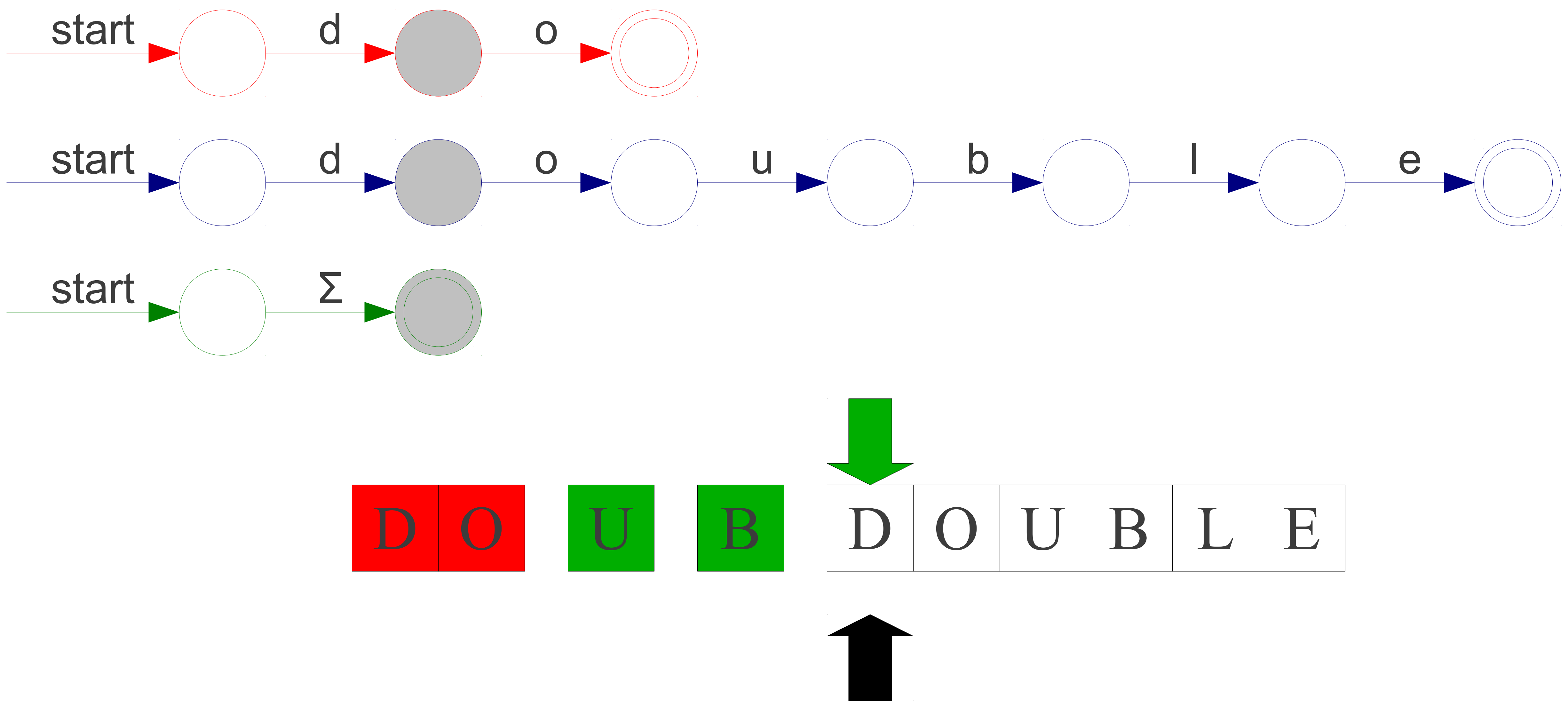
Implementing Maximal Munch

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



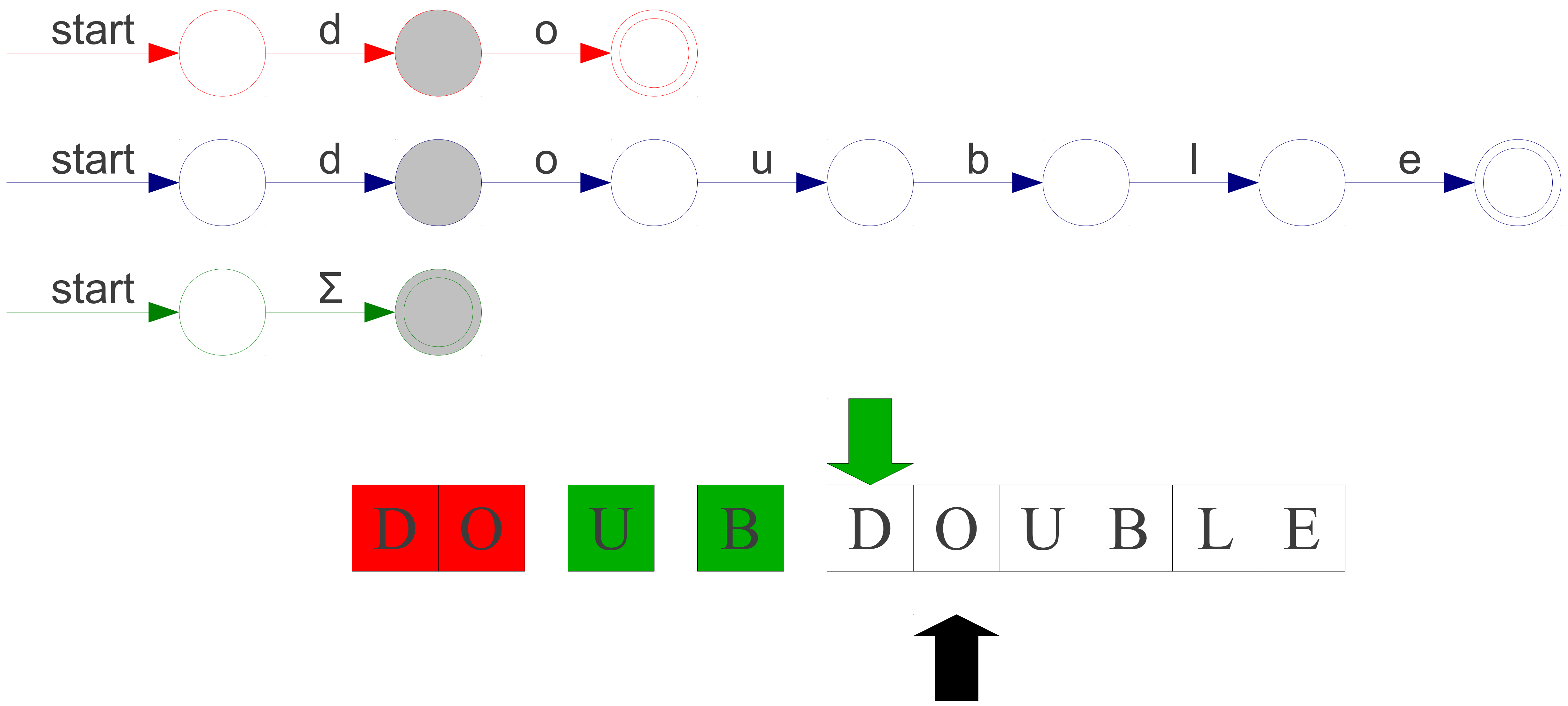
Implementing Maximal Munch

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



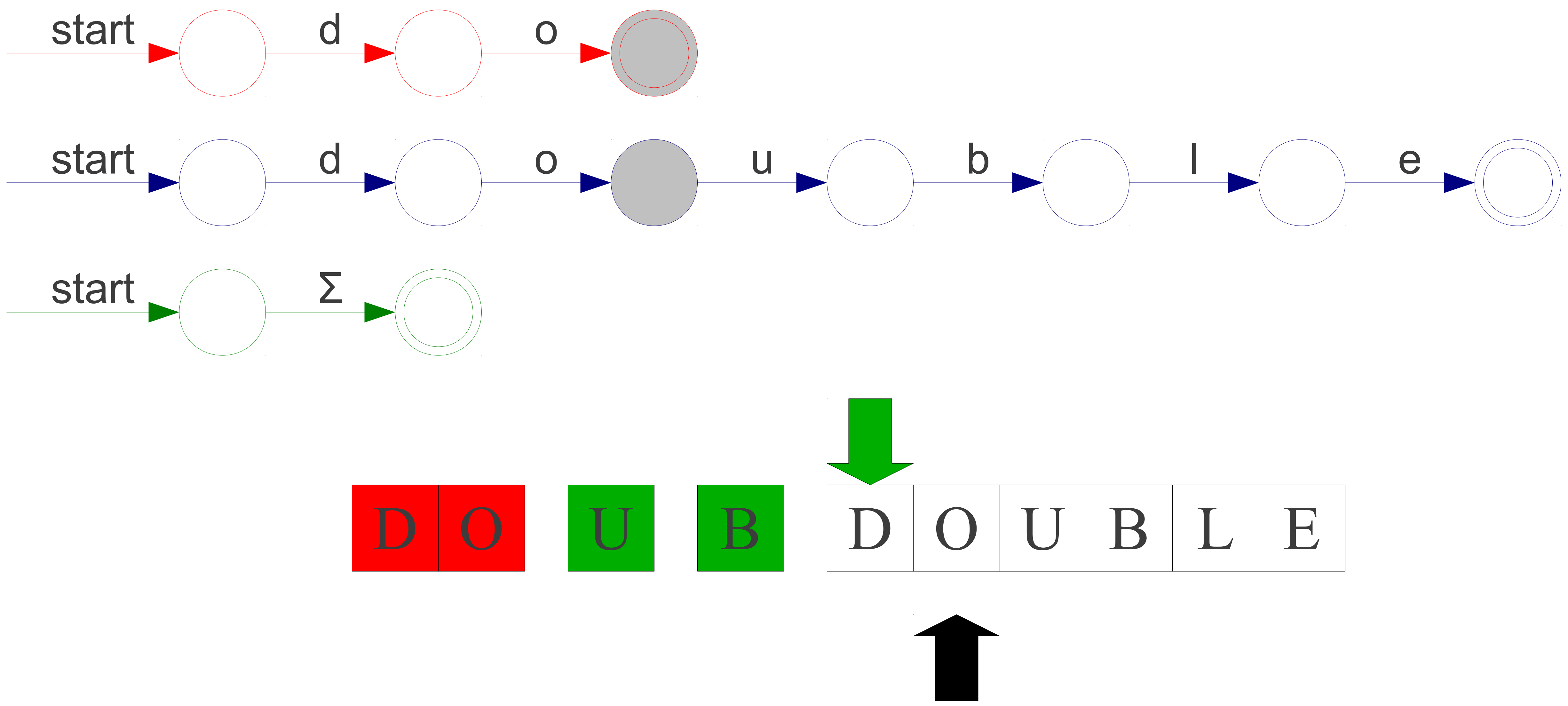
Implementing Maximal Munch

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



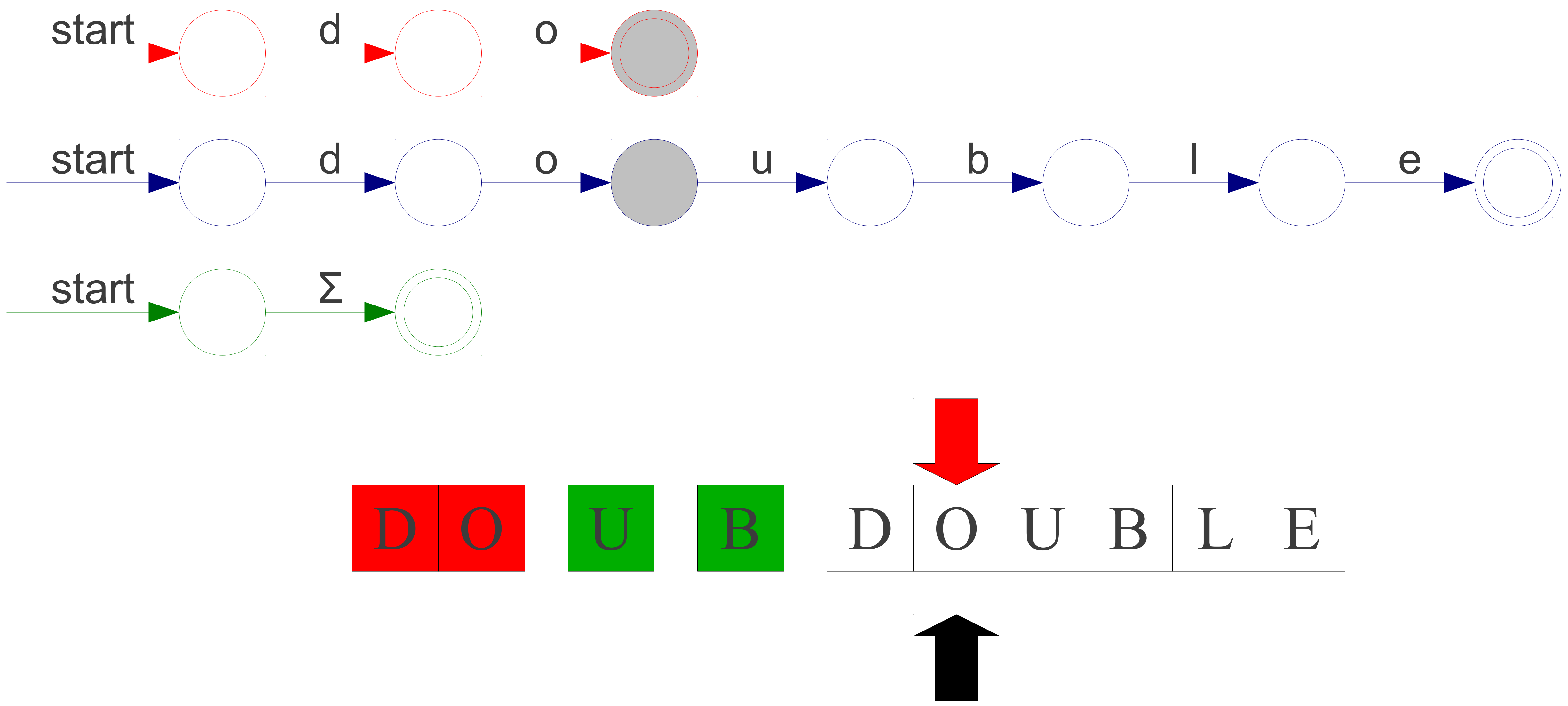
Implementing Maximal Munch

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



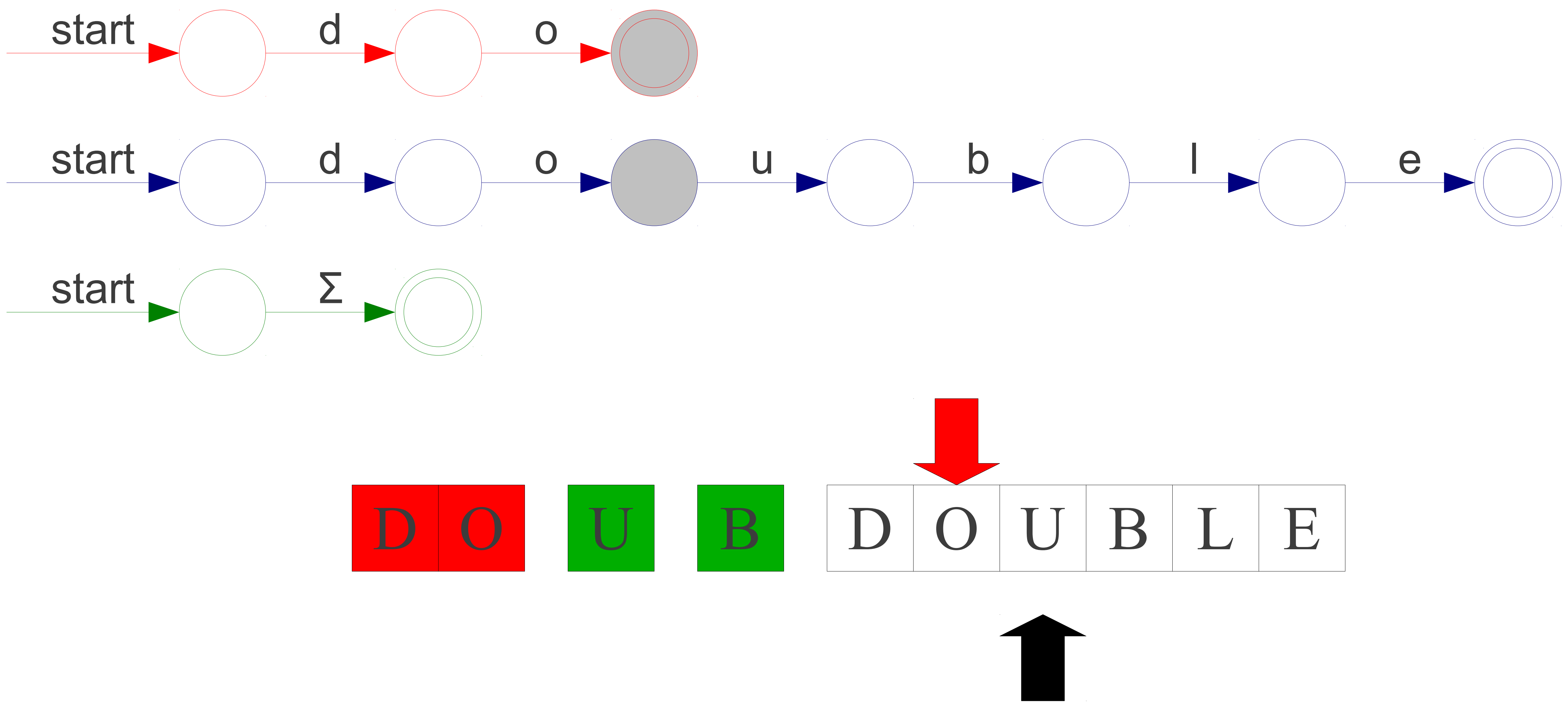
Implementing Maximal Munch

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



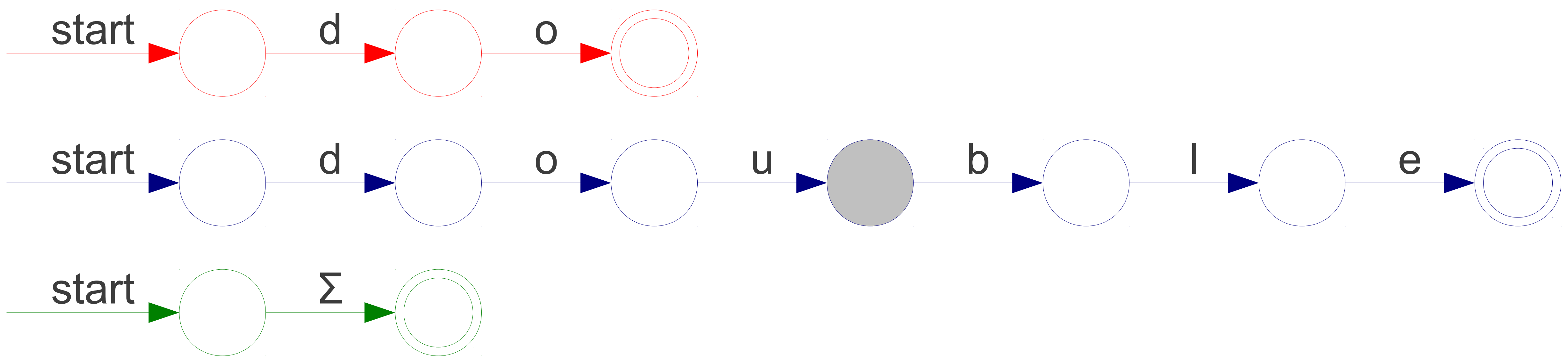
Implementing Maximal Munch

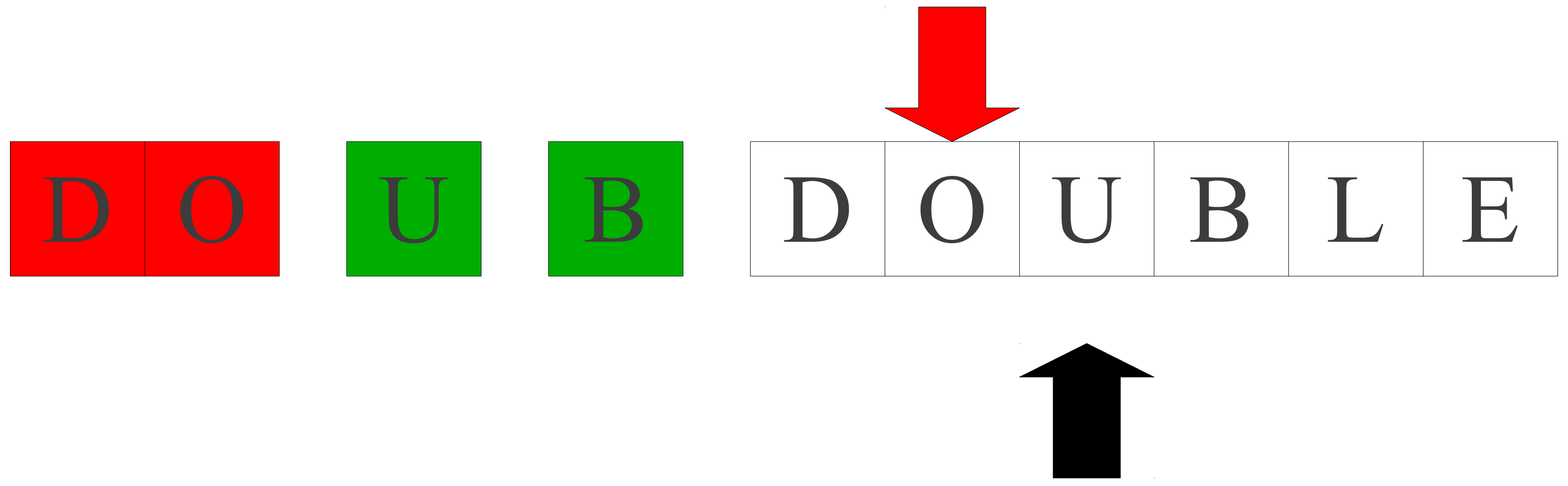
```
T_Do        do
T_Double    double
T_Mystery   [A-Za-z]
```



Implementing Maximal Munch

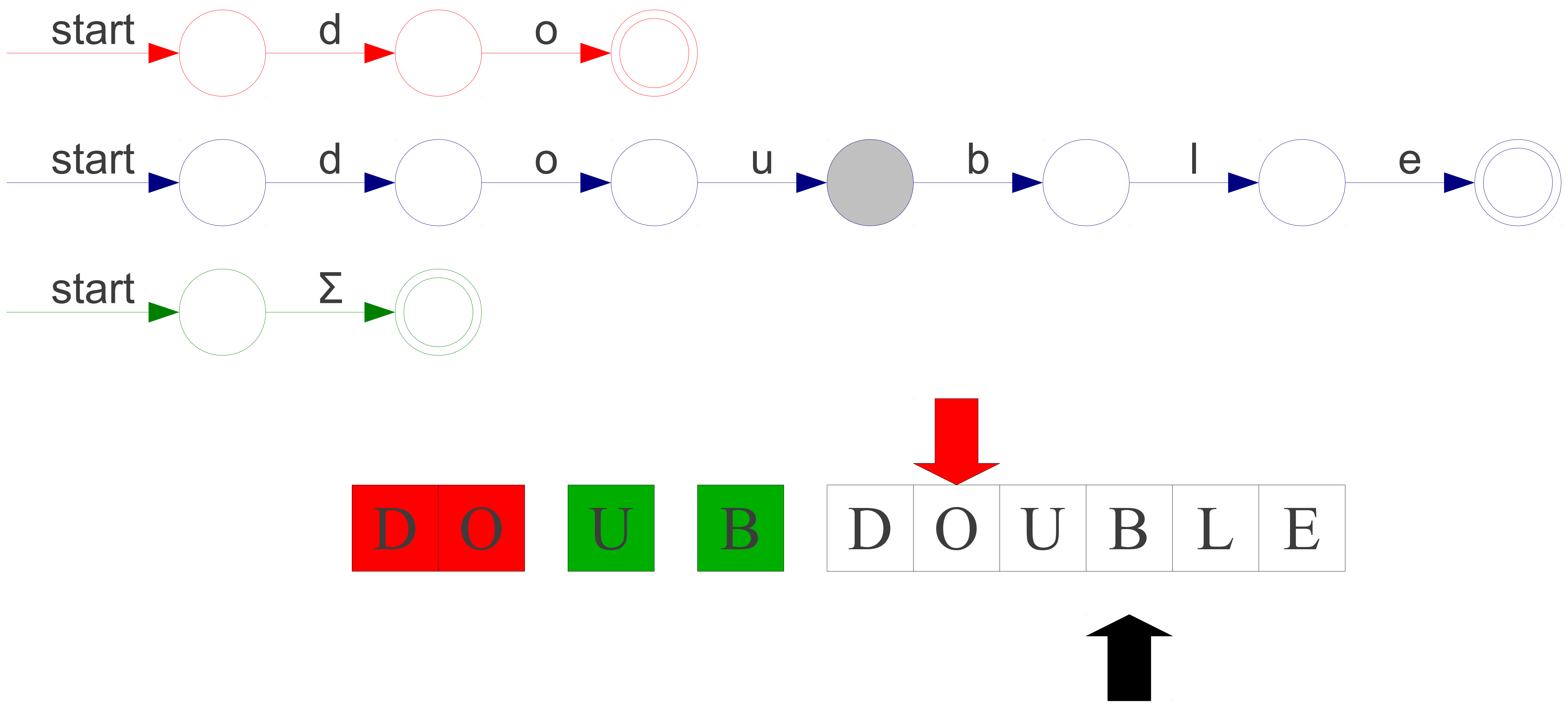
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





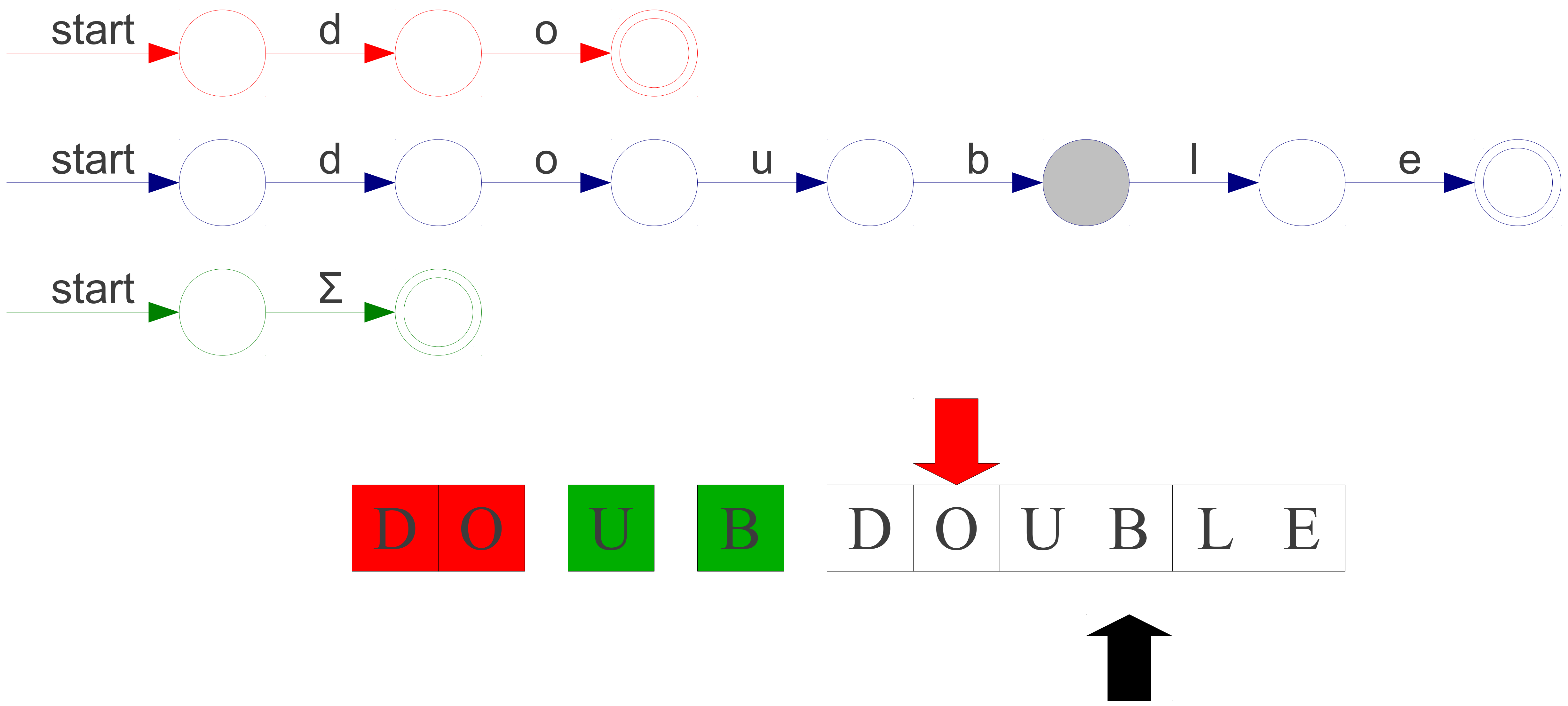
Implementing Maximal Munch

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



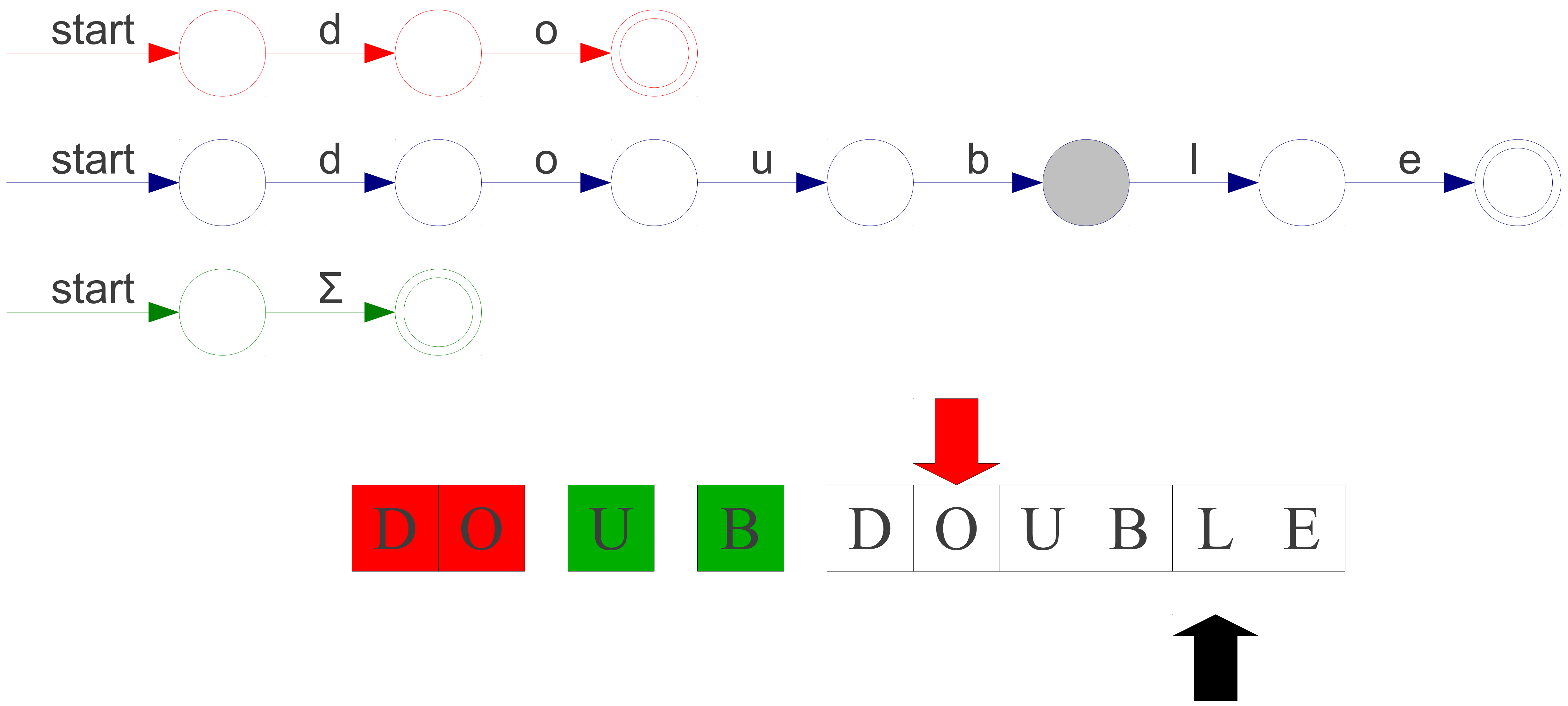
Implementing Maximal Munch

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



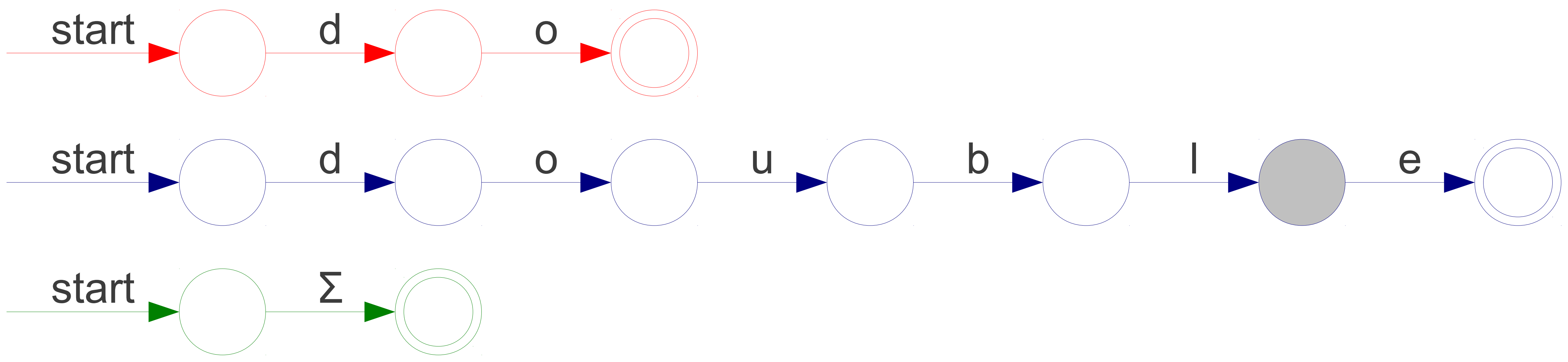
Implementing Maximal Munch

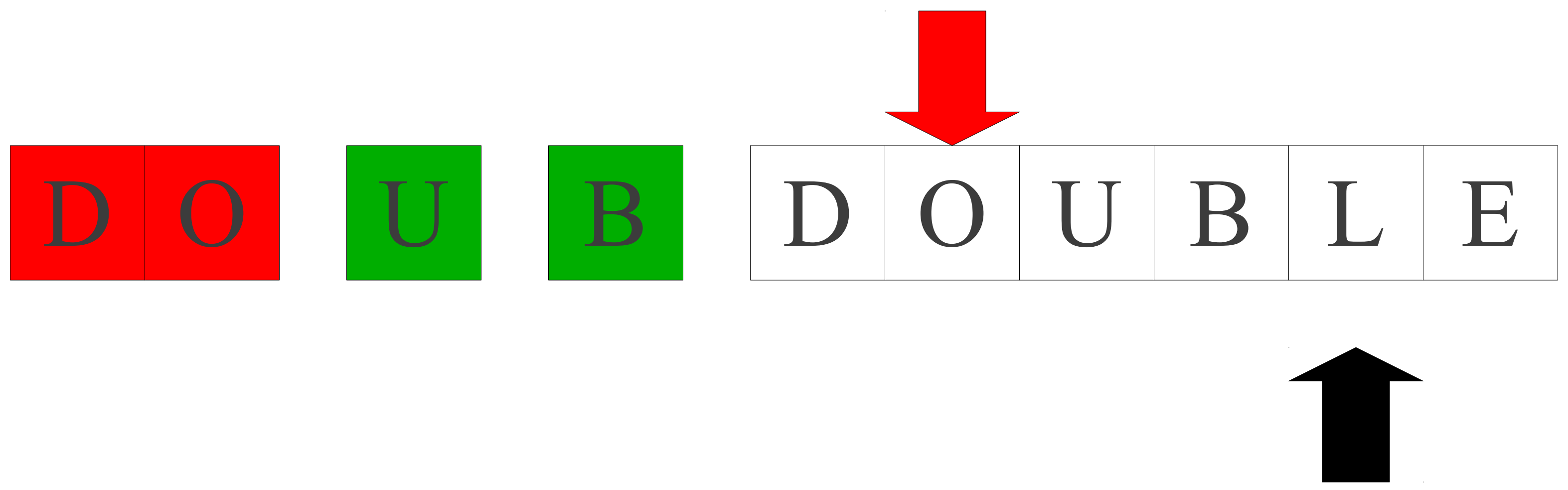
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

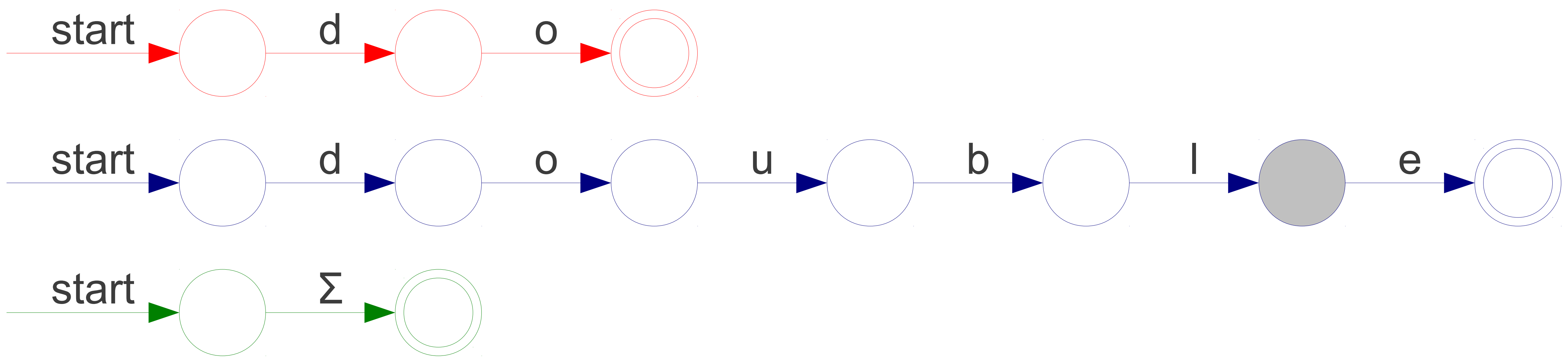
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

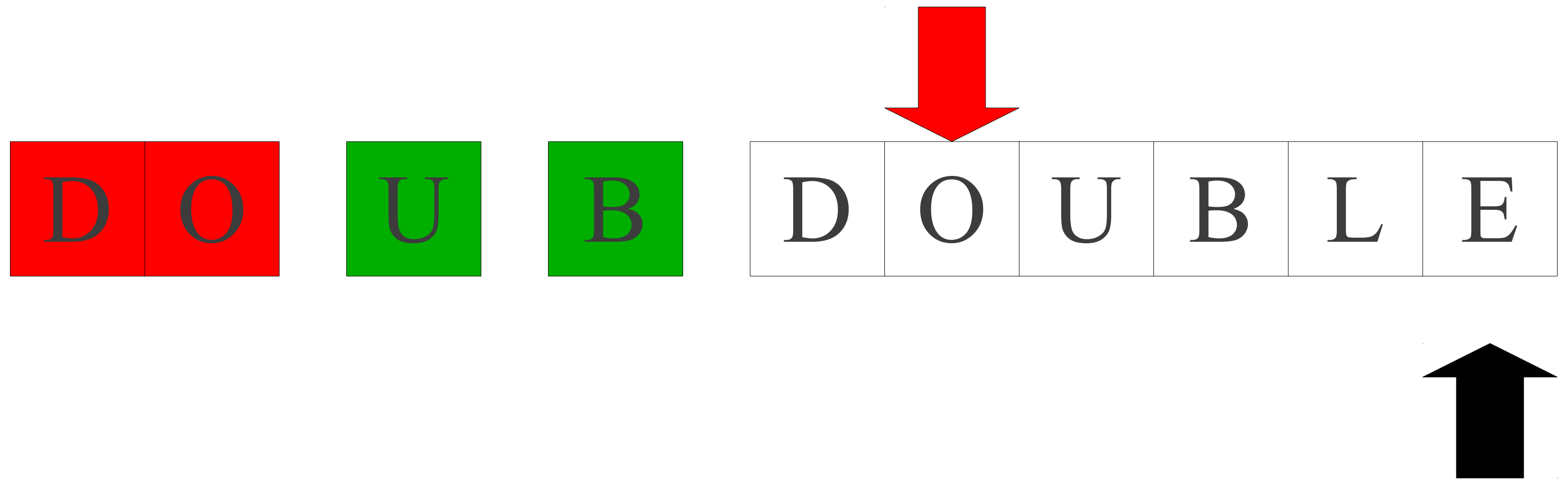




Implementing Maximal Munch

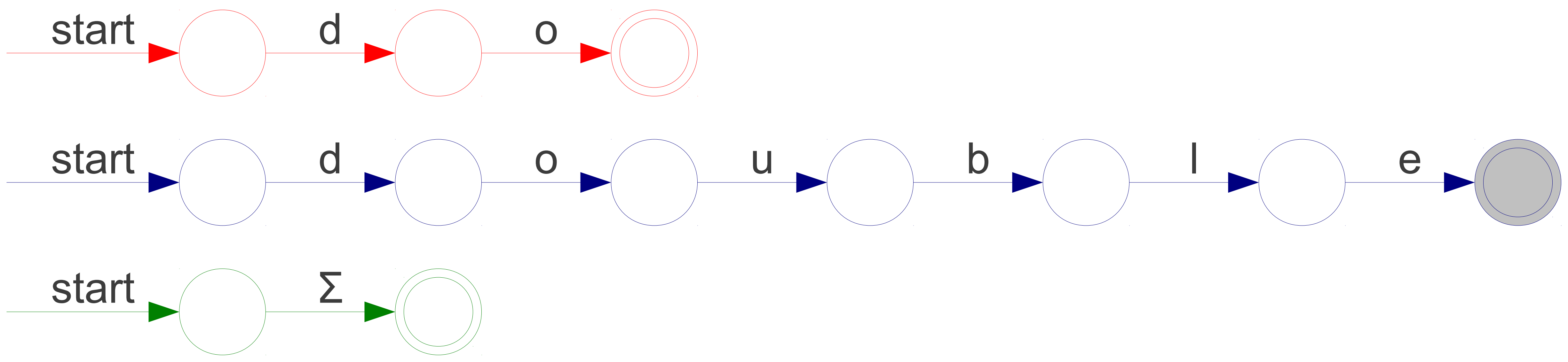
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

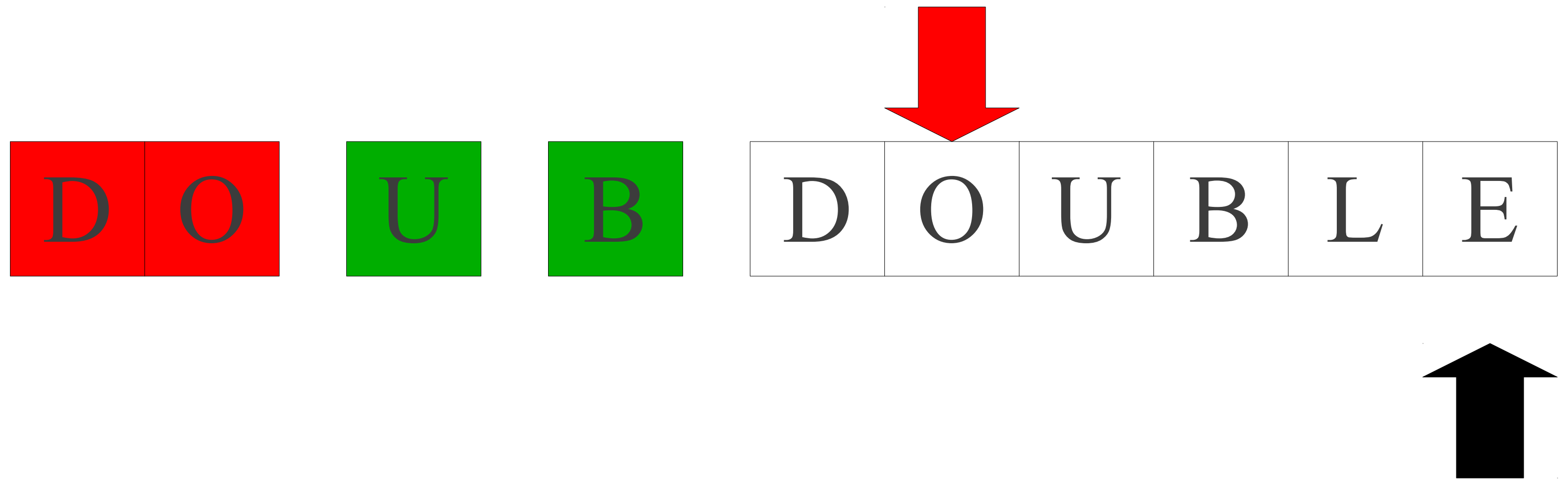




Implementing Maximal Munch

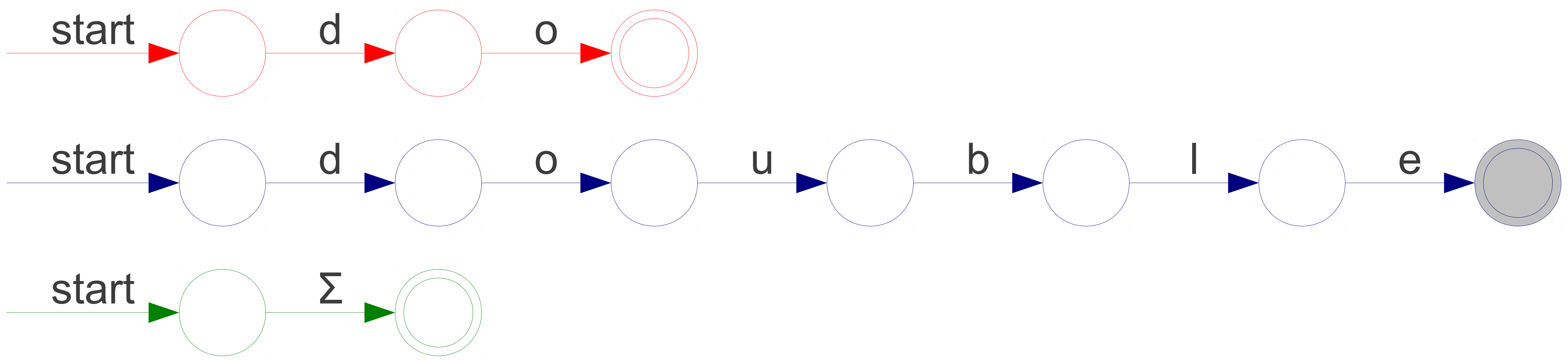
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

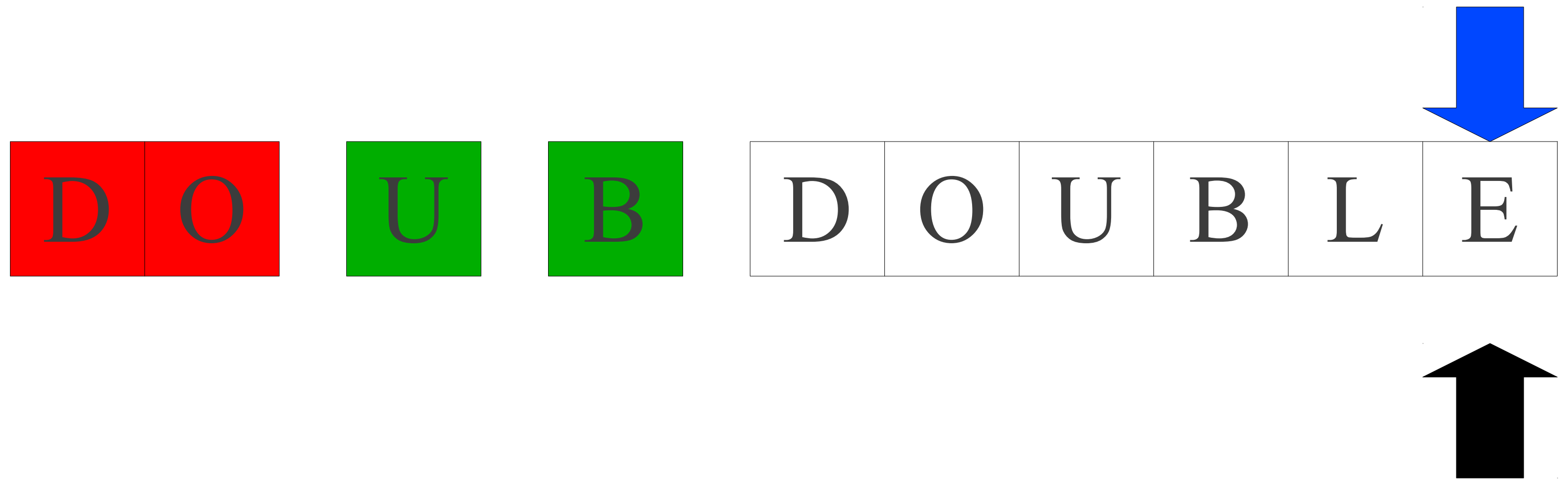




Implementing Maximal Munch

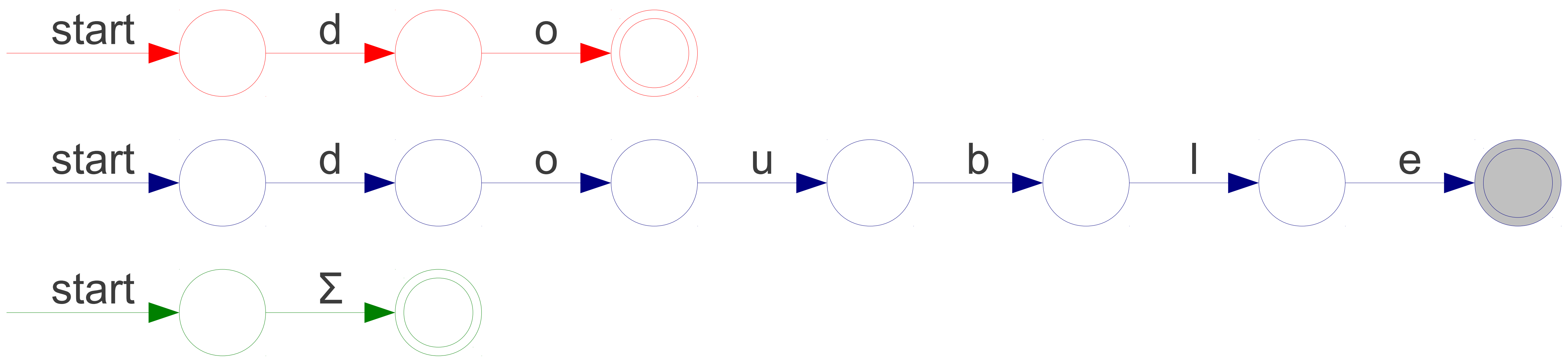
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

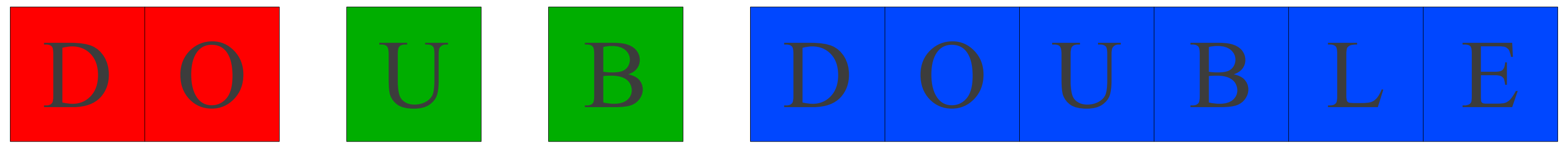




Implementing Maximal Munch

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





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

Lexer Generators as Compilers for Regexes

Source Language: Regexes + associated Token-construction code

Target Language: C or the lang the rest of your compiler is written in

Intermediate Representations: DFAs, NFAs
Passes: NFA -> DFA determinization
Optimization: DFA minimization
Can be mathematically proven to be correct, "optimal"