EEECS 483: Compiler Construction

Lecture 21:
Lexing Part 2, Automata

April 2
Winter Semester 2025

Reminders

Midterm Regrade Requests due this week

Assignment 4 due on Friday

Assignment 5 to be released on Monday

Lexer Generators as Compilers

Lexing is tedious and error-prone to implement manually. Just like assembly code!

Instead, implement a **lexer generator**, a compiler for a domain-specific language for lexers.

Just like the compilers we've been working on this semester:

1. Design a source **language** for lexers: **regular expressions** + action code
2. Describe its **semantics**: regular expressions are a syntax for **formal languages**
3. Transform into an intermediate representation: **non-deterministic finite automata**
4. **Optimize** that intermediate representation: **determinize** and **minimize** the NFA into a DFA
5. **Generate** code from the optimized IR

Terminology

A **regular expression** R is an expression built up from epsilon, empty, single characters, disjunction, sequencing and Kleene star

The semantics of a regular expression is that it represents a **formal language** a subset of all possible input strings

A **recognizer** for a regular expression R, is a function String -> Bool that outputs true if and only if the input string is in the formal language described by the regular expression

The core of implementing a lexer is implementing **recognizers** for regular expressions.

But it's not the entirety: we also need to be able to find the **longest match** for multiple regular expressions.

Recognizing Regular Languages

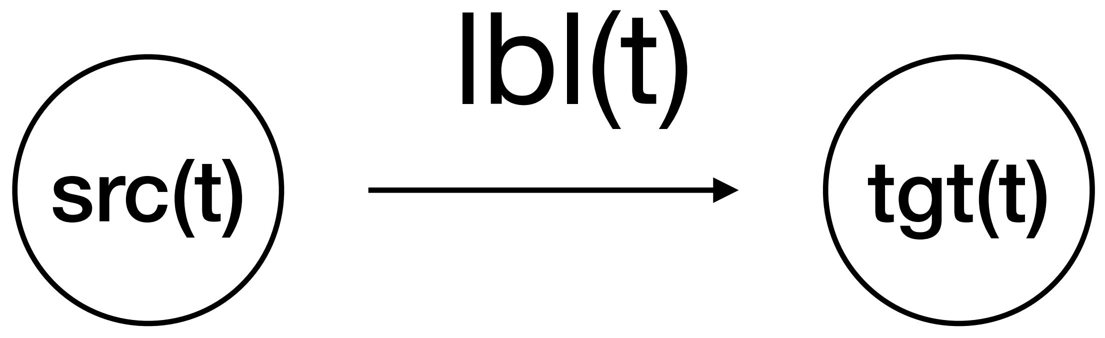
How can we efficiently implement a recognizer for a regular language?

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

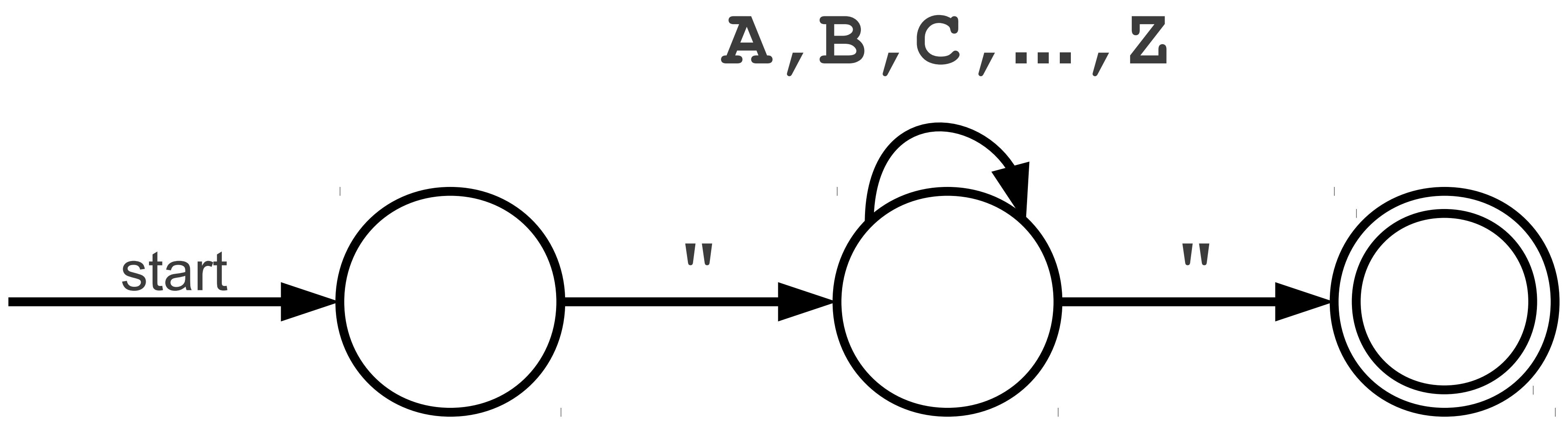
Finite State Automata

A (non-deterministic) finite state automaton over an alphabet Σ consists of

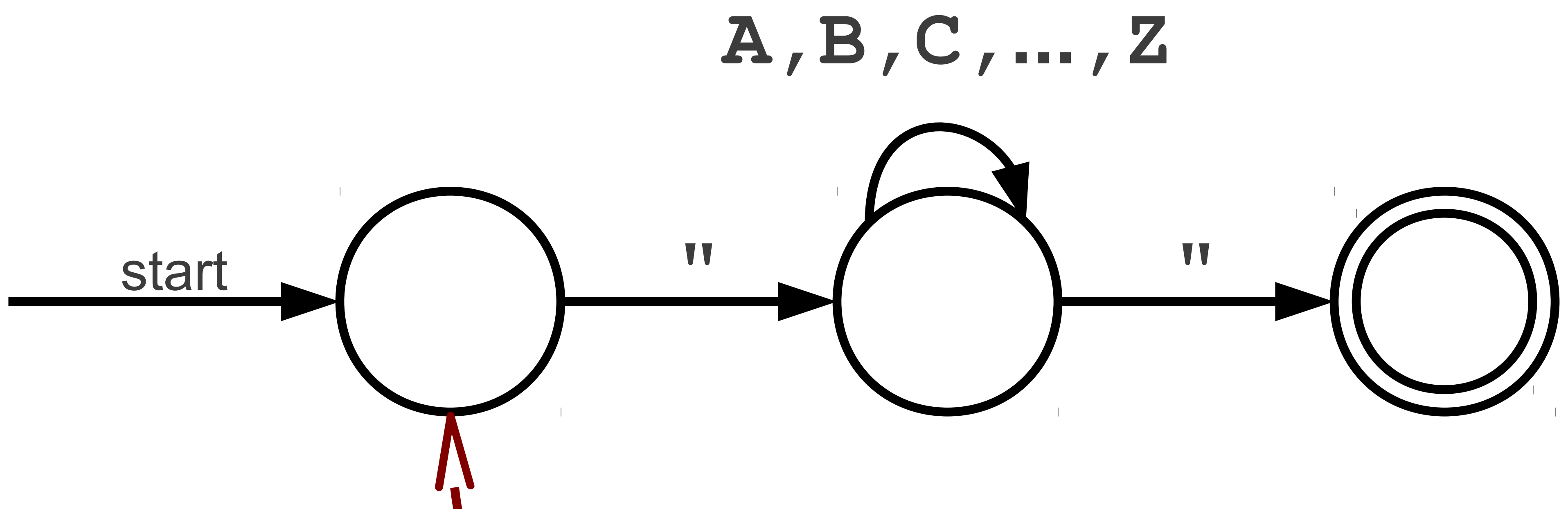
- A finite set of states **S**
- A distinguished start state $\mathbf{s_0} \in \mathbf{S}$
- A subset of accepting states $\mathbf{Acc} \subseteq \mathbf{S}$
- A set of transitions δ, where each transition $t \in \delta$ has
  - a source state **src**(t)
  - a target state **tgt**(t)
  - a label **lbl**(t), which is either a character $c \in \Sigma$ or ε



A Simple Automaton

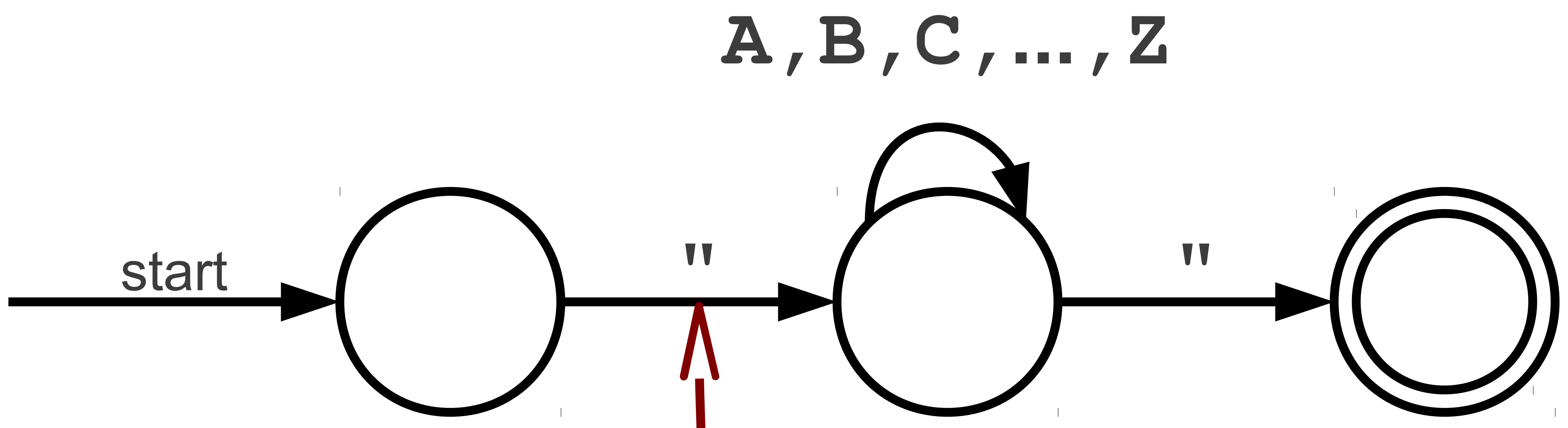


A Simple Automaton



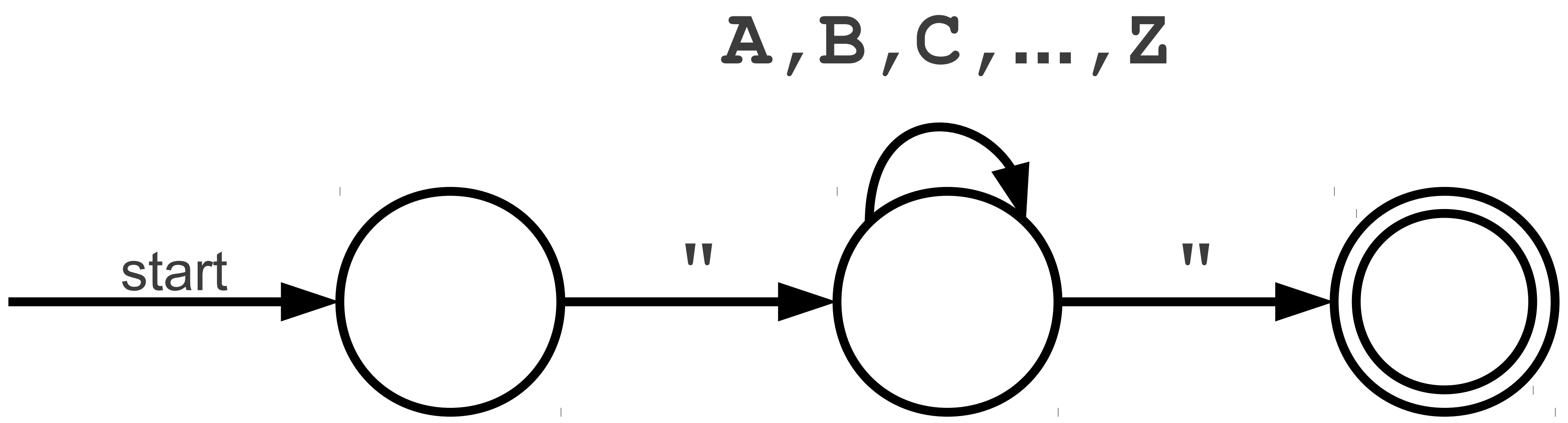
Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

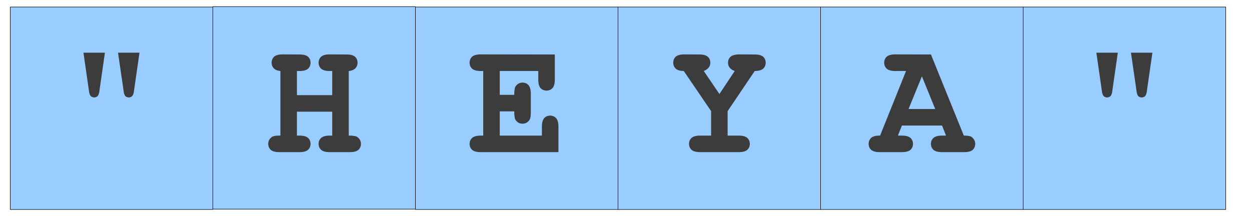
A Simple Automaton



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

A Simple Automaton

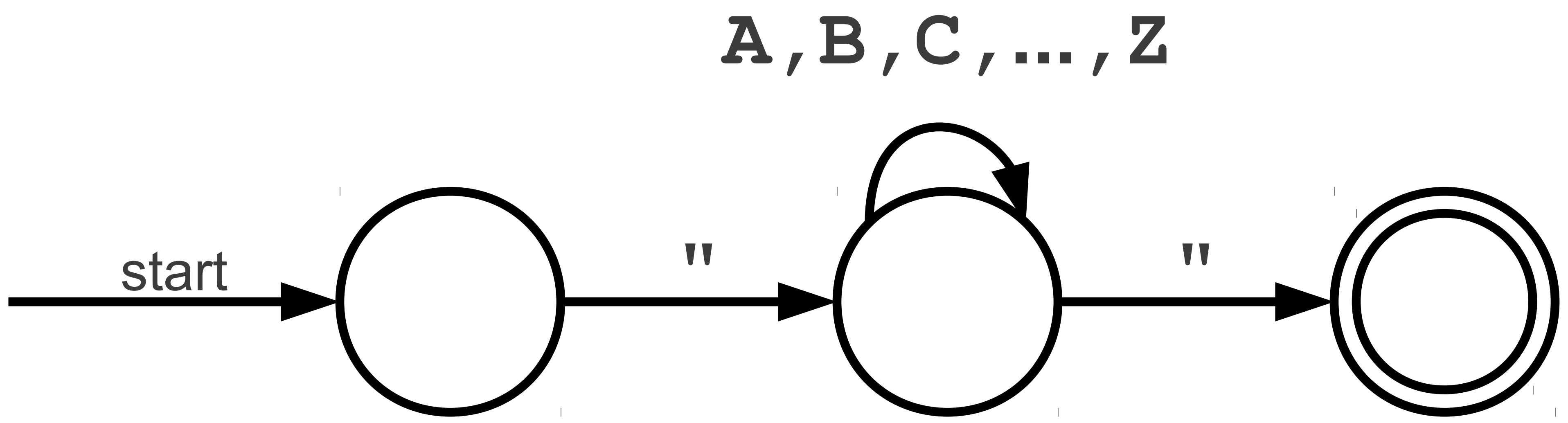




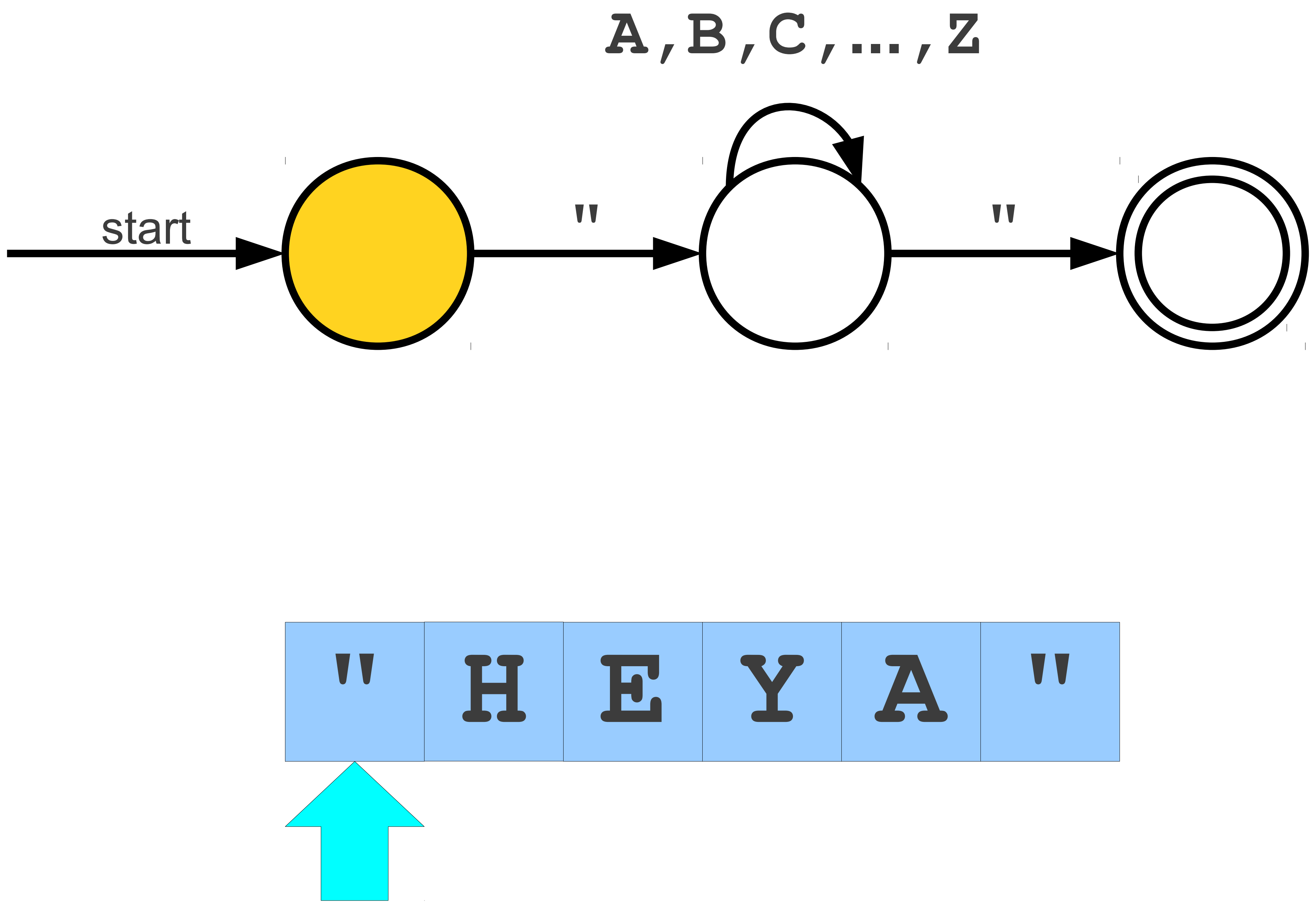
Finite Automata: Takes an input string and determines whether it's a valid sentence of a language

accept or reject

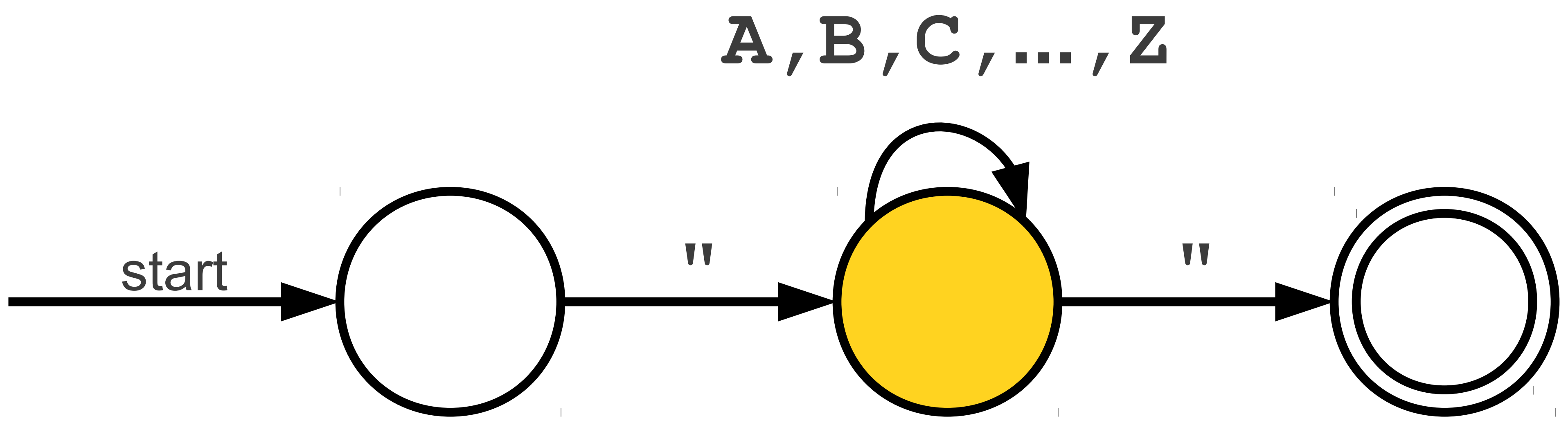
A Simple Automaton



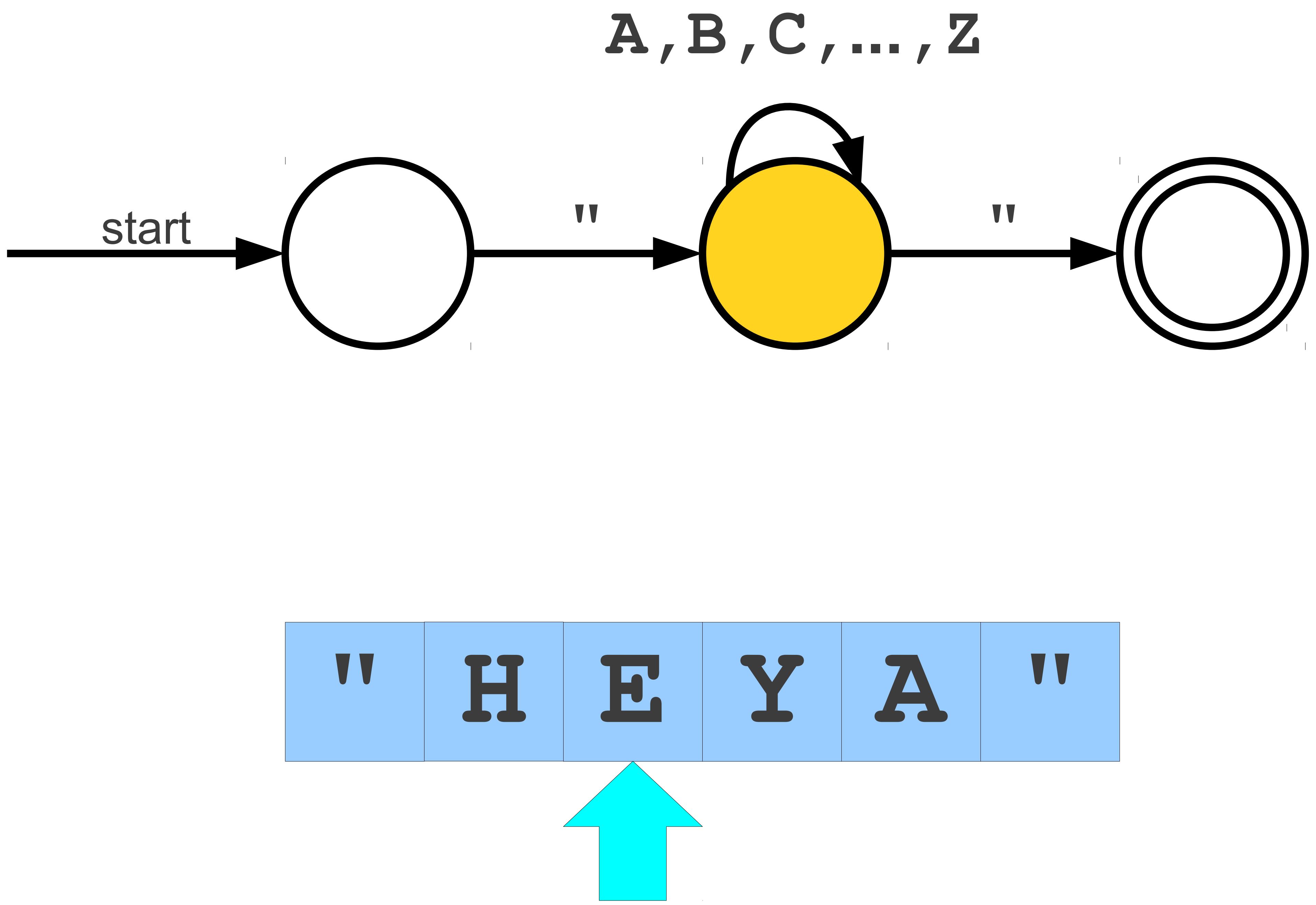
A Simple Automaton



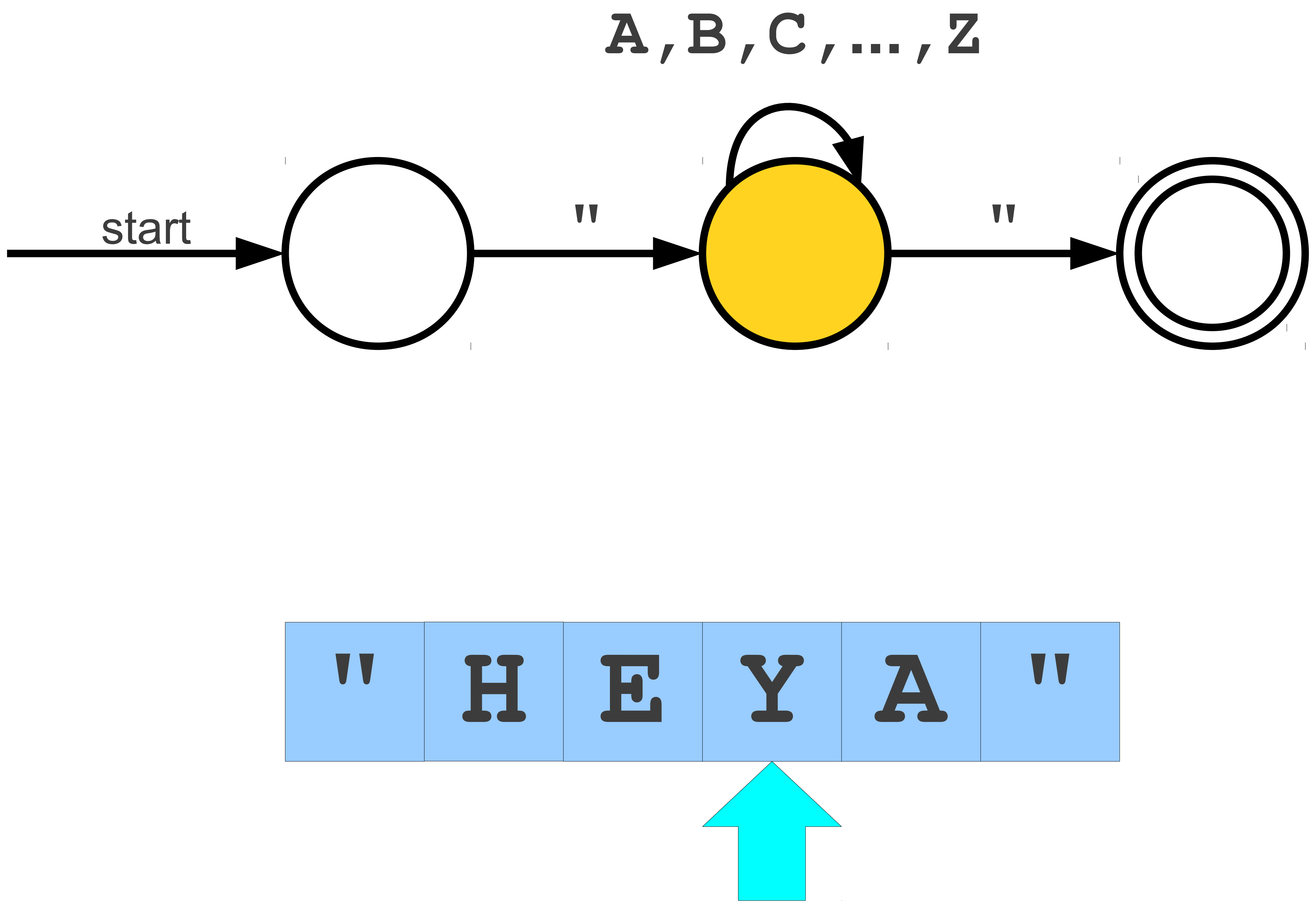
A Simple Automaton



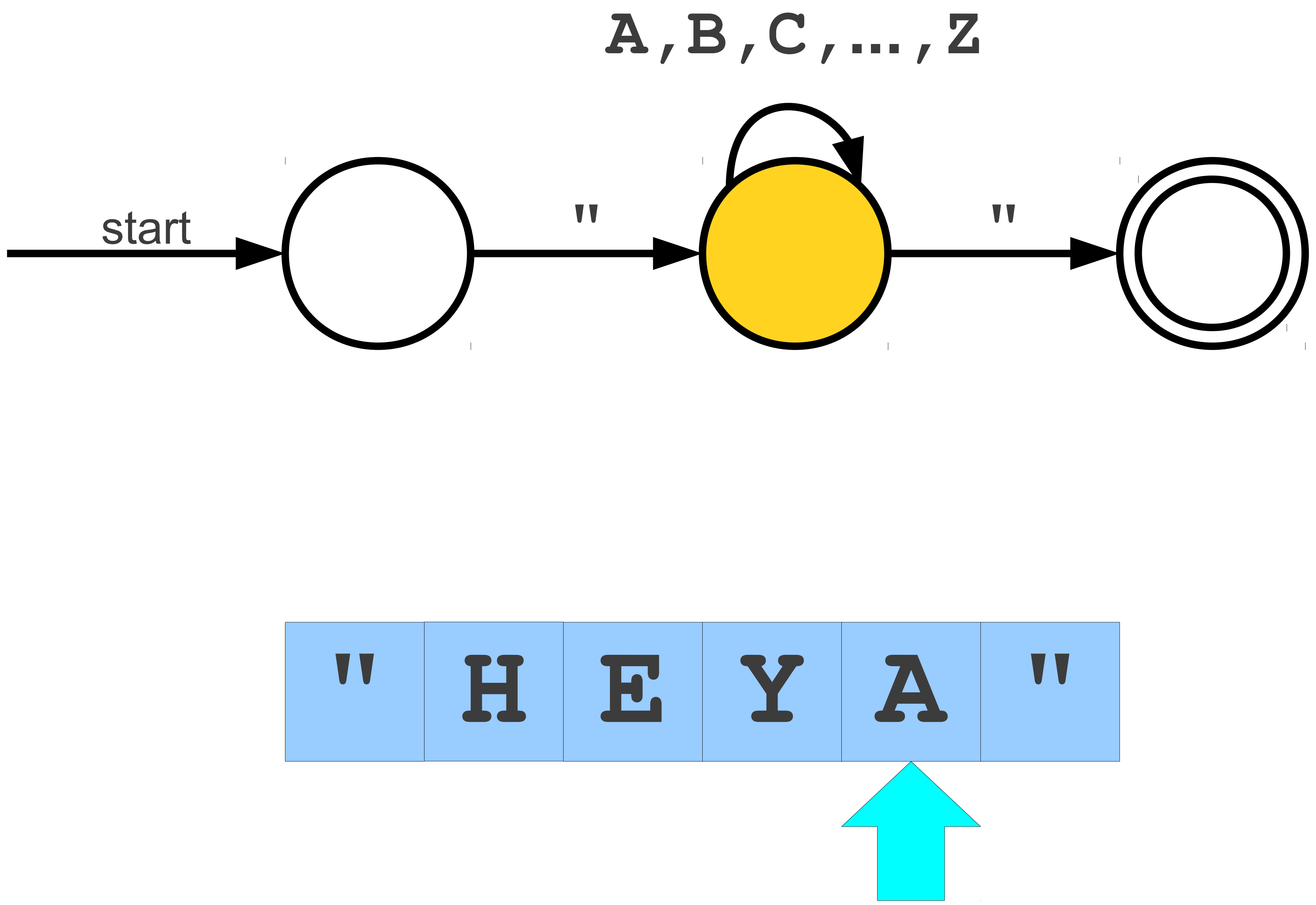
A Simple Automaton



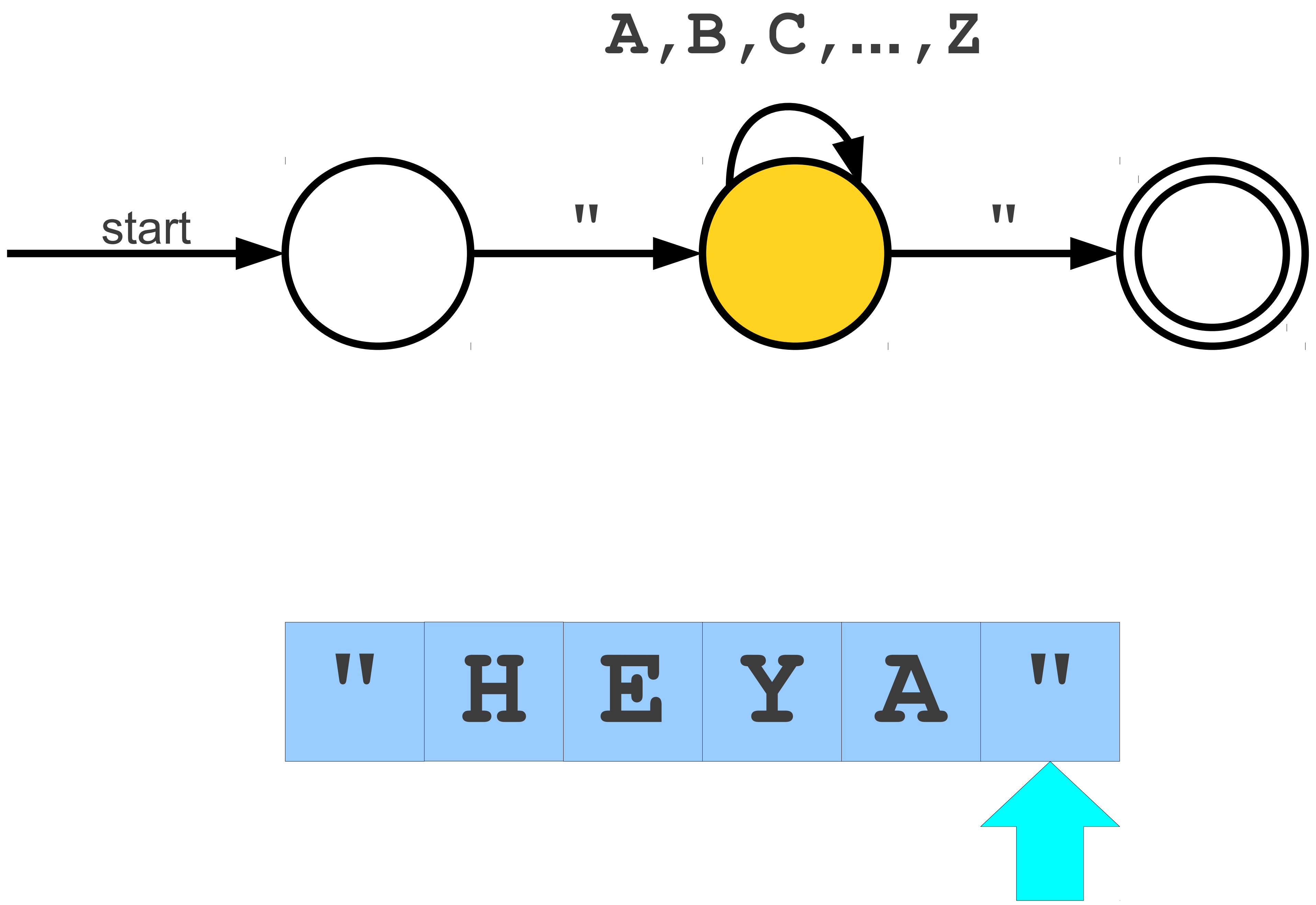
A Simple Automaton



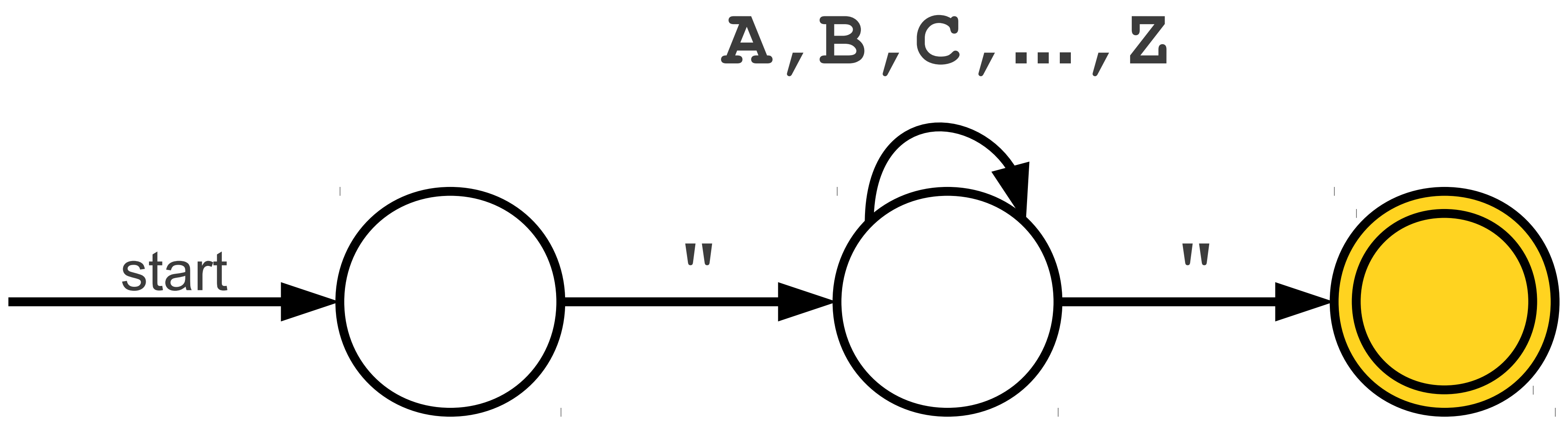
A Simple Automaton



A Simple Automaton

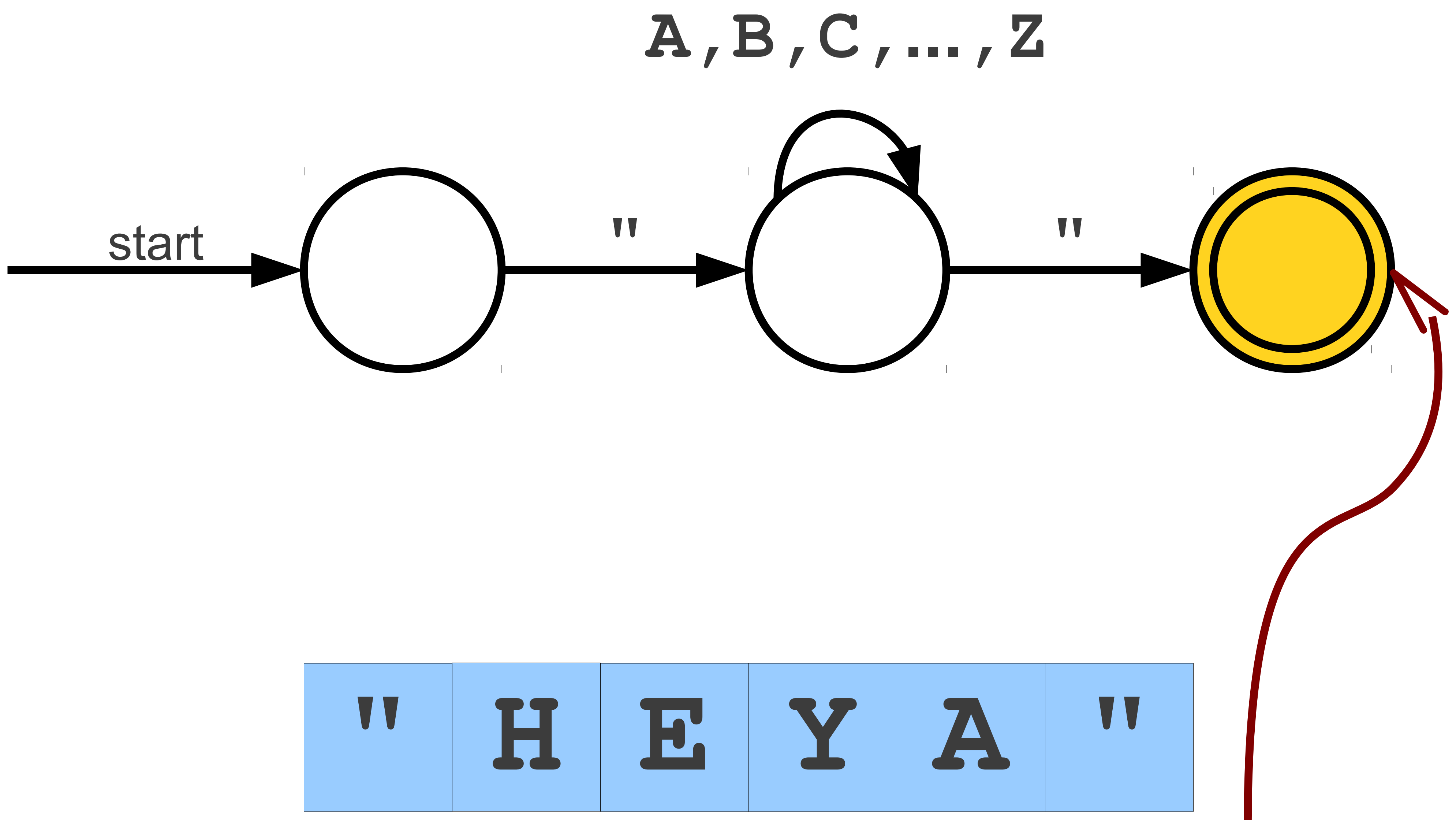


A Simple Automaton



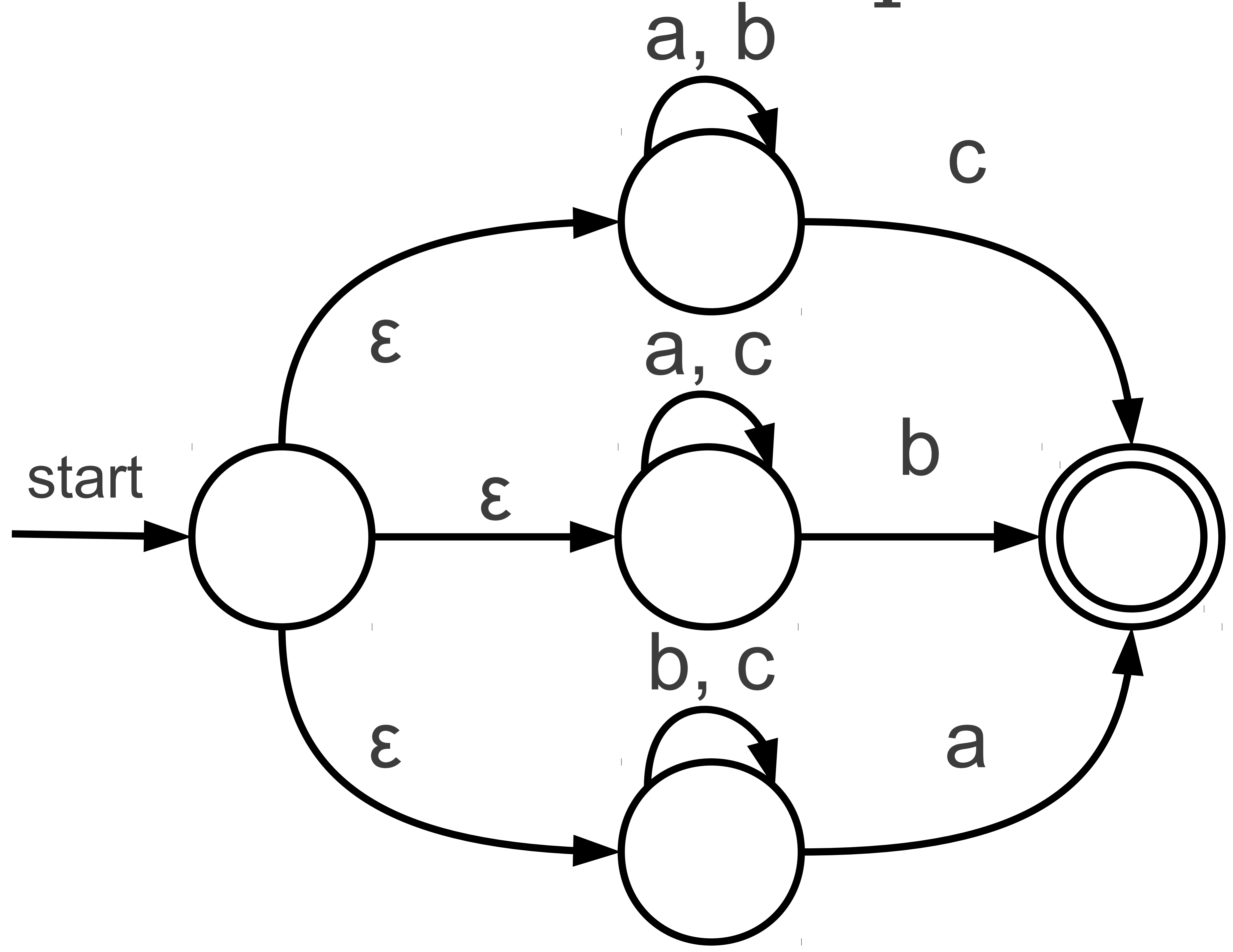
"	H	E	Y	A	"

A Simple Automaton

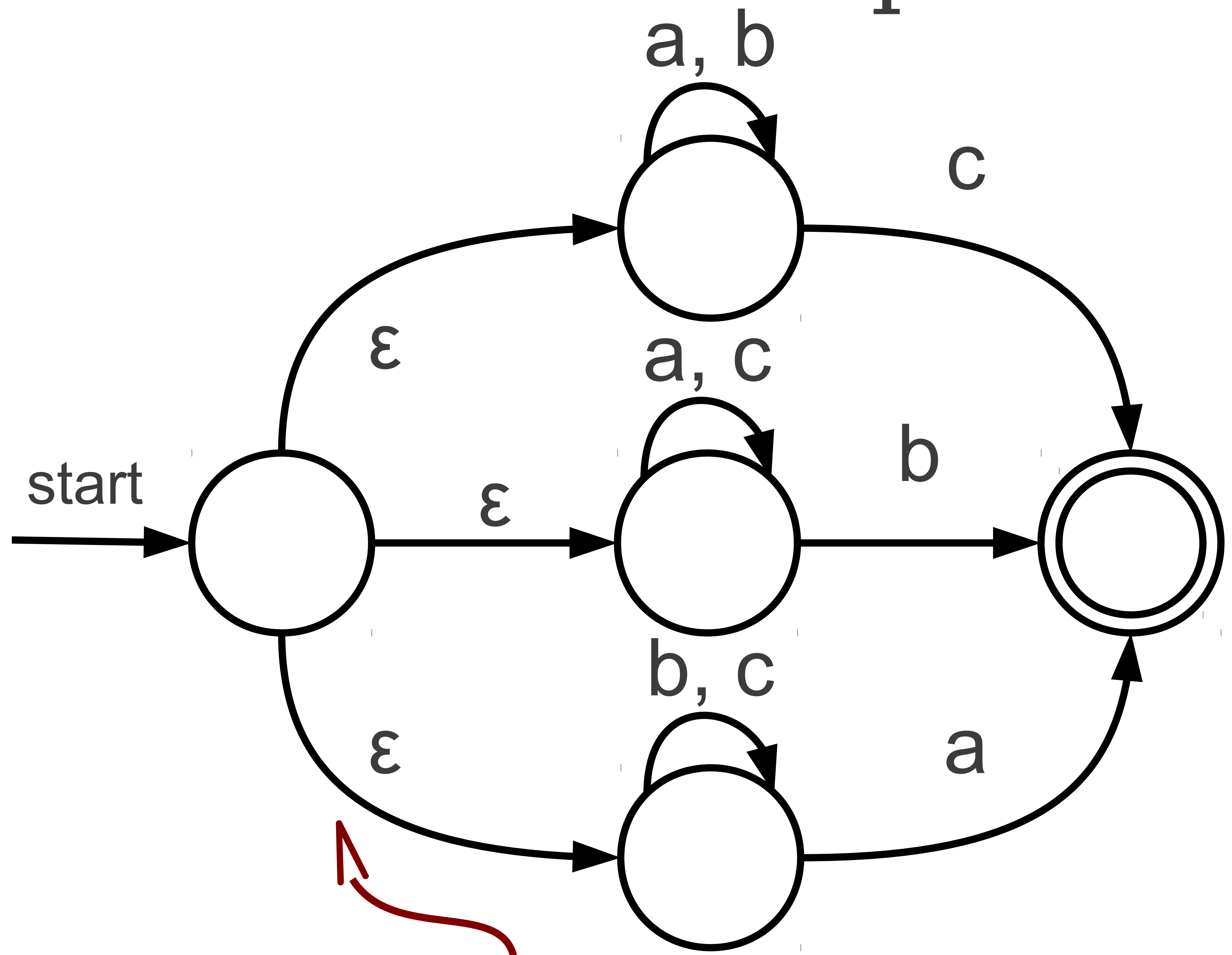


The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.

An Even More Complex Automaton

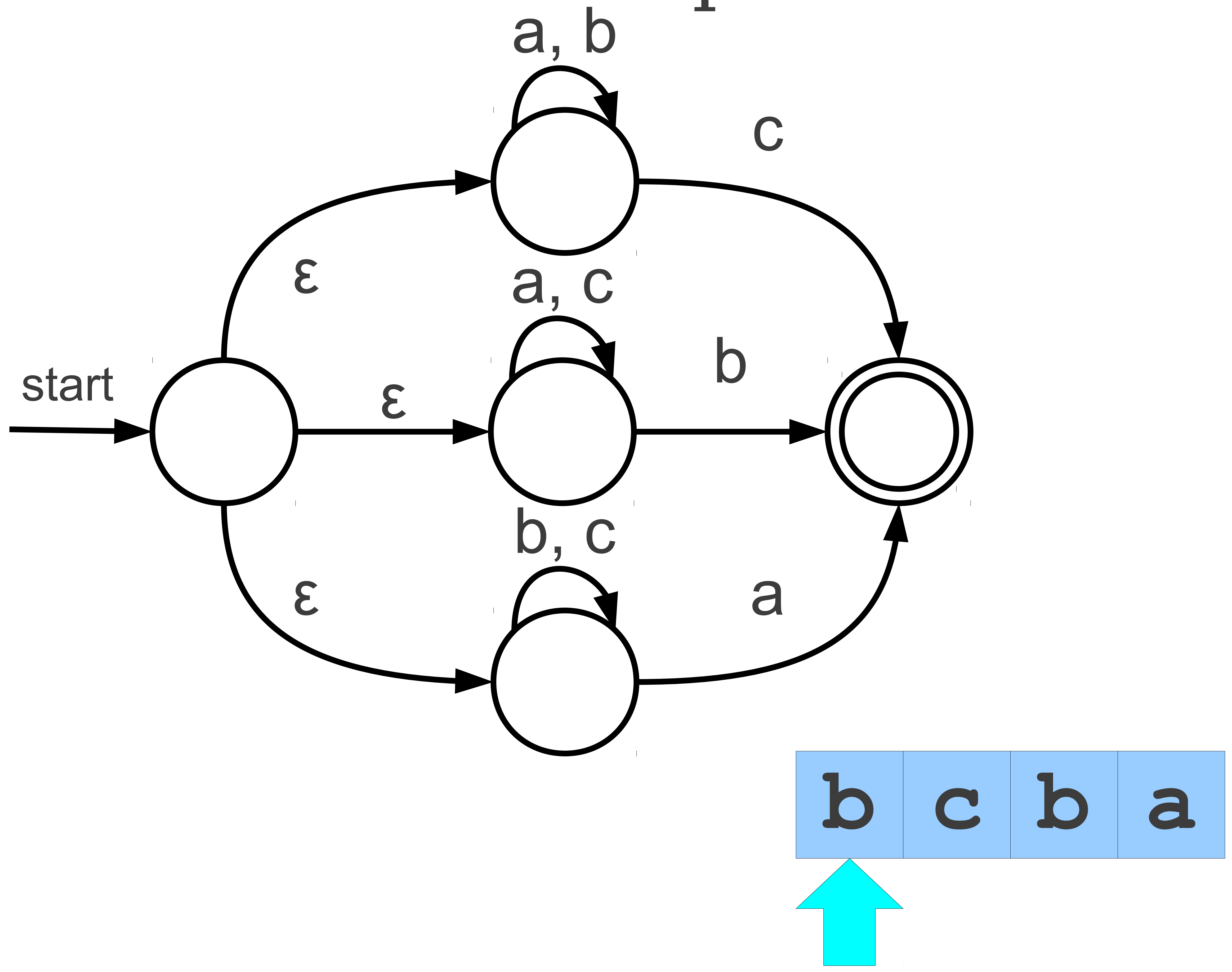


An Even More Complex Automaton

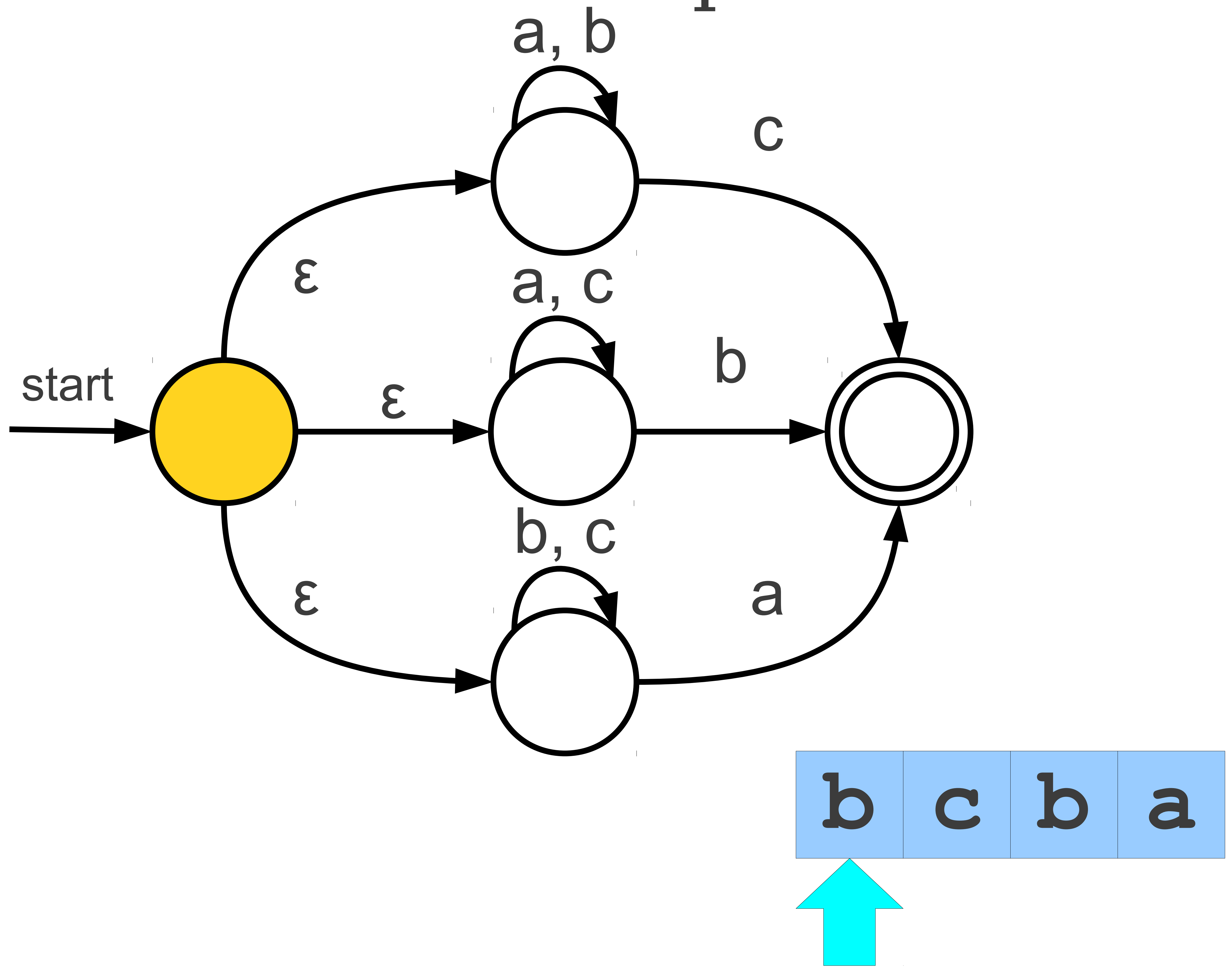


These are called **ε-transitions**. These transitions are followed automatically and without consuming any input.

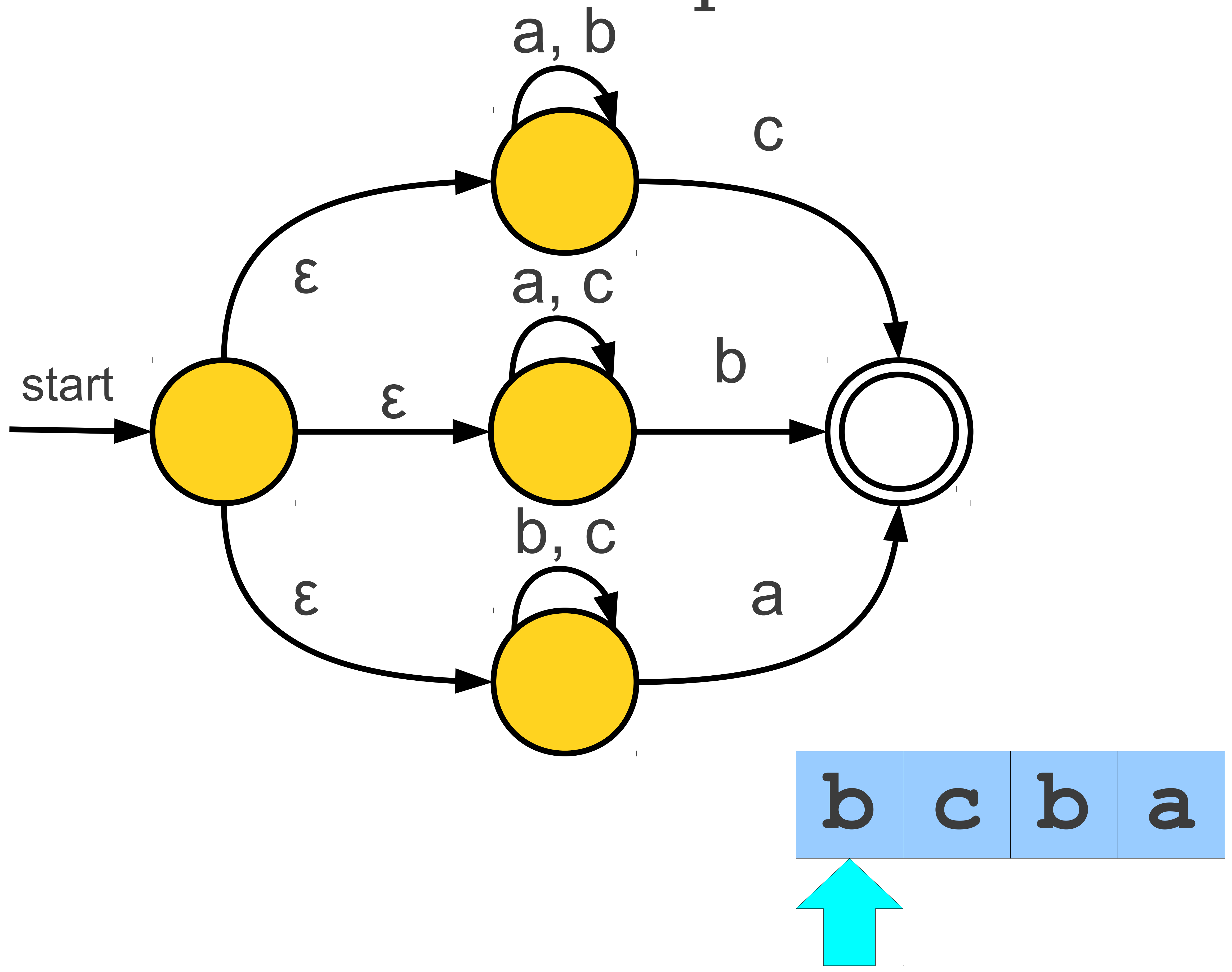
An Even More Complex Automaton



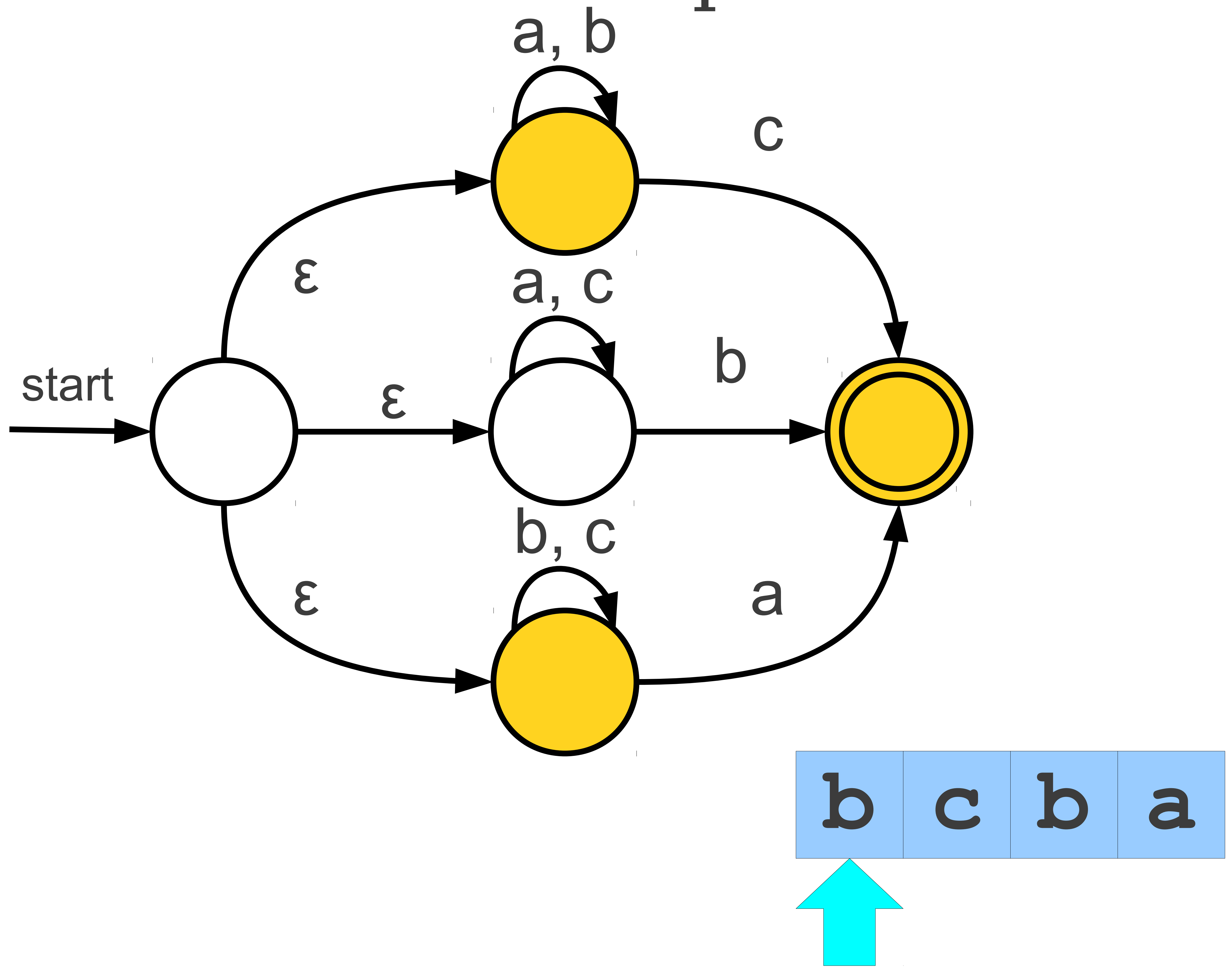
An Even More Complex Automaton



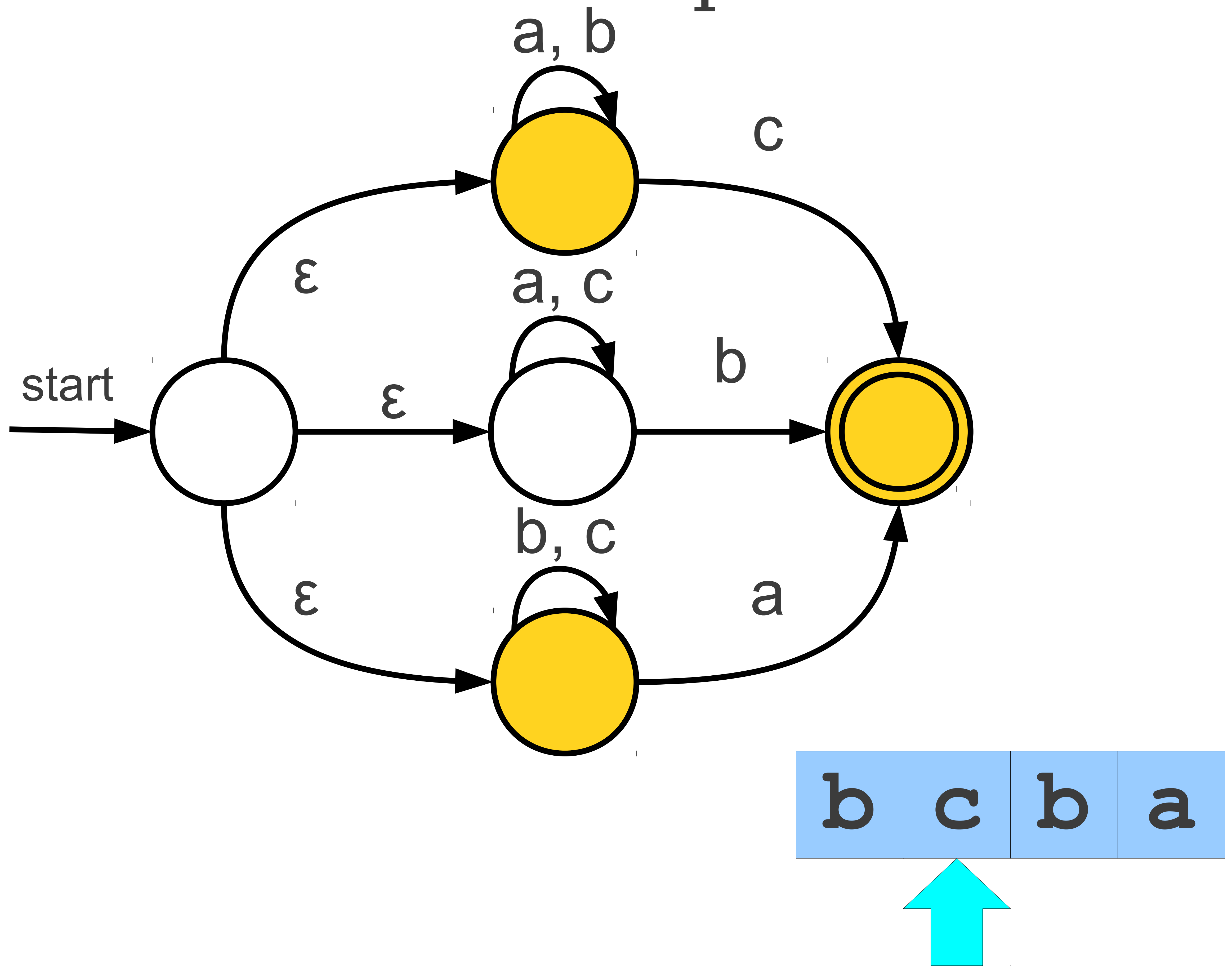
An Even More Complex Automaton



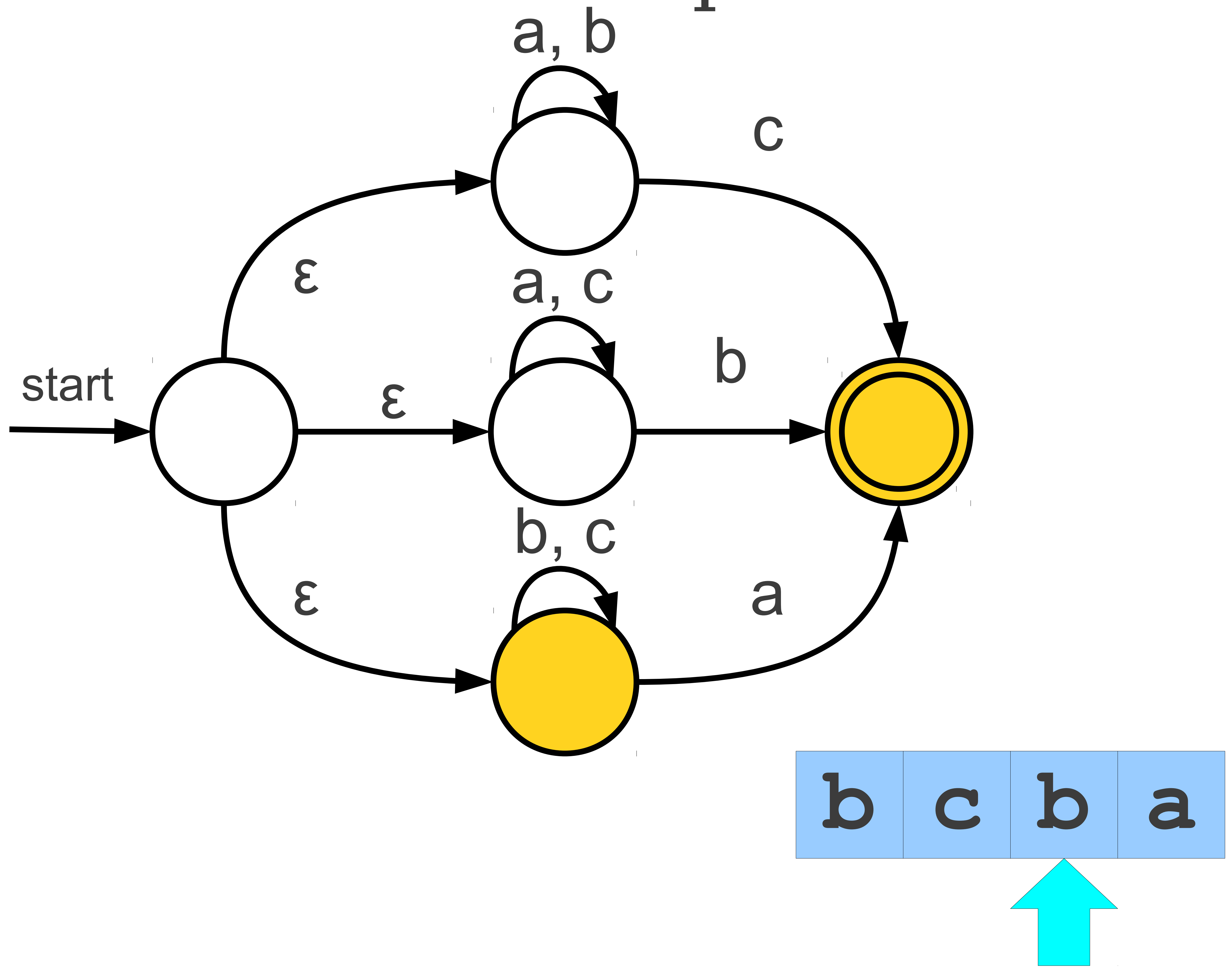
An Even More Complex Automaton



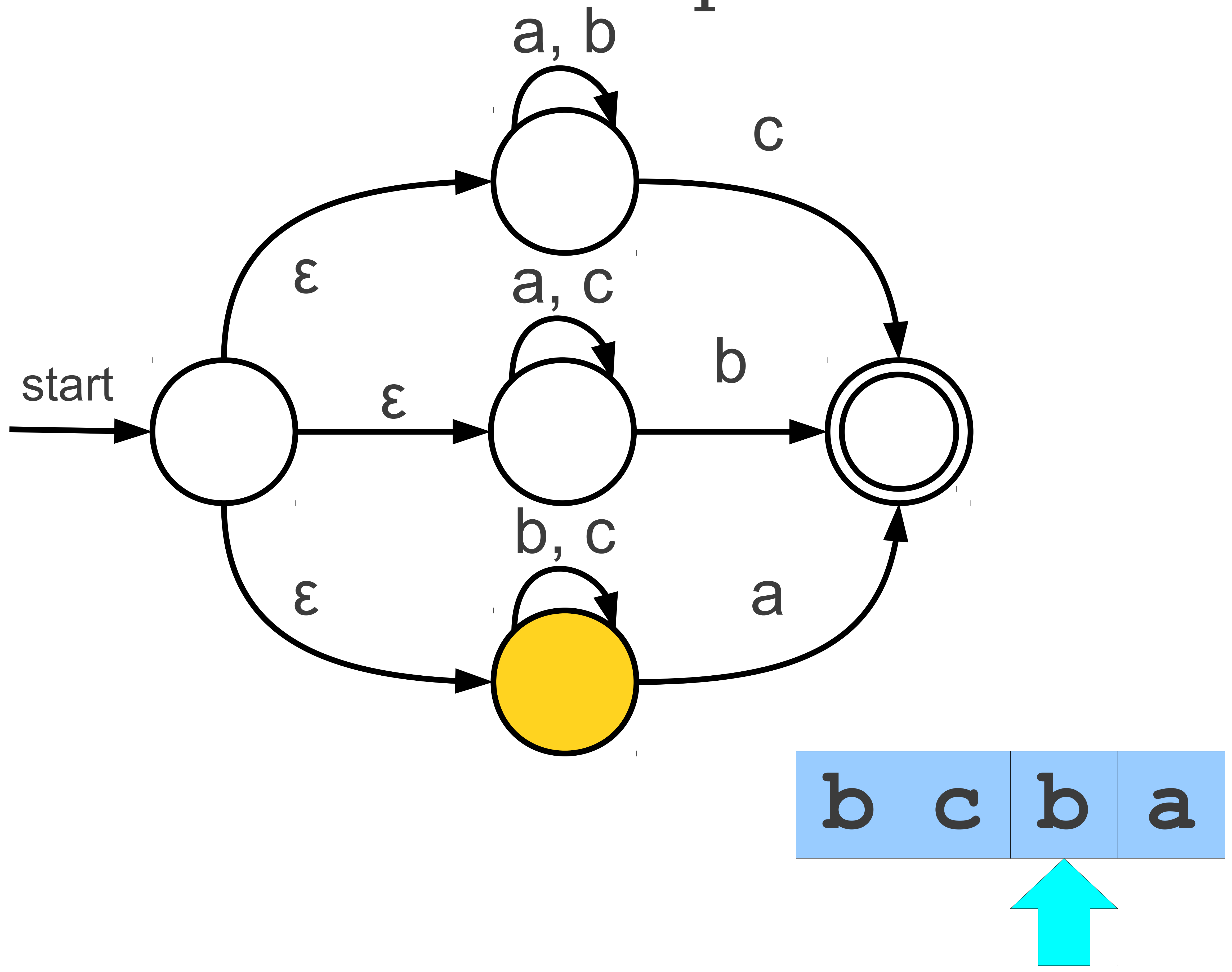
An Even More Complex Automaton



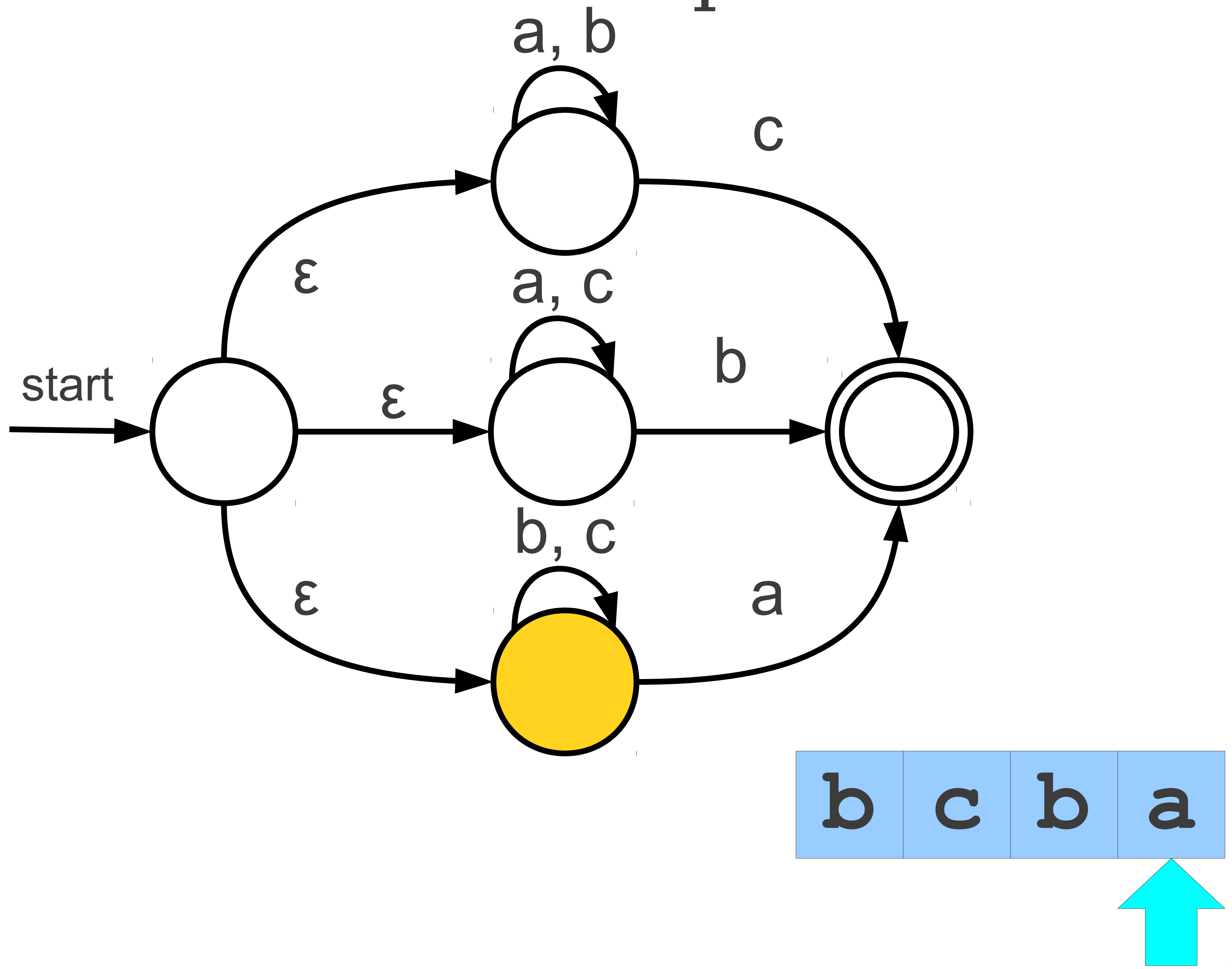
An Even More Complex Automaton



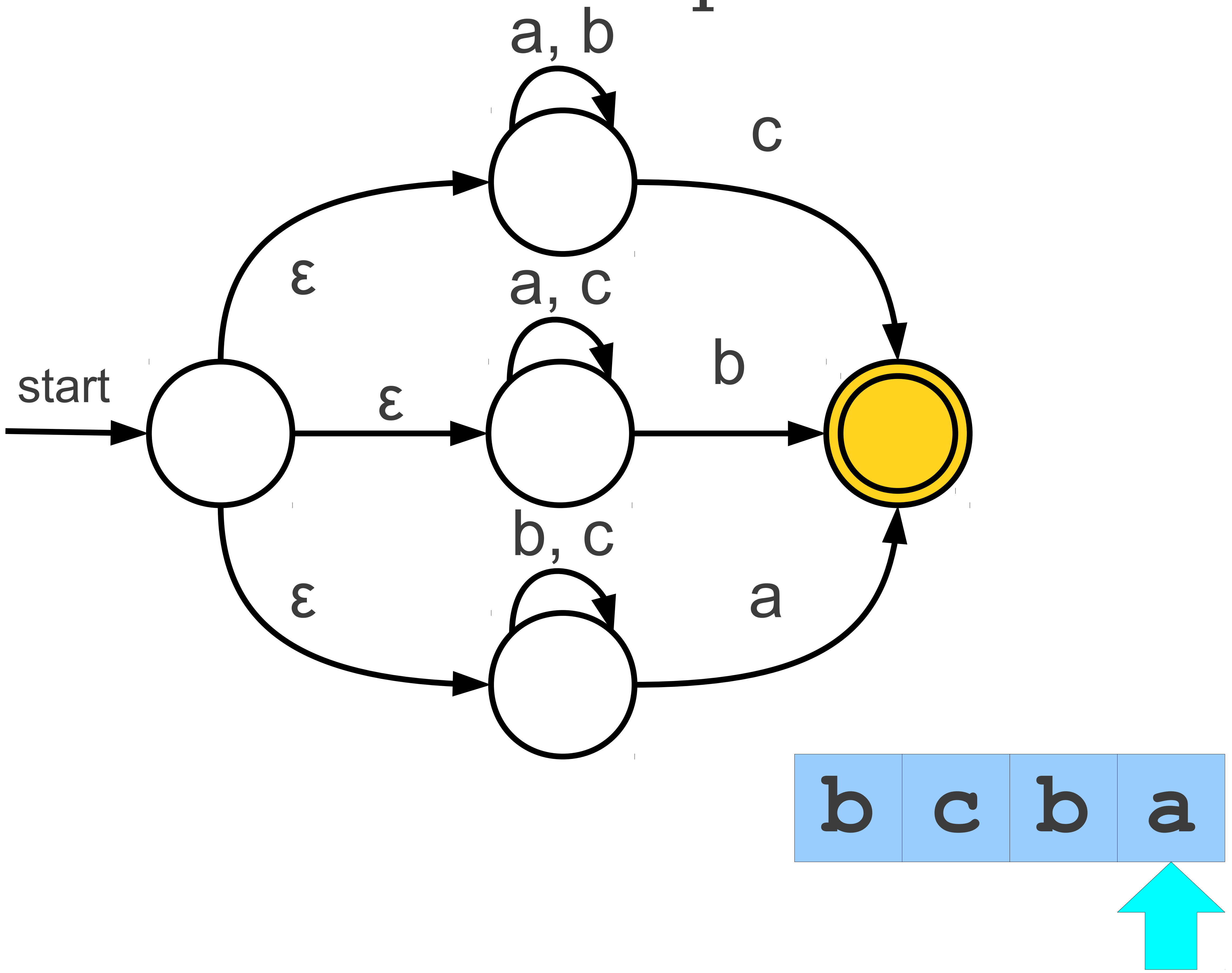
An Even More Complex Automaton



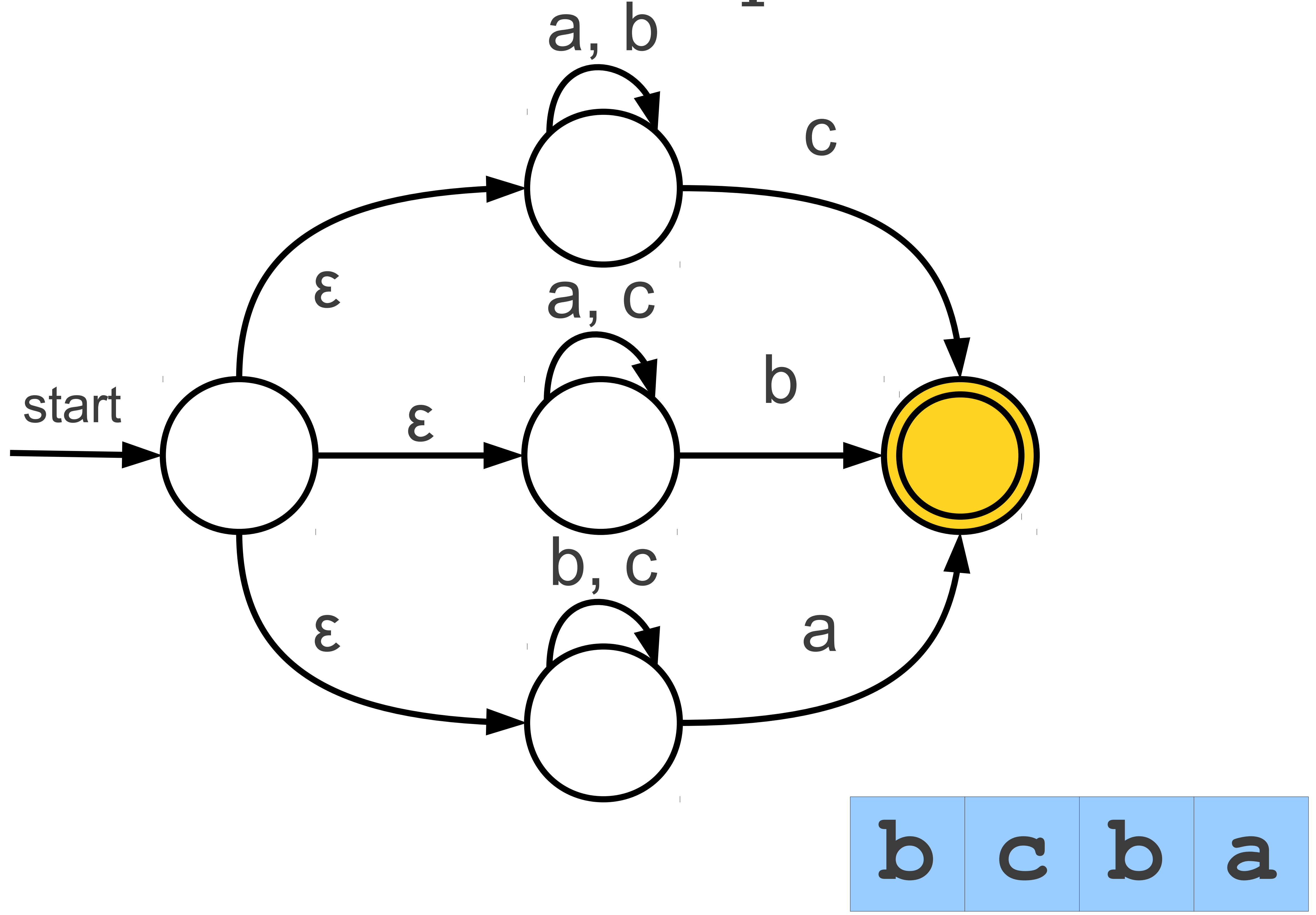
An Even More Complex Automaton



An Even More Complex Automaton



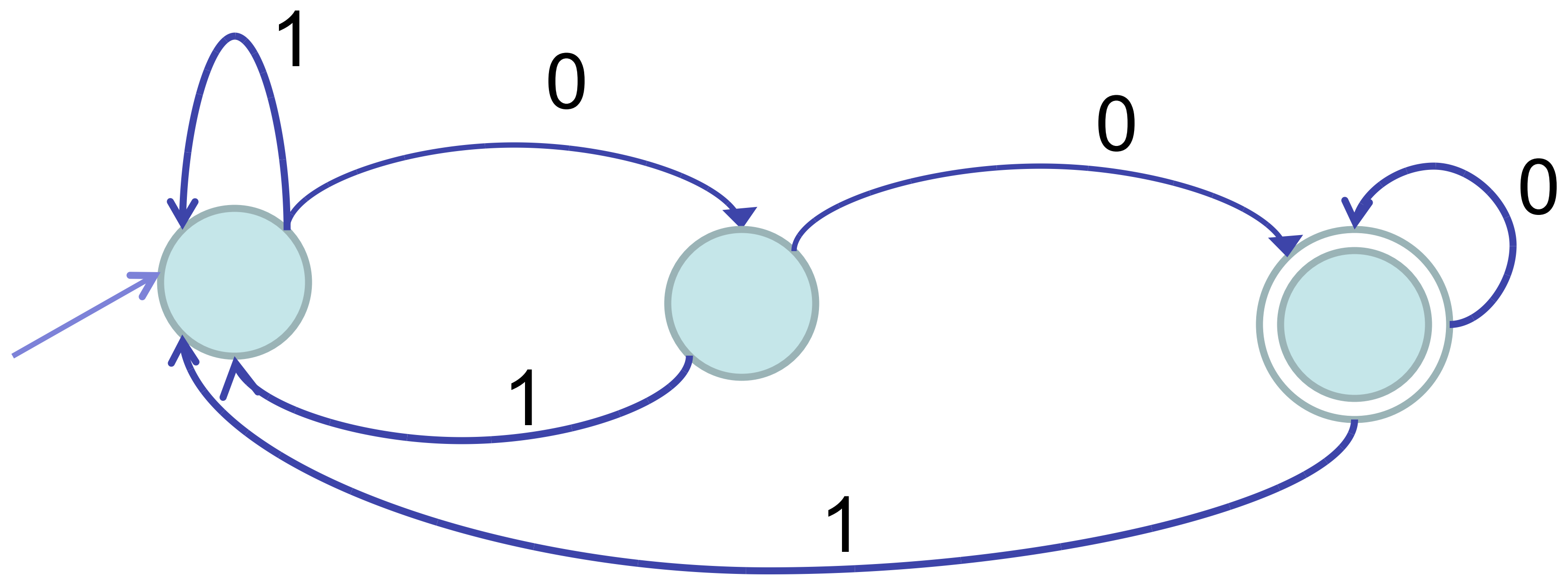
An Even More Complex Automaton



Finite State Automata

What language does this FA recognize?

$\Sigma = \{0,1\}$



Non-determinism

NFAs use "angelic" non-determinism, meaning we non-deterministically branch and if any of the branches succeeds, we succeed.

Think of it as we have an "angel" or "oracle" who will look into the future and tell us what the best choice to make is, if there is one.

Unfortunately, not supported by current hardware. Need to **simulate** this instead.

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)*

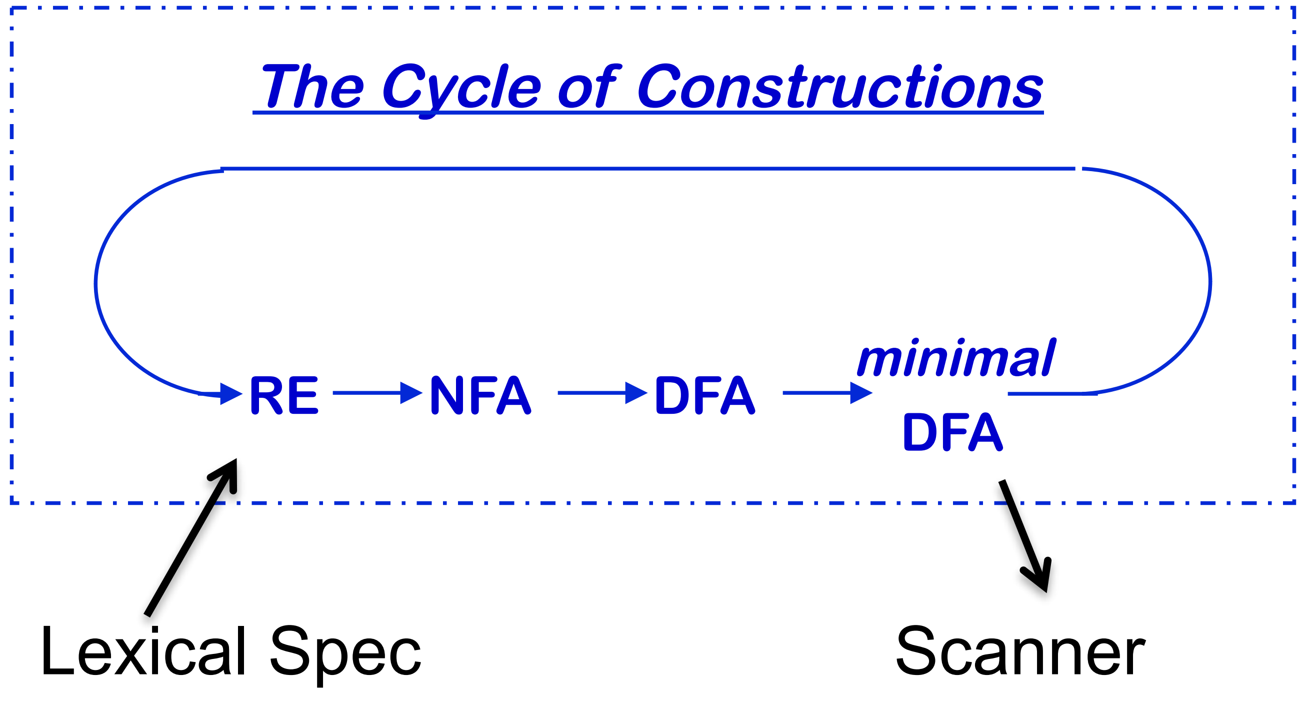
- Build the simulation

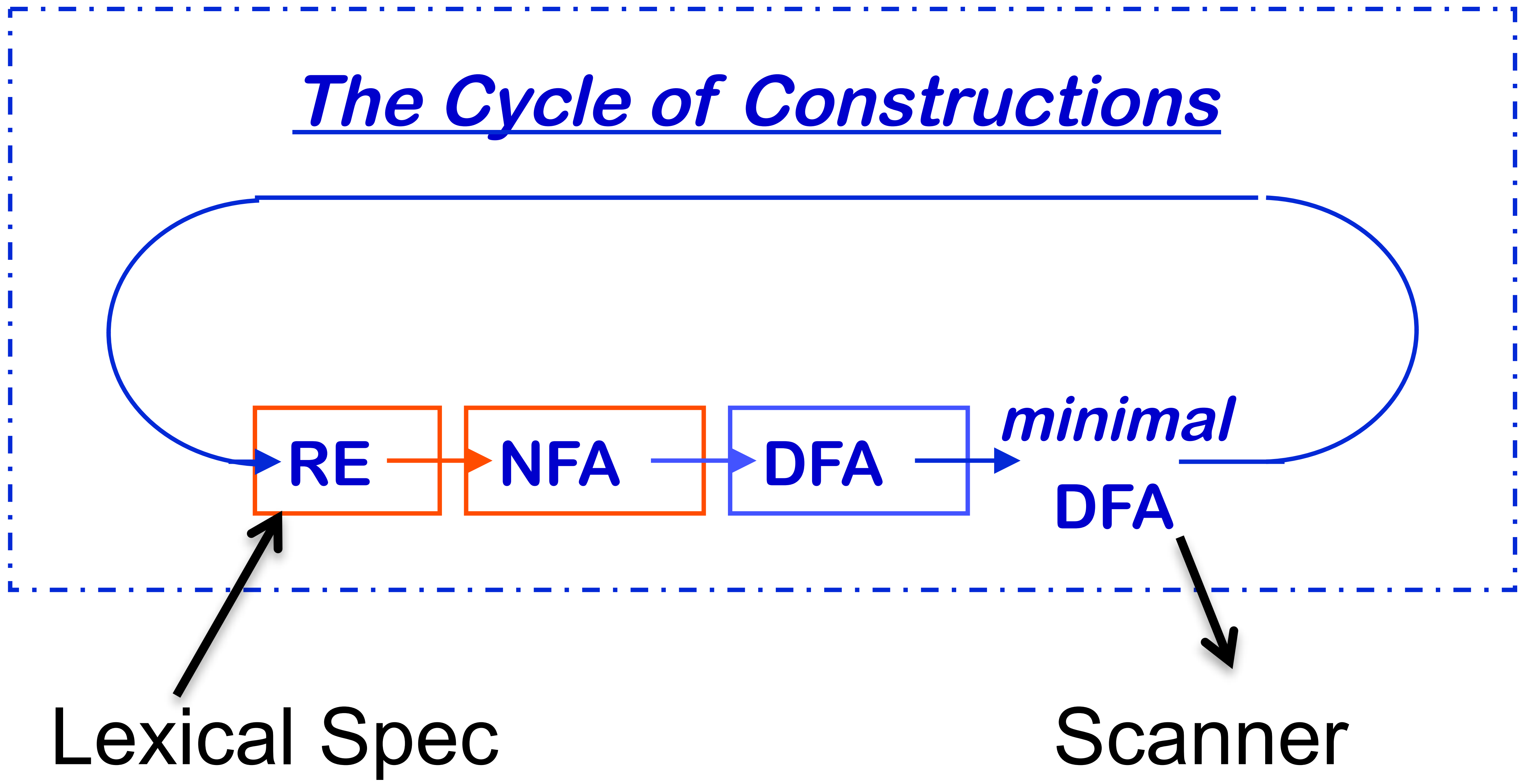
DFA → Minimal DFA

- 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

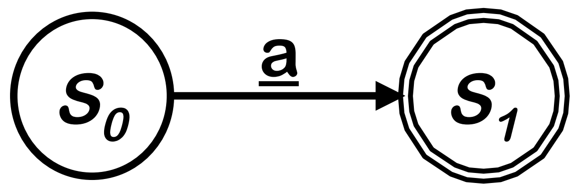



The Cycle of Constructions
RE
NFA
DFA
minimal DFA
Lexical Spec
Scanner

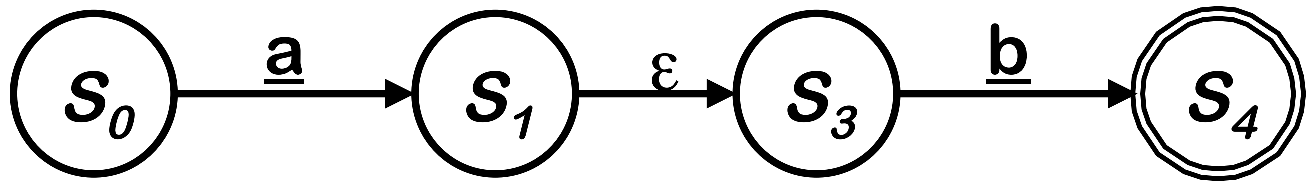
RE →NFA using Thompson's Construction

Key idea

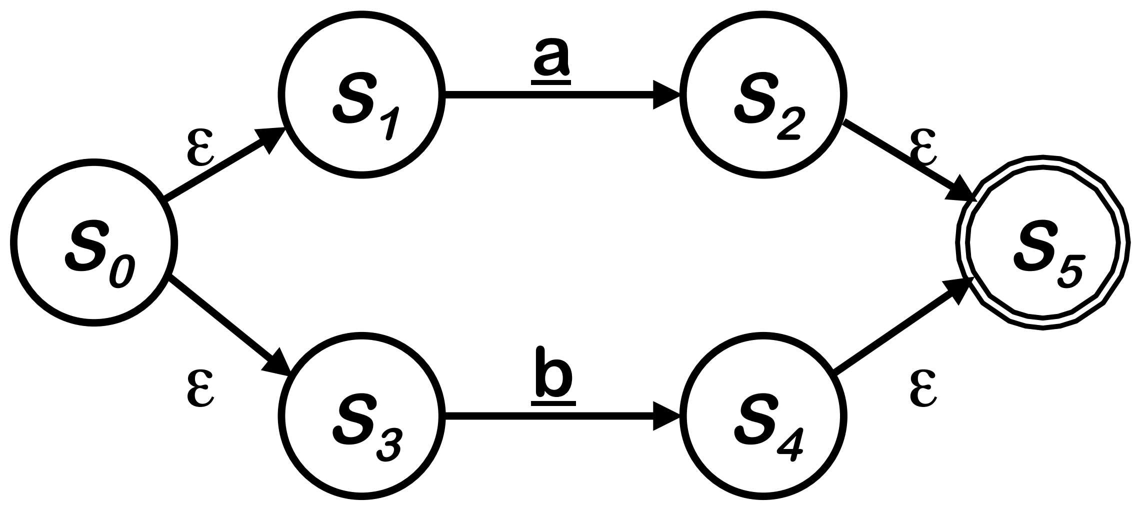
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



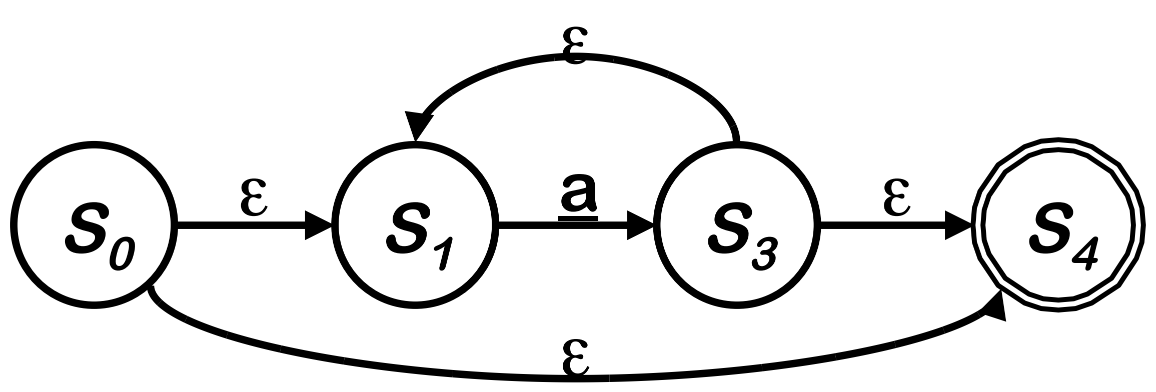
NFA for a



NFA for ab



NFA for a | b



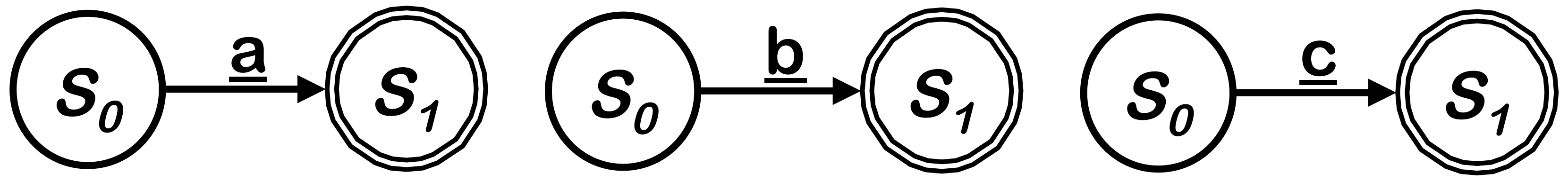
NFA for a*

Ken Thompson, CACM, 1968

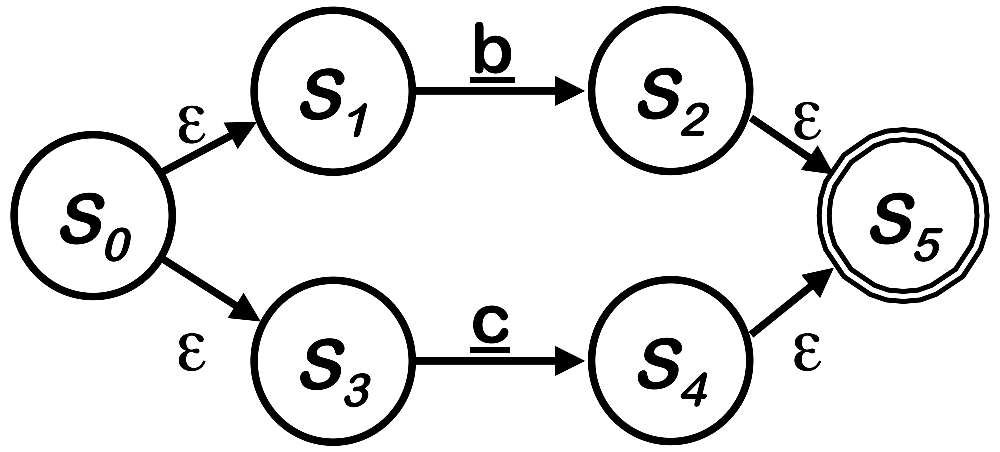
Example of Thompson's Construction

Let's try a (b | c)*

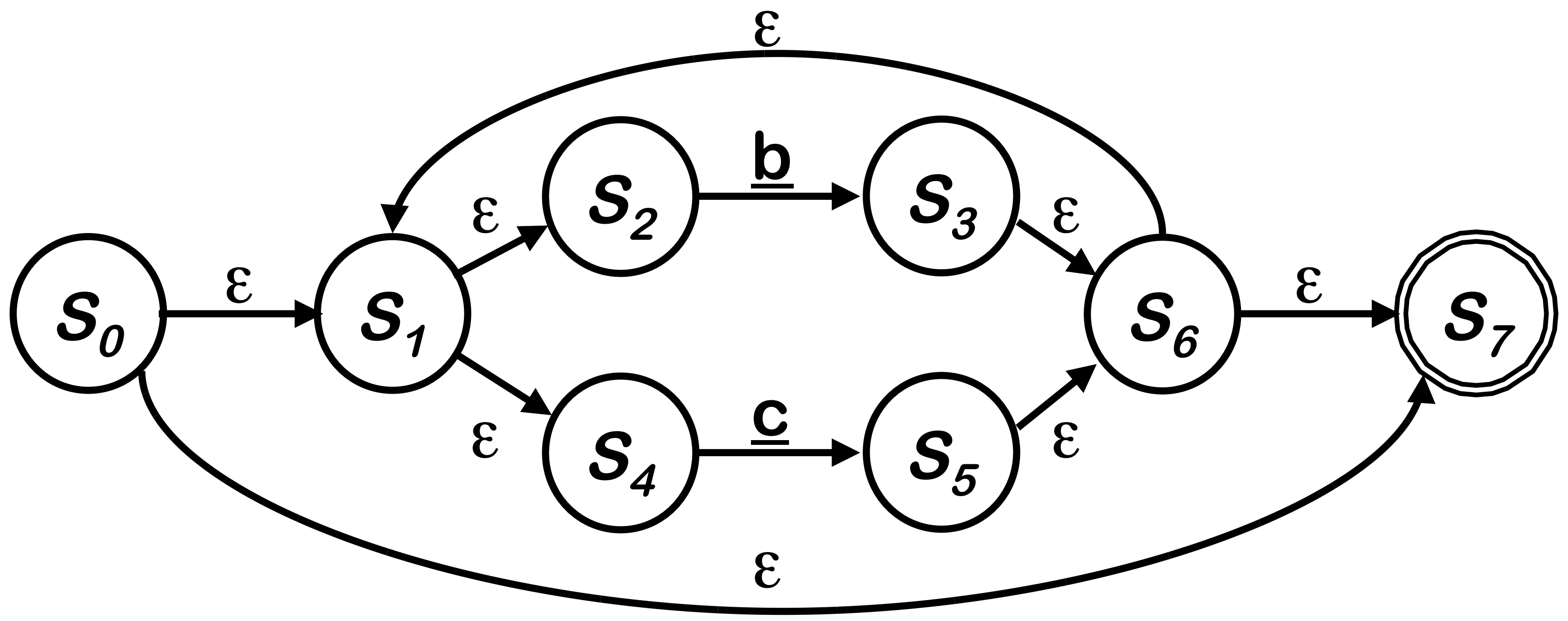
1. a, b, & c



2. b | c

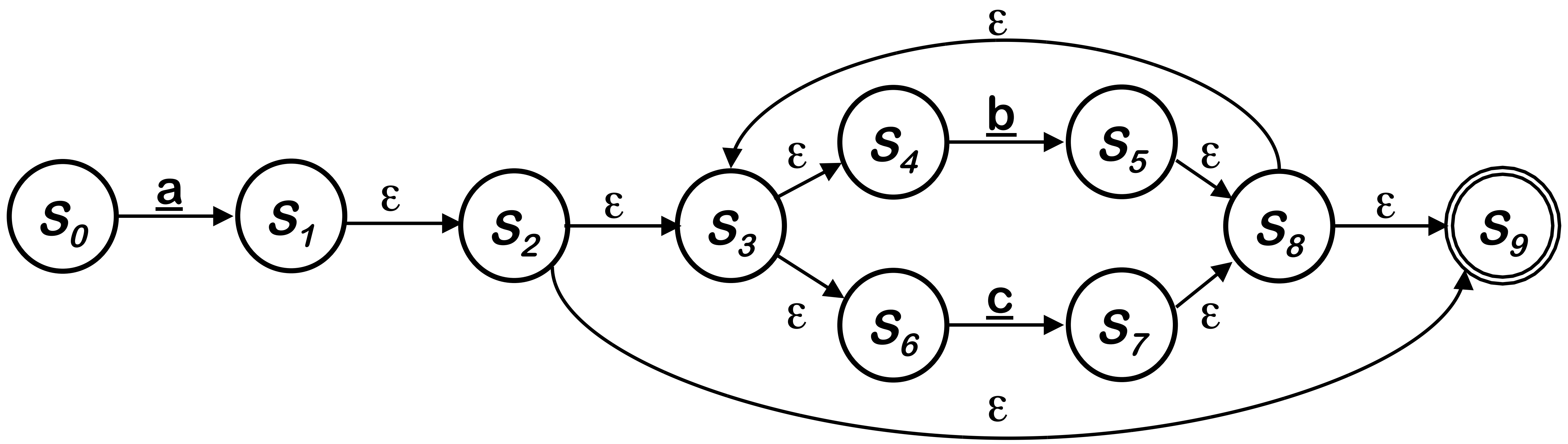


3. (b | c)*

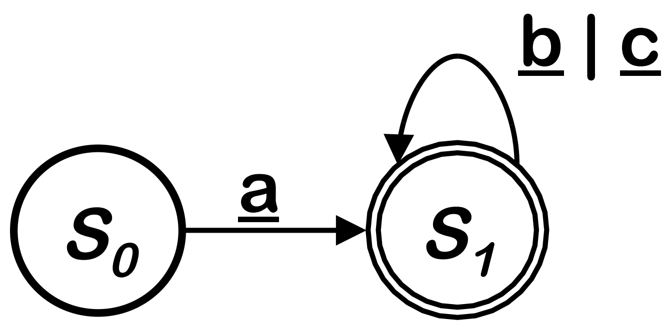


Example of Thompson's Construction *(con't)*

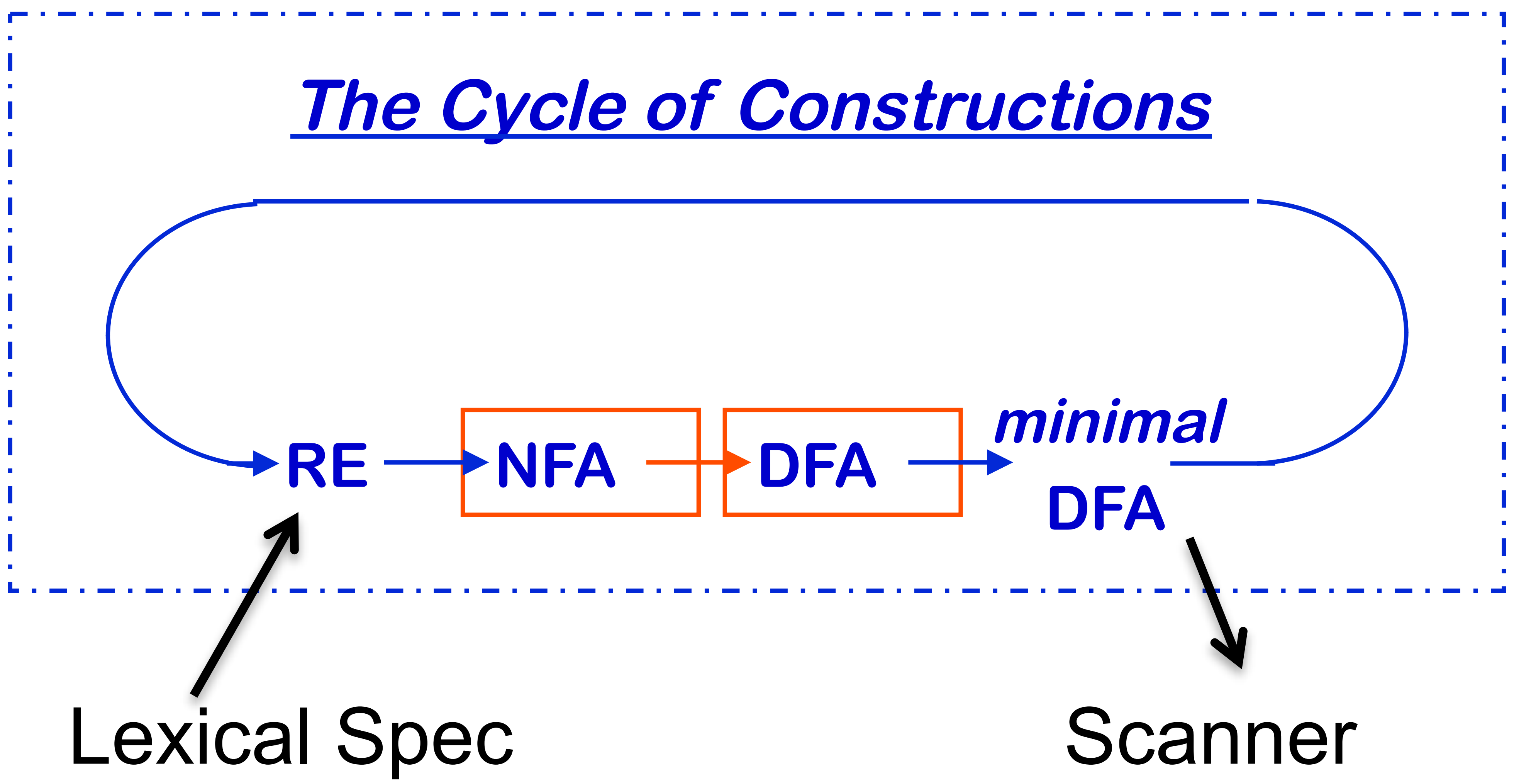
4. <u>a</u> (<u>b</u> | <u>c</u>)*



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

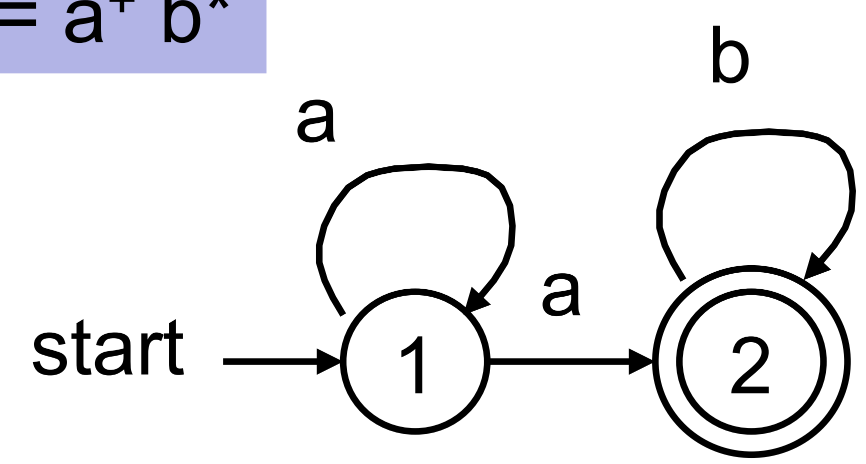
  = a non-empty subset of states of the NFA
- Start state

  = 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 (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= $a^+ b^*$

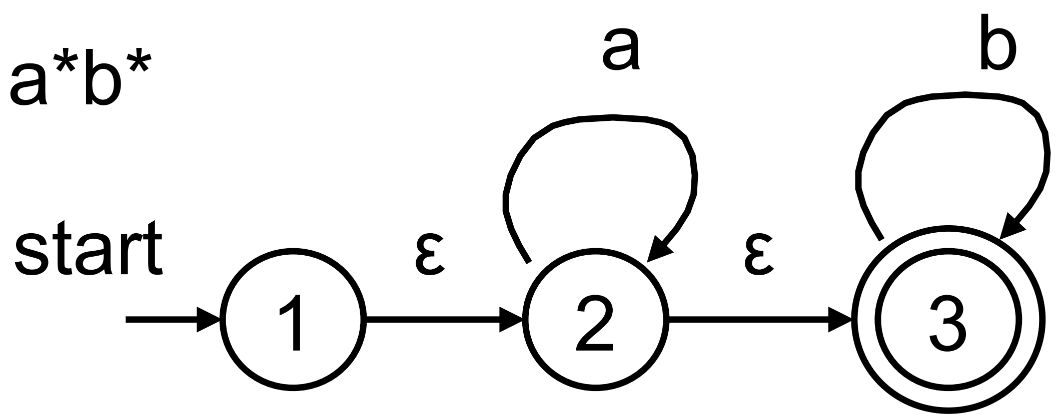


ε-closure(1) = {1} include state "1"
(1,a) → {1,2} include state "1/2"
(1,b) → ERROR

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

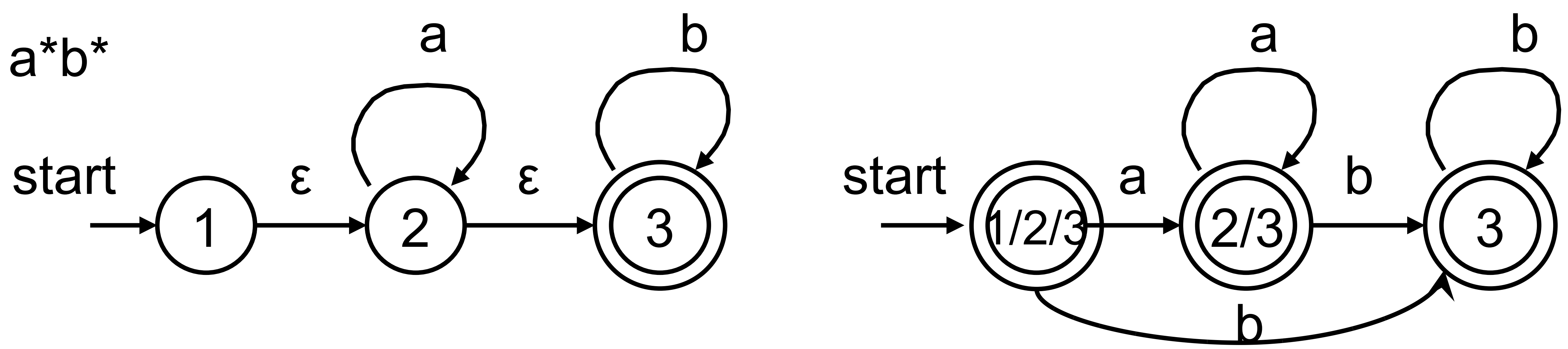
a*b*



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}

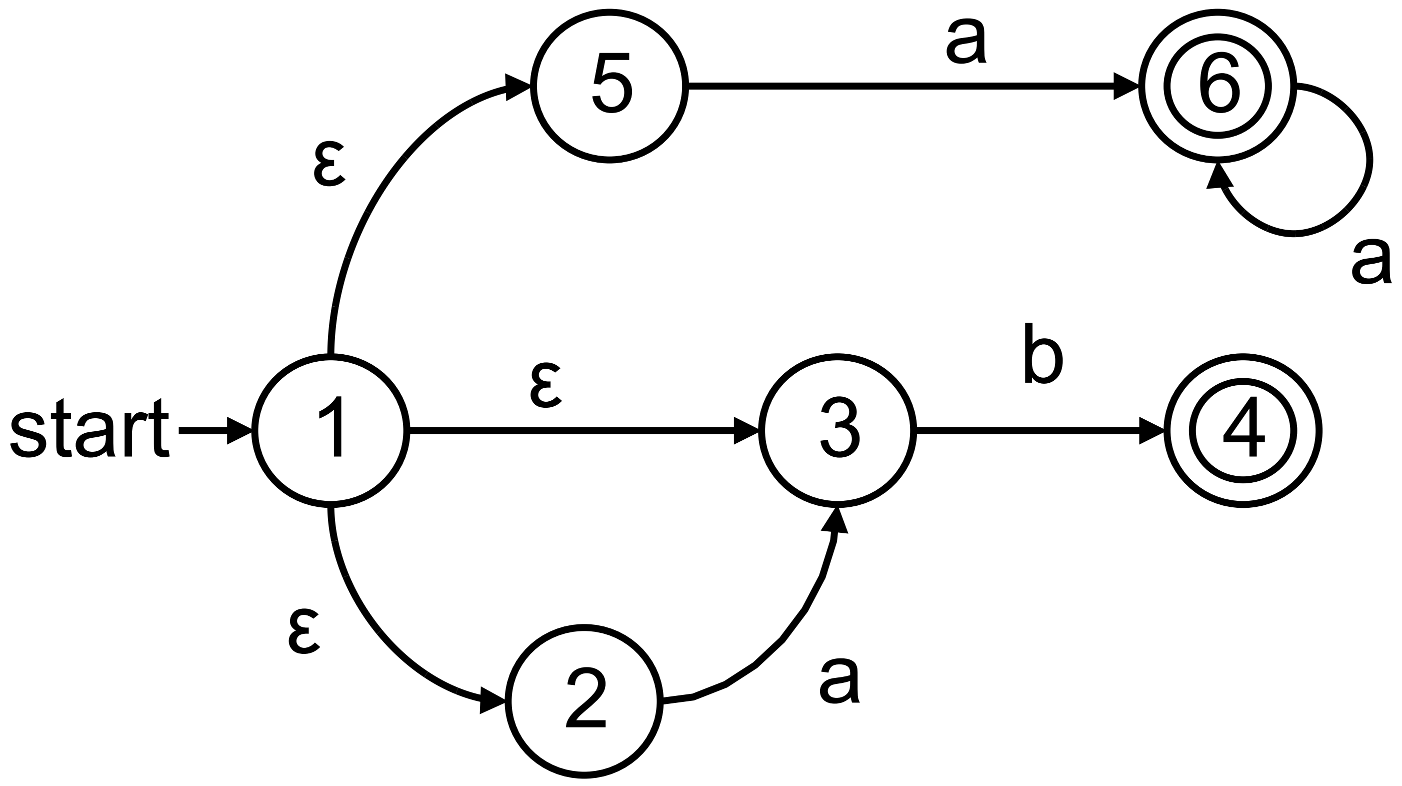
NFA to DFA (3)

- 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

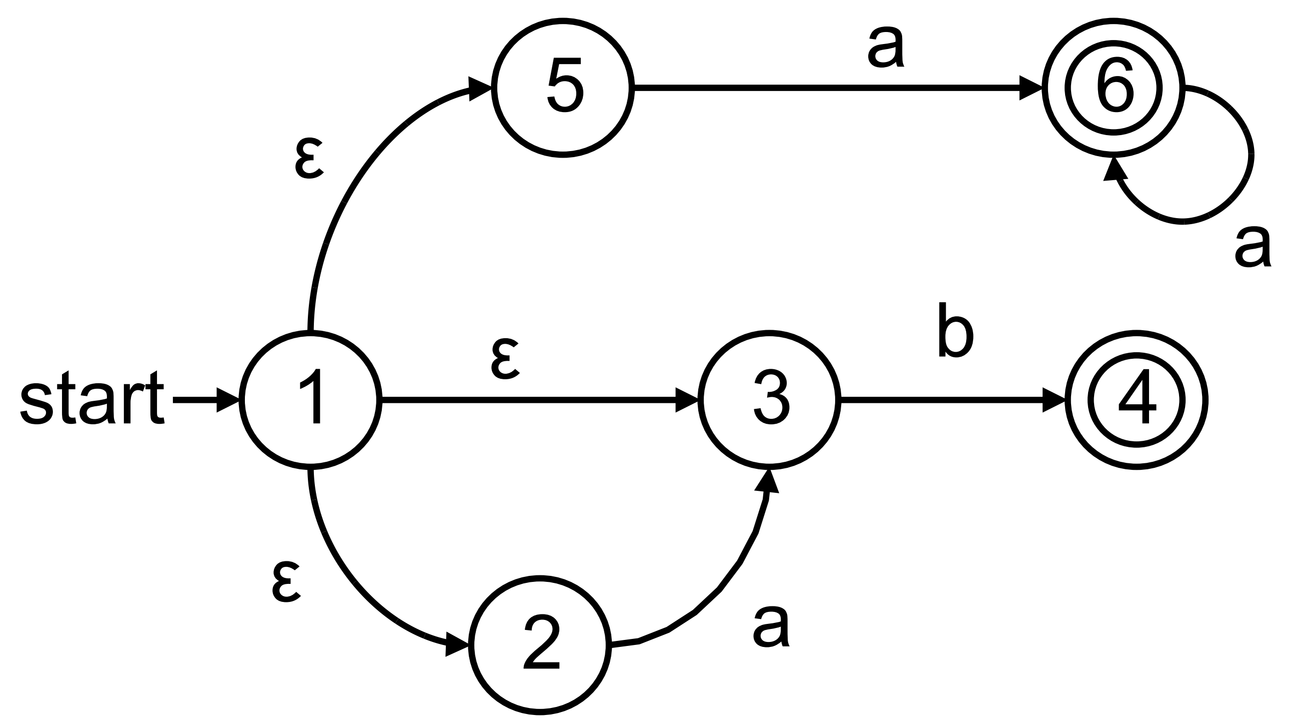


1. ε-closure(1) = {1,2,3}; include1/2/3
2. Move(1/2/3, a) = {2, 3} + ε-closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} + ε-closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} + ε-closure(2) = {2,3}
5. Move(2/3, b) = {3} + ε-closure(3) = {3}
6. Move(3, b) = {3} + ε-closure(3) = {3}

NFA to DFA - Example



NFA to DFA - Example



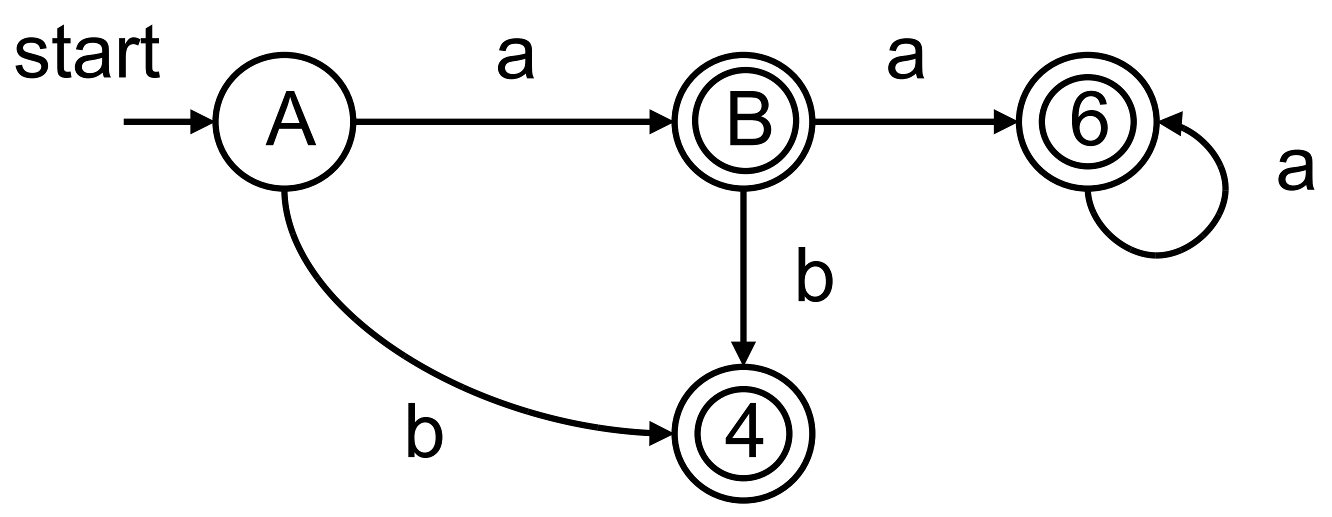
ε-closure(1) = {1, 2, 3, 5}

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}

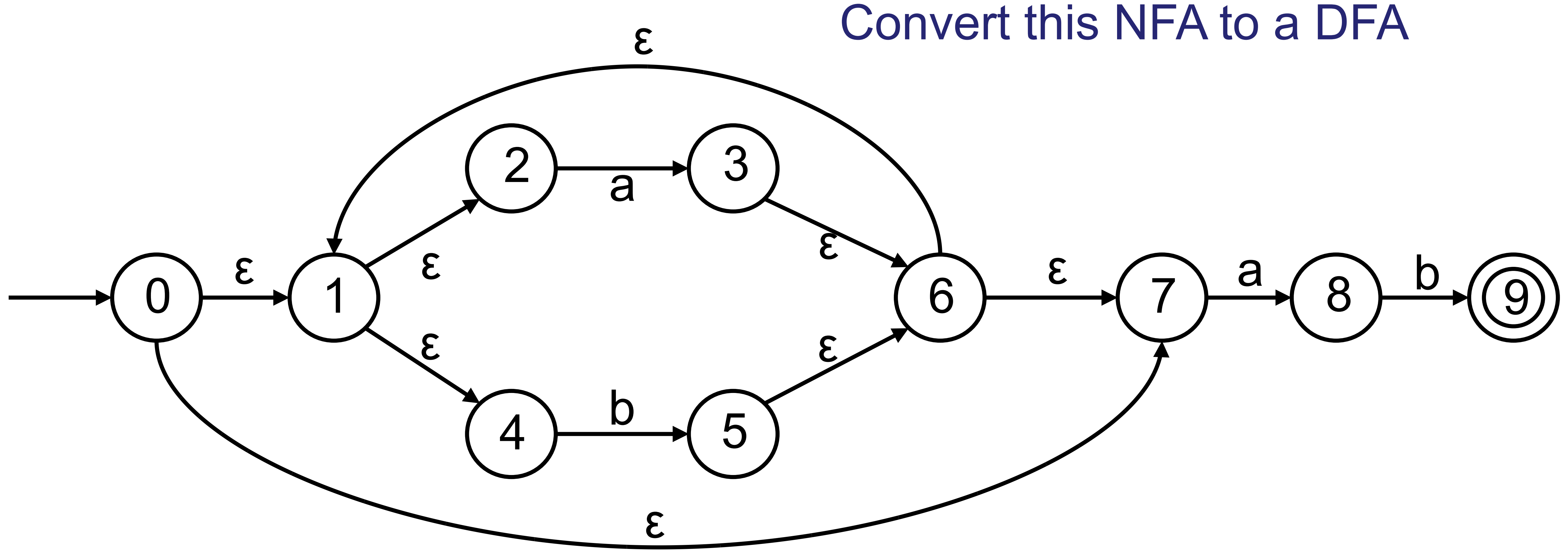
move(6, a) = {6} + ε-closure(6) = {6}

move(6, b) → ERROR

move(4, a|b) → ERROR

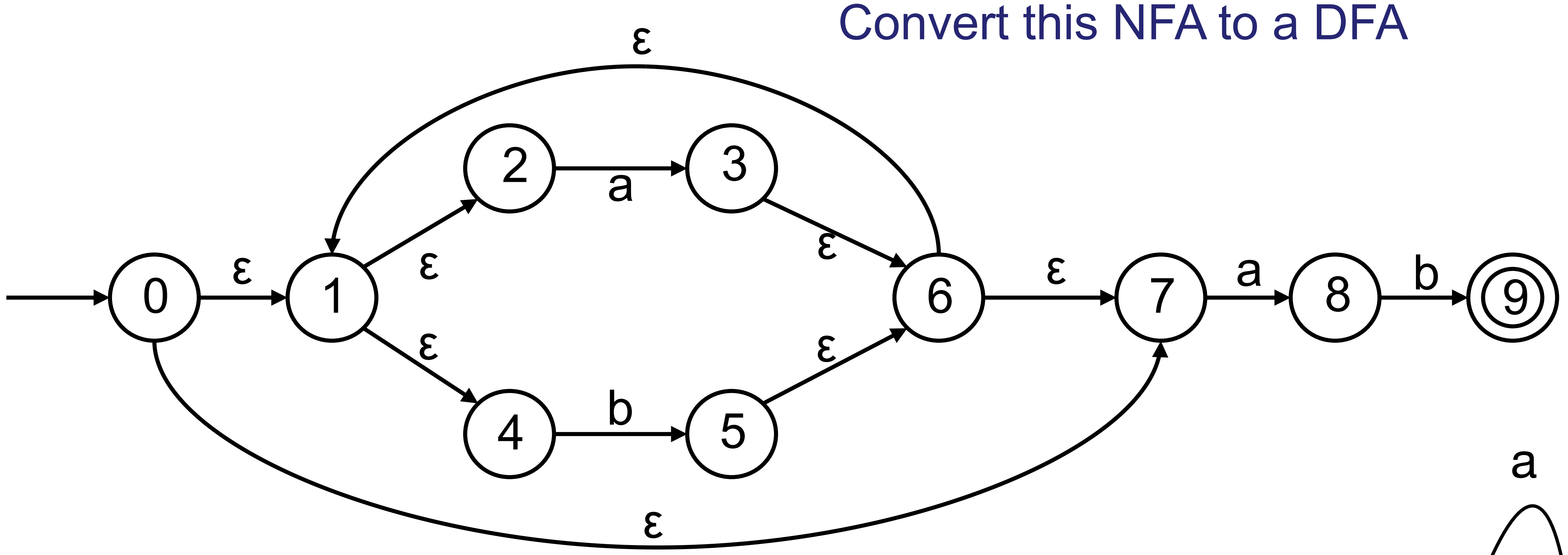
Class Problem

Convert this NFA to a DFA



Class Problem

Convert this NFA to a DFA

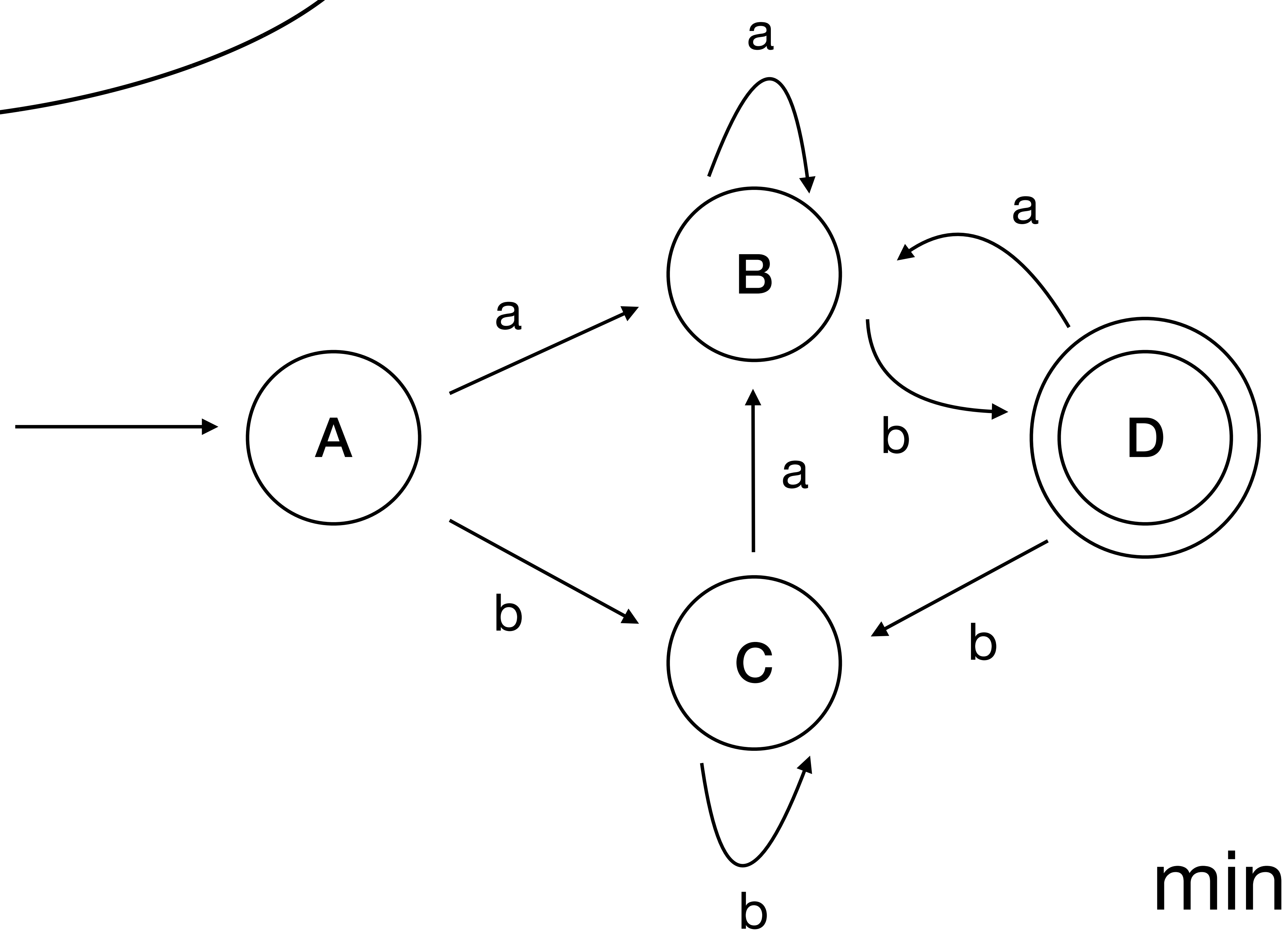


$A = \{ 0, 1, 2, 4, 7 \}$

$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$

$C = \{ 1, 2, 4, 5, 6, 7 \}$

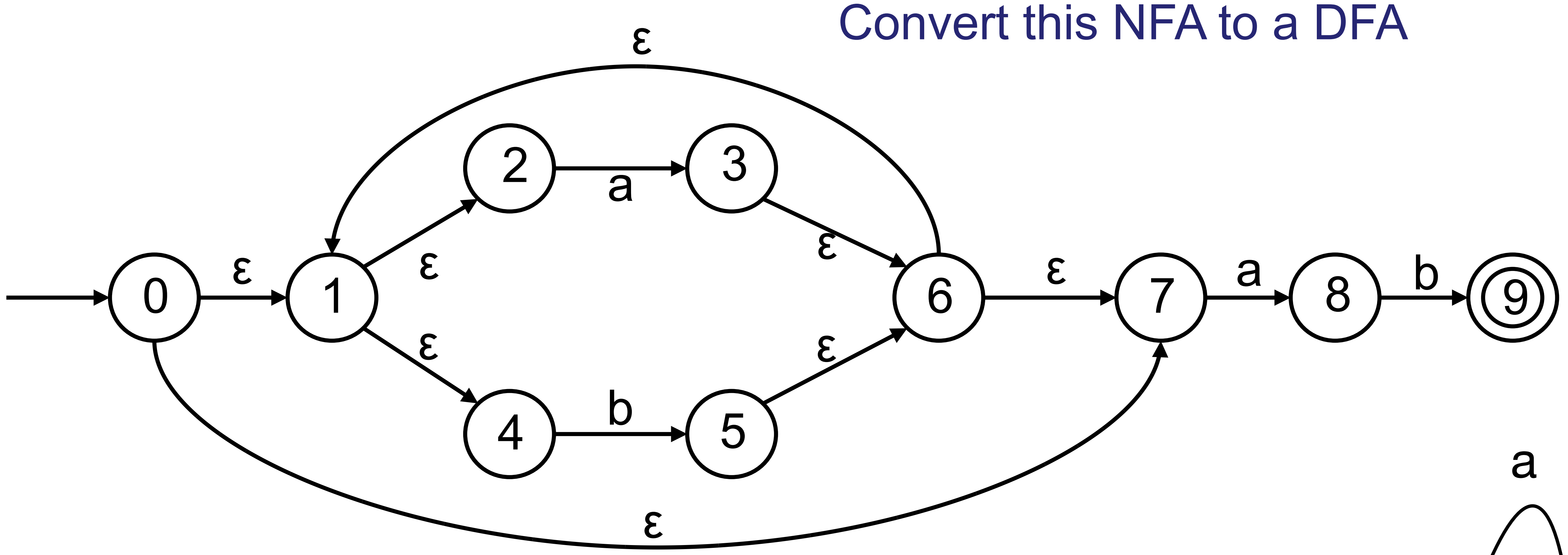
$D = \{ 1, 2, 4, 5, 6, 8, 8, 9 \}$



minimal?

Class Problem

Convert this NFA to a DFA

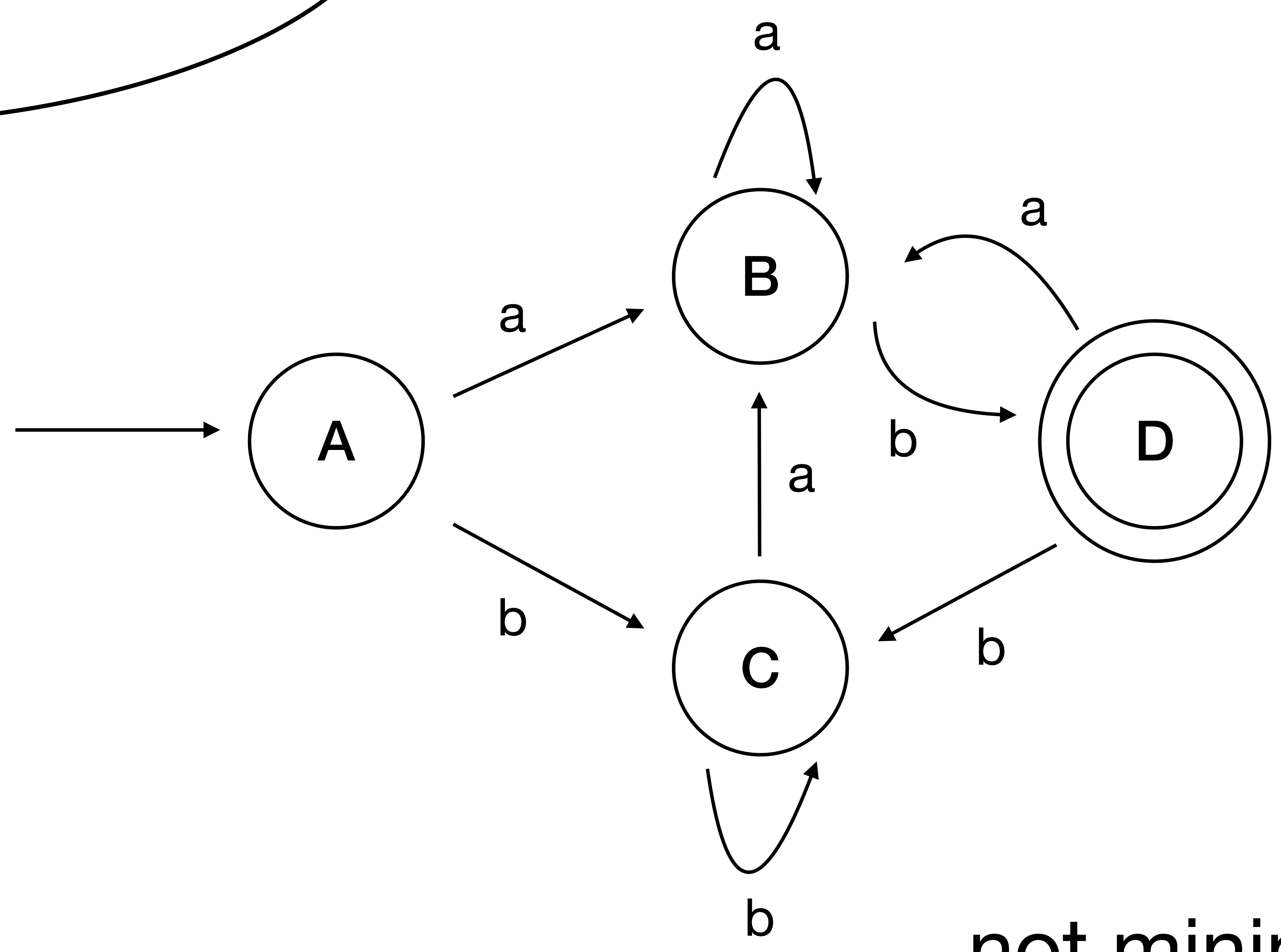


A = { 0, 1, 2, 4, 7 } == (a|b)*ab

B = { 1, 2, 3, 4, 6, 7, 8 } == b|A

C = { 1, 2, 4, 5, 6, 7 } == A

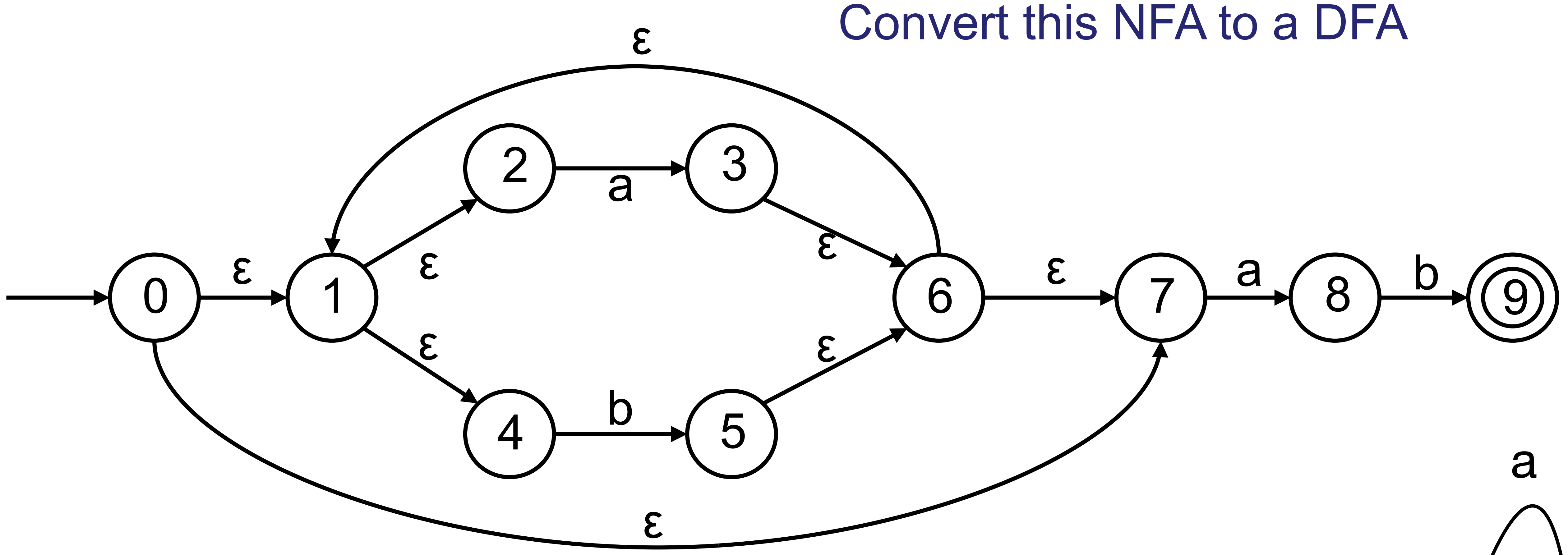
D = { 1, 2, 4, 5, 6, 8, 8, 9 } == ε|A



not minimal, A == C

Class Problem

Convert this NFA to a DFA

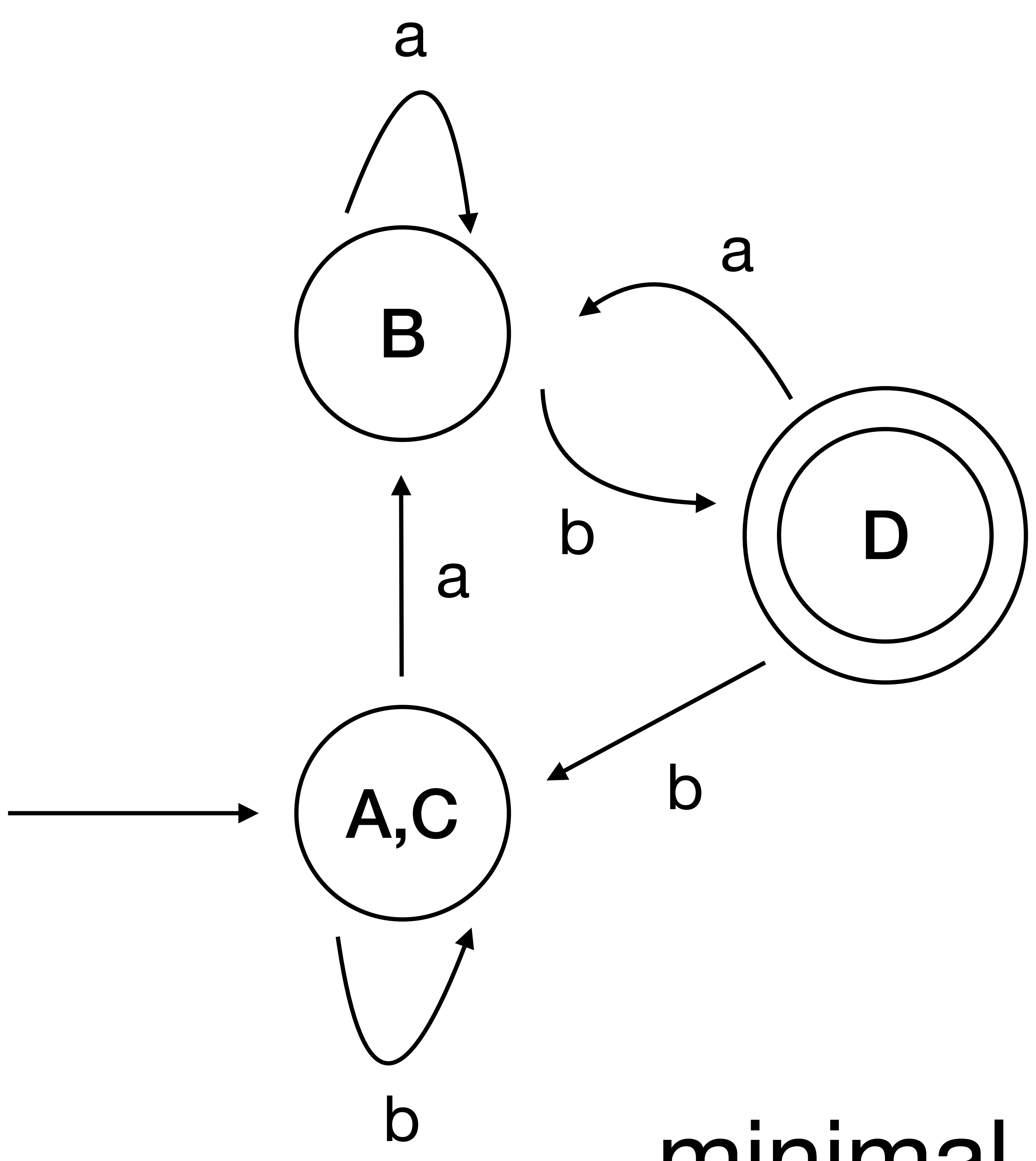


A = { 0, 1, 2, 4, 7 } == (a|b)*ab

B = { 1, 2, 3, 4, 6, 7, 8 } == b|A

C = { 1, 2, 4, 5, 6, 7 } == A

D = { 1, 2, 4, 5, 6, 8, 8, 9 } == ε|A



minimal

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

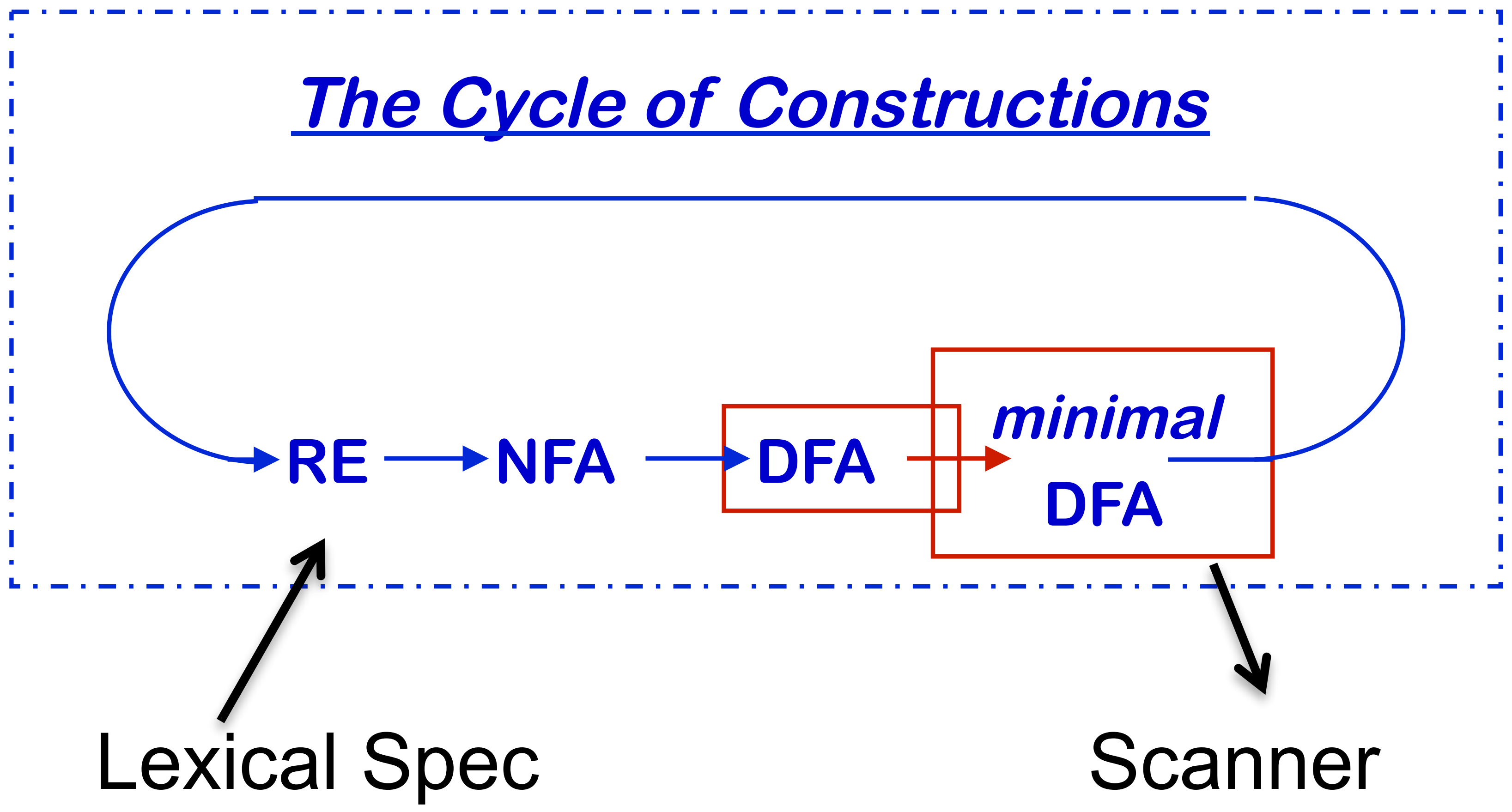
NFA Determinization: Correctness

The powerset construction takes an NFA **N** and constructs an equivalent DFA Pow(N).

Equivalent means that a string has an accepting trace in **N** (starting at the start state) if and only if it does in Pow(N) (starting at the start state).

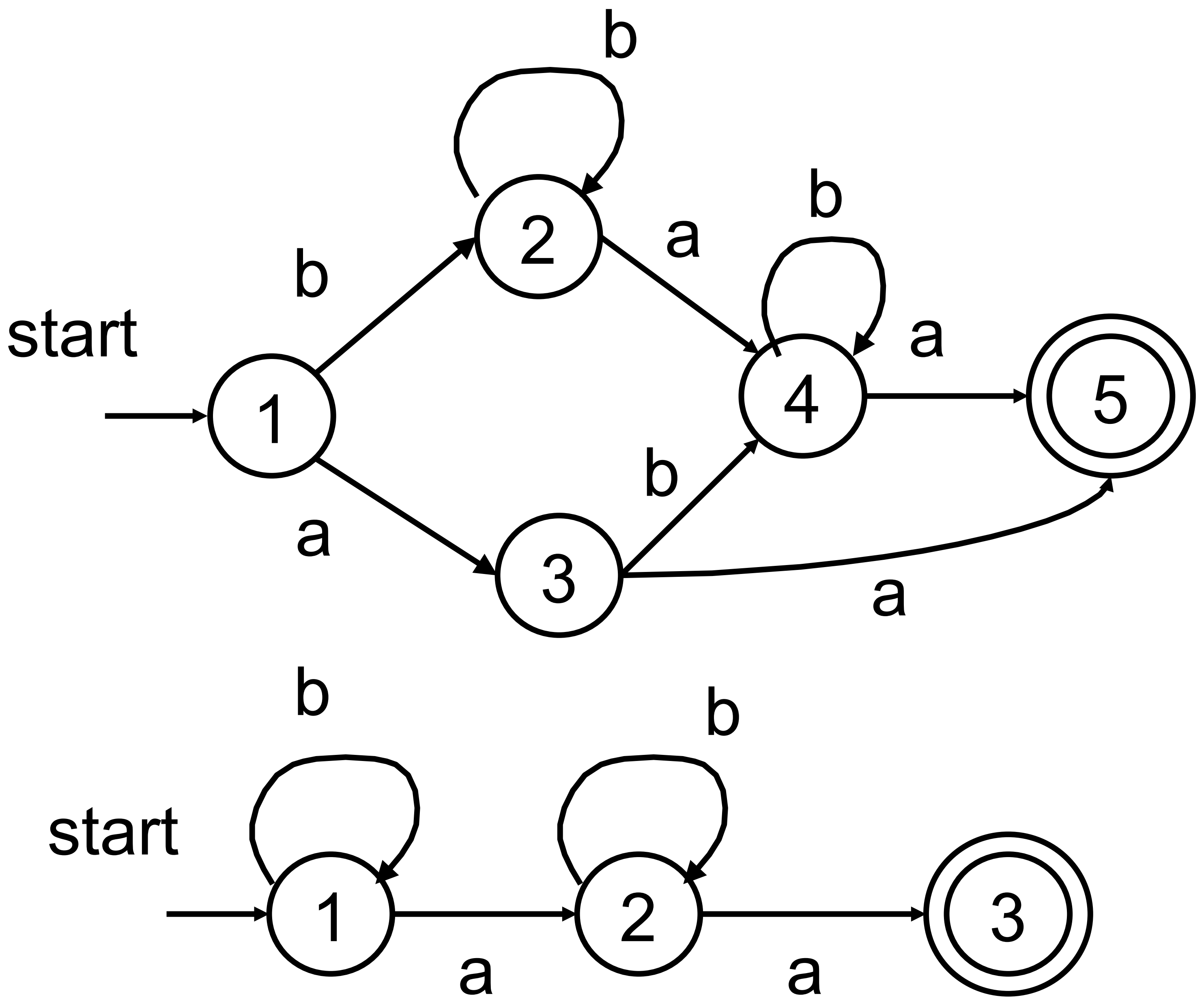
Idea of proof is to generalize to a statement about traces starting at **any** state:

- For any state s in N and accepting trace starting at s, show that **for all states** S in Pow(N) if s in S, then there is an accepting trace starting at S.
- For any state S in Pow(N), show that there **exists** a state s in S that has an accepting trace in N starting at s.
- Proof each of these holds by induction on the length of the trace.


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

State Minimization

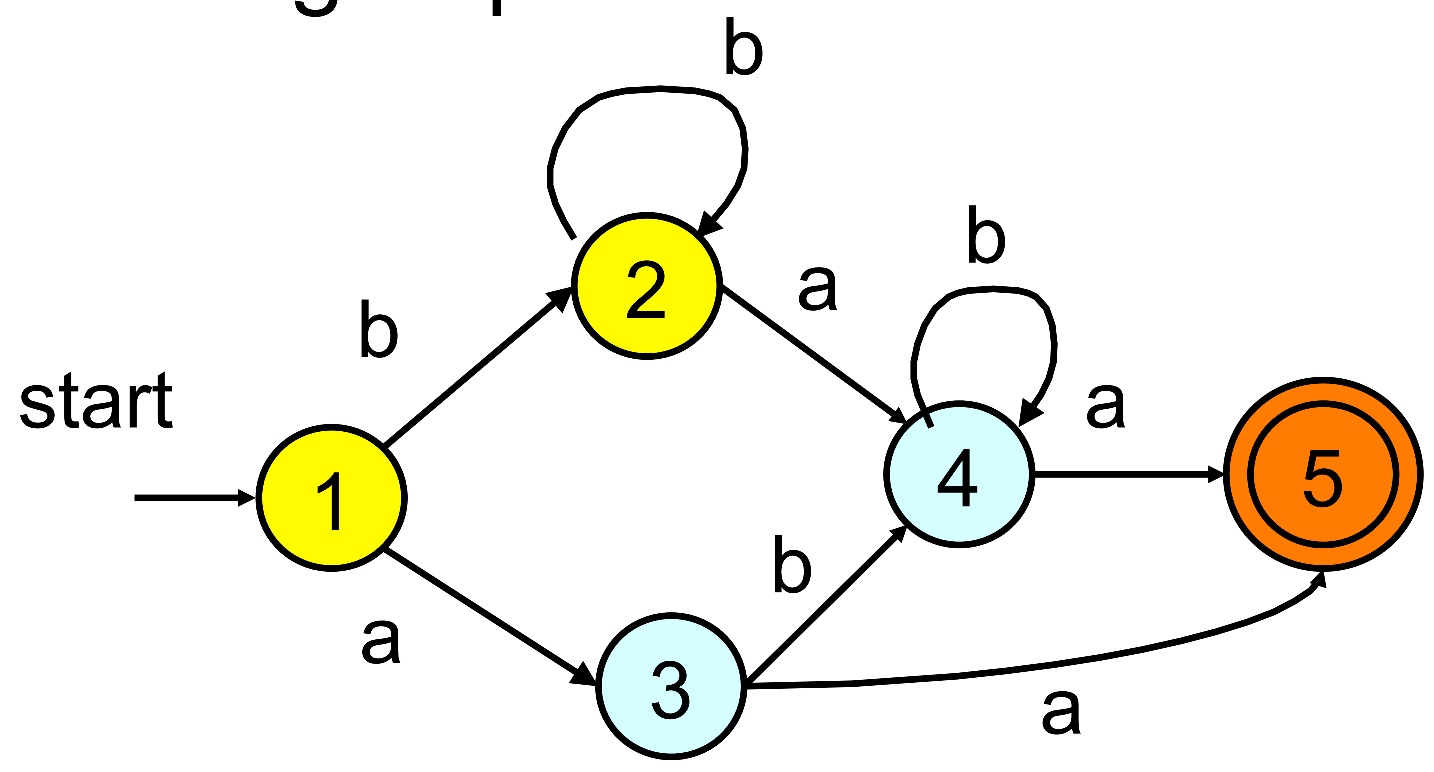
- Resulting DFA can be quite large
  - Contains redundant or equivalent states



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

DFA Minimization

Overview of algorithm:

Produce a **partition** of the states, so that states are in the same partition if they are equivalent.

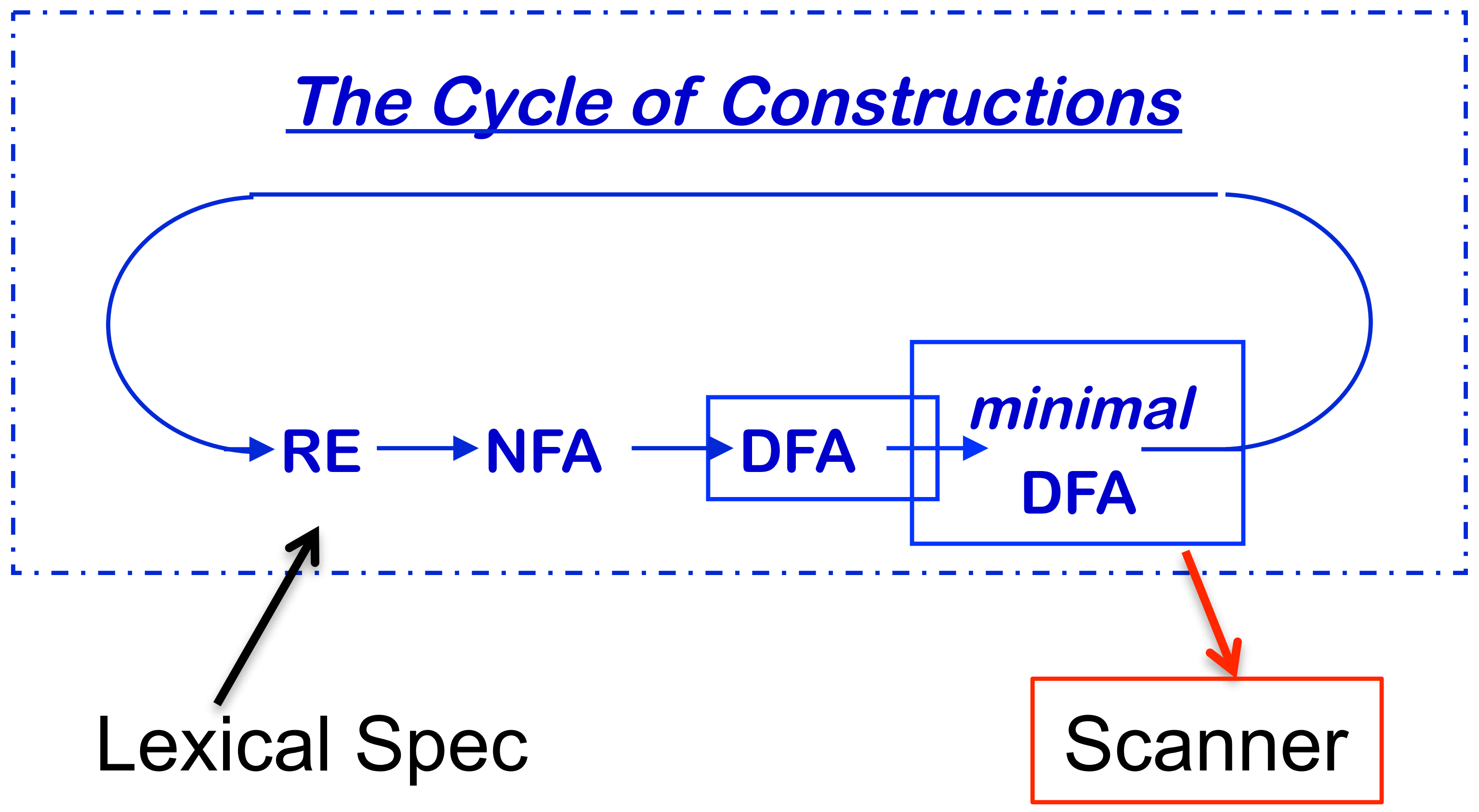
Initialize: two sets, accepting and rejecting

Update: If any states in the same partition make transitions to different partitions, split them.

Repeat until no new partitions are created

Minimized DFA has the partitions as states.

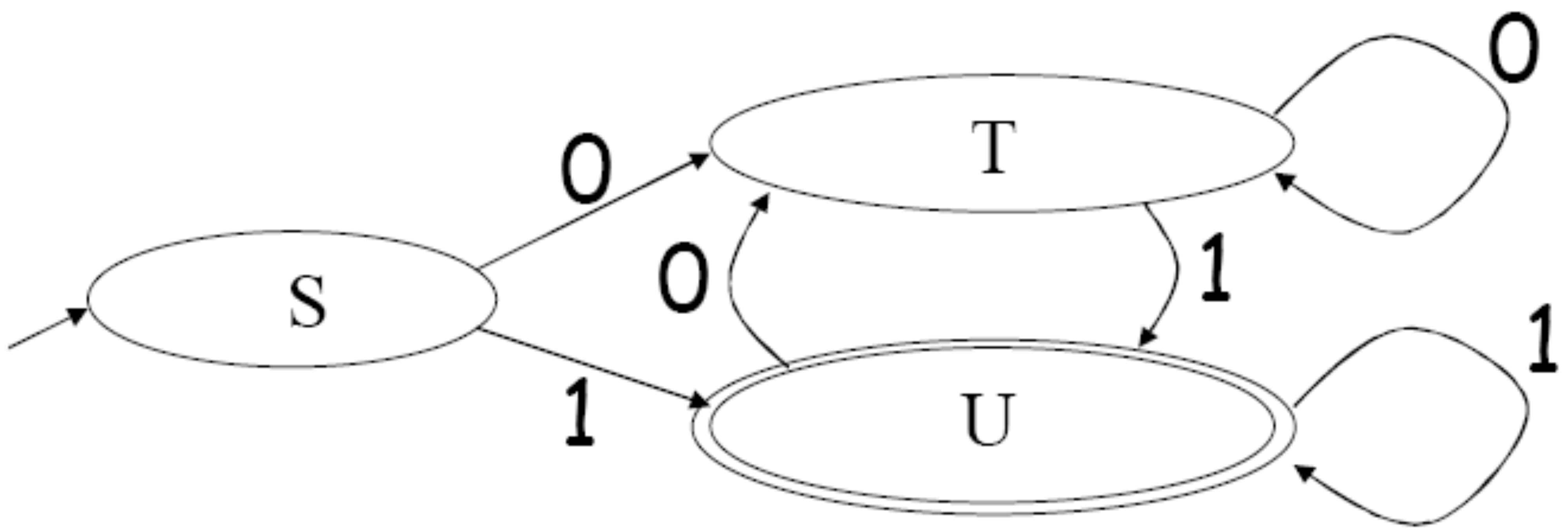
Similar to iterative fixpoint algorithms for dataflow analysis!


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
  - Step 2: Transforms NFA to DFA
- Implemented in various lexer generators tools: lex/flex (C), ocamllex (OCaml), logos/lalrpop (Rust)

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_]*
```

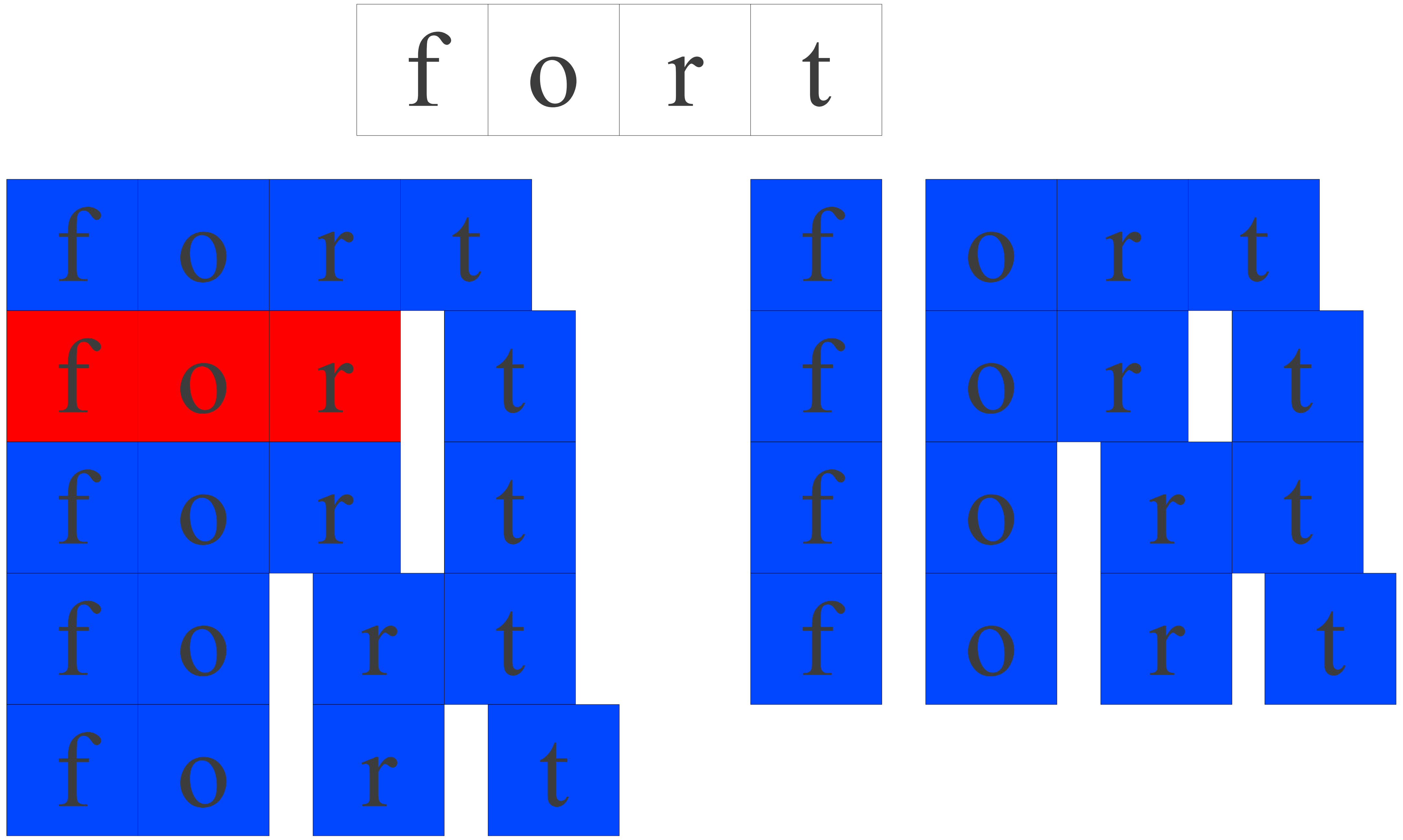
Lexing Ambiguities

```
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.

Implementing Maximal Munch

- Given a set of regular expressions, how can we use them to implement maximum munch?

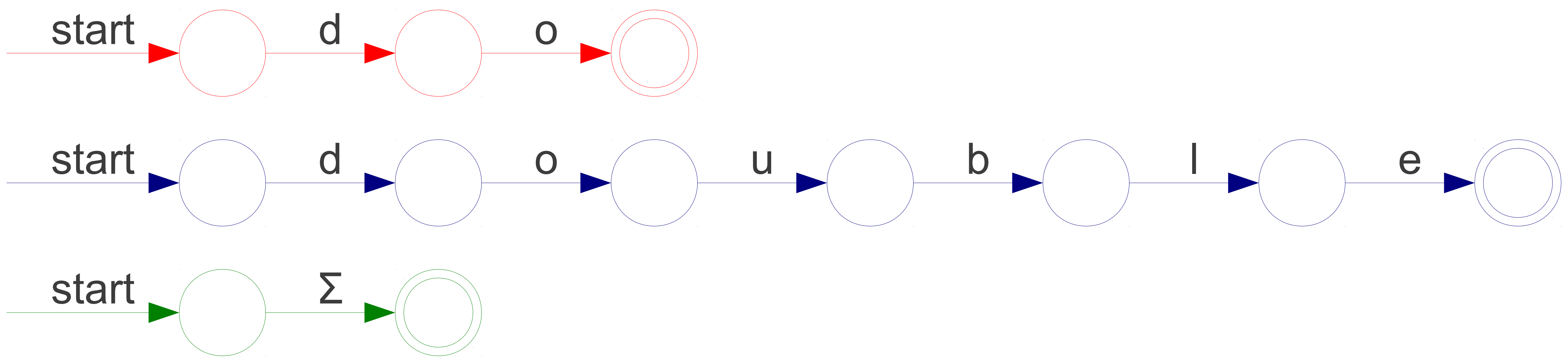
- Example

Implementing Maximal Munch

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

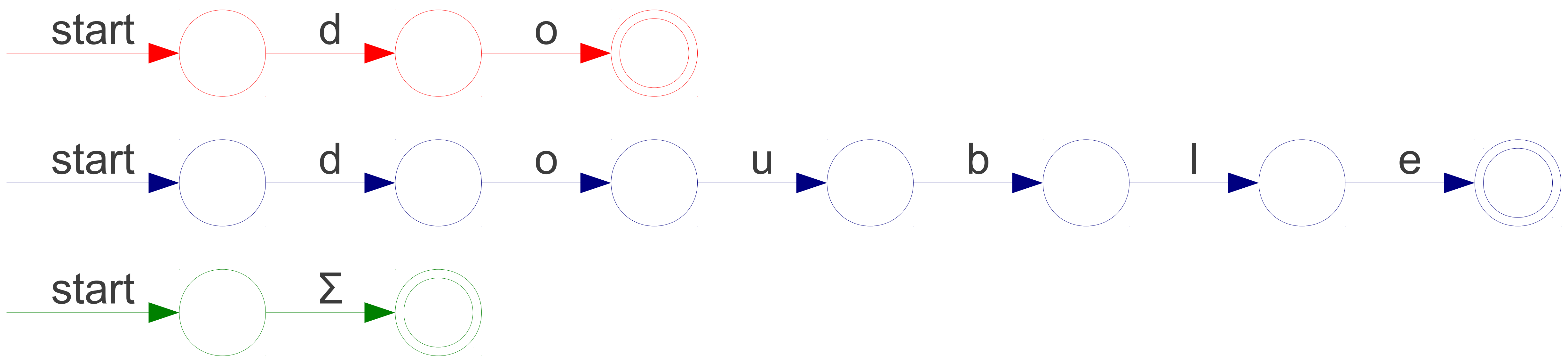
Implementing Maximal Munch

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



Implementing Maximal Munch

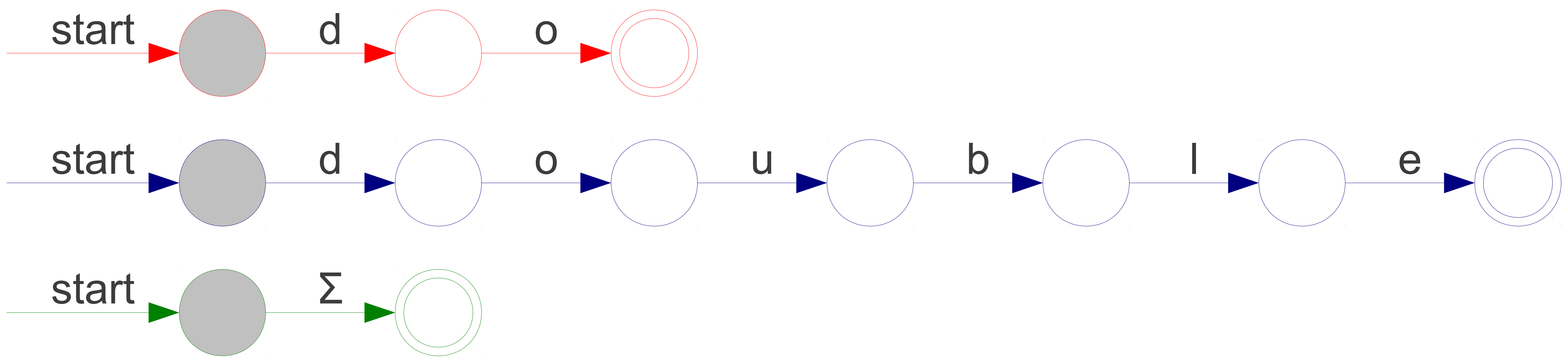
`T_Do`	`do`
`T_Double`	`double`
`T_Mystery`	`[A-Za-z]`



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

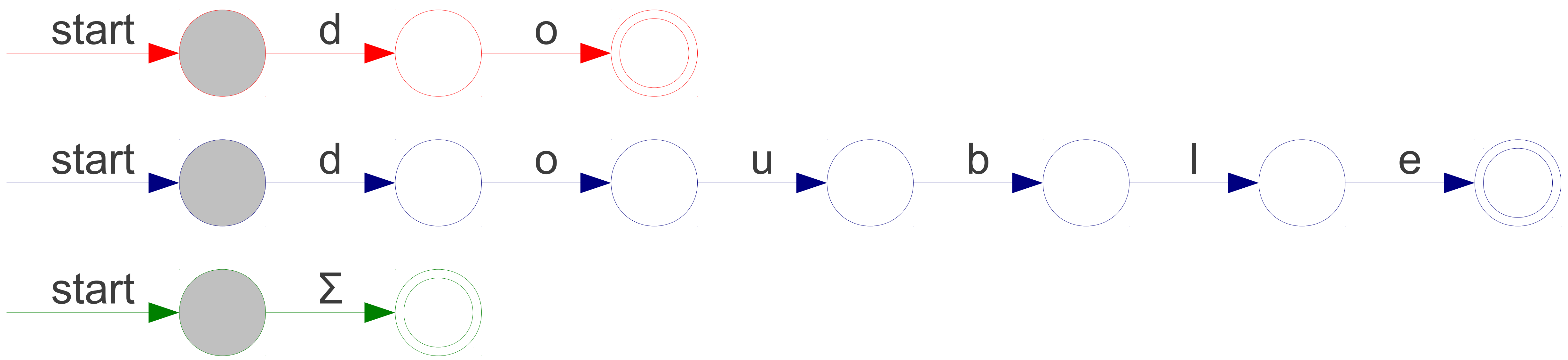
```
T_Do         do
T_Double     double
T_Mystery    [A-Za-z]
```



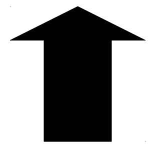
D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

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

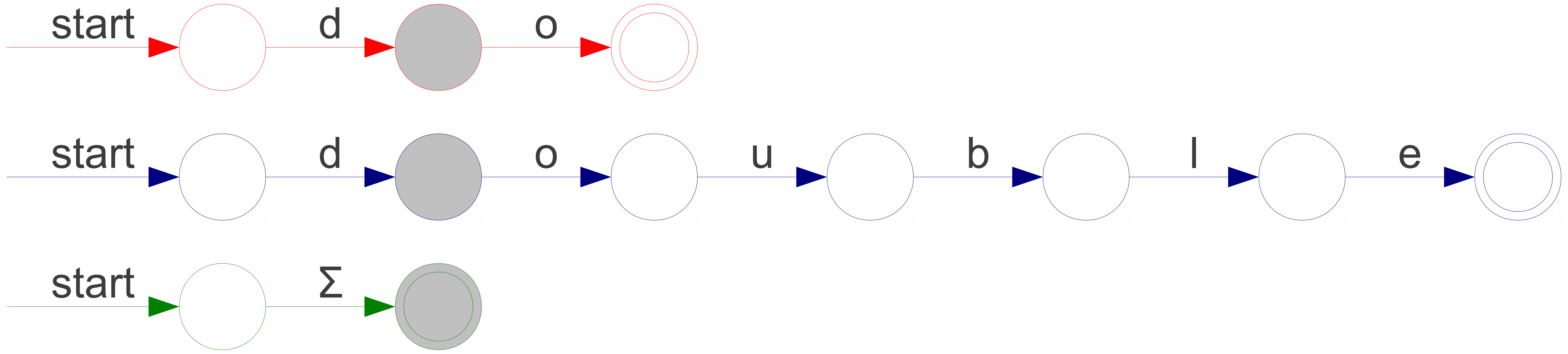


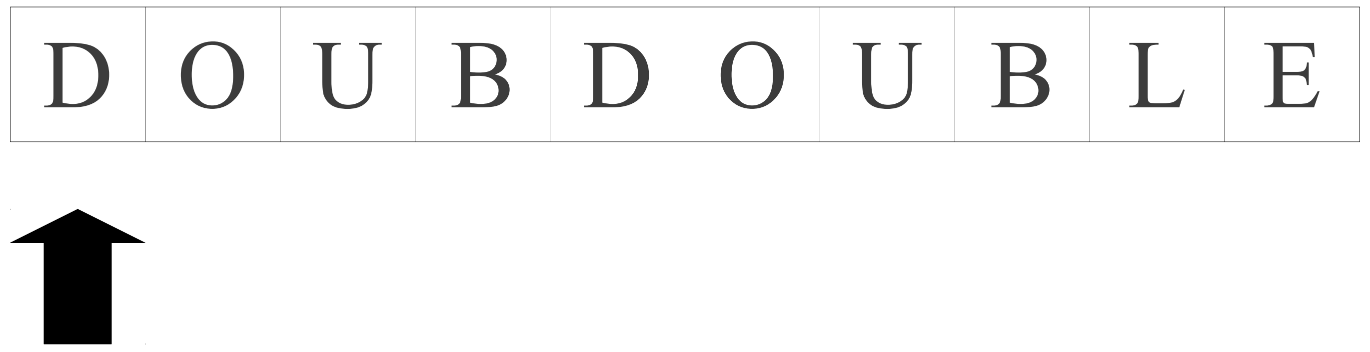
D	O	U	B	D	O	U	B	L	E



Implementing Maximal Munch

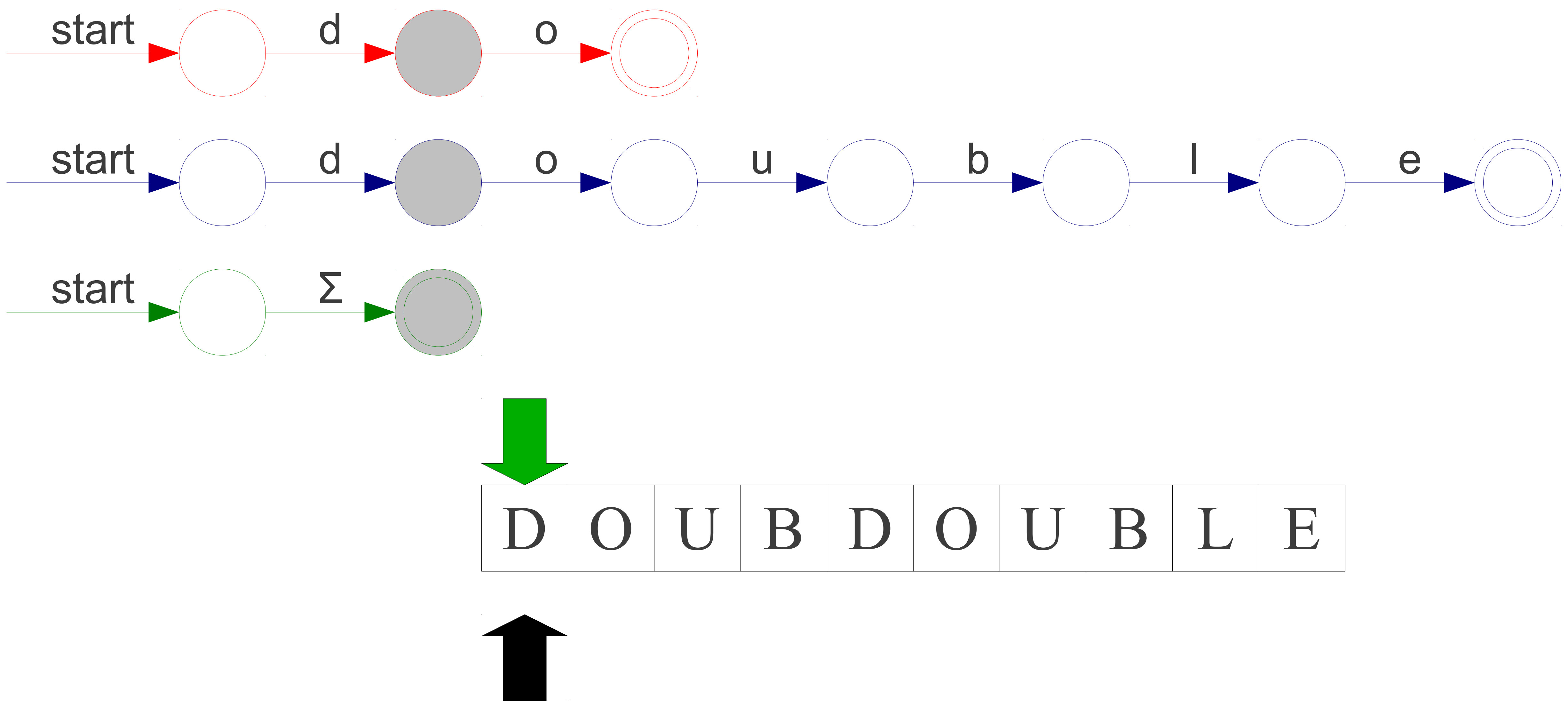
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```





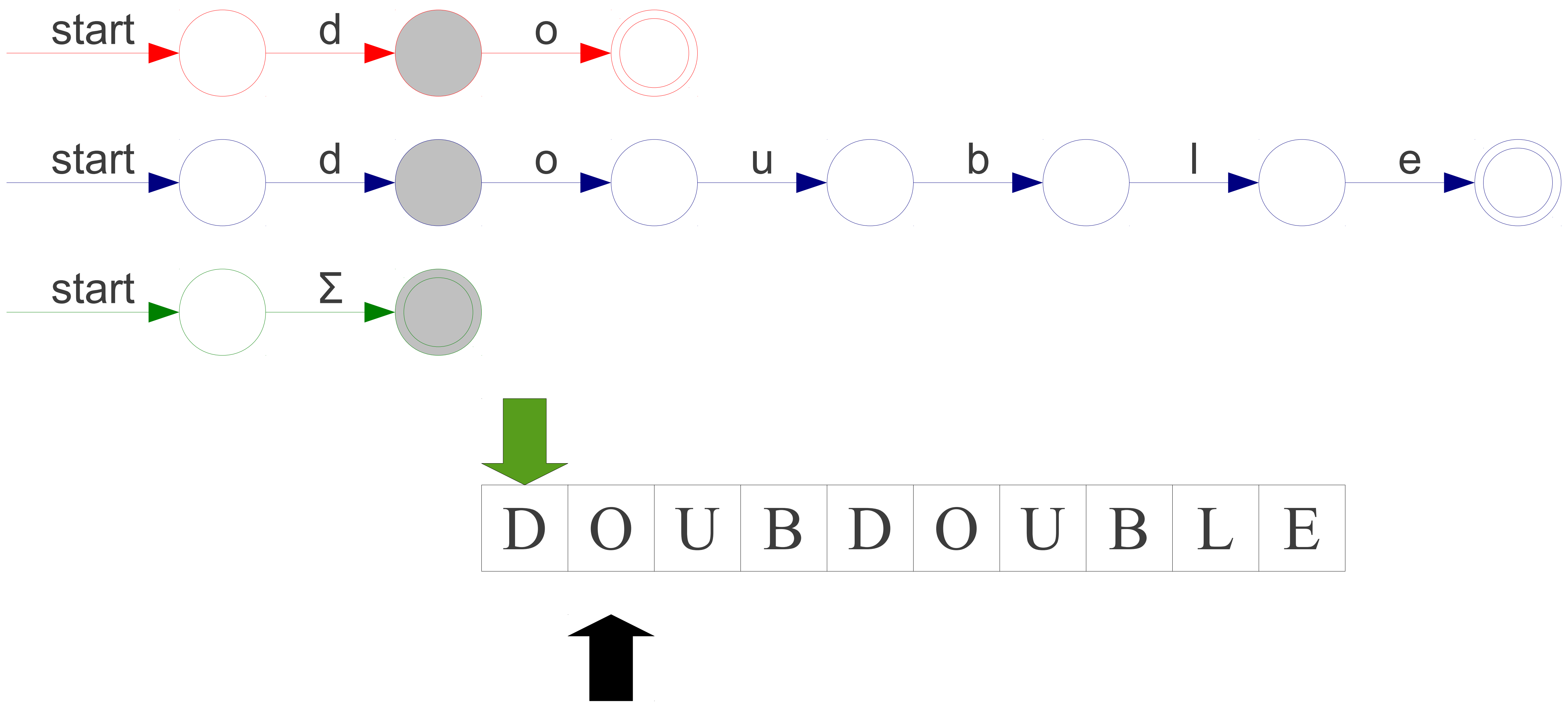
Implementing Maximal Munch

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



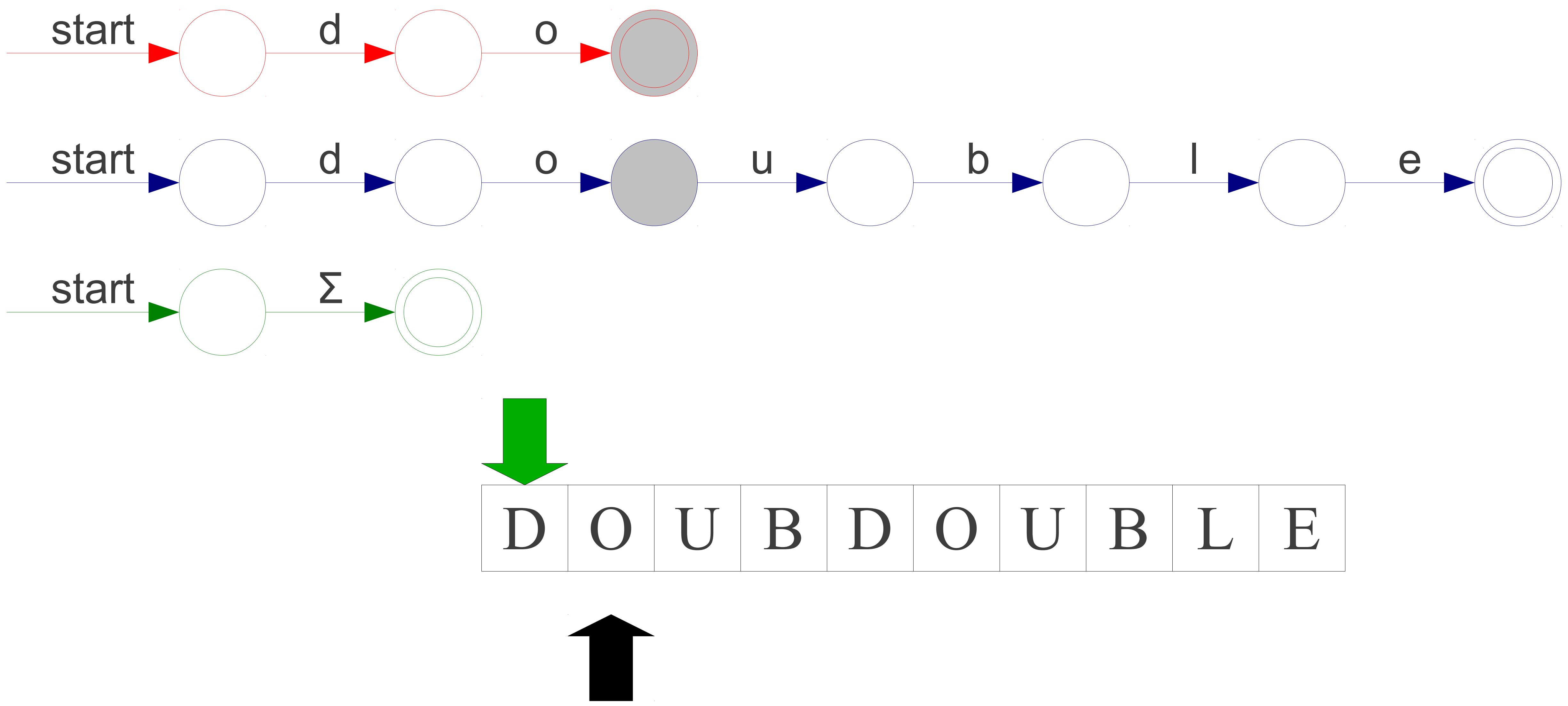
Implementing Maximal Munch

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



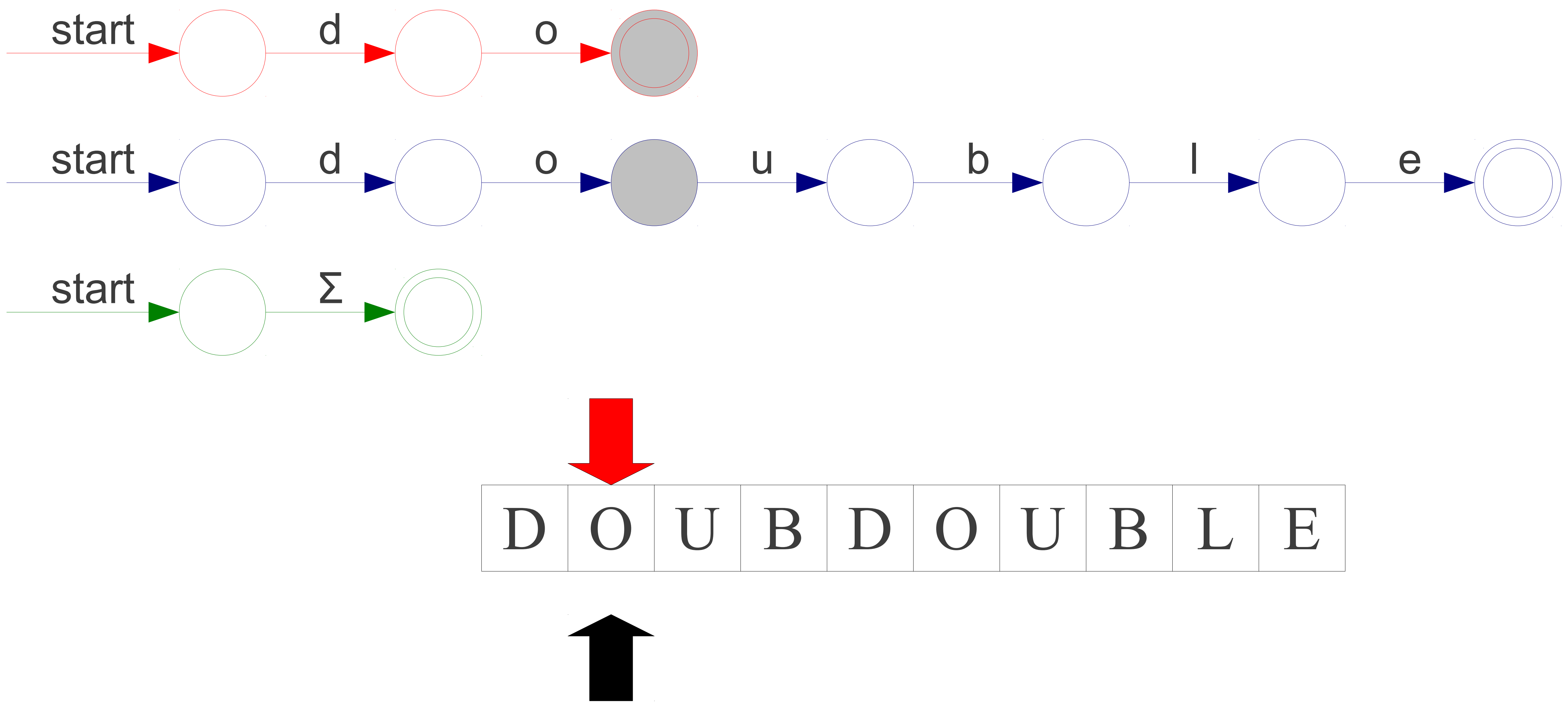
Implementing Maximal Munch

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



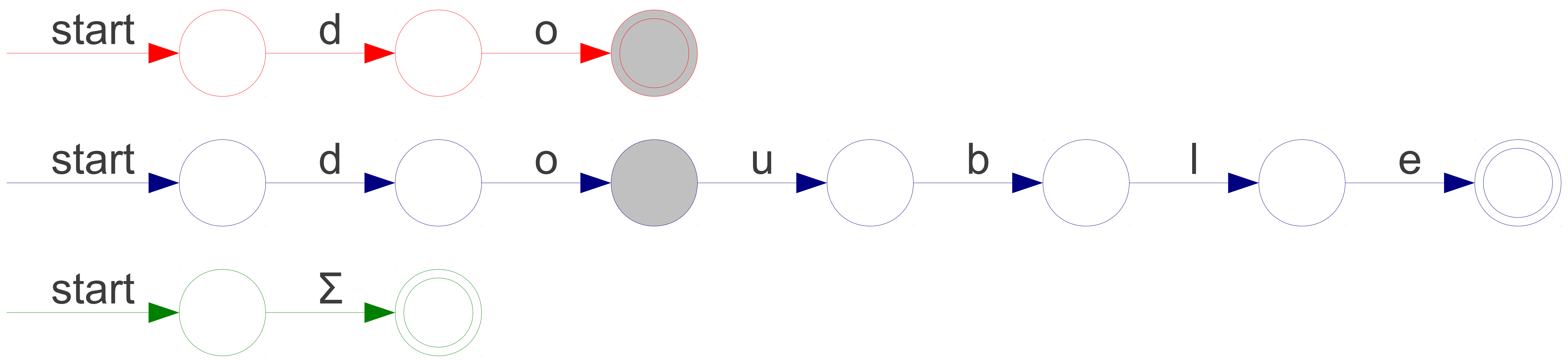
Implementing Maximal Munch

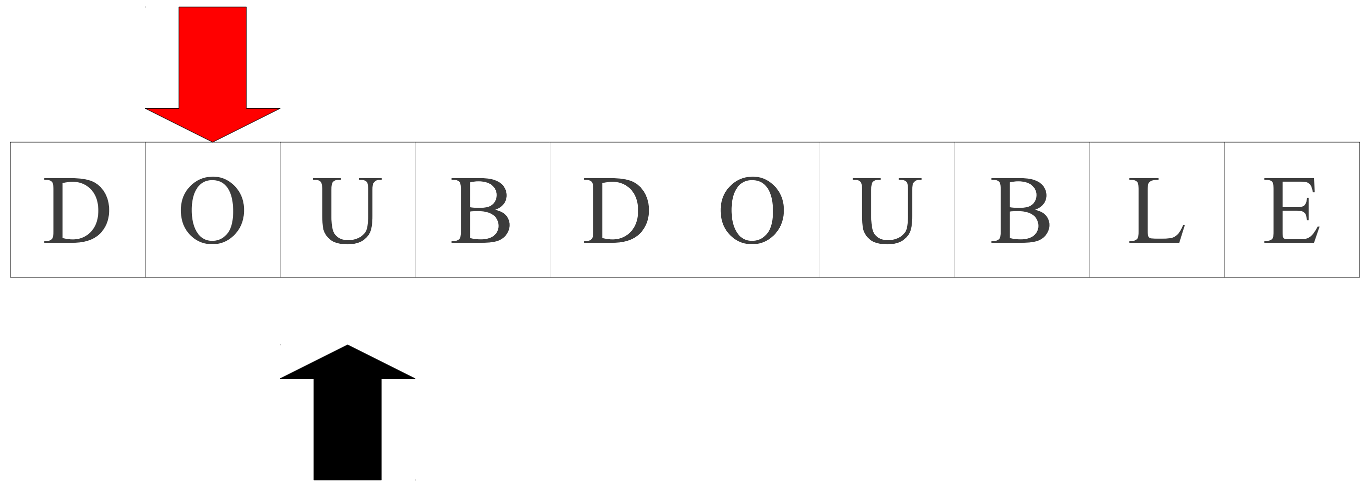
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

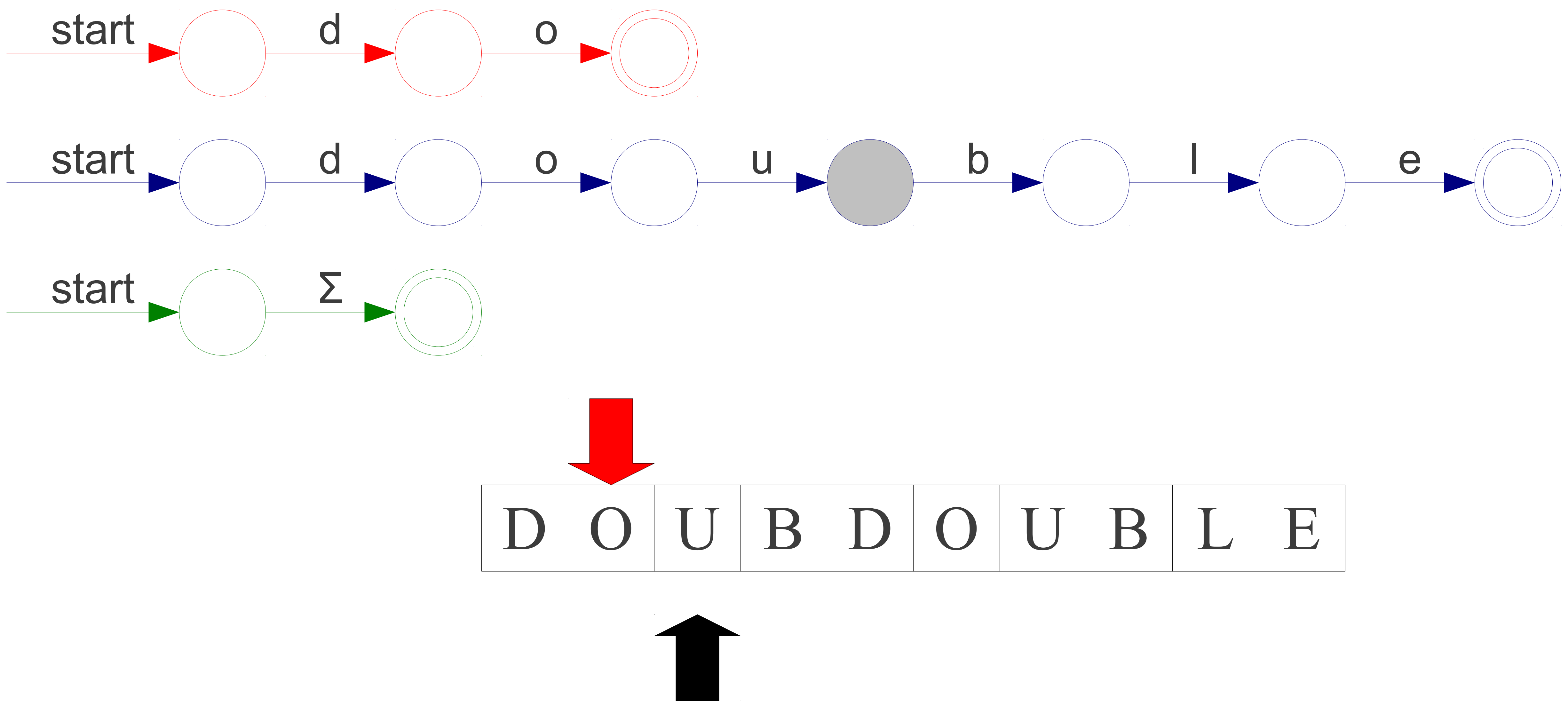
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





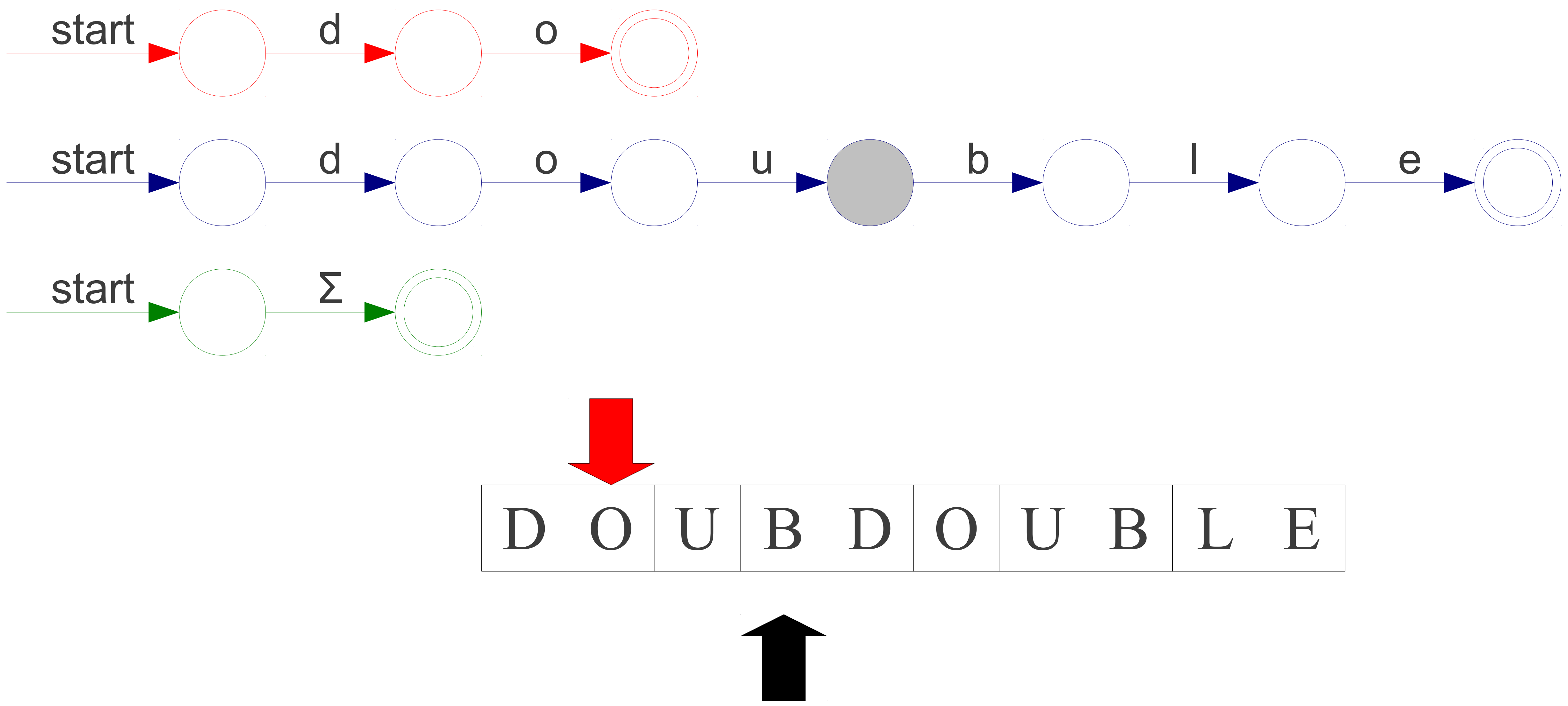
Implementing Maximal Munch

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



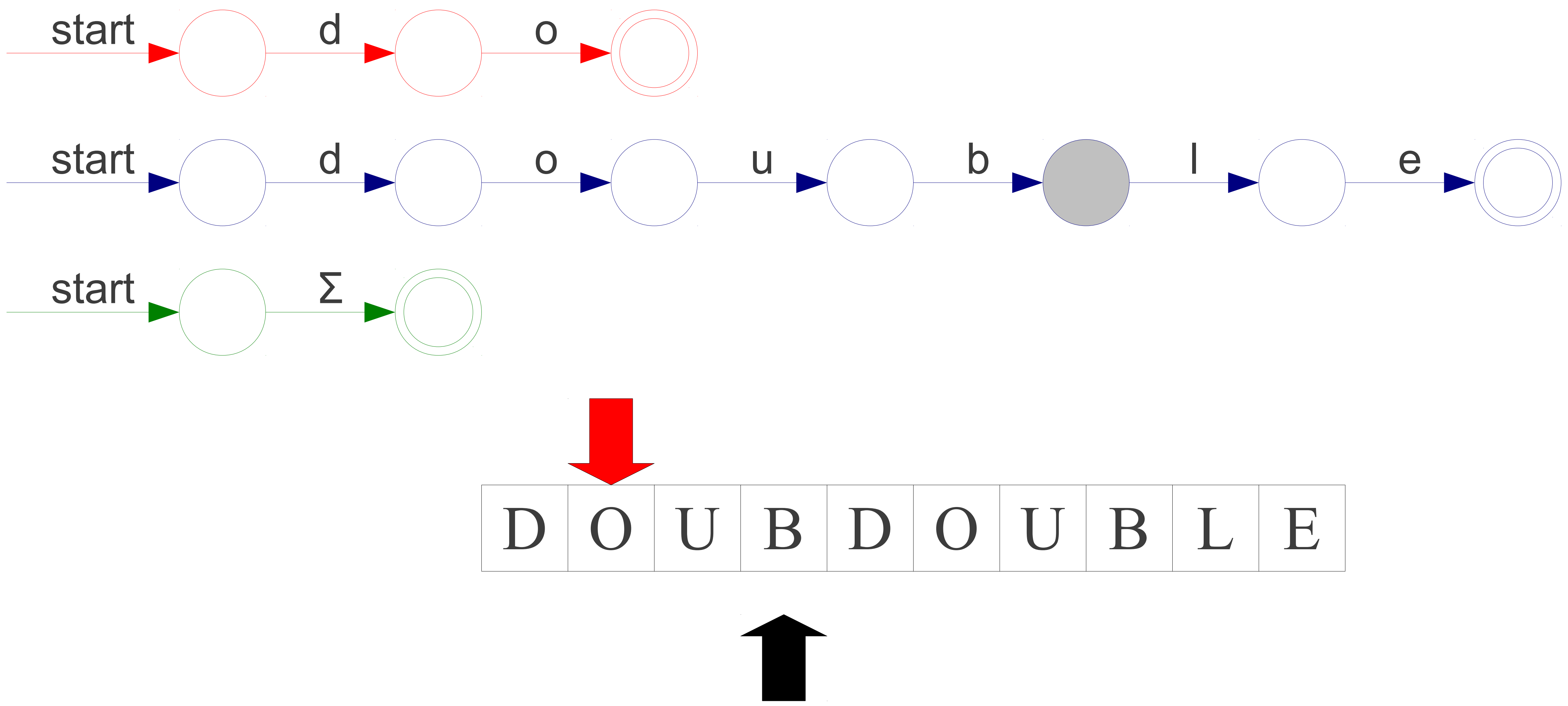
Implementing Maximal Munch

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



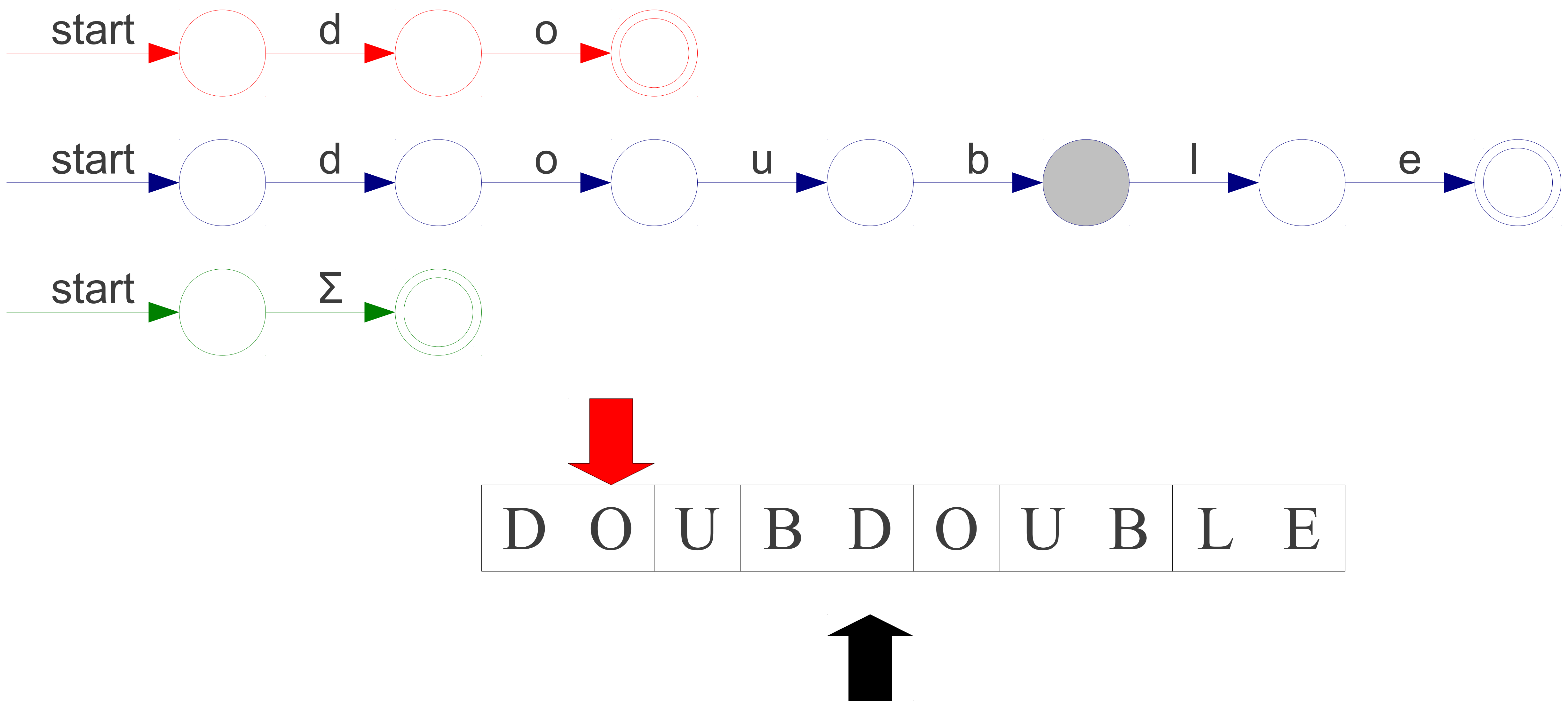
Implementing Maximal Munch

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



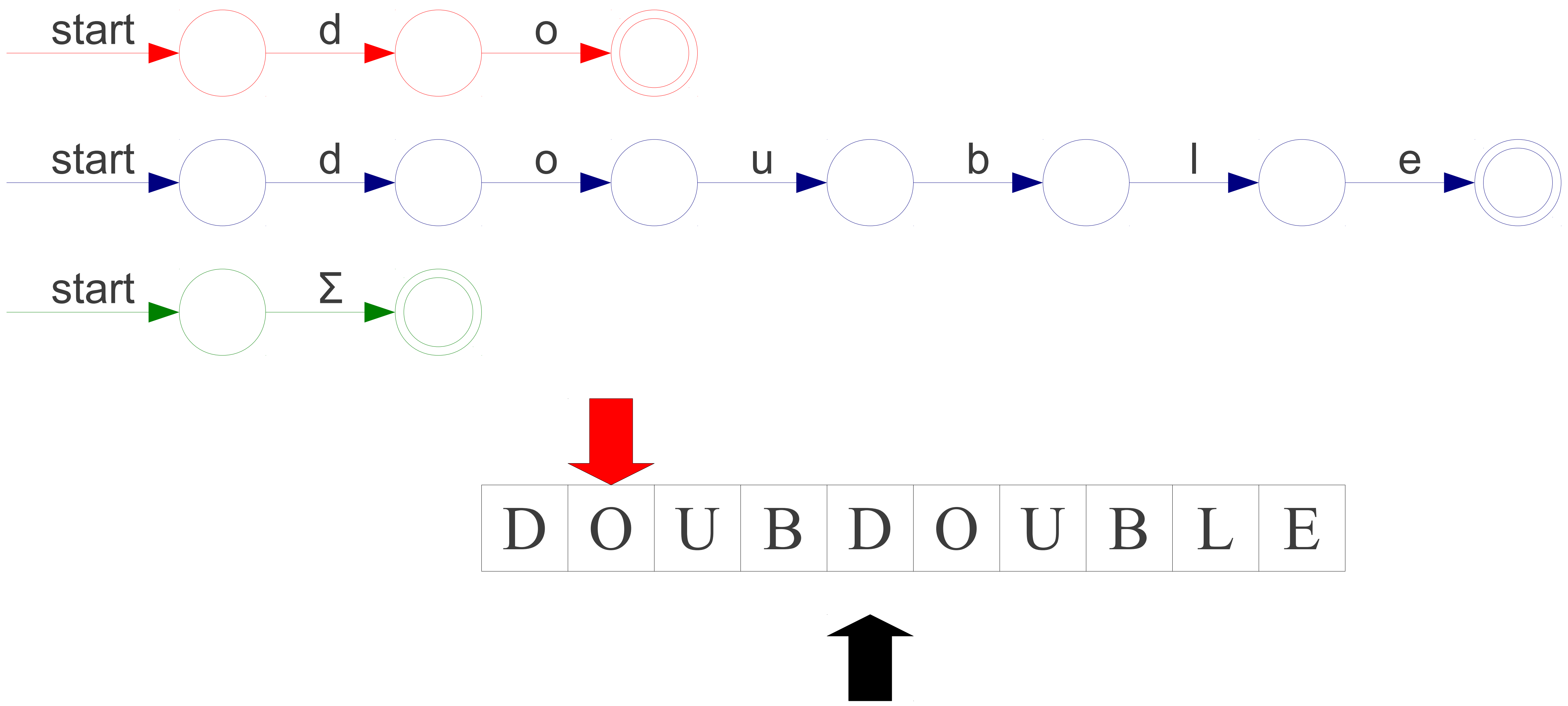
Implementing Maximal Munch

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



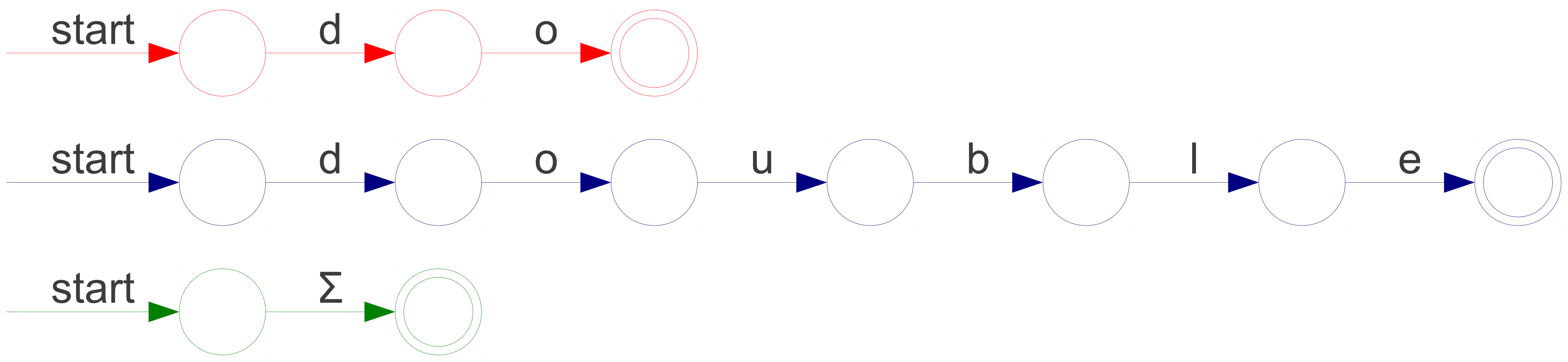
Implementing Maximal Munch

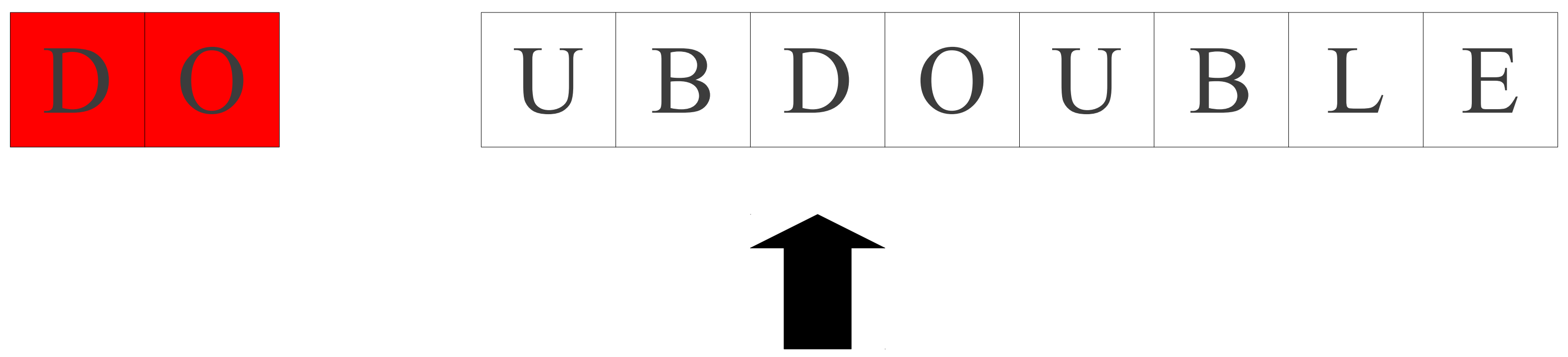
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

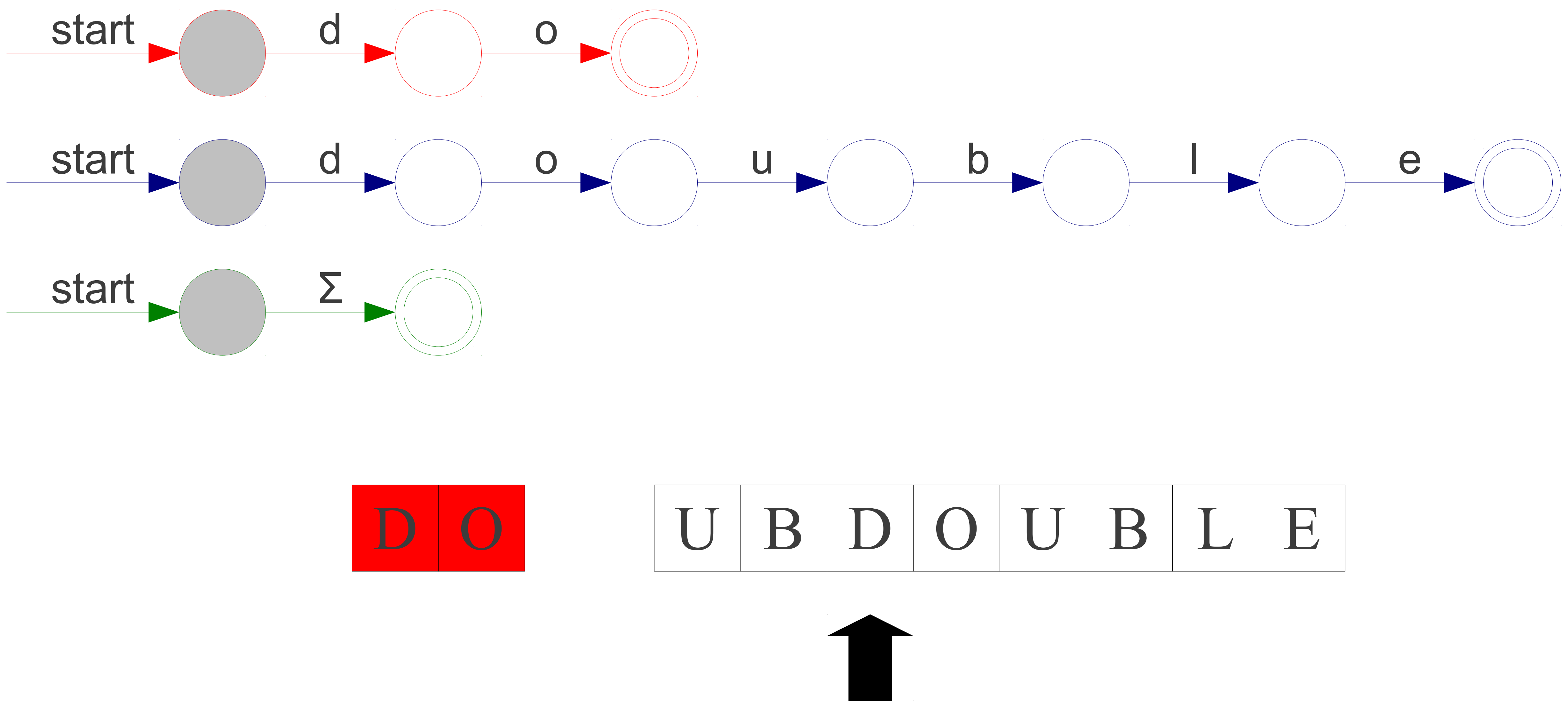
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





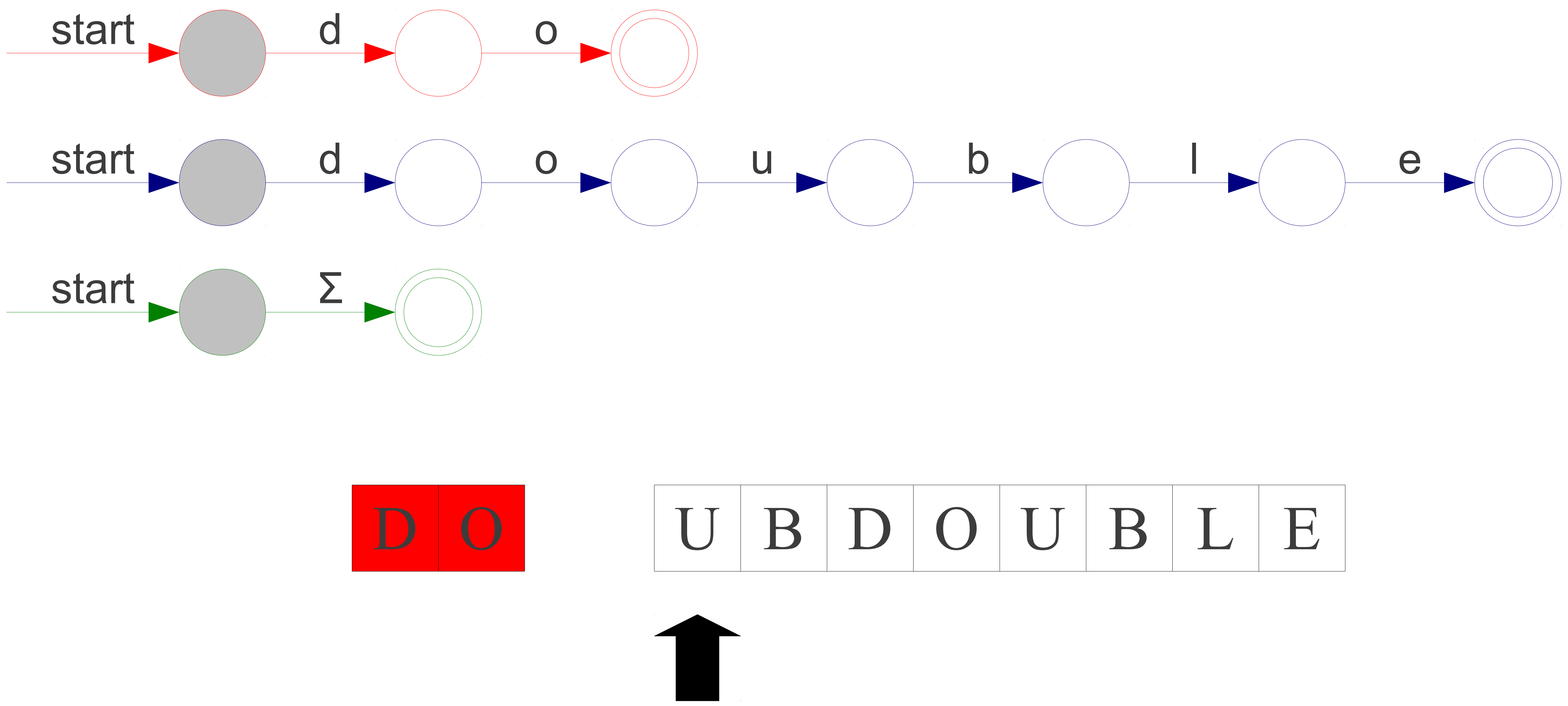
Implementing Maximal Munch

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



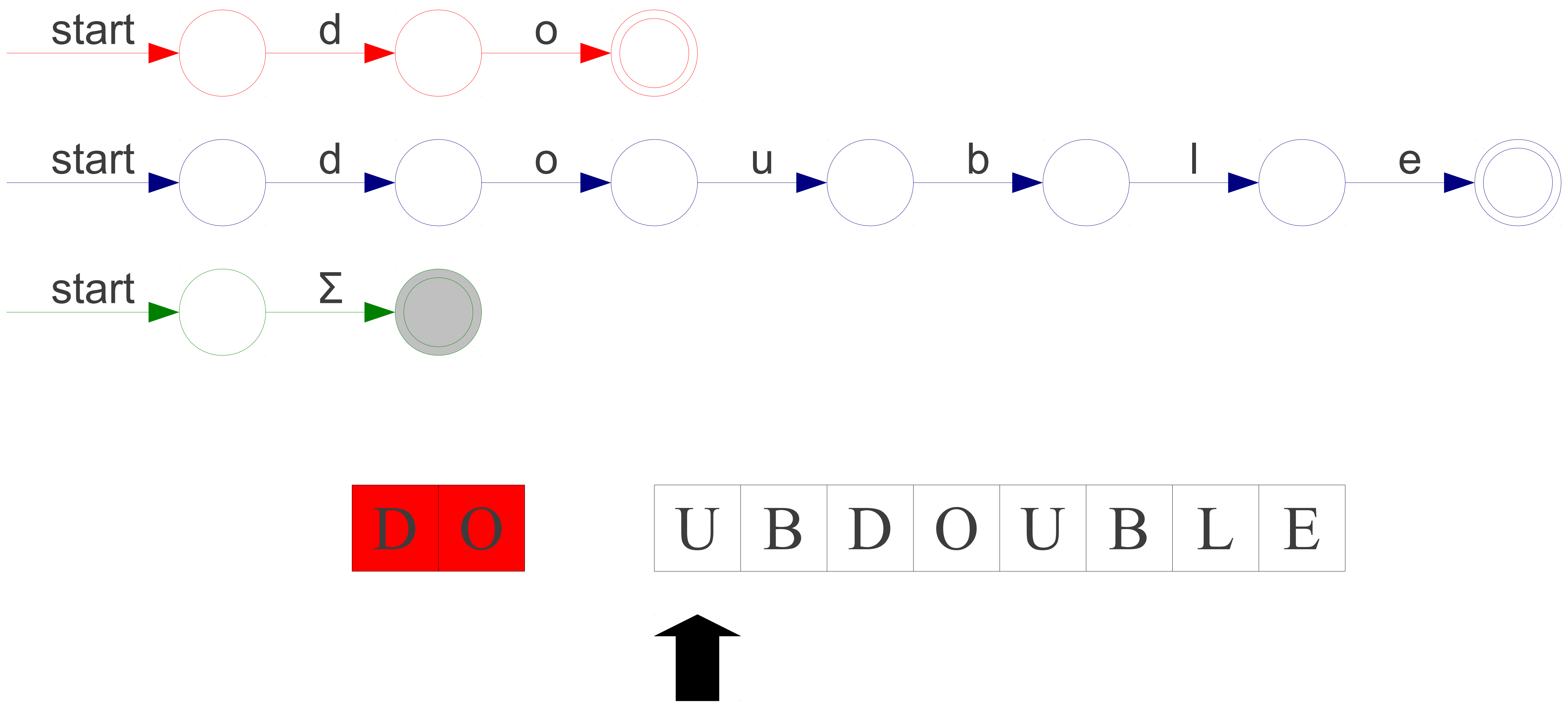
Implementing Maximal Munch

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



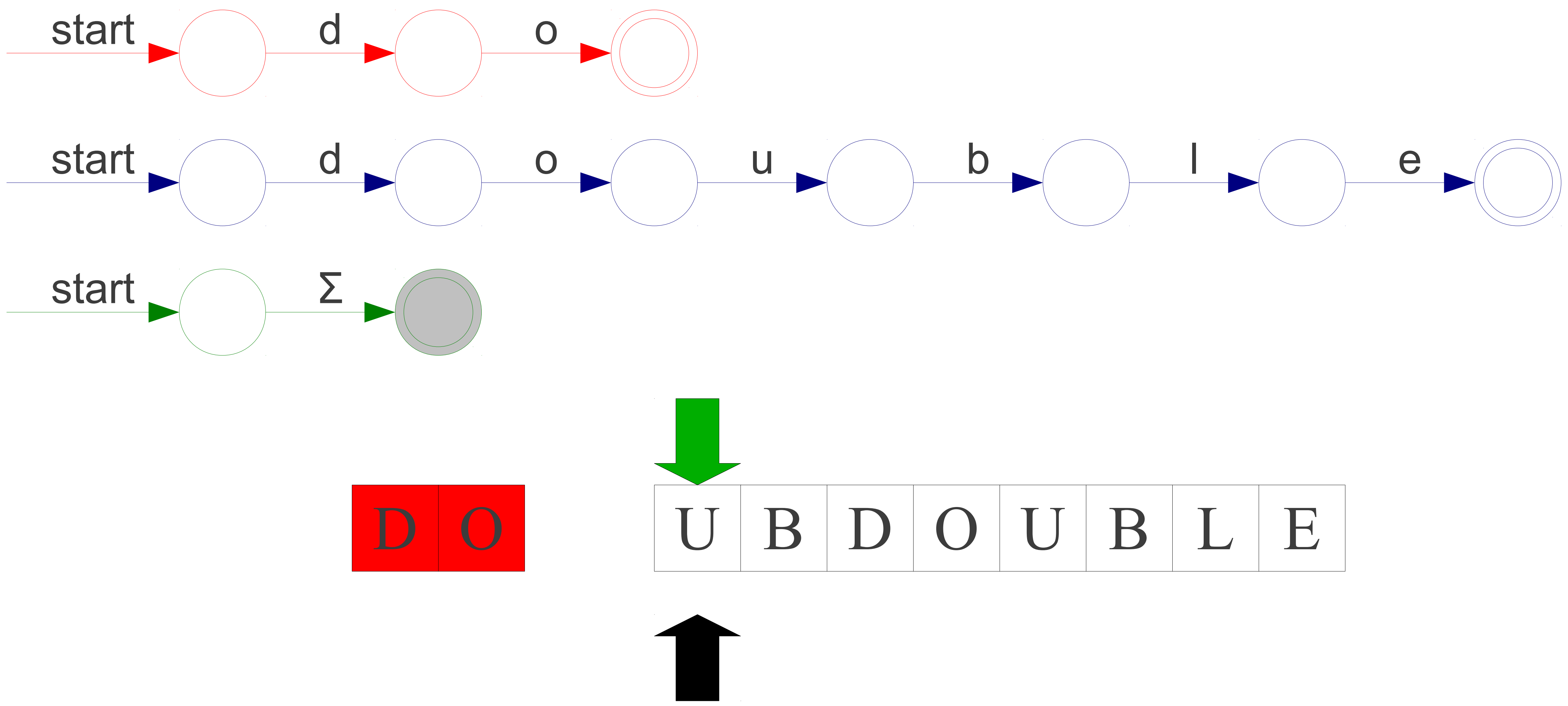
Implementing Maximal Munch

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



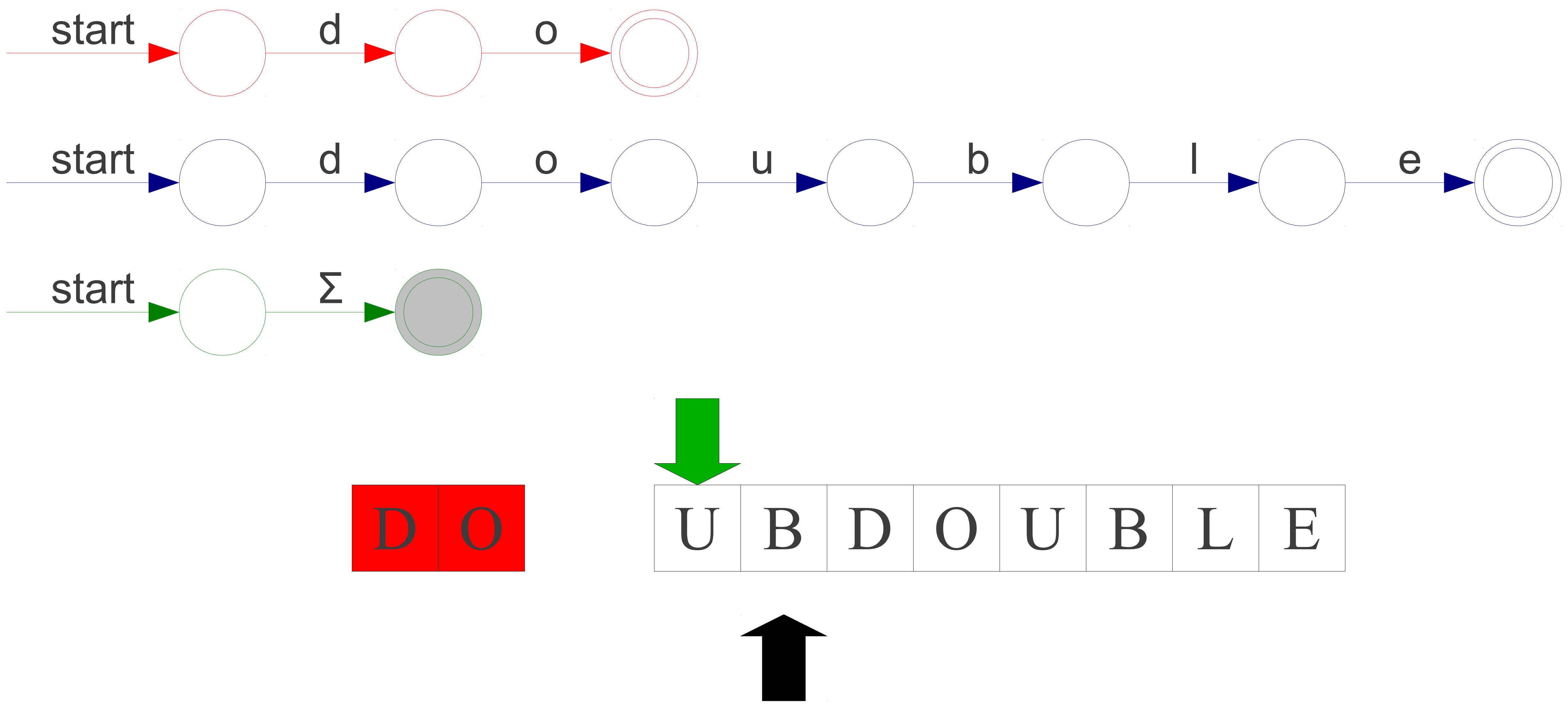
Implementing Maximal Munch

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



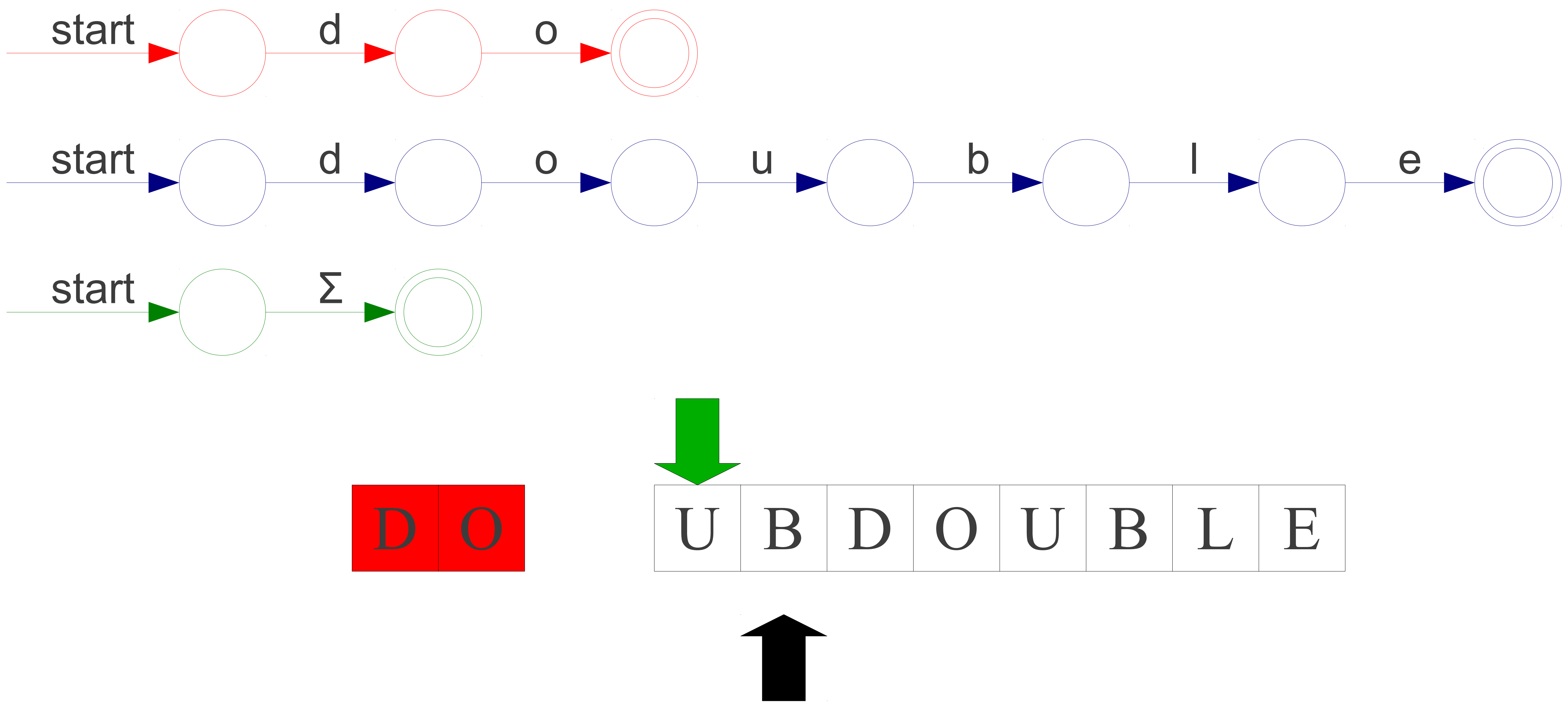
Implementing Maximal Munch

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



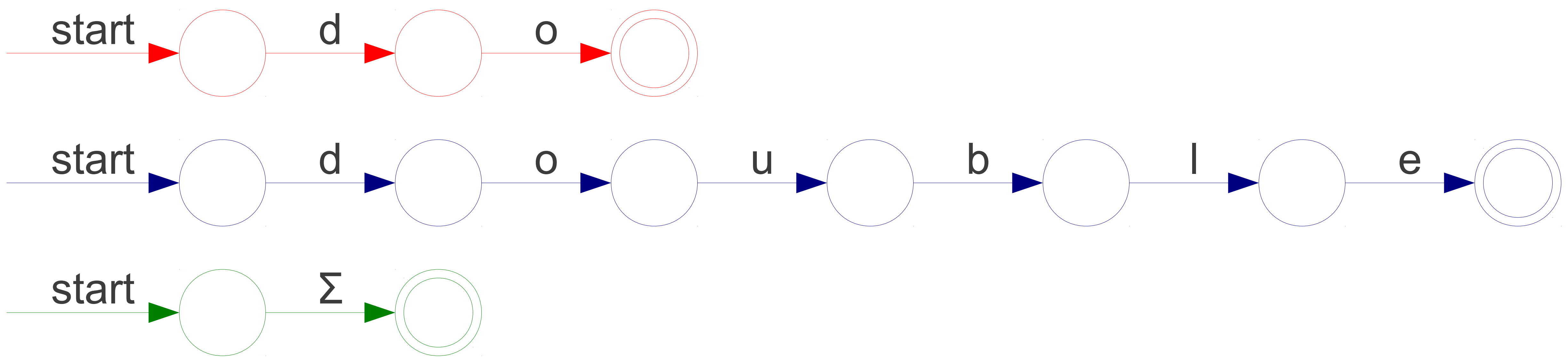
Implementing Maximal Munch

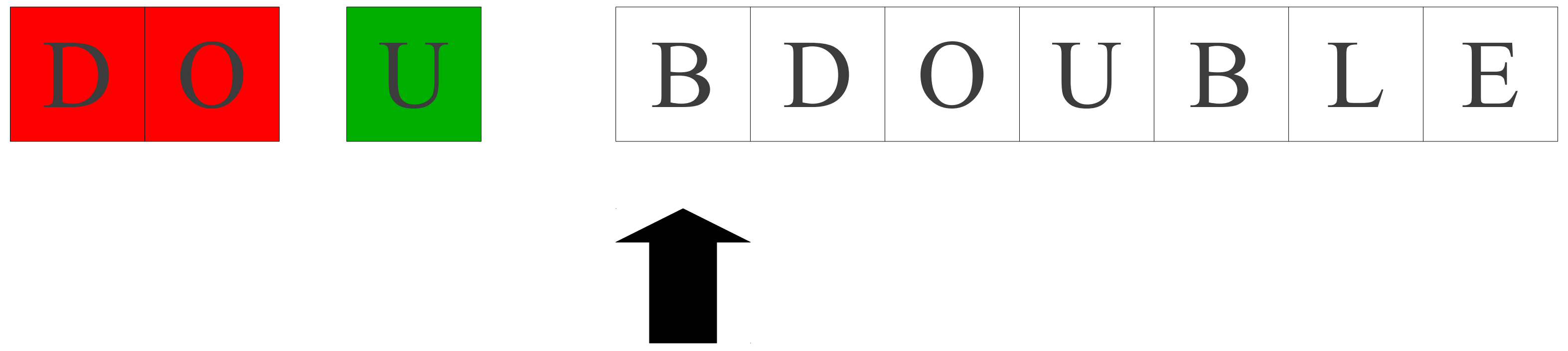
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]



Implementing Maximal Munch

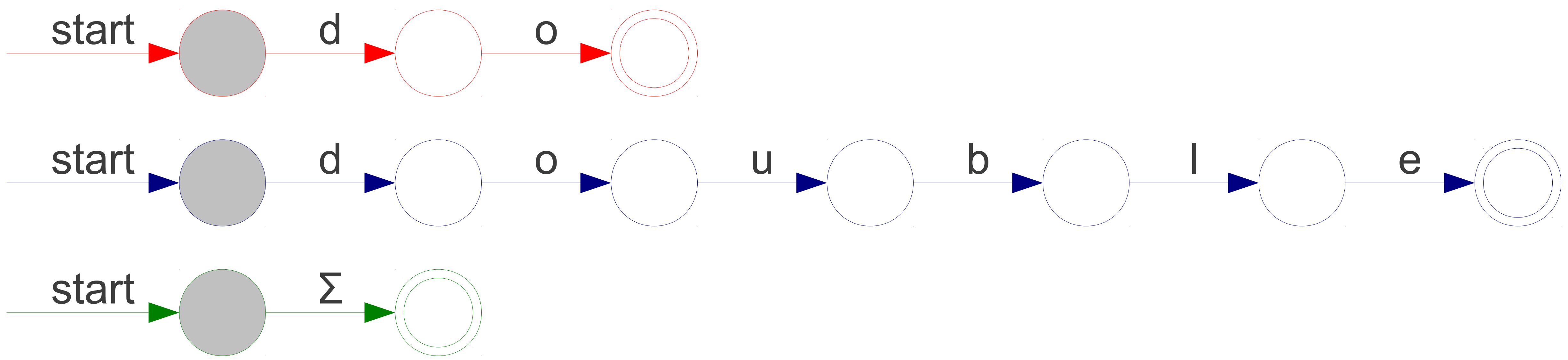
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]

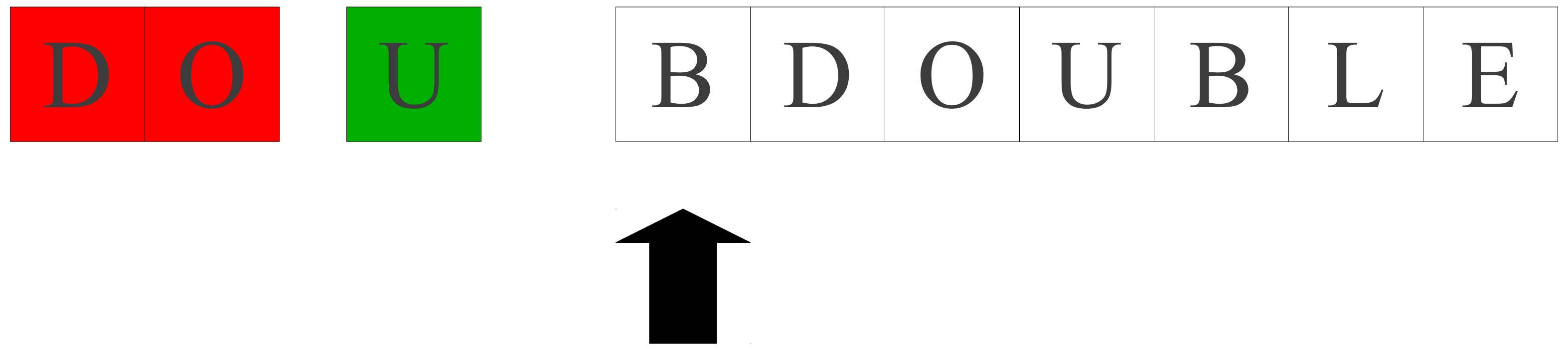




Implementing Maximal Munch

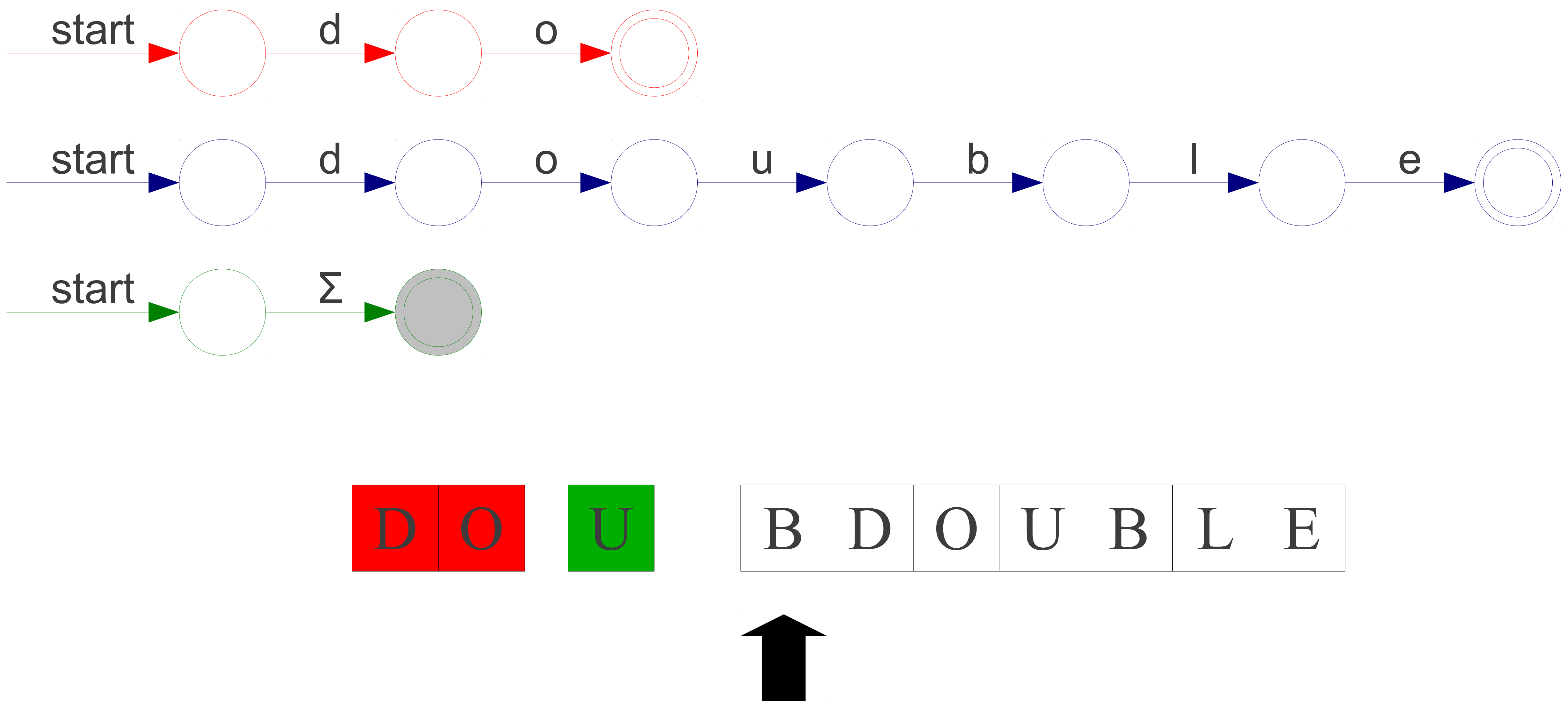
T_Do	do
T_Double	double
T_Mystery	[A-Za-z]





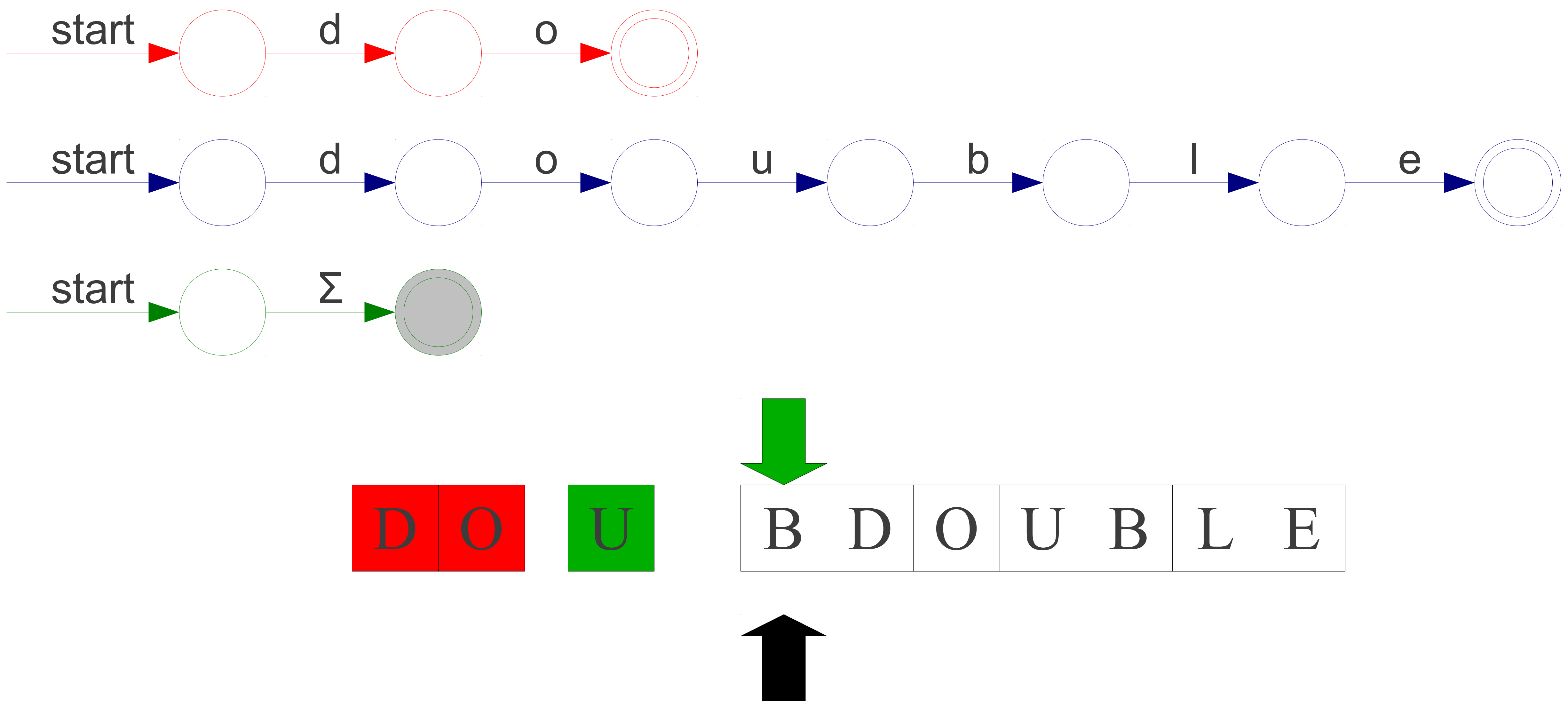
Implementing Maximal Munch

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



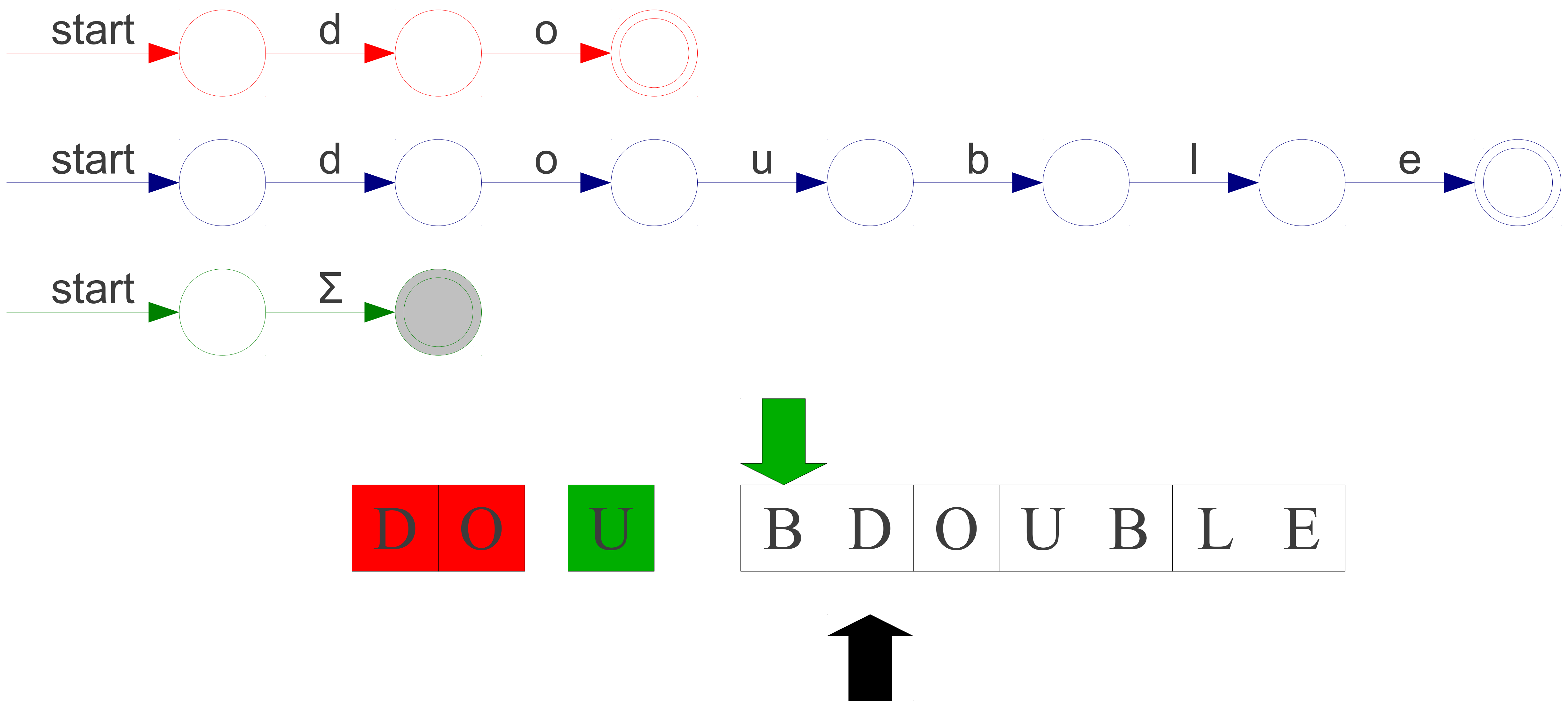
Implementing Maximal Munch

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



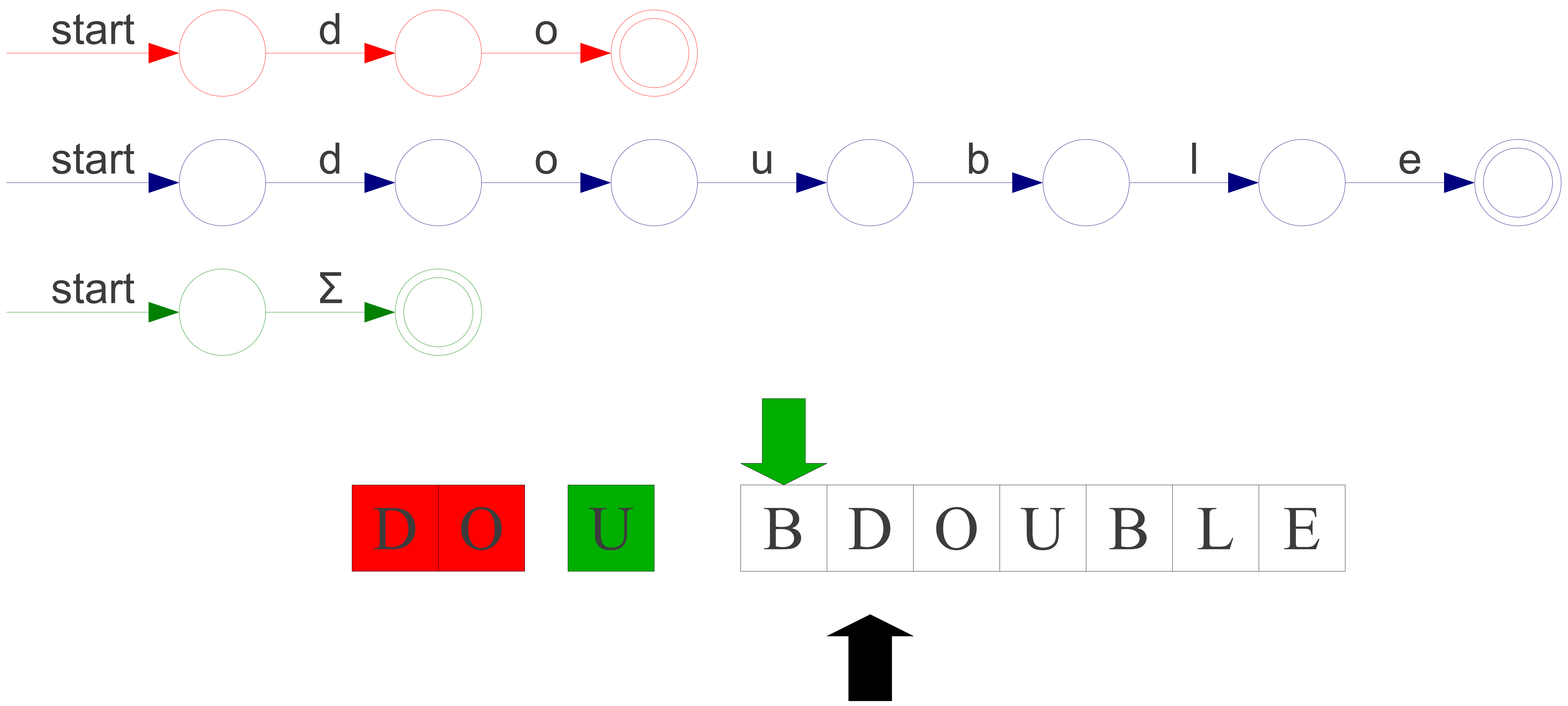
Implementing Maximal Munch

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



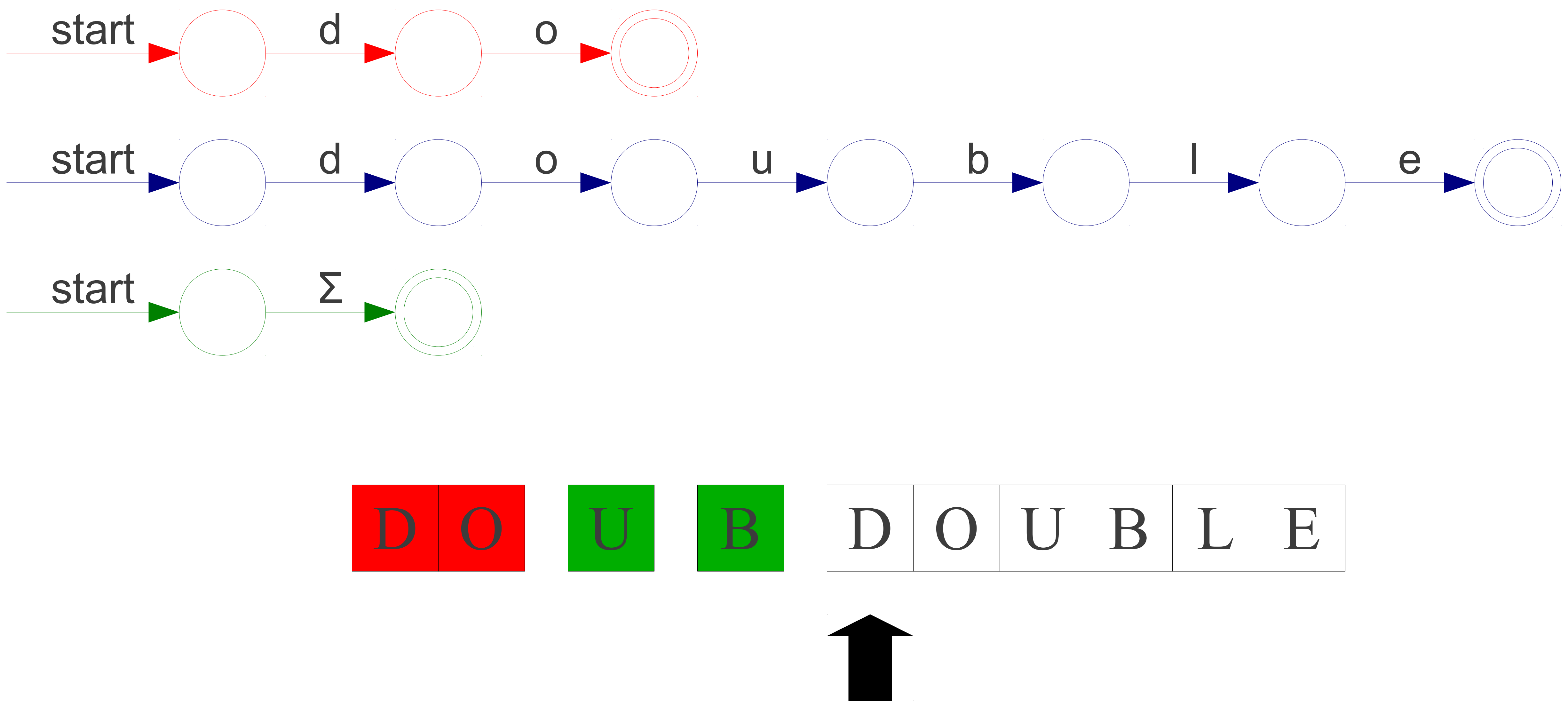
Implementing Maximal Munch

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



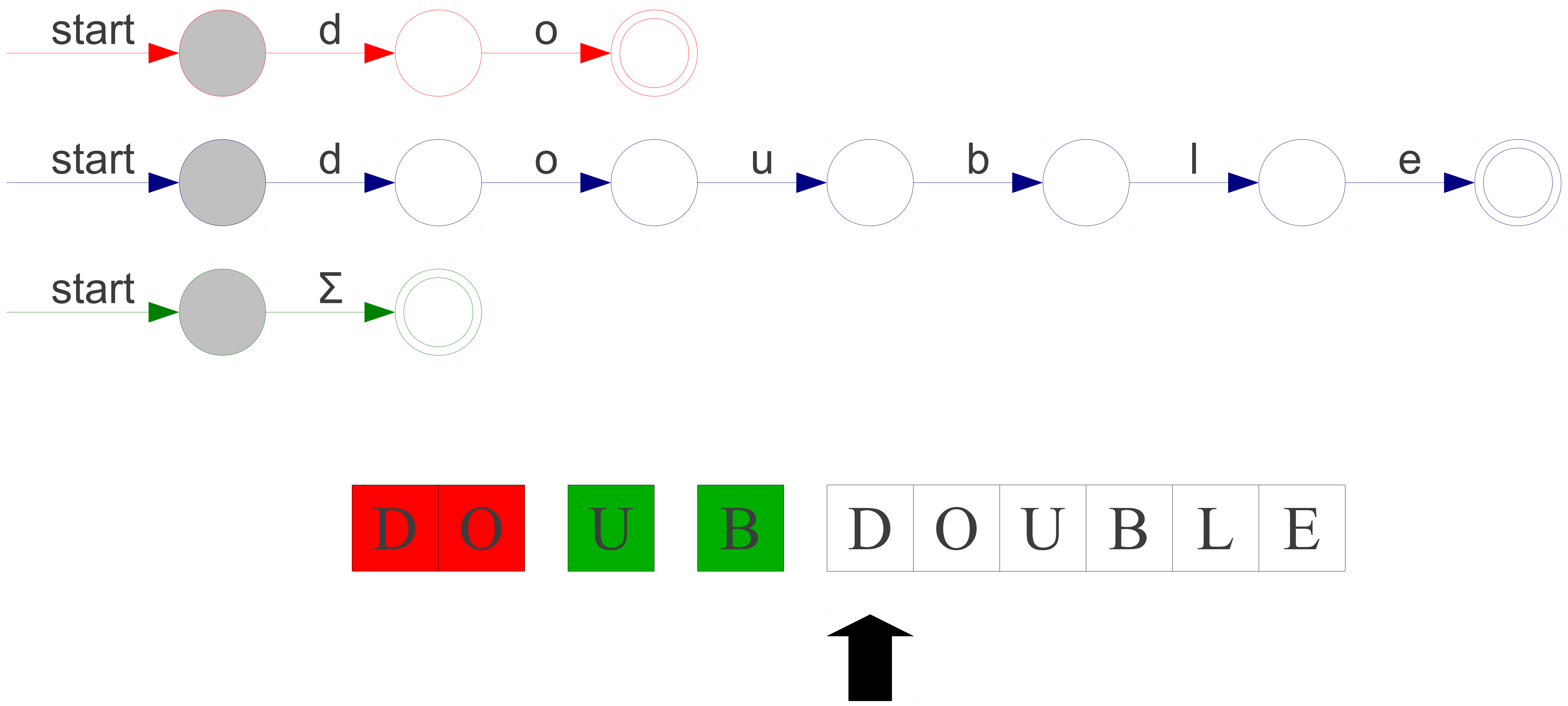
Implementing Maximal Munch

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



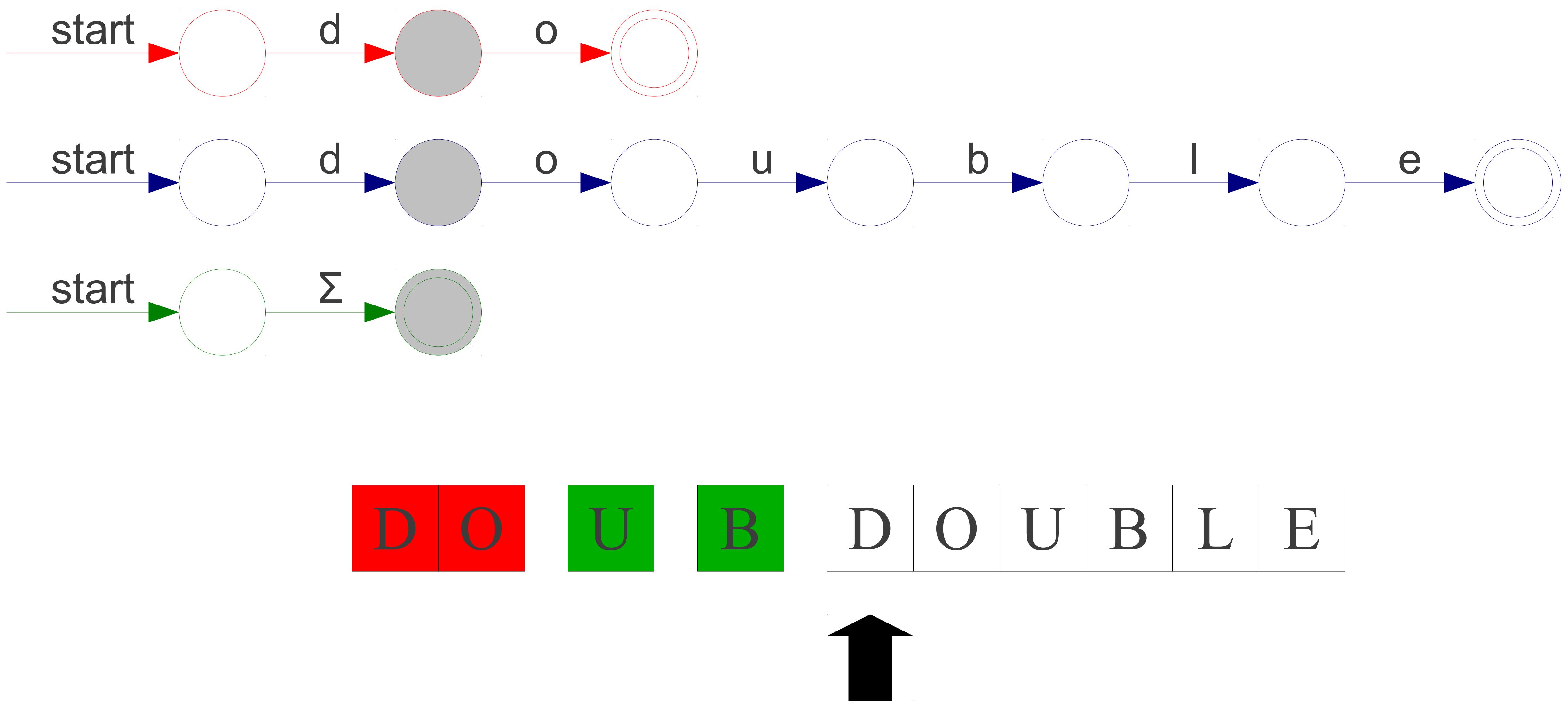
Implementing Maximal Munch

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



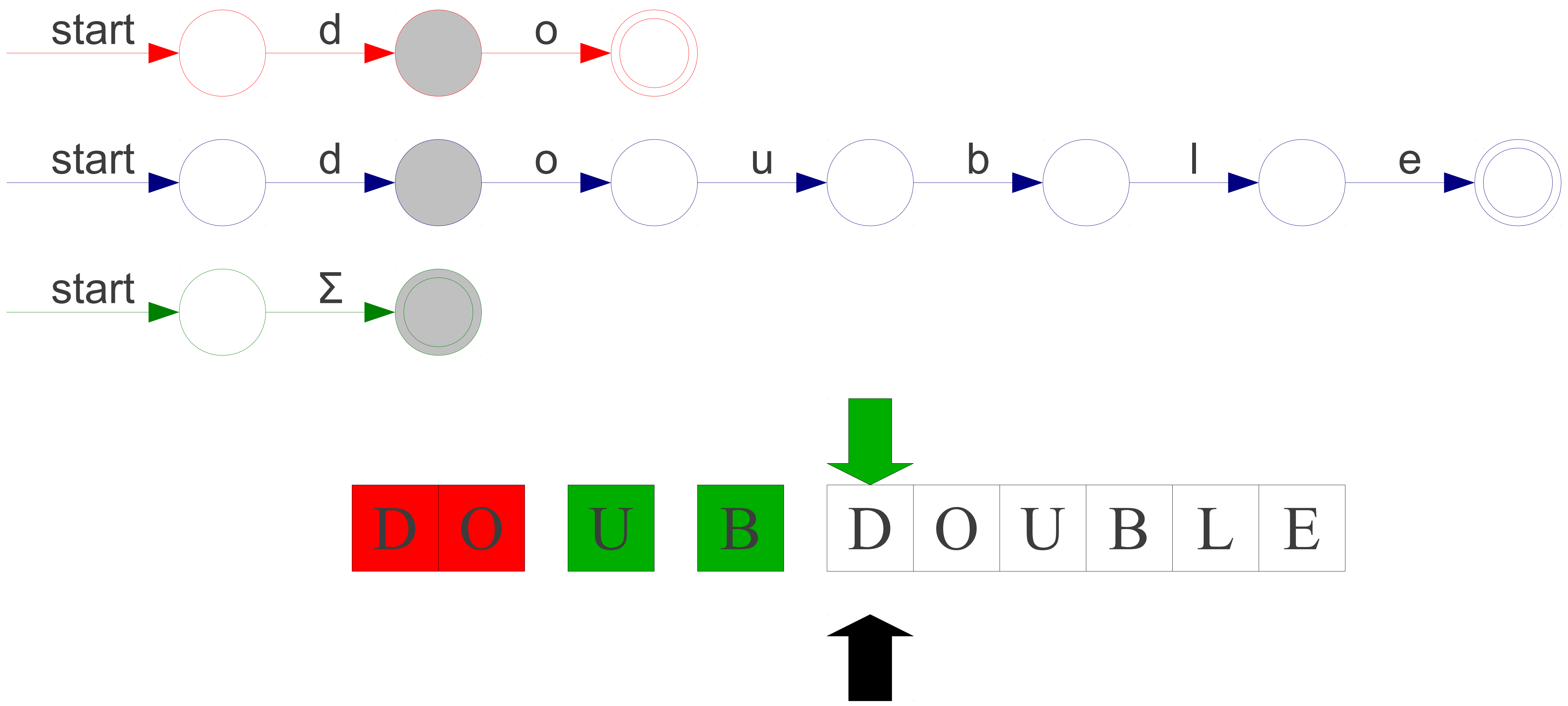
Implementing Maximal Munch

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



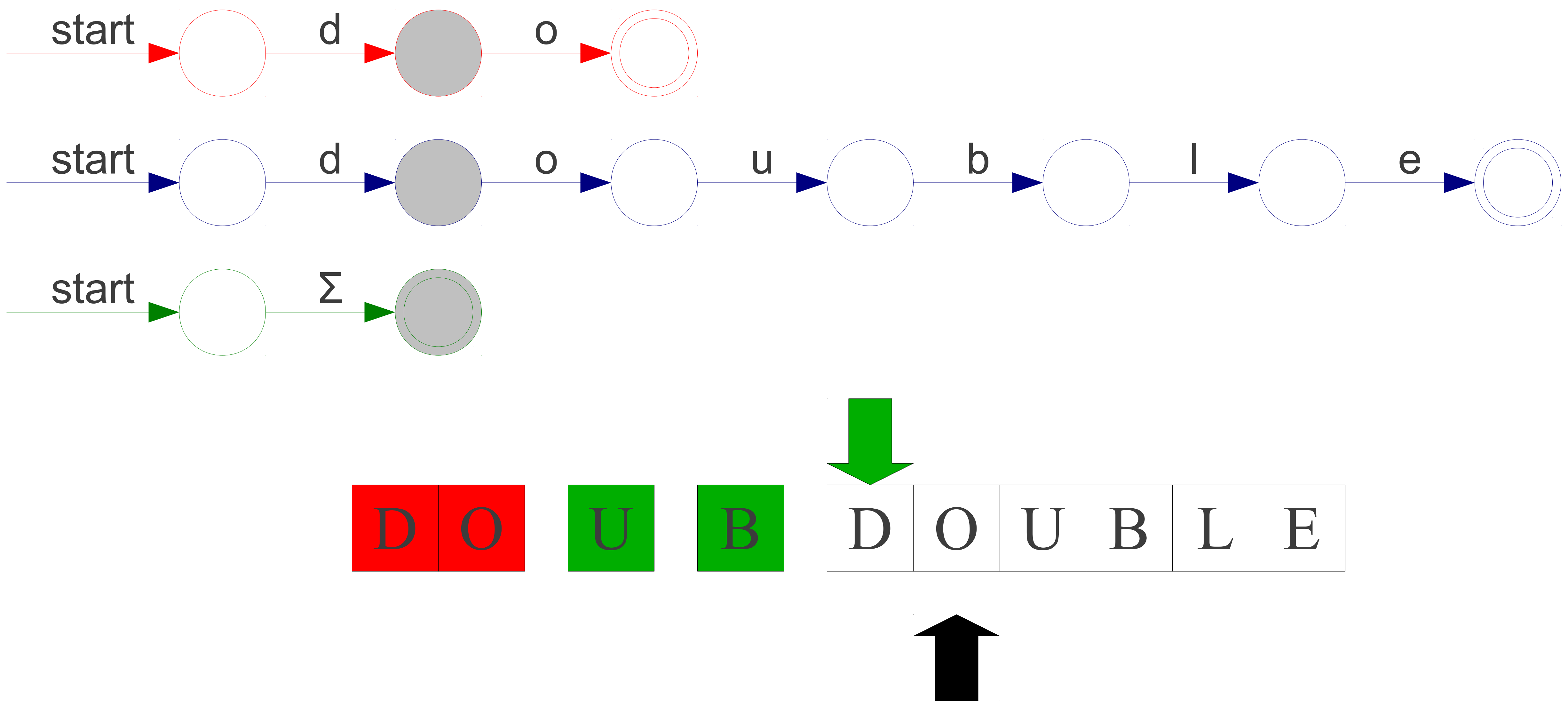
Implementing Maximal Munch

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



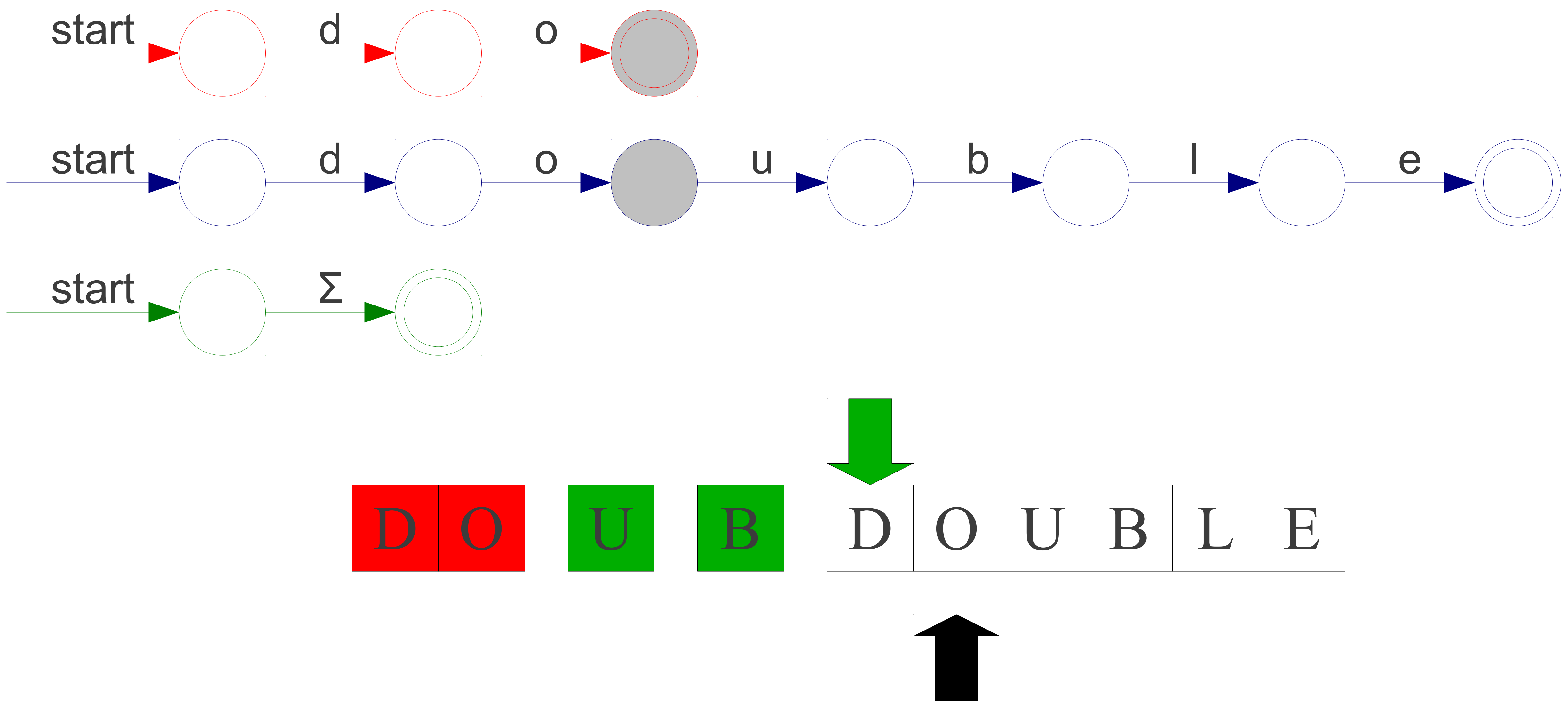
Implementing Maximal Munch

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



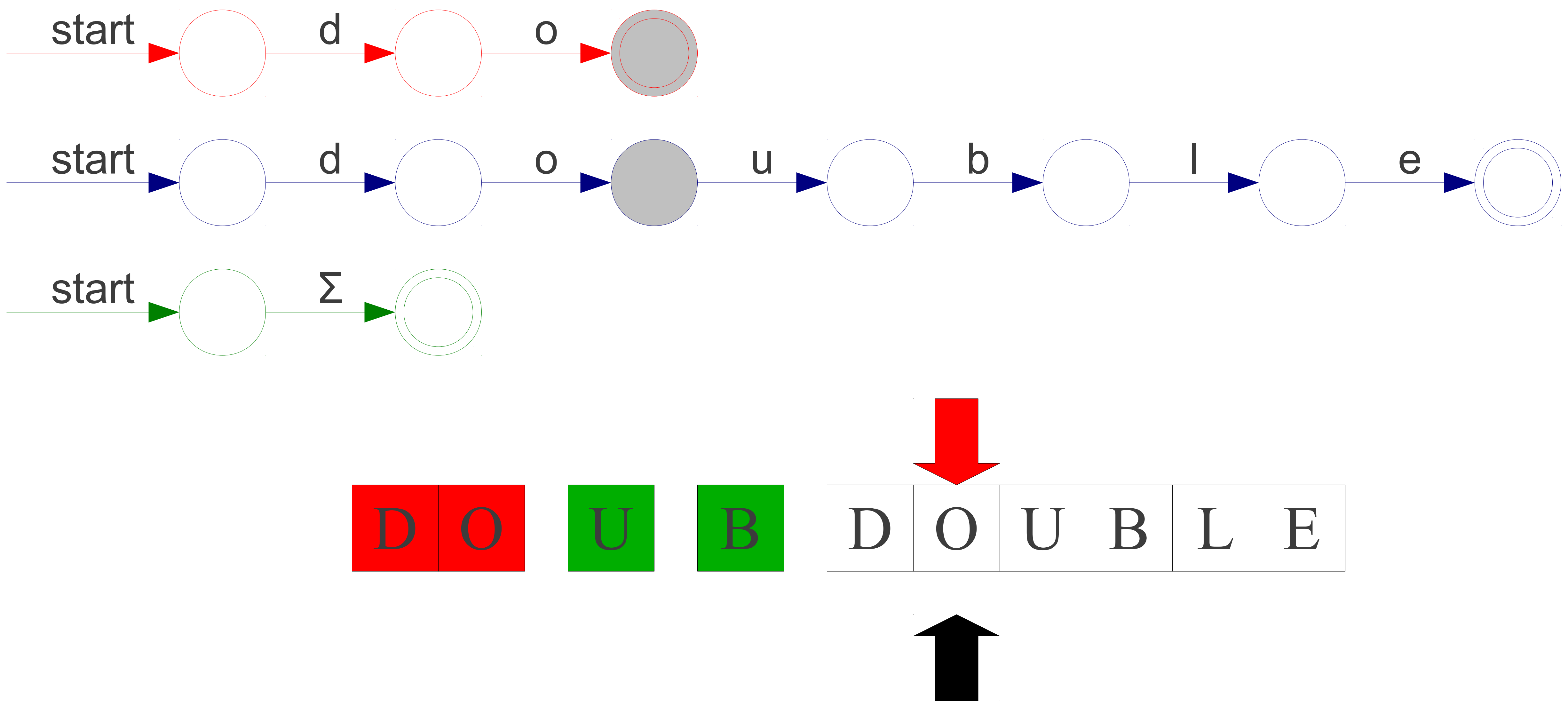
Implementing Maximal Munch

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



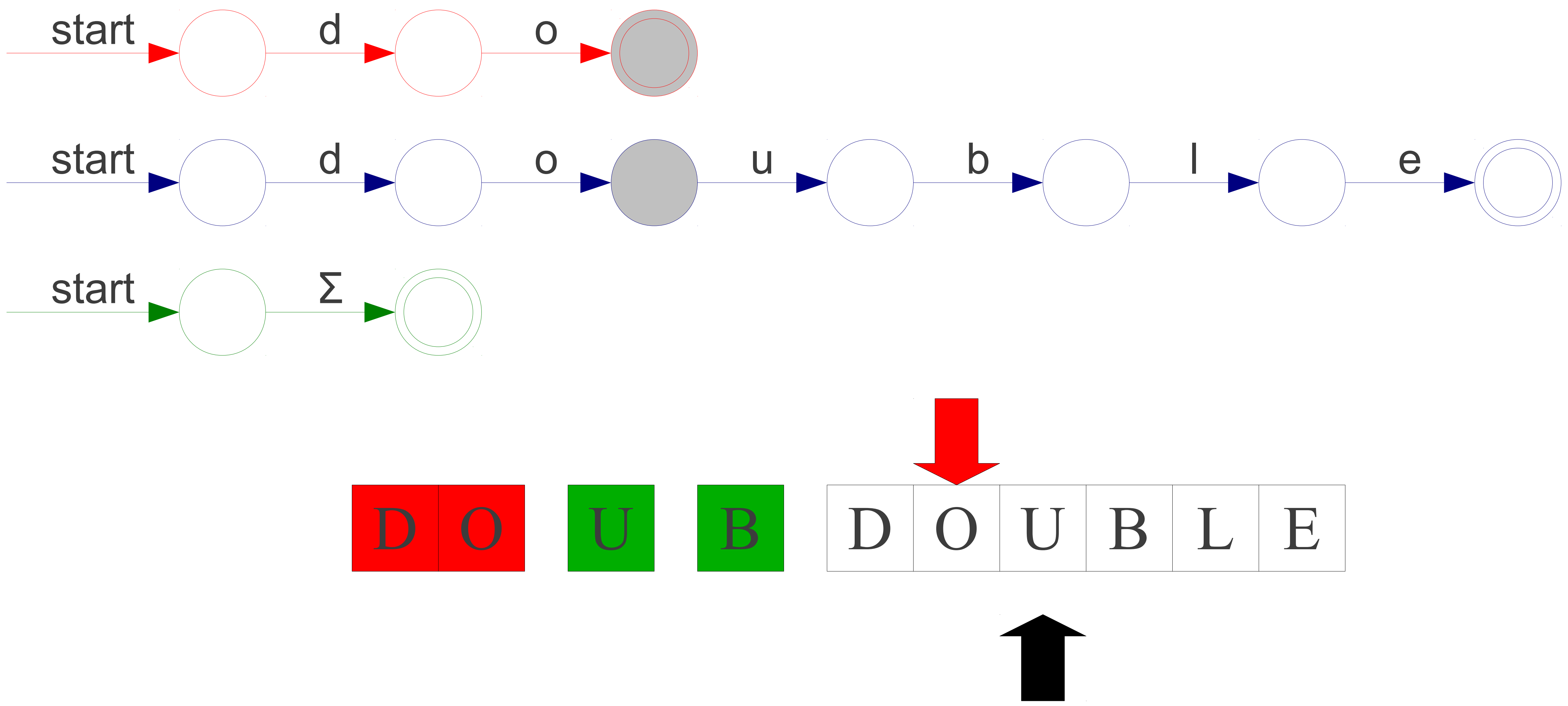
Implementing Maximal Munch

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



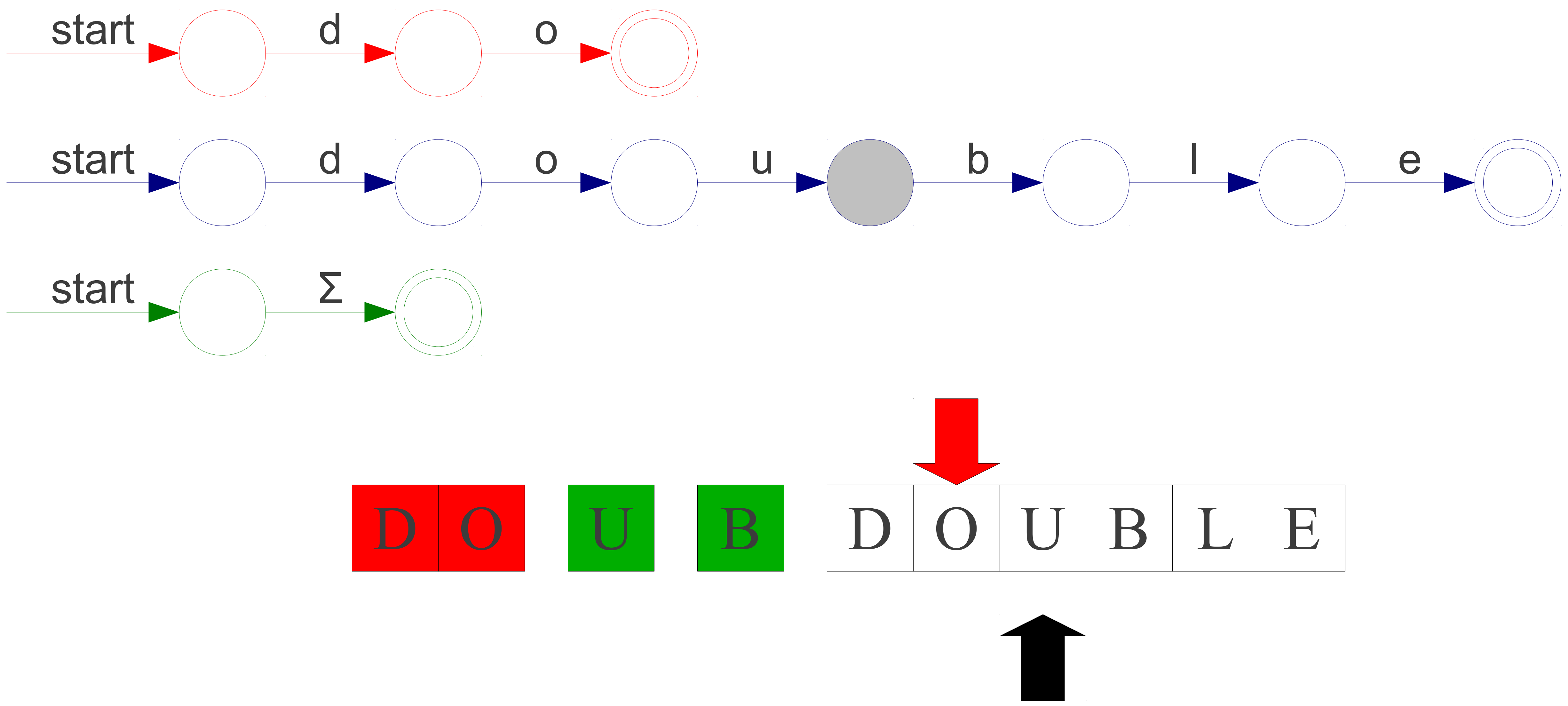
Implementing Maximal Munch

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



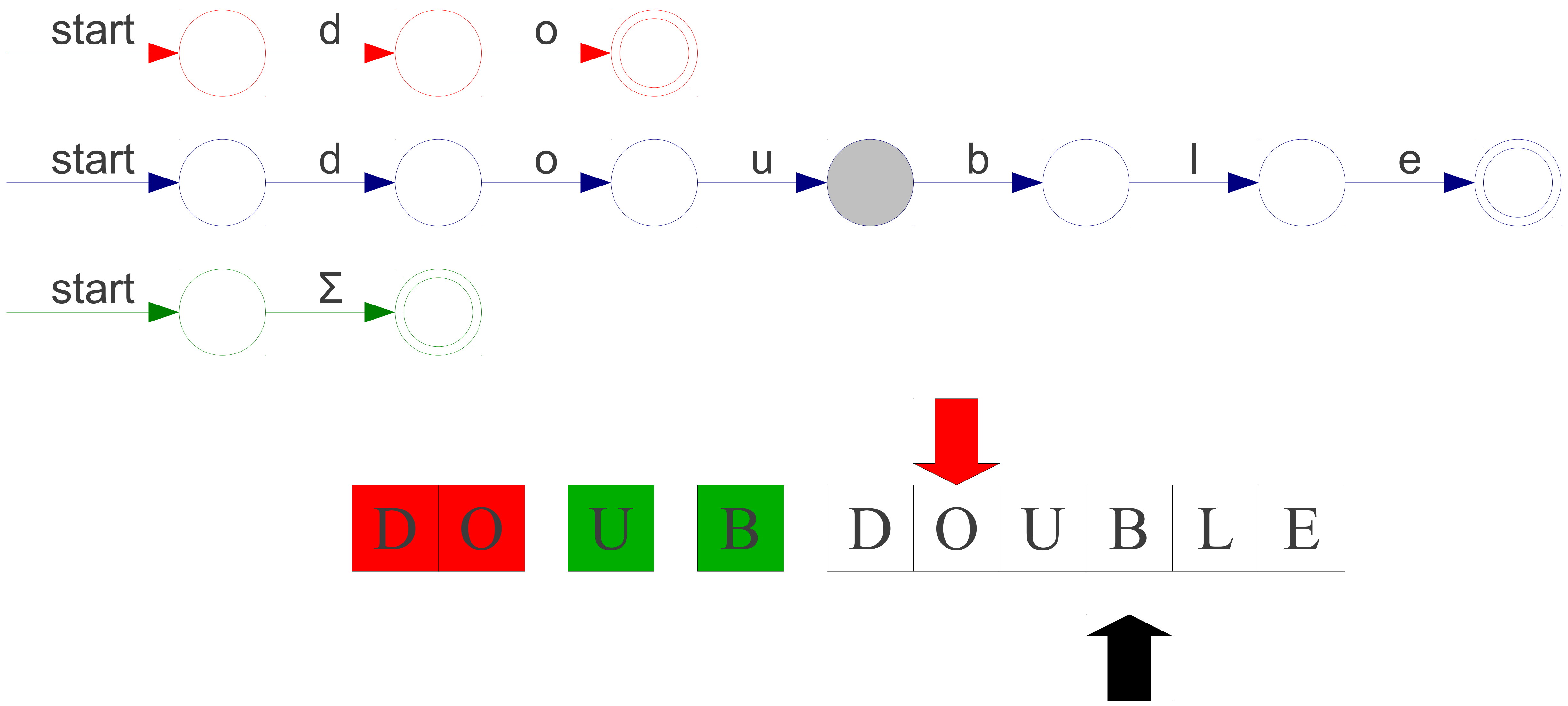
Implementing Maximal Munch

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



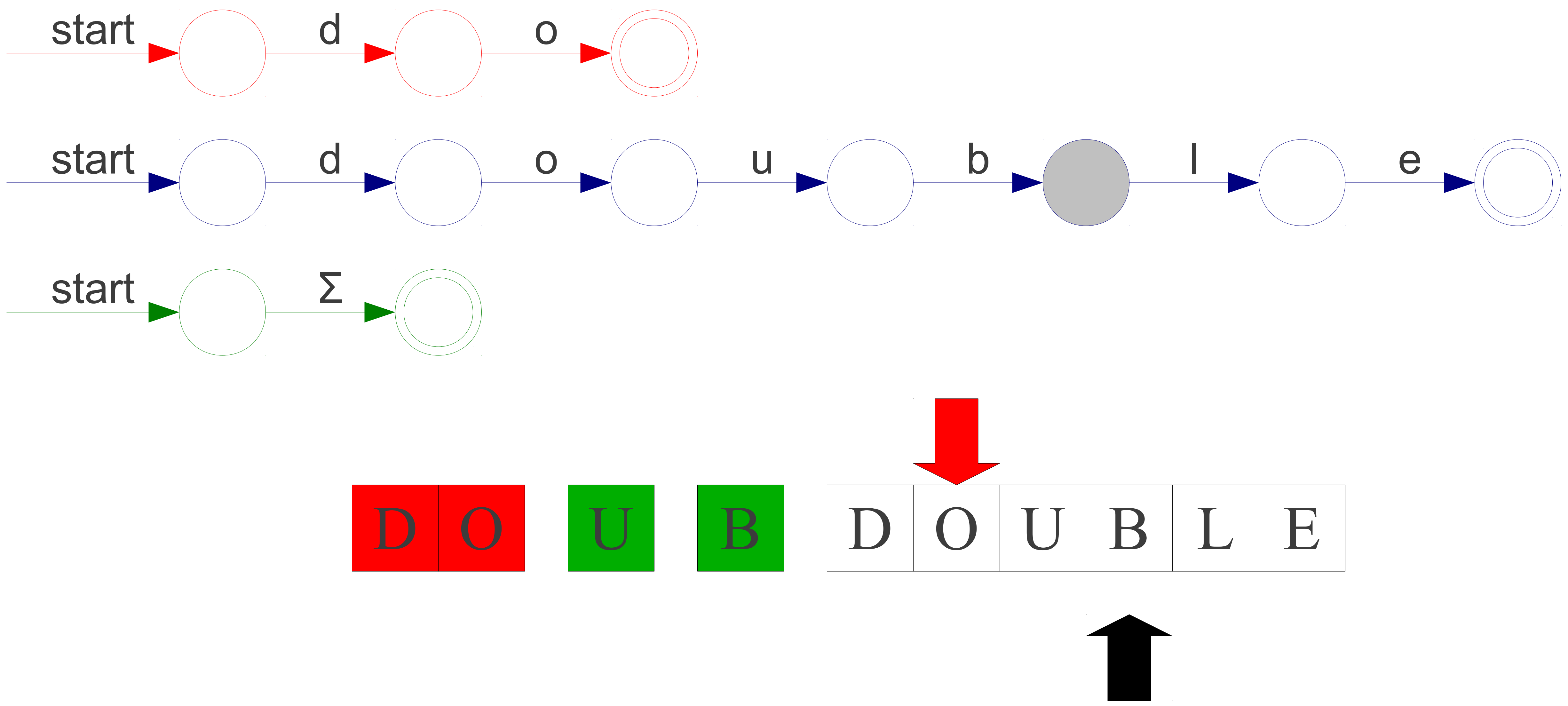
Implementing Maximal Munch

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



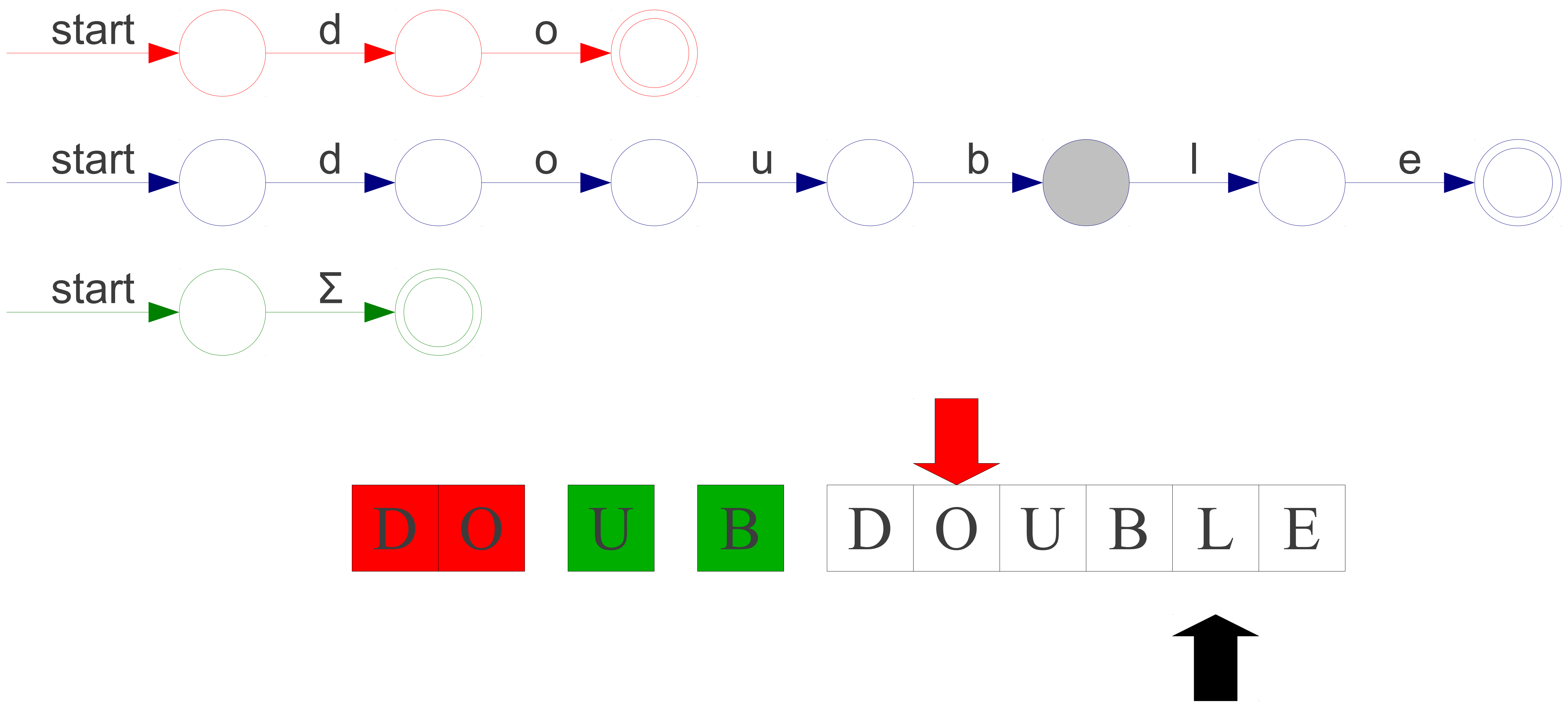
Implementing Maximal Munch

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



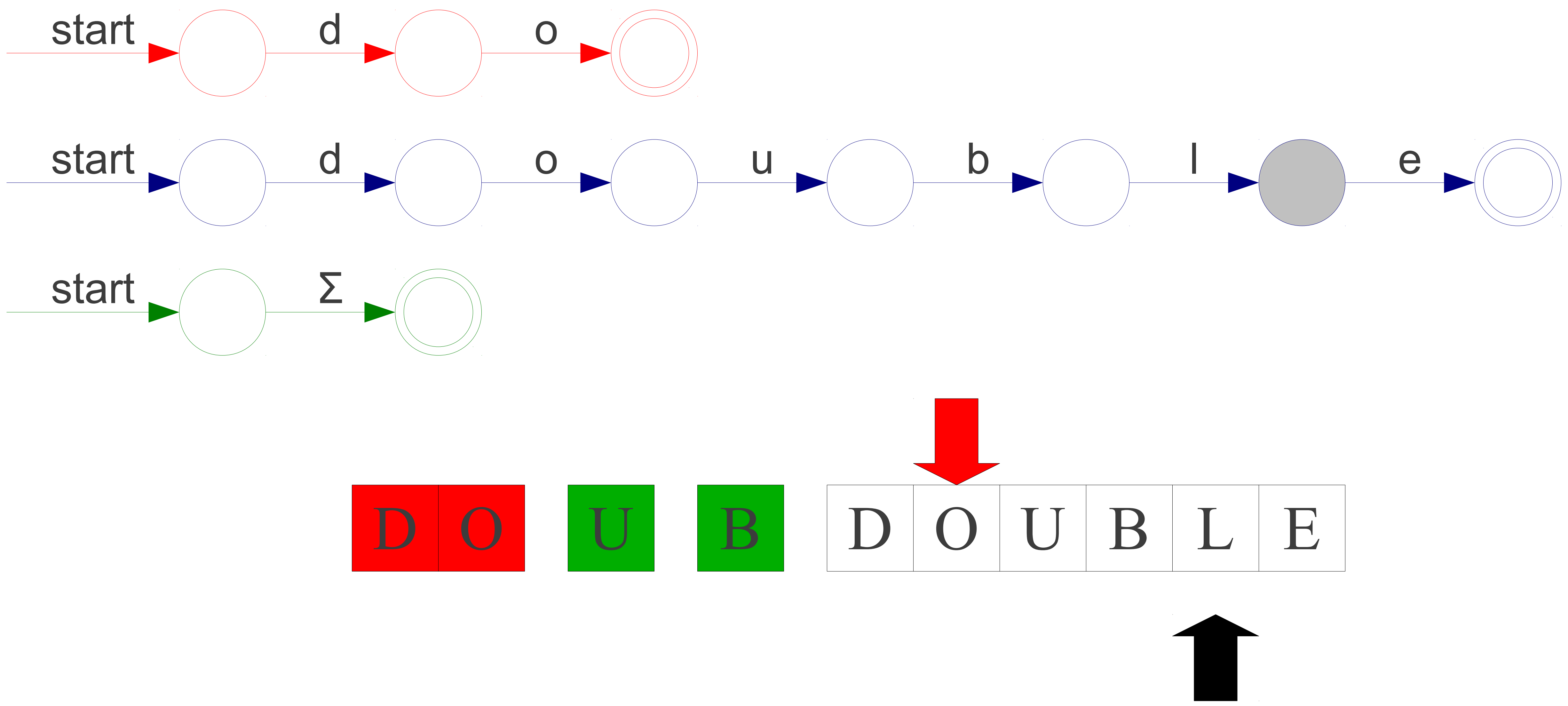
Implementing Maximal Munch

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



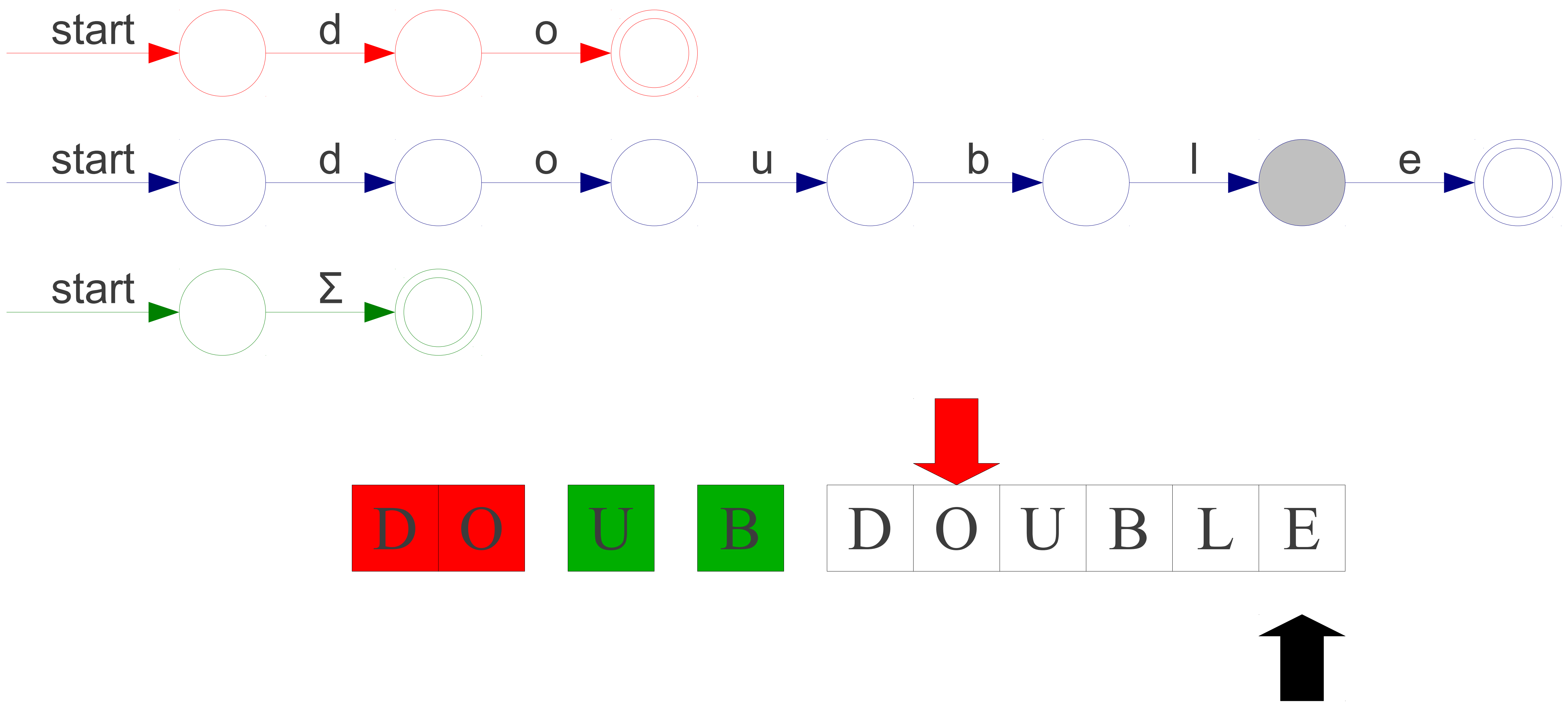
Implementing Maximal Munch

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



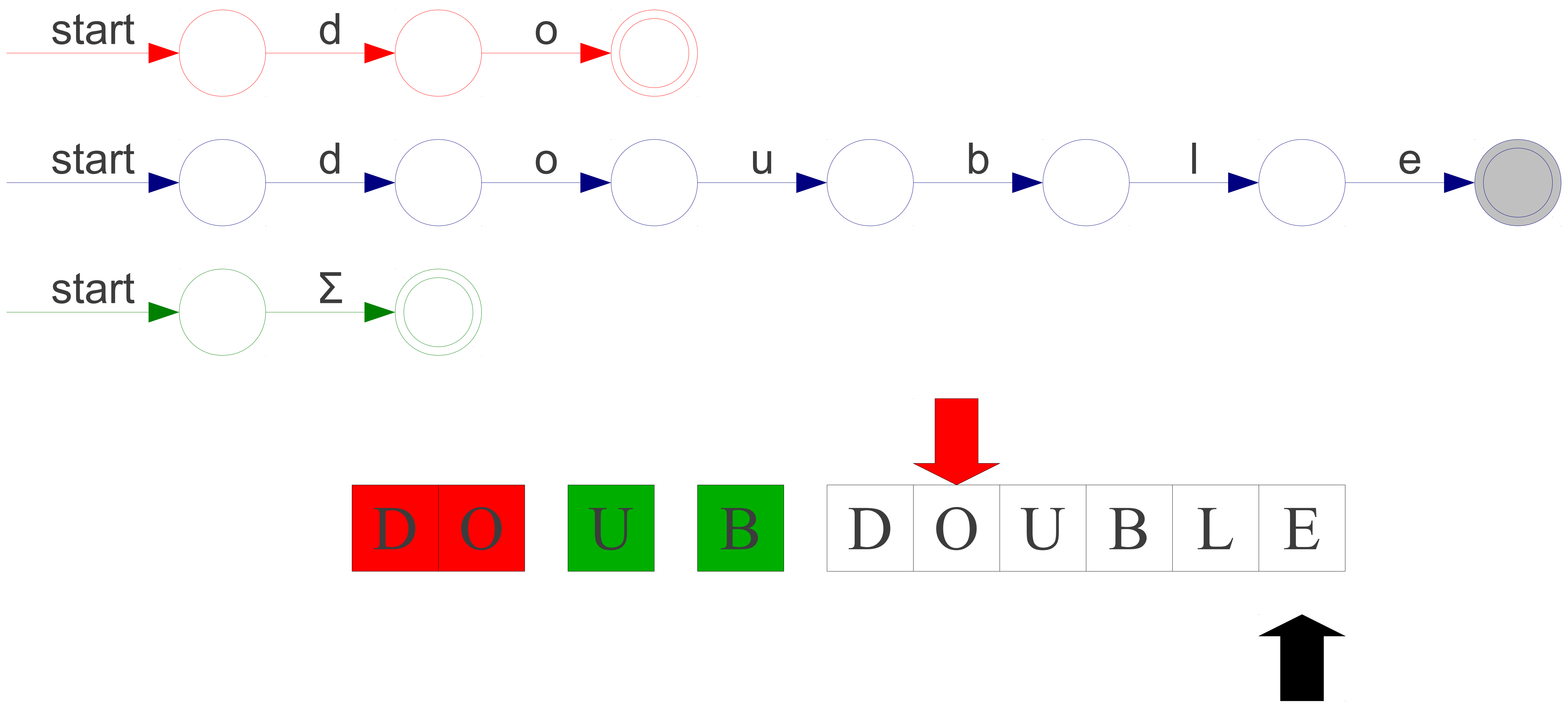
Implementing Maximal Munch

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



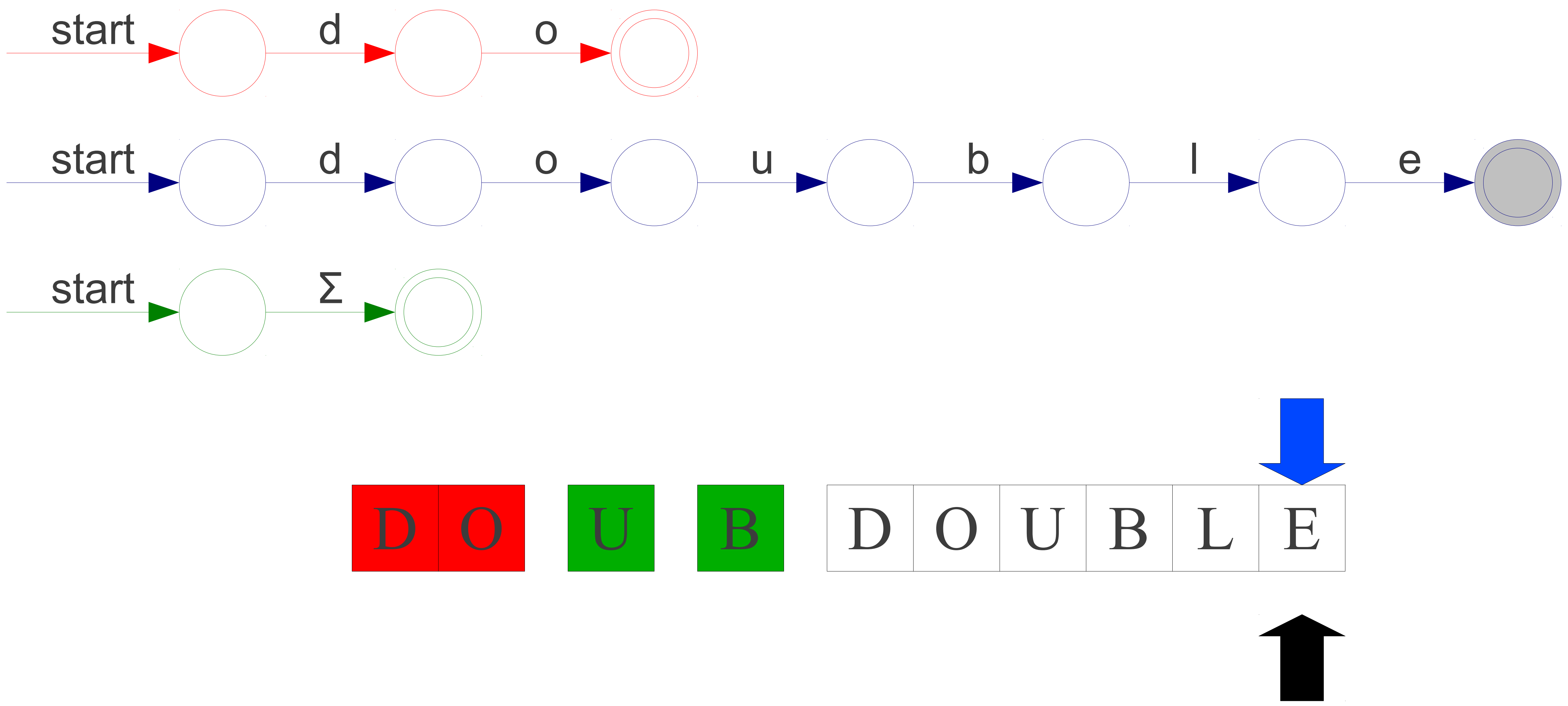
Implementing Maximal Munch

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



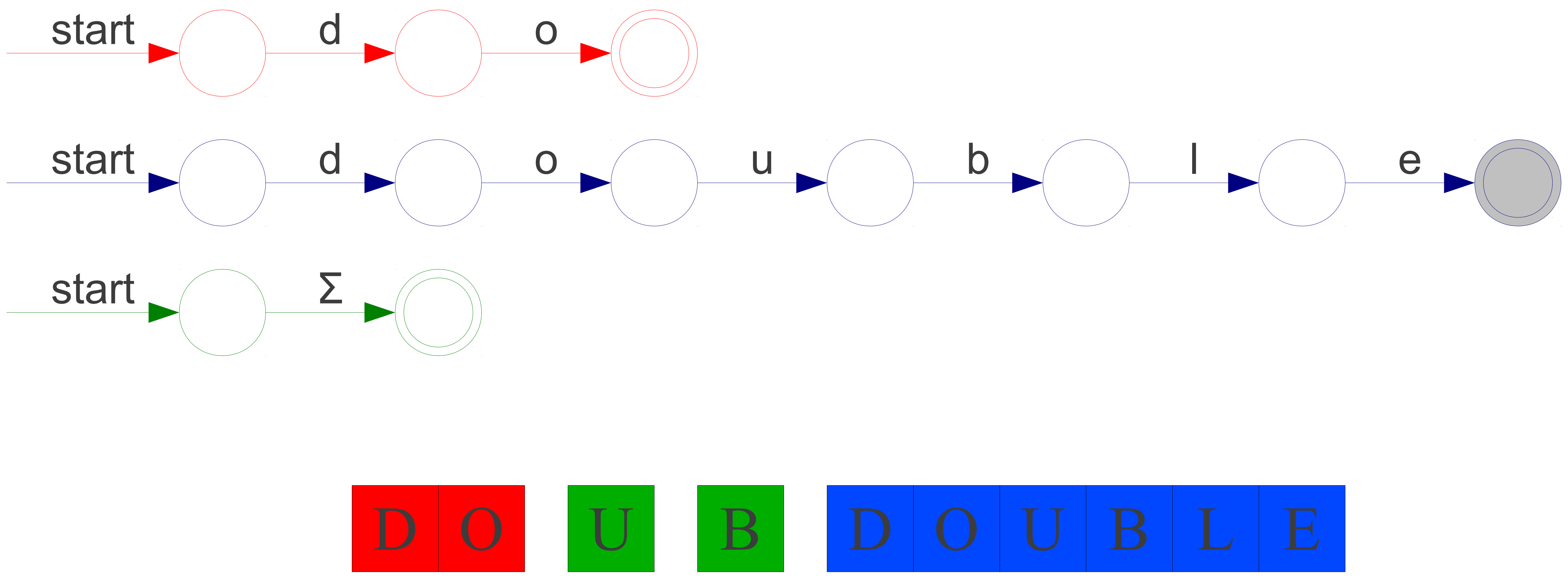
Implementing Maximal Munch

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

disambiguate with an ordering between the choices

Summary

- Lexers scan the input program, grouping substrings into higher level **tokens**
- Lexing is tedious and error-prone to implement manually. Just like assembly code!
- Can use a **lexer generator**, a compiler for a domain-specific language for lexers.
- Use **regular expressions** as syntax, **finite automata** as intermediate representation
- Widely available tools, well known algorithms
  - Caveat: computationally limited, not powerful enough for parsing or semantic analysis. New formalisms next time!