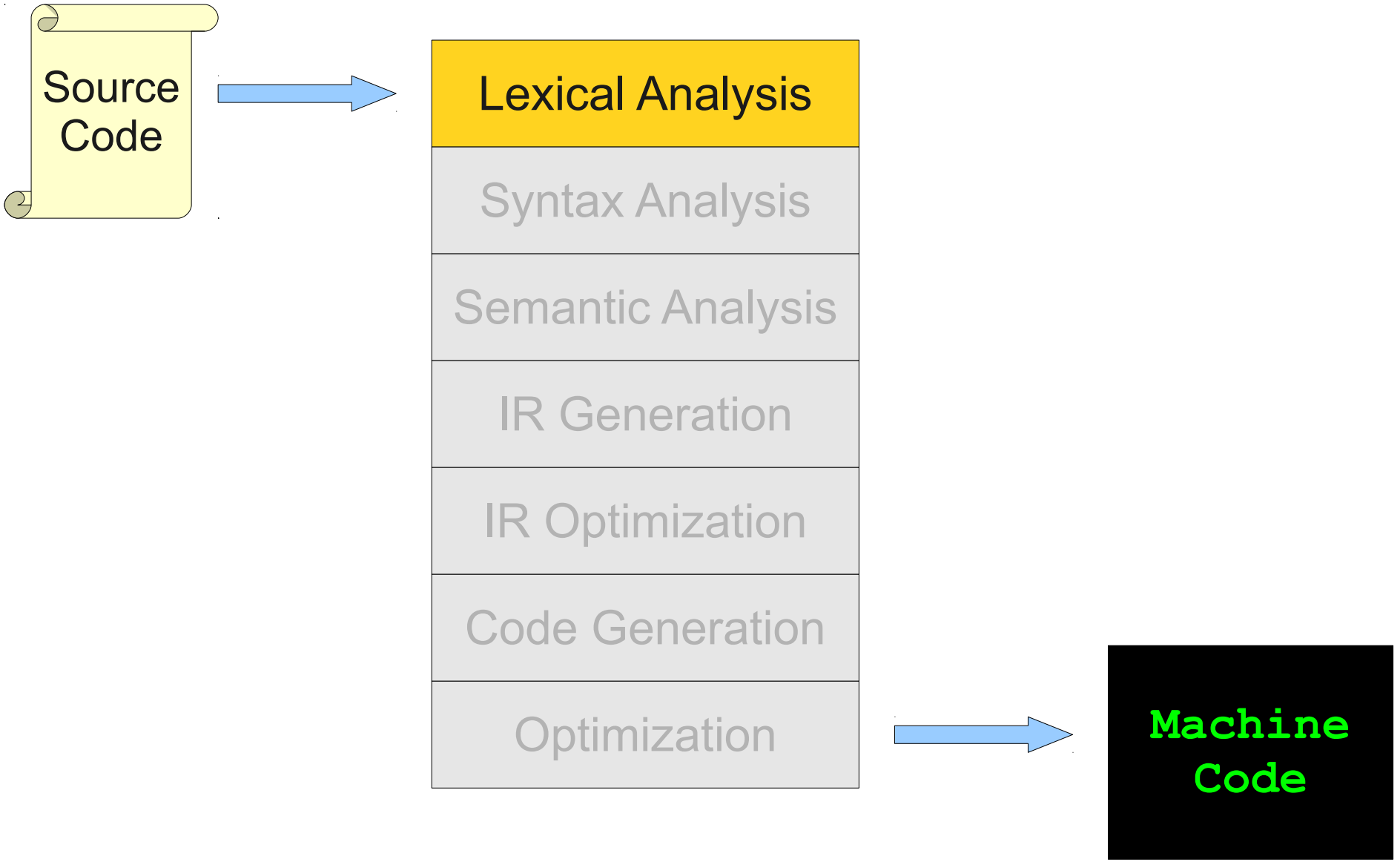
Lexical Analysis

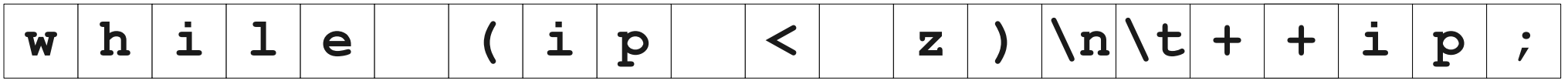
Announcements

- **Programming Assignment 1 Out**
  - Due Friday, July 1 at 11:59 PM.
- Check the website for readings:
  - Decaf Specification
  - Lexical Analysis
  - Intro to **`flex`**
  - Programming Assignment 1

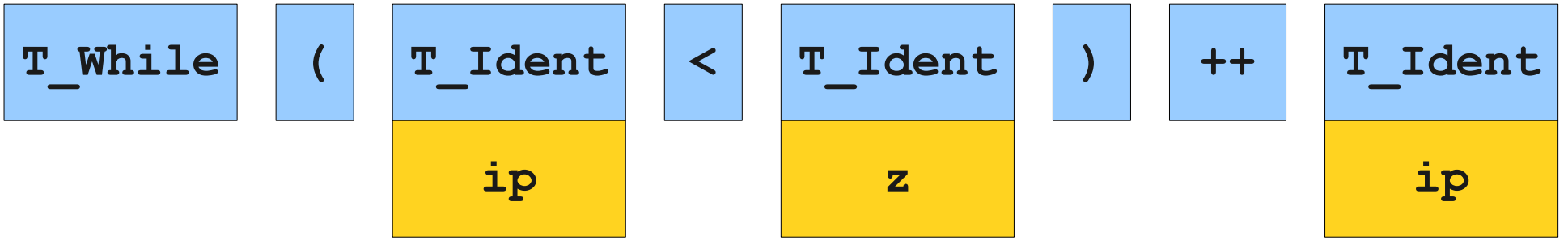
Where We Are


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

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

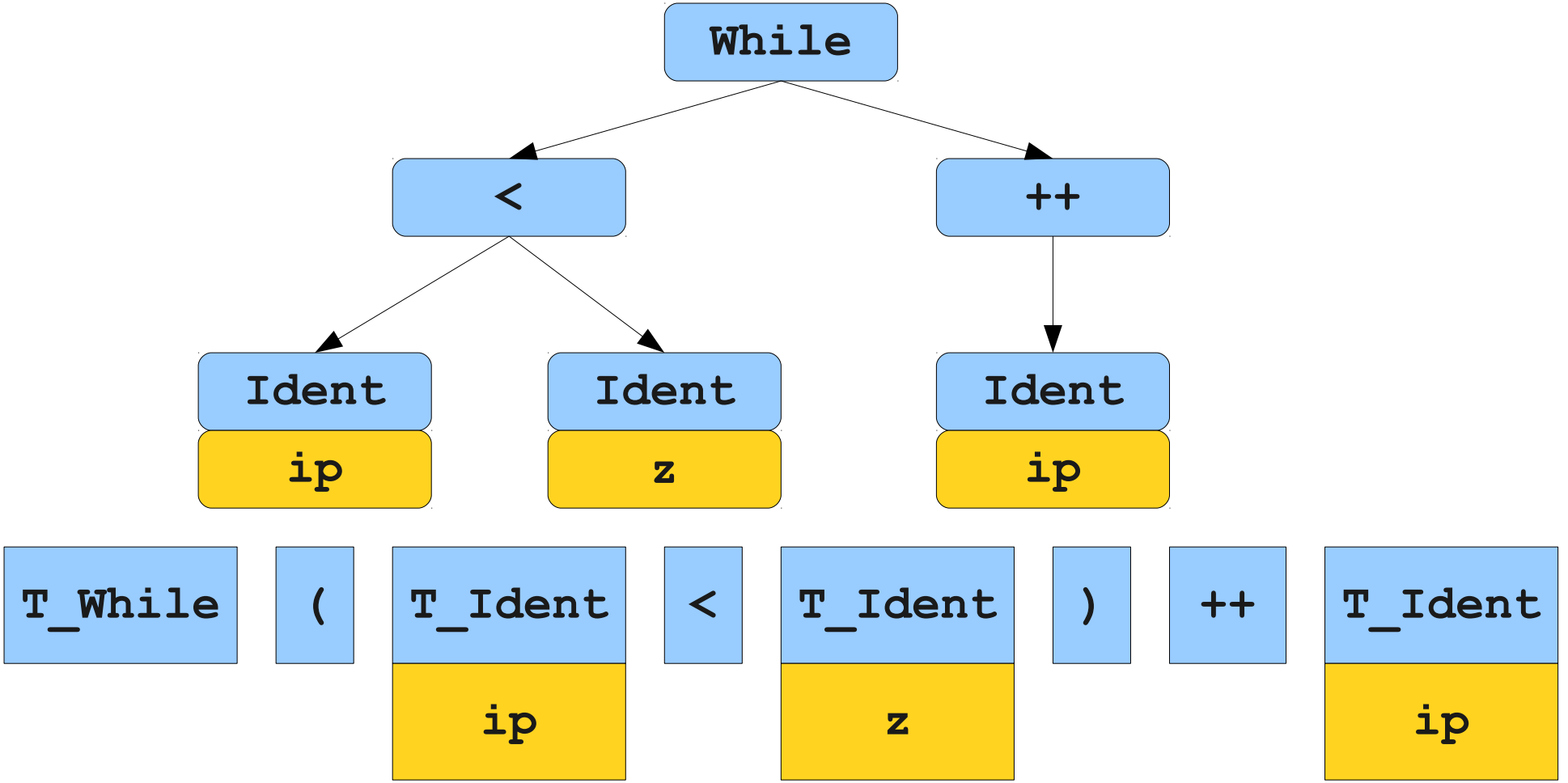



```
while (ip < z)
    ++ip;
```



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

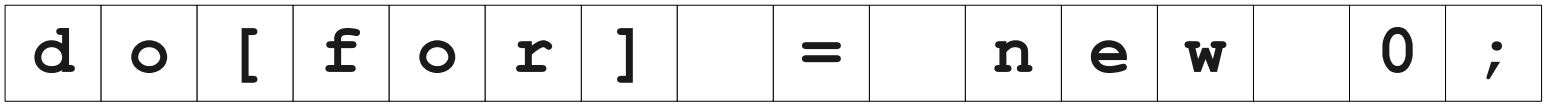
```
while (ip < z)
    ++ip;
```



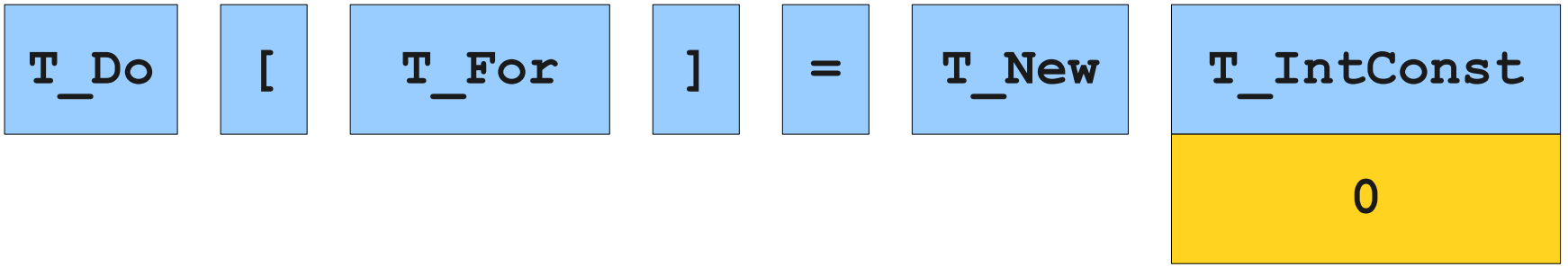
w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

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

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

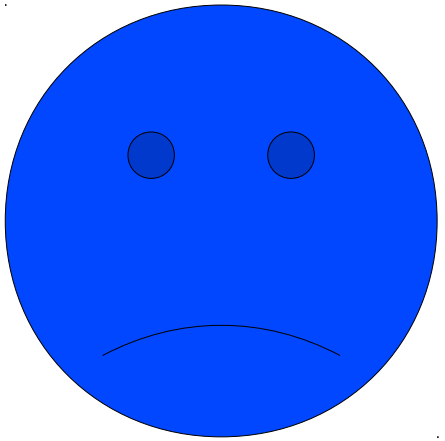



```
do[for] = new 0;
```



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

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



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

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

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

Why do Lexical Analysis?

- Dramatically simplify parsing.
  - Eliminate whitespace.
  - Eliminate comments.
  - Convert data early.
- Separate out logic to read source files.
  - Potentially an issue on multiple platforms.
  - Can optimize reading code independently of parser.
  - An extreme example...

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

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

The program is as follows:

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

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

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

The program is as follows:

```
#include <iostream>

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

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

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

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

The program is as follows:

```
#include <iostream>

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

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

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

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
- Each token is associated with a **lexeme**.
- Each token may have optional **attributes**.
- The token stream will be used in the parser to recover the program structure.

Challenges in Lexical Analysis

- How to partition the program into lexemes?
- How to label each lexeme correctly?

Not-so-Great Moments in Scanning

- FORTRAN: Whitespace is irrelevant

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- FORTRAN: Whitespace is irrelevant

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- FORTRAN: Whitespace is irrelevant

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

- Can be difficult to tell when to partition input.

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- C++: Nested template declarations

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

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

- C++: Types nested in templates

Not-so-Great Moments in Scanning

- C++: Types nested in templates

```
template <typename T>
void MyFunction(T val) {
    T::iterator itr;
}
```

Not-so-Great Moments in Scanning

- C++: Types nested in templates

```
template <typename T>
void MyFunction(T val) {
    typename T::iterator itr;
}
```

Not-so-Great Moments in Scanning

- C++: Types nested in templates

```
template <typename T>
void MyFunction(T val) {
    typename T::iterator itr;
}
```

- Can be hard to label tokens correctly.

Not-so-Great Moments in Scanning

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

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

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

Thanks to Prof. Alex Aiken

Not-so-Great Moments in Scanning

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

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

- Can be difficult to determine how to label lexemes.

Thanks to Prof. Alex Aiken

Defining a Lexical Analysis

- Define a set of tokens.
- Define the set of lexemes associated with each token.
- Define an algorithm for resolving conflicts that arise in the lexemes.

Choosing Tokens

What Tokens are Useful Here?

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

What Tokens are Useful Here?

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

T_For	**{**
T_IntConstant	**}**
T_Int	**<<**
T_Identifier	**;**
=	**<**
(	**[**
)	**]**
++	

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Discard irrelevant information (whitespace, comments)

Defining Sets of Strings

String Terminology

- An **alphabet** Σ is a set of characters.
- A **string** over Σ is a finite sequence of elements from Σ.
- Example:
  - Σ = { ☺ , ☼ }
  - Valid strings include ☺ , ☺ ☺ , ☺☼, ☺ ☺☼, etc.
- The **empty string** of no characters is denoted ε.

Languages

- A **language** is a set of strings.
- Example: Even numbers
  - Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - L = {0, 2, 4, 6, 8, 10, 12, 14, … }
- Example: C variable names
  - Σ = ASCII characters
  - L = {a, b, c, …, A, B, C, …, _, aa, ab, … }
- Example: L = {static}

Regular Languages

- A subset of all languages that can be defined by **regular expressions**.
- Small subset of all languages, but surprisingly powerful in practice:
  - Capture a useful class of languages.
  - Can be implemented efficiently (more on that later).

Regular Expressions

- Any character is a regular expression matching itself.
- ε is a regular expression matching the empty string.
- If R_1 and R_2 are regular expressions, then
  - R_1R_2 is a regular expression matching the **concatenation** of the languages.
  - $R_1 \mid R_2$ is a regular expression matching the **disjunction** of the languages.
  - R_1^* is a regular expression matching the **Kleene closure** of the language.
  - (R) is a regular expression matching R.

Regular Expressions In Action

- Example: Even Numbers

  (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

Regular Expressions In Action

- Example: Even Numbers

  (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- Notational simplification: Definitions

  Sign = + | -

  OptSign = Sign | ε

  Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  EvenDigit = 0 | 2 | 4 | 6 | 8

  EvenNumber = OptSign Digit* EvenDigit

Regular Expressions In Action

- Example: Even Numbers

  (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- Notational simplification: Multi-Way Disjunction

  Sign = + | -

  OptSign = Sign | ε

  Digit = **[0123456789]**

  EvenDigit = **[02468]**

  EvenNumber = OptSign Digit* EvenDigit

Regular Expressions In Action

- Example: Even Numbers

  (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- Notational simplification: Ranges

  Sign = + | -

  OptSign = Sign | ε

  Digit = **[0-9]**

  EvenDigit = [02468]

  EvenNumber = OptSign Digit* EvenDigit

Regular Expressions In Action

- Example: Even Numbers

  (+|-|ε) (0|1|2|3|4|5|6|7|8|9)* (0|2|4|6|8)

- Notational simplification: Zero-or-One

  Sign = + | -

  OptSign = Sign**?**

  Digit = [0-9]

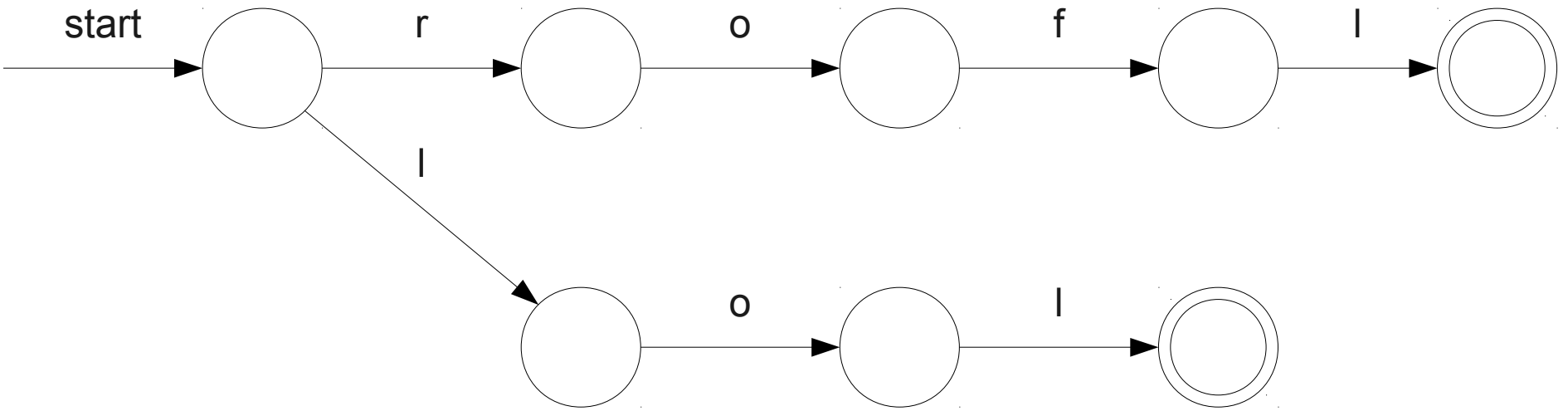
  EvenDigit = [02468]

  EvenNumber = OptSign Digit* EvenDigit

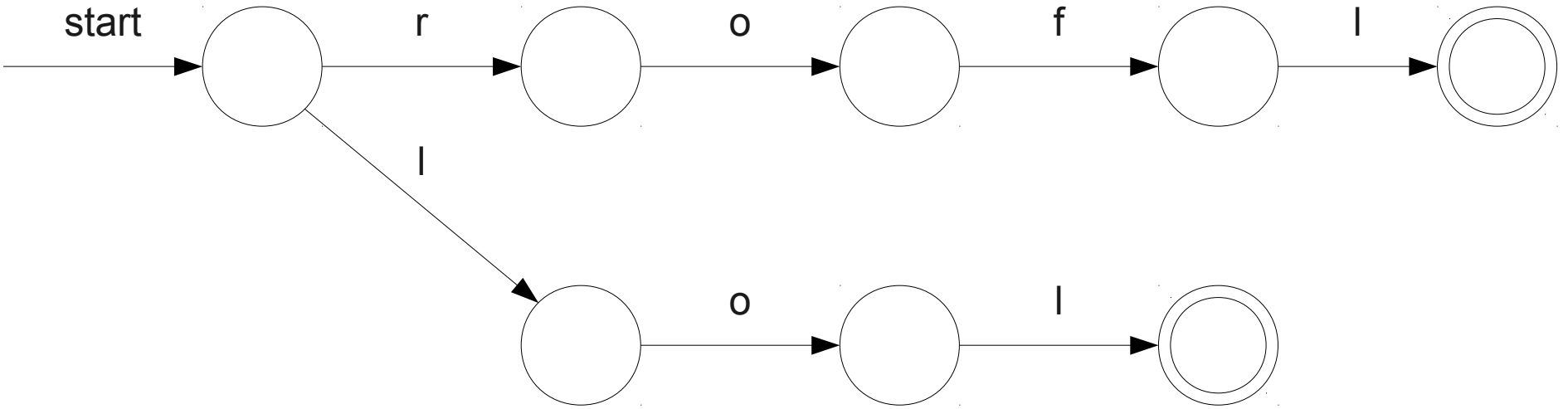
Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
- There are two kinds of finite automata:
  - **NFA**s (**nondeterministic** finite automata), which we'll see in a second, and
  - **DFA**s (**deterministic** finite automata), which we'll see later.
- Automata are best explained by example...

A Simple NFA

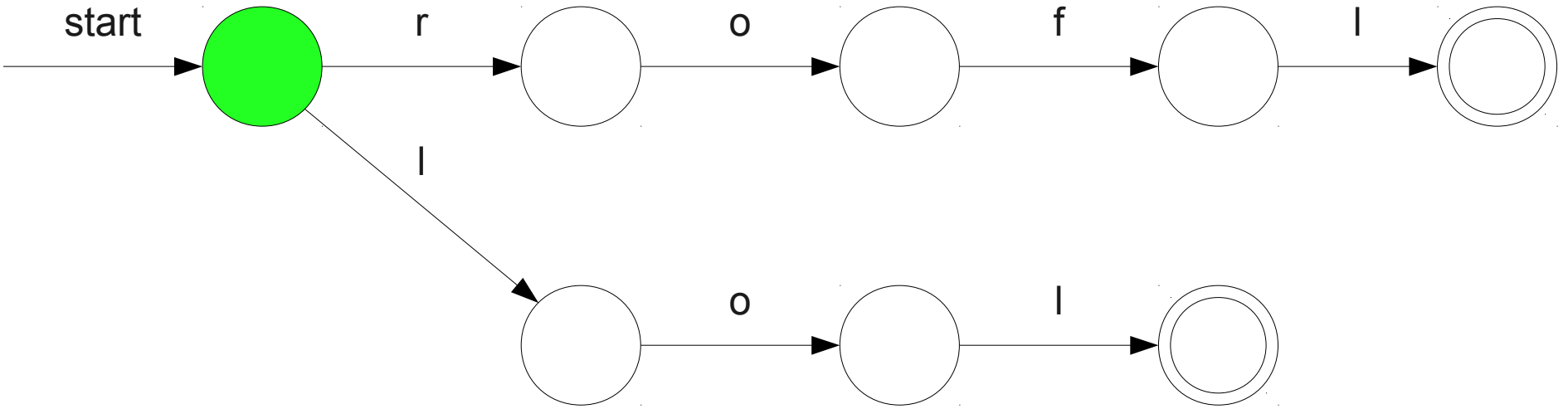


A Simple NFA



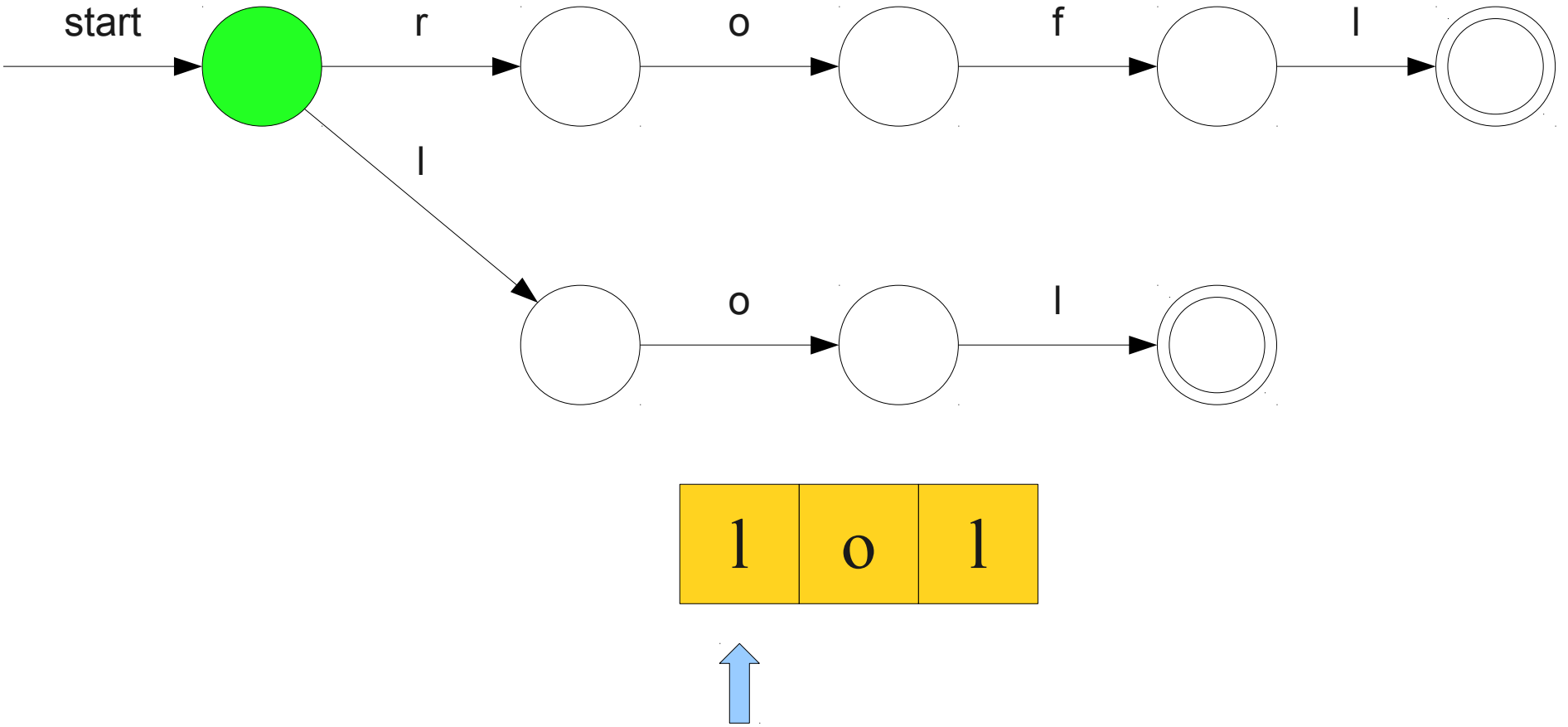
l	o	l

A Simple NFA

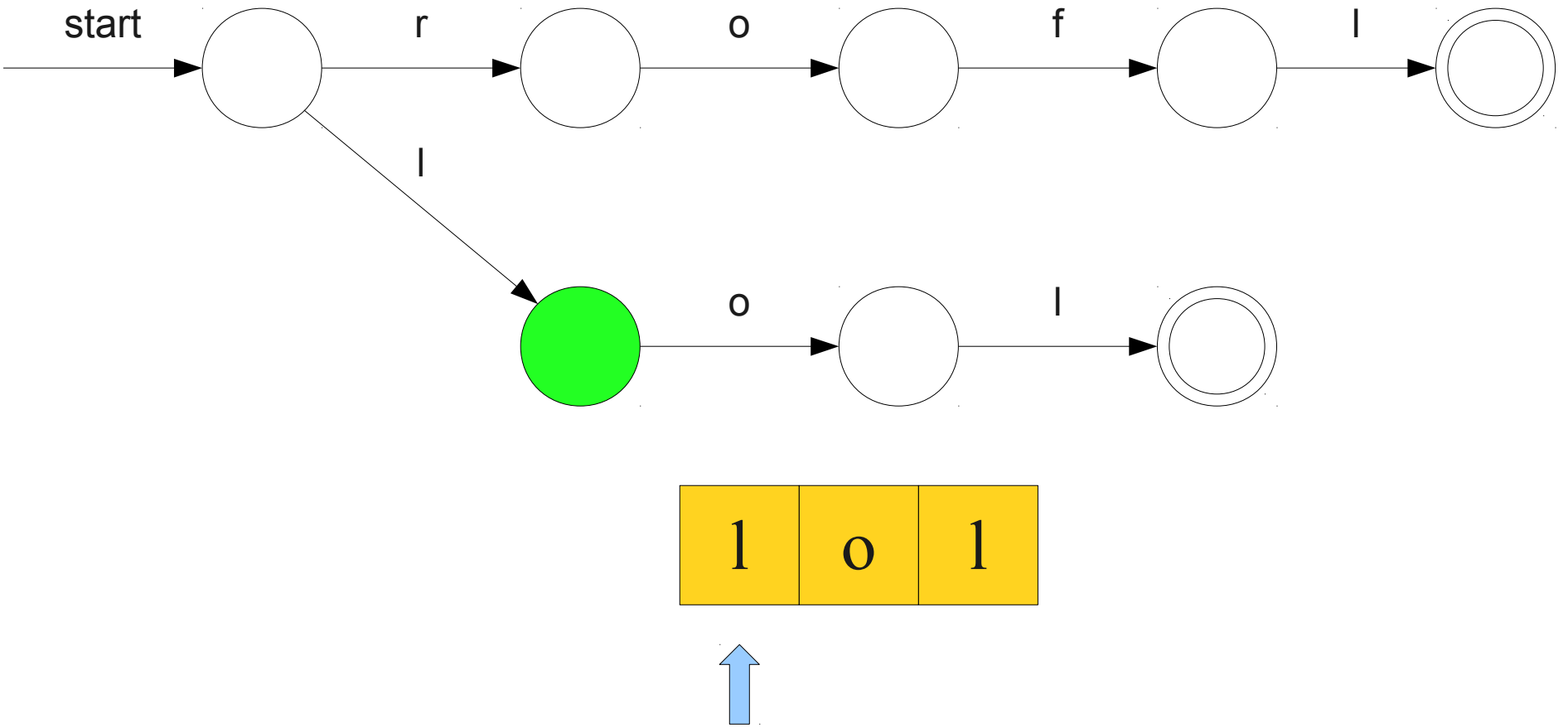


l	o	l

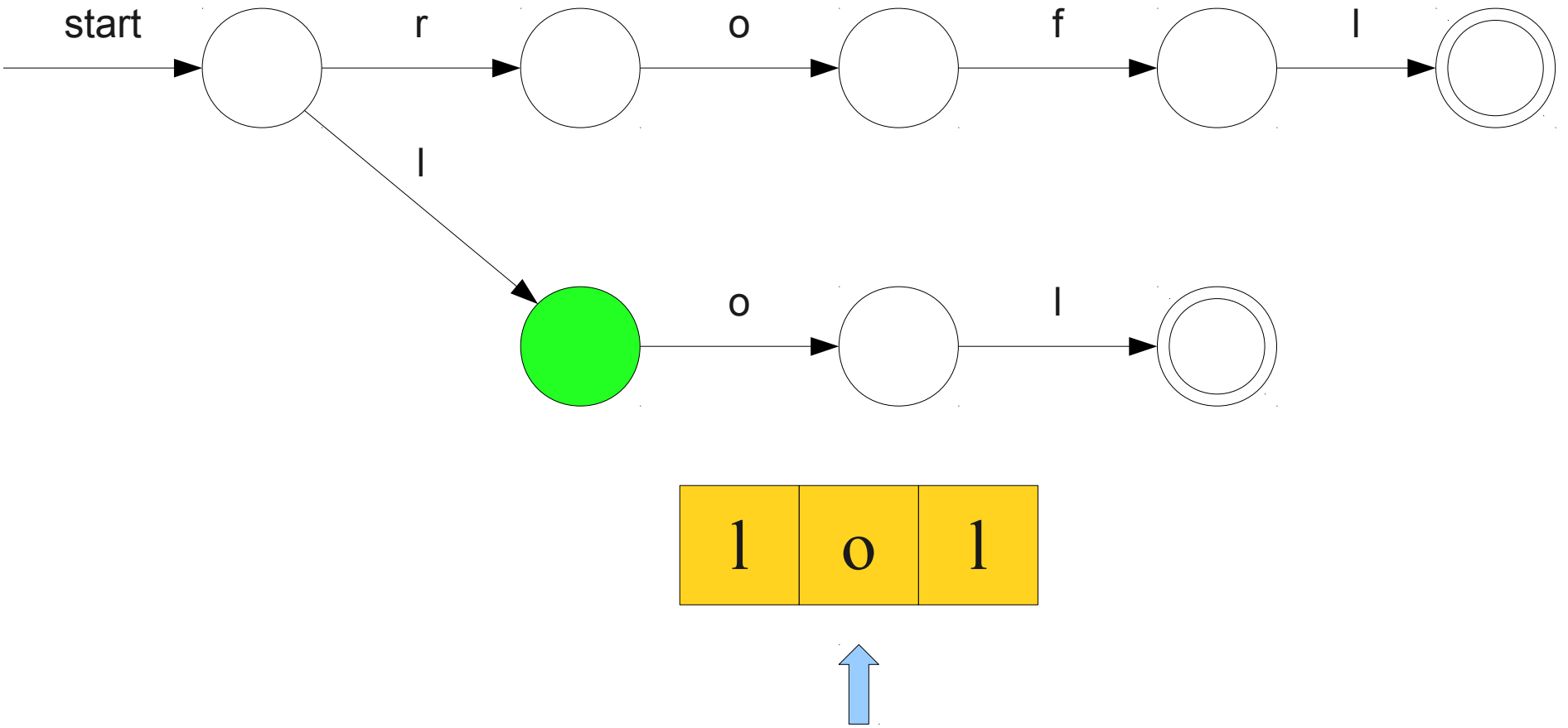
A Simple NFA



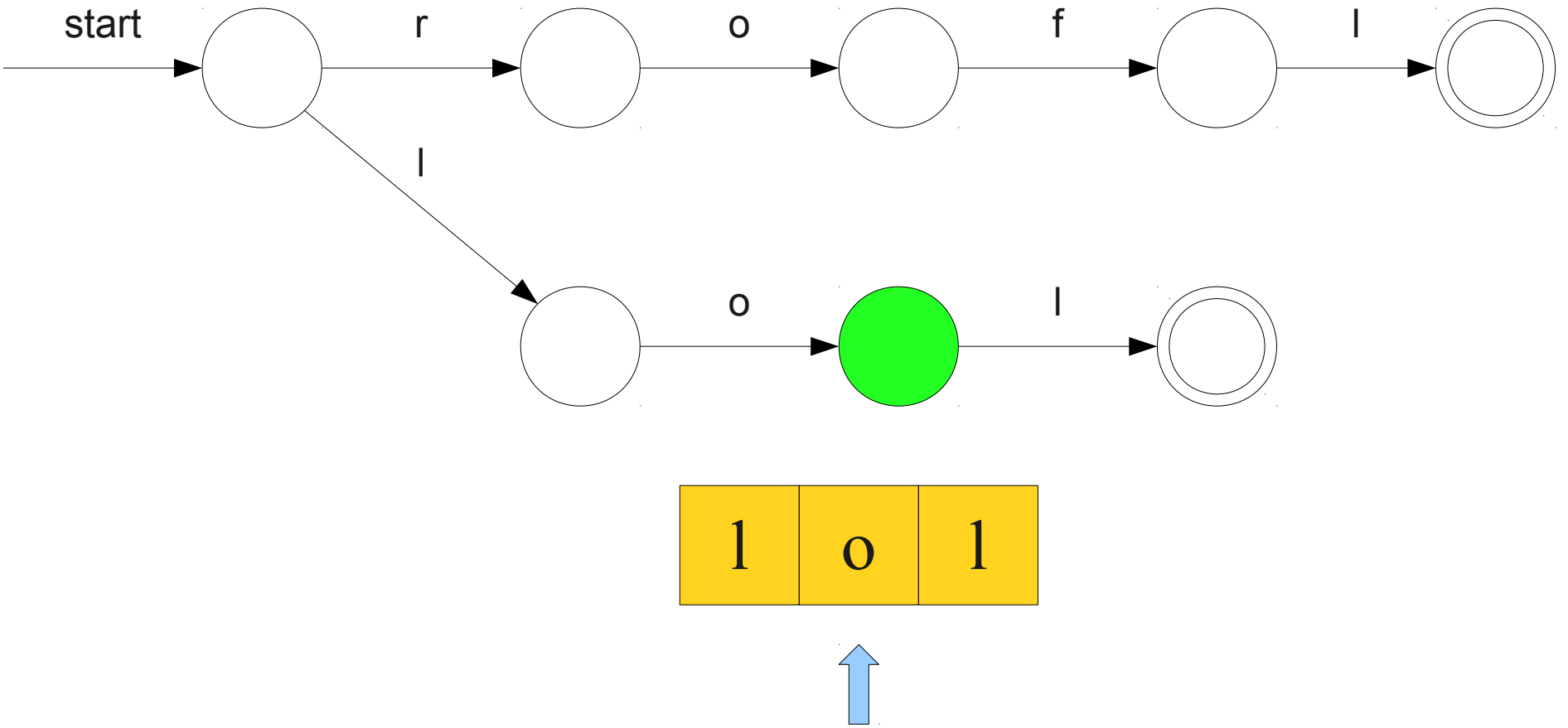
A Simple NFA



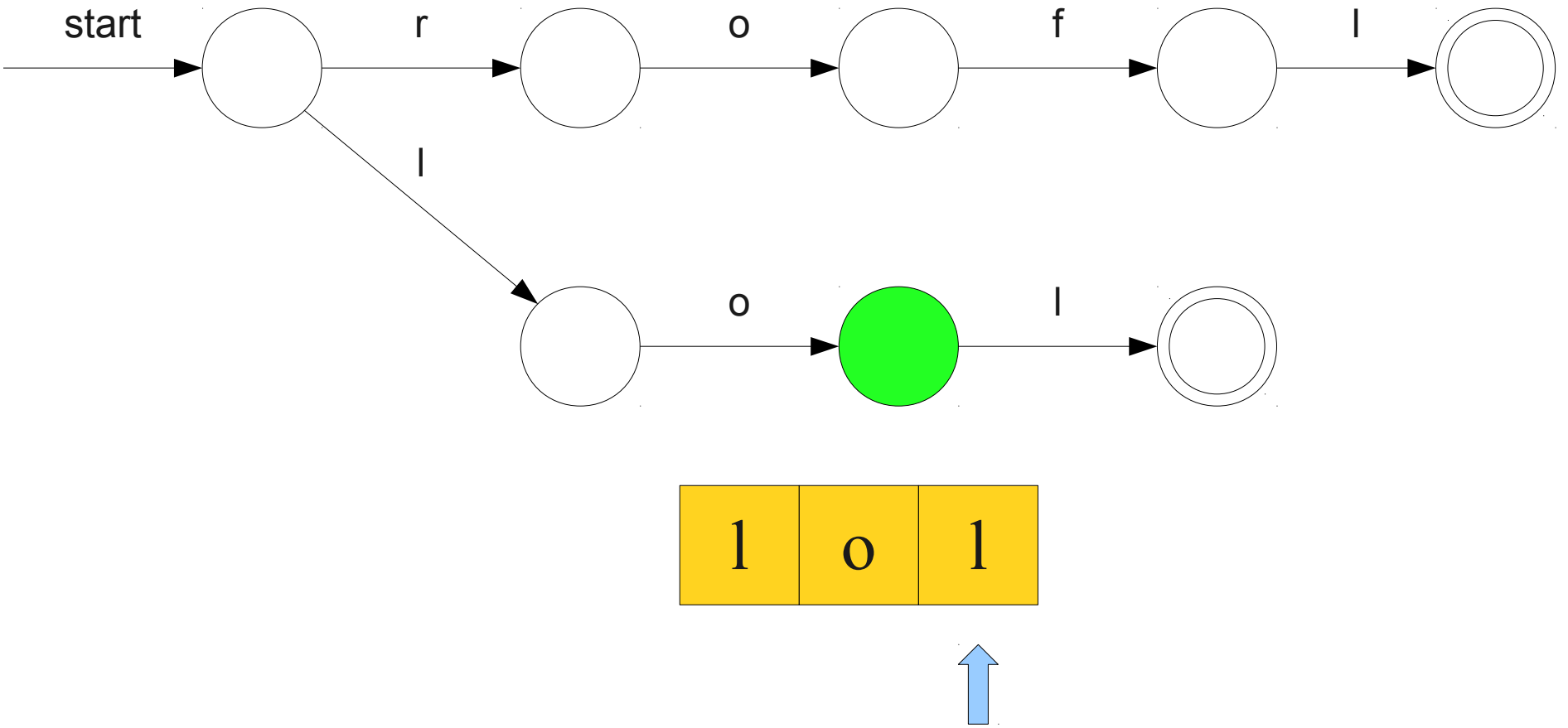
A Simple NFA



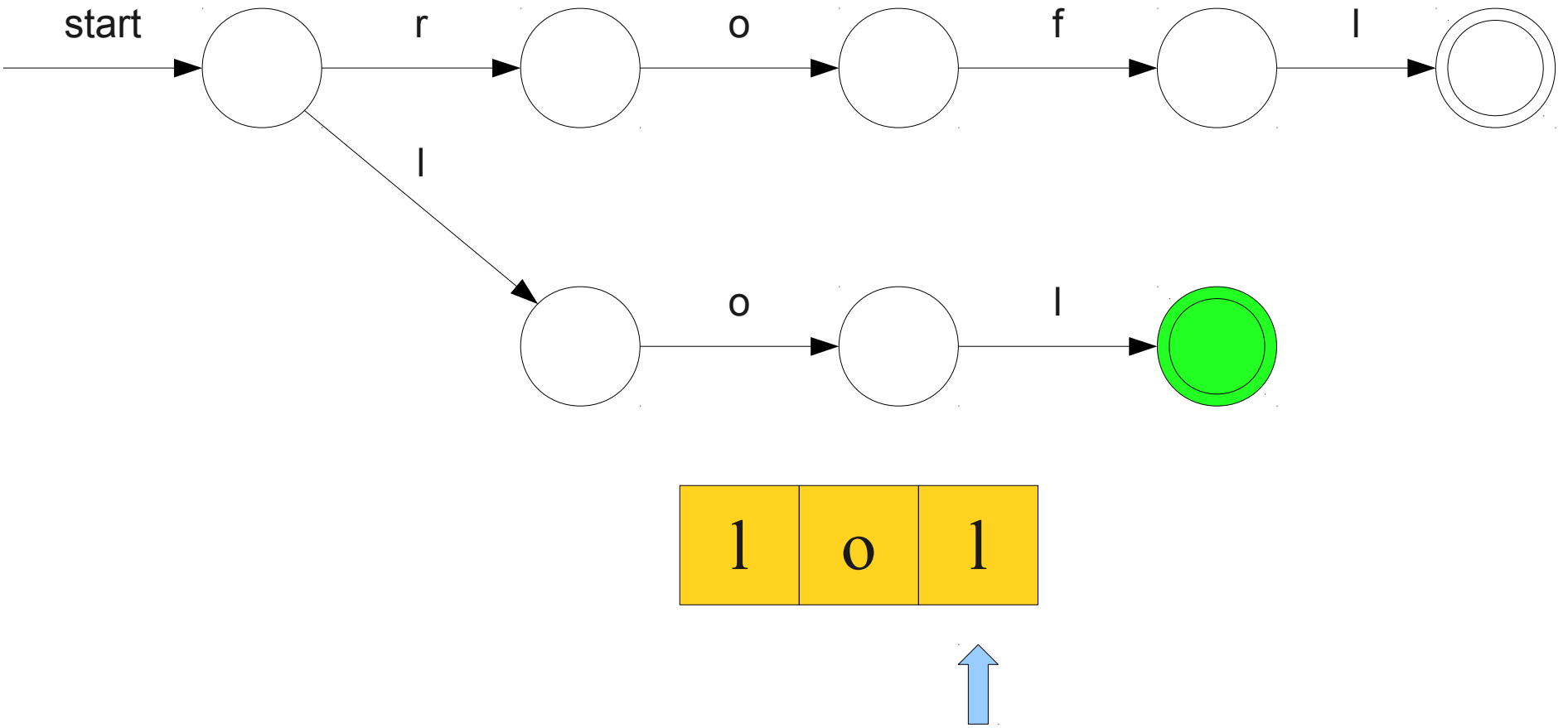
A Simple NFA



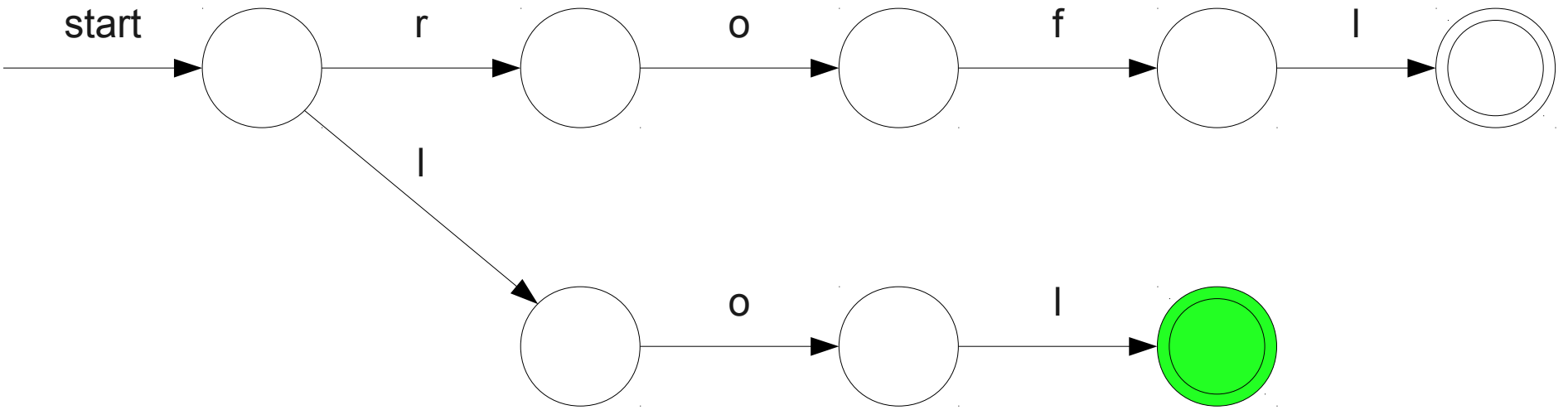
A Simple NFA



A Simple NFA

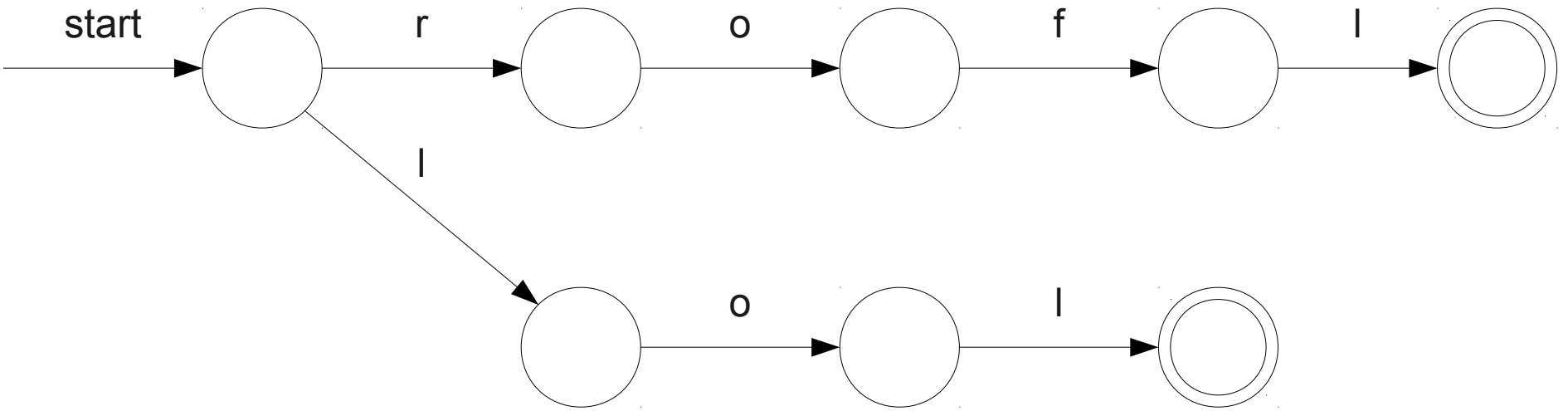


A Simple NFA

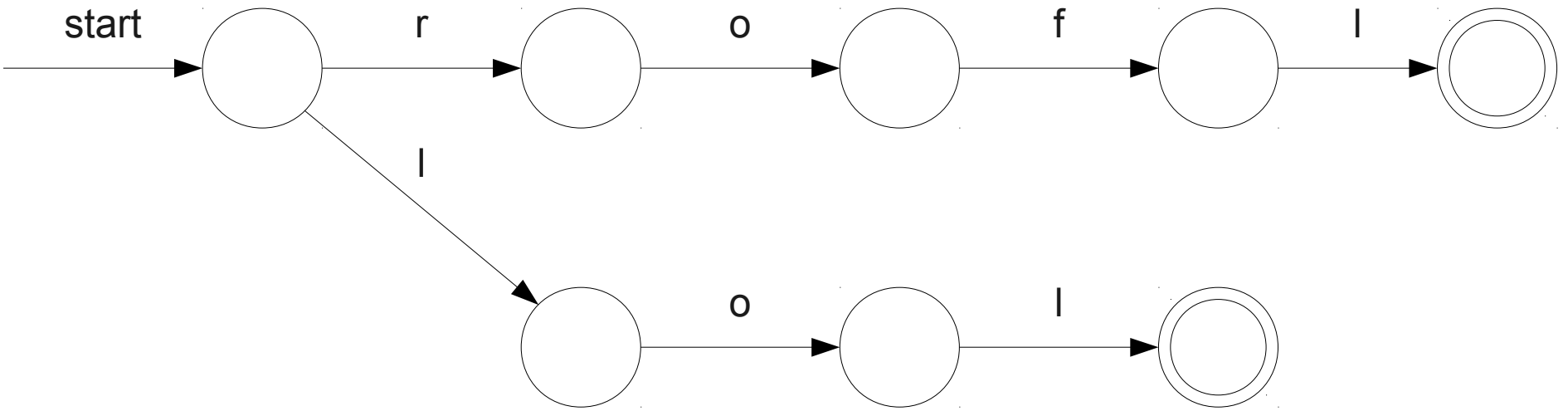


l	o	l

A Simple NFA

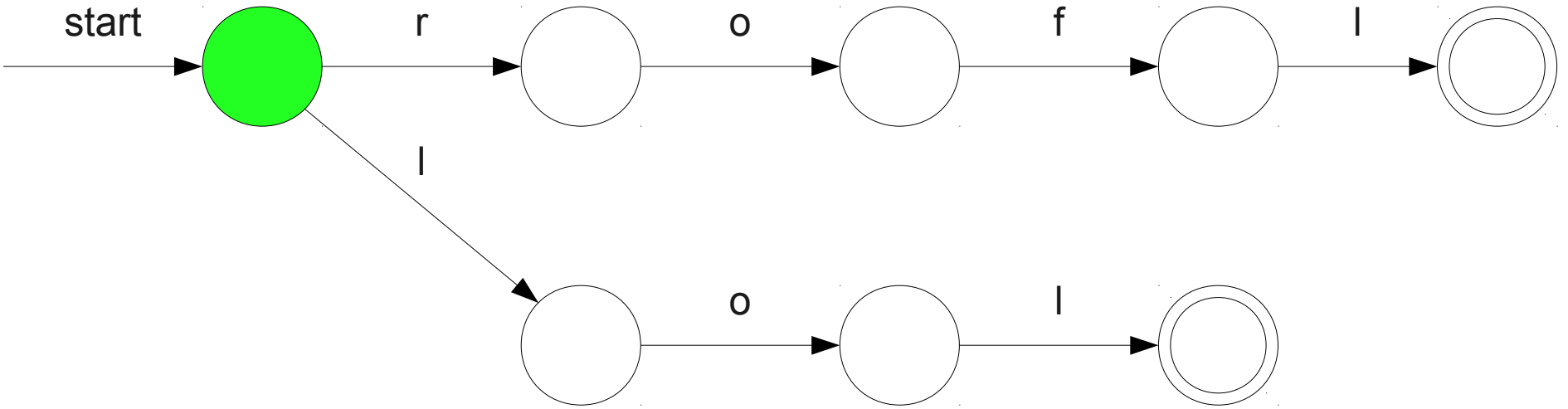


A Simple NFA



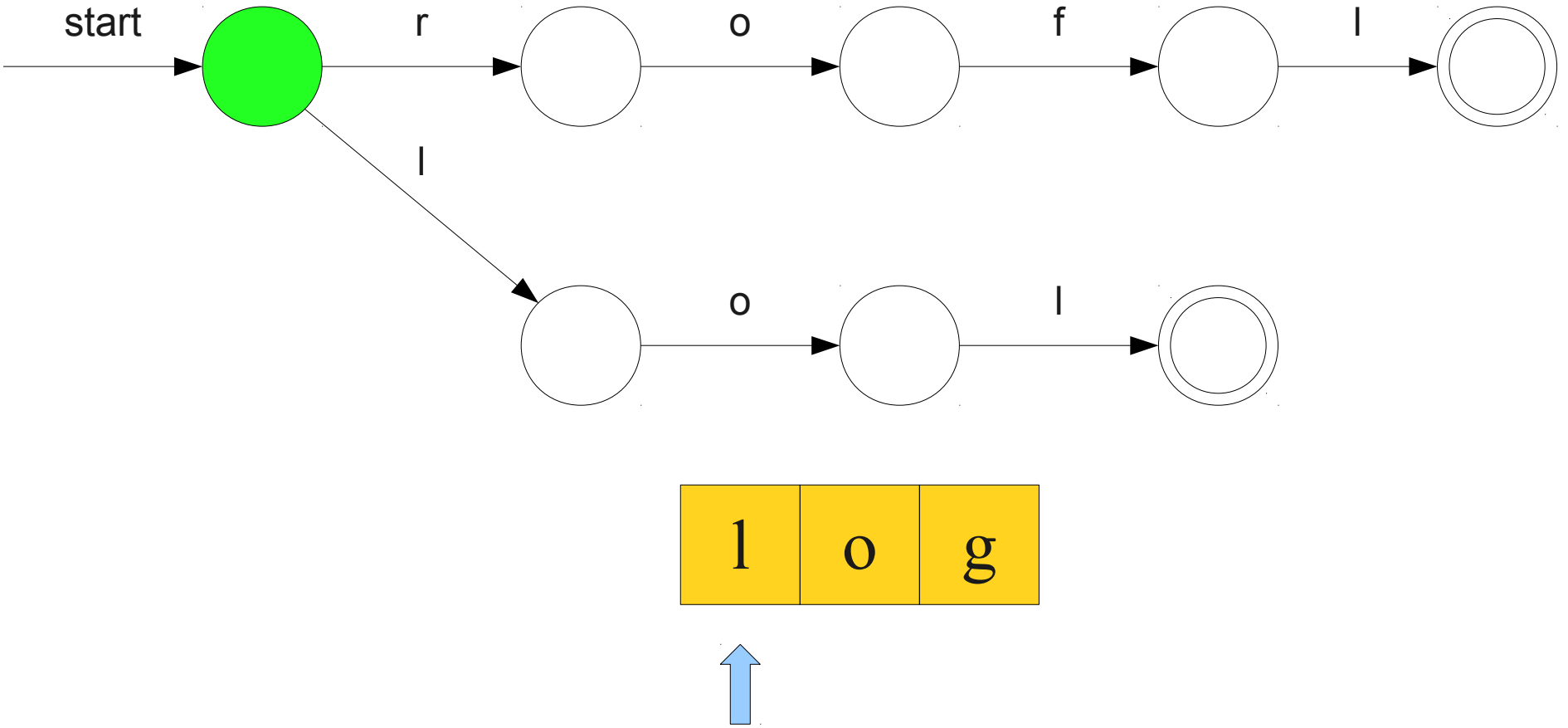
l	o	g

A Simple NFA

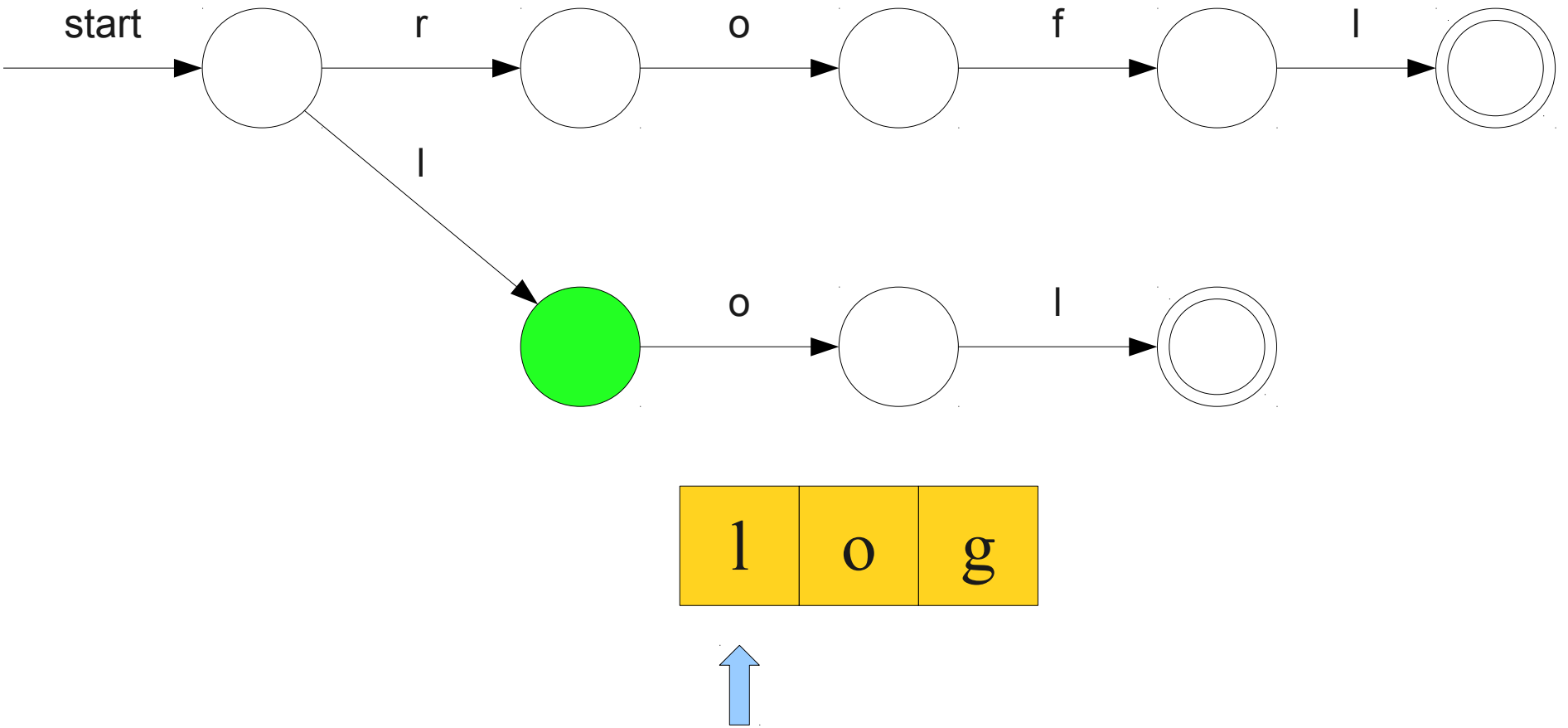


l	o	g

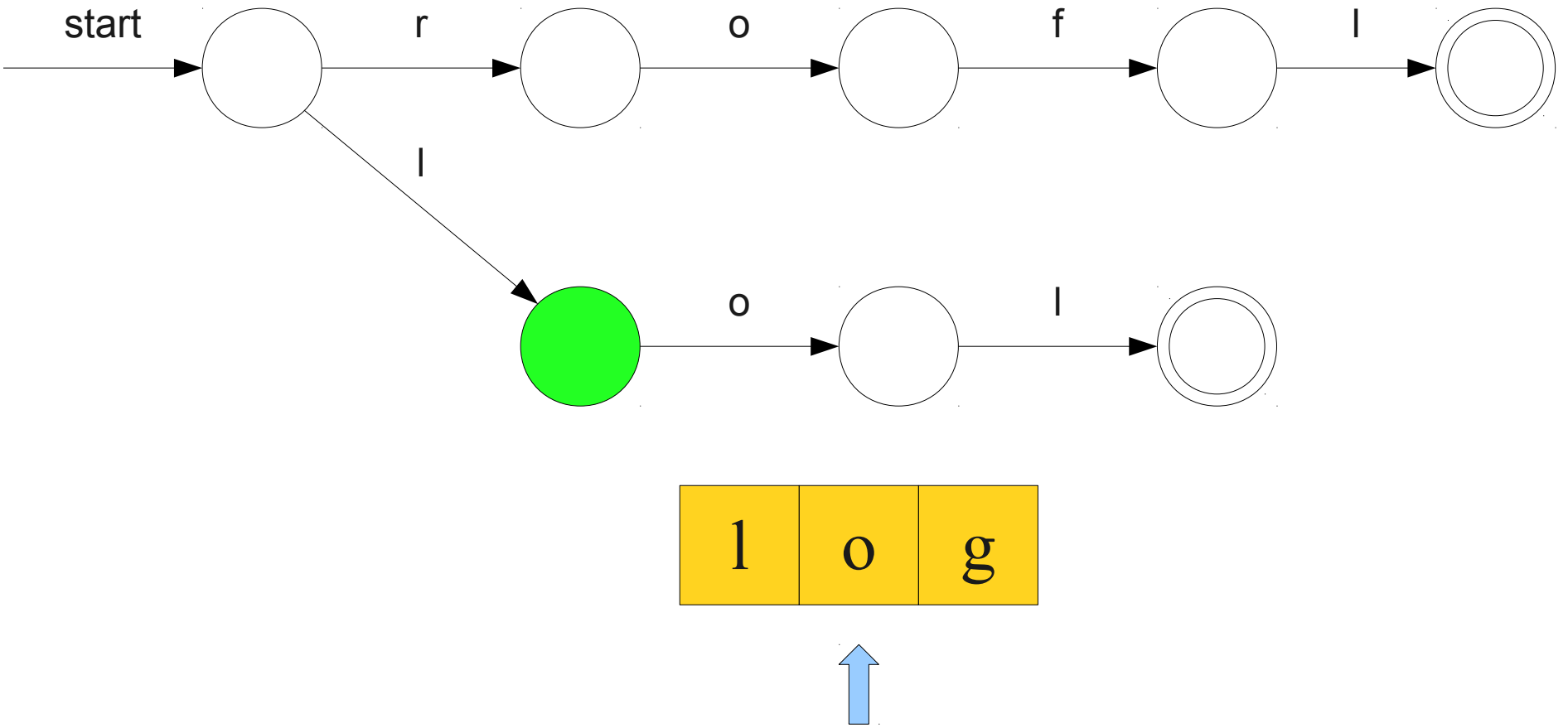
A Simple NFA



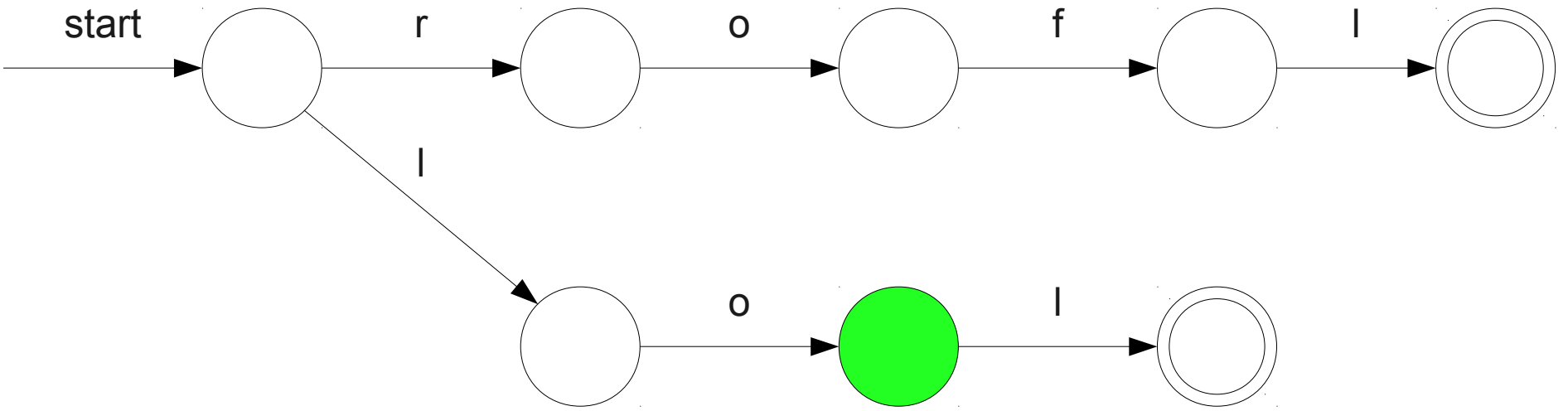
A Simple NFA

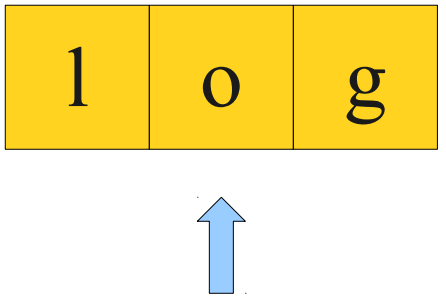


A Simple NFA

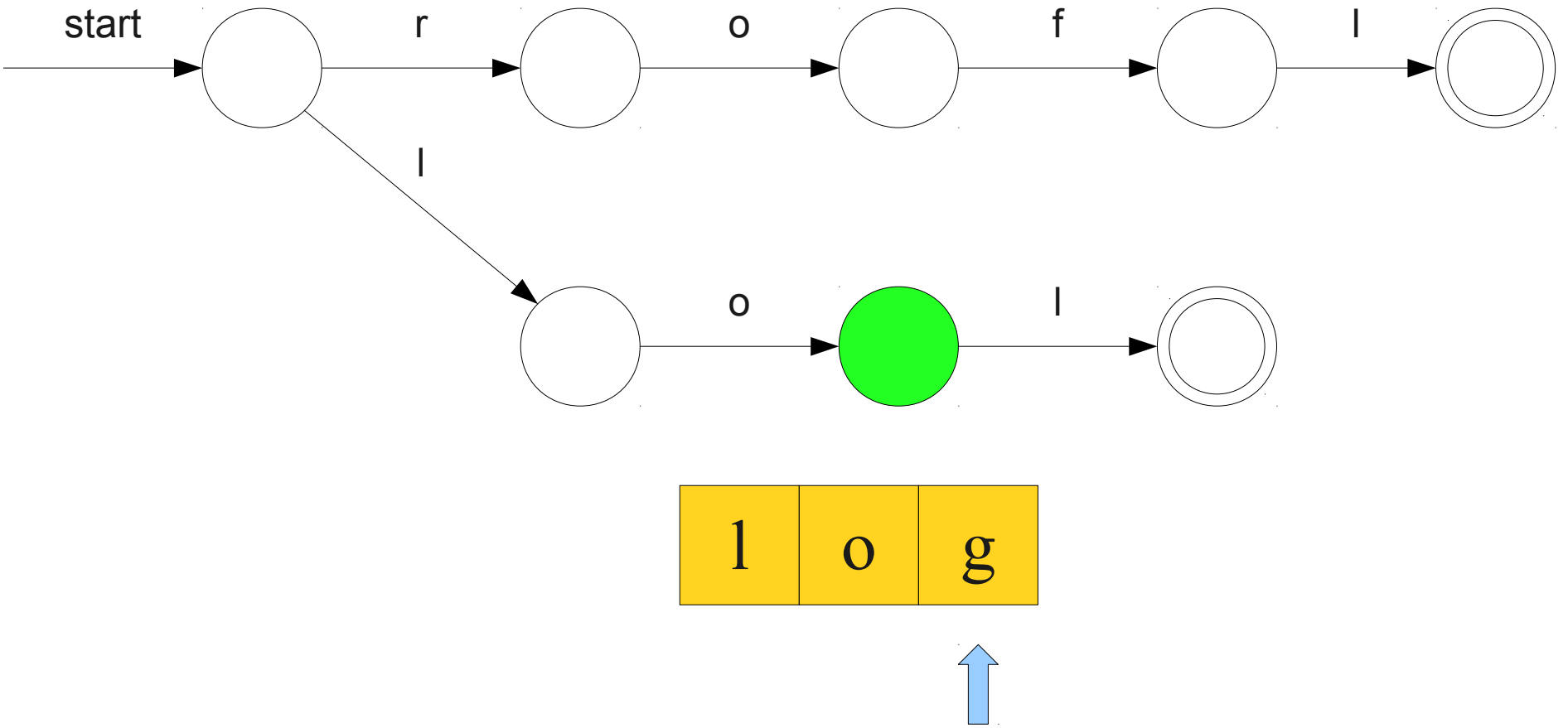


A Simple NFA

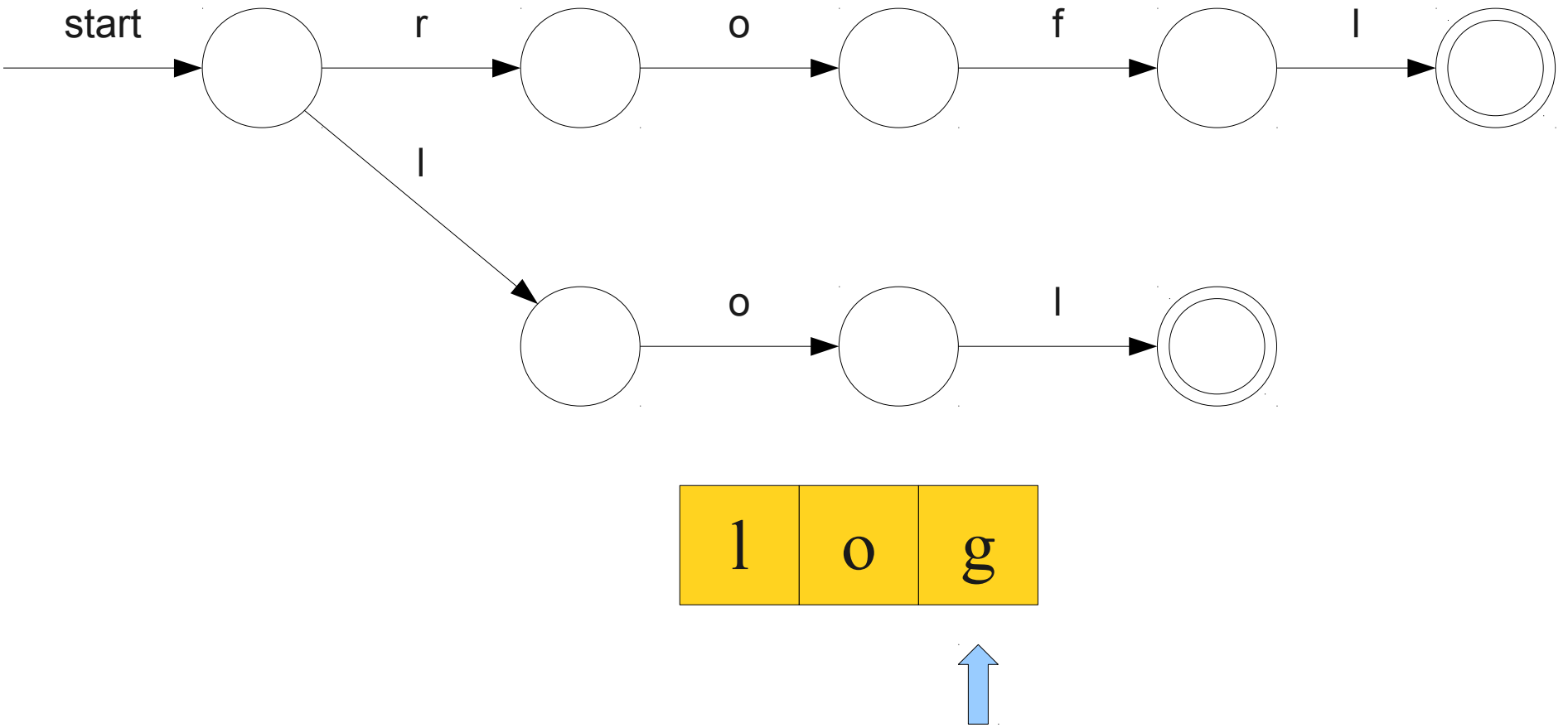




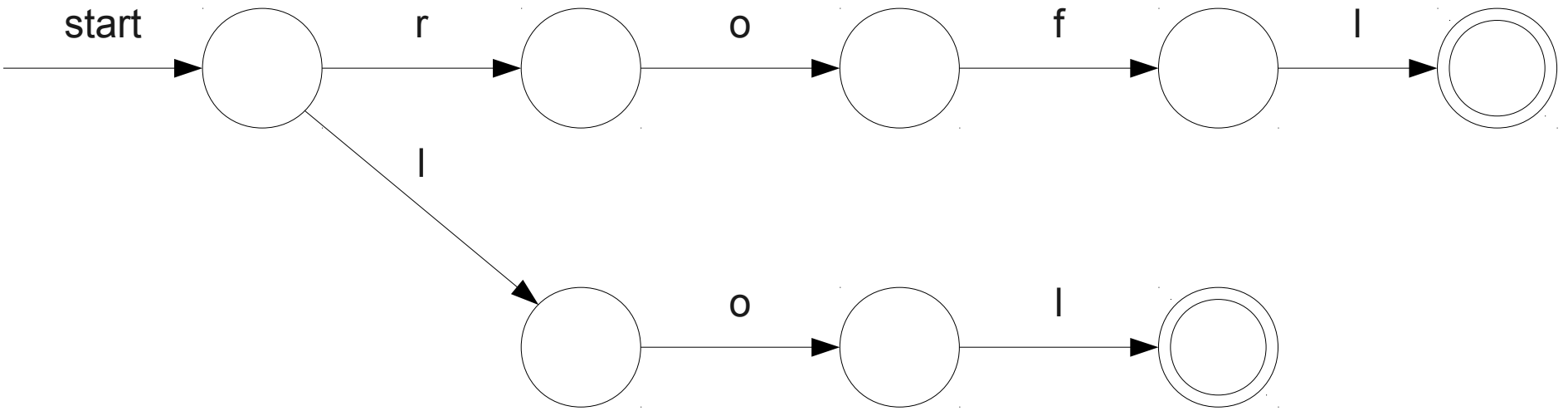
A Simple NFA



A Simple NFA

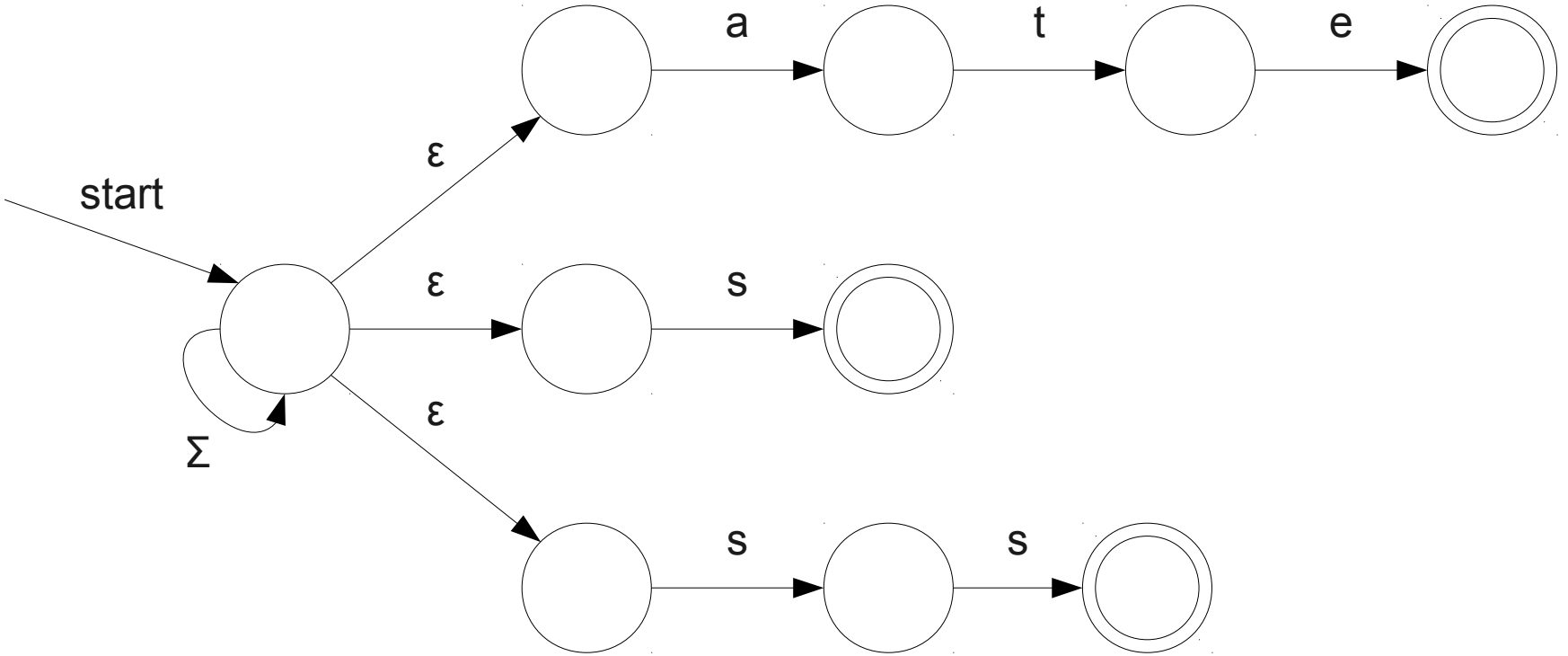


A Simple NFA

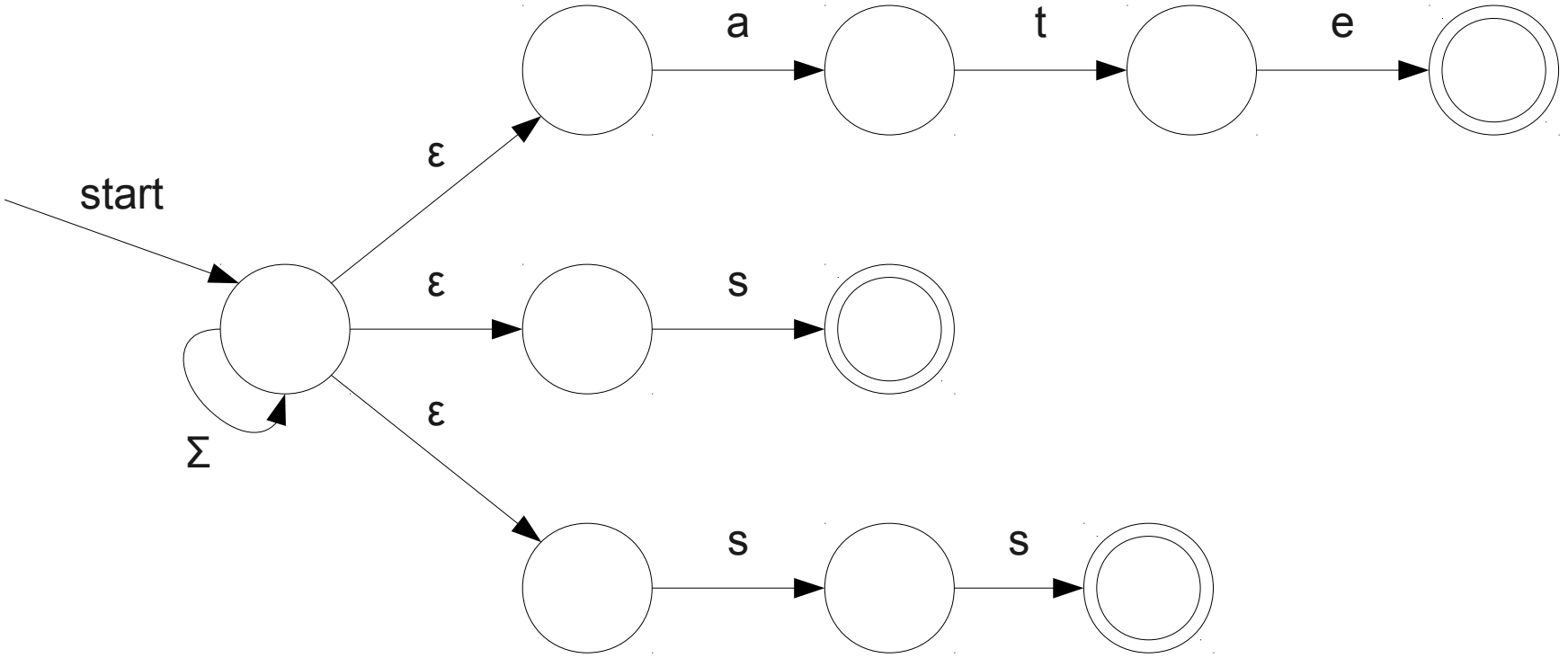


l	o	g

A More Complex NFA

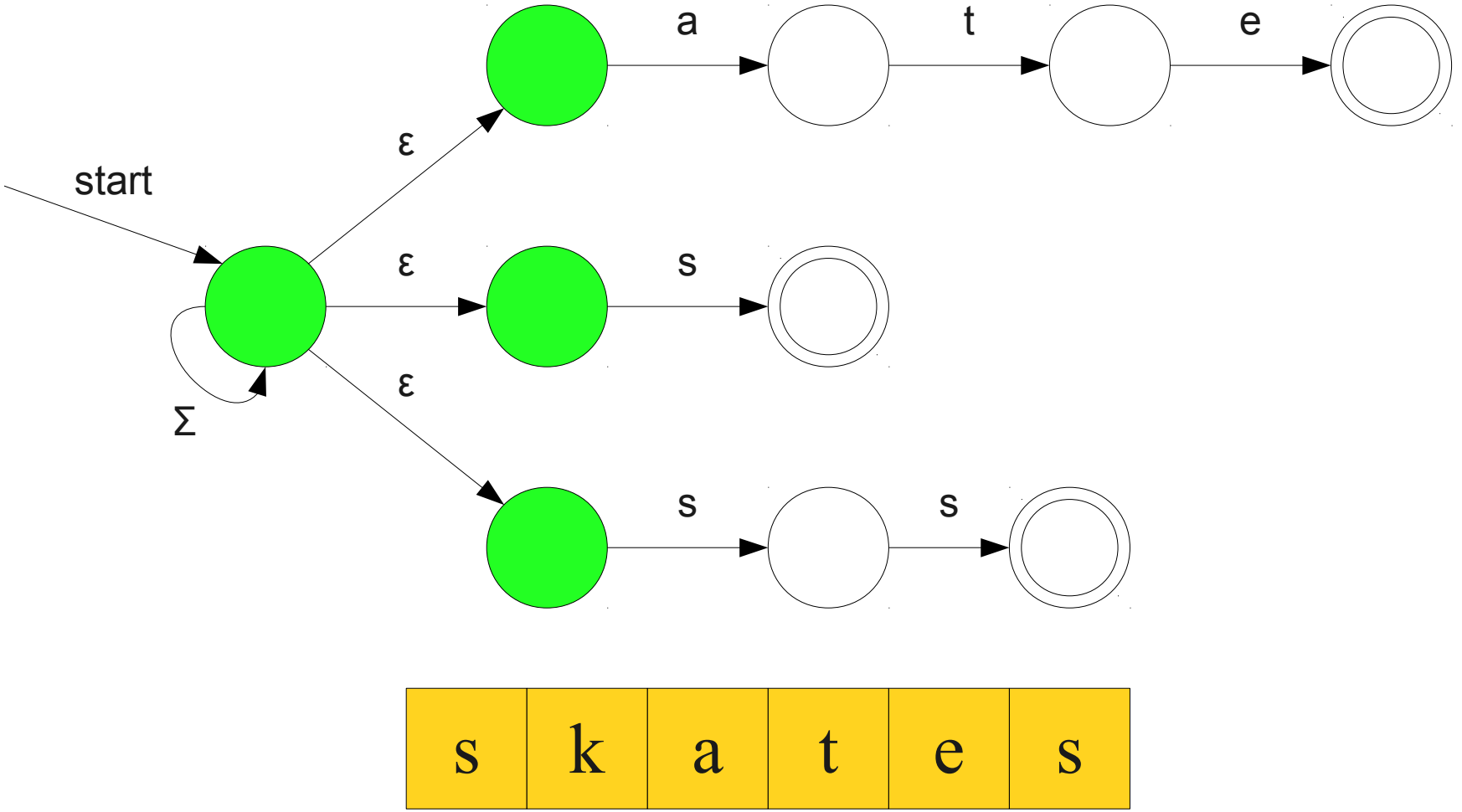


A More Complex NFA



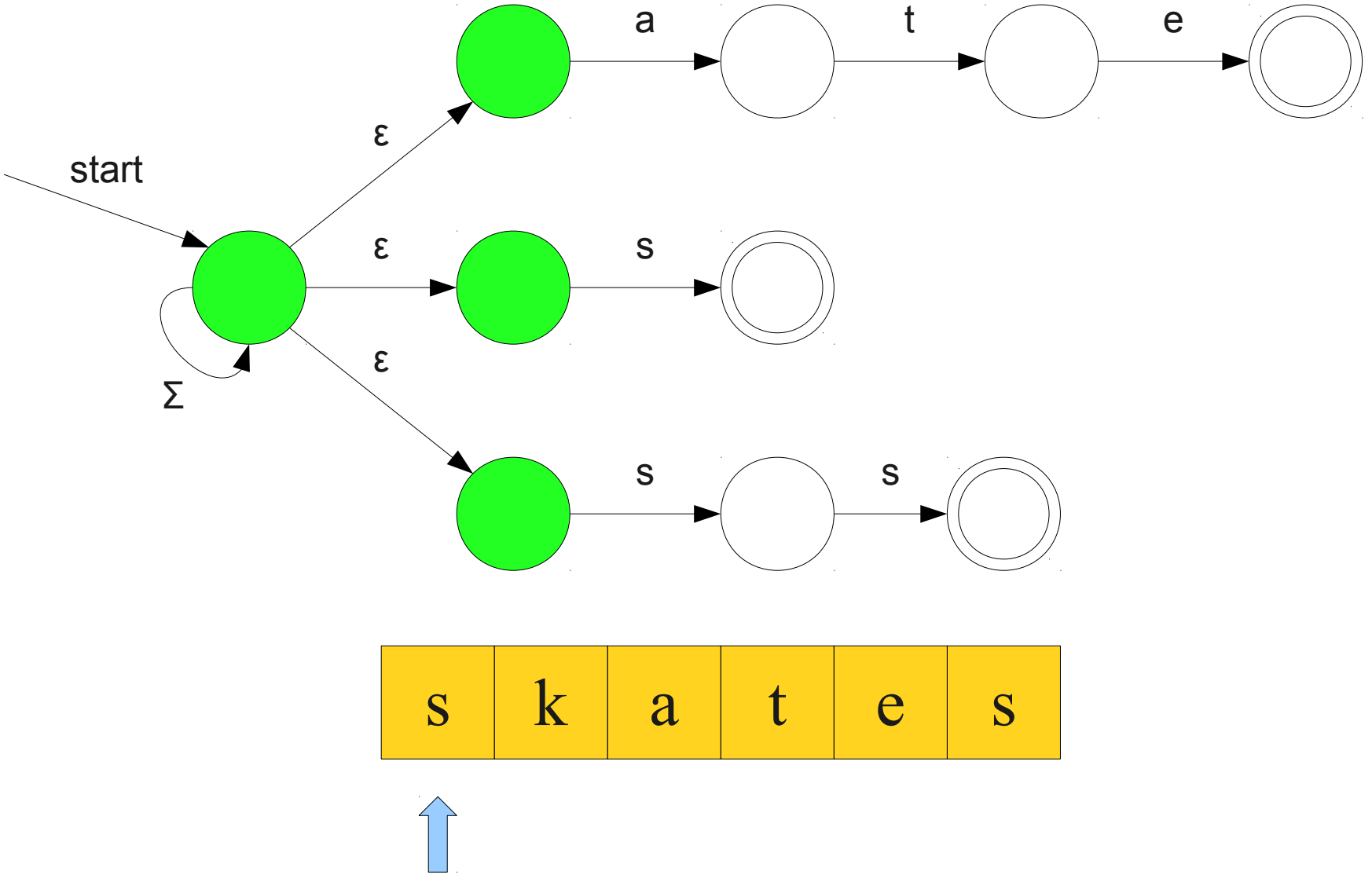
s	k	a	t	e	s

A More Complex NFA

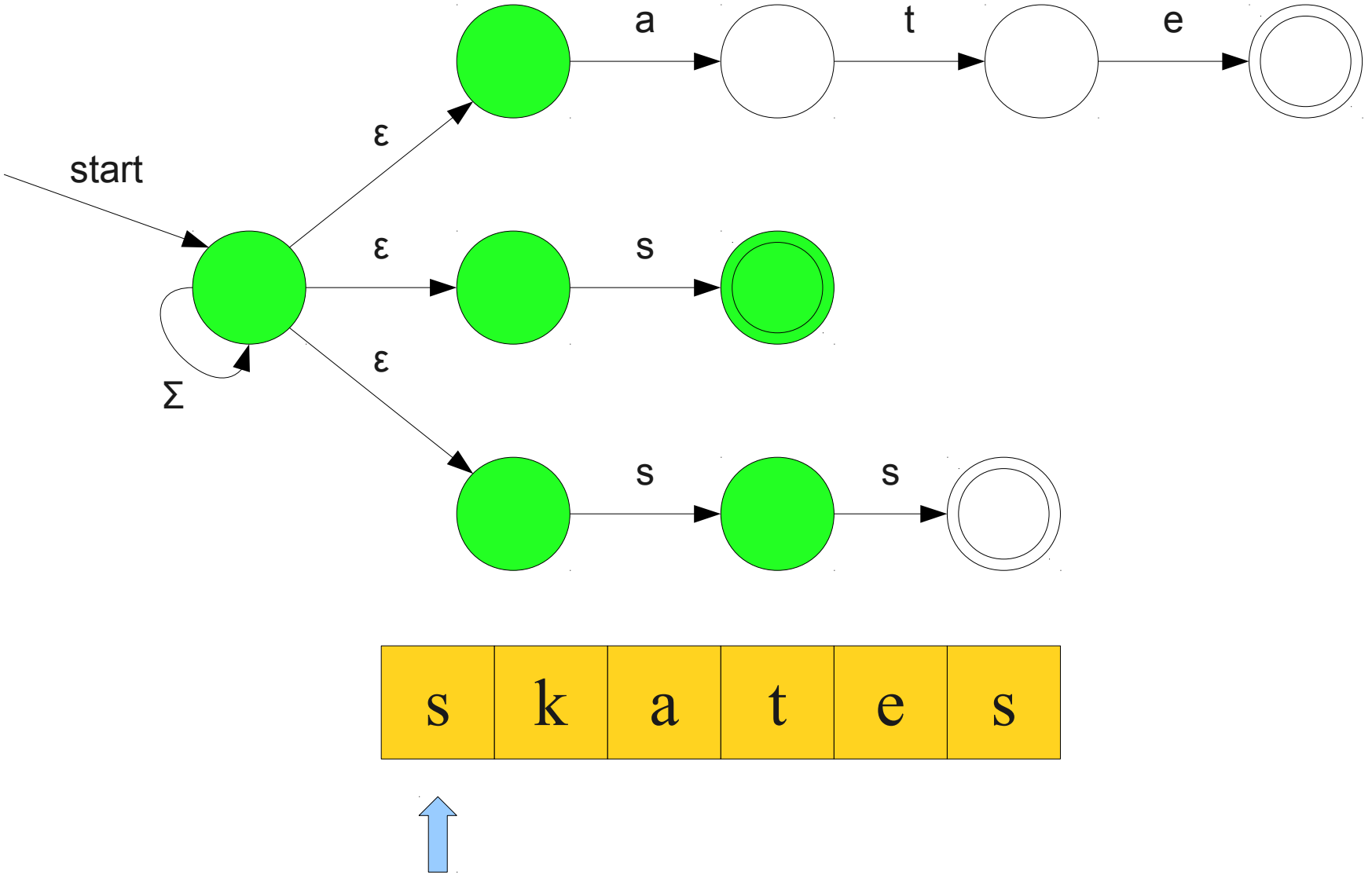


s	k	a	t	e	s

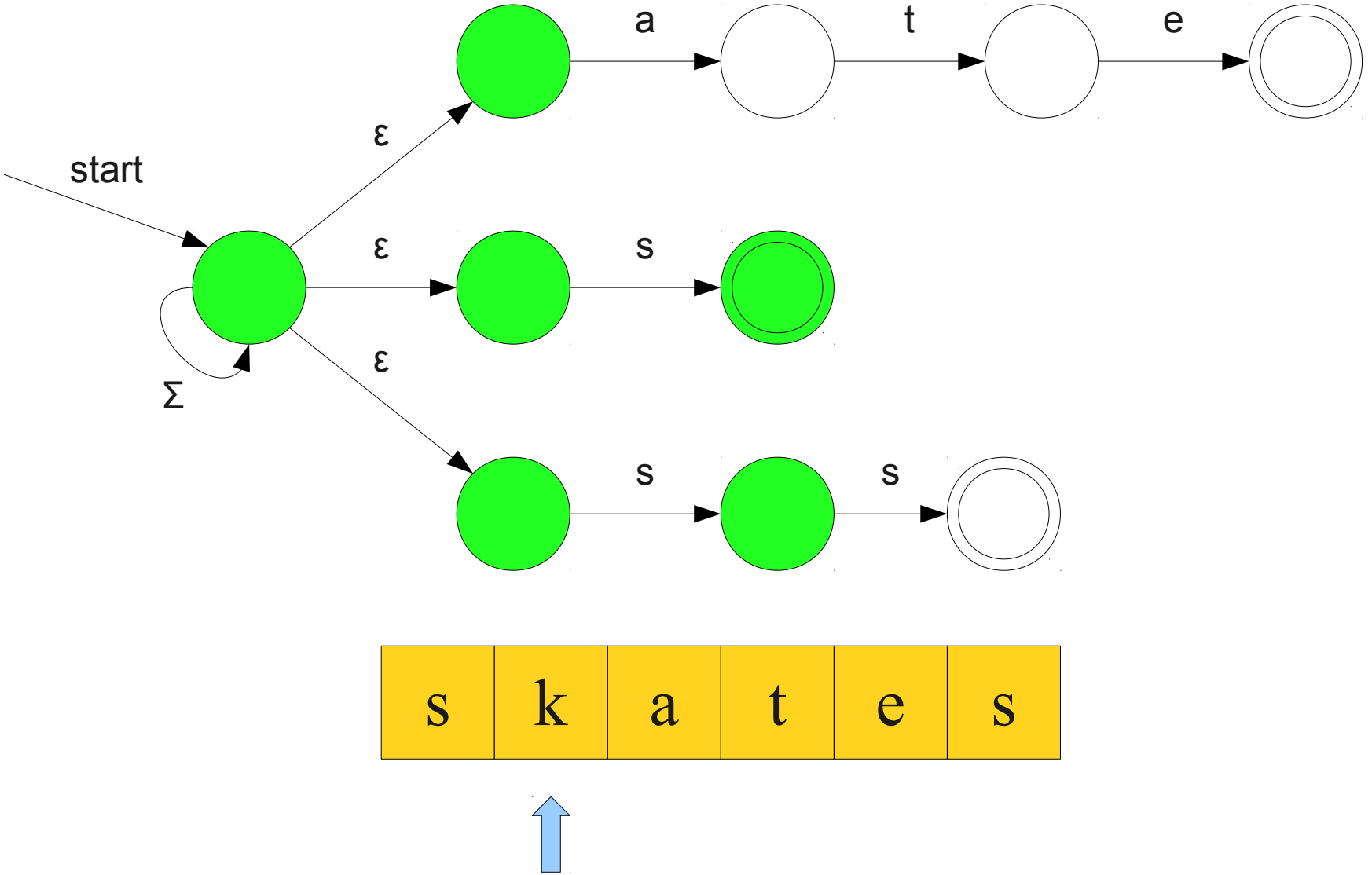
A More Complex NFA



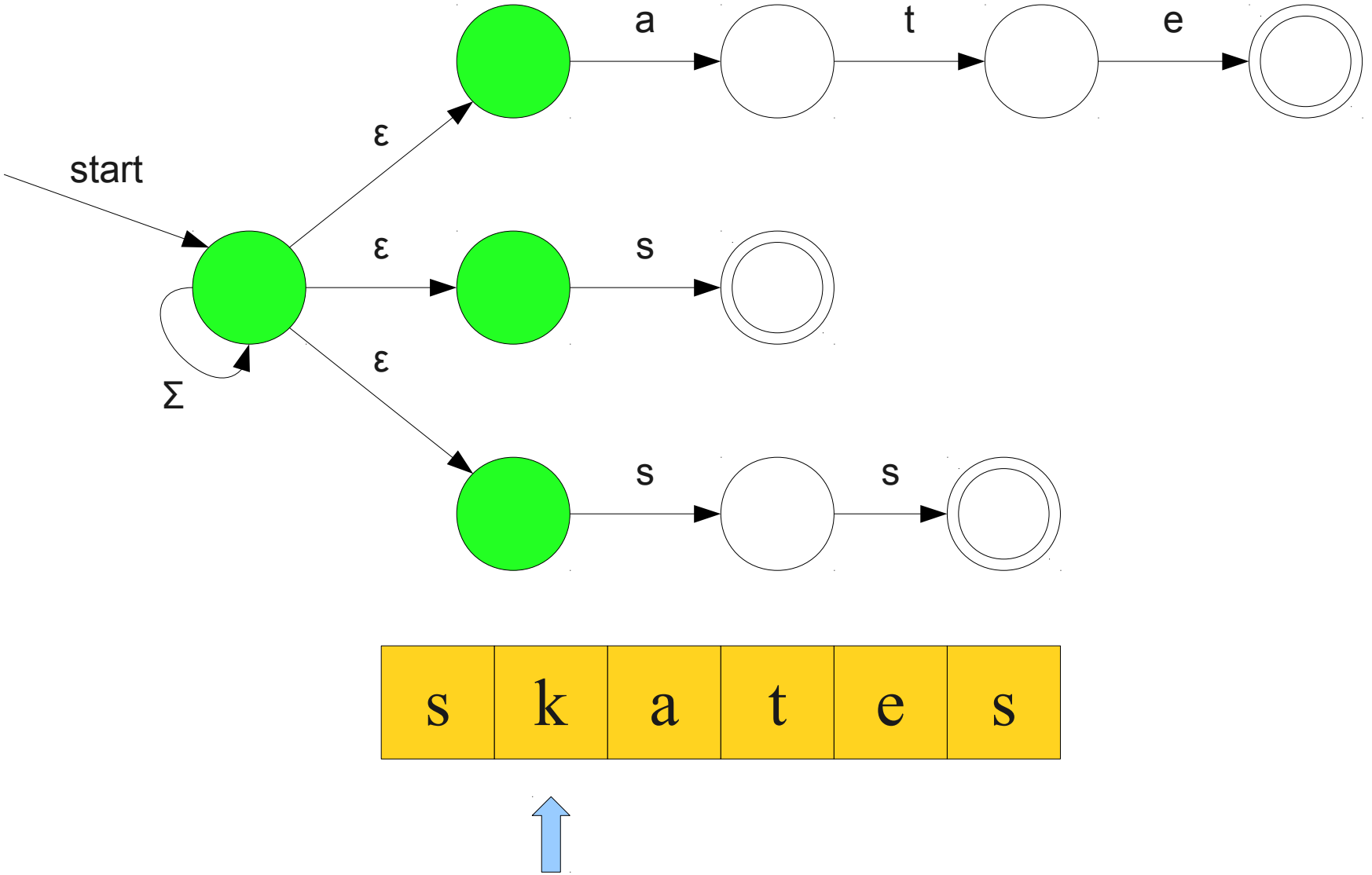
A More Complex NFA



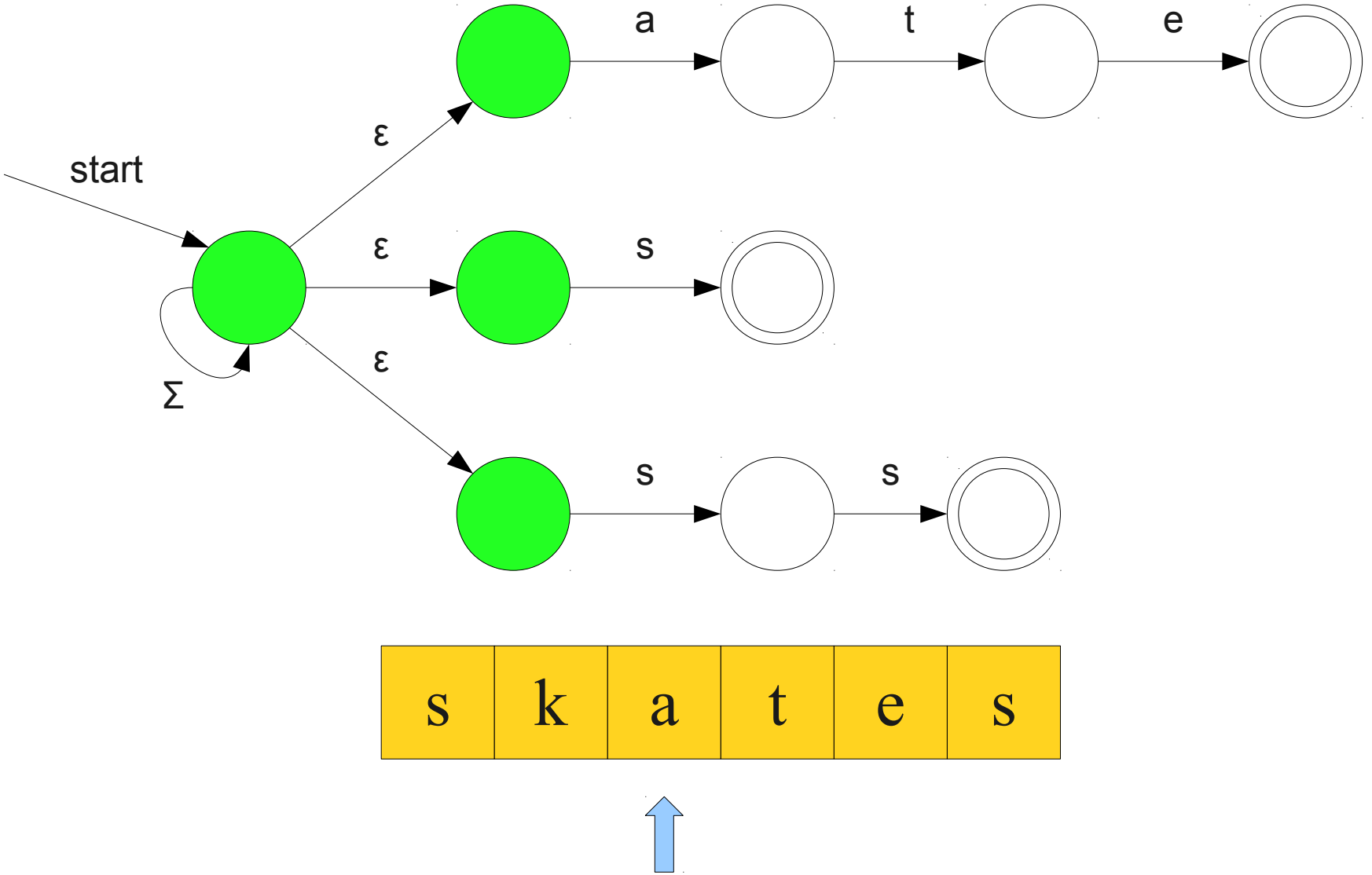
A More Complex NFA



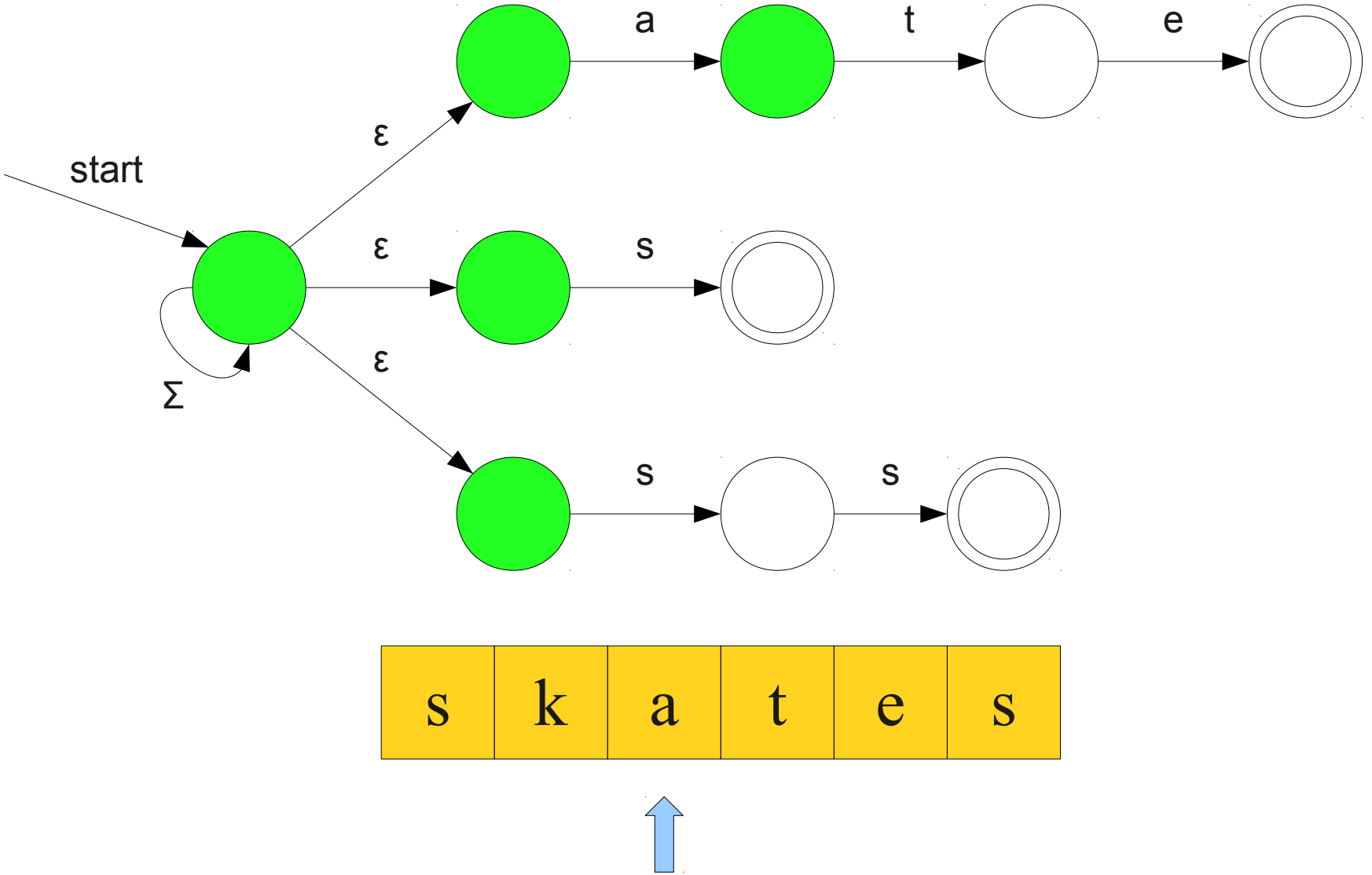
A More Complex NFA



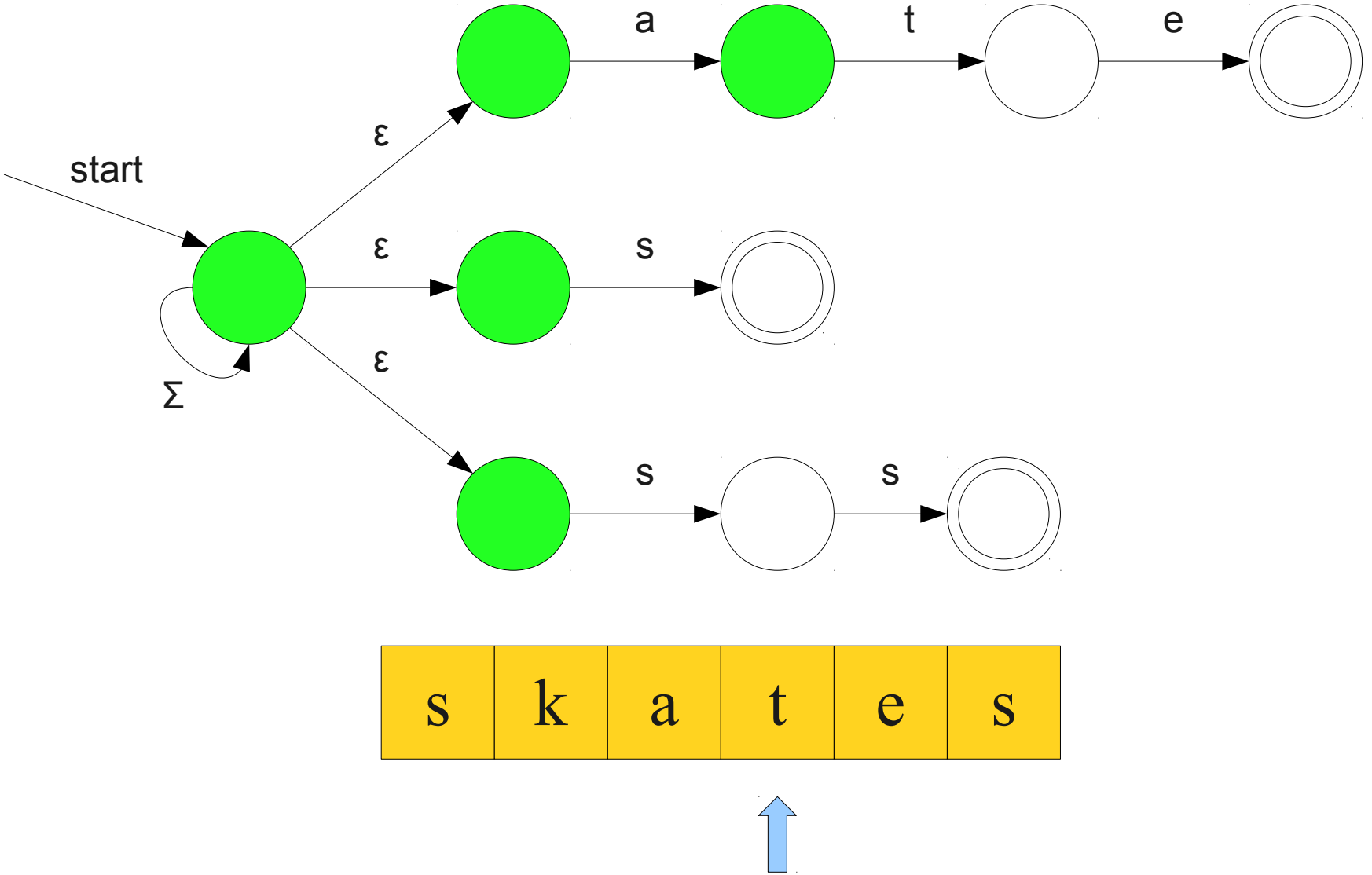
A More Complex NFA



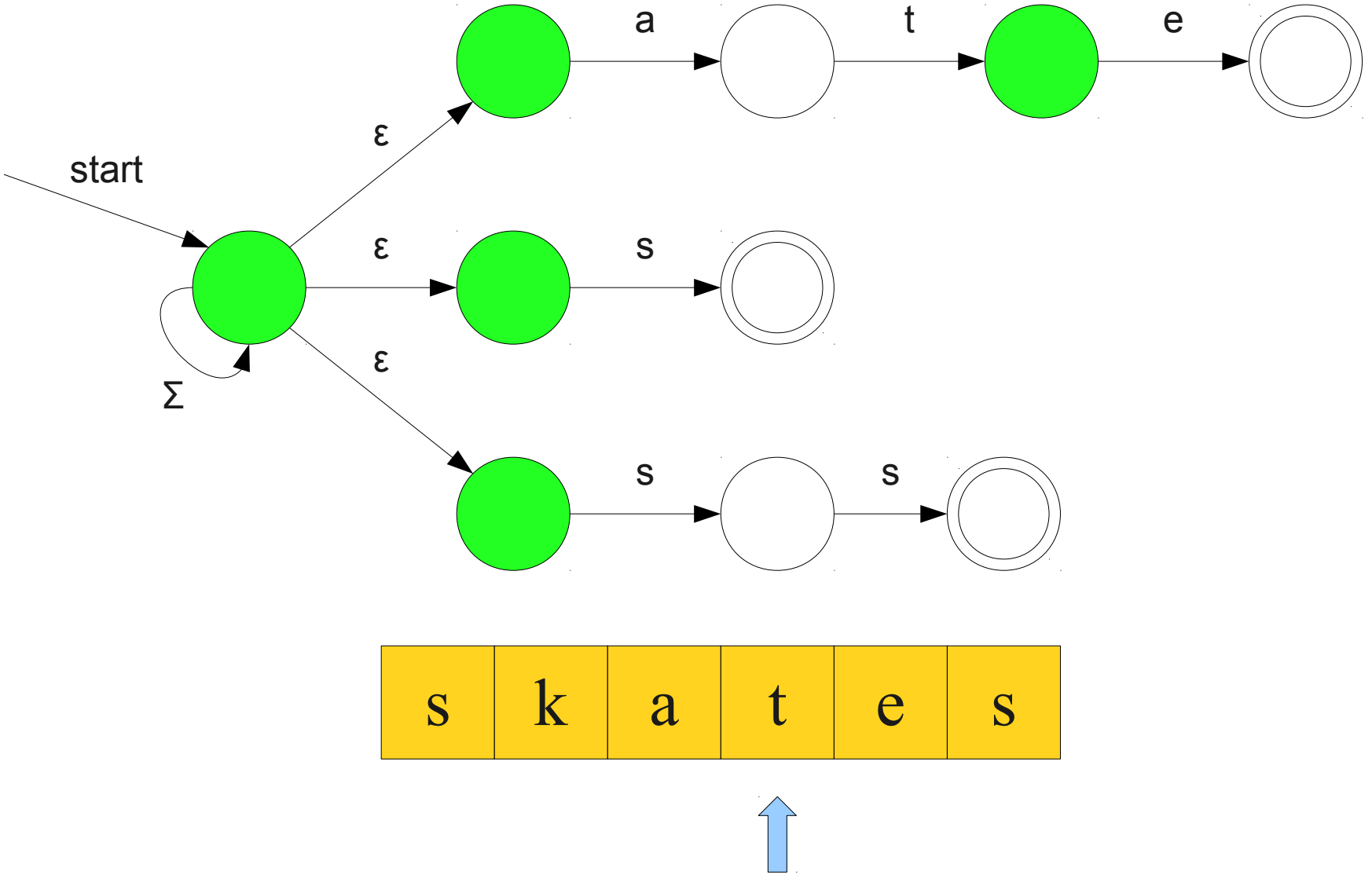
A More Complex NFA



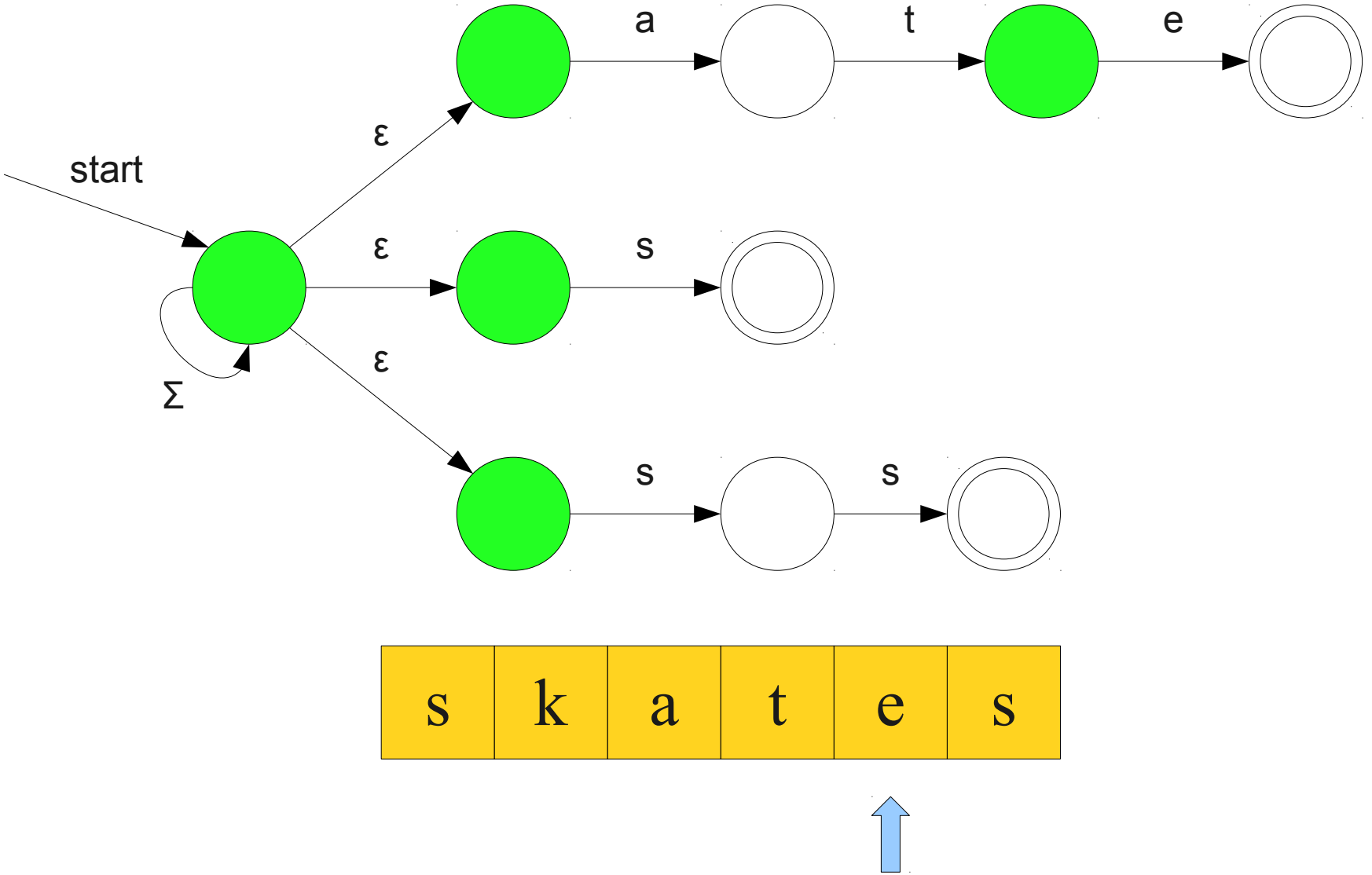
A More Complex NFA



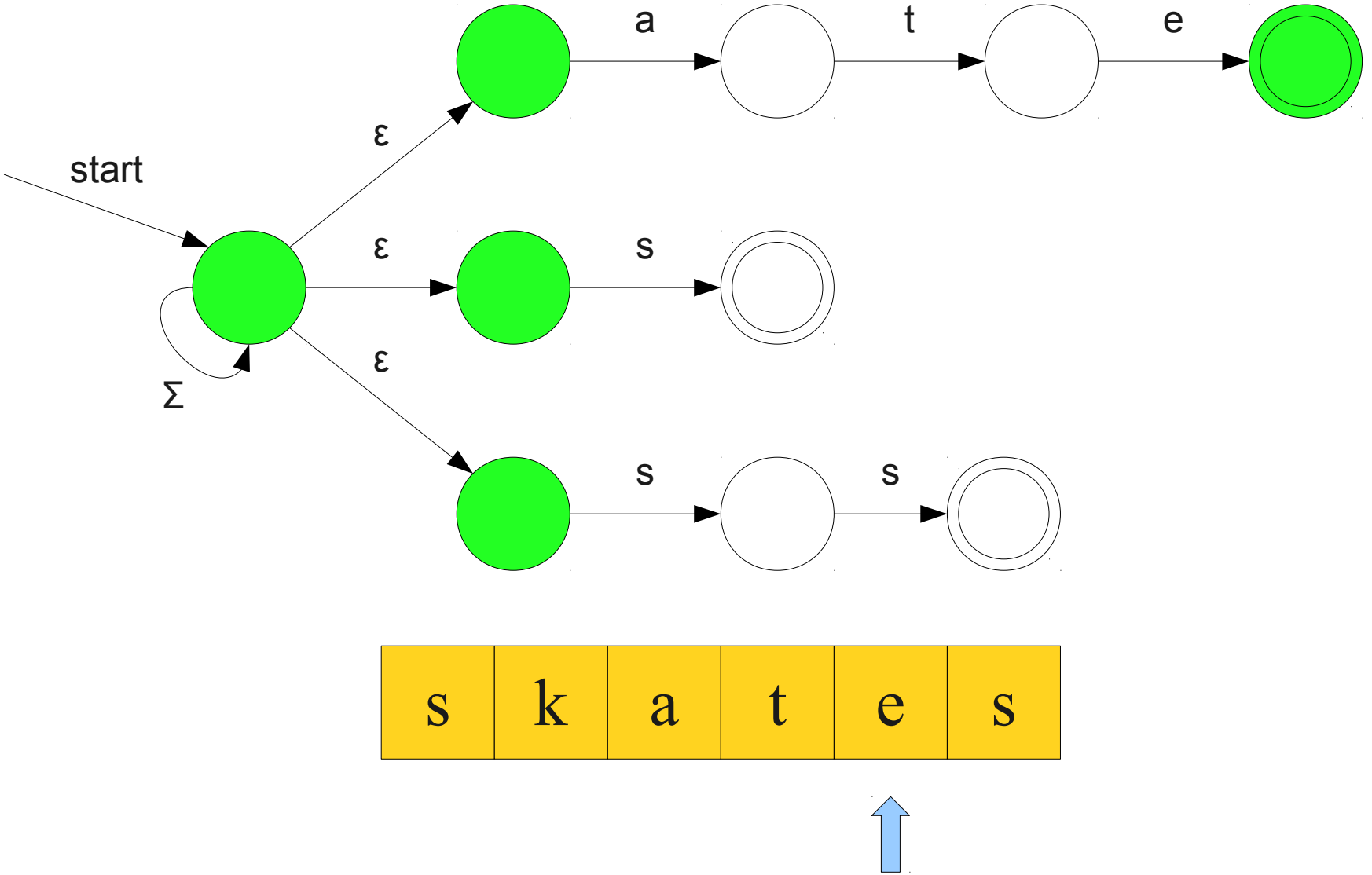
A More Complex NFA



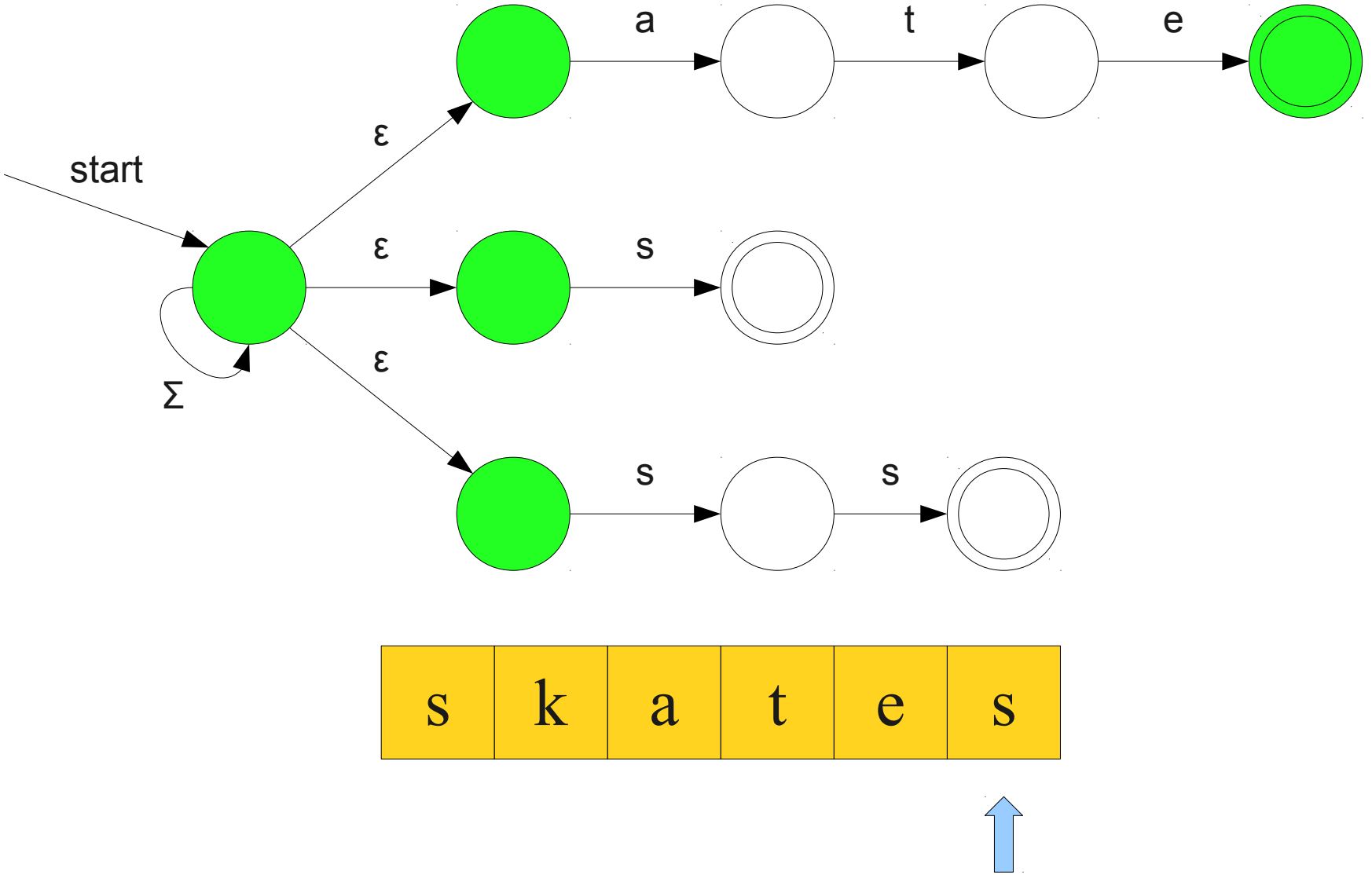
A More Complex NFA



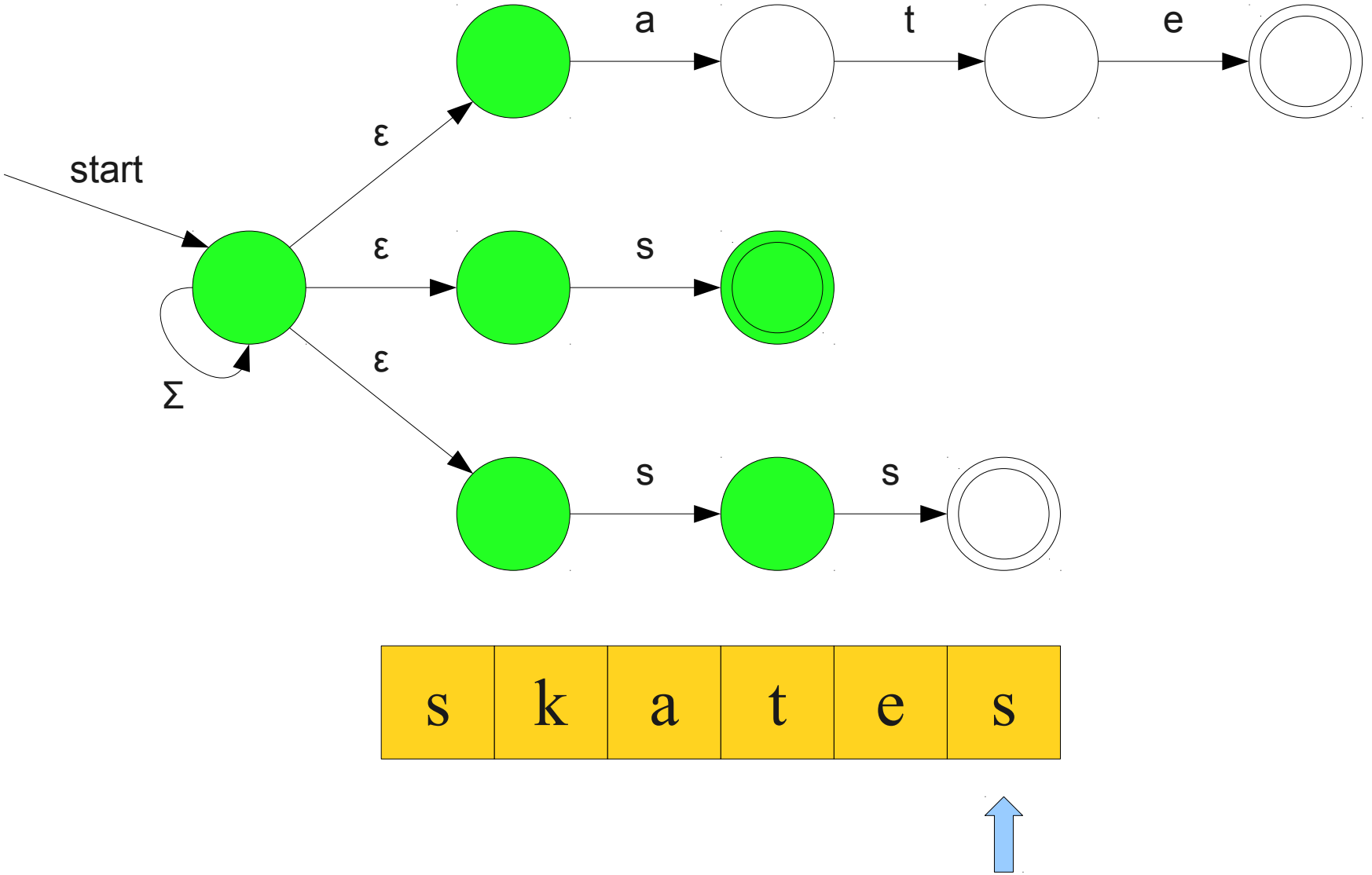
A More Complex NFA



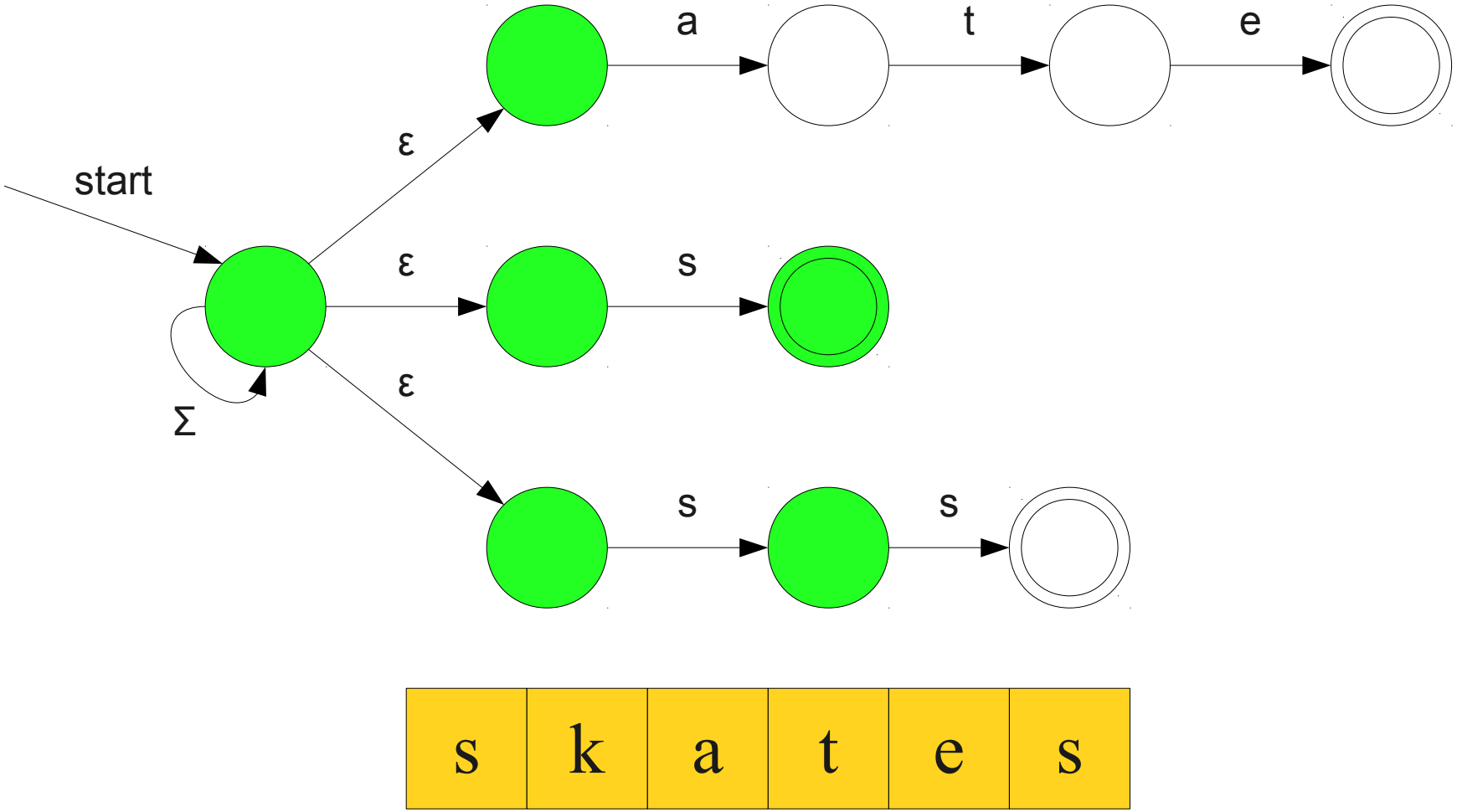
A More Complex NFA



A More Complex NFA



A More Complex NFA

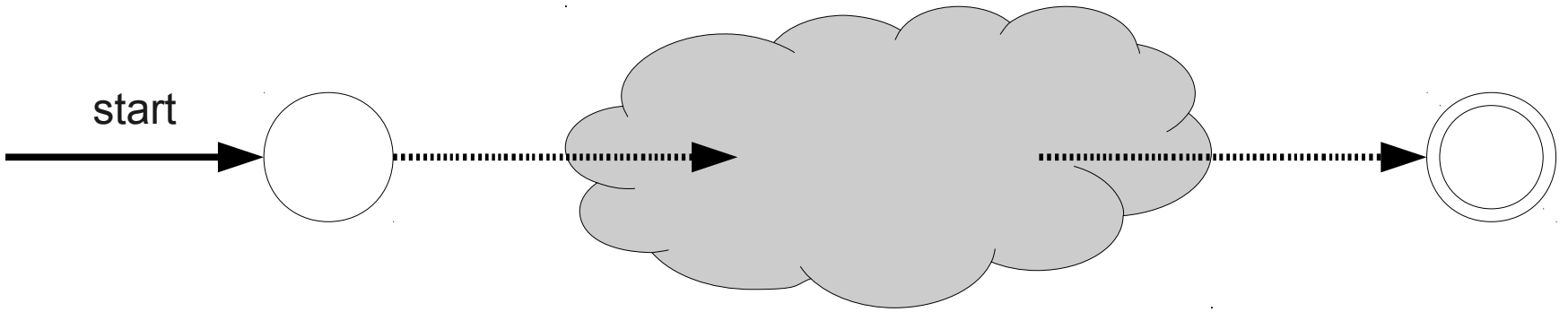


Simulating an NFA

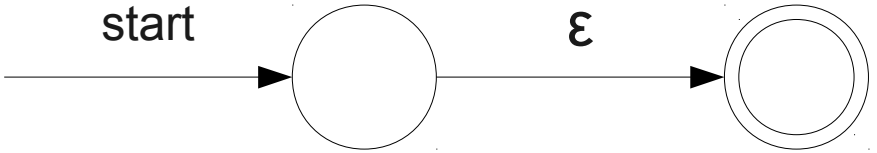
- Keep track of a set of states, initially the start state and everything reachable by ε-moves.
- For each character in the input:
  - Maintain a set of next states, initially empty.
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an ε-move to the set of next states.
- Accept if at least one state in the set of states is accepting.
- Complexity: $O(|S||Q|^2)$ for strings of length |S| and |Q| states.

From Regular Expressions to NFAs

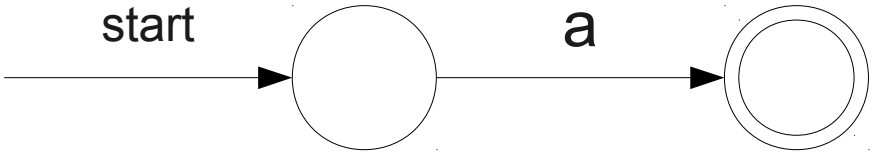
- There is a (beautiful!) procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting state.



Base Cases



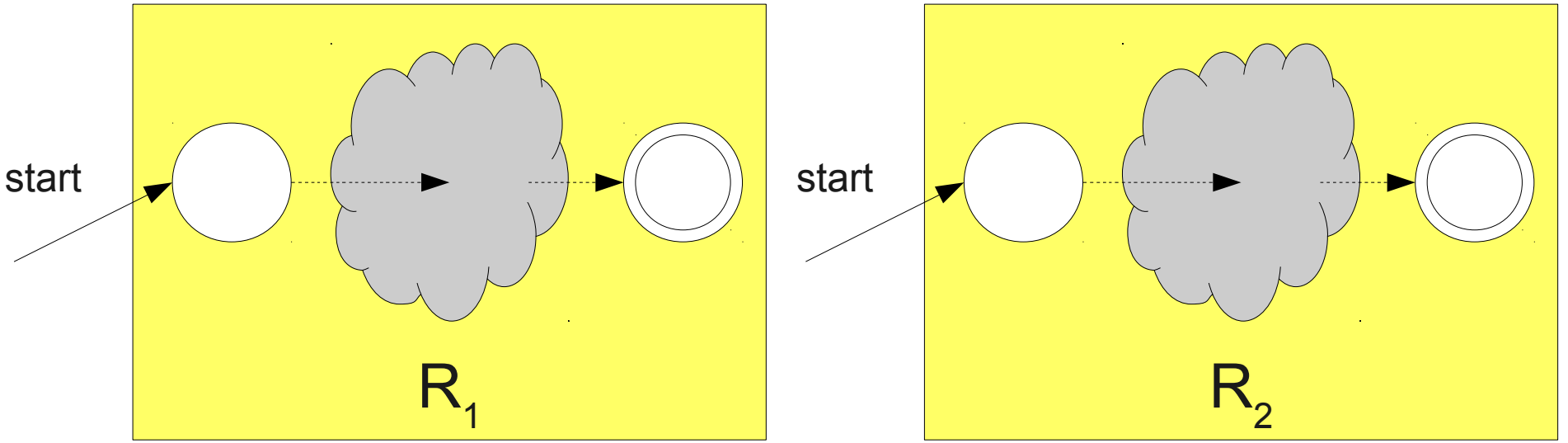
Automaton for ε



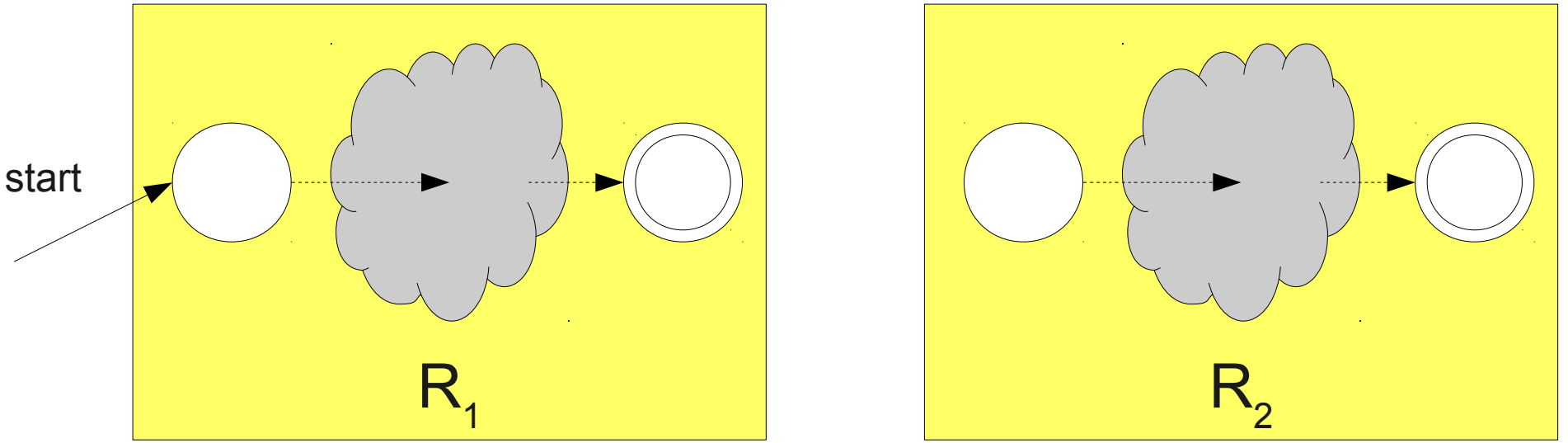
Automaton for single character **`a`**

Construction for R_1R_2

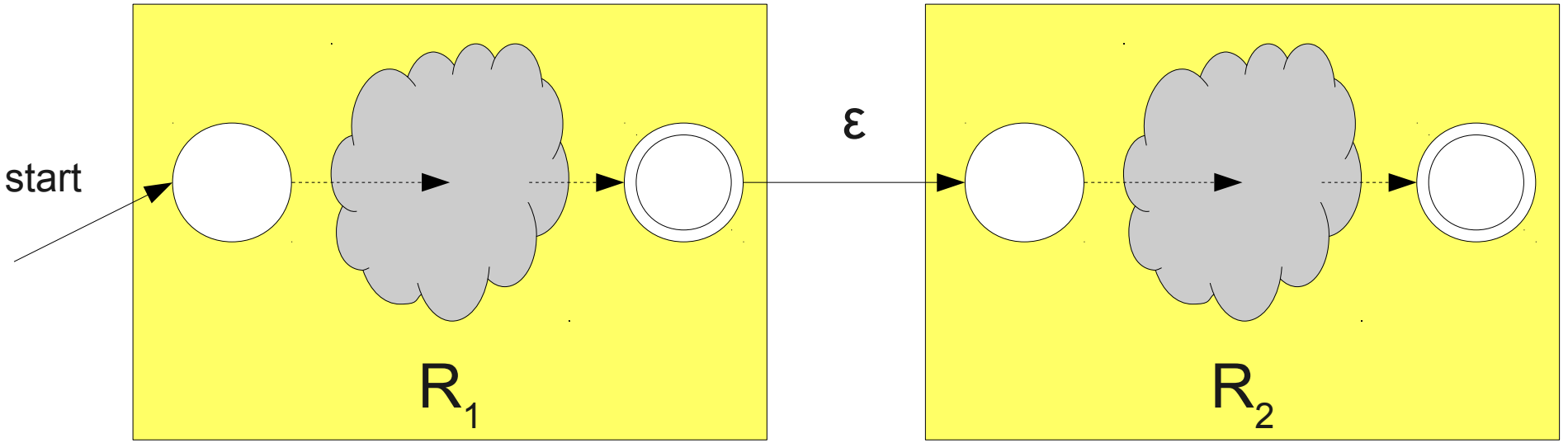
Construction for R_1R_2



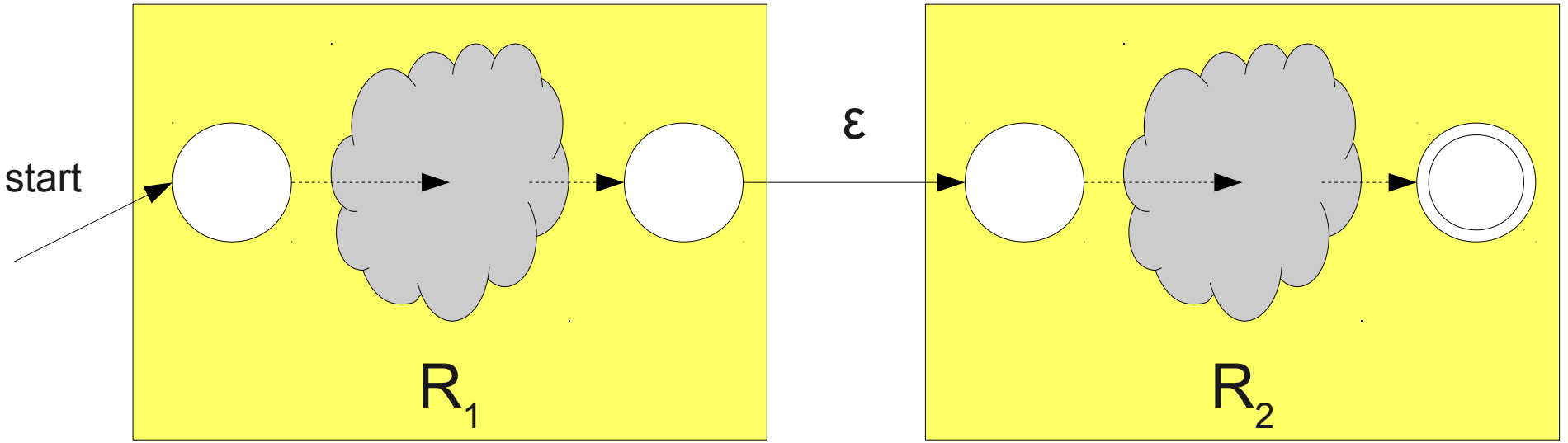
Construction for R_1R_2



Construction for R_1R_2

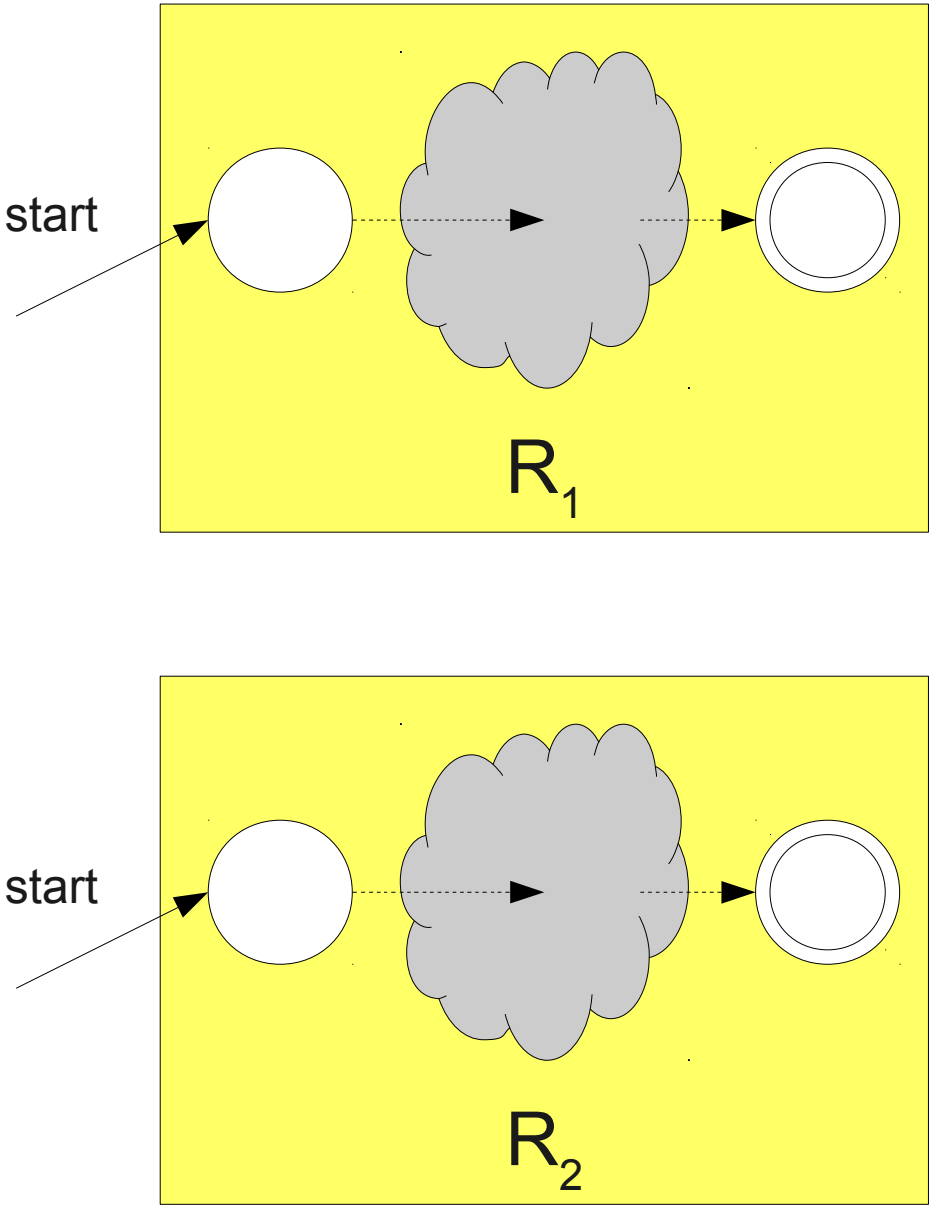


Construction for R_1R_2

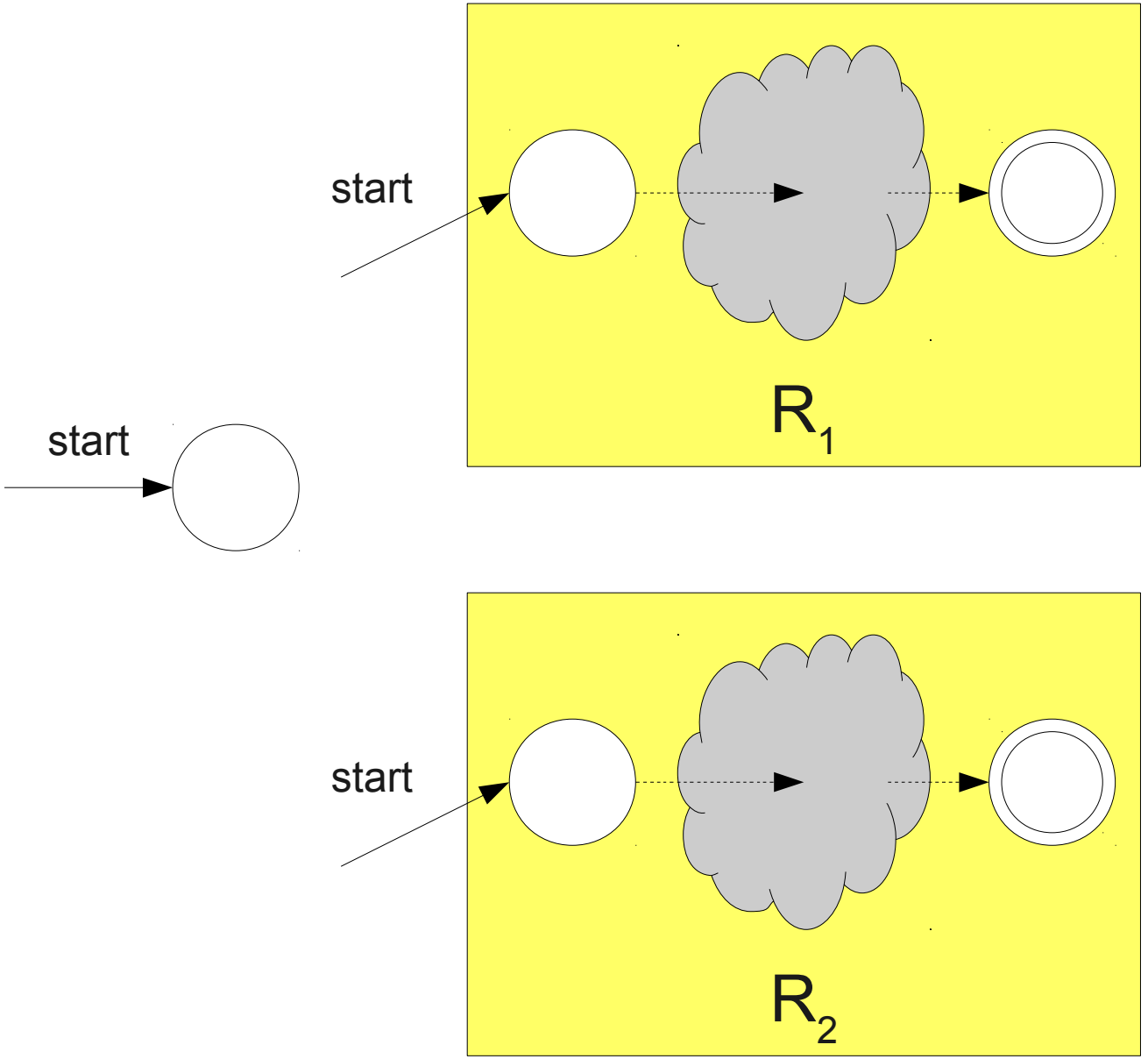


Construction for $R_1 \mid R_2$

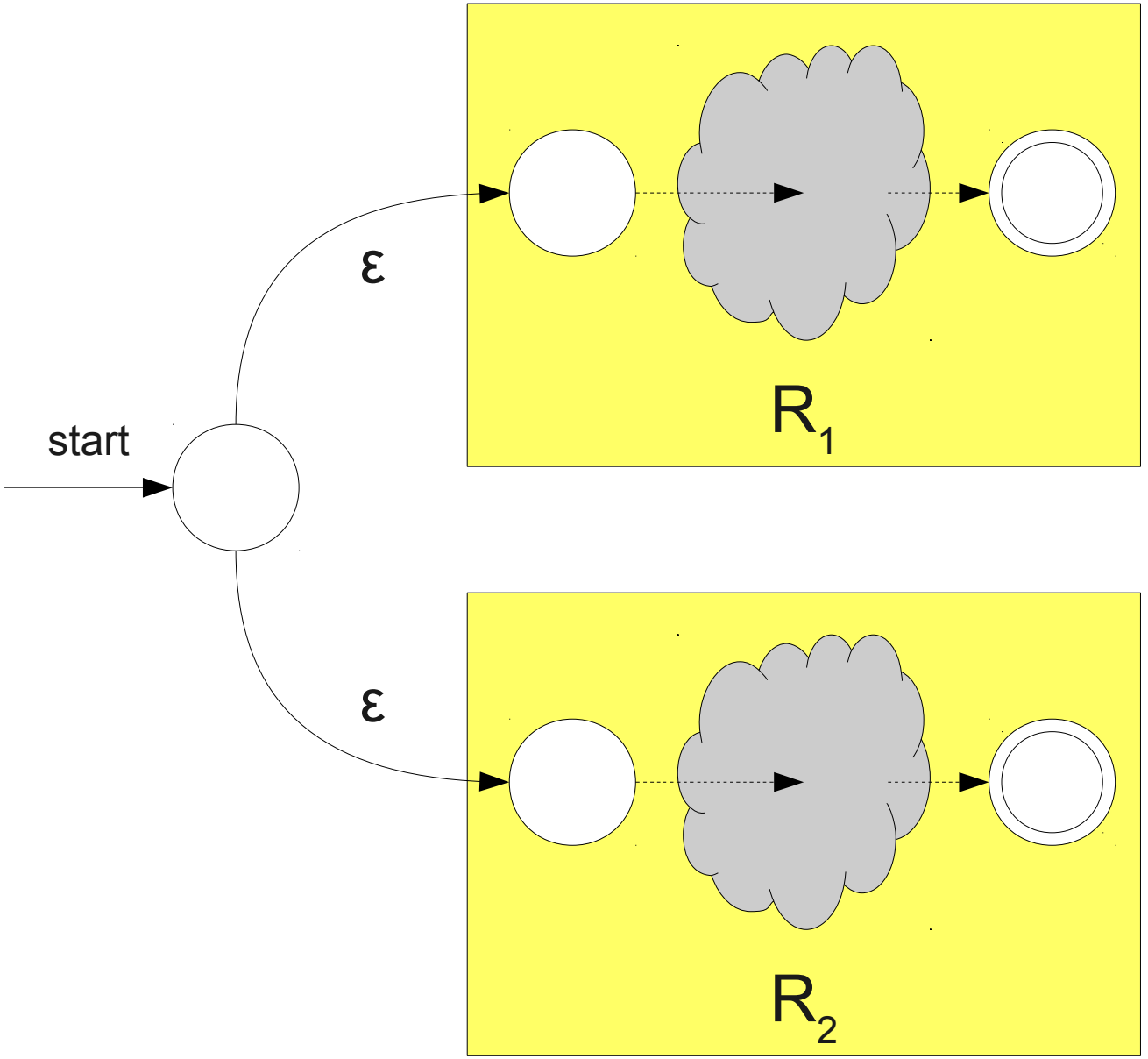
Construction for $R_1 \mid R_2$



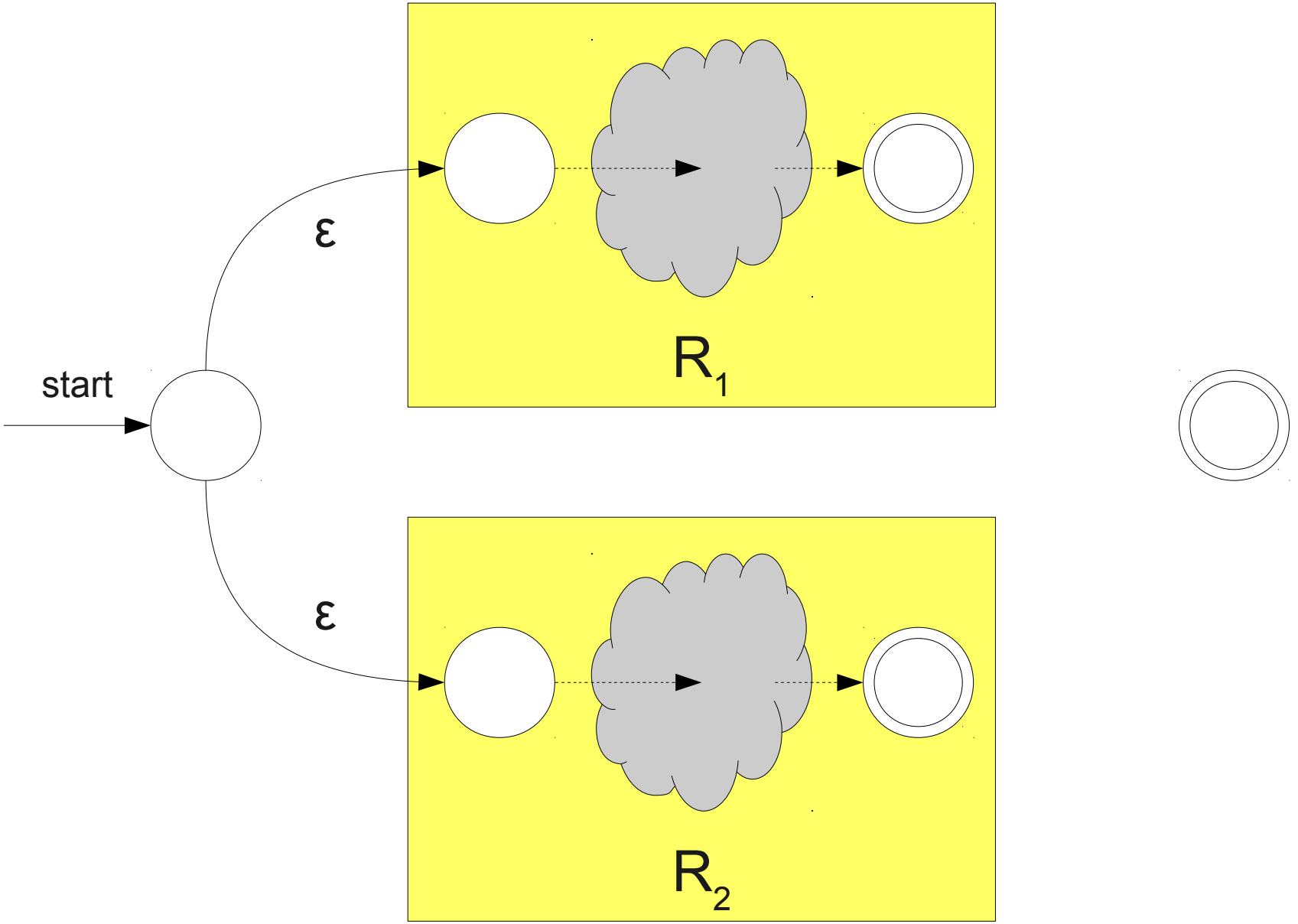
Construction for $R_1 \mid R_2$



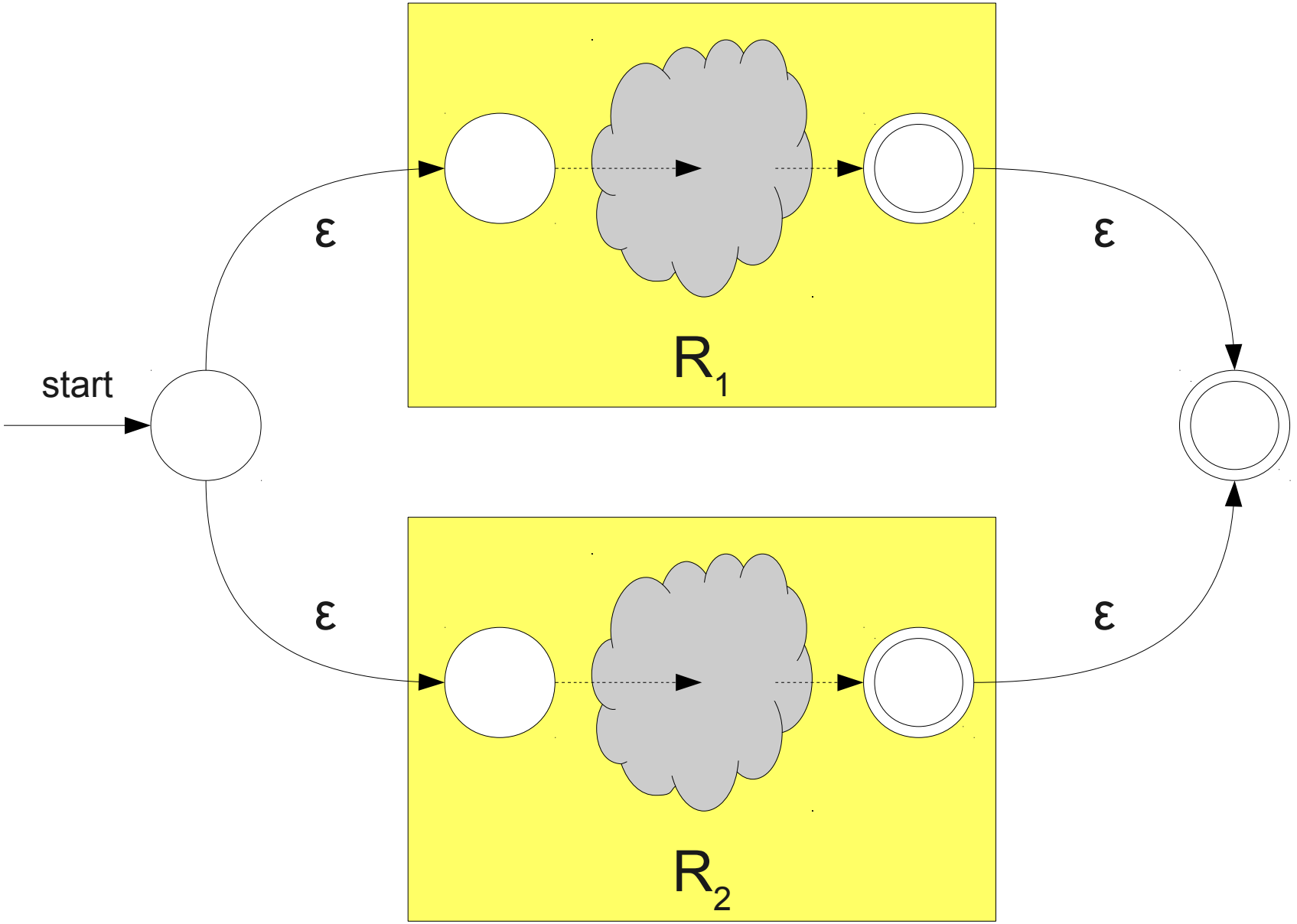
Construction for $R_1 \mid R_2$



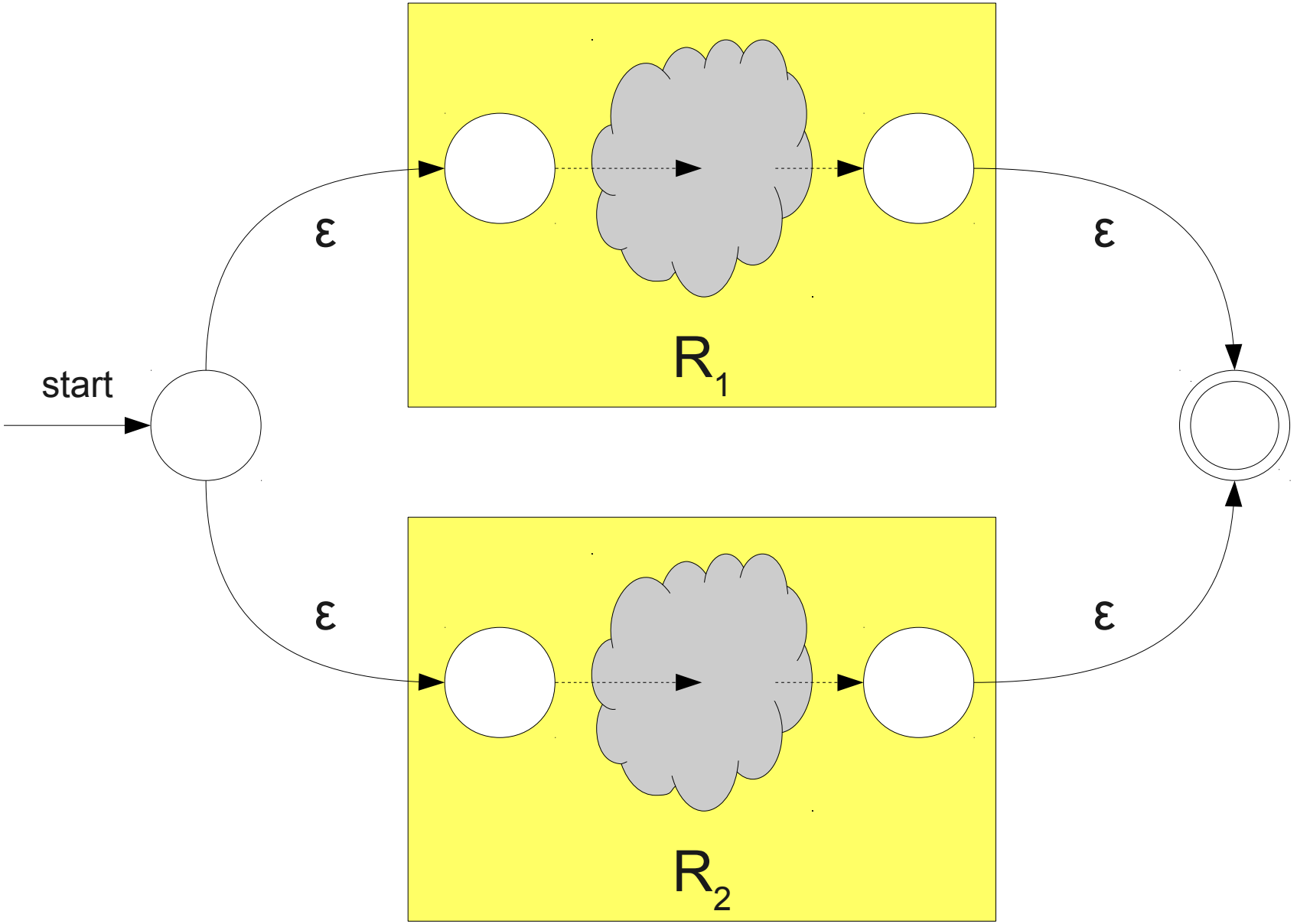
Construction for $R_1 \mid R_2$



Construction for $R_1 \mid R_2$

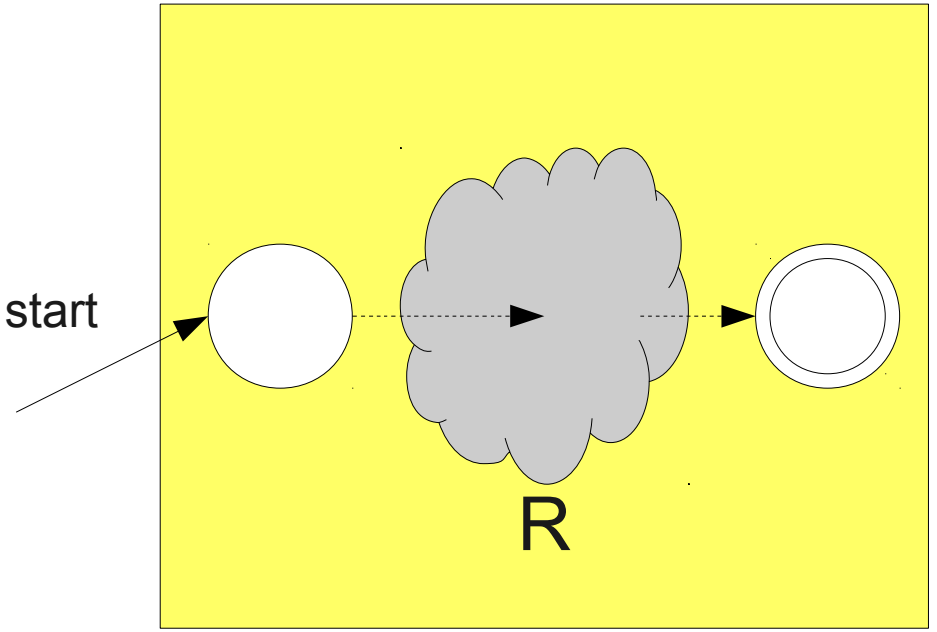


Construction for $R_1 \mid R_2$

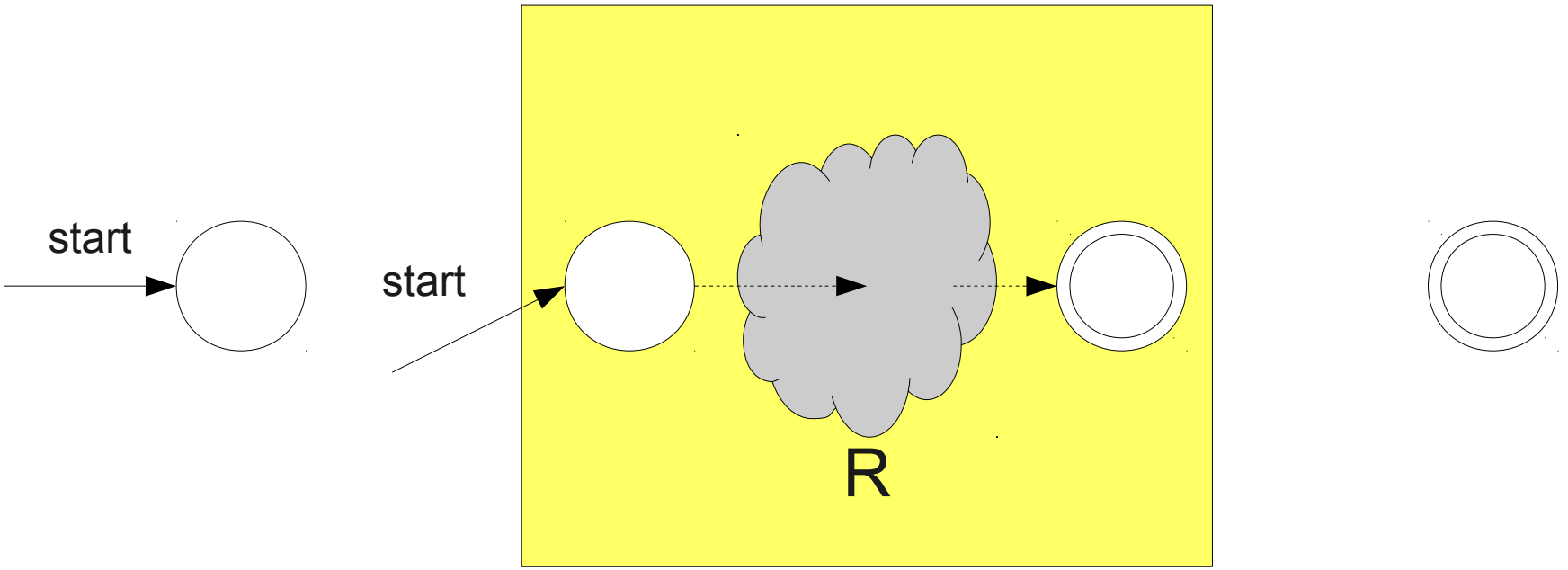


Construction for R*

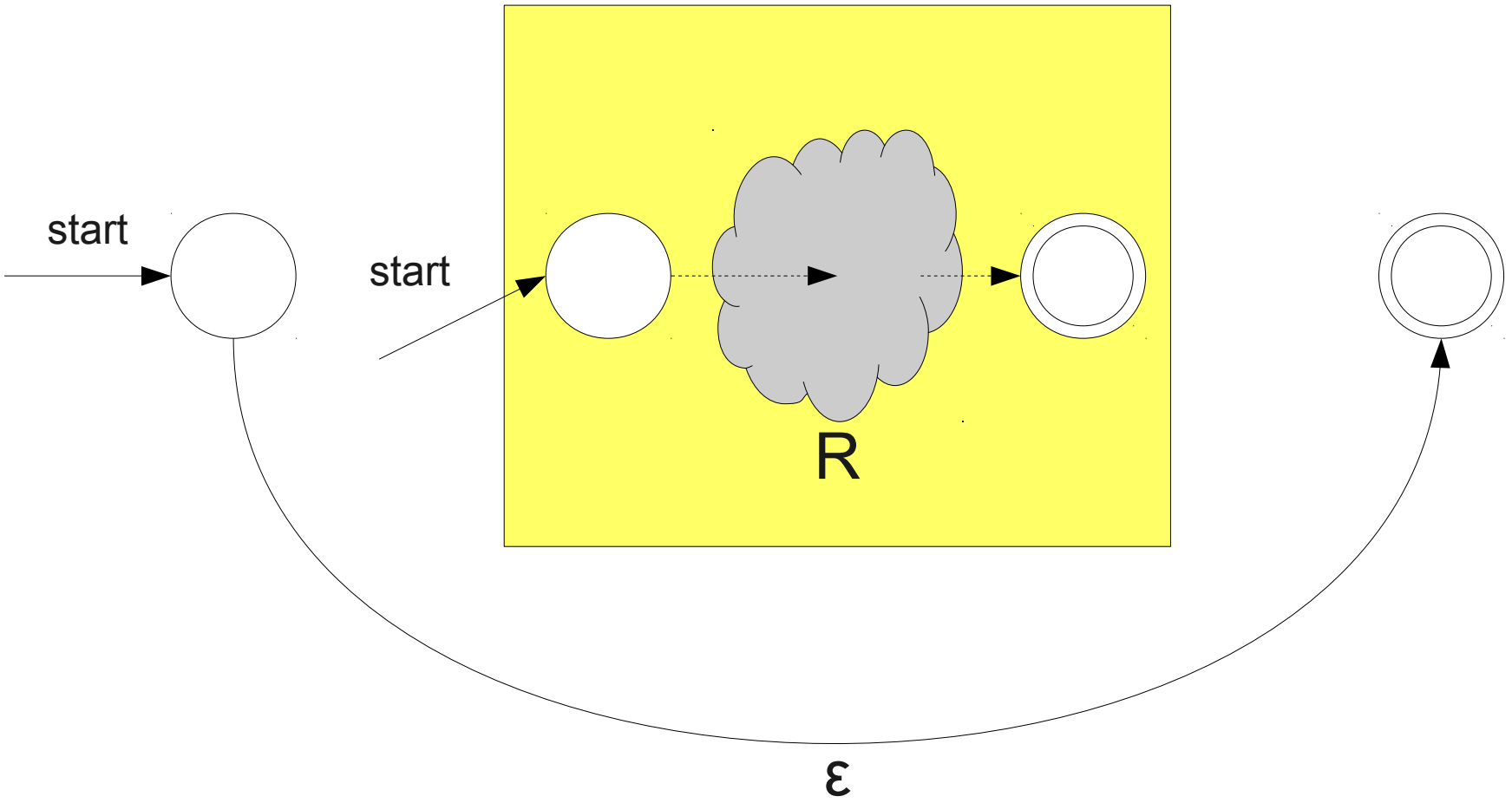
Construction for R*



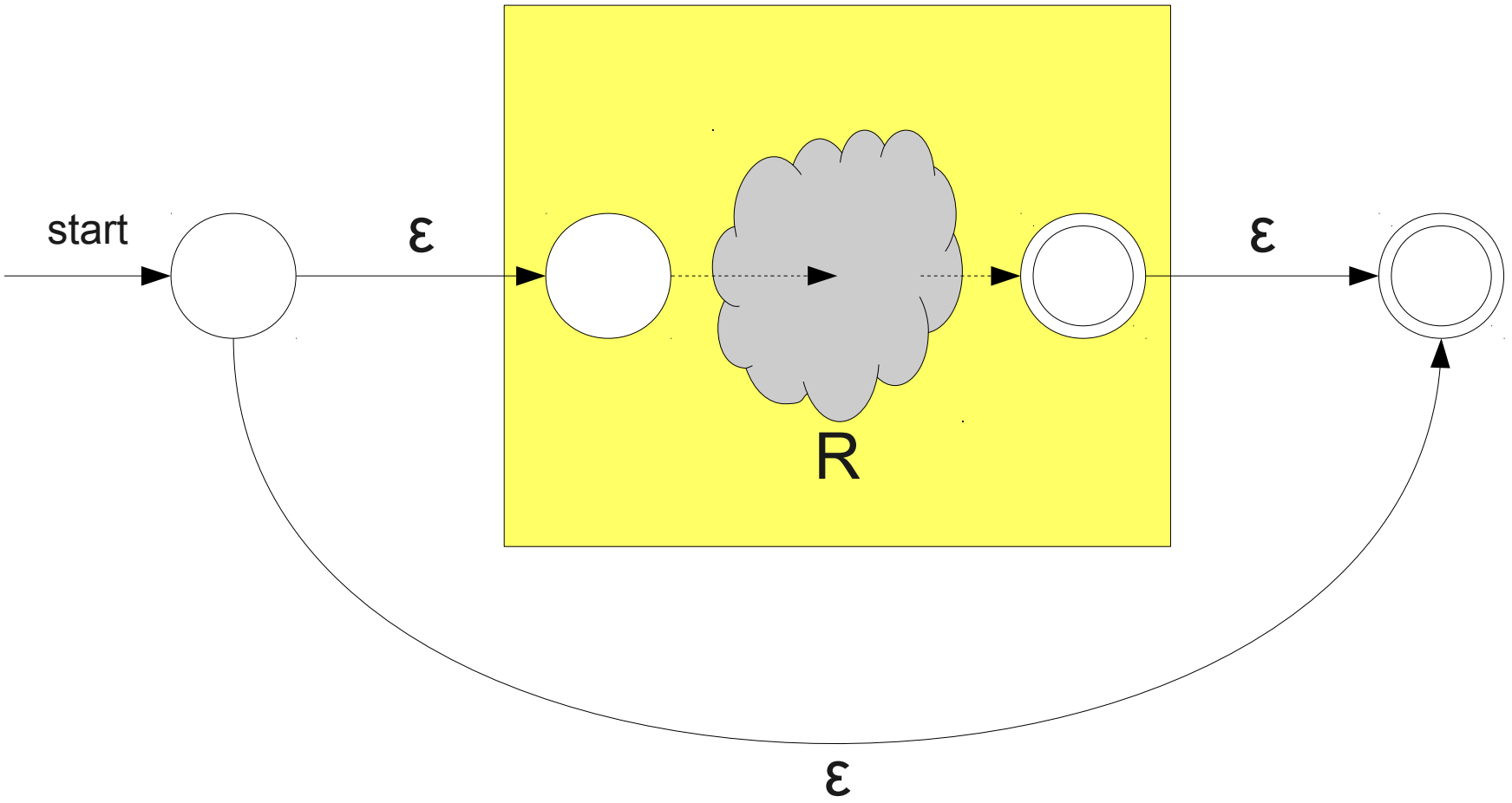
Construction for R*



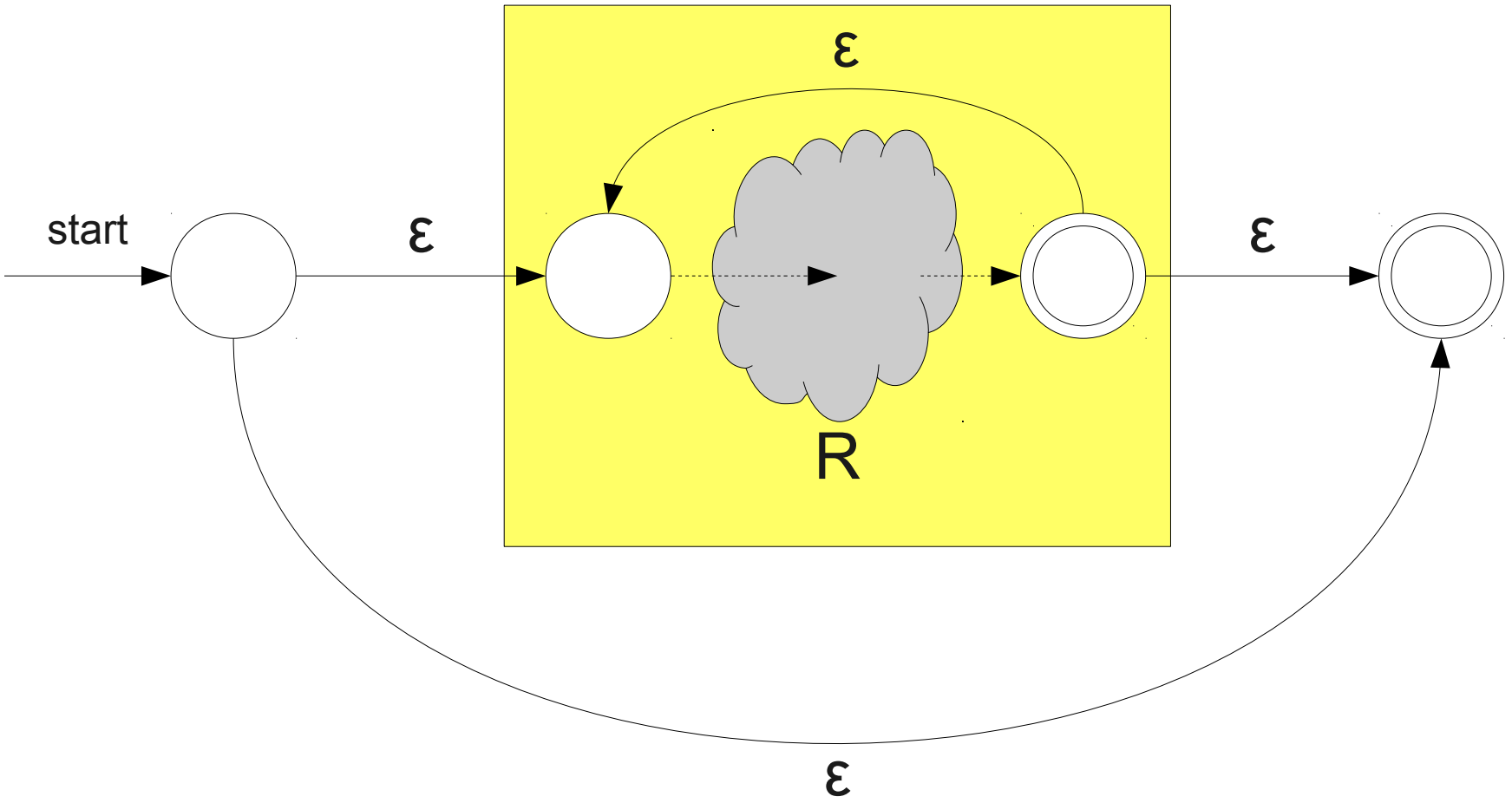
Construction for R*



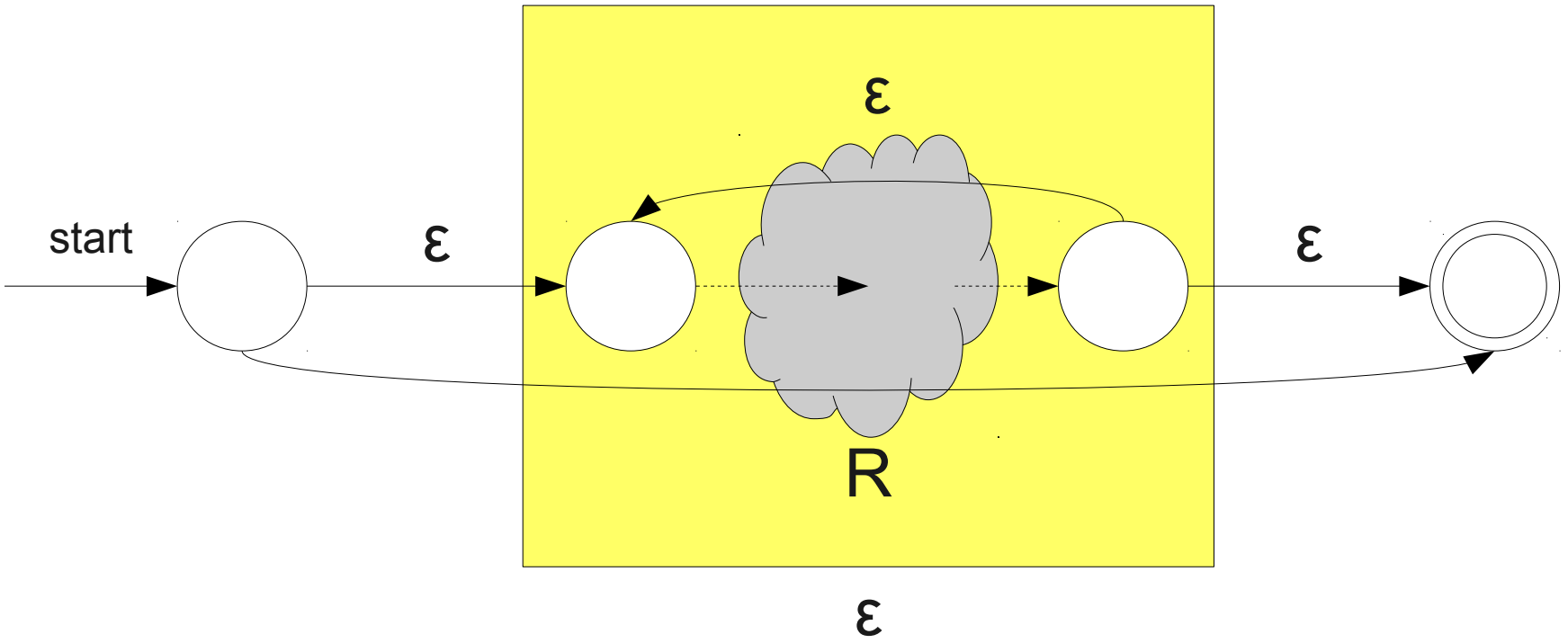
Construction for R*



Construction for R*



Construction for R*



Overall Result

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

The Story so Far

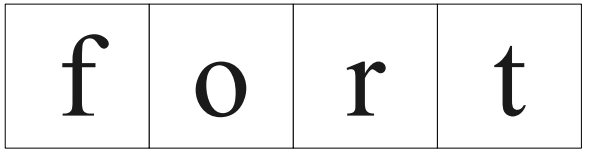
- We have a way of describing sets of strings for each token using regular expressions.
- We have an implementation of regular expressions using NFAs.
- How do we cut apart the input text?

Lexing Ambiguities

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

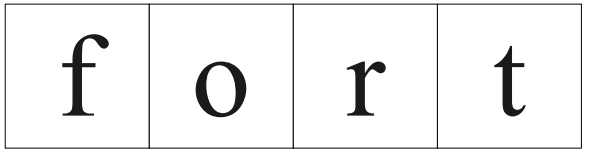
Lexing Ambiguities

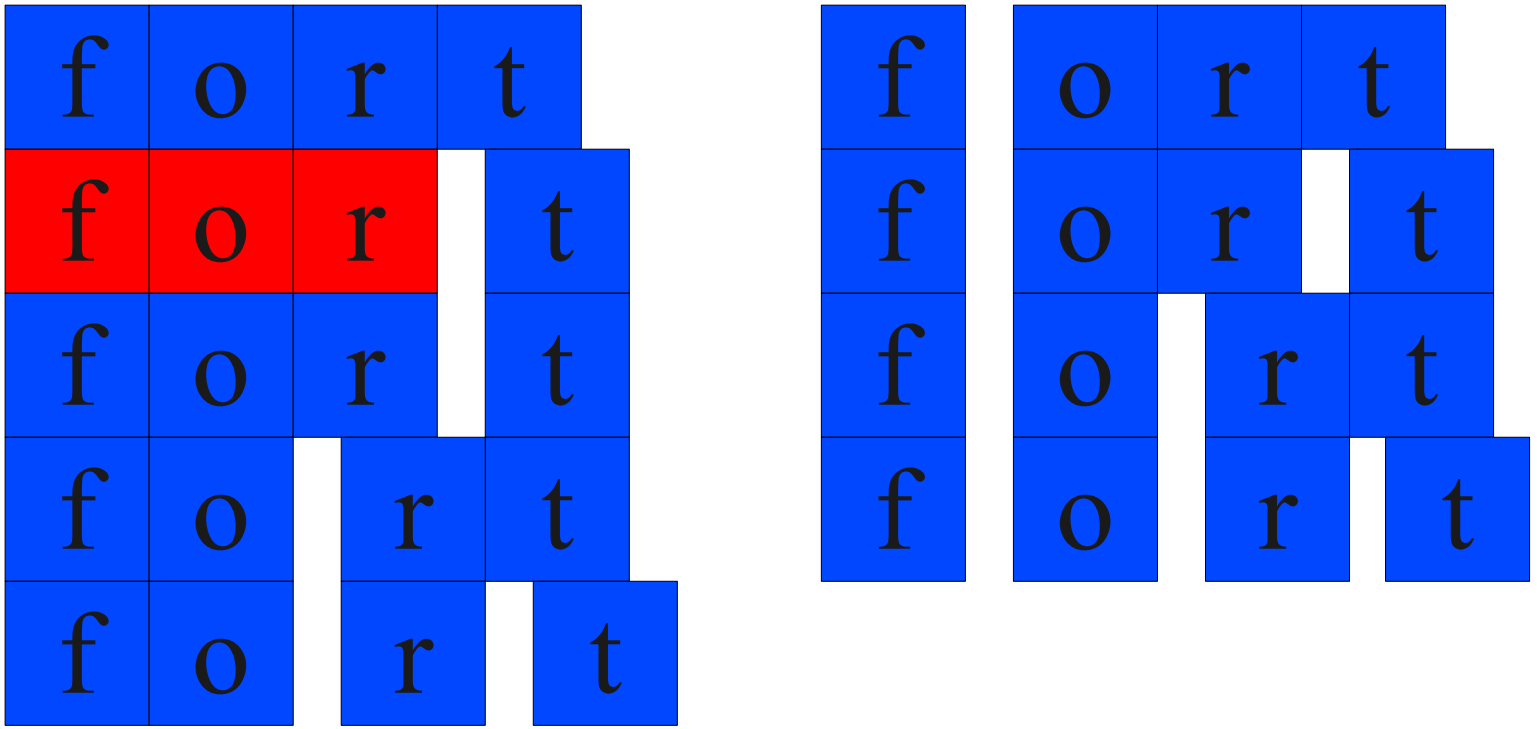
```
T_For              for
T_Identifier       [A-Za-z_][A-Za-z0-9_]*
```



Lexing Ambiguities

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



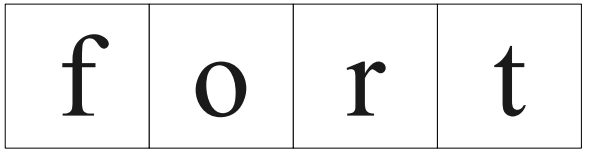


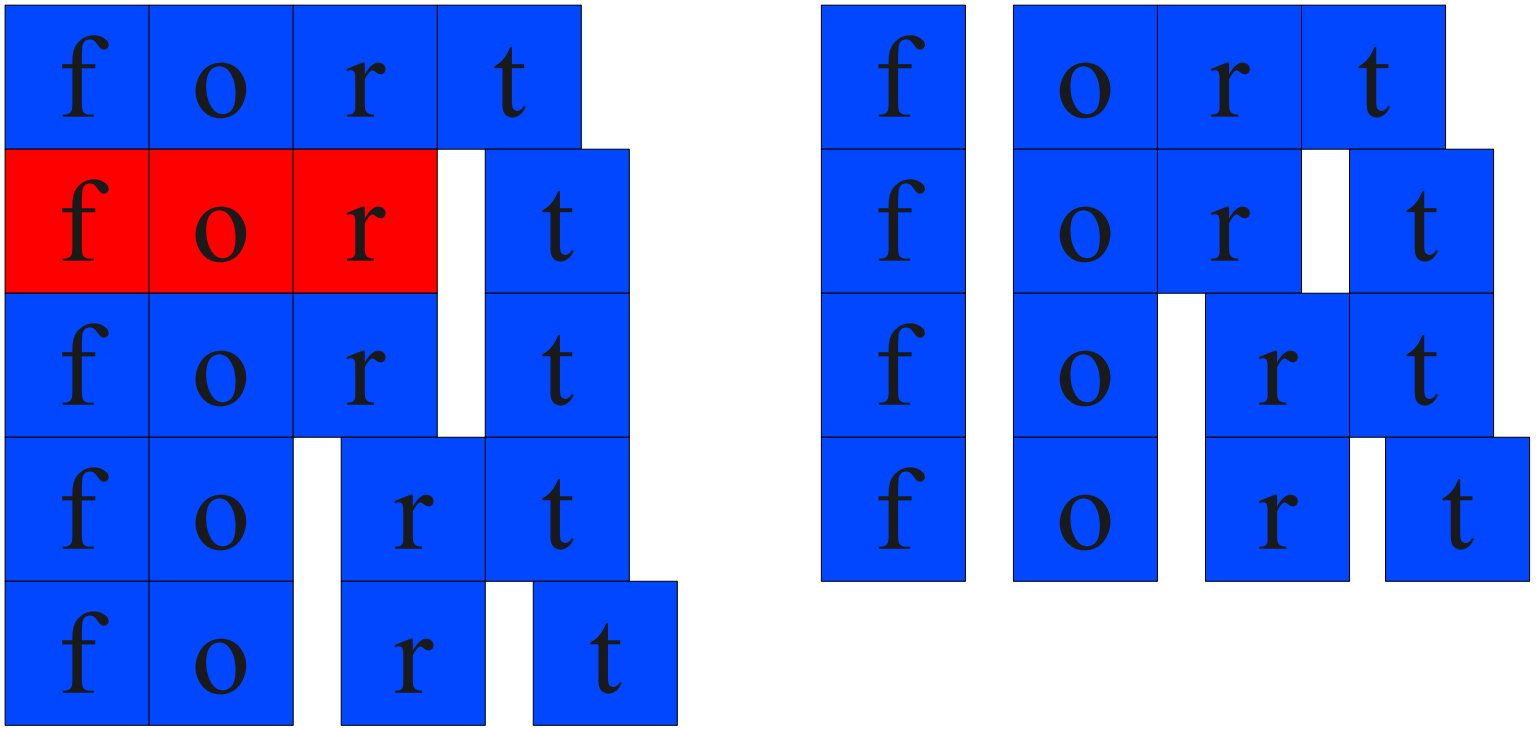
Conflict Resolution

- Assume all tokens are specified as regular expressions.
- Algorithm: **Left-to-right scan**.
- Tiebreaking rule one: **Maximal munch**.
  - Always match the longest possible prefix of the remaining text.

Lexing Ambiguities

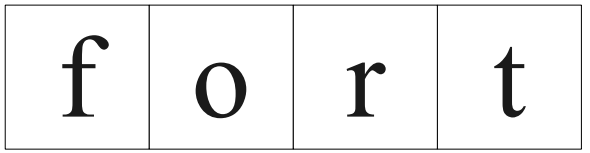
```
T_For                for
T_Identifier         [A-Za-z_][A-Za-z0-9_]*
```

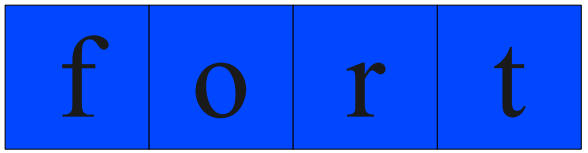




Lexing Ambiguities

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





Implementing Maximal Munch

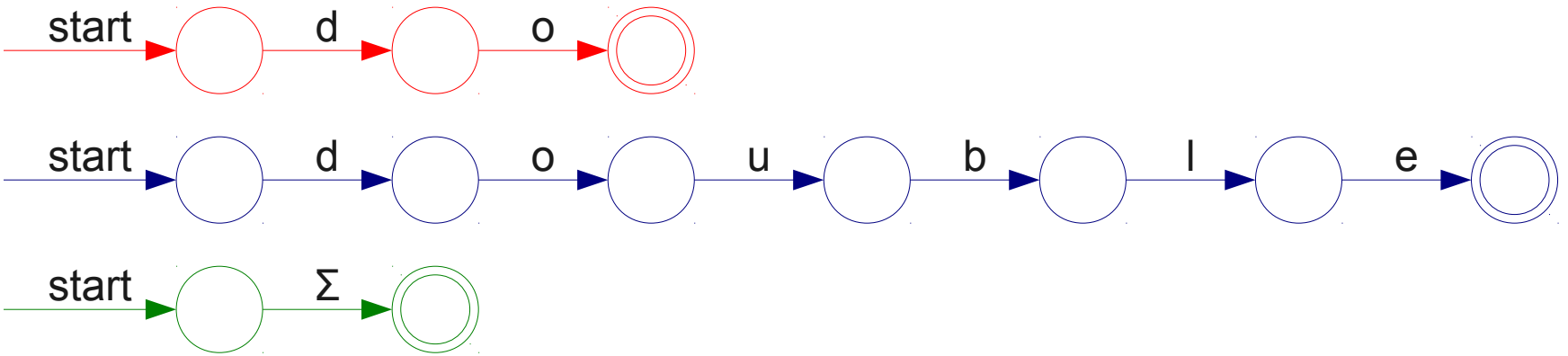
- Given a set of regular expressions, how can we use them to implement maximum munch?
- Idea:
  - Convert expressions to NFAs.
  - Run all NFAs in parallel, keeping track of the last match.
  - When all automata get stuck, report the last match and restart the search at that point.

Implementing Maximal Munch

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

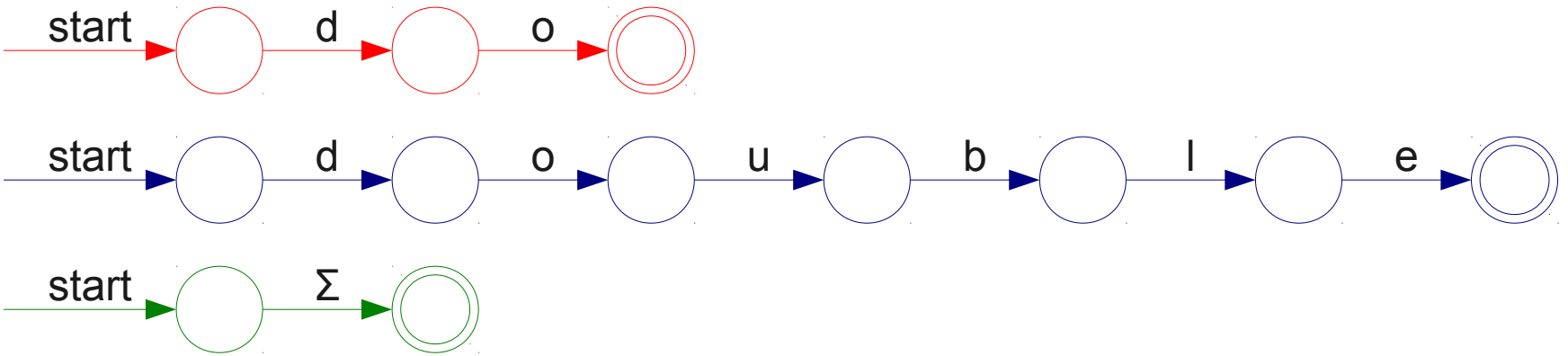
Implementing Maximal Munch

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



Implementing Maximal Munch

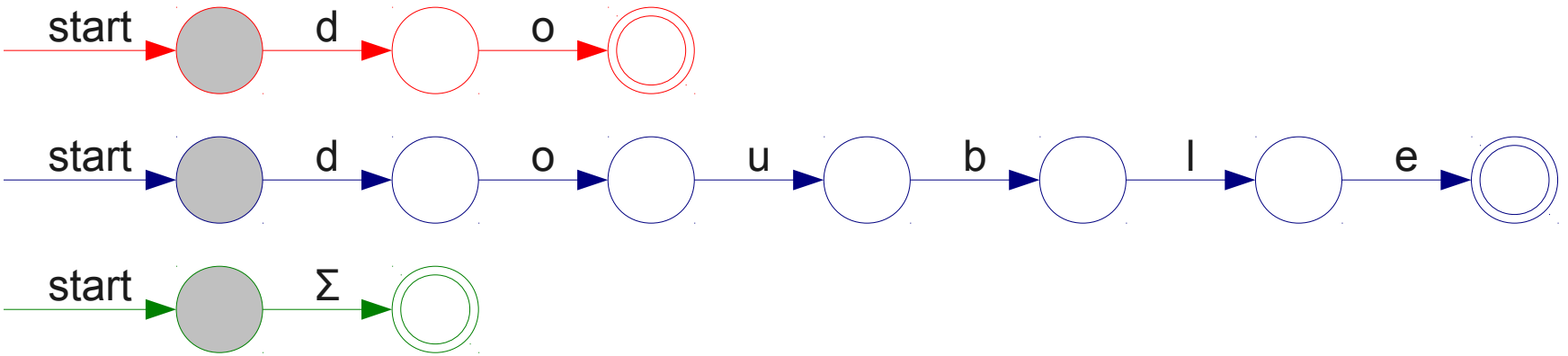
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```



D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

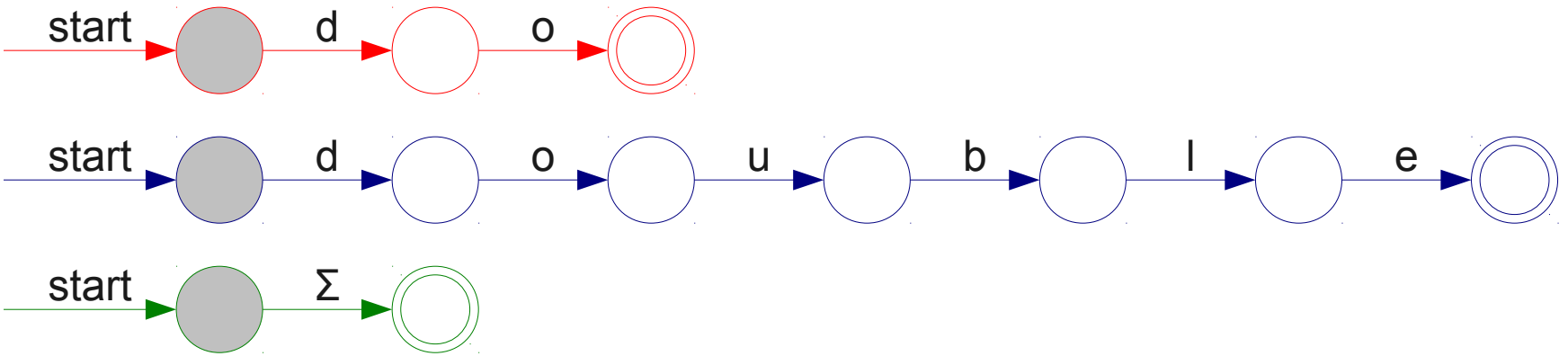
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

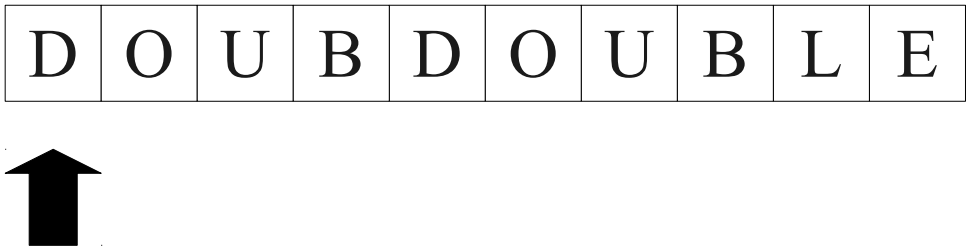


D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

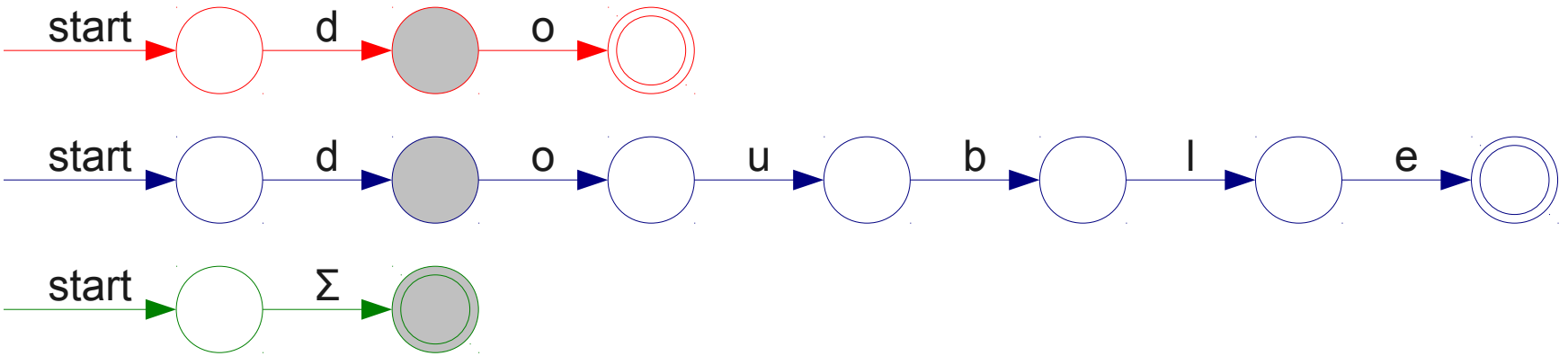
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```





Implementing Maximal Munch

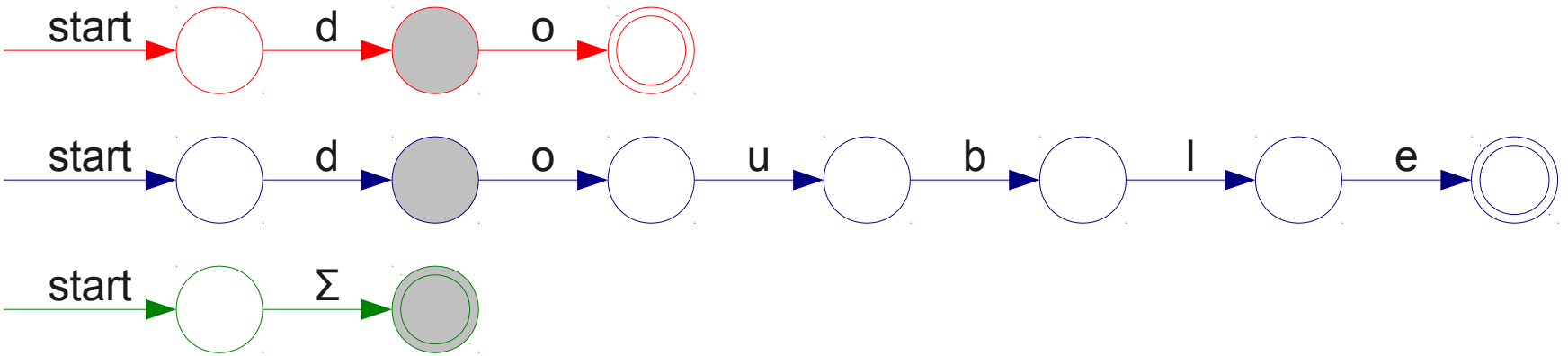
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

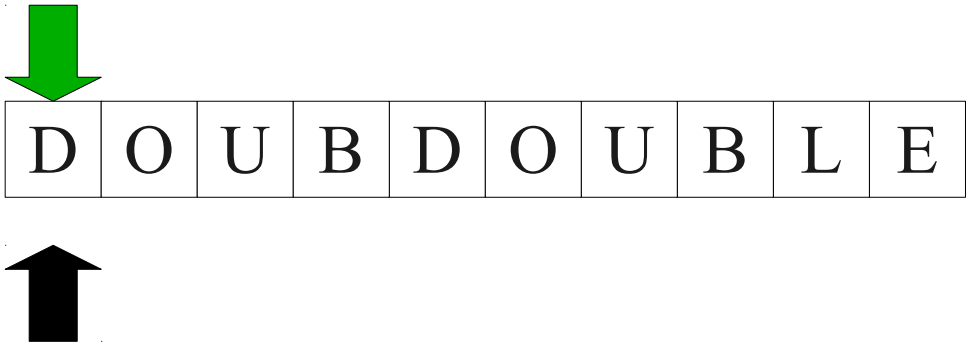


D	O	U	B	D	O	U	B	L	E

Implementing Maximal Munch

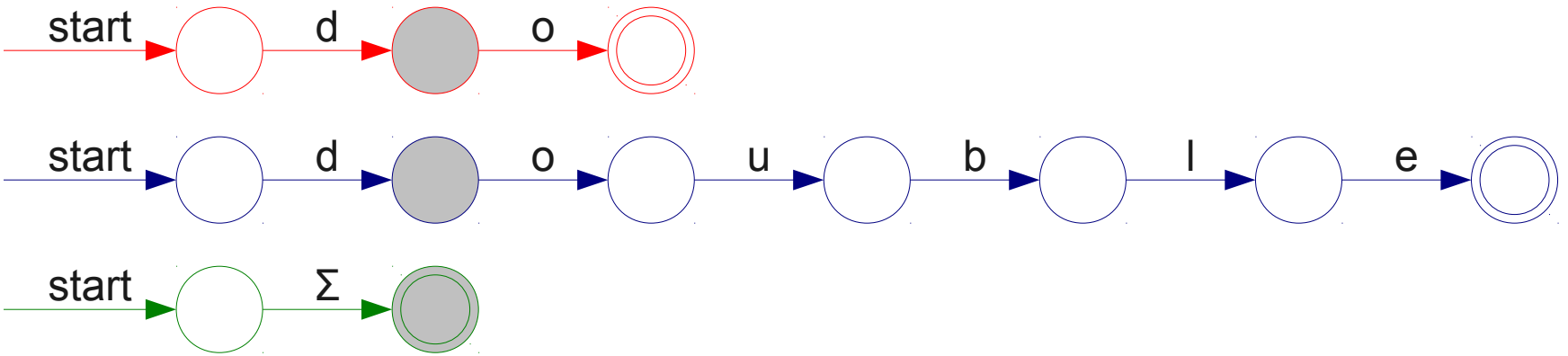
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

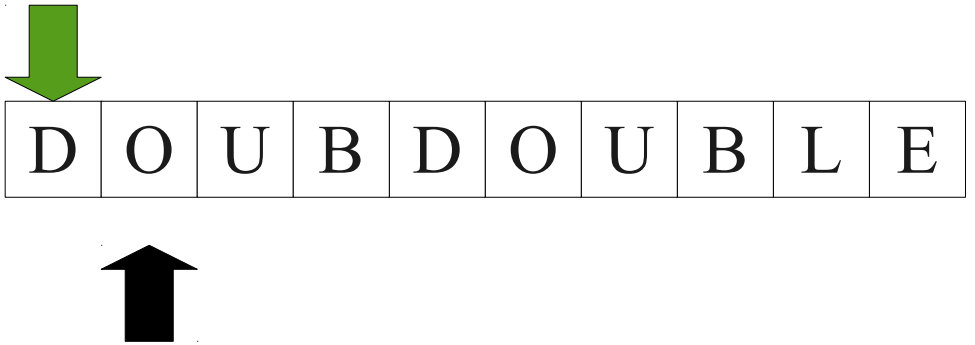




Implementing Maximal Munch

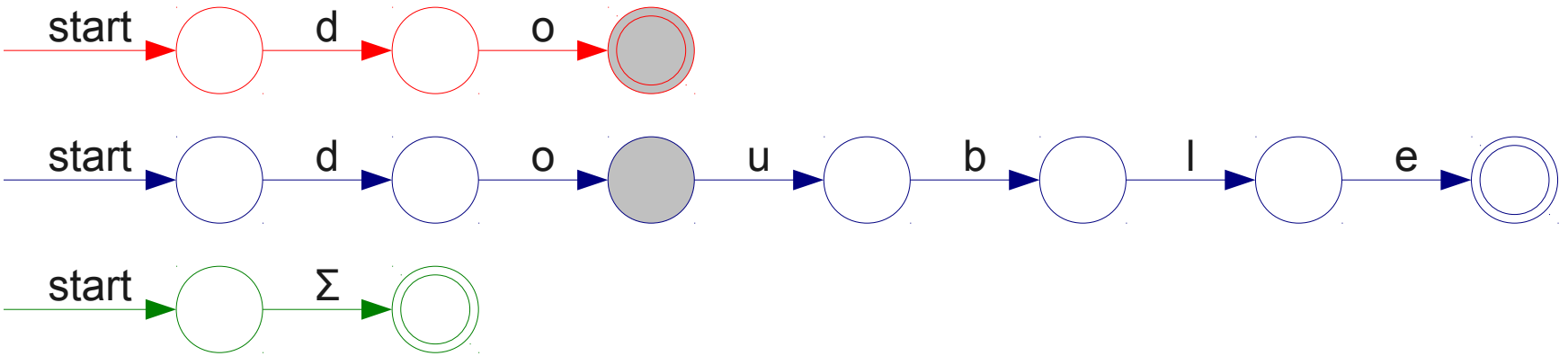
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

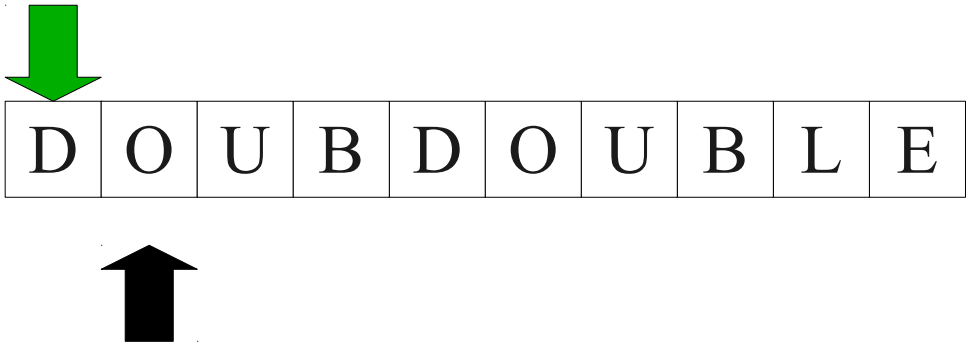




Implementing Maximal Munch

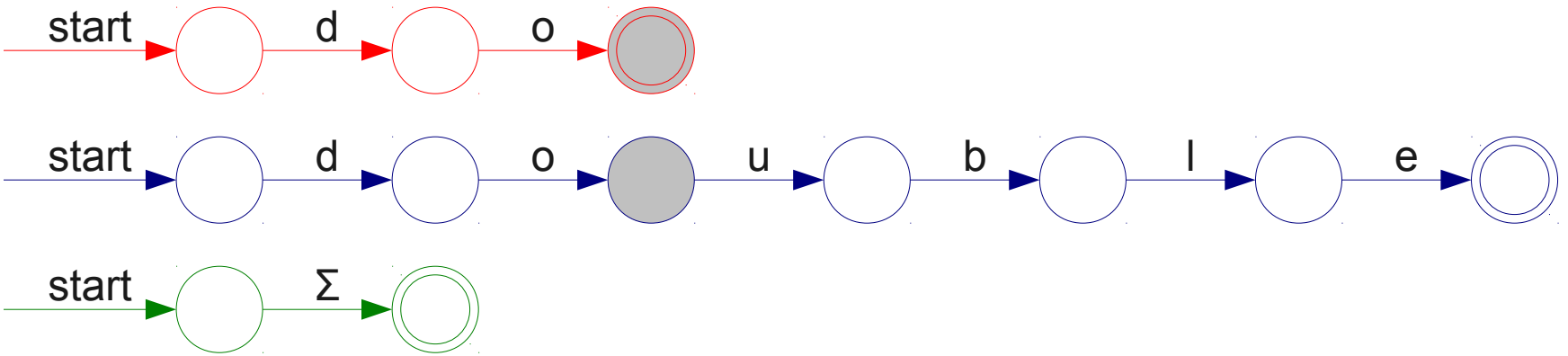
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

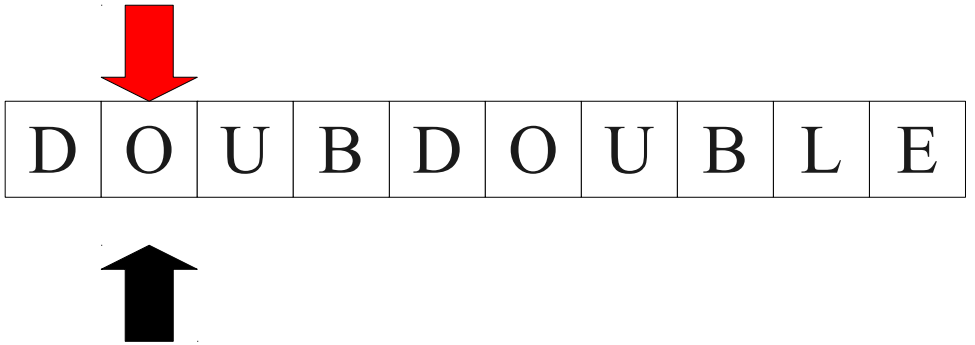




Implementing Maximal Munch

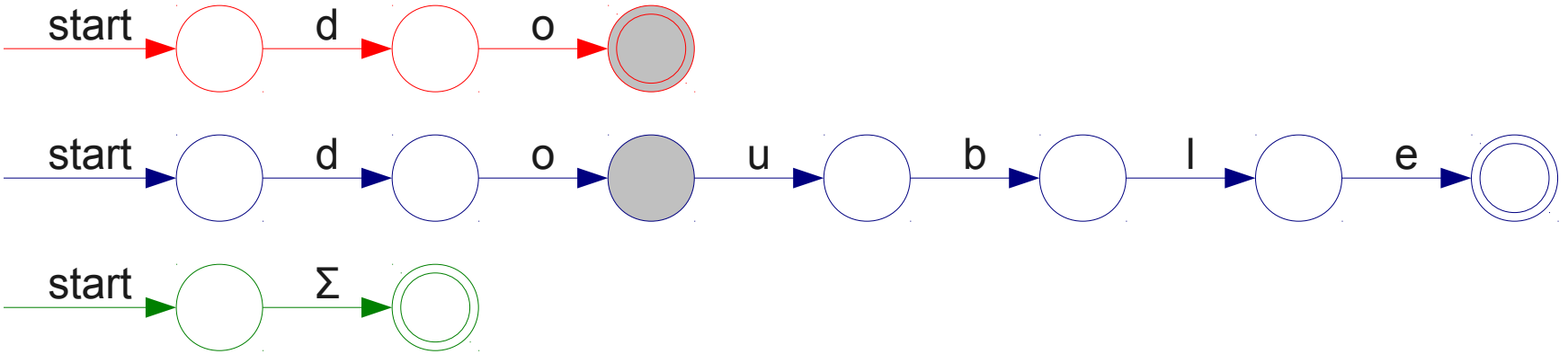
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

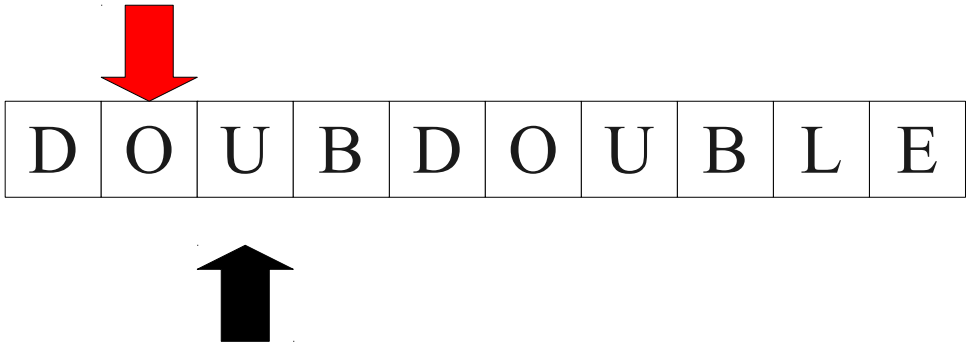




Implementing Maximal Munch

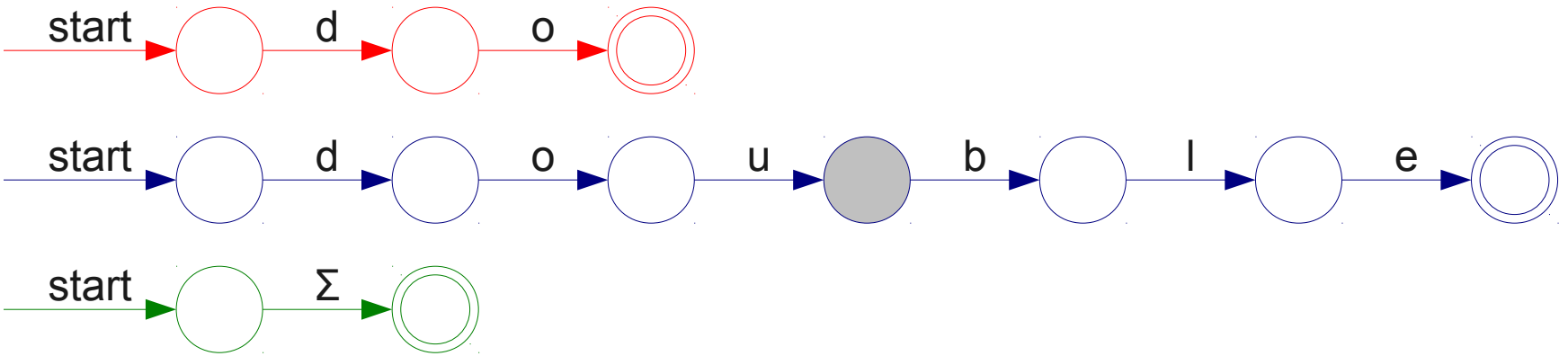
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

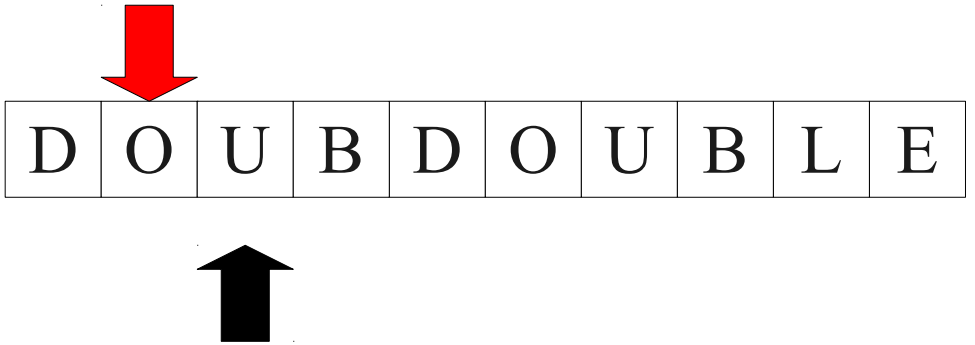




Implementing Maximal Munch

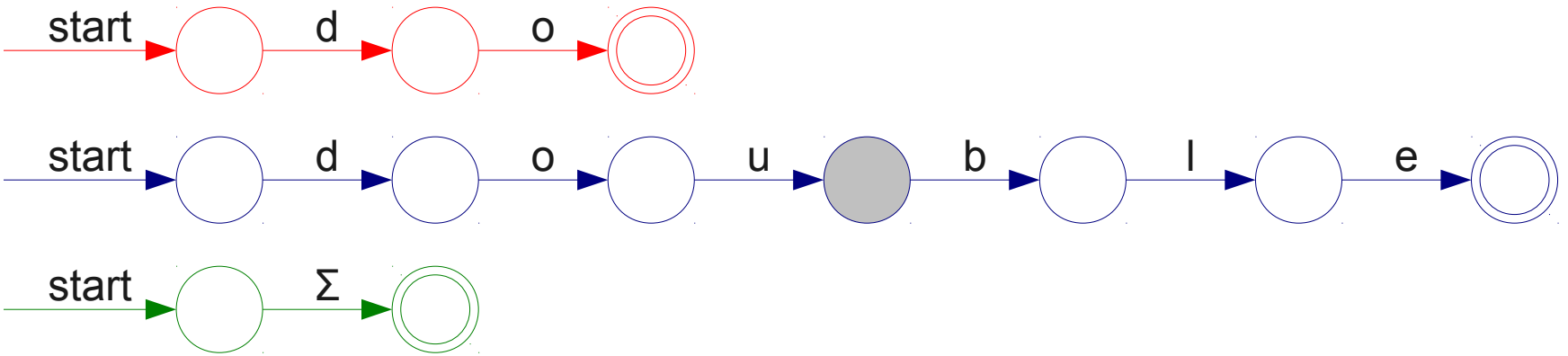
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

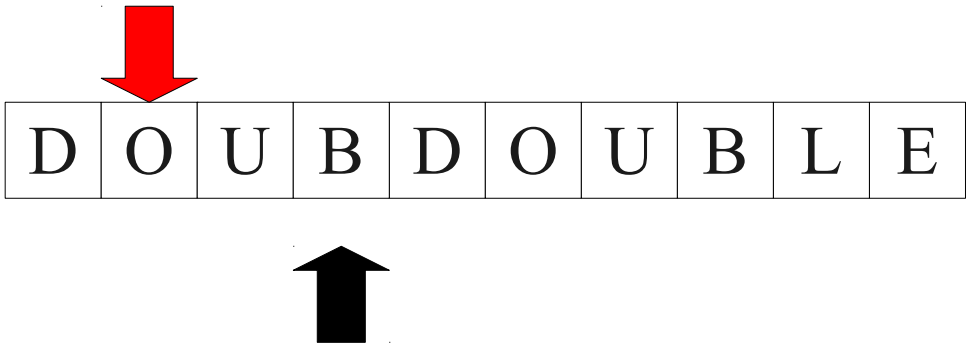




Implementing Maximal Munch

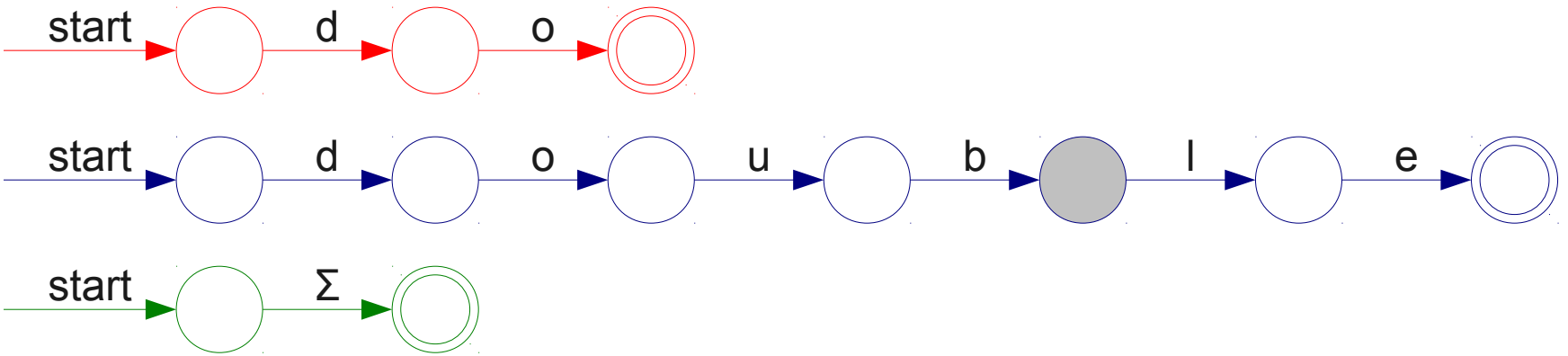
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

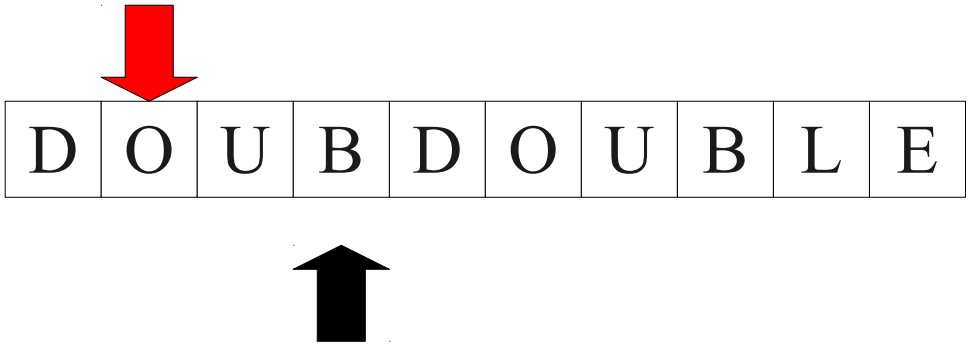




Implementing Maximal Munch

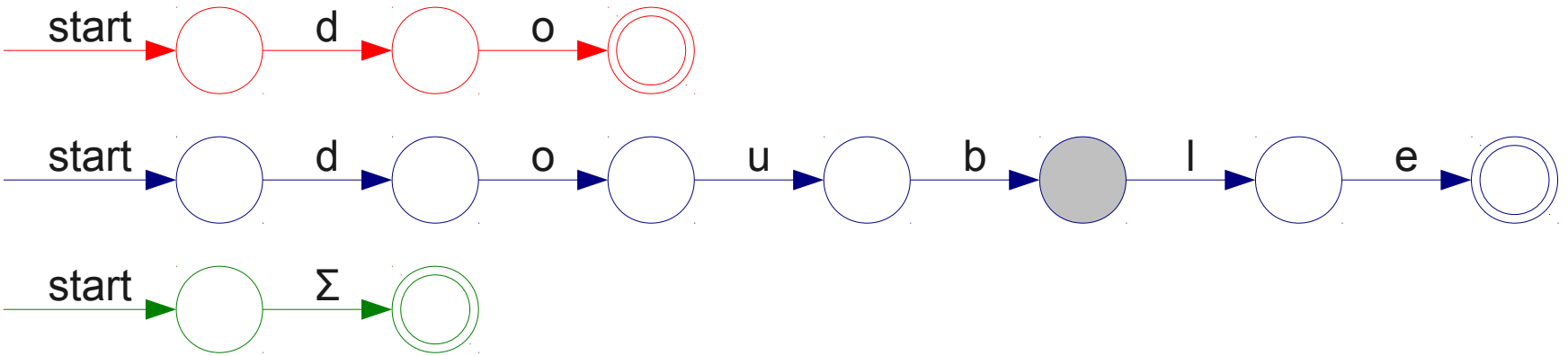
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

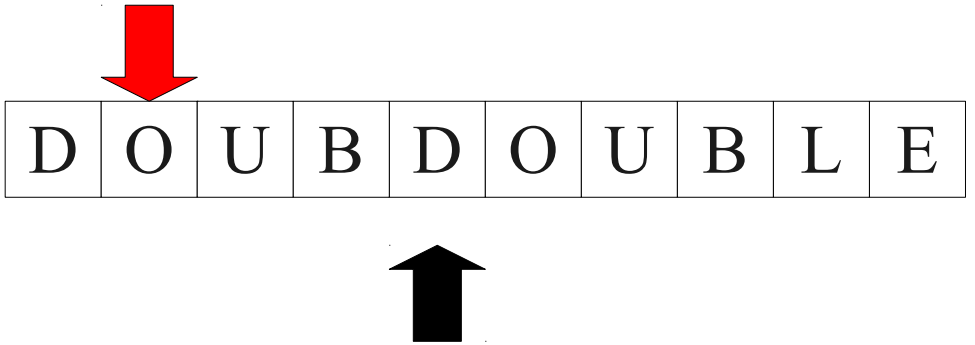




Implementing Maximal Munch

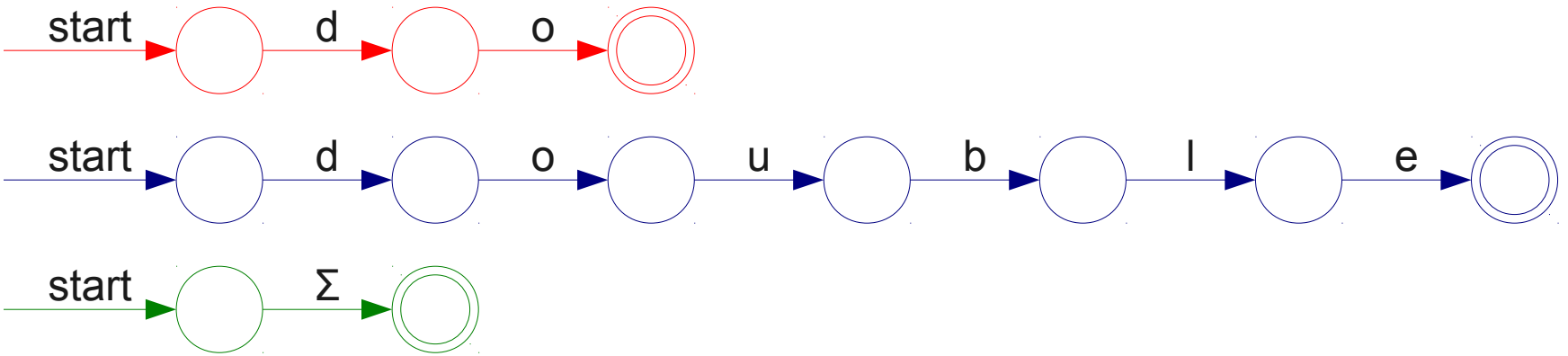
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

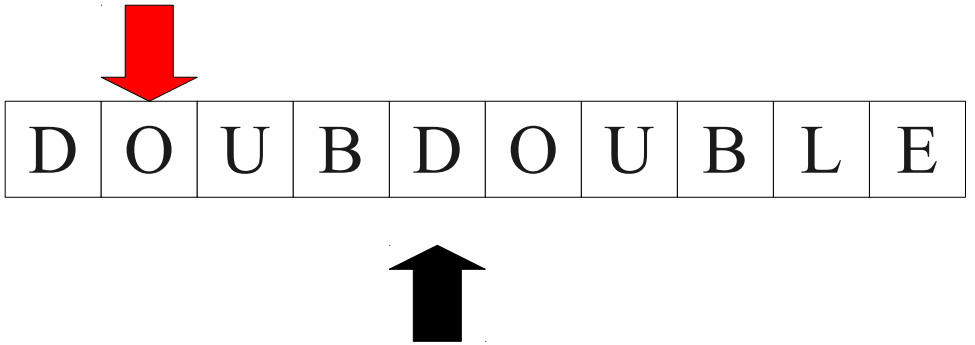




Implementing Maximal Munch

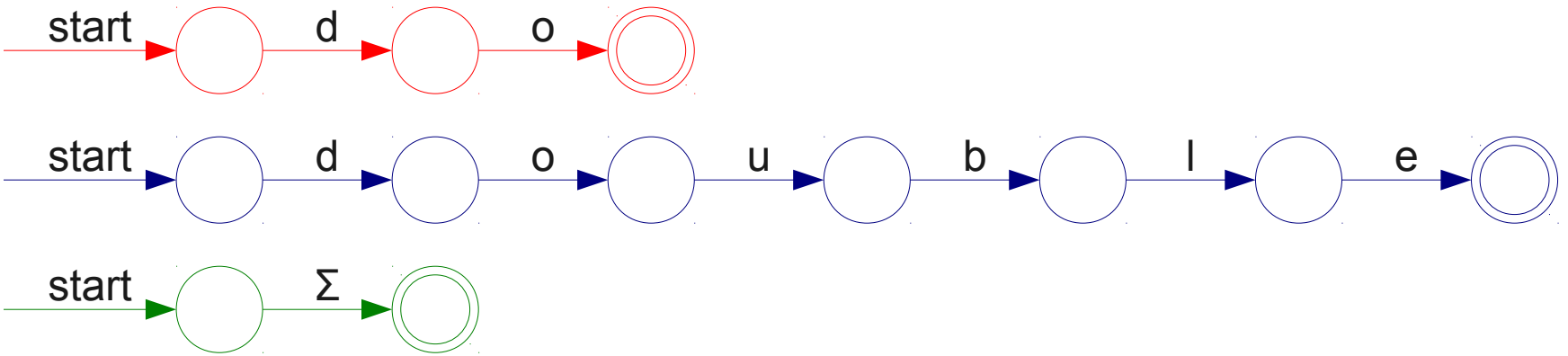
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

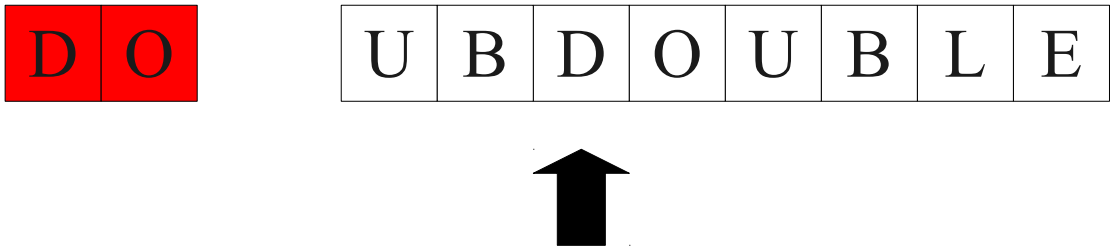




Implementing Maximal Munch

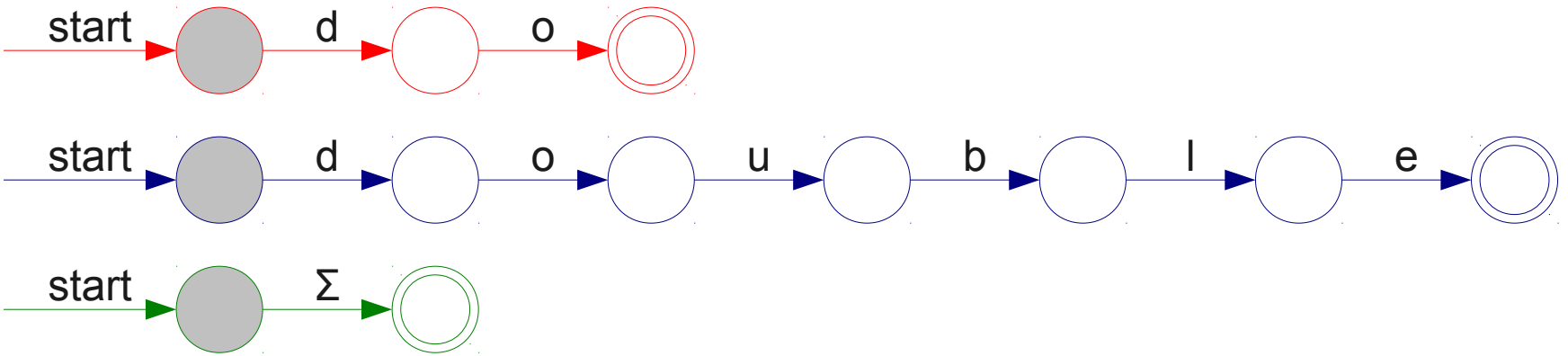
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

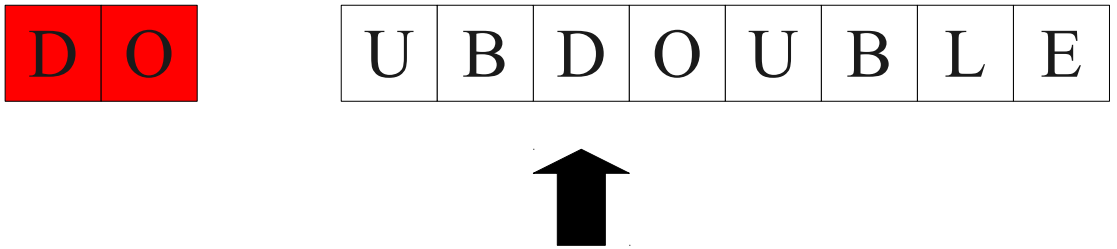




Implementing Maximal Munch

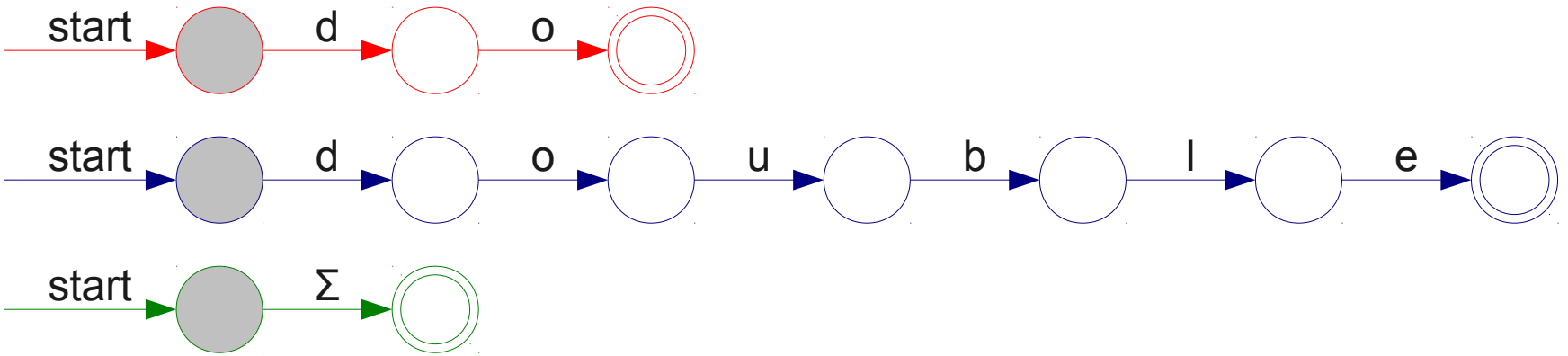
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

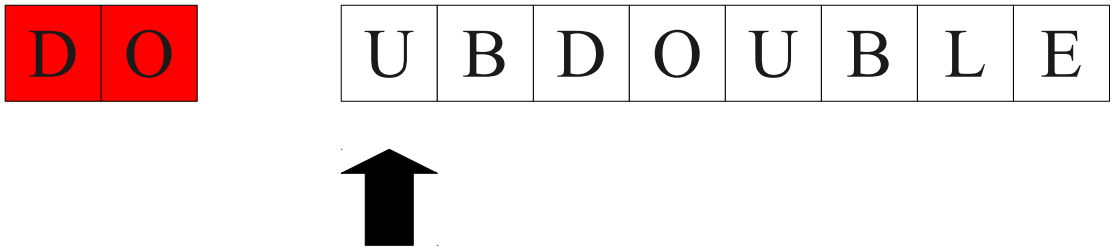




Implementing Maximal Munch

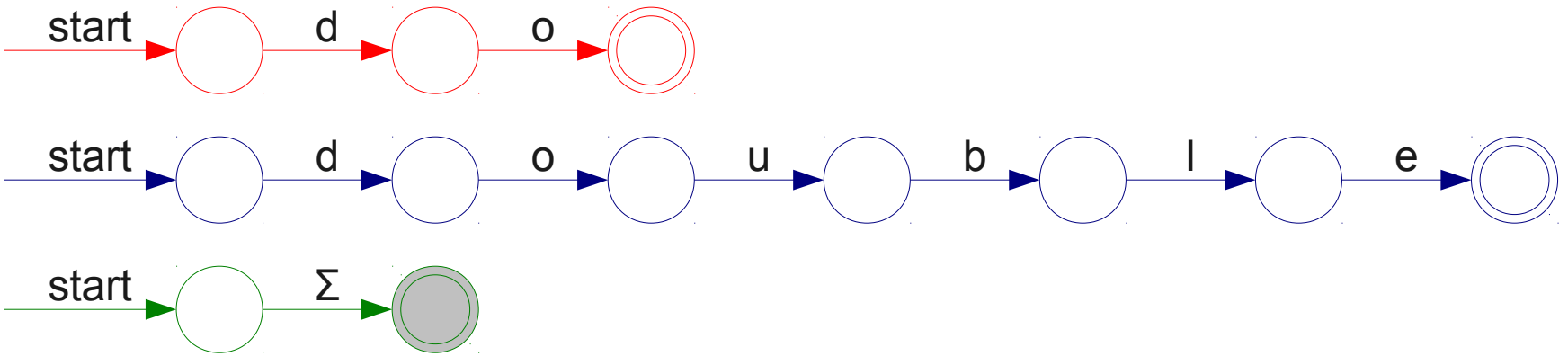
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

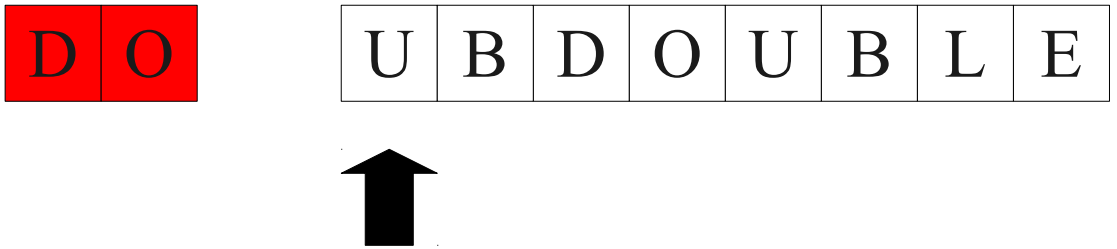




Implementing Maximal Munch

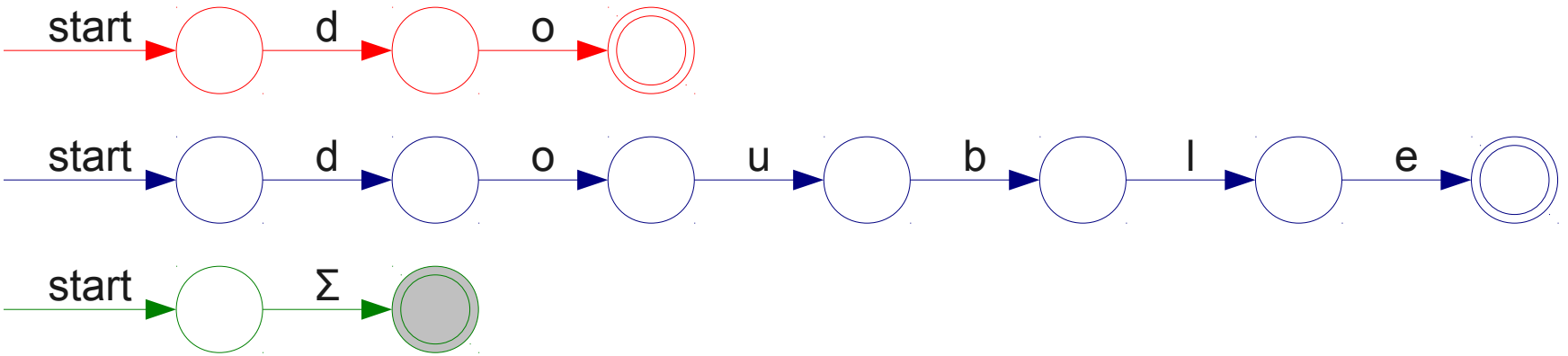
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

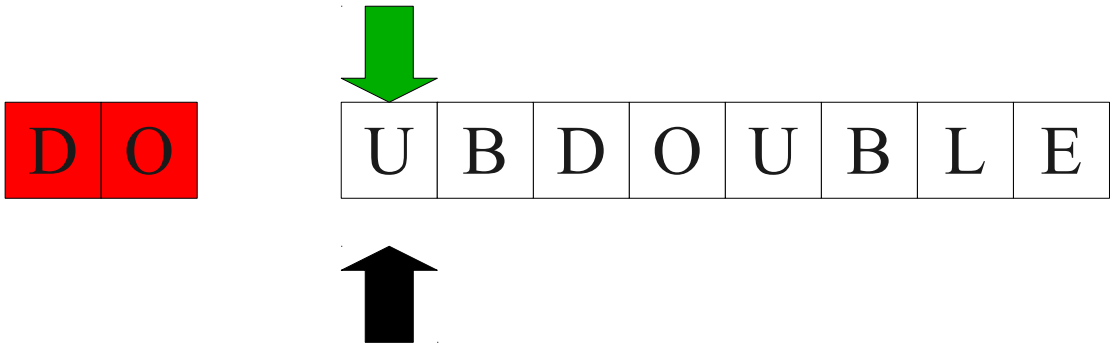




Implementing Maximal Munch

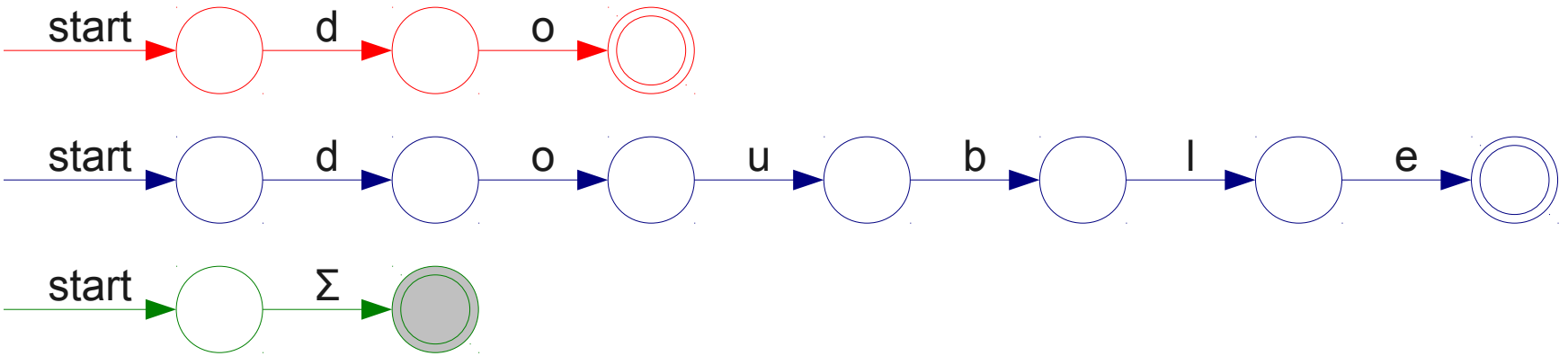
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

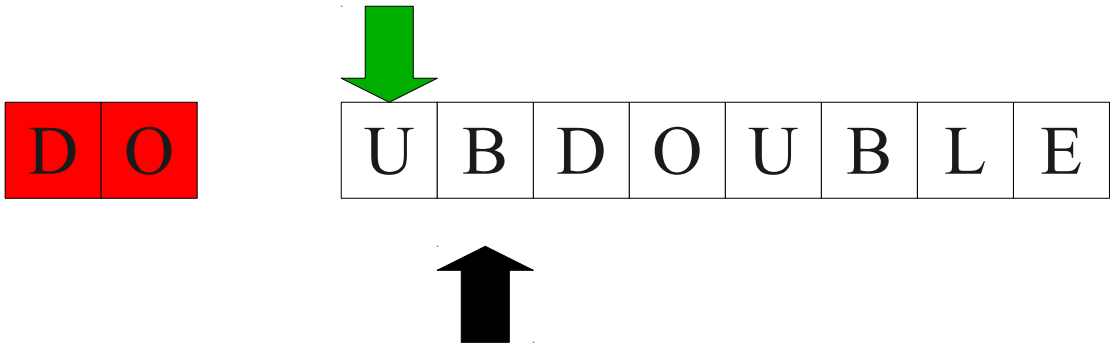




Implementing Maximal Munch

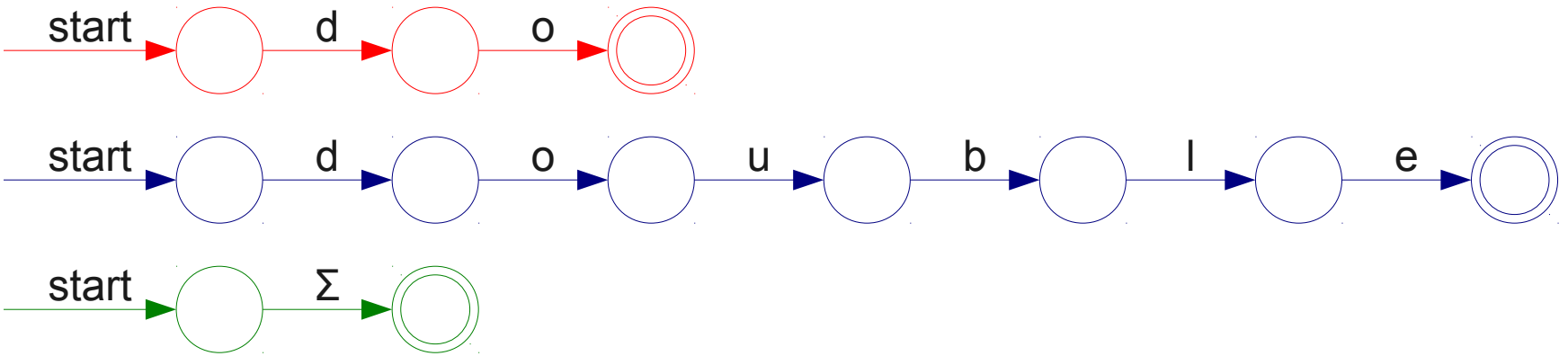
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

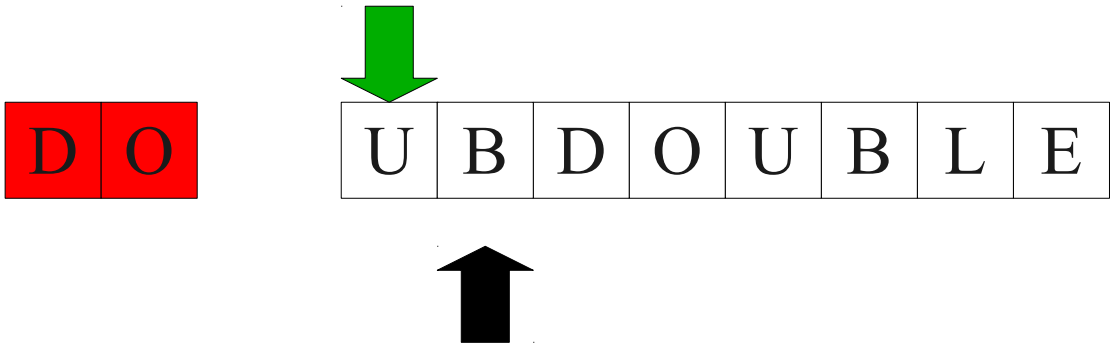




Implementing Maximal Munch

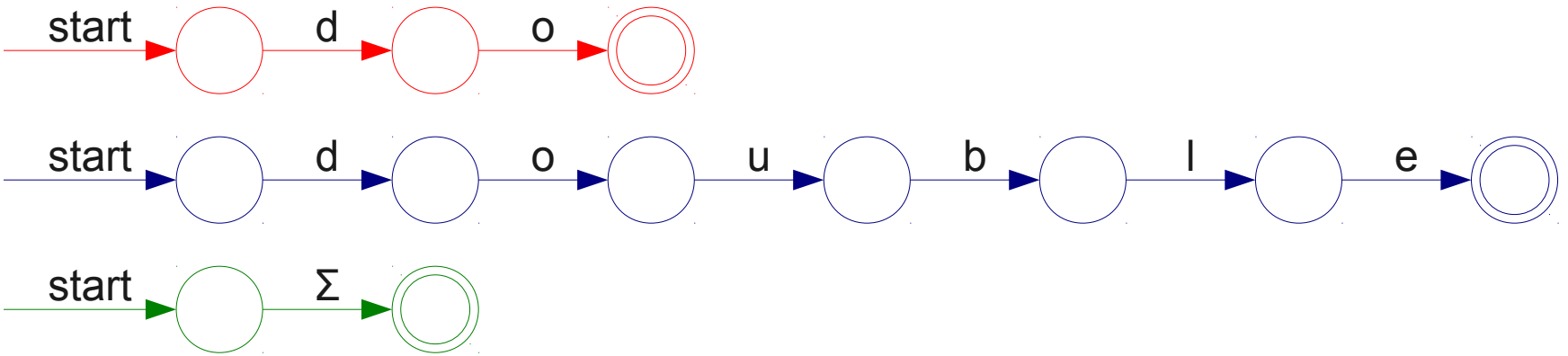
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

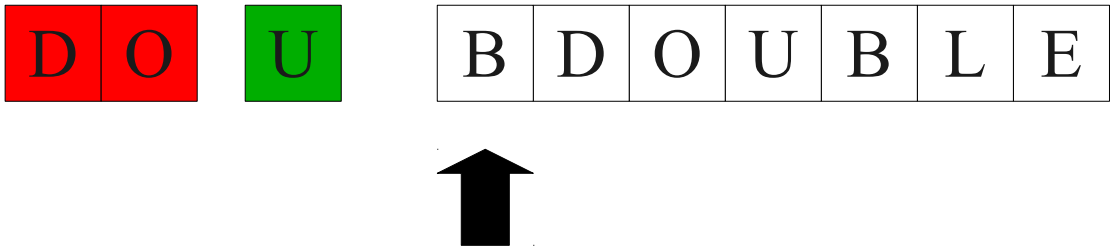




Implementing Maximal Munch

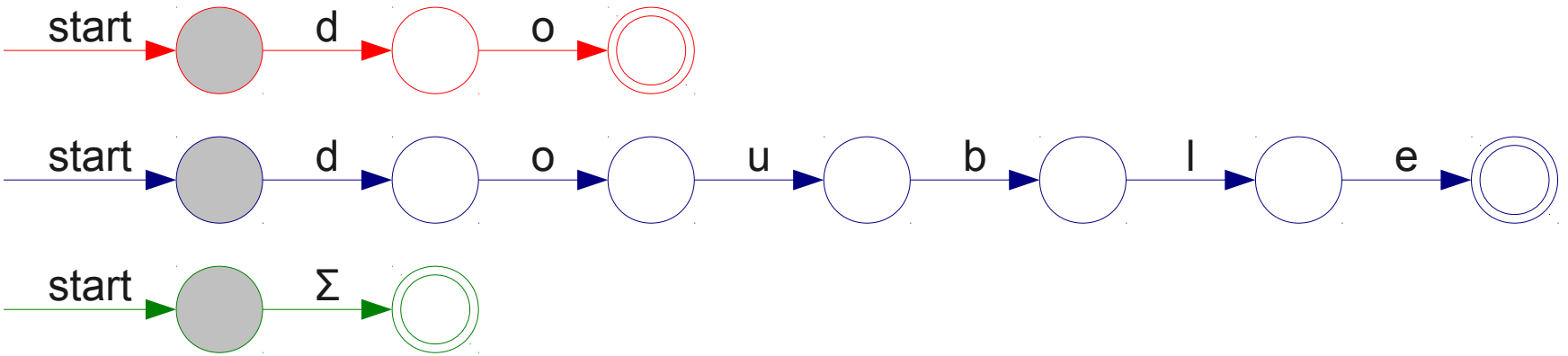
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

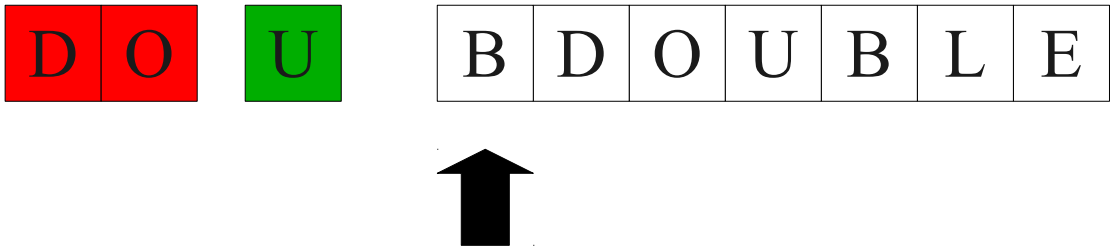




Implementing Maximal Munch

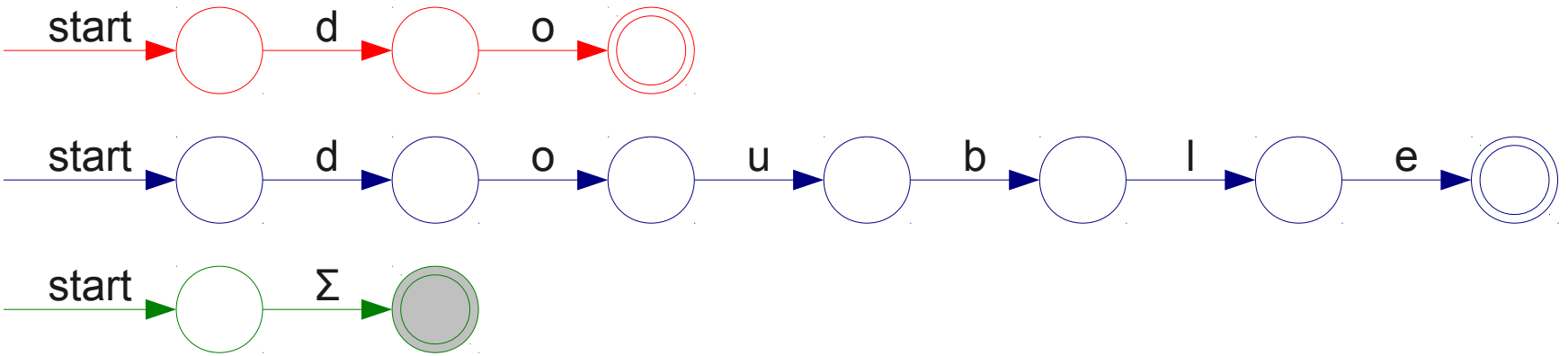
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

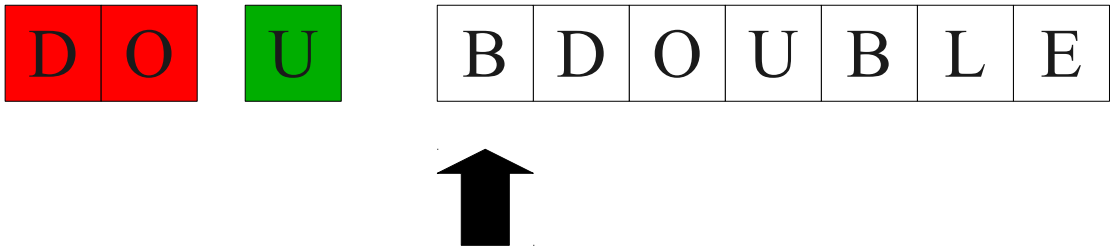




Implementing Maximal Munch

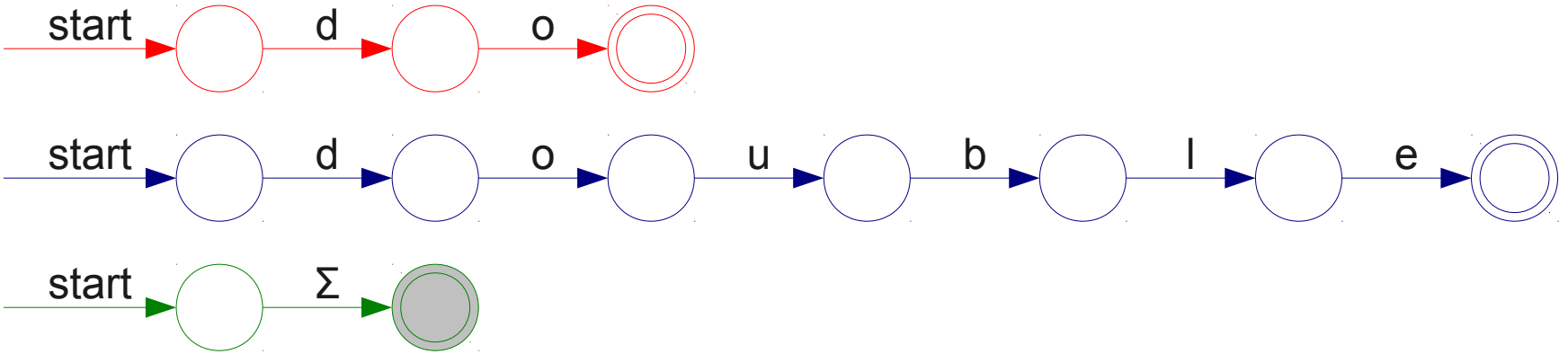
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

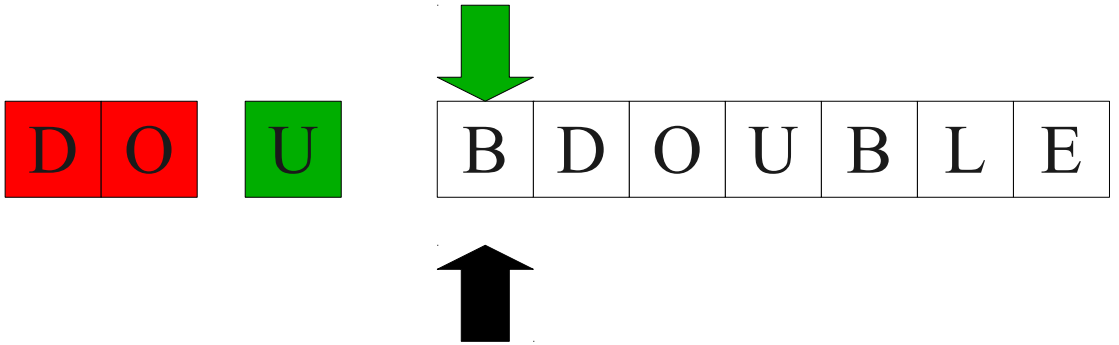




Implementing Maximal Munch

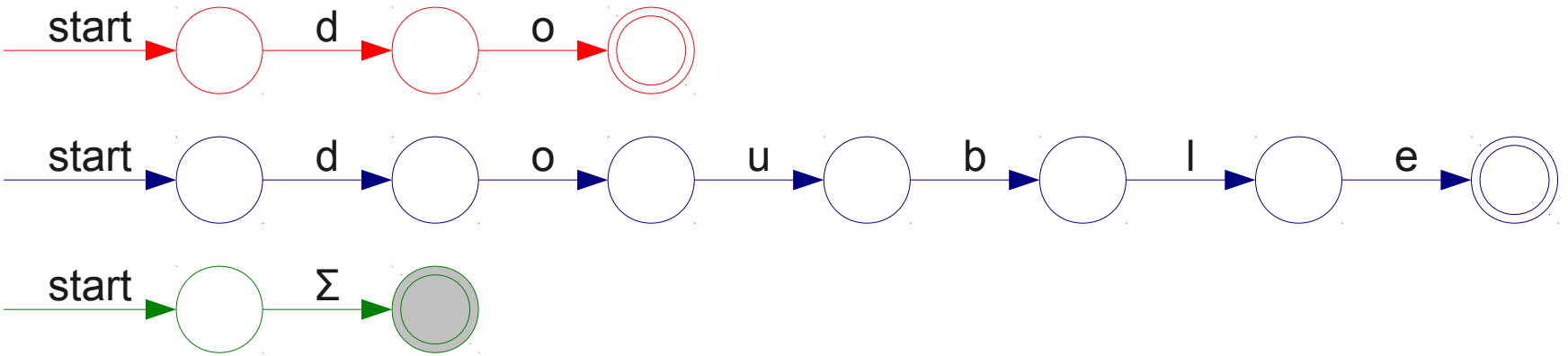
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

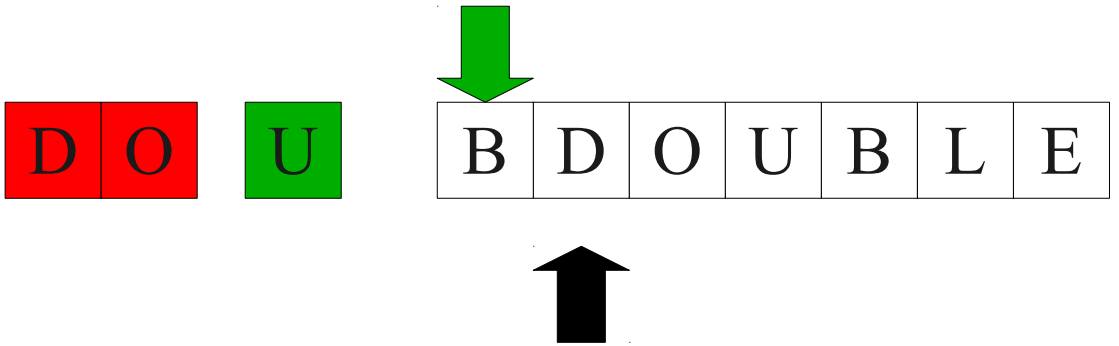




Implementing Maximal Munch

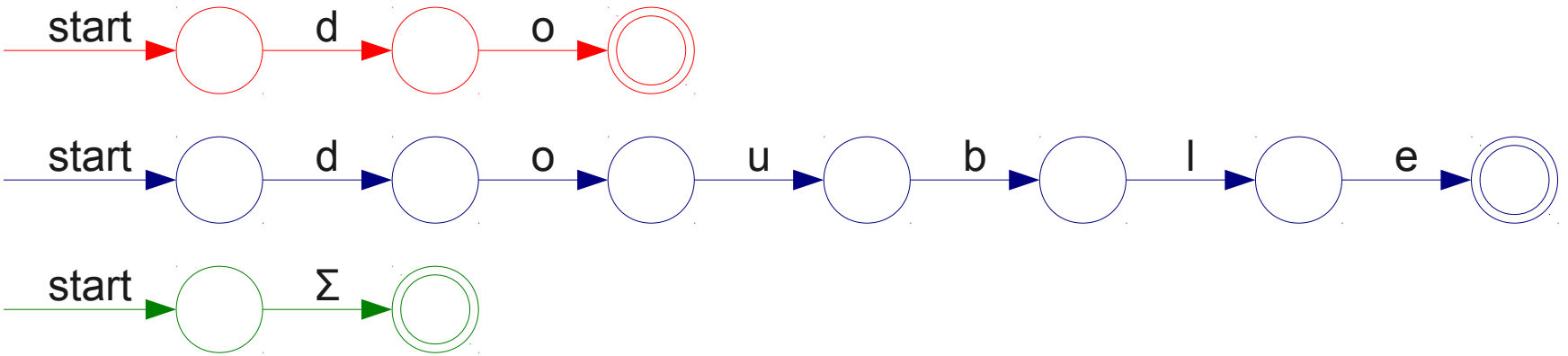
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

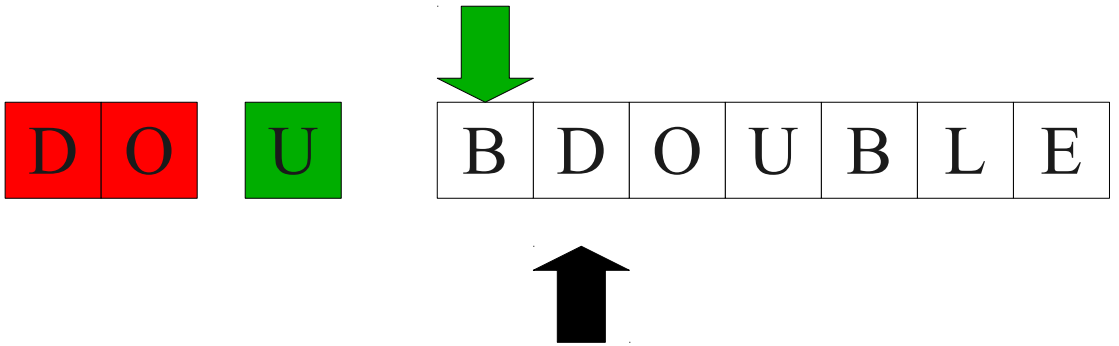




Implementing Maximal Munch

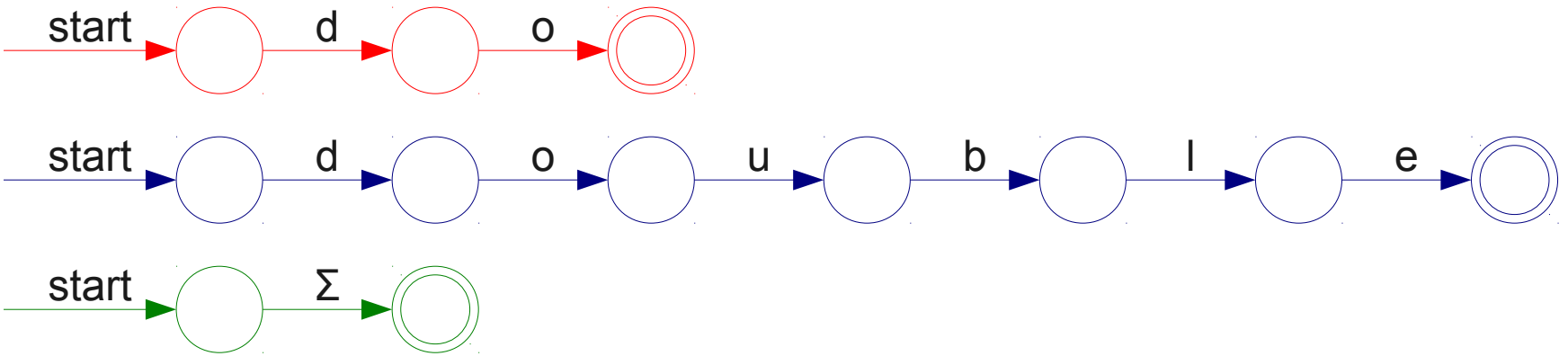
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

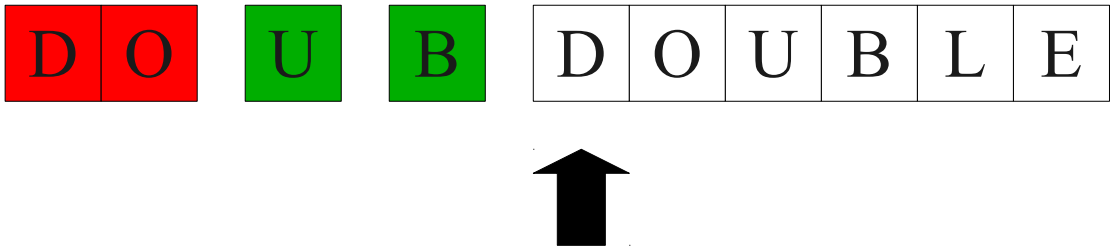




Implementing Maximal Munch

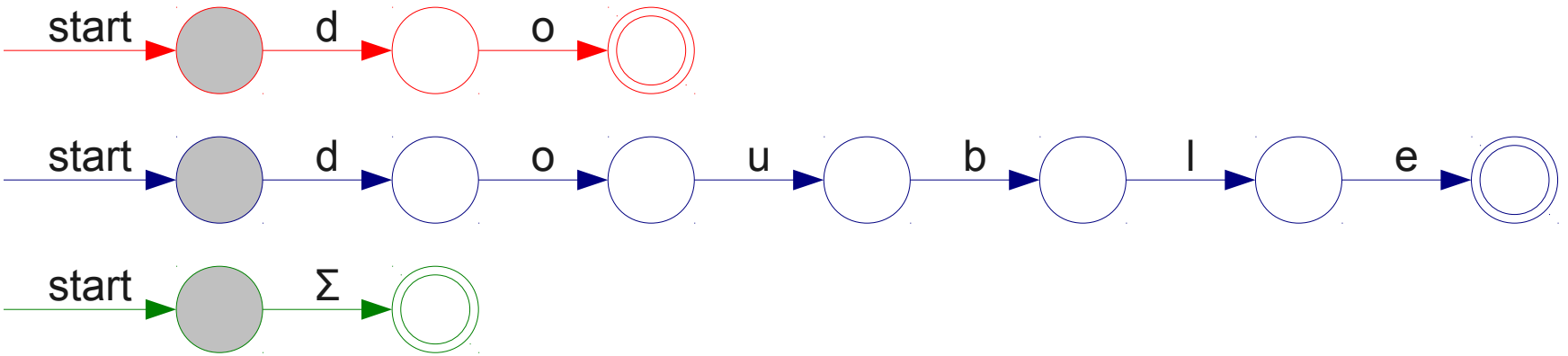
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

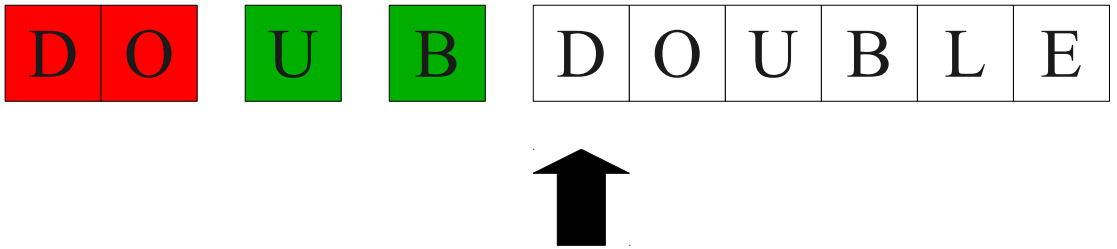




Implementing Maximal Munch

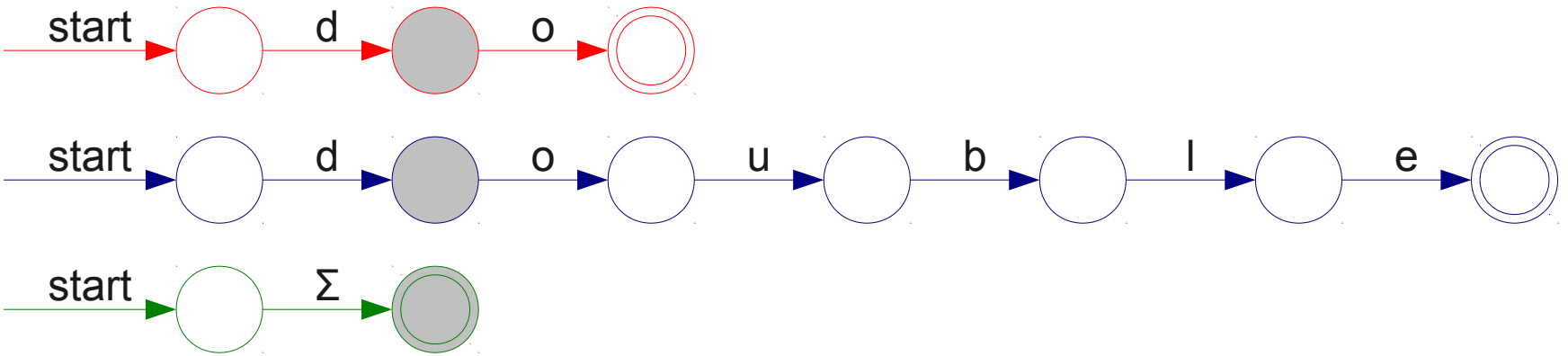
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

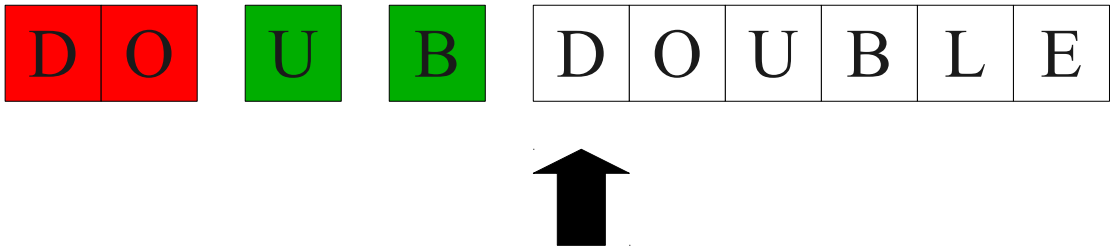




Implementing Maximal Munch

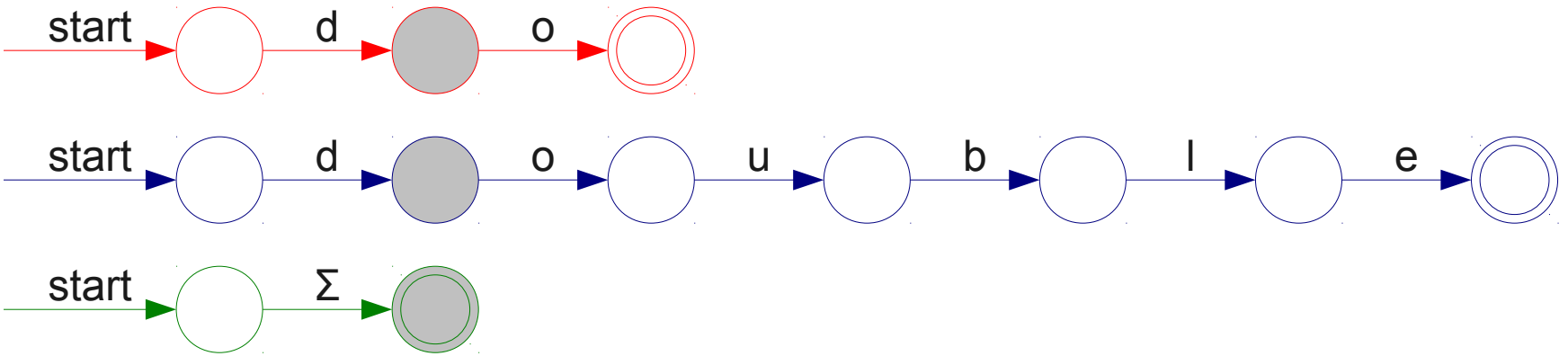
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

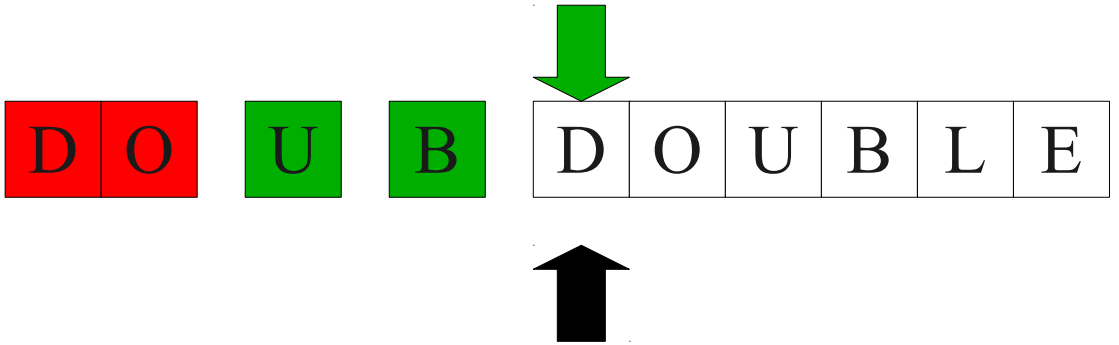




Implementing Maximal Munch

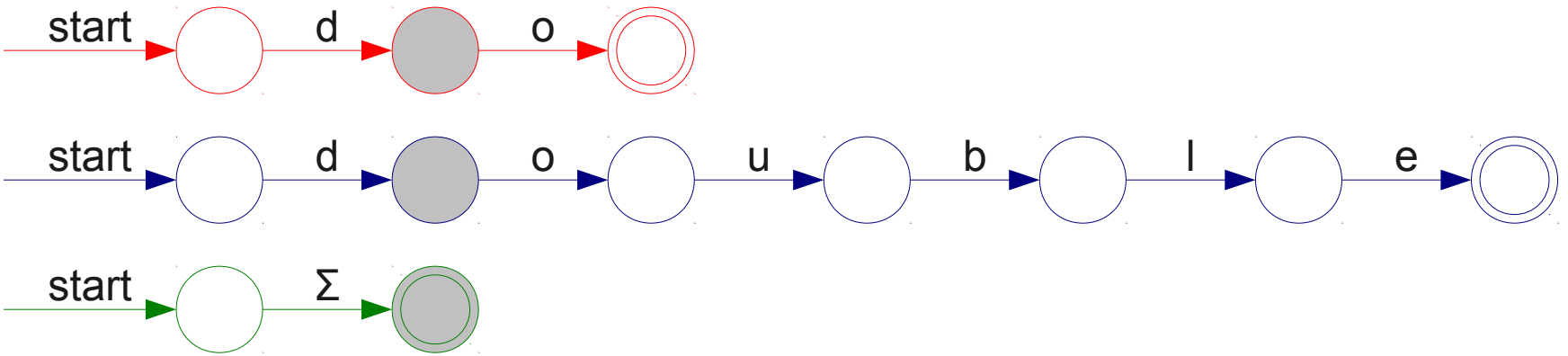
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

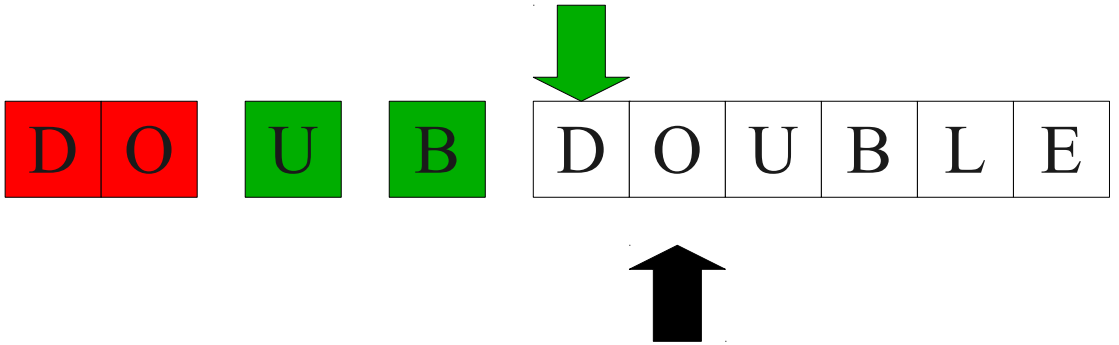




Implementing Maximal Munch

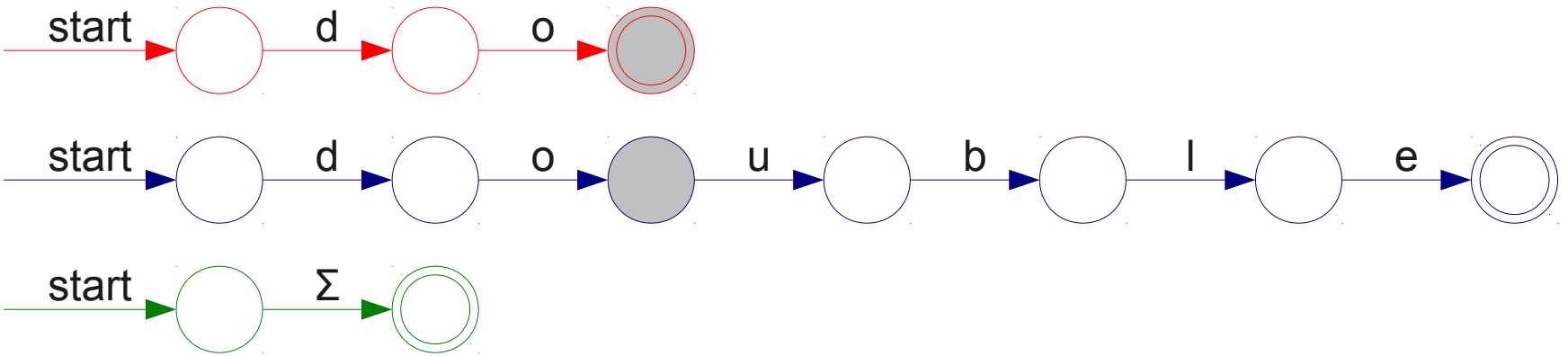
```
T_Do          do
T_Double      double
T_Mystery     [A-Za-z]
```

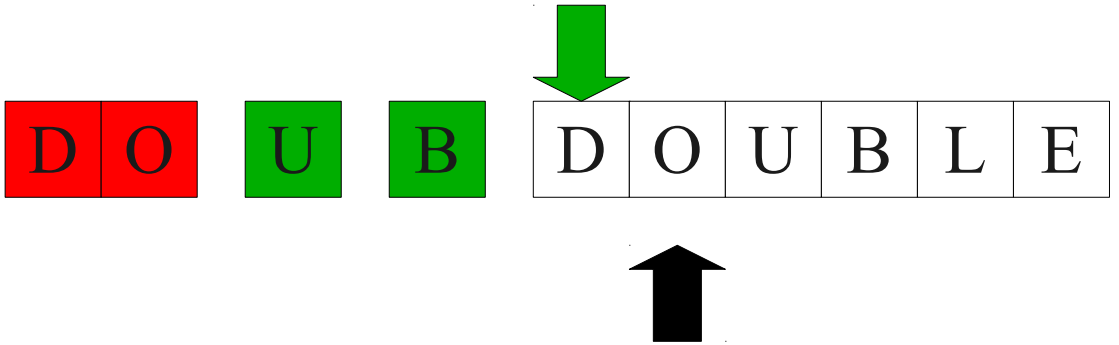




Implementing Maximal Munch

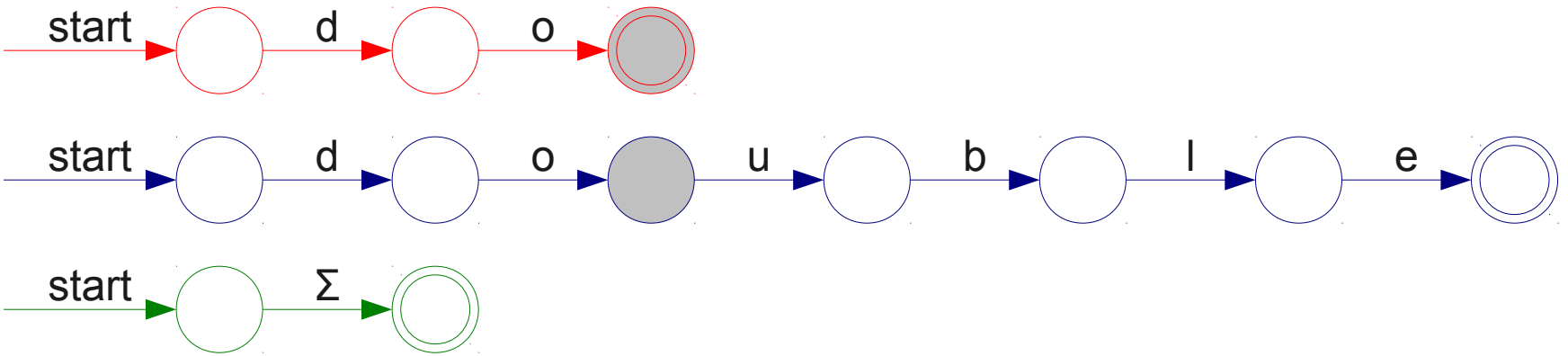
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

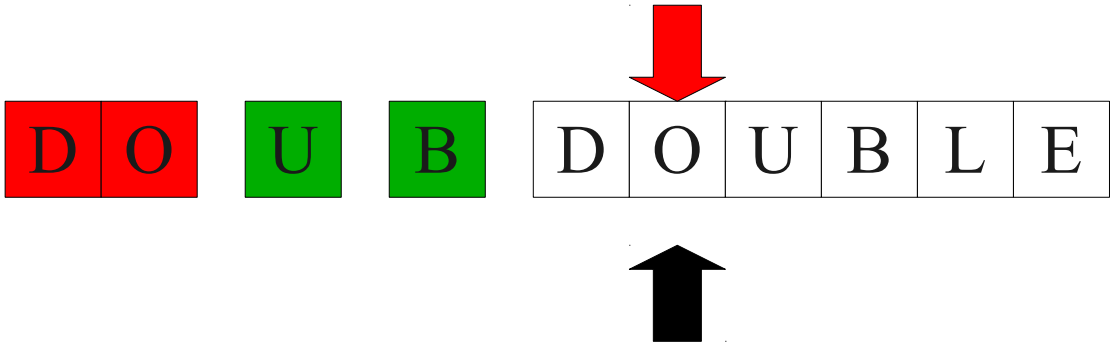




Implementing Maximal Munch

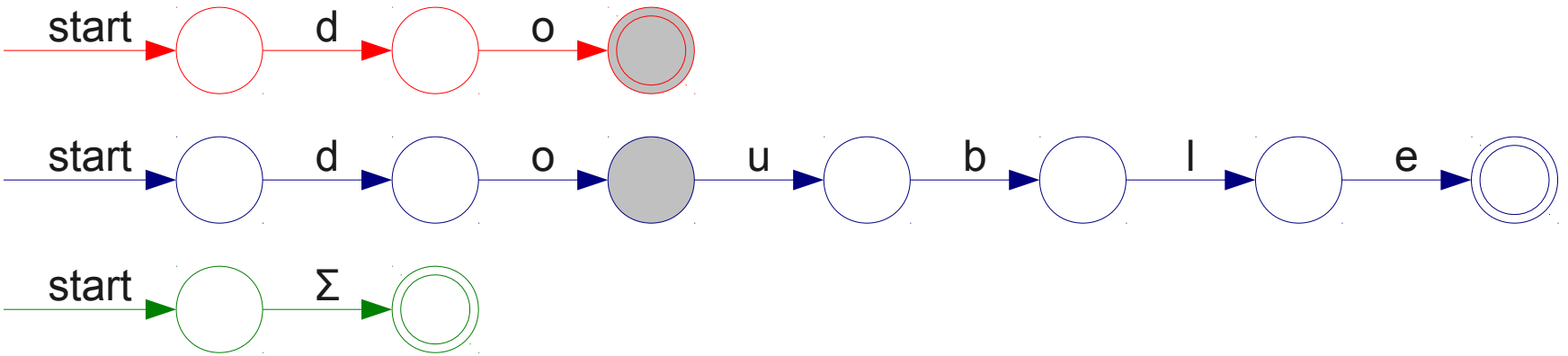
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

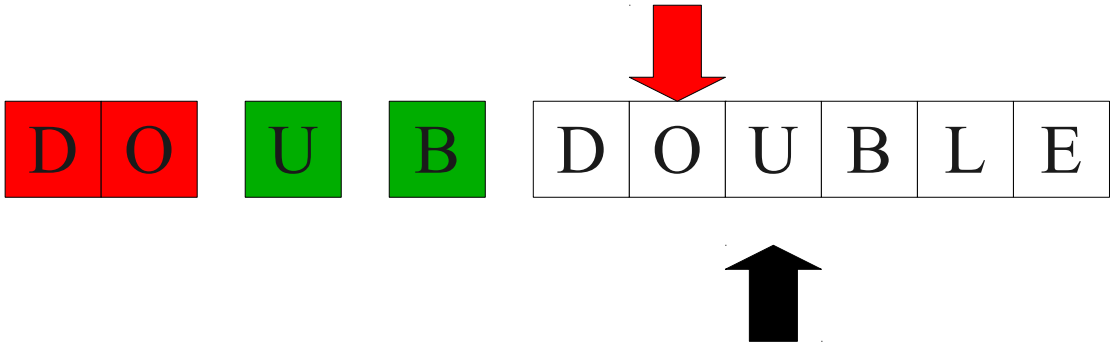




Implementing Maximal Munch

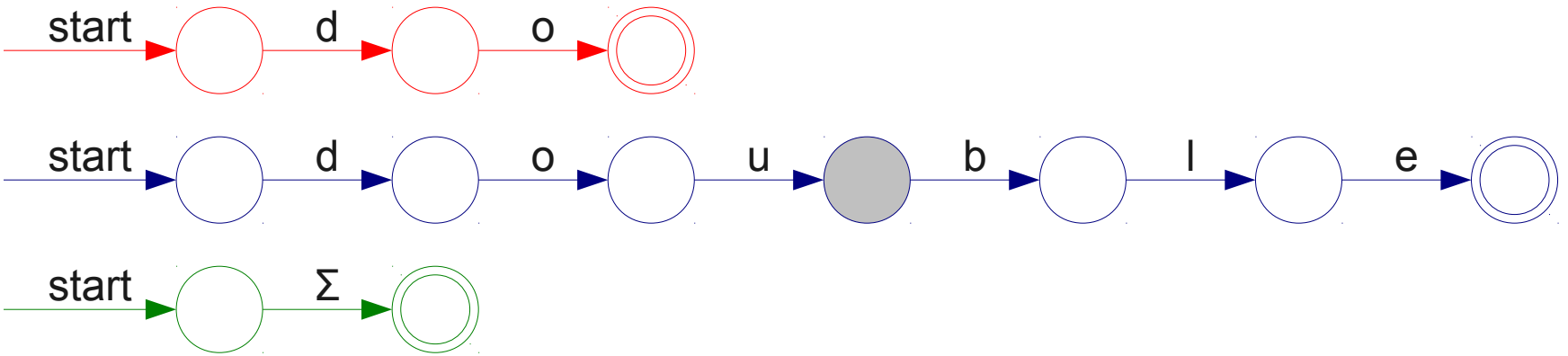
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

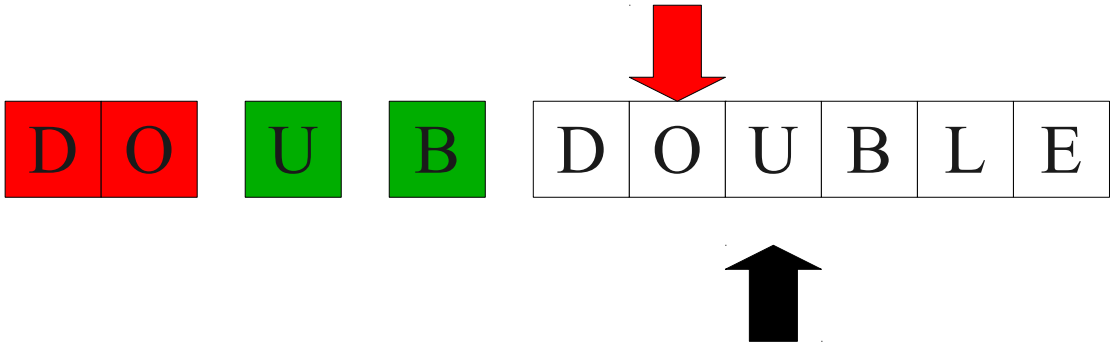




Implementing Maximal Munch

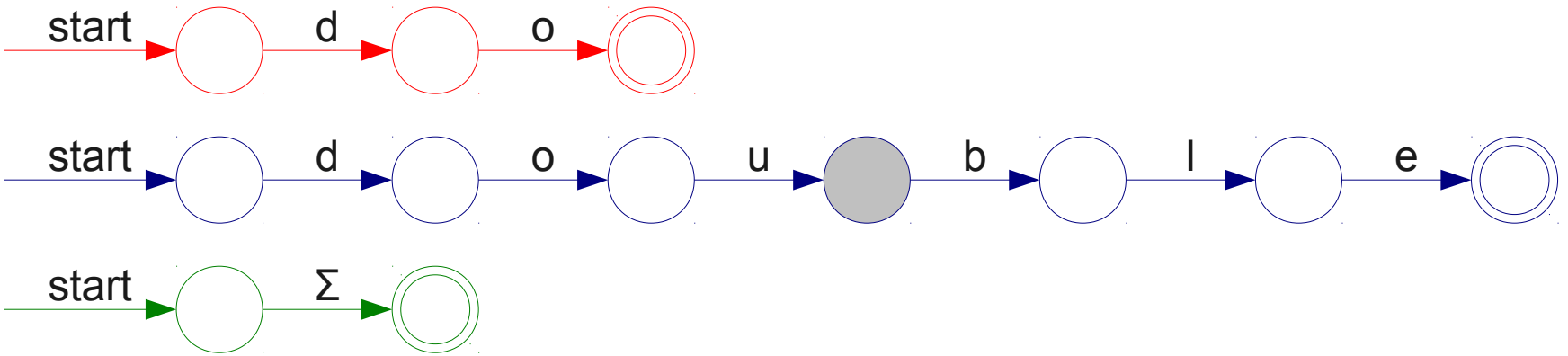
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

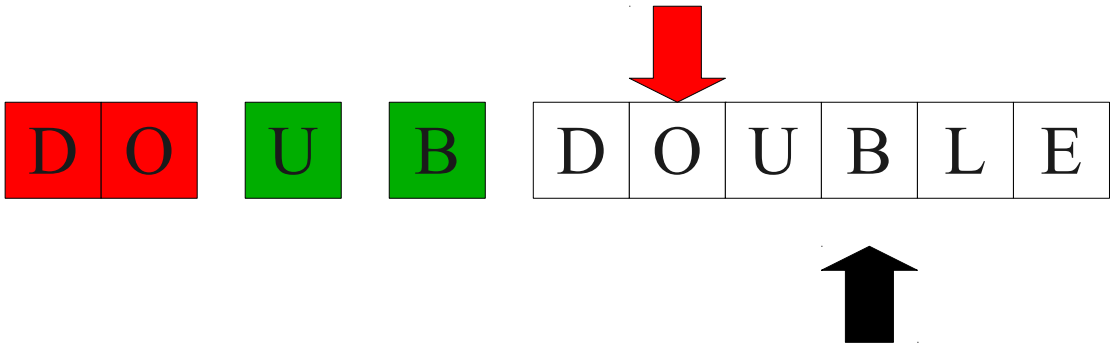




Implementing Maximal Munch

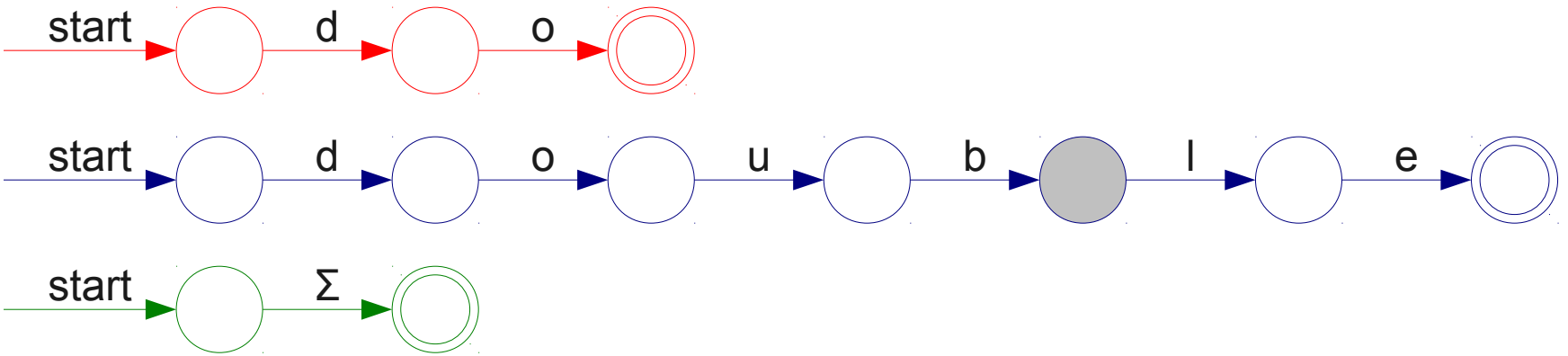
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

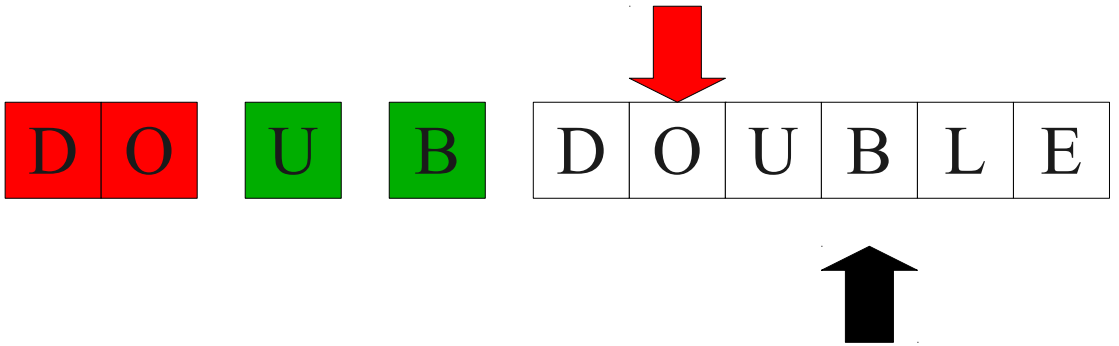




Implementing Maximal Munch

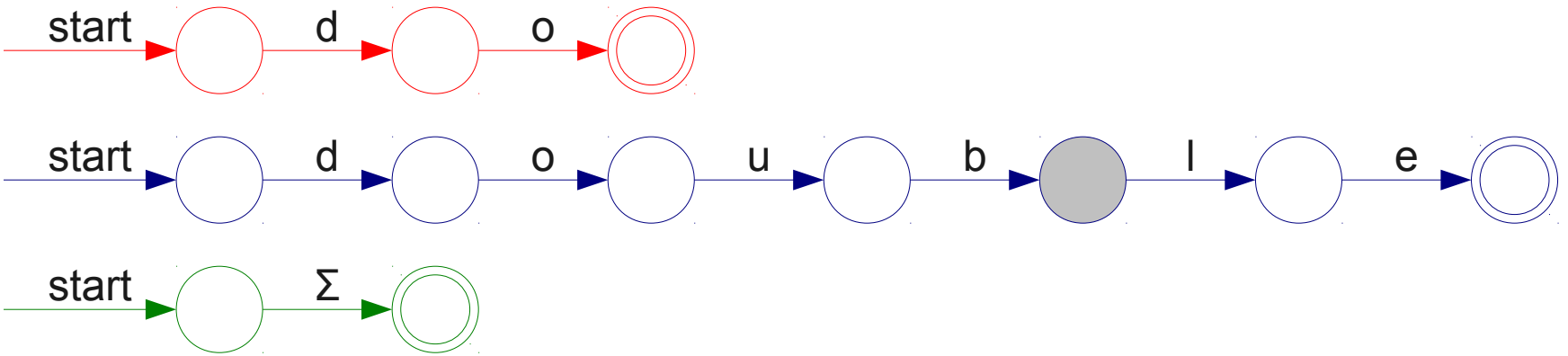
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

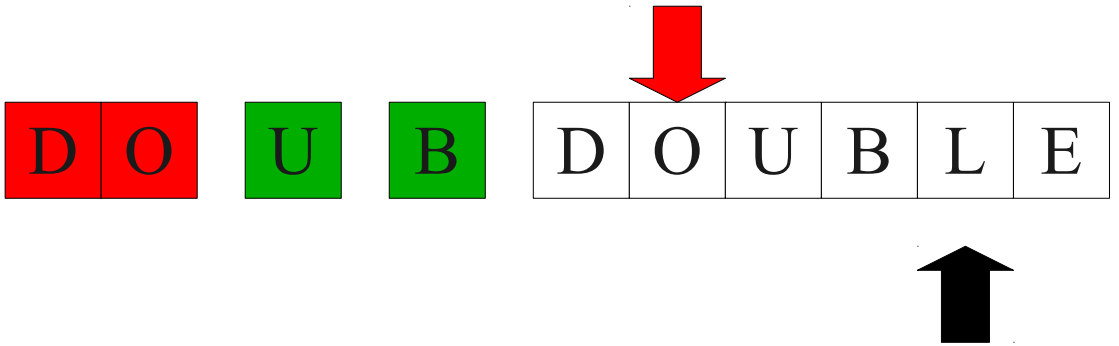




Implementing Maximal Munch

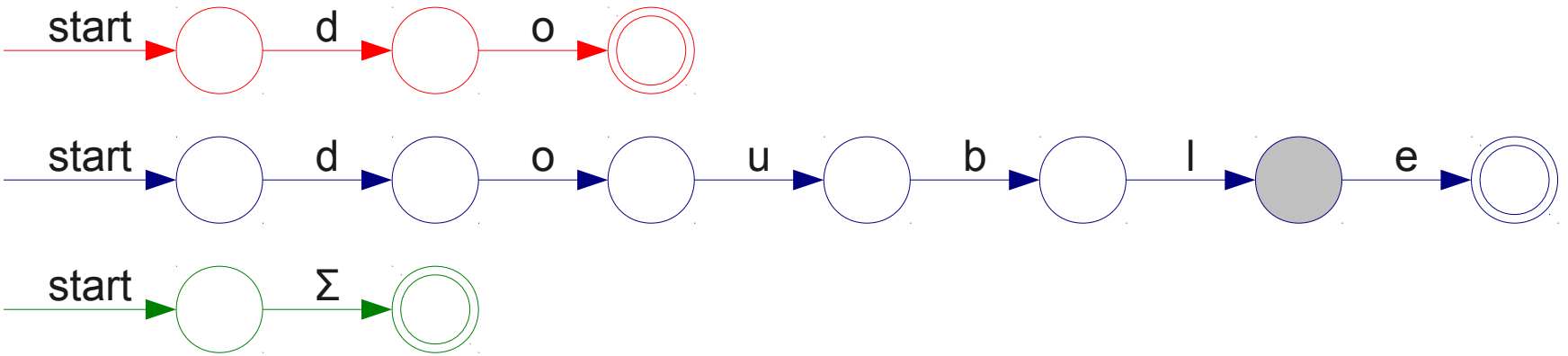
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

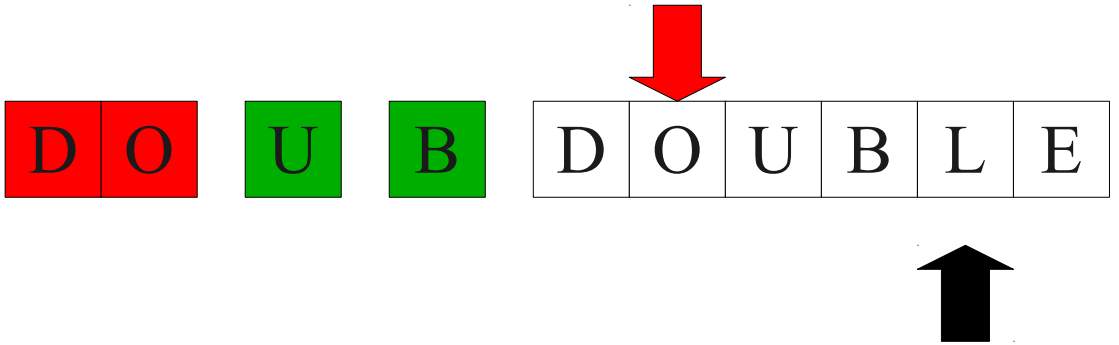




Implementing Maximal Munch

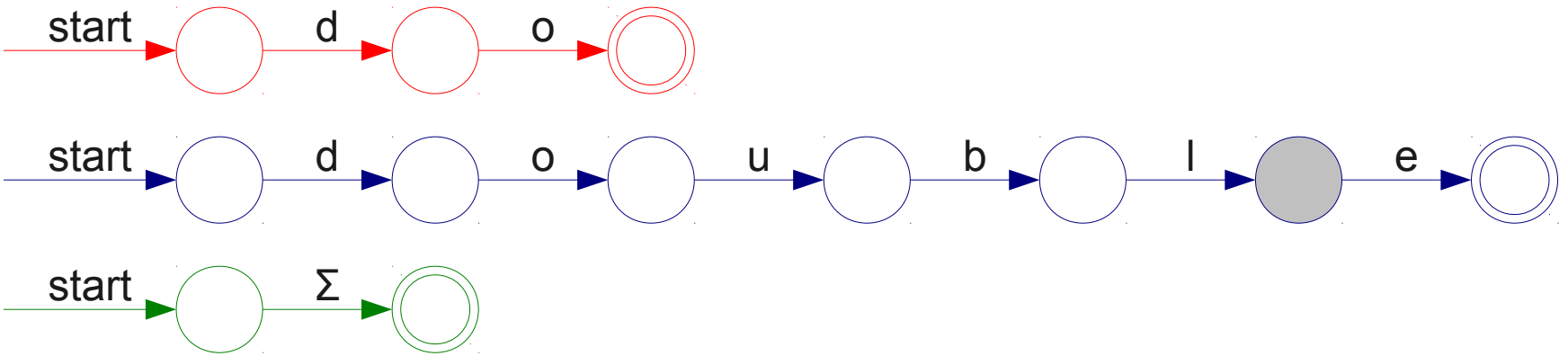
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

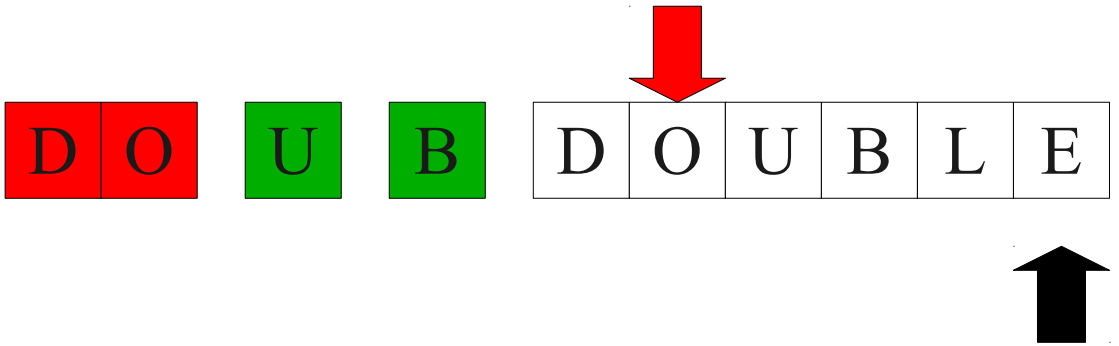




Implementing Maximal Munch

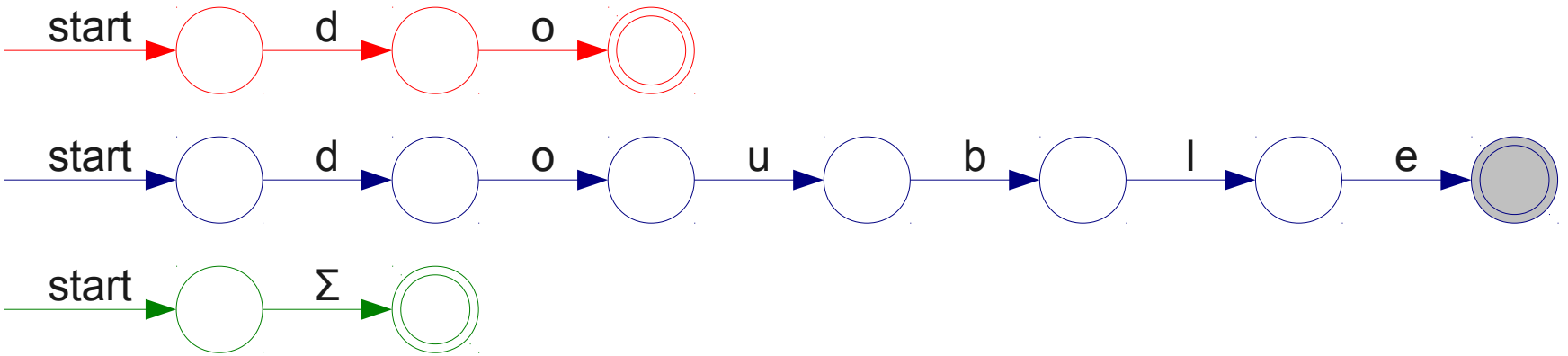
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

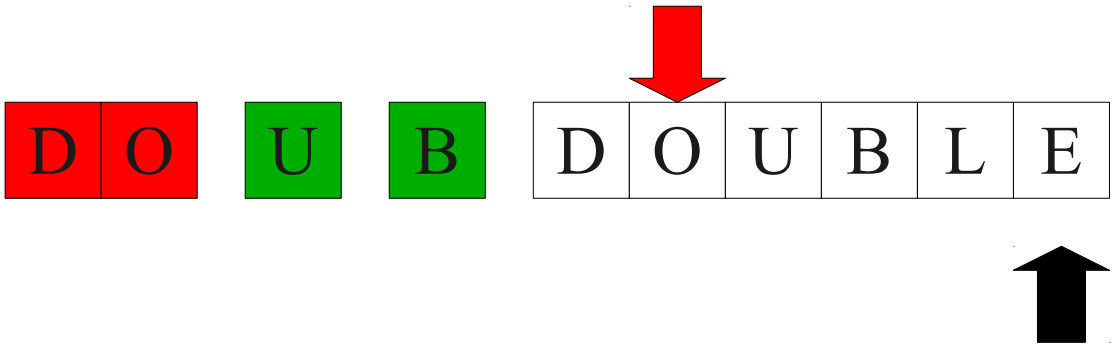




Implementing Maximal Munch

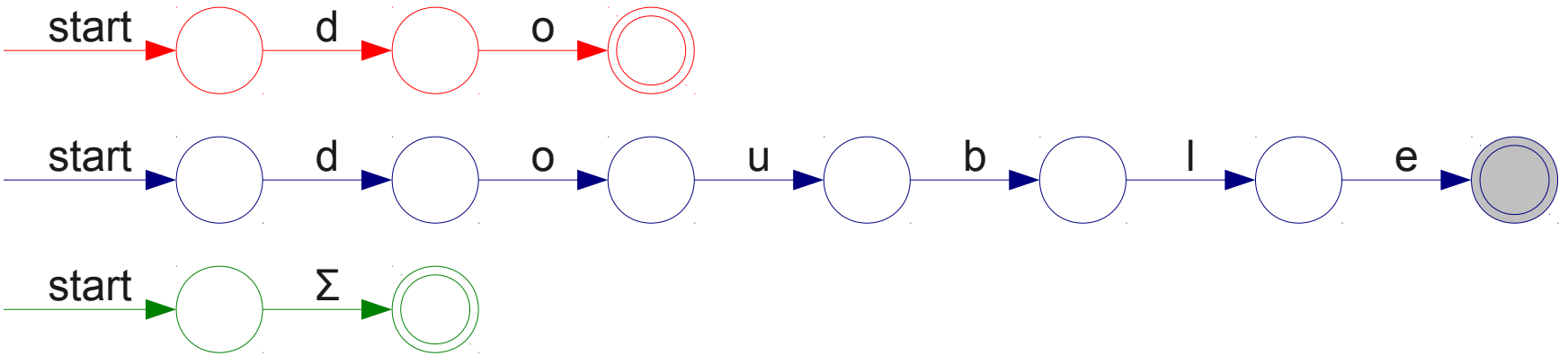
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

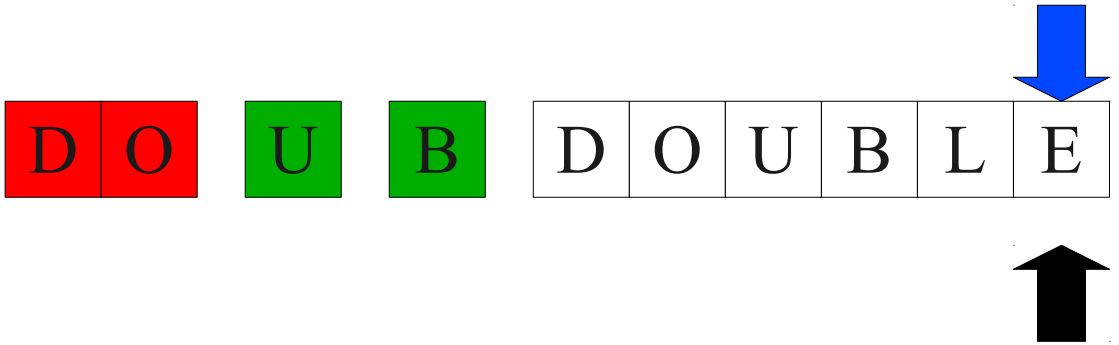




Implementing Maximal Munch

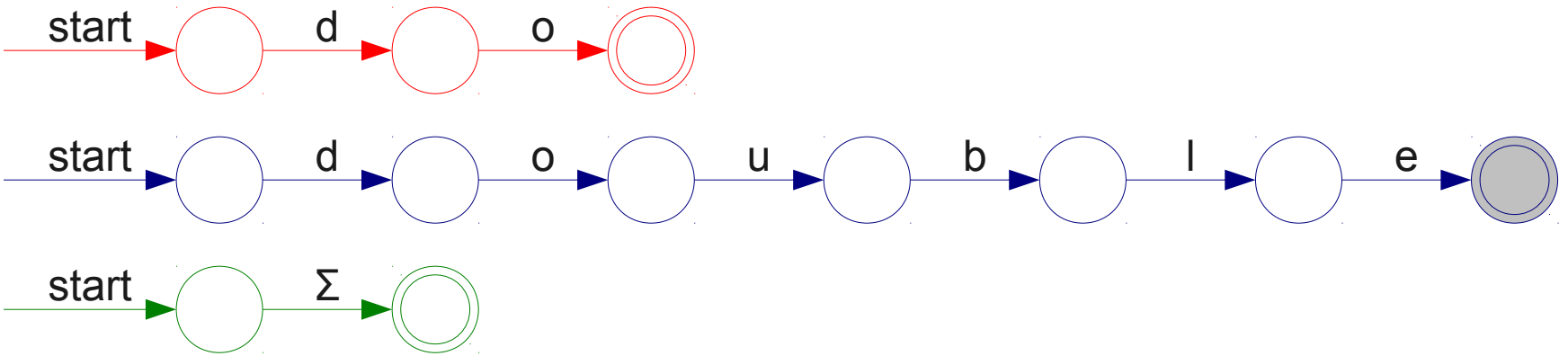
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```

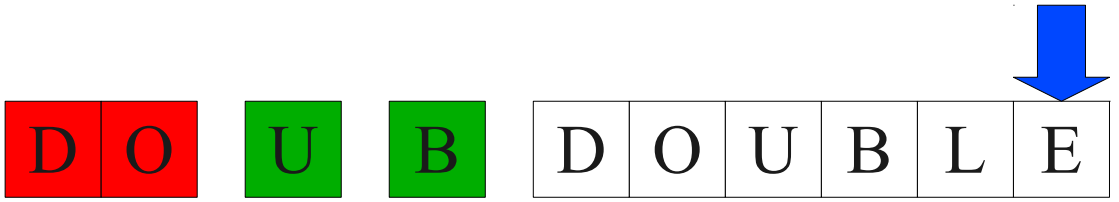




Implementing Maximal Munch

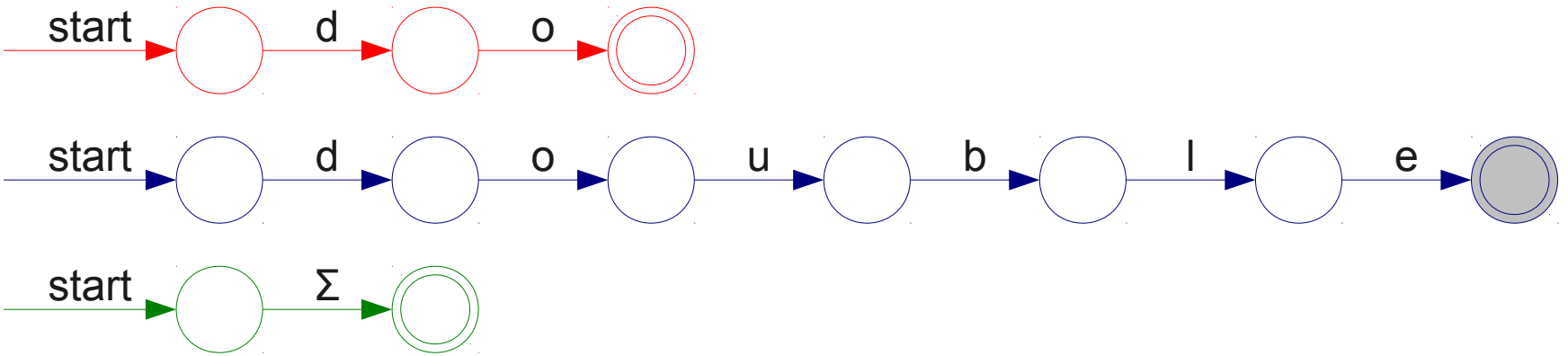
```
T_Do            do
T_Double        double
T_Mystery       [A-Za-z]
```





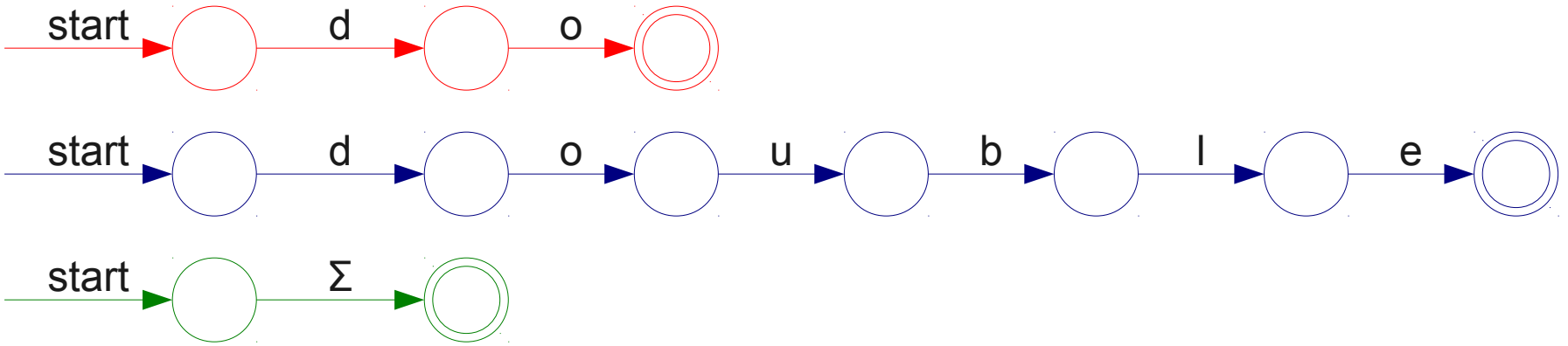
Implementing Maximal Munch

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

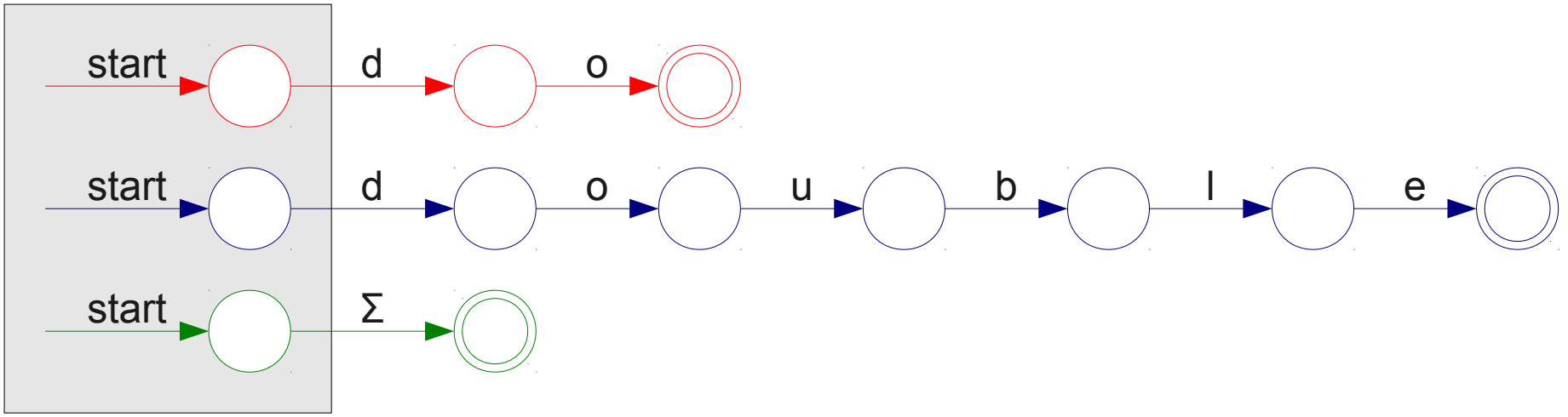


A Minor Simplification

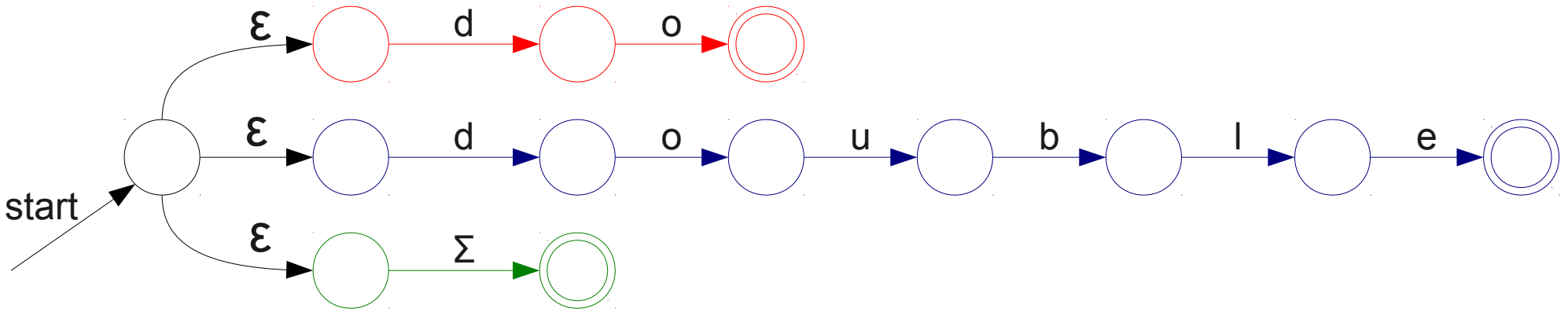
A Minor Simplification



A Minor Simplification



A Minor Simplification

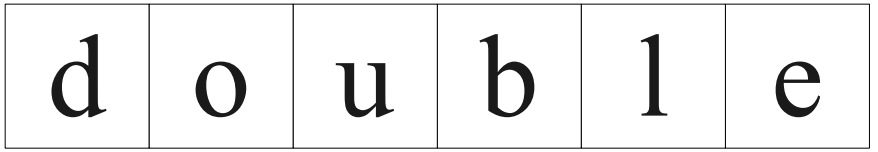


Other Conflicts

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

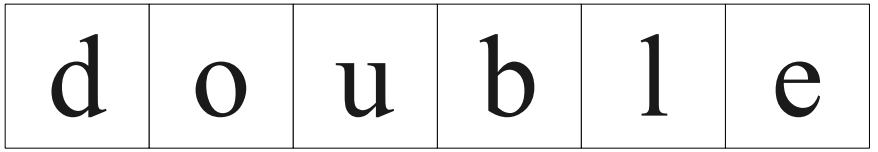
Other Conflicts

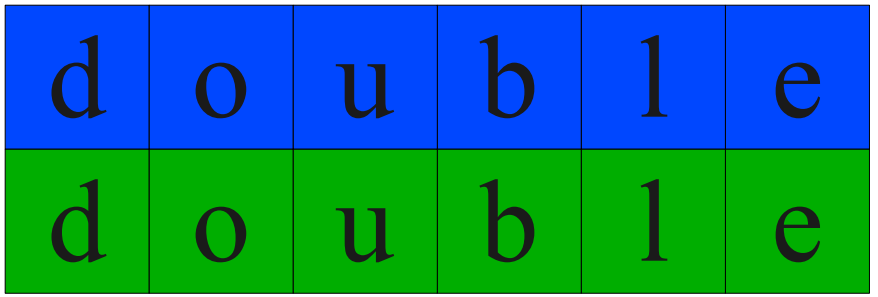
```
T_Do           do
T_Double       double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```



Other Conflicts

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



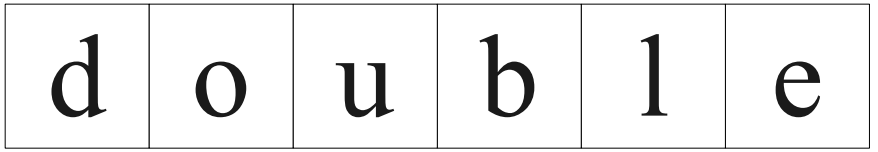


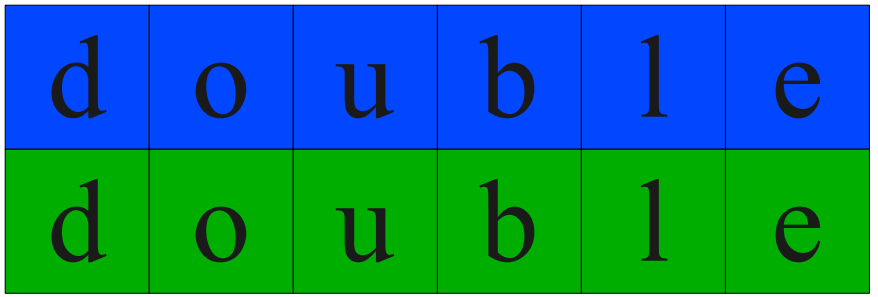
More Tiebreaking

- When two regular expressions apply, choose the one with the greater “priority.”
- Simple priority system: **which rule was defined first?**

Other Conflicts

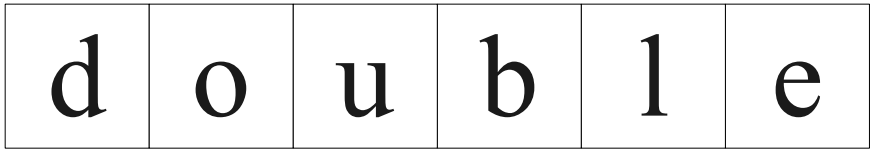
```
T_Do            do
T_Double        double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

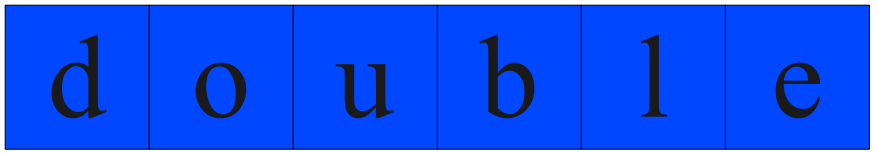




Other Conflicts

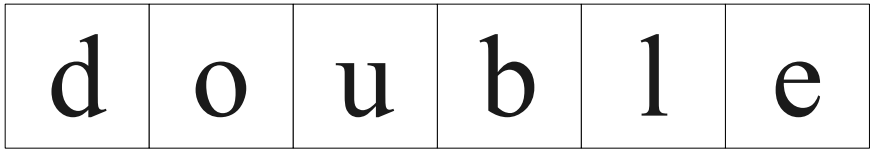
```
T_Do            do
T_Double        double
T_Identifier [A-Za-z_][A-Za-z0-9_]*
```

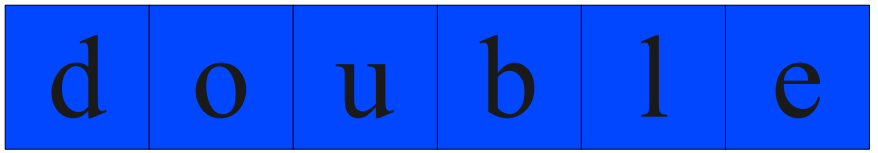




Other Conflicts

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





Why isn't this a problem?

One Last Detail...

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

Summary of Conflict Resolution

- Construct an automaton for each regular expression.
- Merge them into one automaton by adding a new start state.
- Scan the input, keeping track of the last known match.
- Break ties by choosing higher-precedence matches.
- Have a catch-all rule to handle errors.

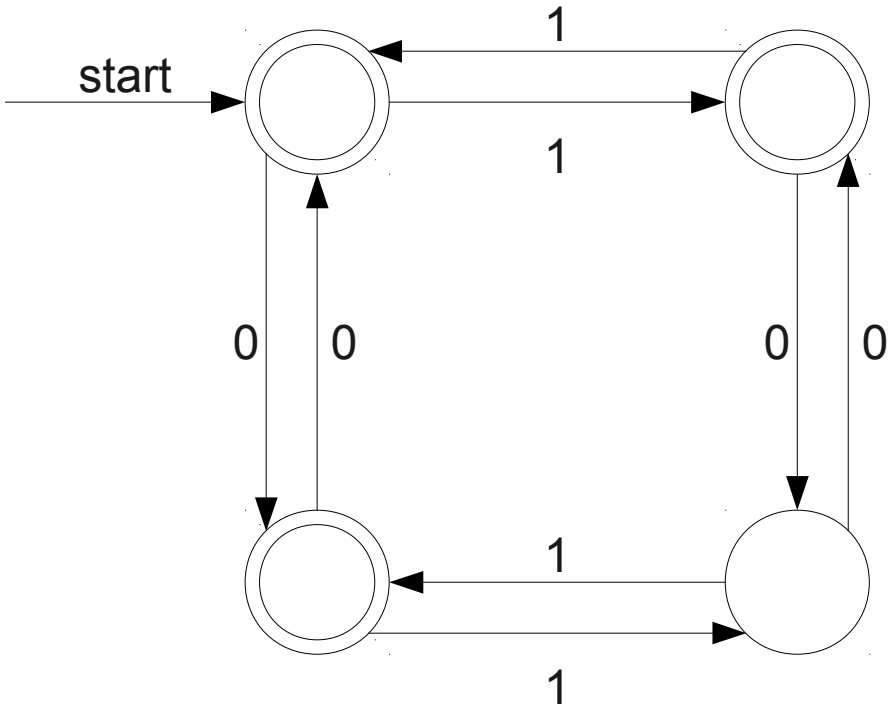
Speeding up the Scanner

DFAs

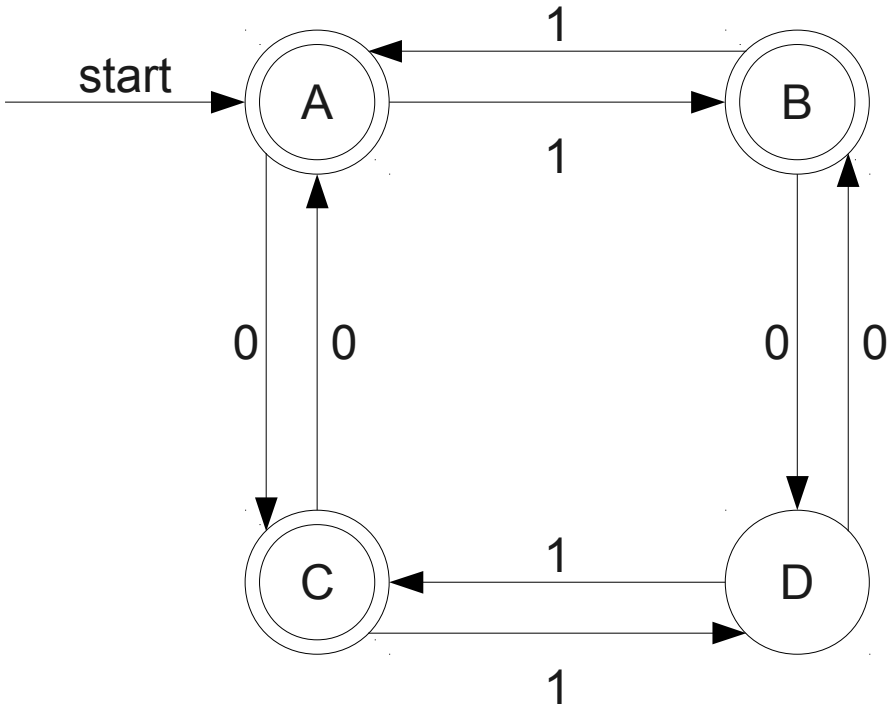
- The automata we've seen so far have all been NFAs.
- A **DFA** is like an NFA, but with tighter restrictions:
  - Every state must have **exactly one** transition defined for every letter.
  - ε-moves are not allowed.

A Sample DFA

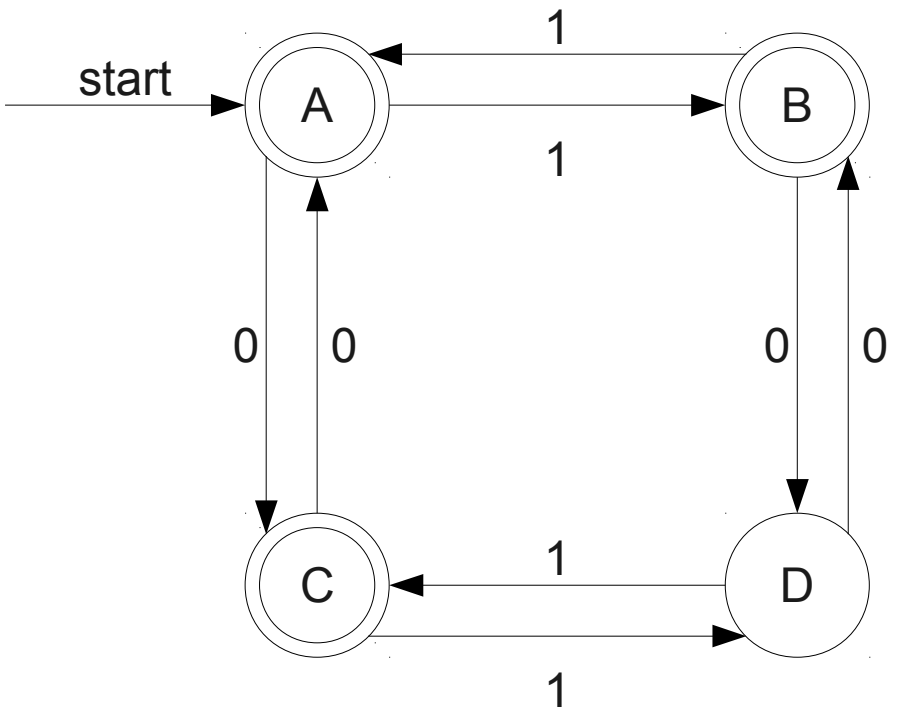
A Sample DFA



A Sample DFA



A Sample DFA



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

Simulating a DFA

- Begin in the start state.
- For each character of the input:
  - Follow the indicated transition to the next state.
- Assuming transitions are in a table, time to scan a string of length S is O(|S|).

Speeding up Matching

- In the worst-case, an NFA takes $O(|S||Q|^2)$ time to match a string.
- DFAs, on the other hand, take only $O(|S|)$.
- There is another (beautiful!) algorithm to convert NFAs to DFAs.

Subset Construction

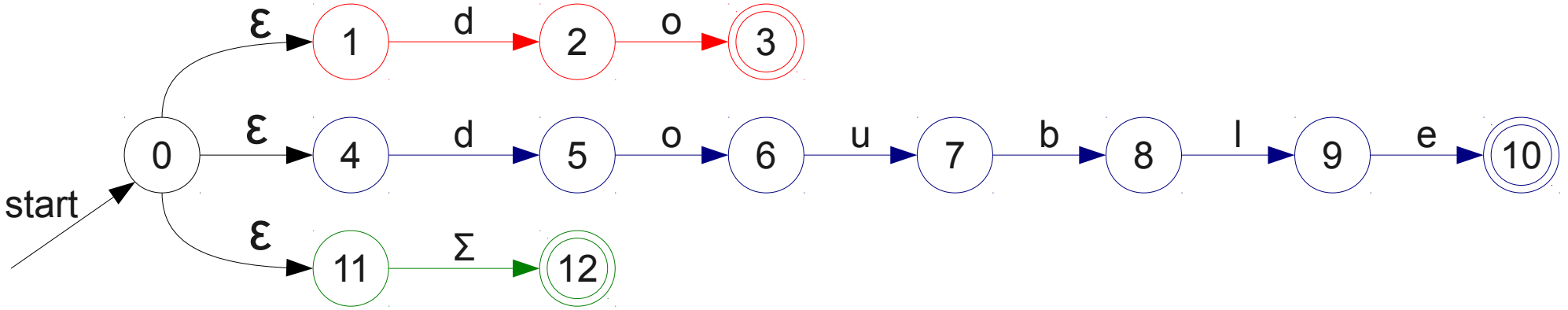
- NFAs can be in many states at once, while DFAs can only be in a single state at a time.
- Key idea: **Make the DFA simulate the NFA**.
- Have the states of the DFA correspond to the **sets of states** of the NFA.
- Transitions between states of DFA correspond to transitions between **sets of states**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

Formally...

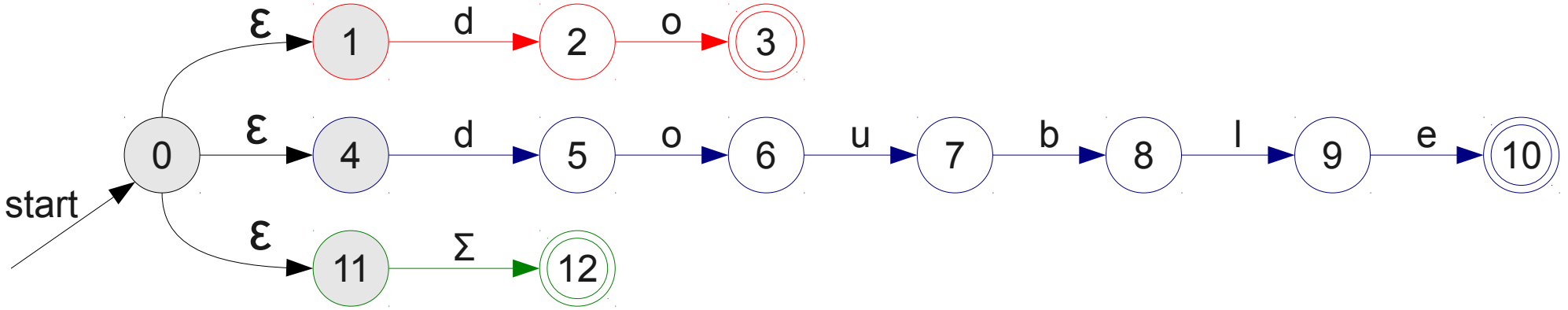
- Given an NFA $N = (\Sigma, Q, \delta, s, F)$, define the DFA $D = (\Sigma', Q', \delta', s', F')$ as follows:
  - $\Sigma' = \Sigma$
  - $Q' = 2^Q$
  - $\delta'(S, a) = \{\text{ECLOSE}(s') \mid s \in S \text{ and } \delta(s, a) = s'\}$
  - $s' = \text{ECLOSE}(s)$
  - $F' = \{ S \mid \exists\, s \in S.\ s \in F \}$
- Here, ECLOSE(s) is the set of states reachable from s via ε-moves.

From NFA to DFA

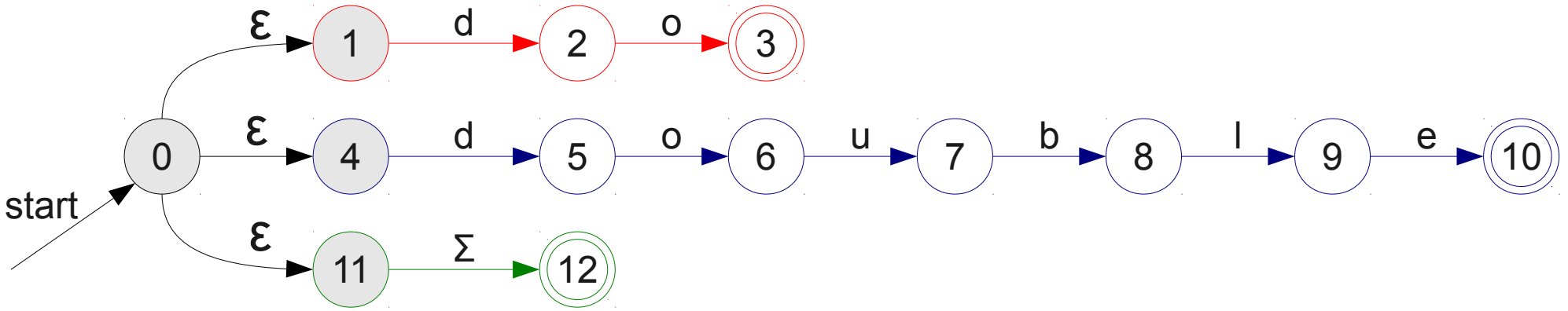
From NFA to DFA

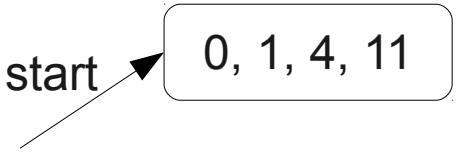


From NFA to DFA

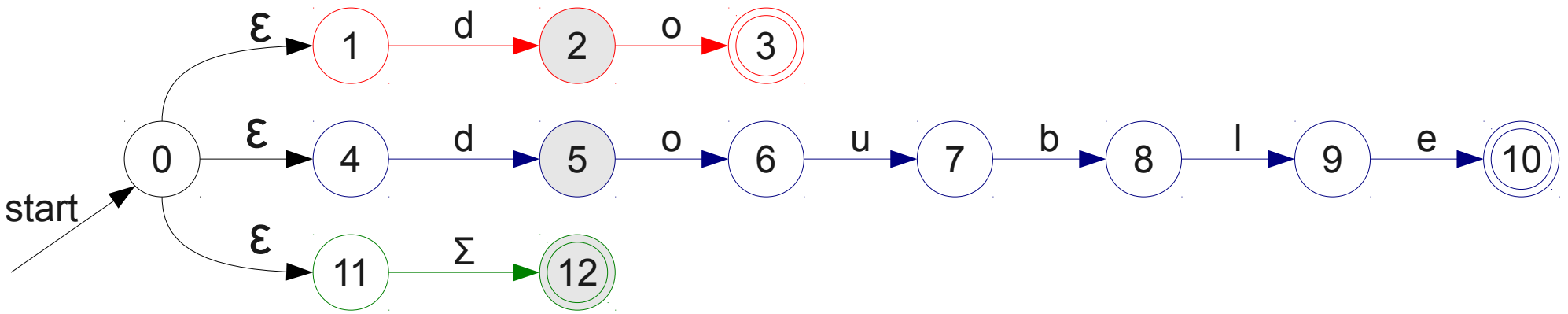


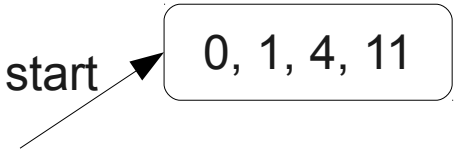
From NFA to DFA



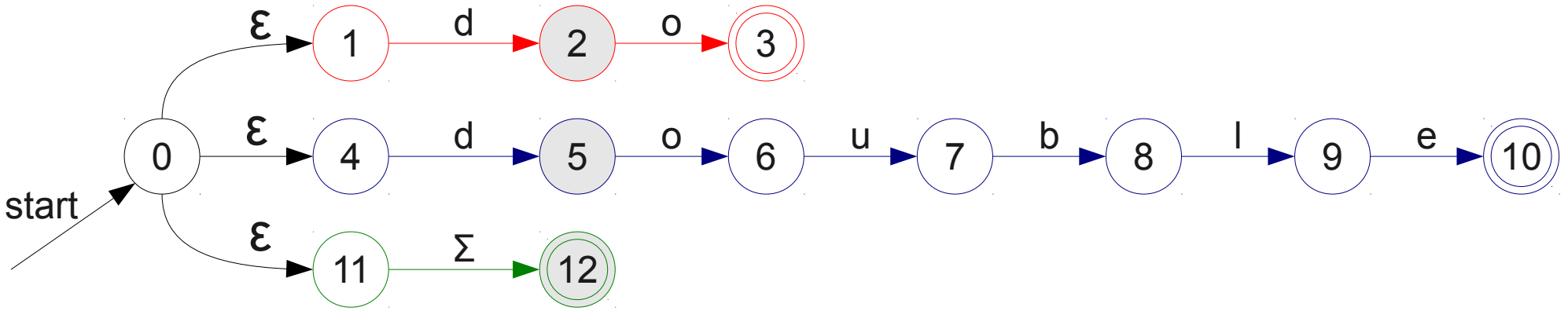


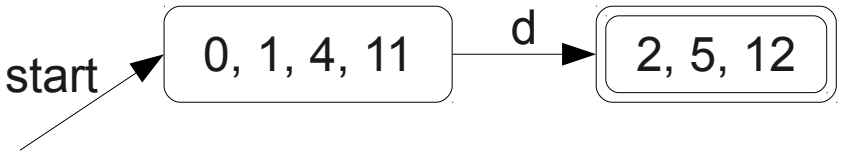
From NFA to DFA



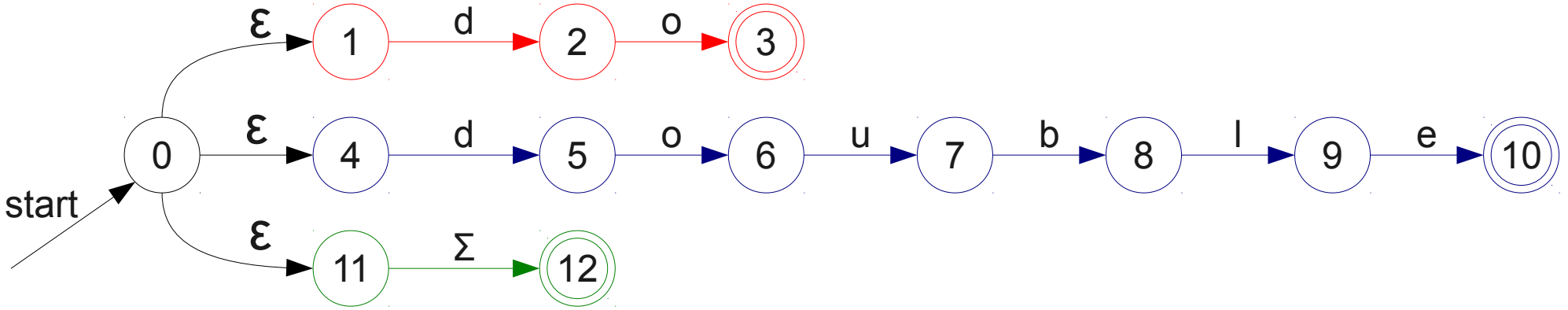


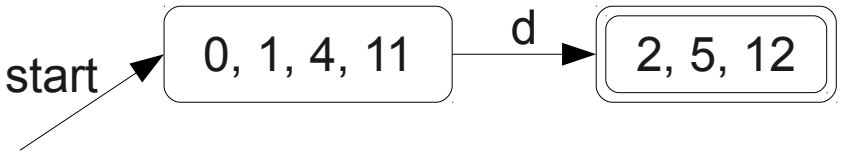
From NFA to DFA



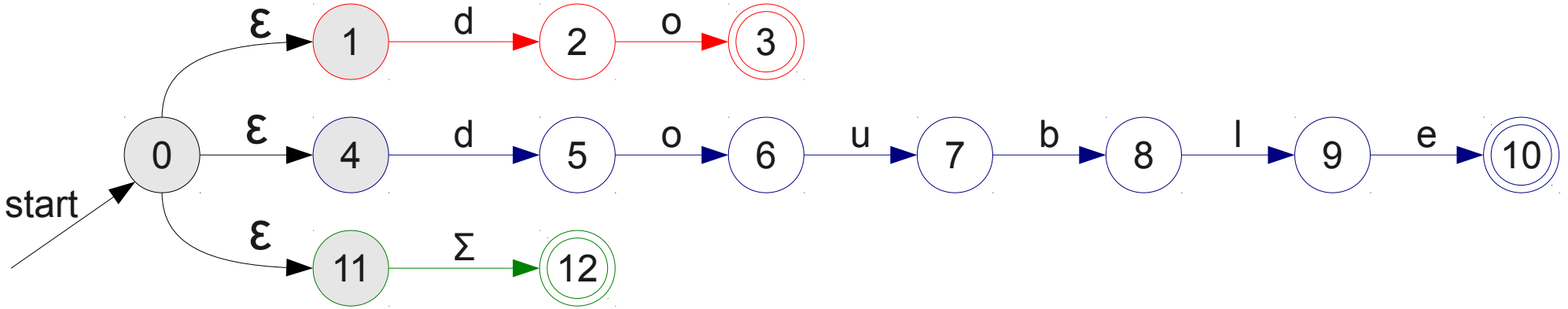


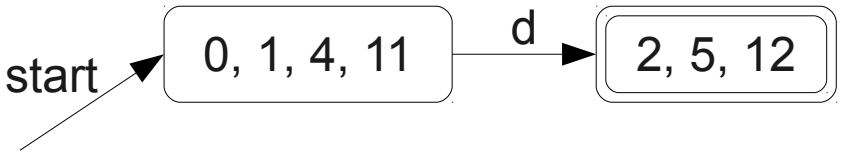
From NFA to DFA



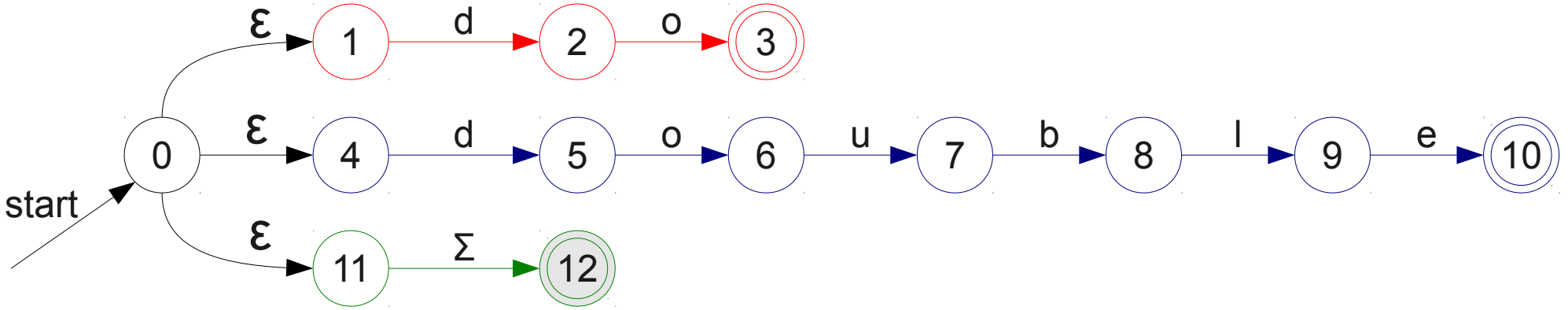


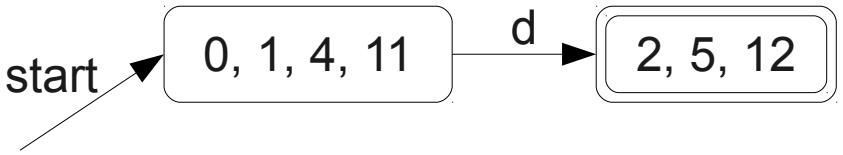
From NFA to DFA



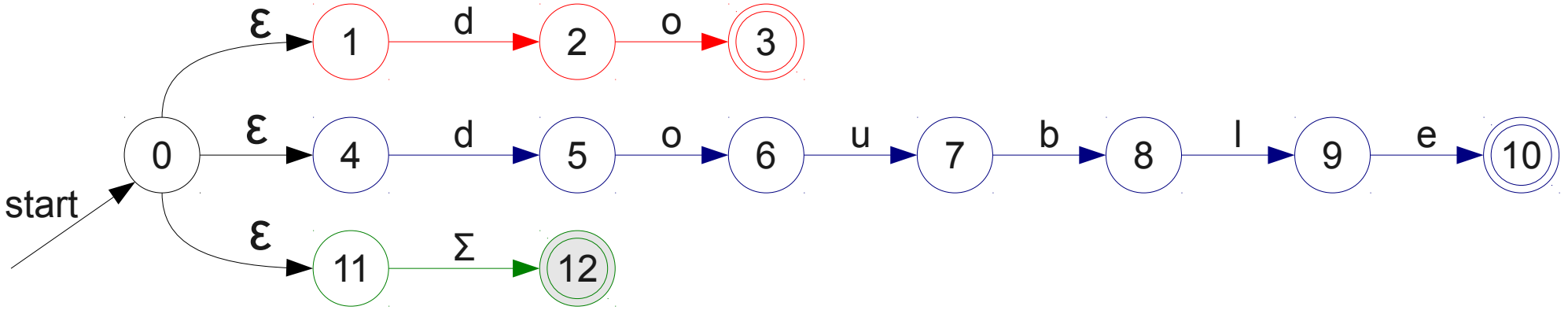


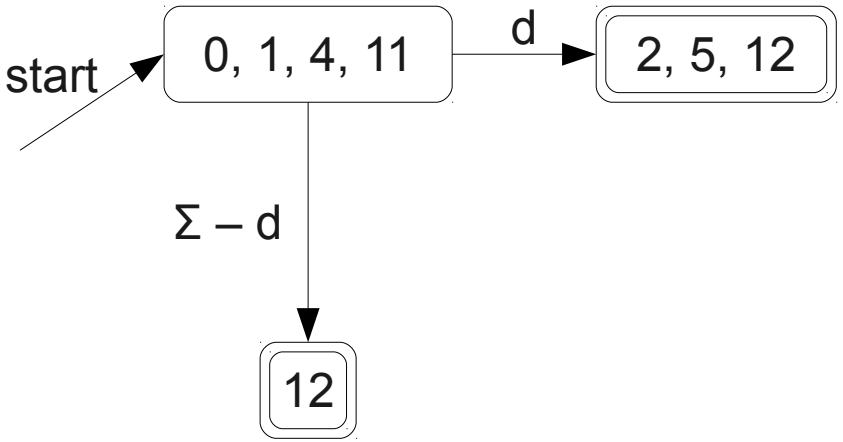
From NFA to DFA



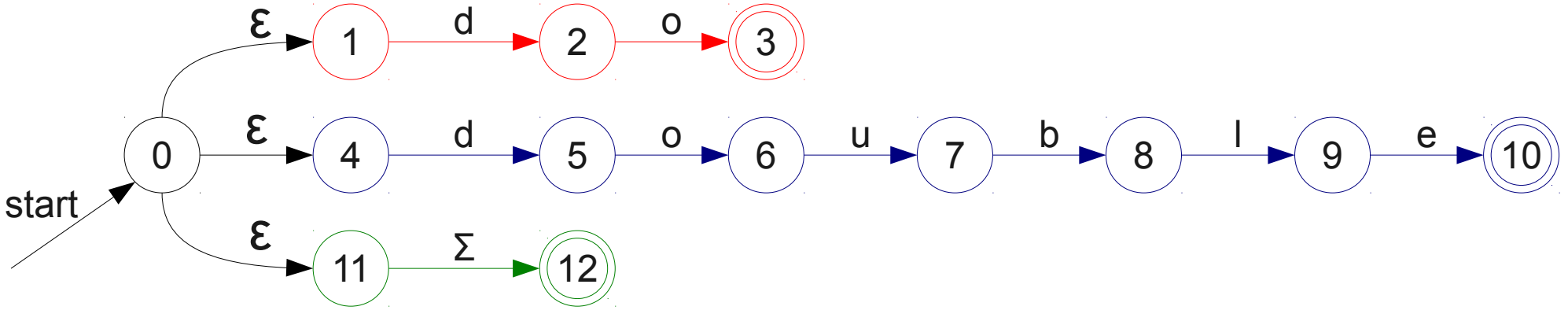


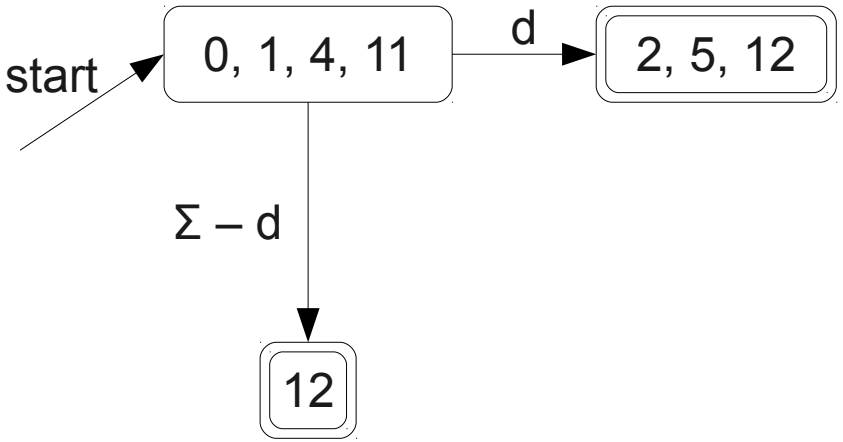
From NFA to DFA



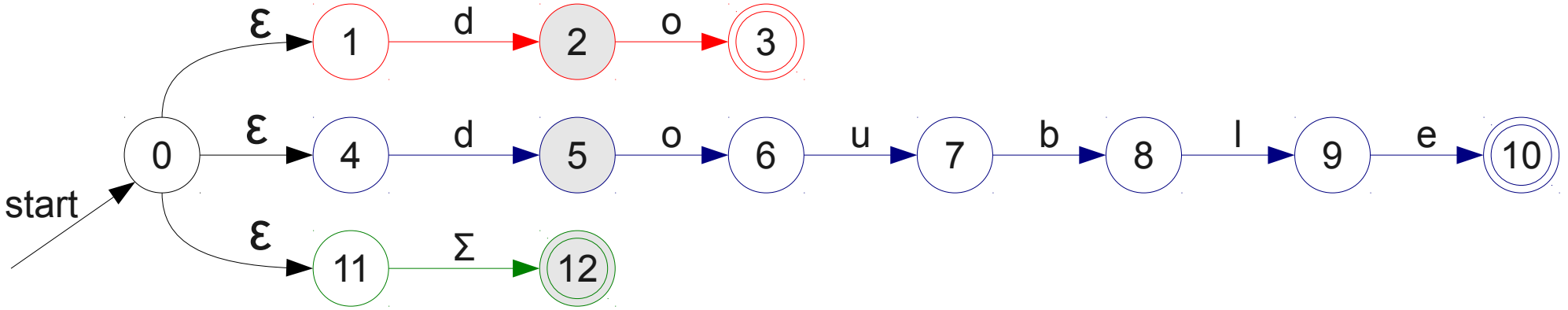


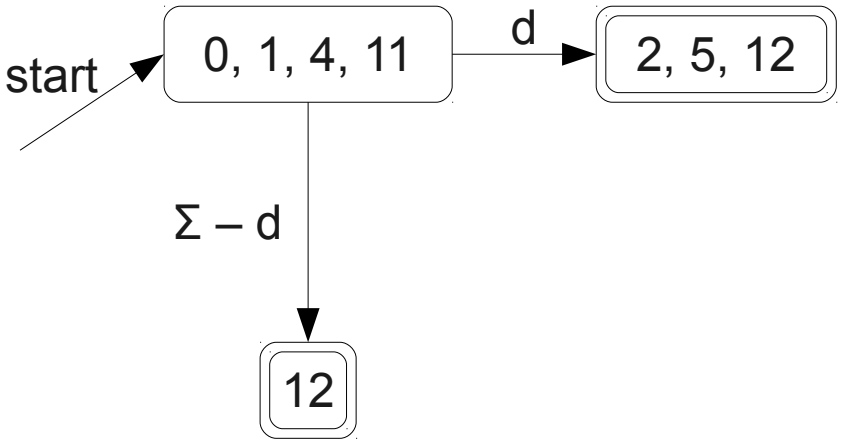
From NFA to DFA



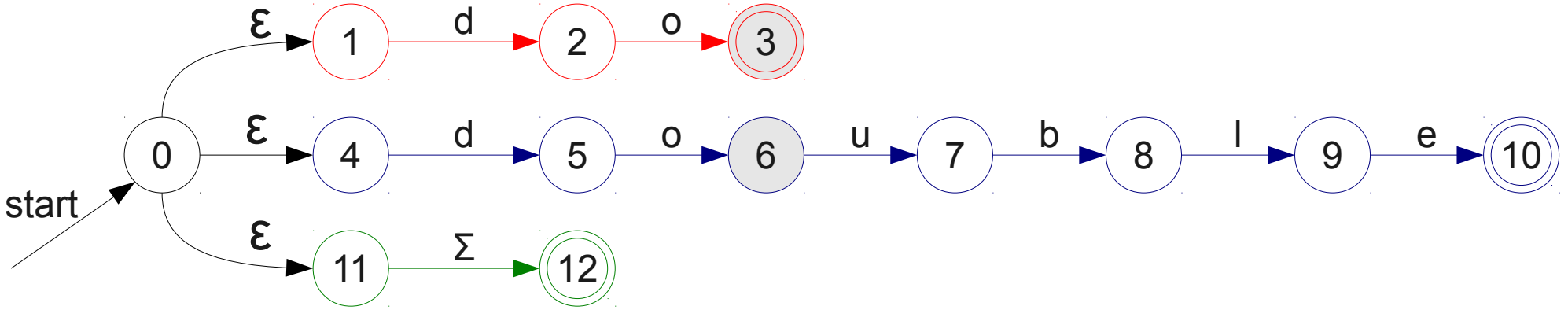


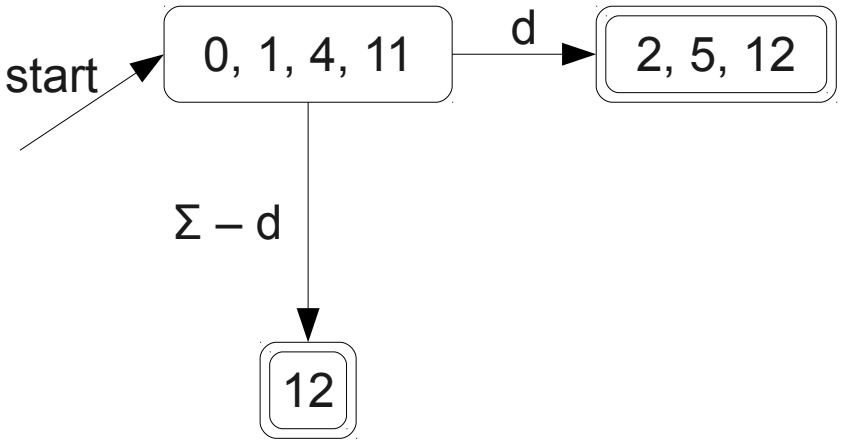
From NFA to DFA



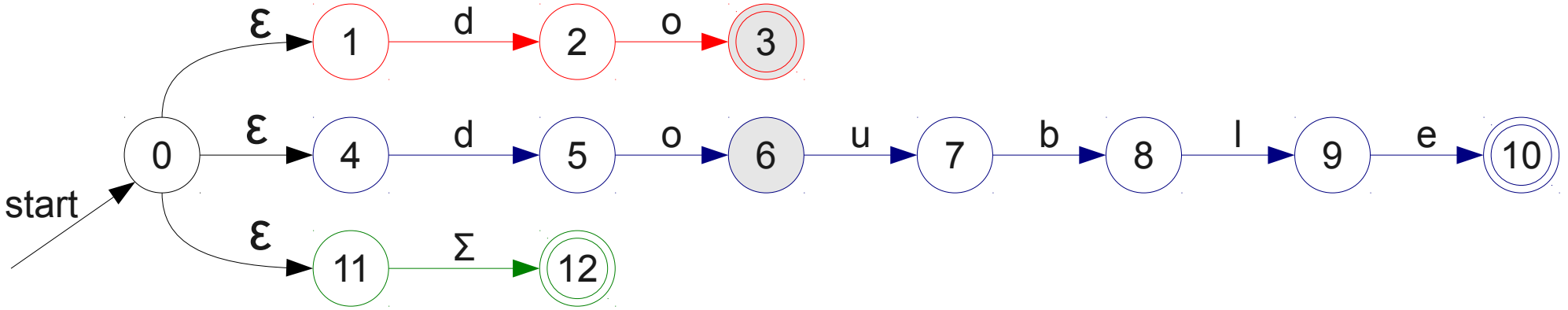


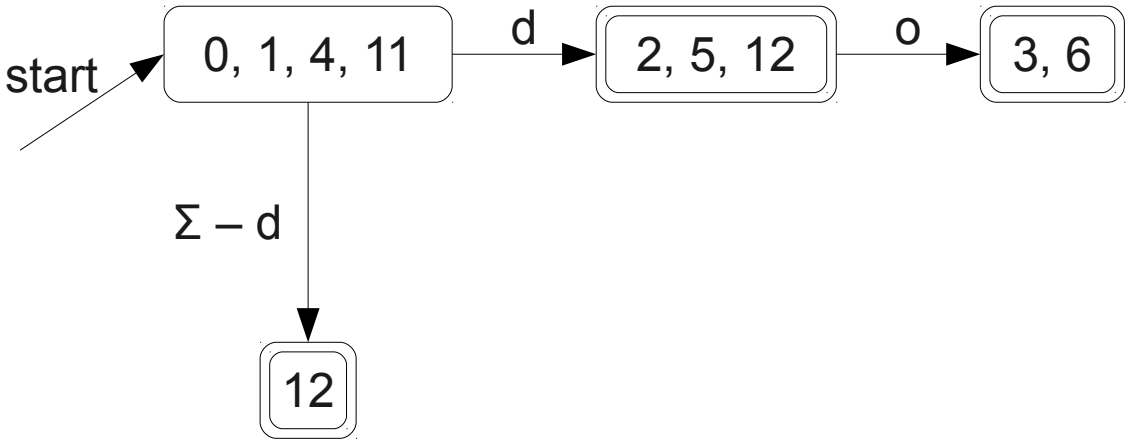
From NFA to DFA



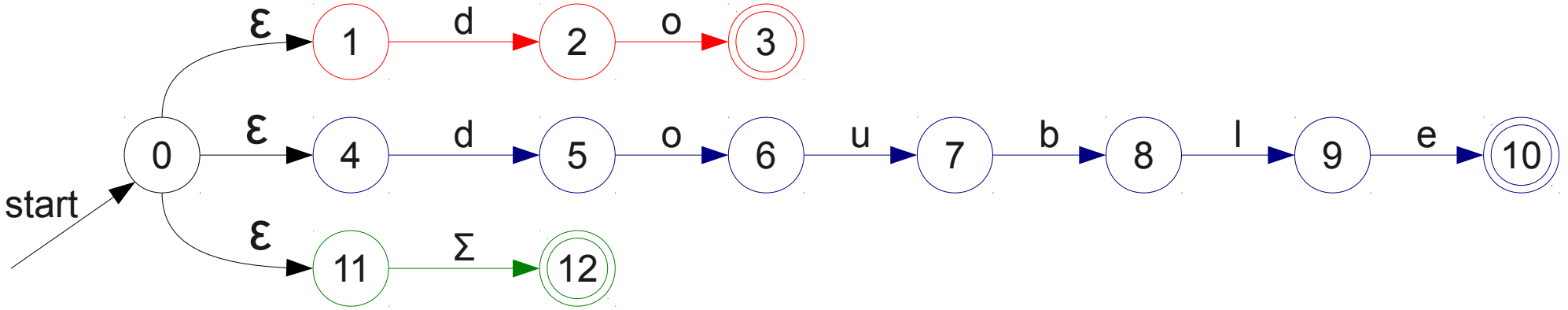


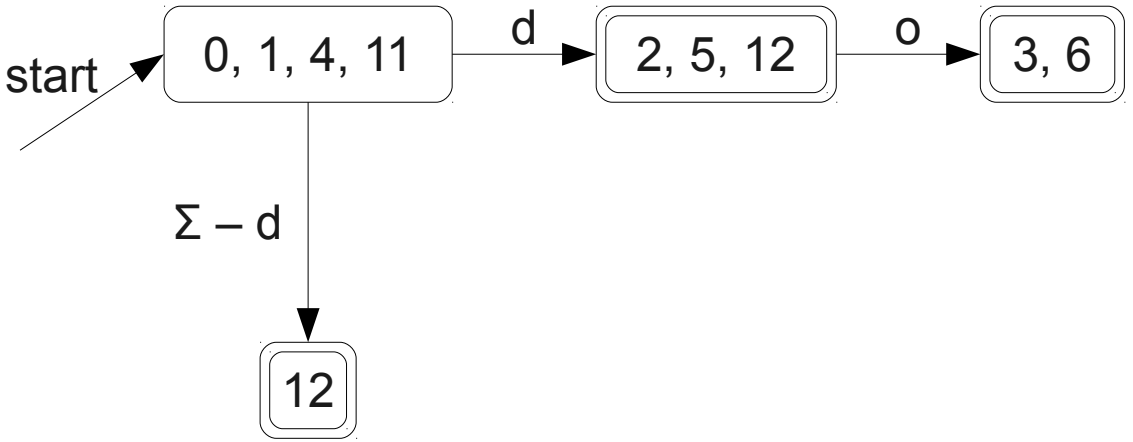
From NFA to DFA



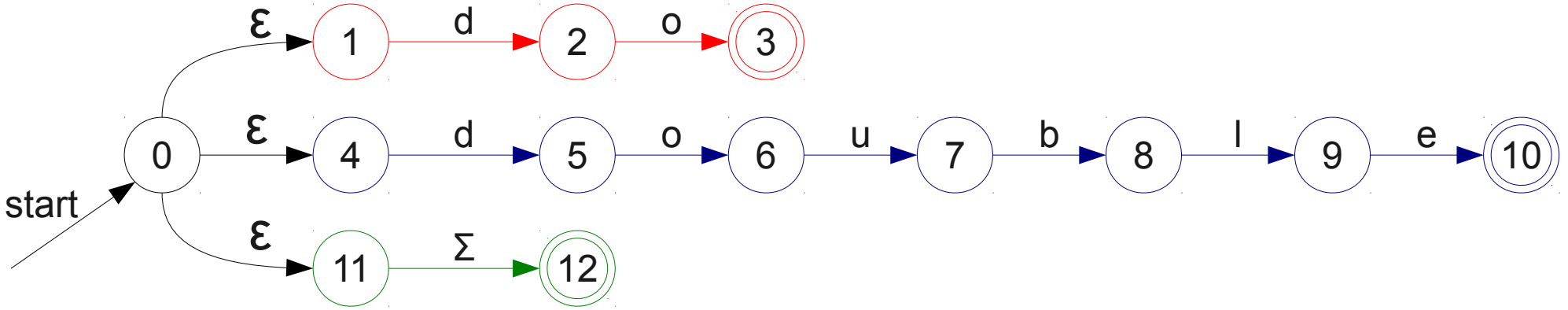


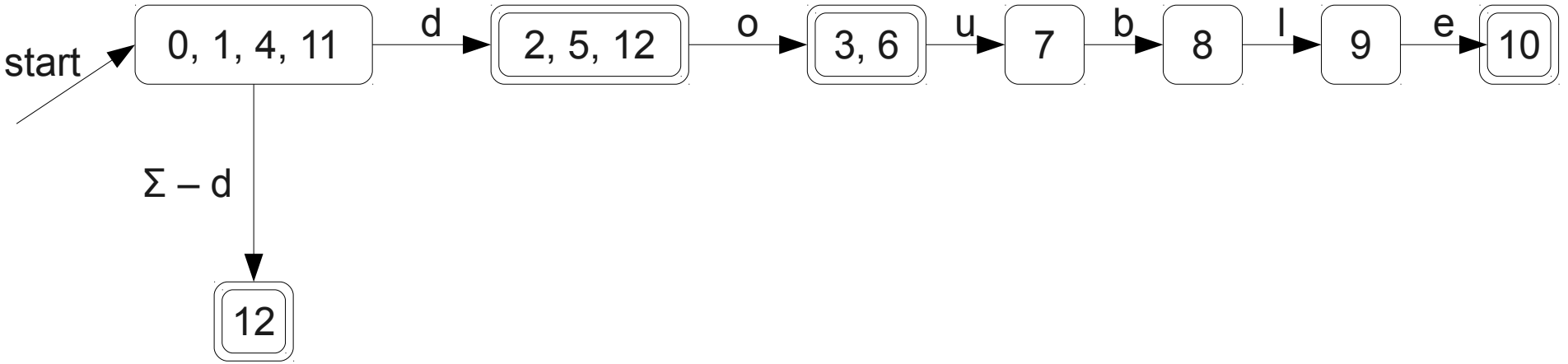
From NFA to DFA



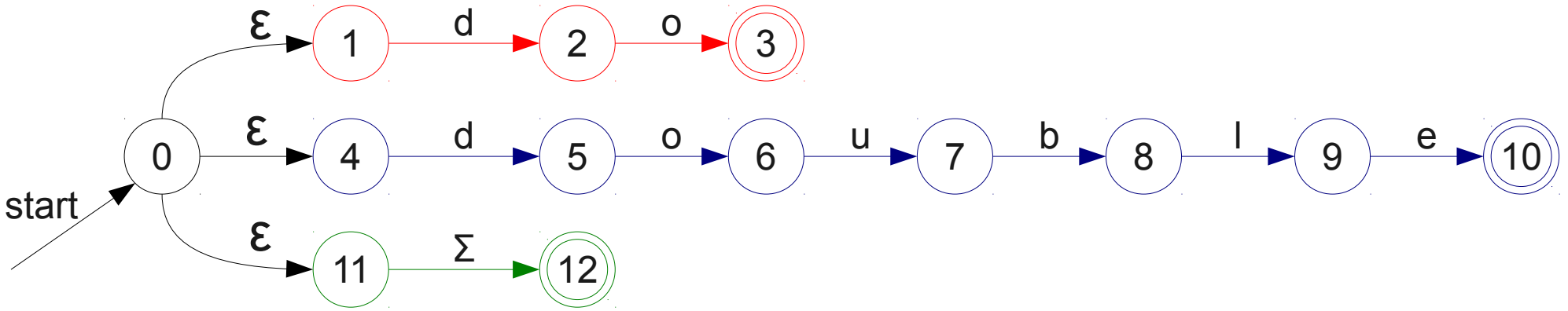


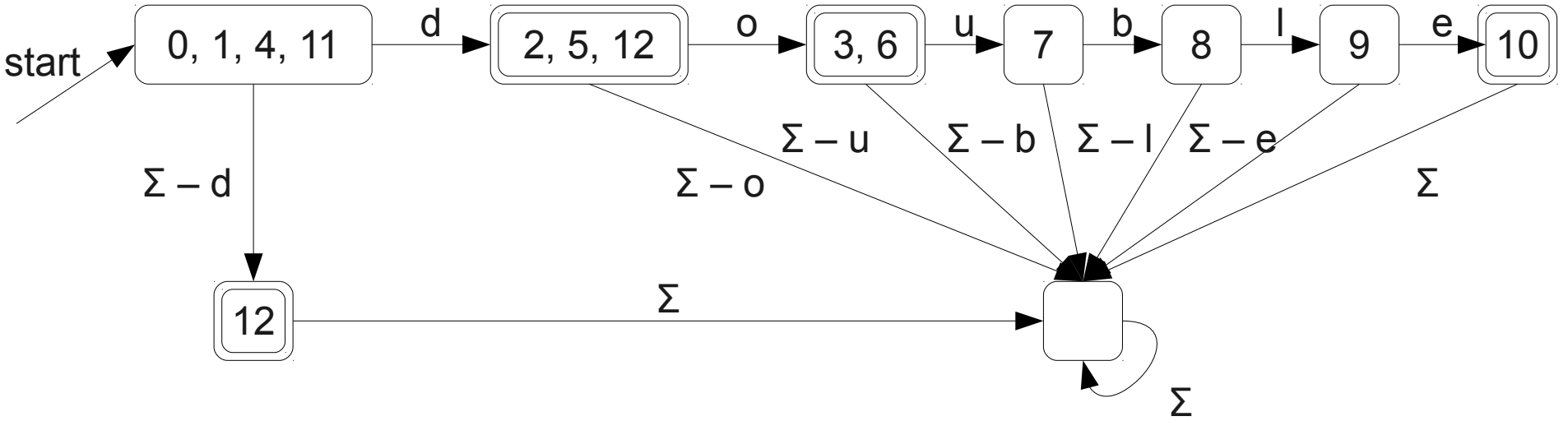
From NFA to DFA



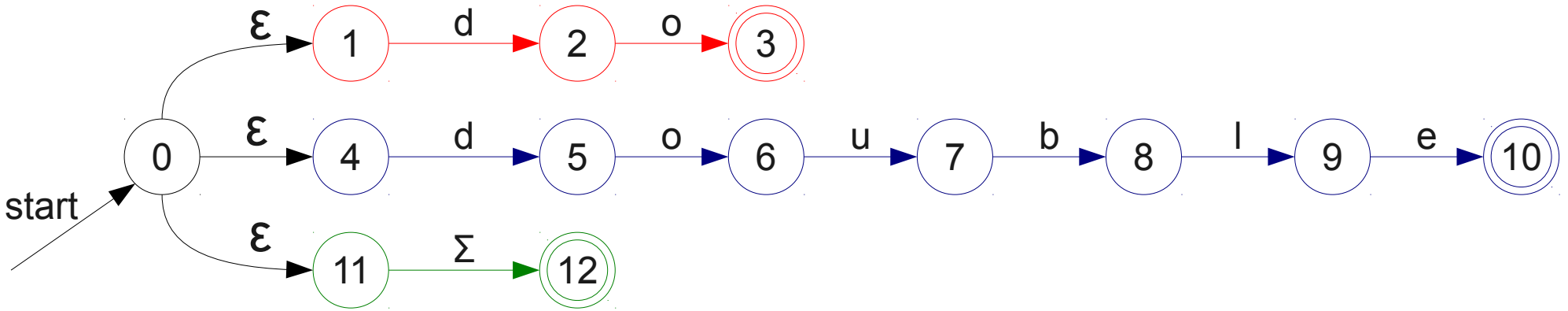


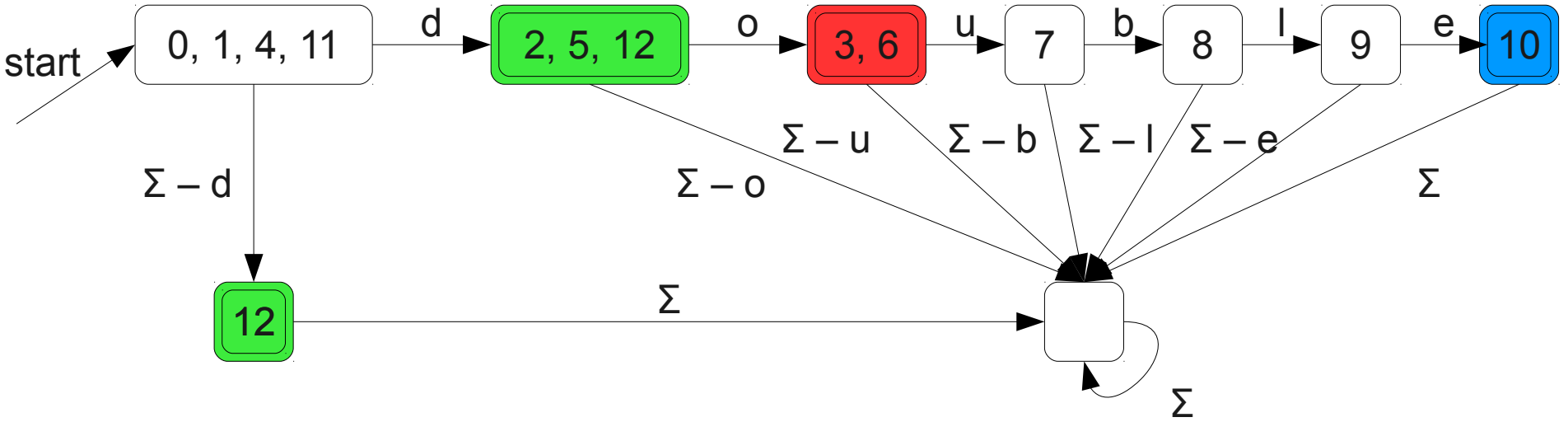
From NFA to DFA





From NFA to DFA





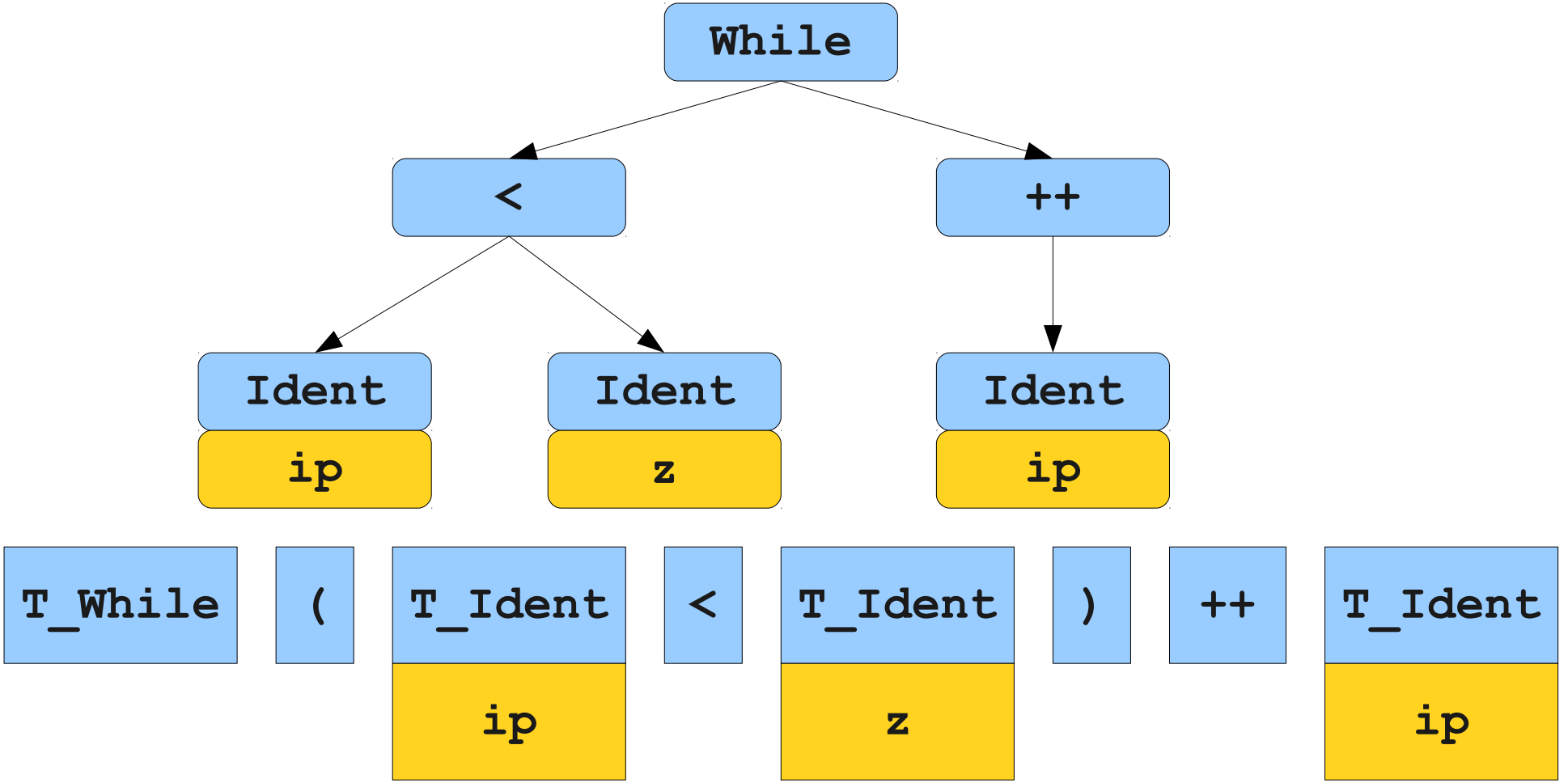
Modified Subset Construction

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

Performance Concerns

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

Real-World Scanning: **Python**



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

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

Python Blocks

- Scoping handled by whitespace:

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

- What does that mean for the scanner?

Whitespace Tokens

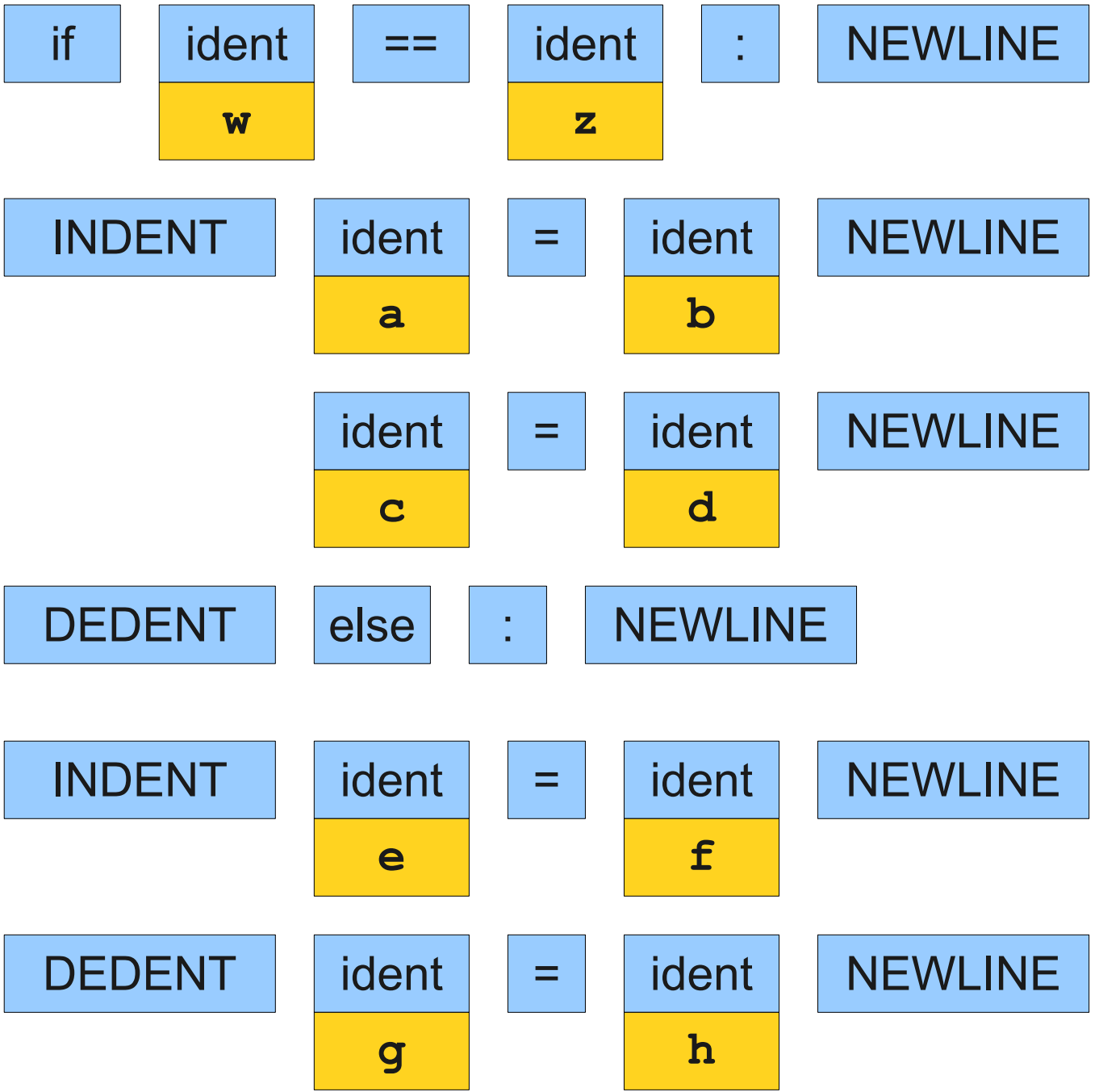
- Special tokens inserted to indicate changes in levels of indentation.
- **NEWLINE** marks the end of a line.
- **INDENT** indicates an increase in indentation.
- **DEDENT** indicates a decrease in indentation.
- Note that INDENT and DEDENT encode **change** in indentation, not the total amount of indentation.

Scanning Python

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

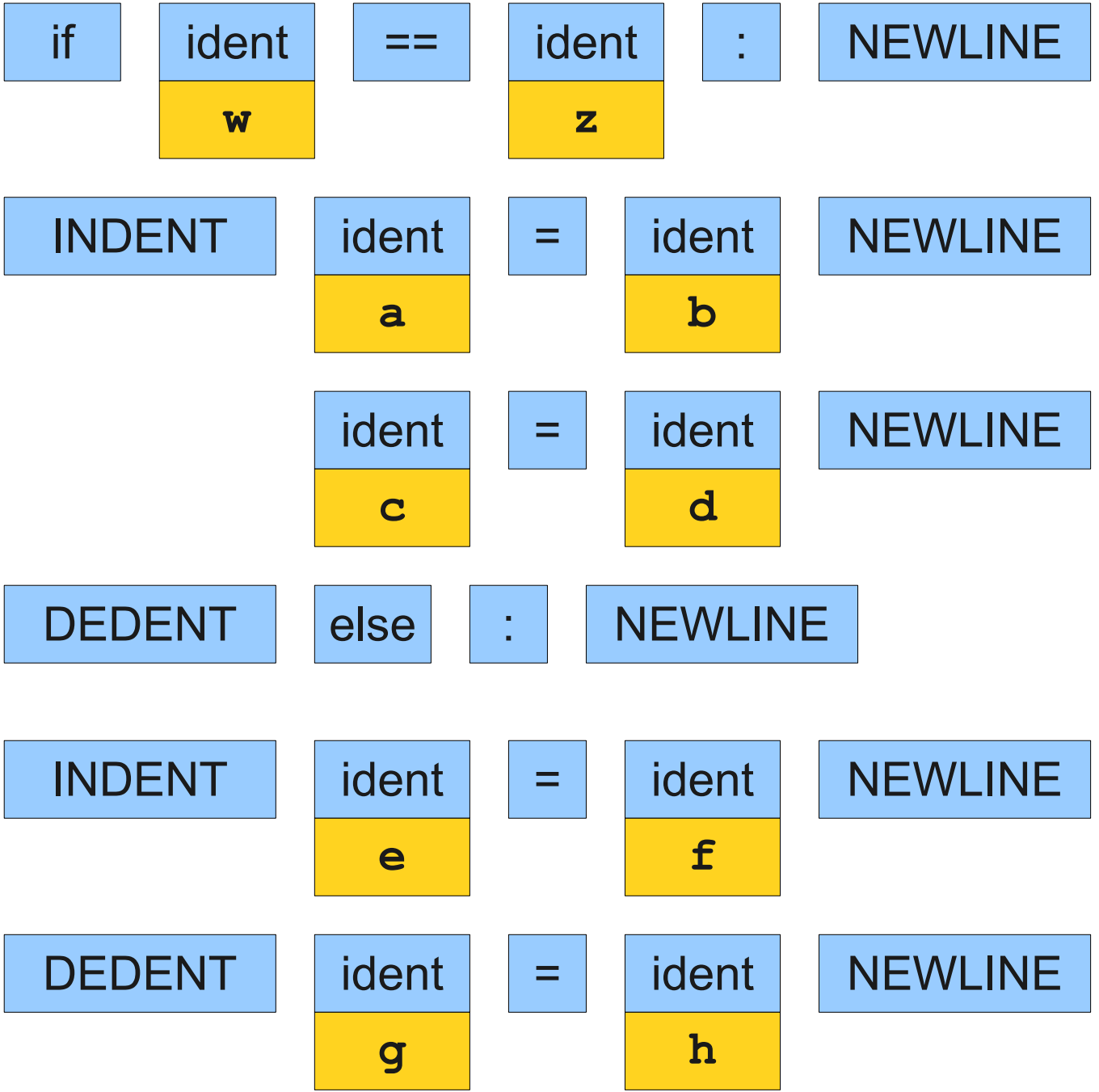
Scanning Python

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



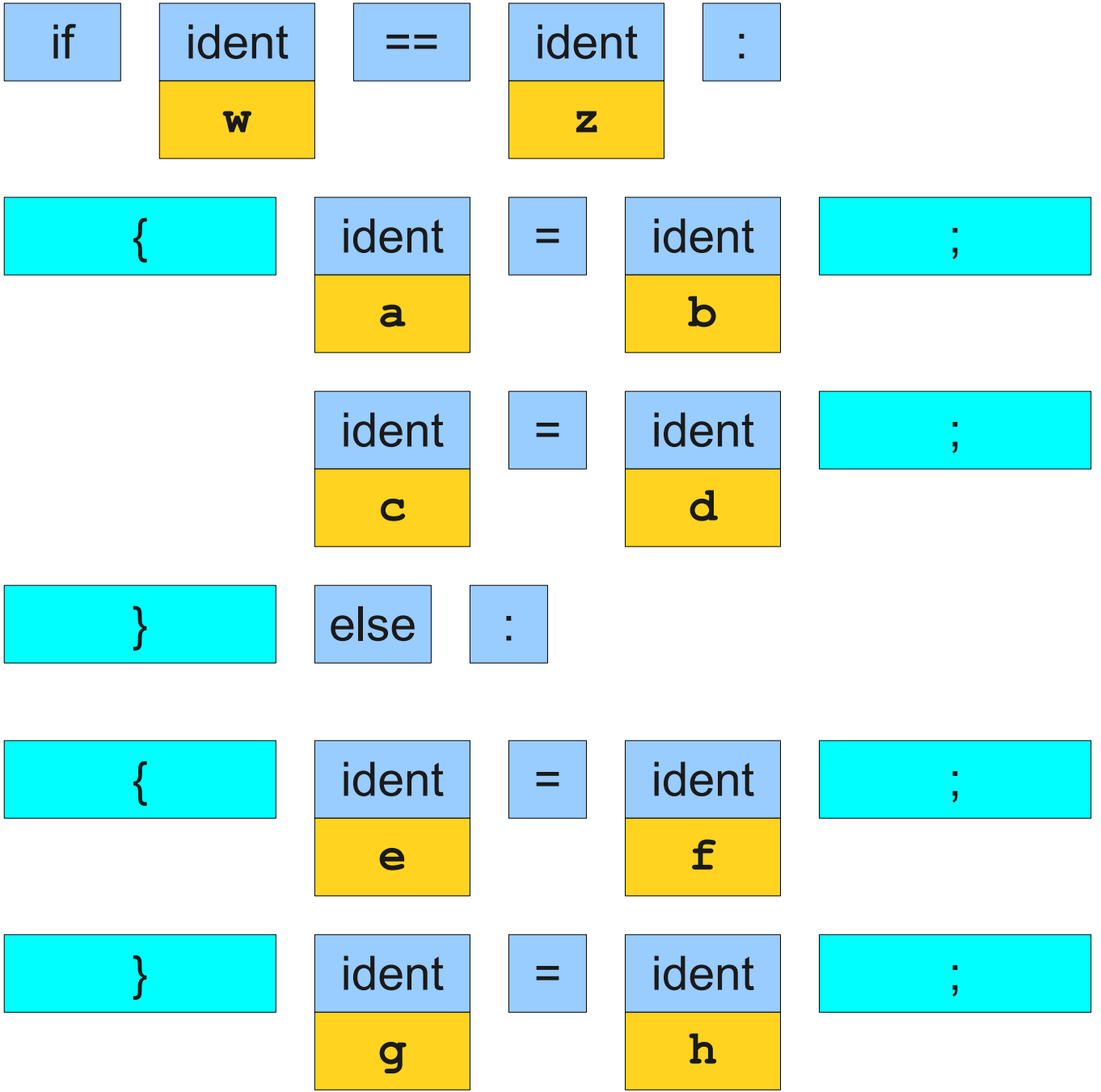
Scanning Python

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



Scanning Python

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



Where to INDENT/DEDENT?

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

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

Interesting Observation

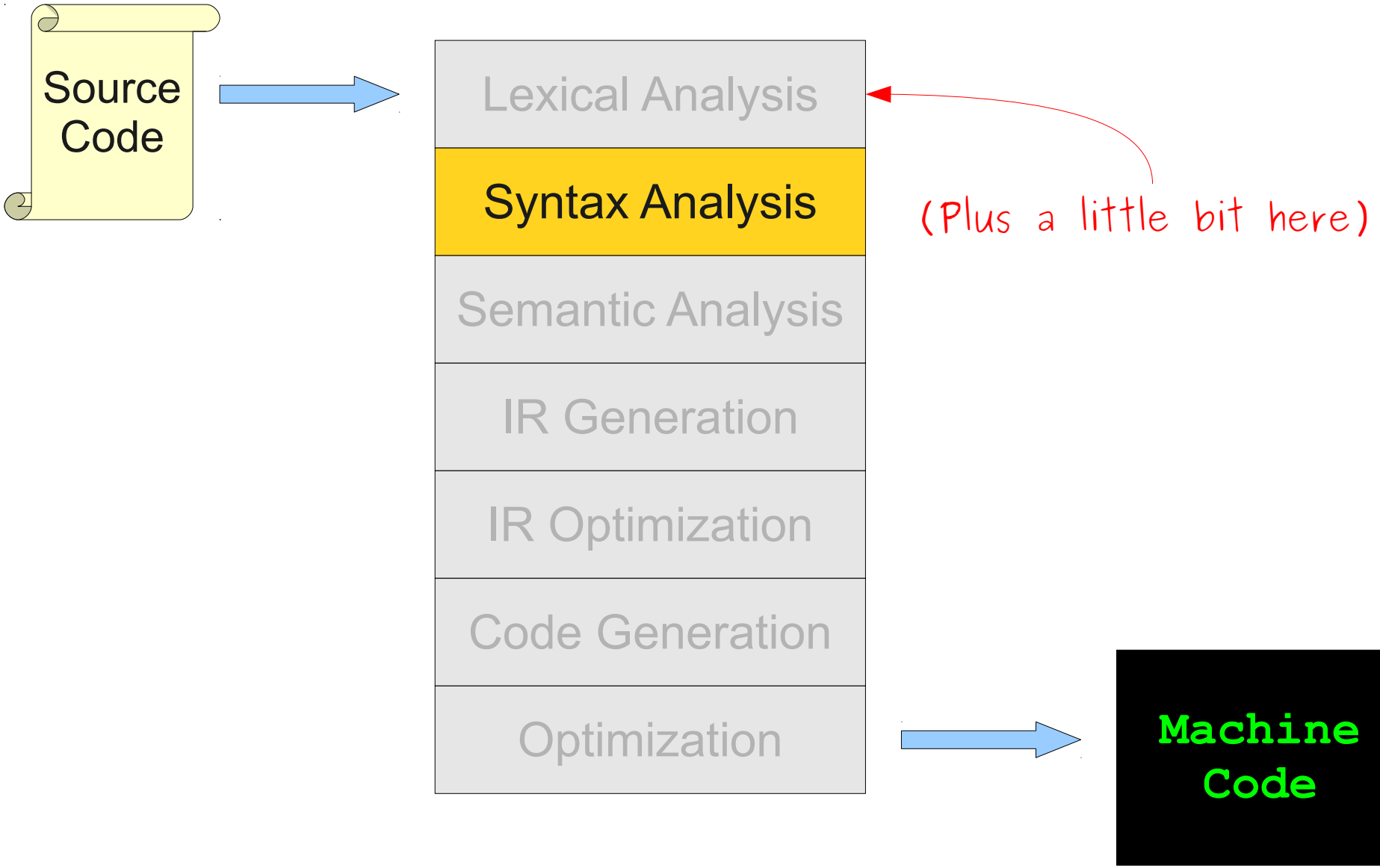
- Normally, more text on a line translates into more tokens.
- With DEDENT, **less** text on a line often means more tokens:

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

Summary

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

Next Time


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