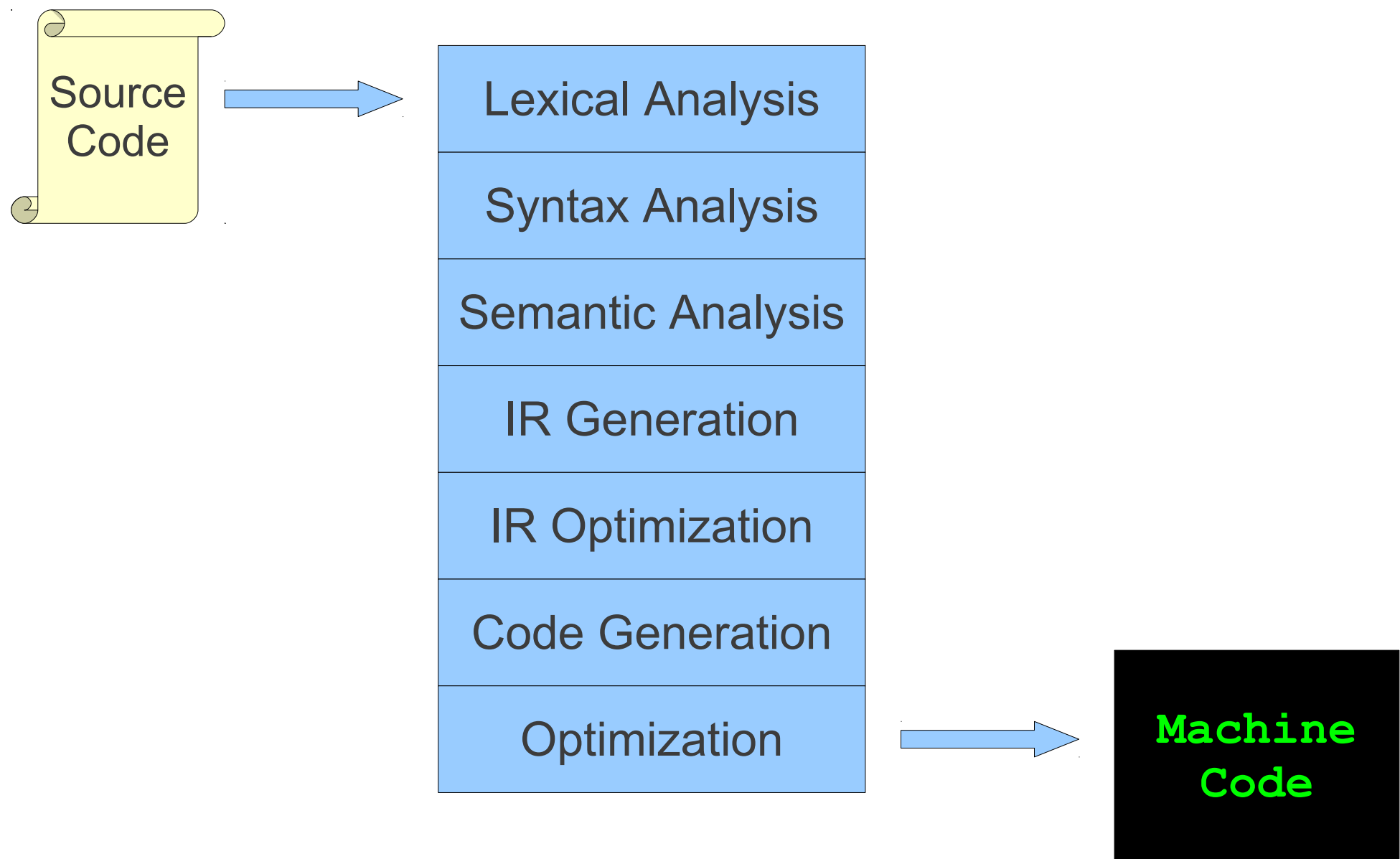


Lexical Analysis

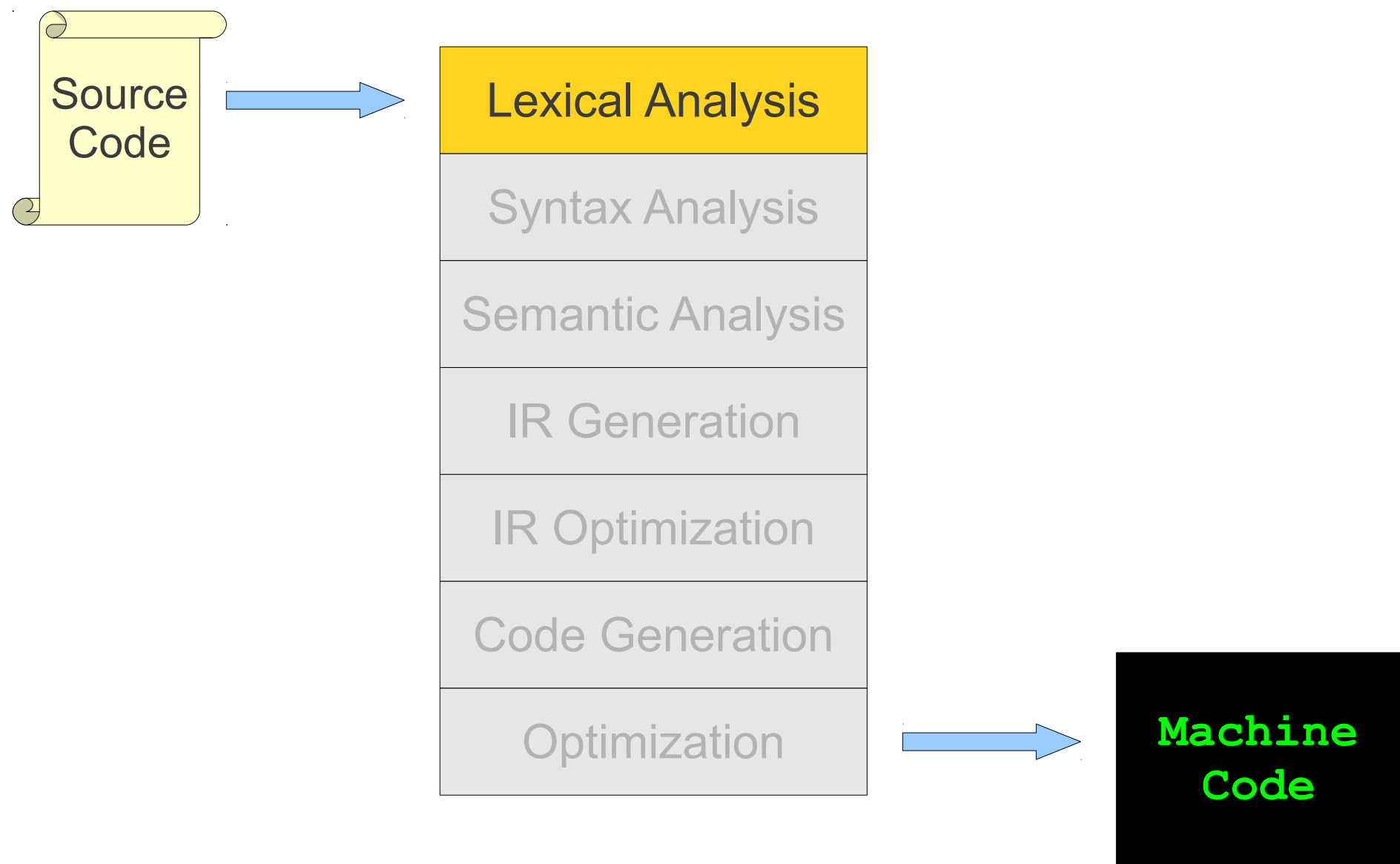
April 3, 2013

Previously on CSE 131b...

Structure of a modern compiler



Where are we?



Goal of Lexical Analysis

Breaking the program down into words or “tokens”

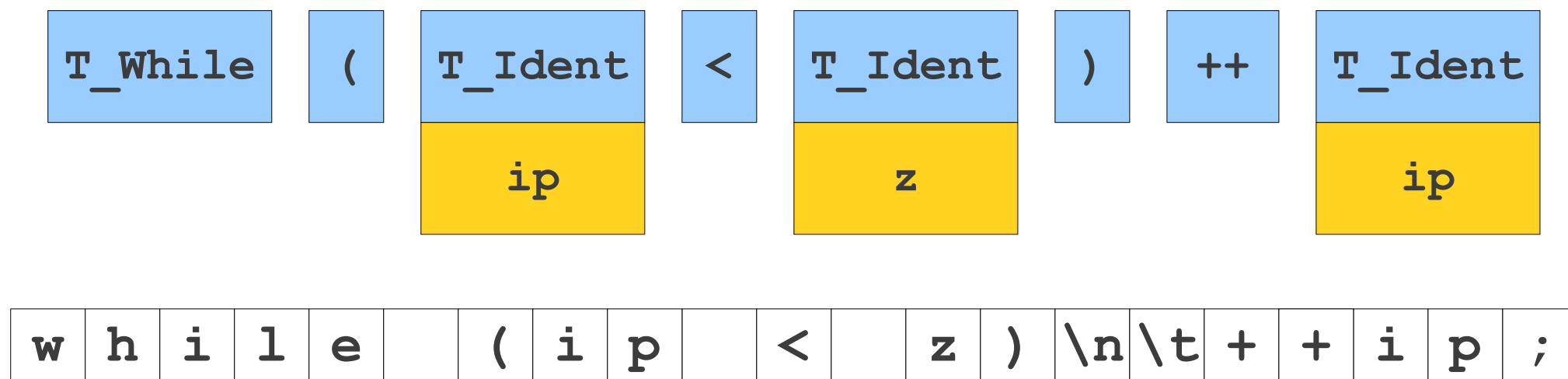
Input: code (character stream)

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

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

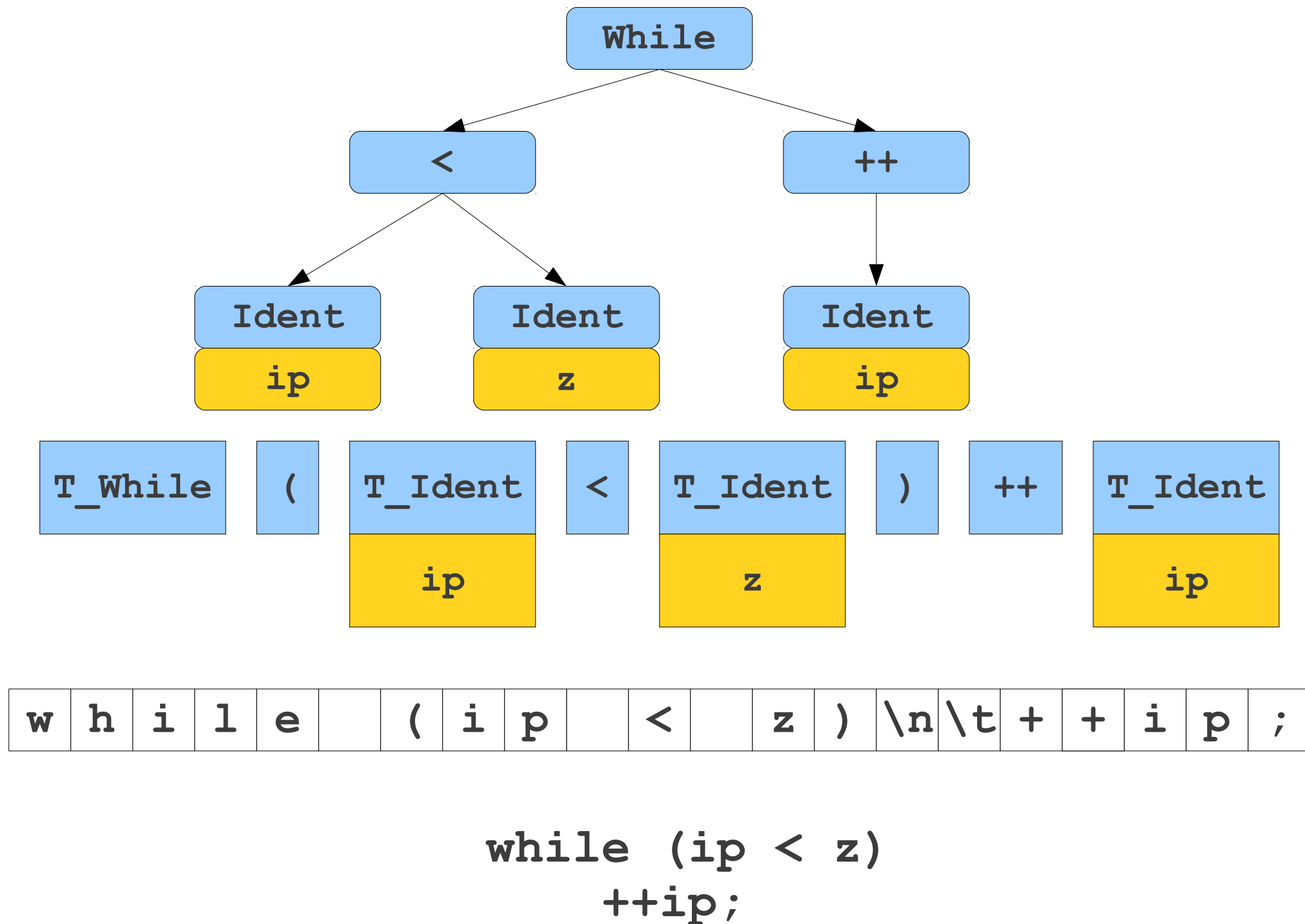
Goal of Lexical Analysis

Output: Token Stream



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

The Token Stream is then used as input for Parser (syntax analysis)



What's a token?

- What's a lexical unit of code?

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

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

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

What is my name ?

Token Type

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

Token Type

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

- Keyword: **for** **int** **if** **else** **while**

Token Type

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

- Keyword: **for** **int** **if** **else** **while**
- Punctuation: **(** **)** **{** **}** **;**

Token Type

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

- Keyword: **for** **int** **if** **else** **while**
- Punctuation: **(** **)** **{** **}** **;**
- Operand: **+** **-** **++**

Token Type

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

- Keyword: **for** **int** **if** **else** **while**
- Punctuation: **(** **)** **{** **}** **;**
- Operand: **+** **-** **++**
- Relation: **<** **>** **=**

Token Type

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

- Keyword: `for` `int` `if` `else` `while`
- Punctuation: `(` `)` `{` `}` `;`
- Operand: `+` `-` `++`
- Relation: `<` `>` `=`
- Identifier: (variable name,function name) `foo`
`foo_2`

Token Type

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

- Keyword: `for` `int` `if` `else` `while`
- Punctuation: `(` `)` `{` `}` `;`
- Operand: `+` `-` `++`
- Relation: `<` `>` `=`
- Identifier: (variable name,function name) `foo`
`foo_2`
- Integer, float point, string: `2345` `2.0` `"hello world"`

Token Type

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

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

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

Scanning a Source File

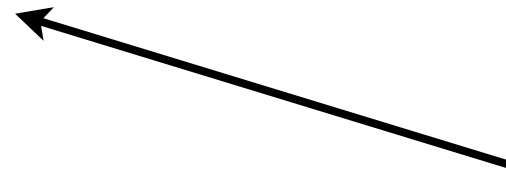
w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;
---	---	---	---	---	--	---	---	---	---	--	---	--	---	---	----	----	---	---	---	---

T_While

Scanning a Source File

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

T_While



Token

Scanning a Source File

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

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

T_While

Token

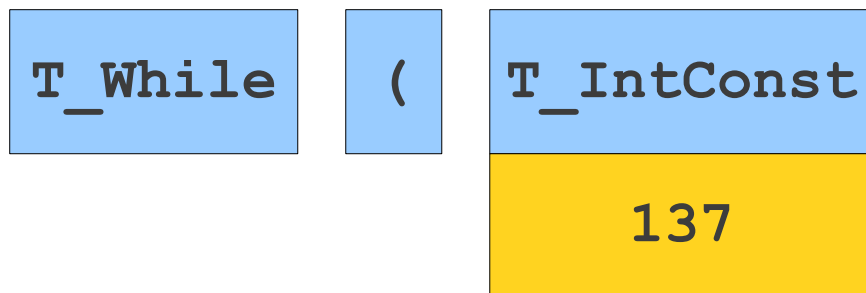
Scanning a Source File

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

T_While	(	T_IntConst
		137

Scanning a Source File

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



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

Lexical Analyzer

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

Why is this process hard?

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

DO 5 I = 1,25

DO5I = 1.25

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- C++: Nested template declarations

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

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

Thanks to Prof. Alex Aiken

Challenges for Lexical Analyzer

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

Associate Lexemes to Tokens

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

Token: Lexemes

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

Token: Lexemes

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

Token: Lexemes

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

- How do we describe which (potentially infinite) set of lexemes is associated with each token type?

Formal Languages

- A **formal language** is a set of strings.
- Many infinite languages have finite descriptions:
 - Define the language using an automaton.
 - Define the language using a grammar.
 - Define the language using a regular expression.
- We can use these compact descriptions of the language to define sets of strings.
- Over the course of this class, we will use all of these approaches.

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

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (the *regular languages*).
- Often provide a compact and human-readable description of the language.
- Used as the basis for numerous software systems, including the **flex** tool we will use in this course.

Atomic Regular Expressions

- The regular expressions we will use in this course begin with two simple building blocks.
- The symbol ϵ is a regular expression matches the empty string.
- For any symbol a , the symbol a is a regular expression that just matches a .

Compound Regular Expressions

- If R_1 and R_2 are regular expressions, $\mathbf{R_1R_2}$ is a regular expression representing the **concatenation** of the languages of R_1 and R_2 .
- If R_1 and R_2 are regular expressions, $\mathbf{R_1 \mid R_2}$ is a regular expression representing the **union** of R_1 and R_2 .
- If R is a regular expression, $\mathbf{R^*}$ is a regular expression for the **Kleene closure** of R .
- If R is a regular expression, $\mathbf{(R)}$ is a regular expression with the same meaning as R .

Simple Regular Expressions

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

$(0 \mid 1)^*00(0 \mid 1)^*$

Simple Regular Expressions

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

$(0 \mid 1)^*00(0 \mid 1)^*$

Simple Regular Expressions

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

$(0 | 1)^*00(0 | 1)^*$

11011100101
0000
1111101111001111

Simple Regular Expressions

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

$(0 | 1)^*00(0 | 1)^*$

11011100101
0000
11111011110011111

Applied Regular Expressions

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

?

Applied Regular Expressions

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

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

Applied Regular Expressions

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

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

42
+1370
-3248
-9999912

- More examples
 - Whitespace: `[ \t\n]+`
 - Integers: `[+\-]?[0-9]+`
 - Hex numbers: `0x[0-9a-f]+`
 - identifier

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

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

Recognizing Regular Language

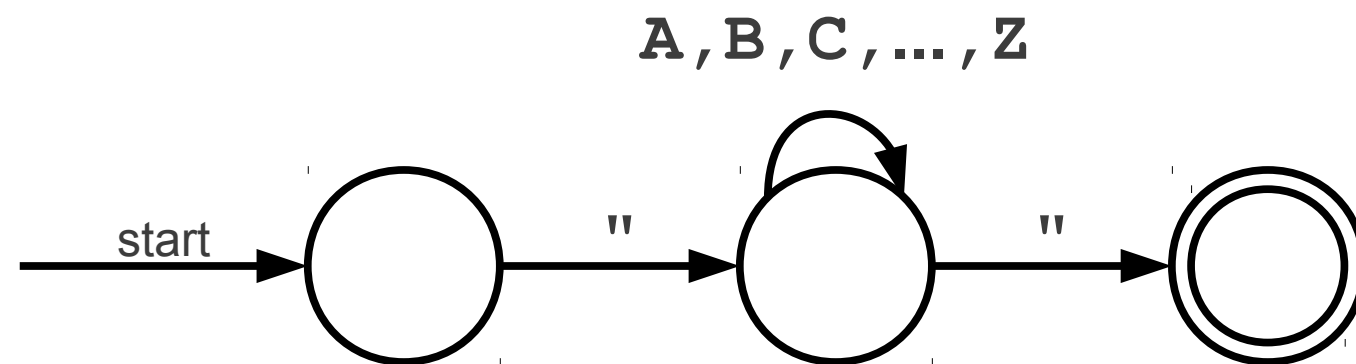
What is the machine that recognize regular language??

Recognizing Regular Language

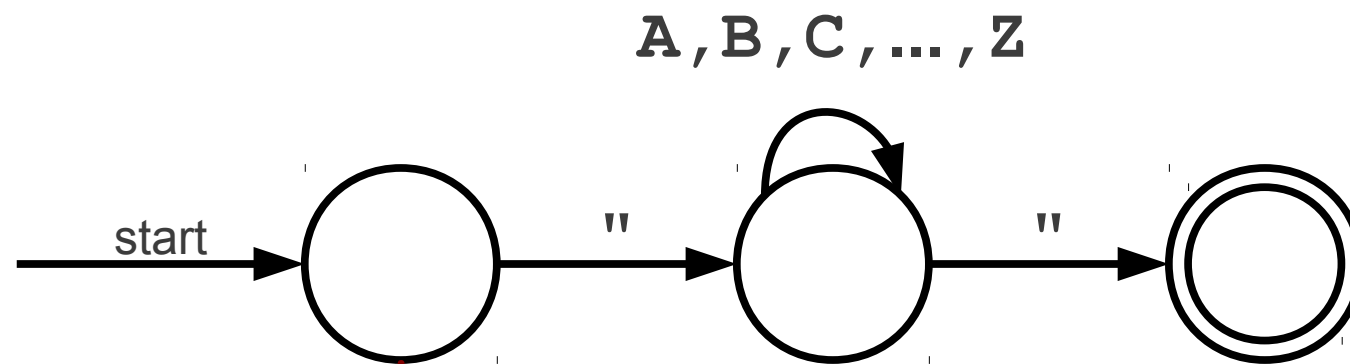
What is the machine that recognize regular language??

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

A Simple Automaton

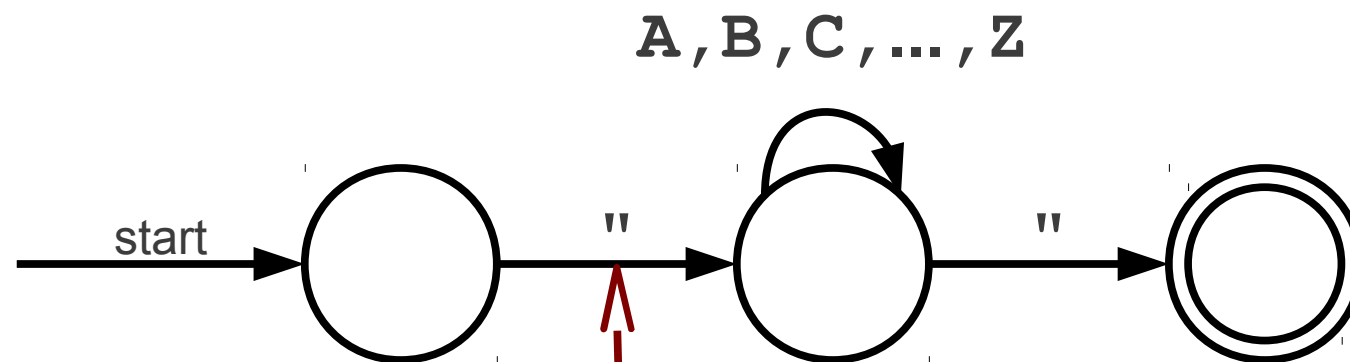


A Simple Automaton



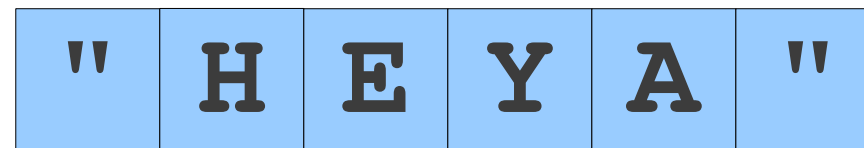
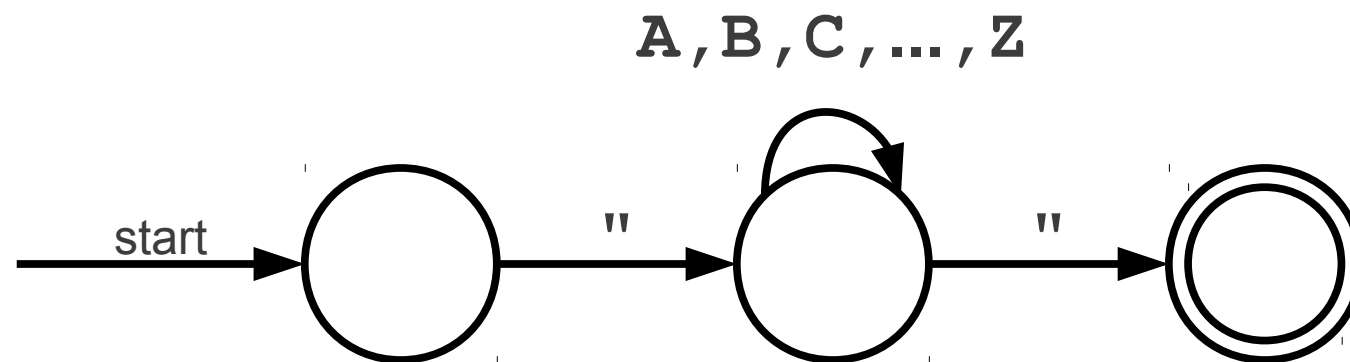
Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

A Simple Automaton



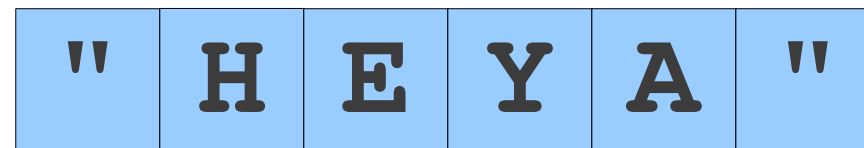
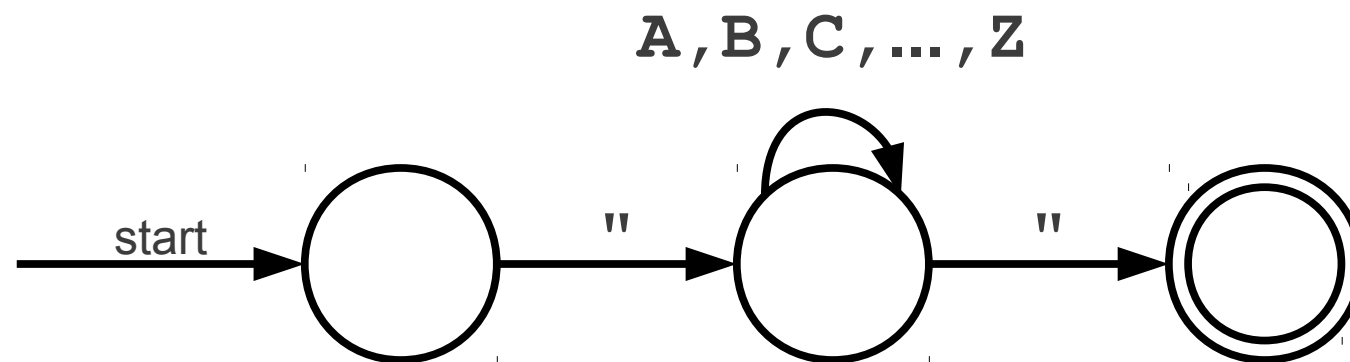
These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

A Simple Automaton



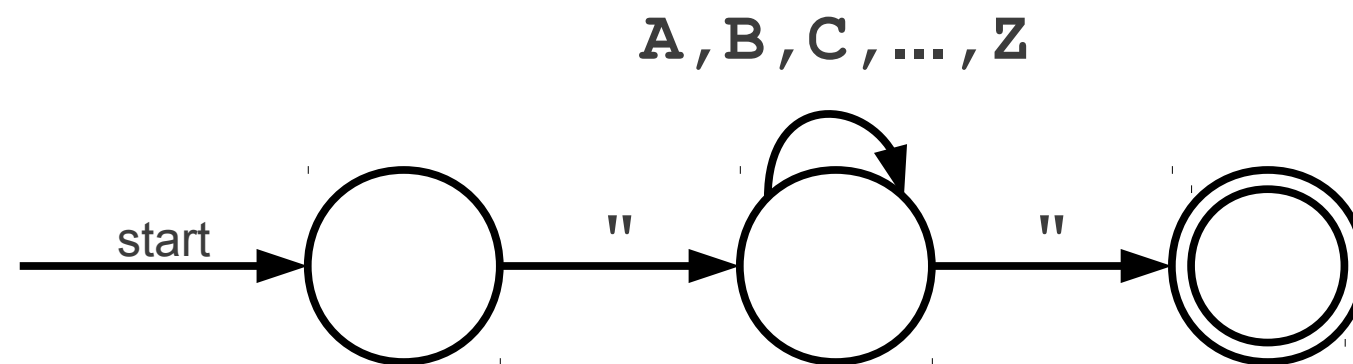
Finite Automata: Takes an input string and determines whether it's a valid sentence of a language
accept or reject

A Simple Automaton

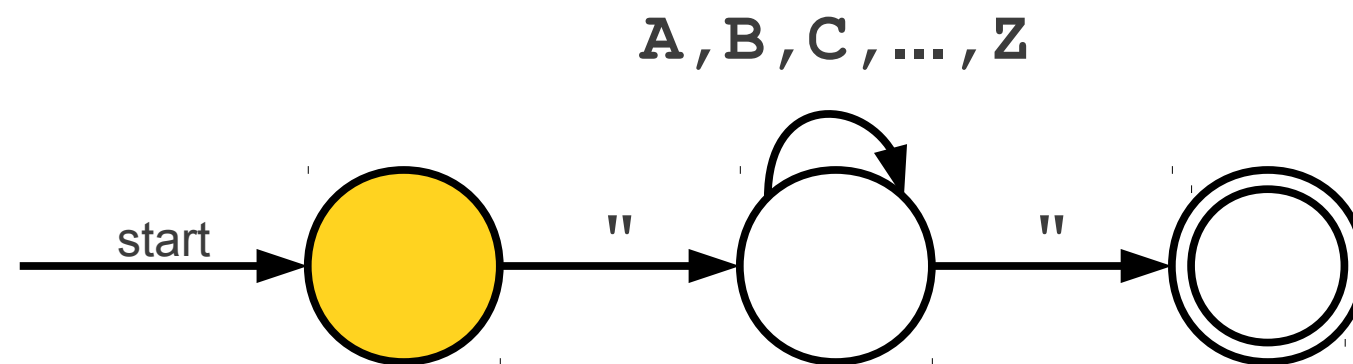


The automaton takes a string as input and decides whether to accept or reject the string.

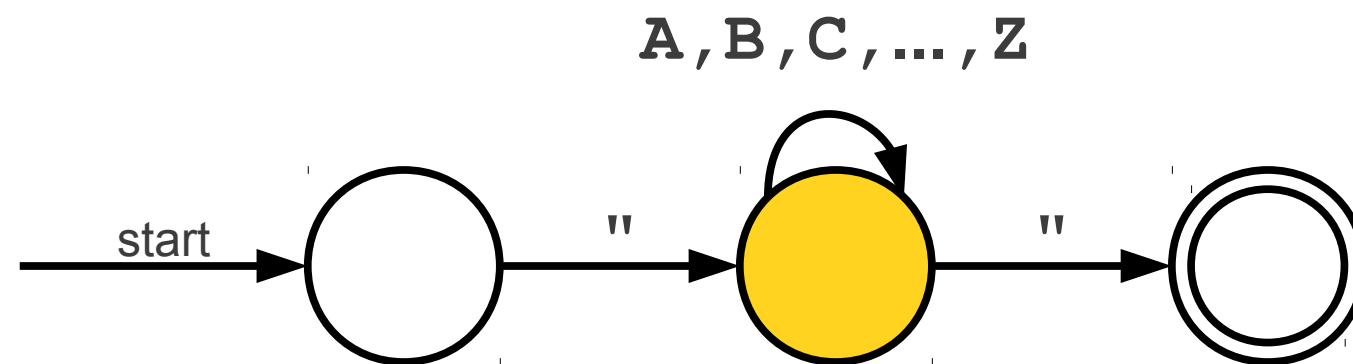
A Simple Automaton



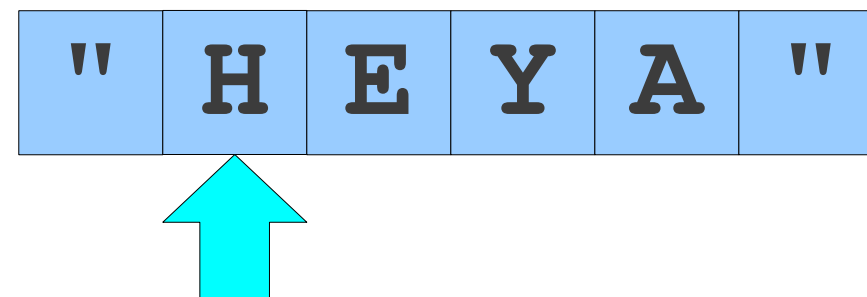
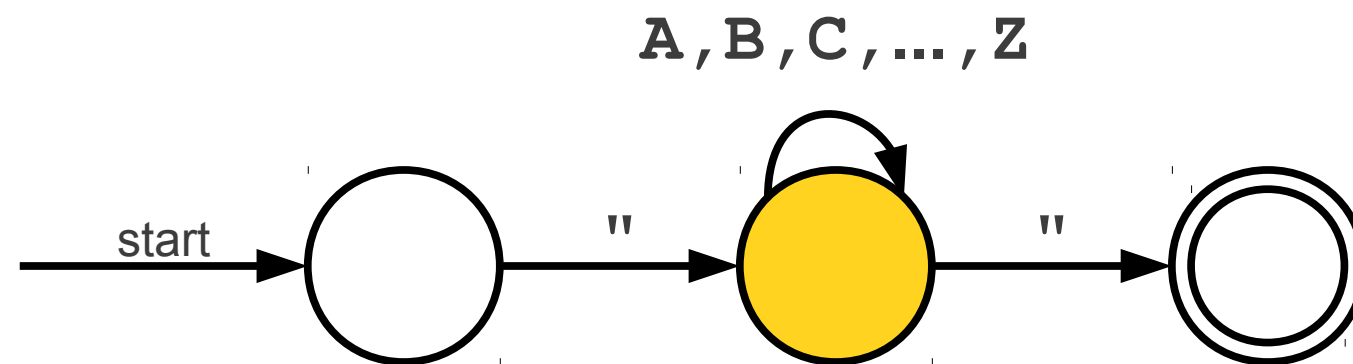
A Simple Automaton



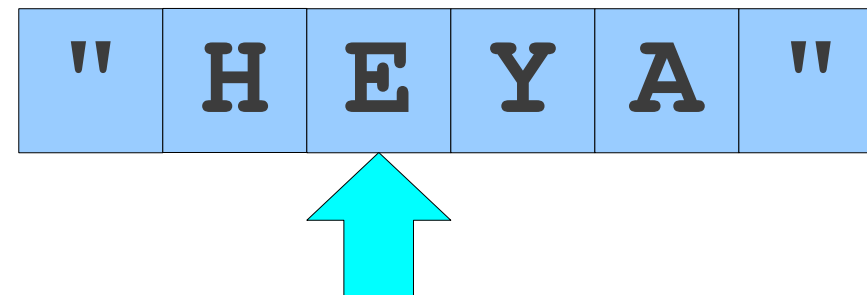
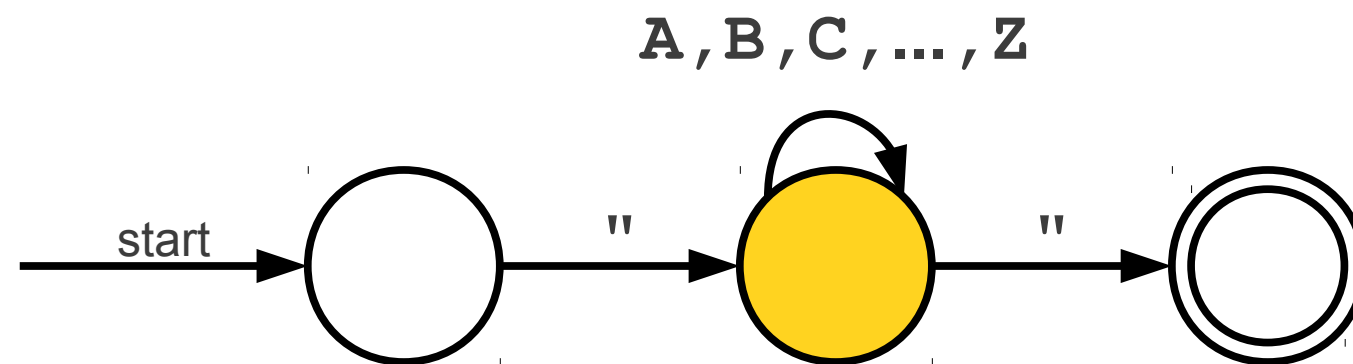
A Simple Automaton



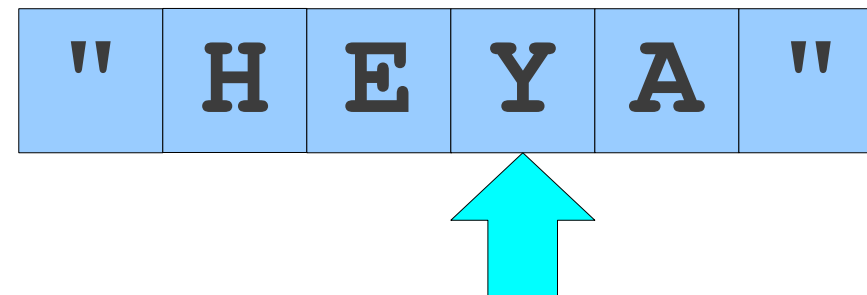
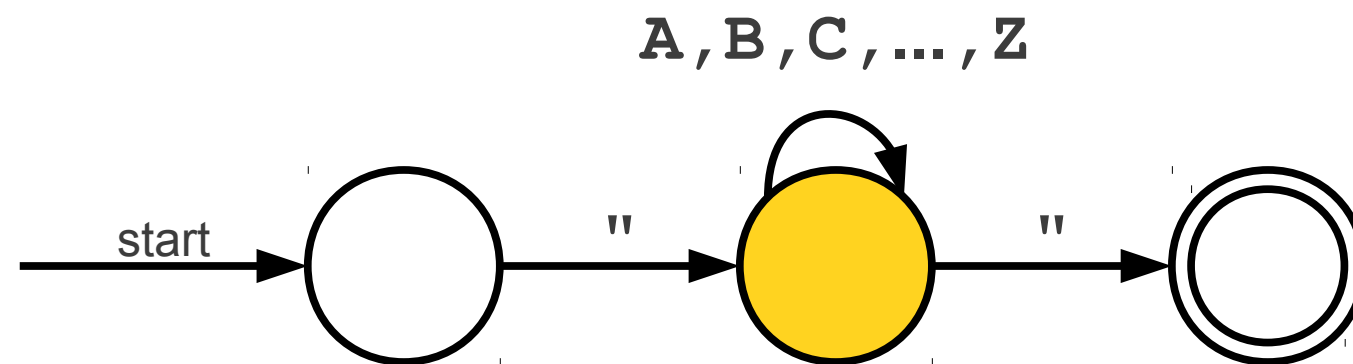
A Simple Automaton



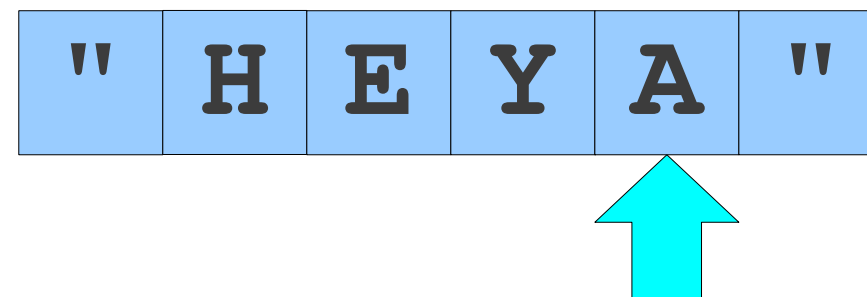
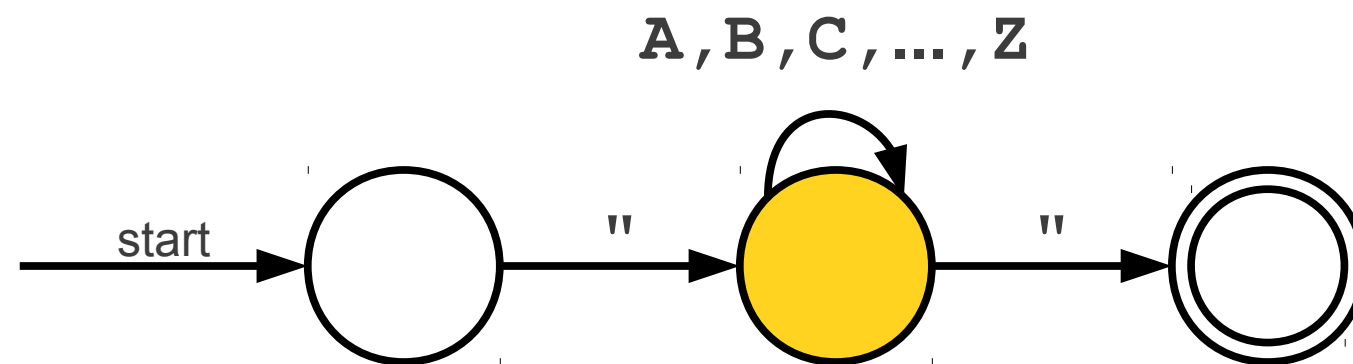
A Simple Automaton



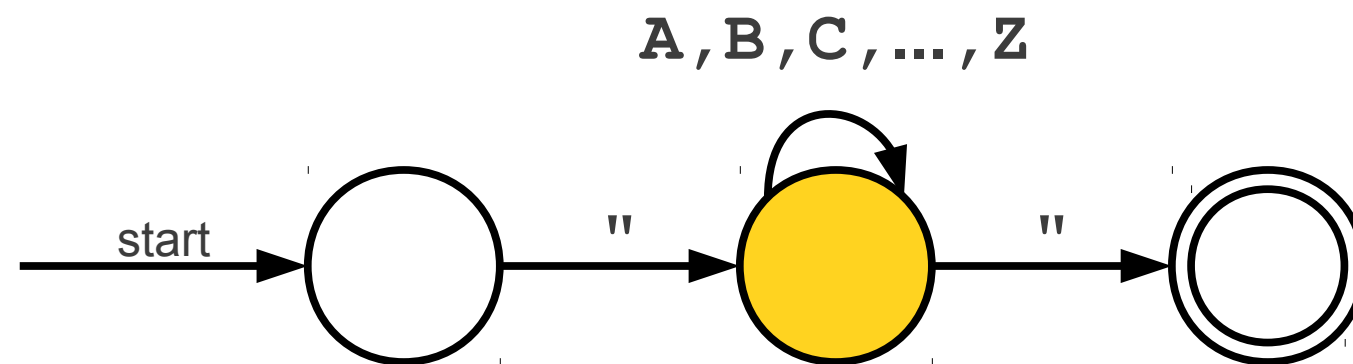
A Simple Automaton



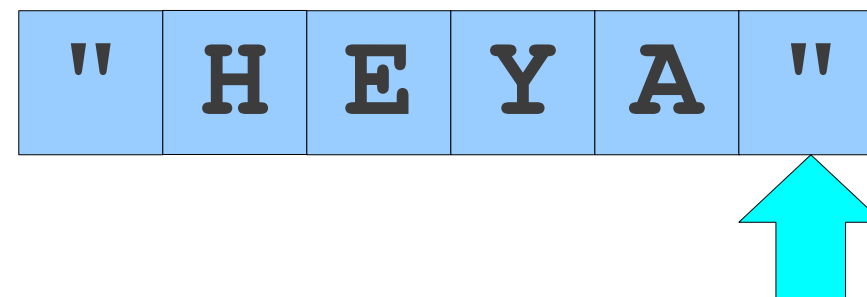
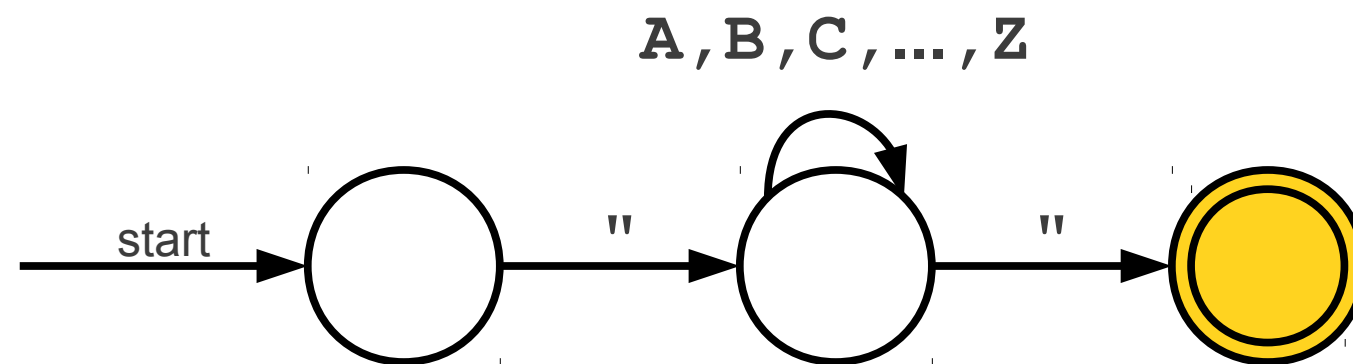
A Simple Automaton



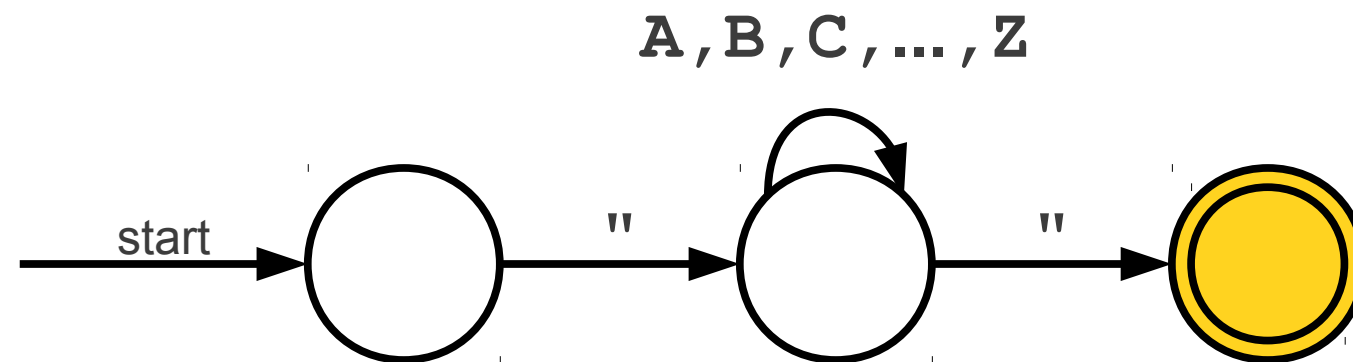
A Simple Automaton



A Simple Automaton

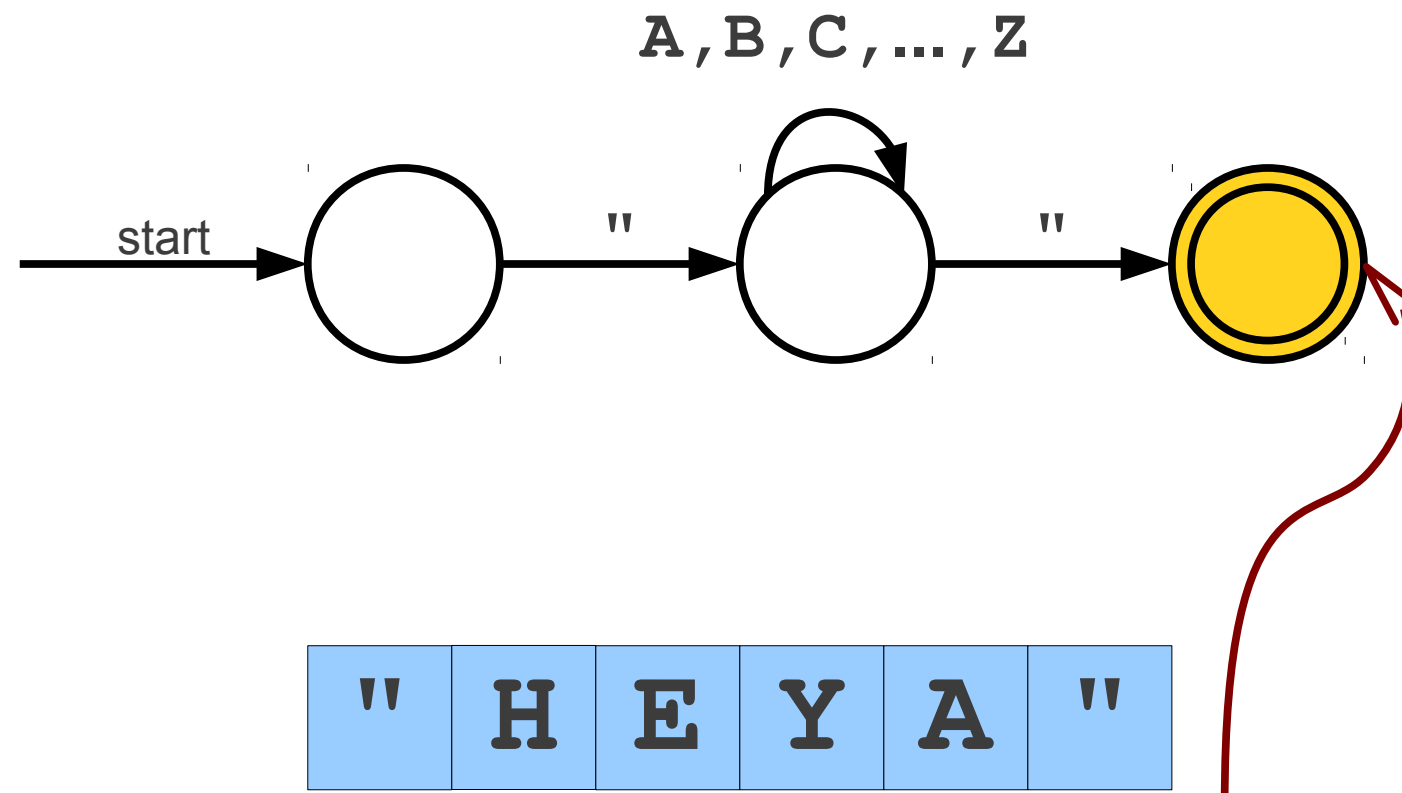


A Simple Automaton



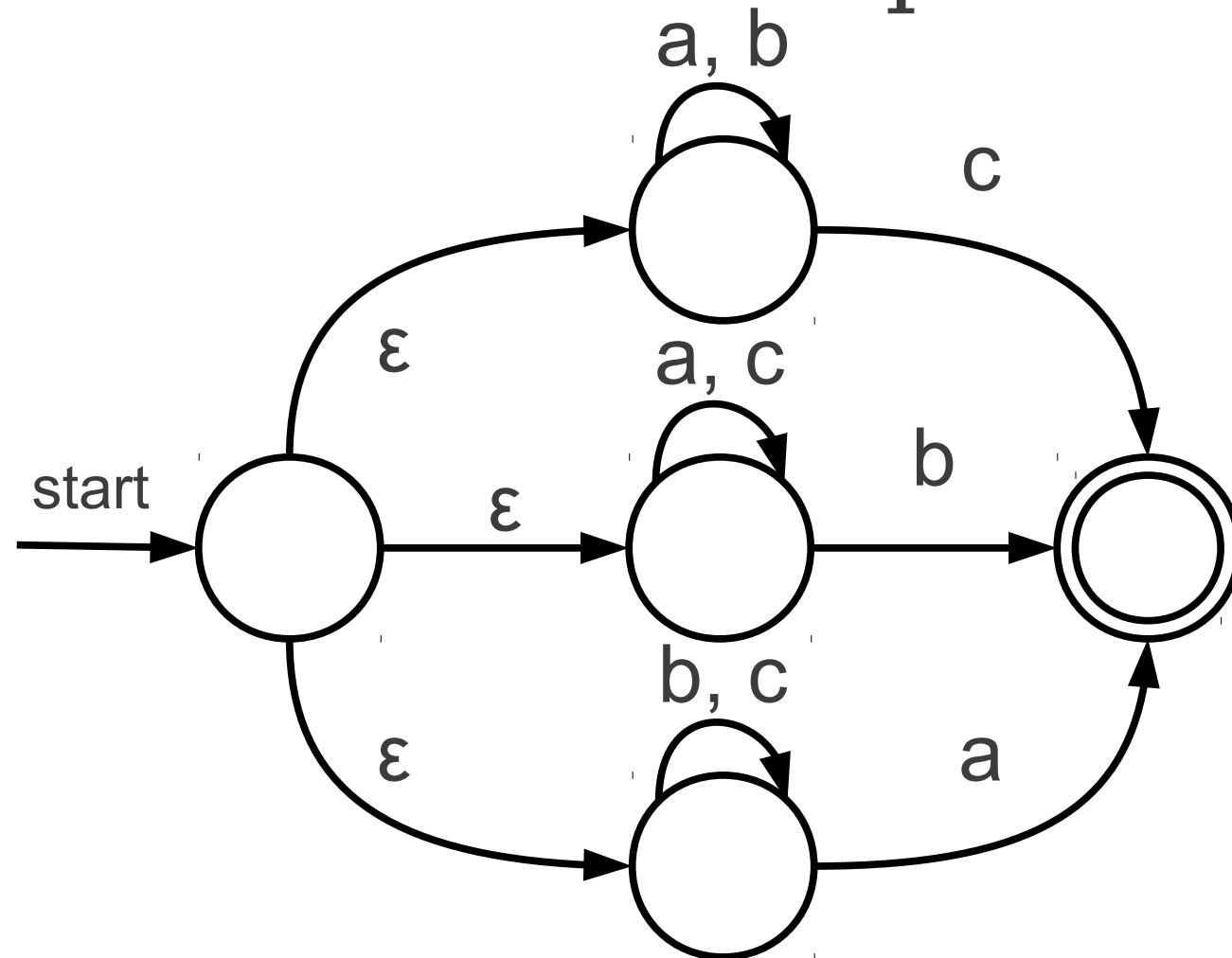
"	H	E	Y	A	"
---	---	---	---	---	---

A Simple Automaton

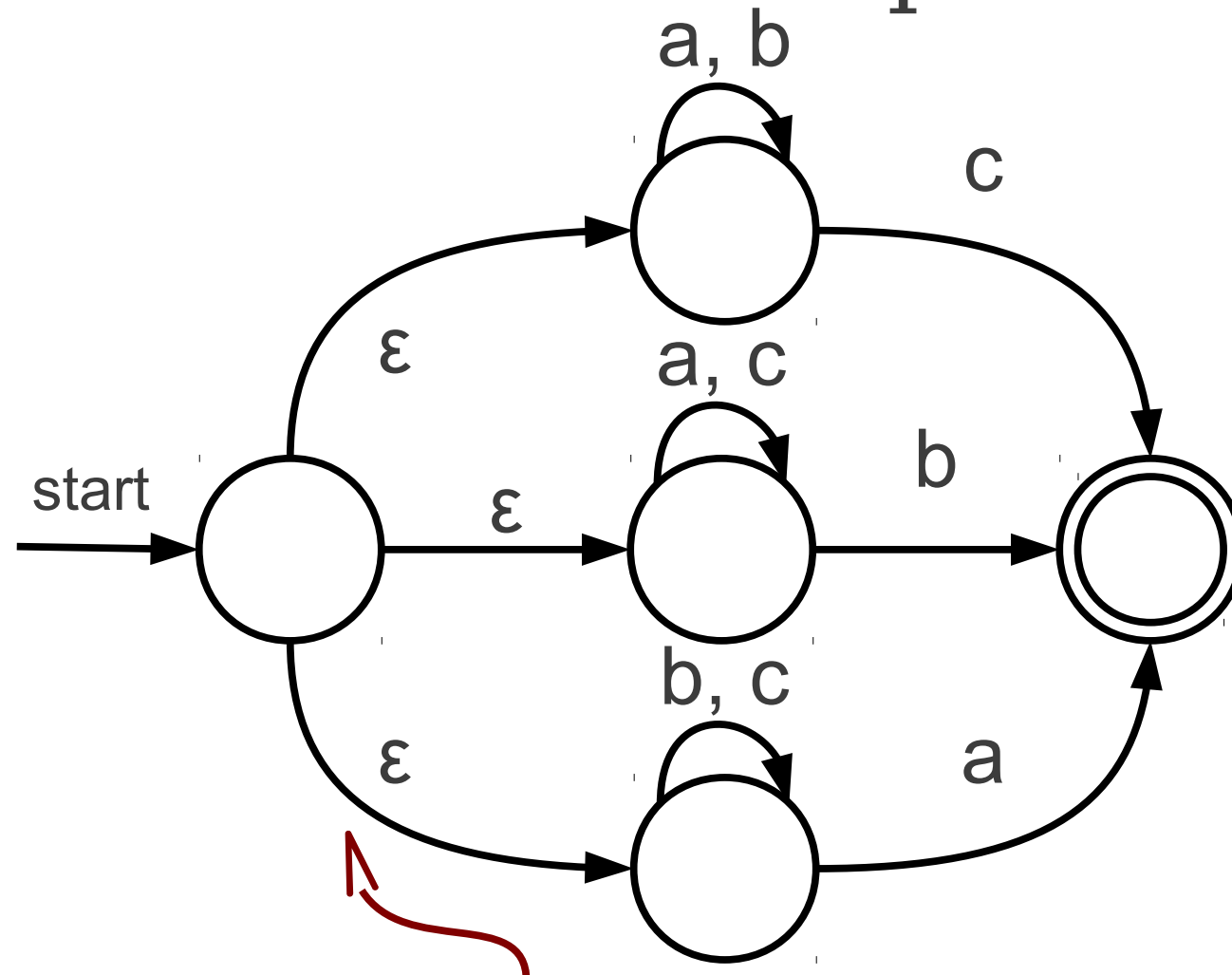


The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.

An Even More Complex Automaton

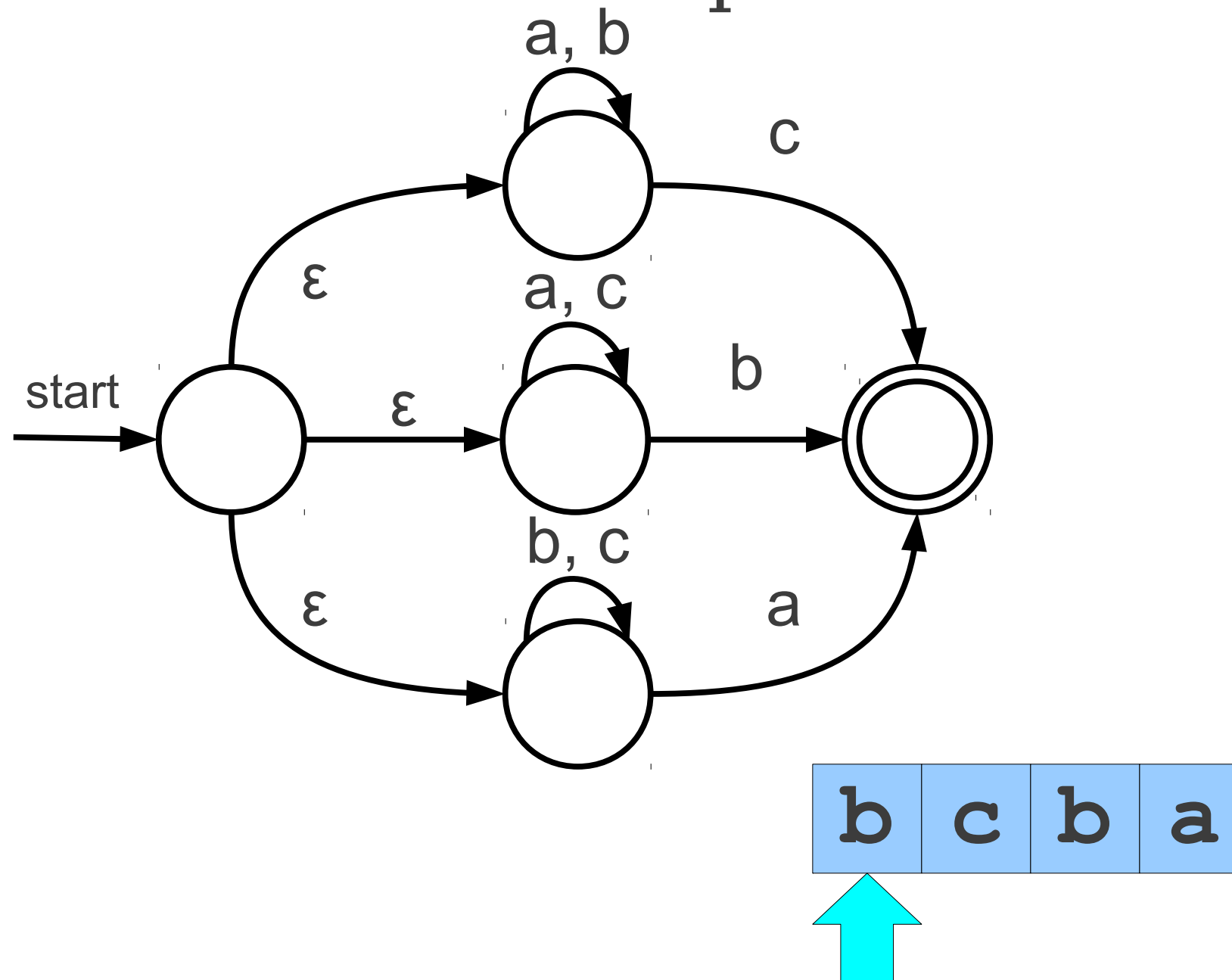


An Even More Complex Automaton

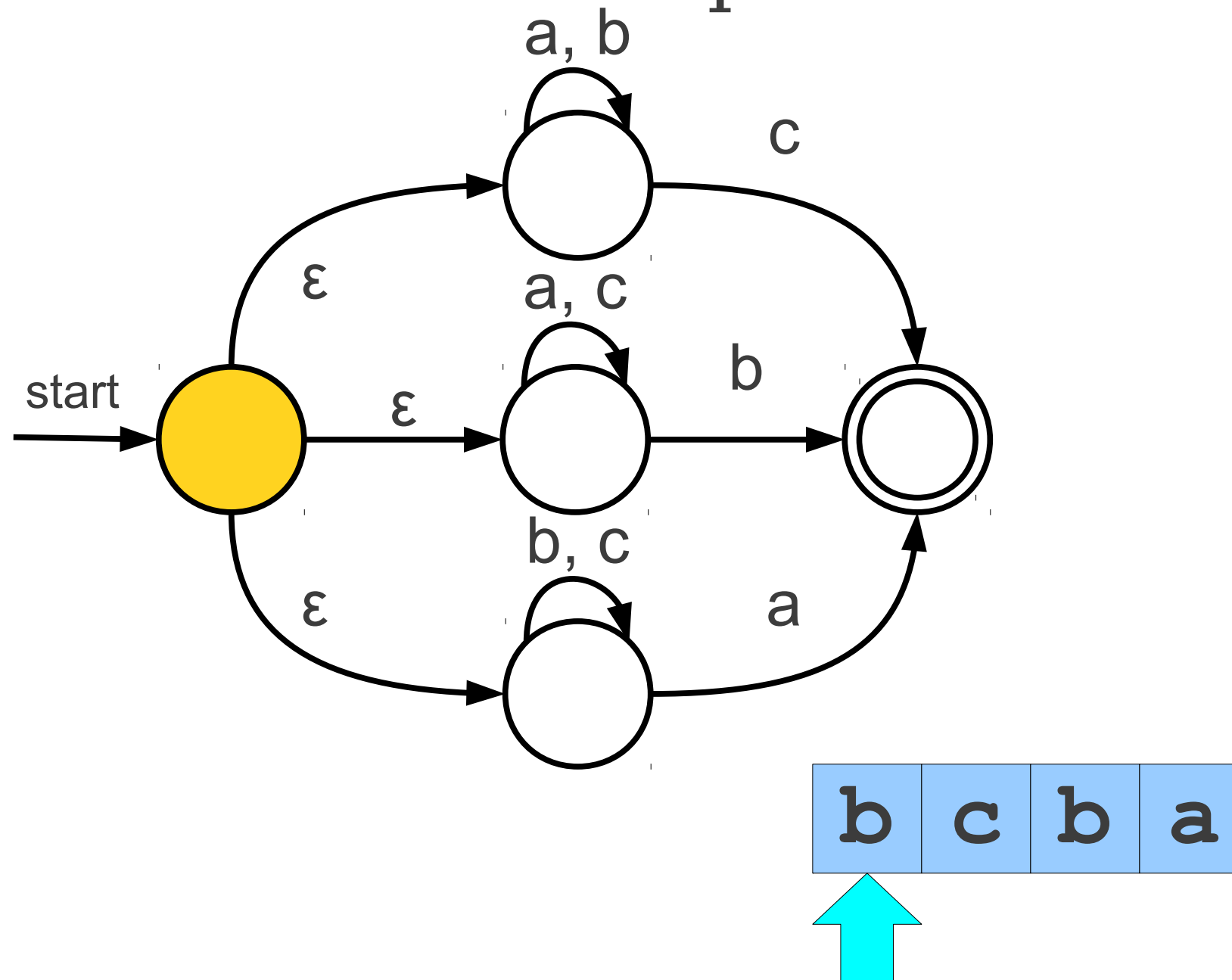


These are called ϵ -transitions. These transitions are followed automatically and without consuming any input.

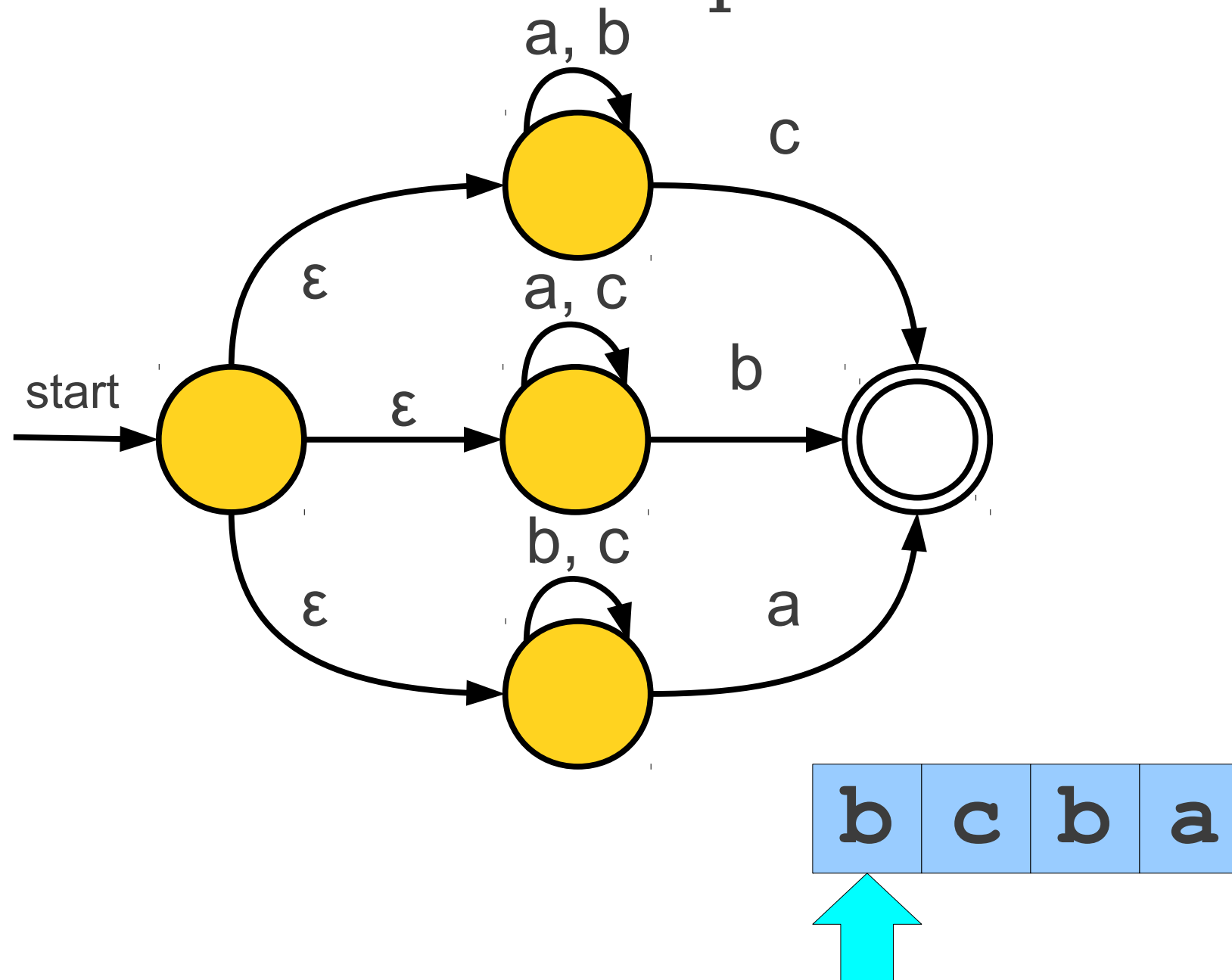
An Even More Complex Automaton



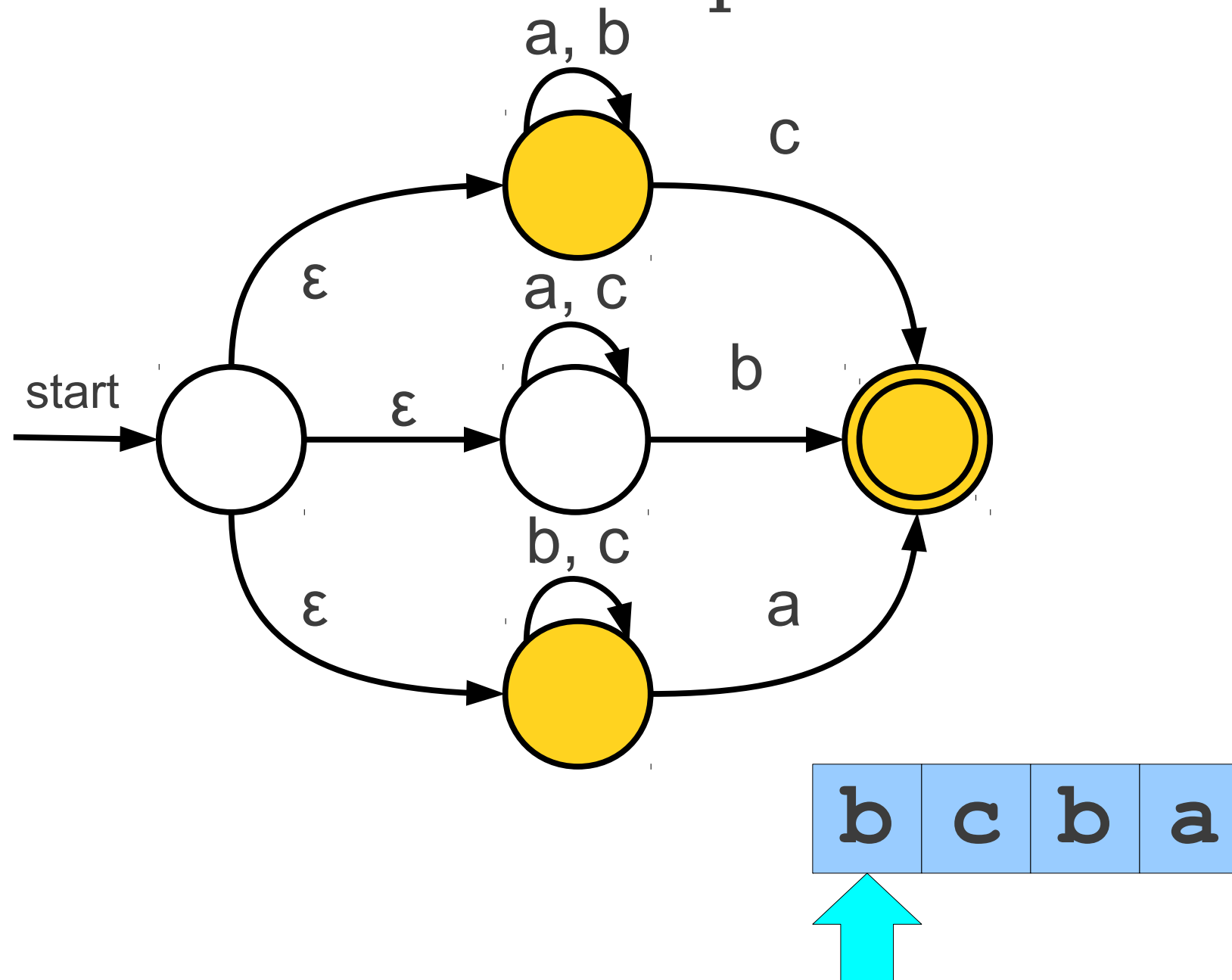
An Even More Complex Automaton



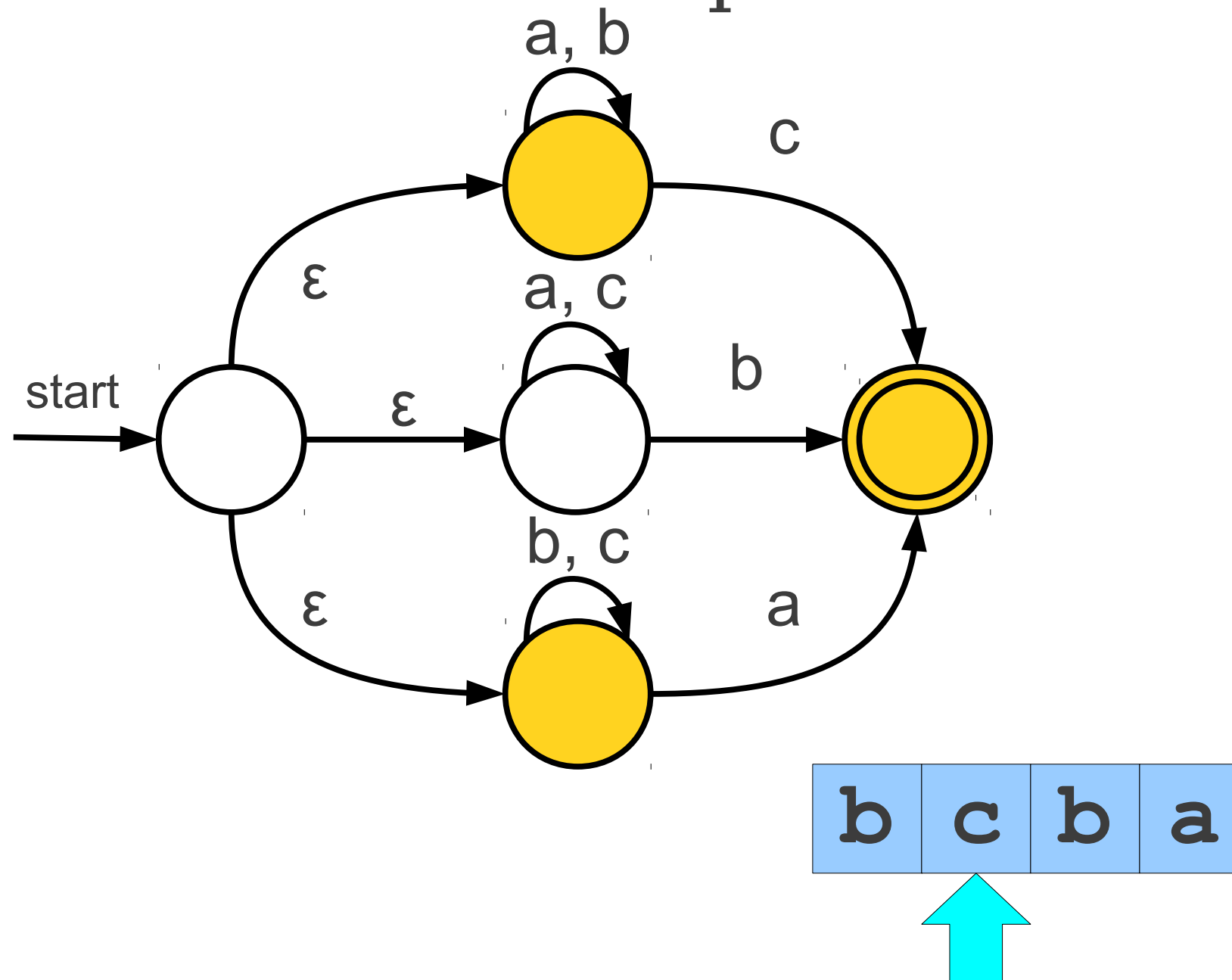
An Even More Complex Automaton



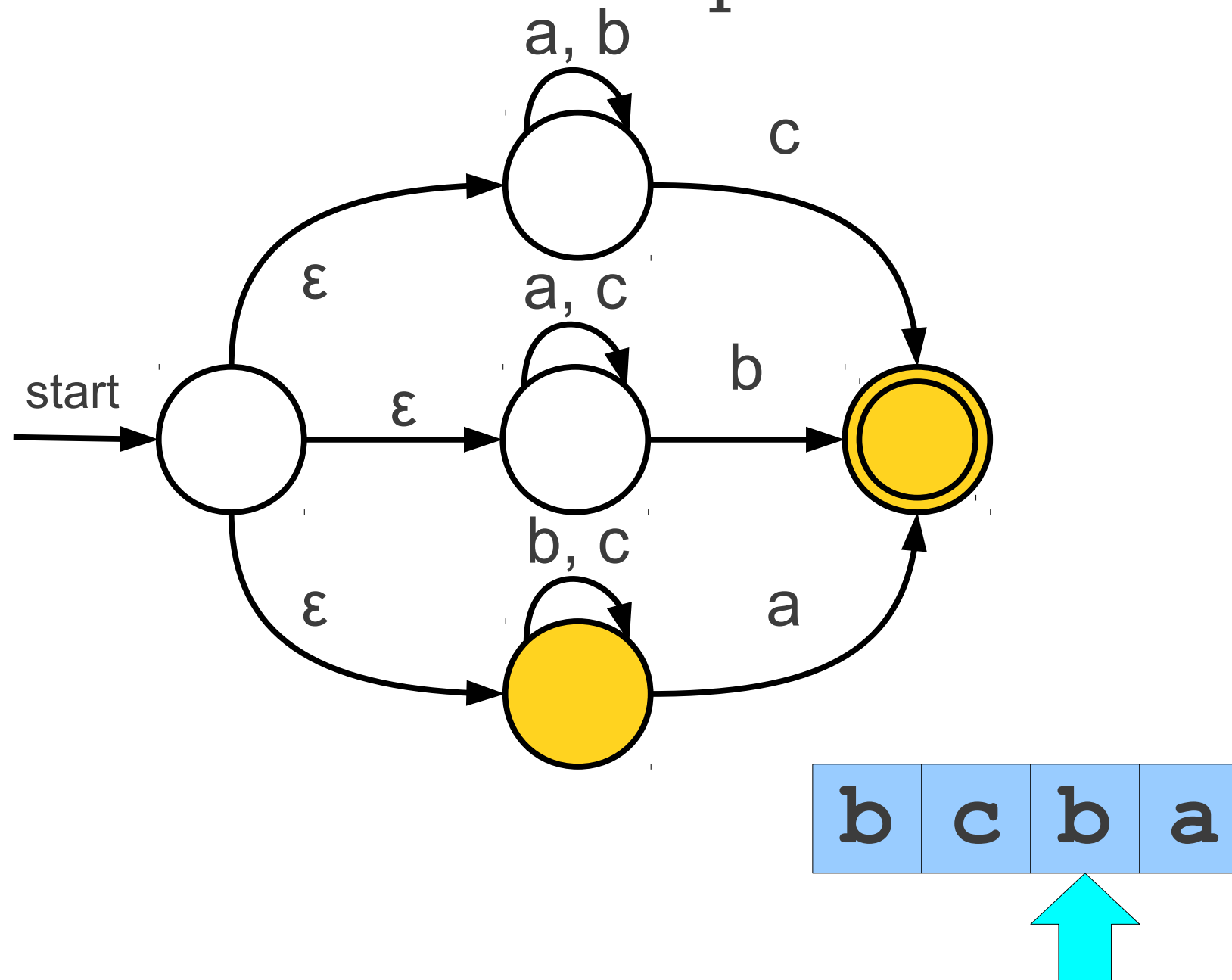
An Even More Complex Automaton



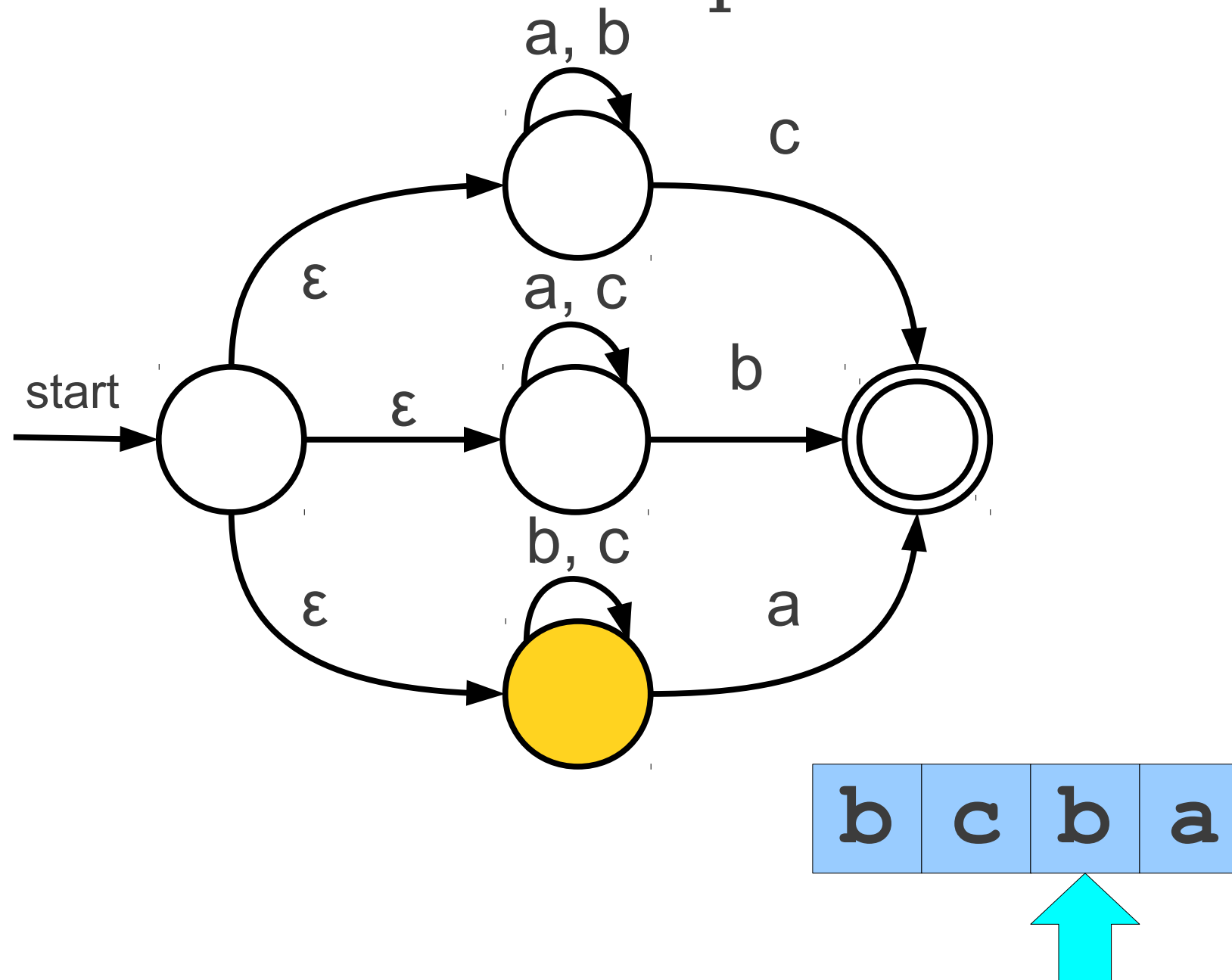
An Even More Complex Automaton



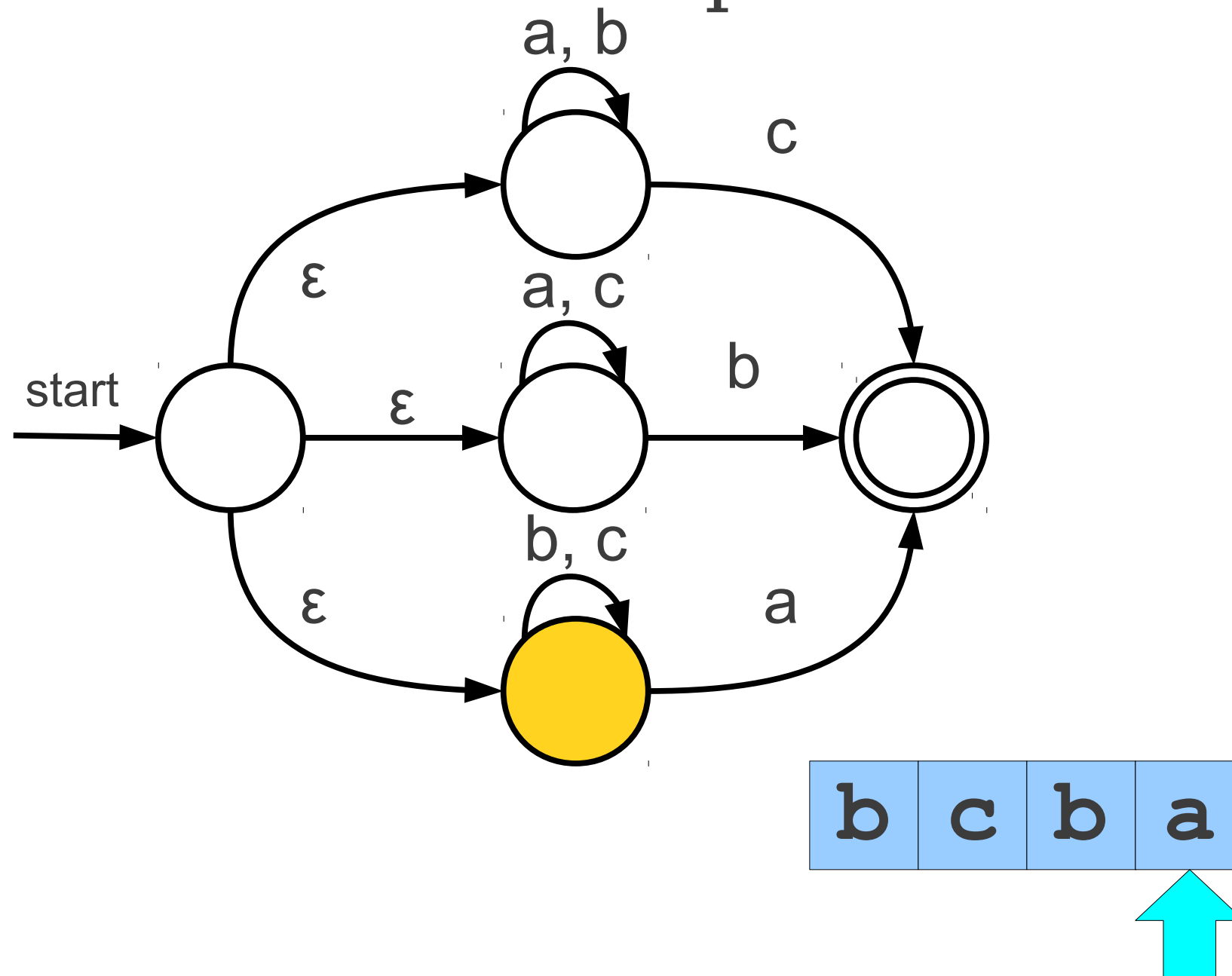
An Even More Complex Automaton



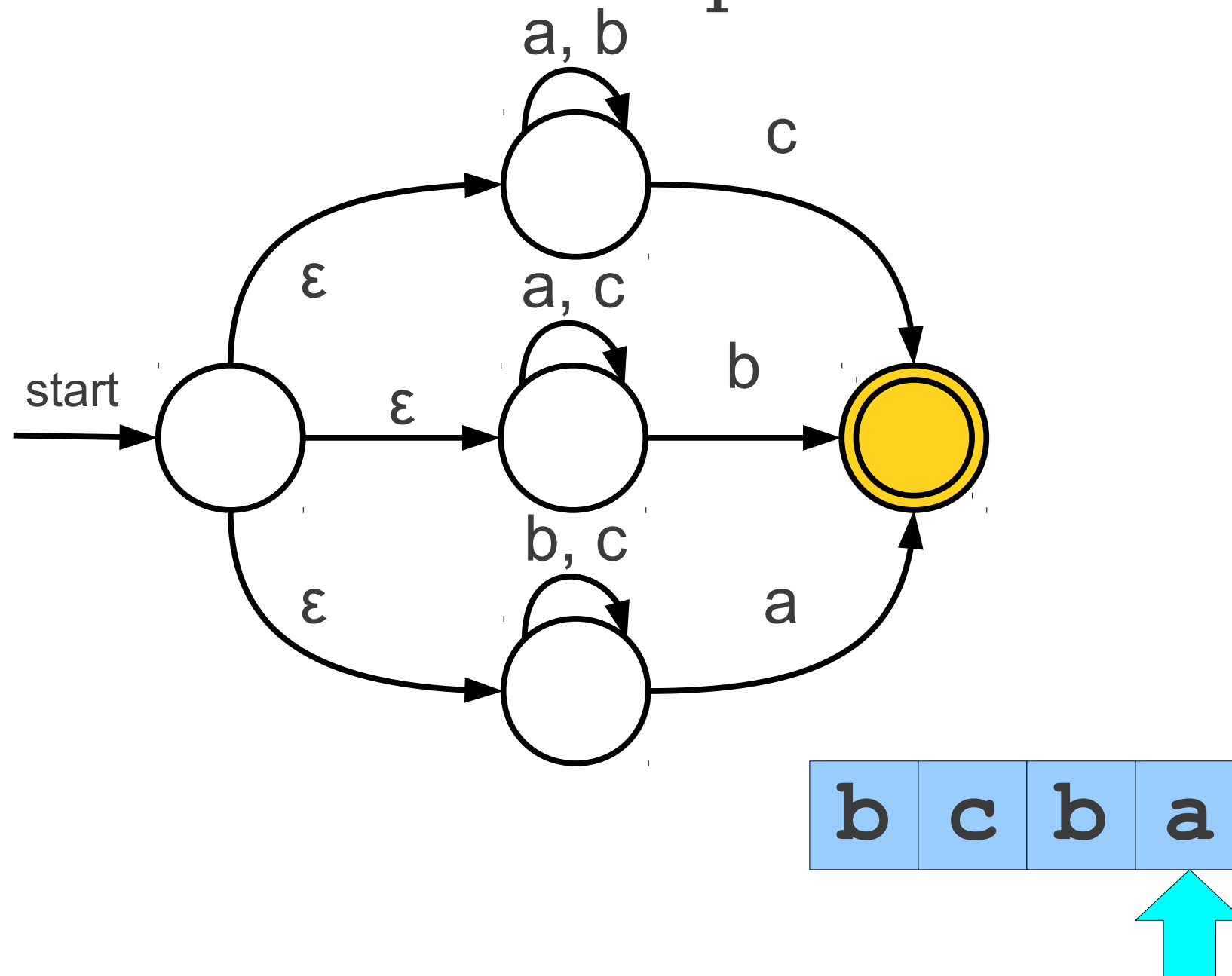
An Even More Complex Automaton



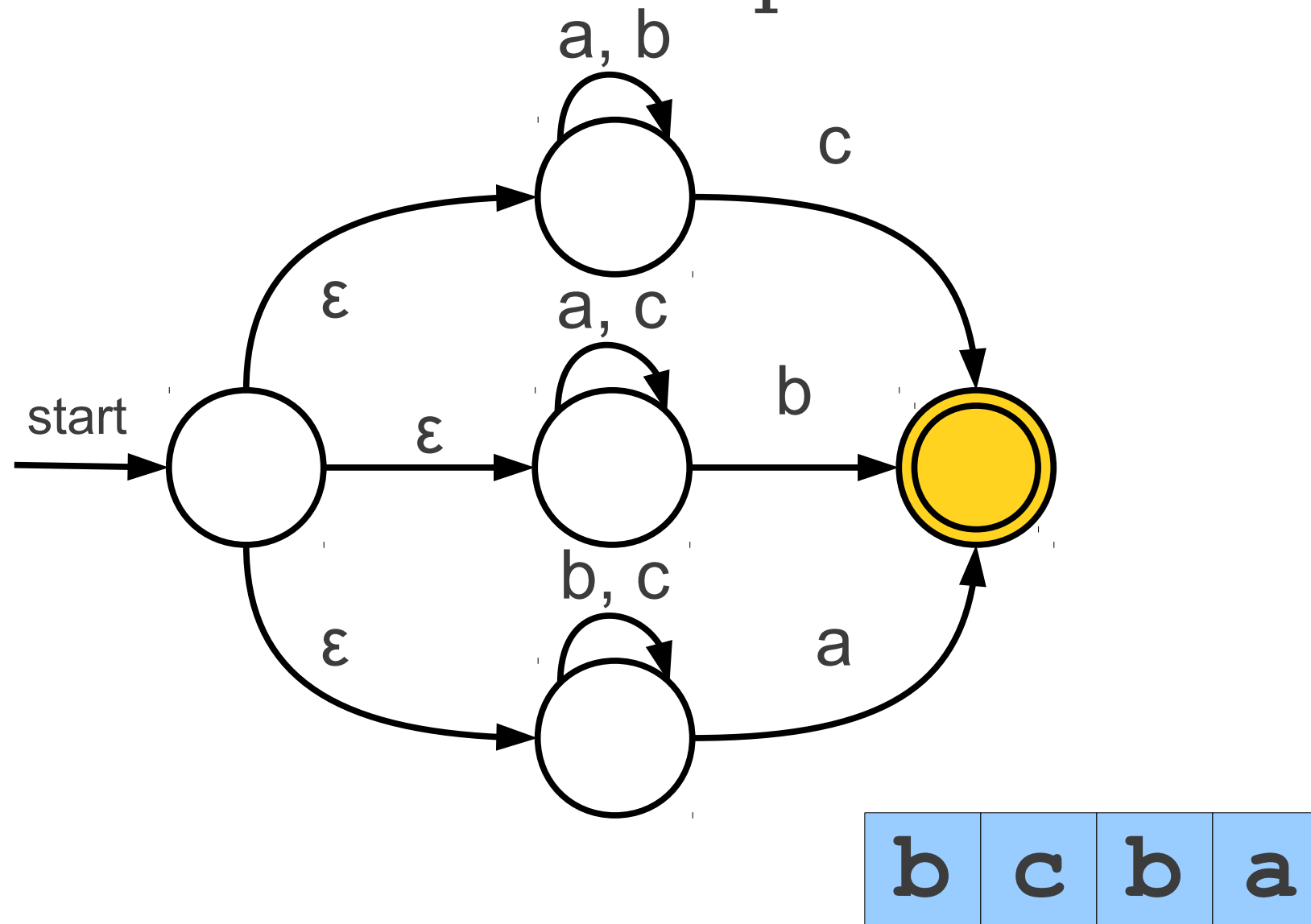
An Even More Complex Automaton



An Even More Complex Automaton



An Even More Complex Automaton



Lexer Generator

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

Challenges for Lexical Analyzer

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

Lexing Ambiguities

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

Lexing Ambiguities

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

f	o	r	t
---	---	---	---

Lexing Ambiguities

T_For

for

T_Identifier

[A-Za-z_][A-Za-z0-9_]*

f	o	r	t
---	---	---	---

f	o	r	t
f	o	r	t
f	o	r	t
f	o	r	t
f	o	r	t

f	o	r	t
f	o	r	t
f	o	r	t
f	o	r	t

Conflict Resolution

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

Lexing Ambiguities

T_For

for

T_Identifier

[A-Za-z_] [A-Za-z0-9_]*

f	o	r	t
---	---	---	---

f	o	r	t
---	---	---	---

Implementing Maximal Munch

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

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.

- Example

Implementing Maximal Munch

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

Implementing Maximal Munch

T_Do

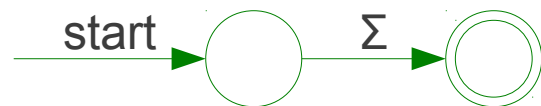
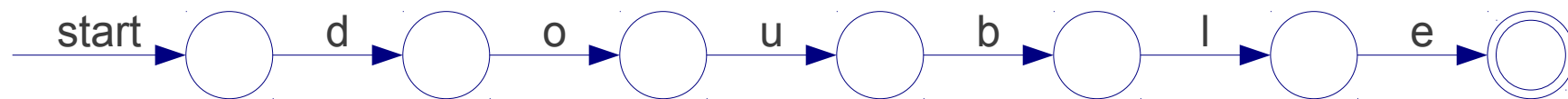
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

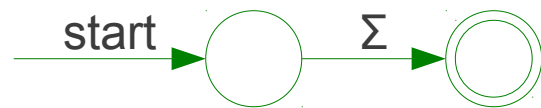
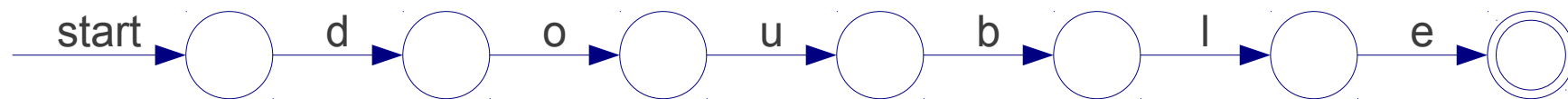
do

T_Double

double

T_Mystery

[A-Za-z]



D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---

Implementing Maximal Munch

T_Do

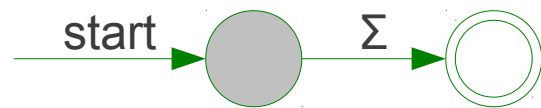
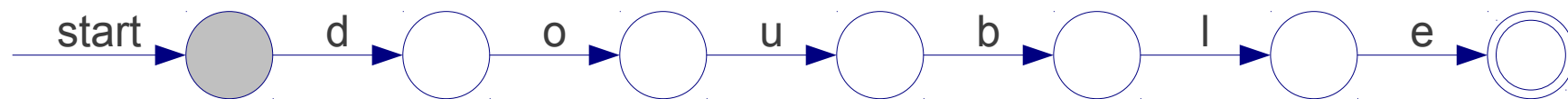
do

T_Double

double

T_Mystery

[A-Za-z]



D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---

Implementing Maximal Munch

T_Do

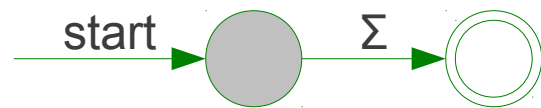
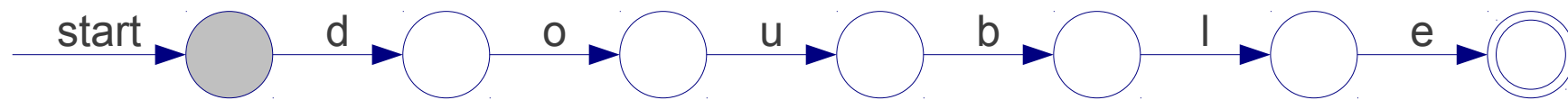
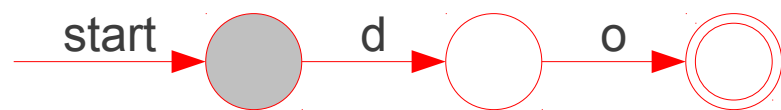
do

T_Double

double

T_Mystery

[A-Za-z]



D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---



Implementing Maximal Munch

T_Do

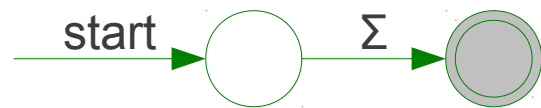
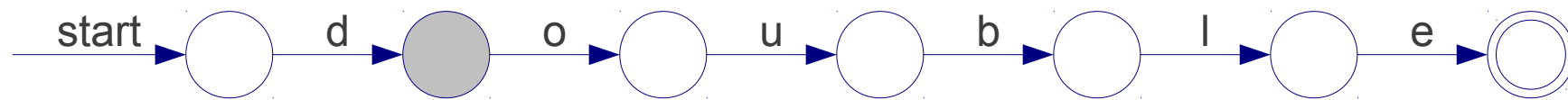
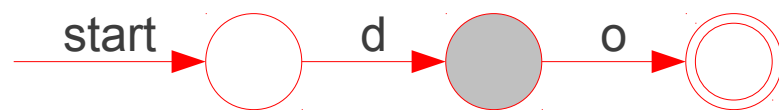
do

T_Double

double

T_Mystery

[A-Za-z]



D	O	U	B	D	O	U	B	L	E
---	---	---	---	---	---	---	---	---	---



Implementing Maximal Munch

T_Do

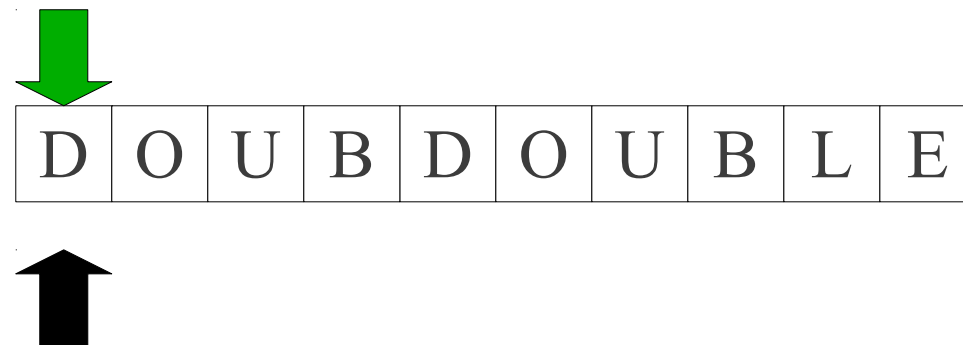
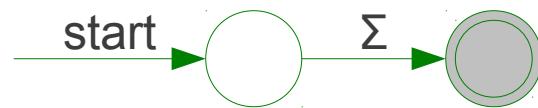
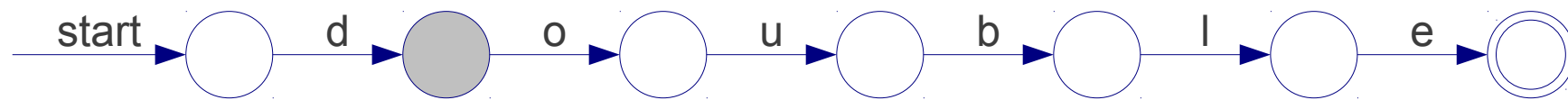
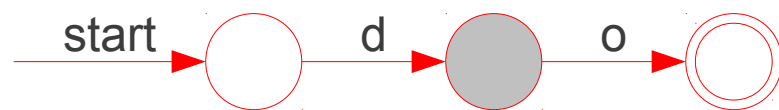
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

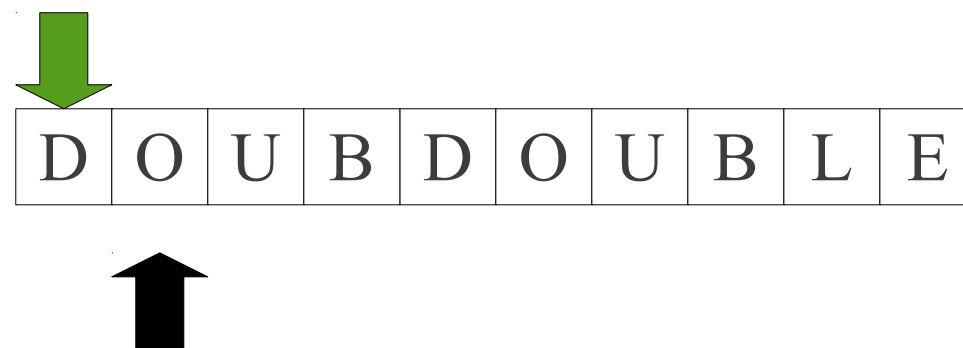
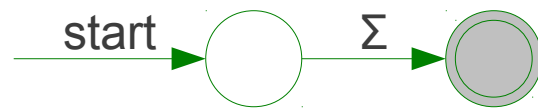
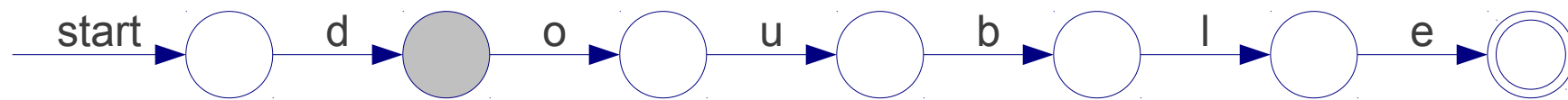
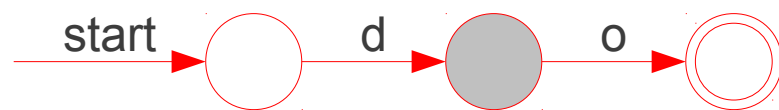
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

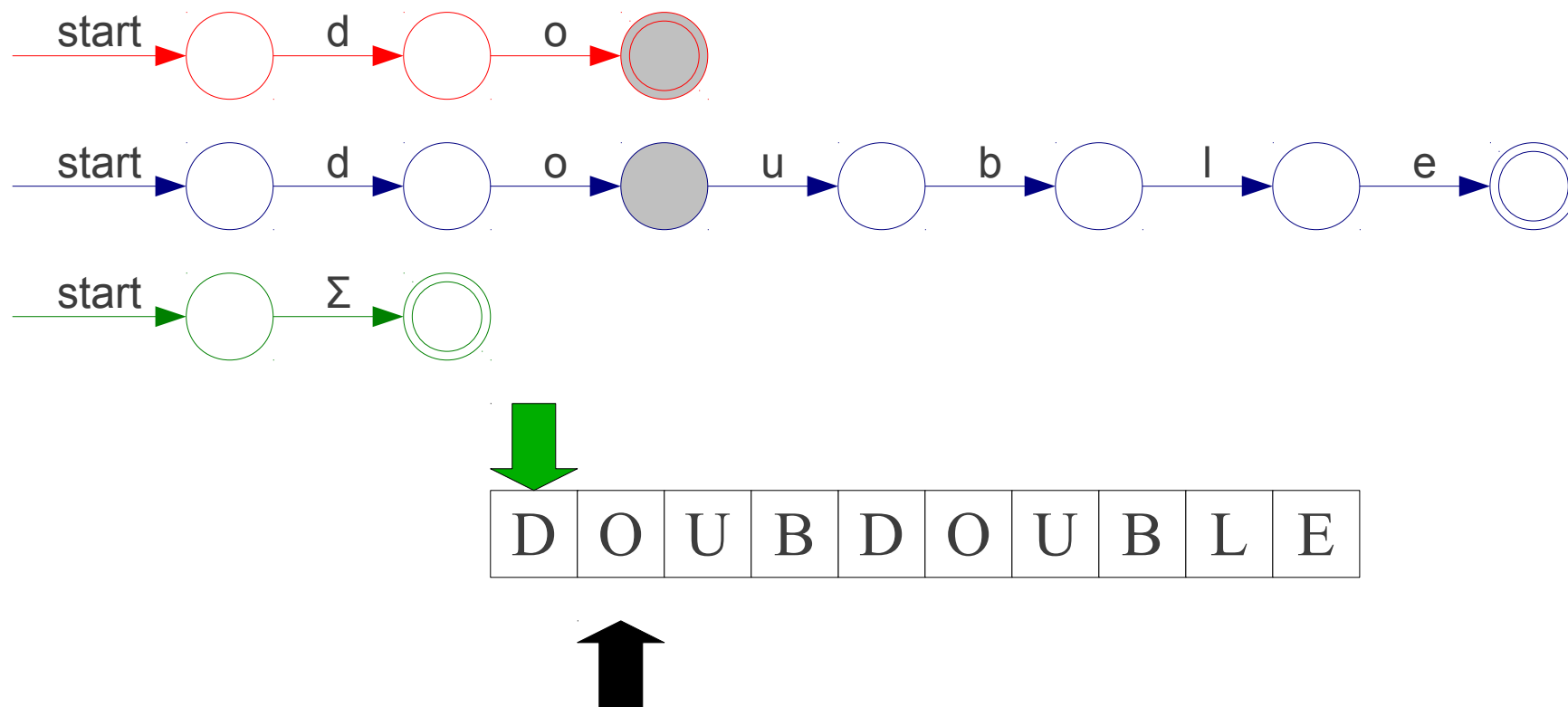
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

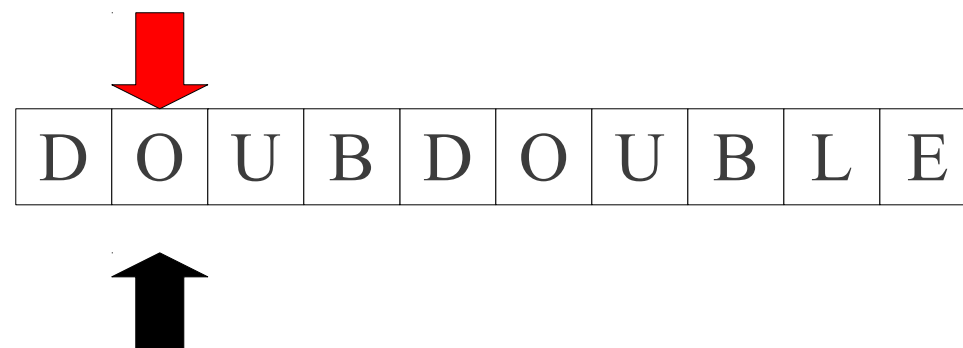
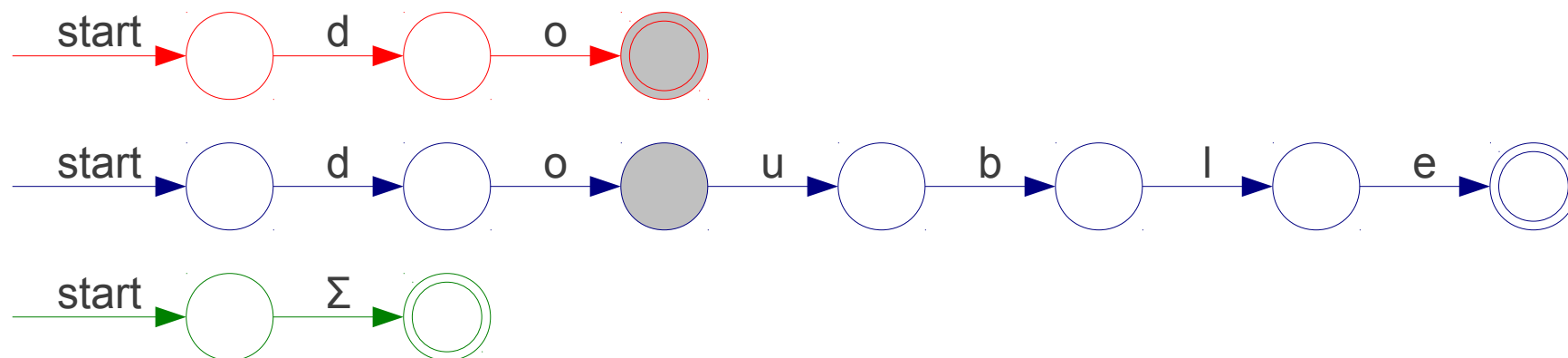
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

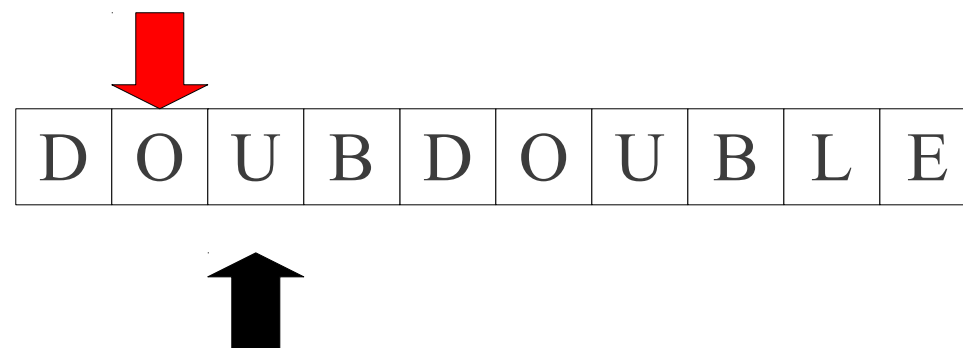
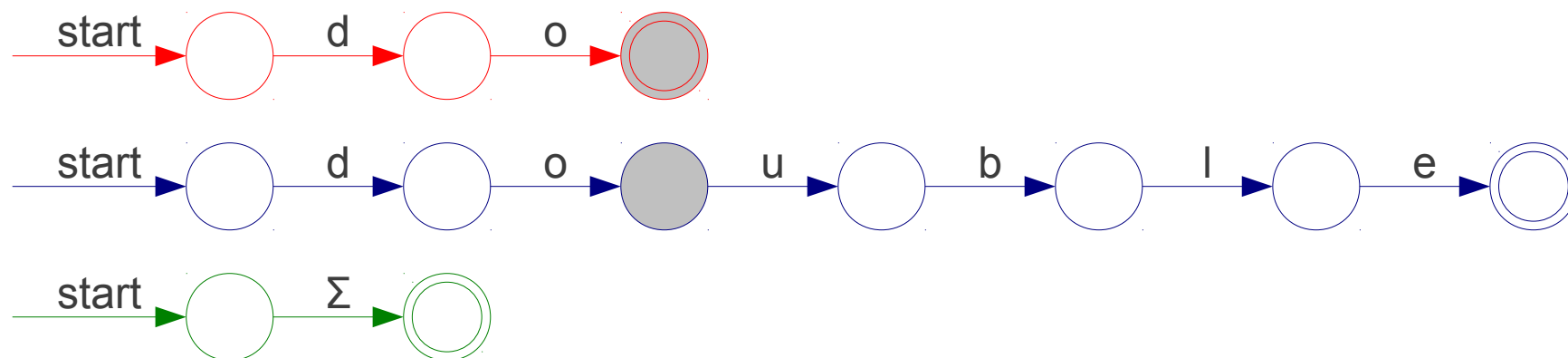
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

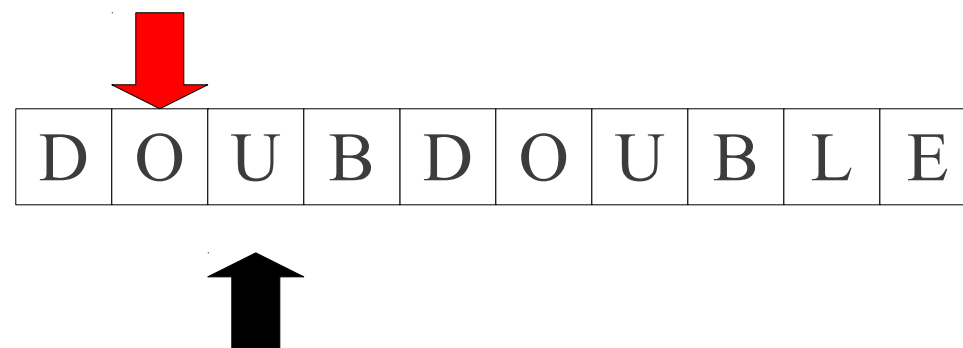
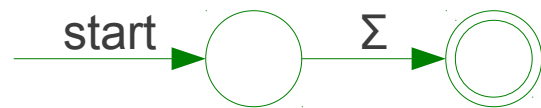
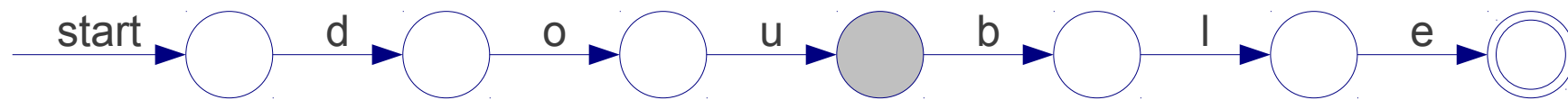
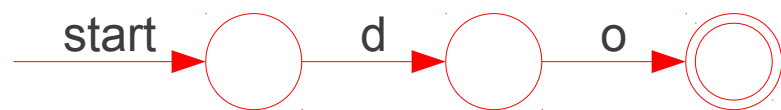
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

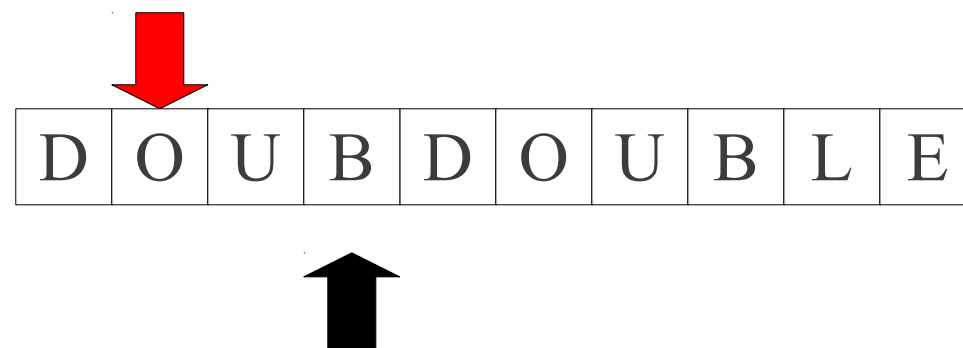
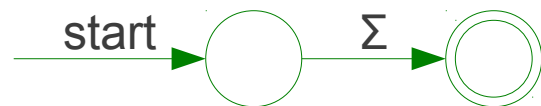
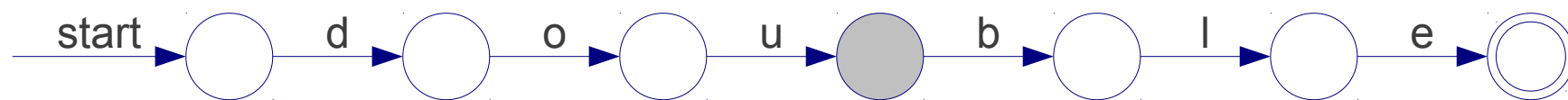
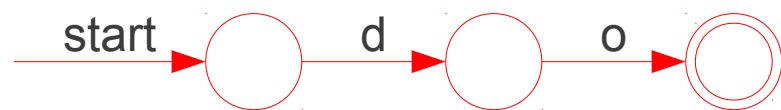
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

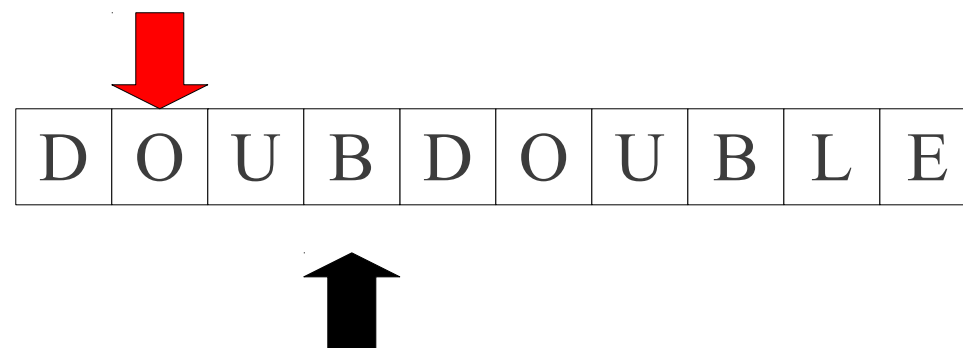
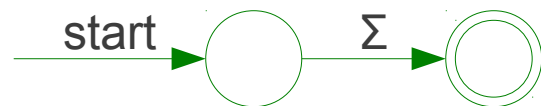
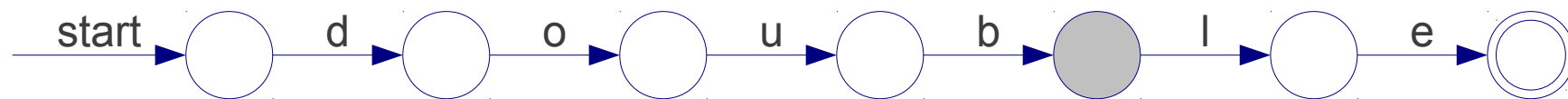
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

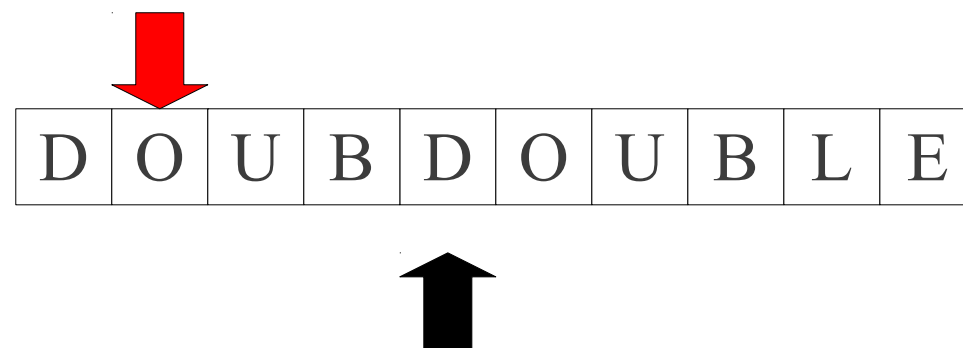
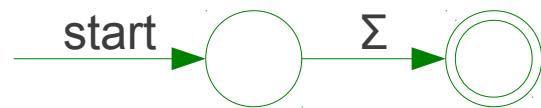
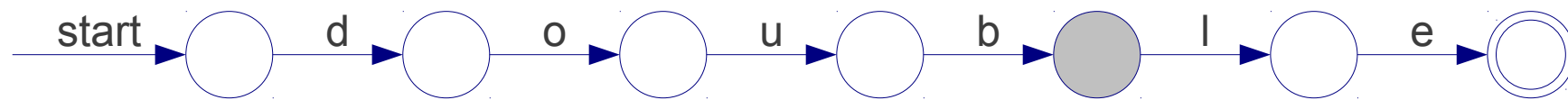
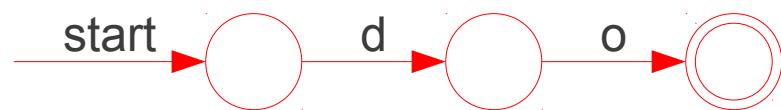
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

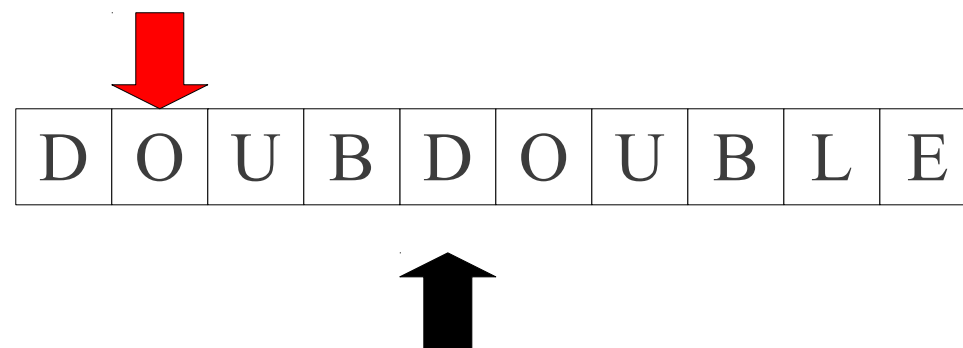
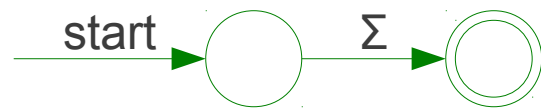
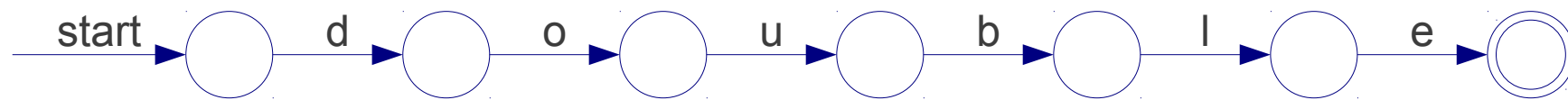
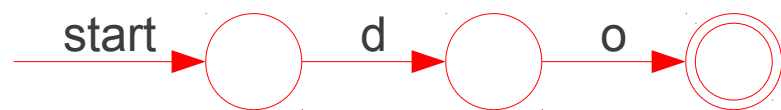
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

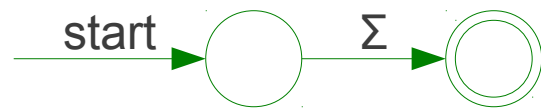
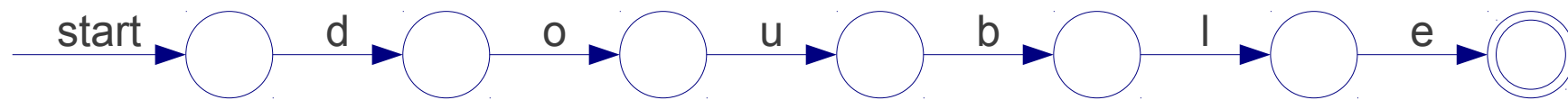
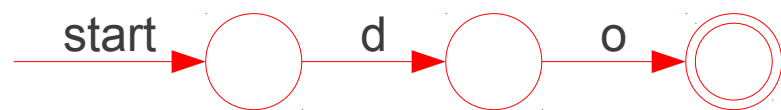
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U B D O U B L E



Implementing Maximal Munch

T_Do

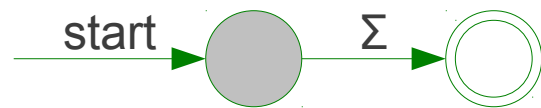
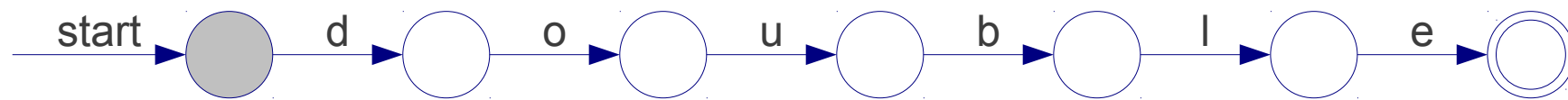
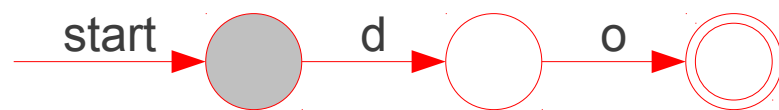
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U B D O U B L E



Implementing Maximal Munch

T_Do

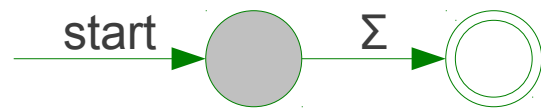
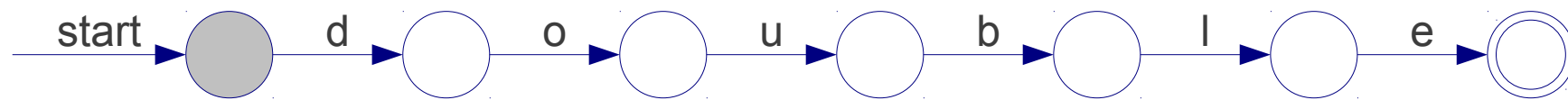
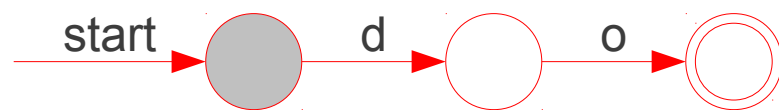
do

T_Double

double

T_Mystery

[A-Za-z]



DO

U B D O U B L E



Implementing Maximal Munch

T_Do

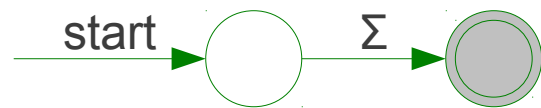
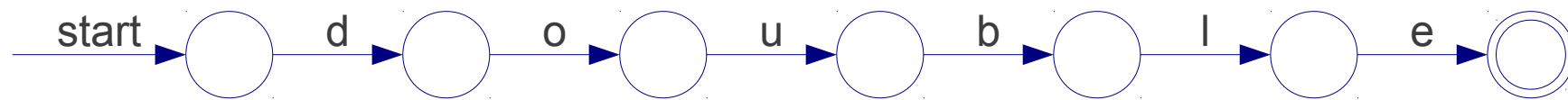
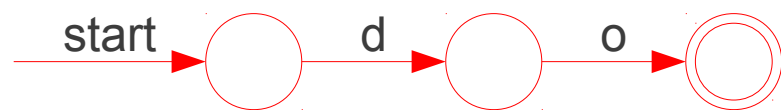
do

T_Double

double

T_Mystery

[A-Za-z]



DO

U B D O U B L E



Implementing Maximal Munch

T_Do

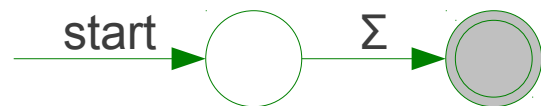
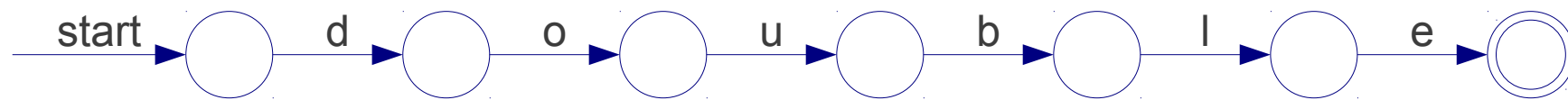
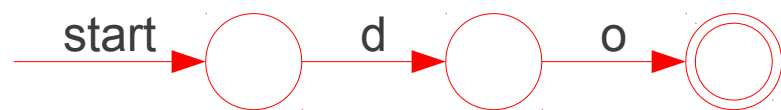
do

T_Double

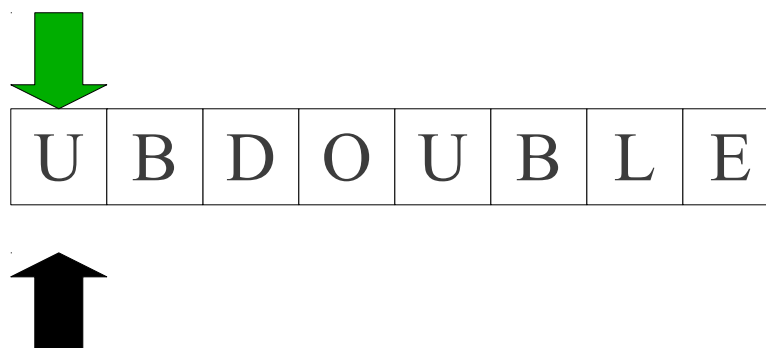
double

T_Mystery

[A-Za-z]



D O



Implementing Maximal Munch

T_Do

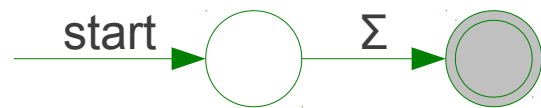
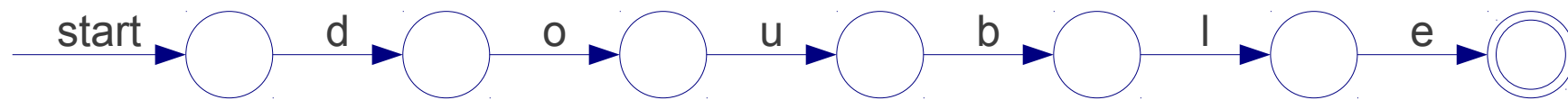
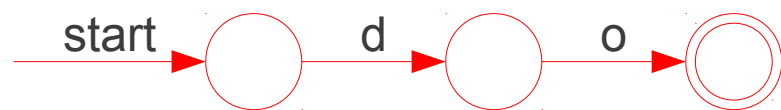
do

T_Double

double

T_Mystery

[A-Za-z]



DO

↓
U B D O U B L E
↑

Implementing Maximal Munch

T_Do

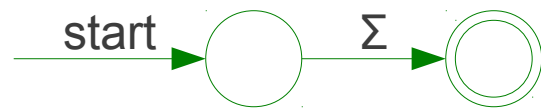
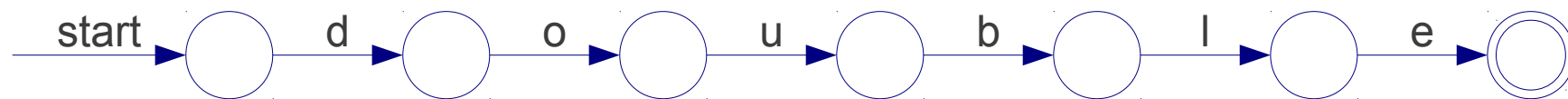
do

T_Double

double

T_Mystery

[A-Za-z]



DO

↓
U B D O U B L E
↑

Implementing Maximal Munch

T_Do

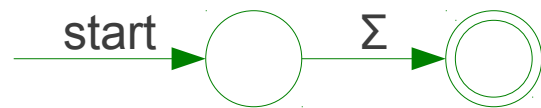
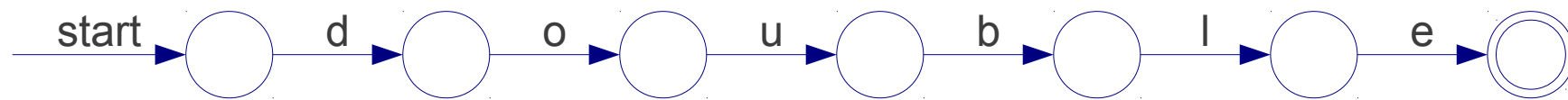
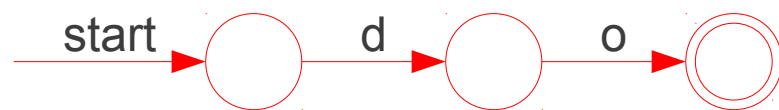
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B D O U B L E



Implementing Maximal Munch

T_Do

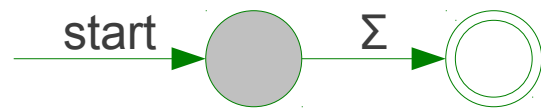
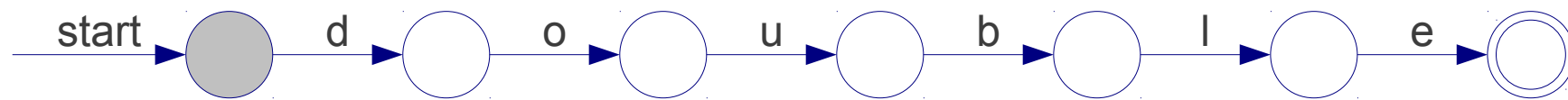
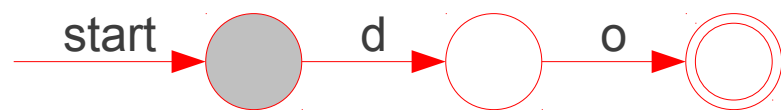
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B D O U B L E



Implementing Maximal Munch

T_Do

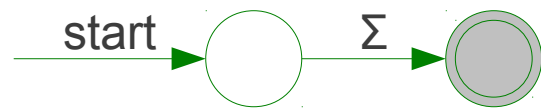
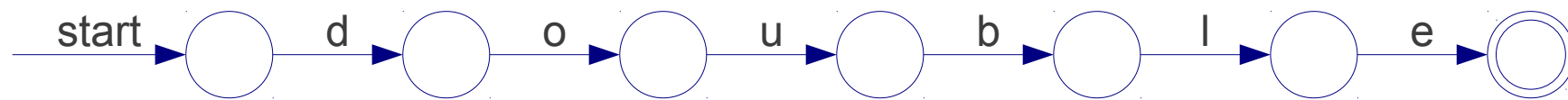
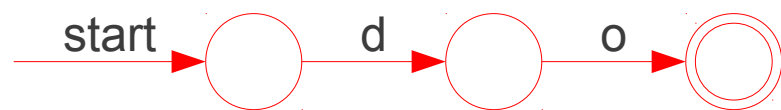
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B D O U B L E



Implementing Maximal Munch

T_Do

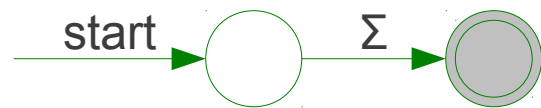
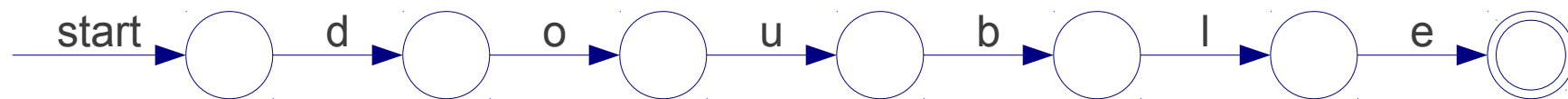
do

T_Double

double

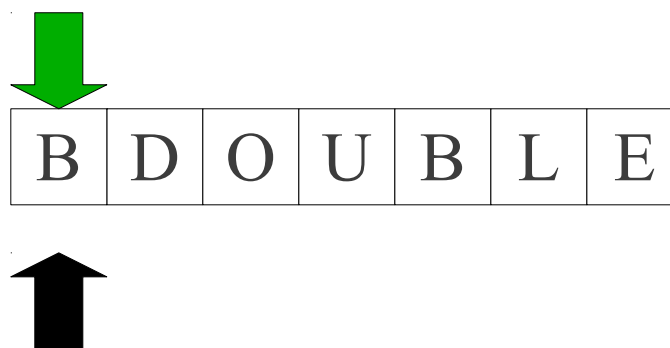
T_Mystery

[A-Za-z]



D O

U



Implementing Maximal Munch

T_Do

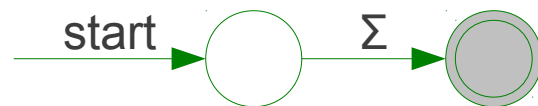
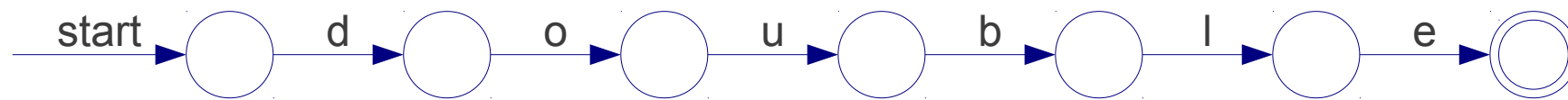
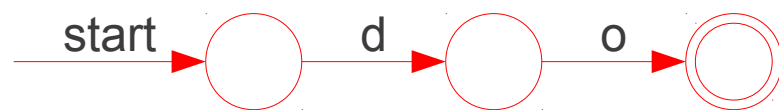
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U



Implementing Maximal Munch

T_Do

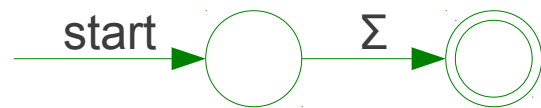
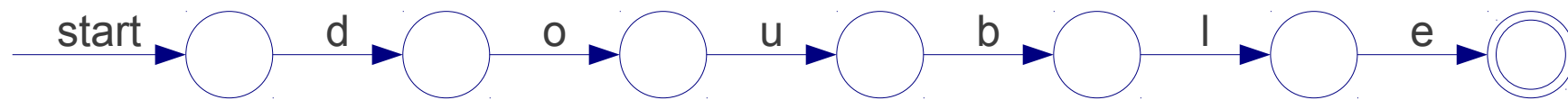
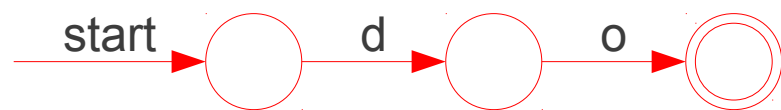
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B D O U B L E



Implementing Maximal Munch

T_Do

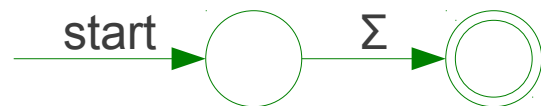
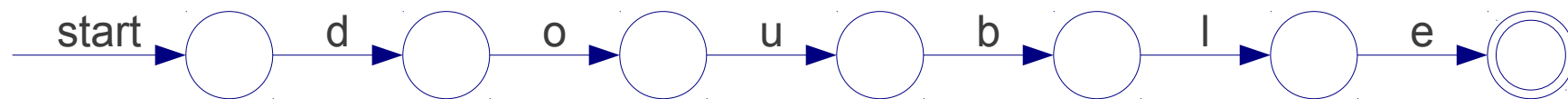
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B

D O U B L E



Implementing Maximal Munch

T_Do

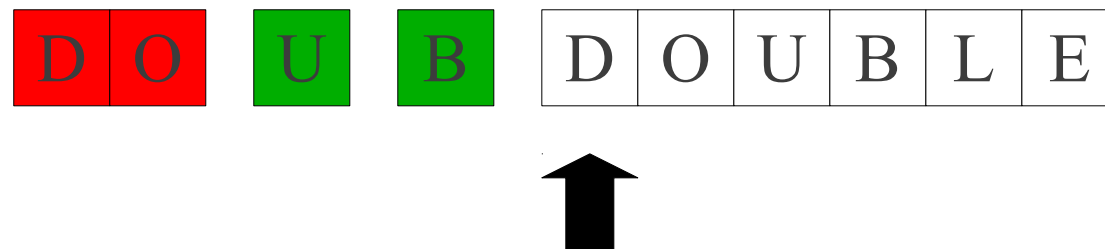
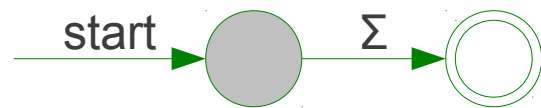
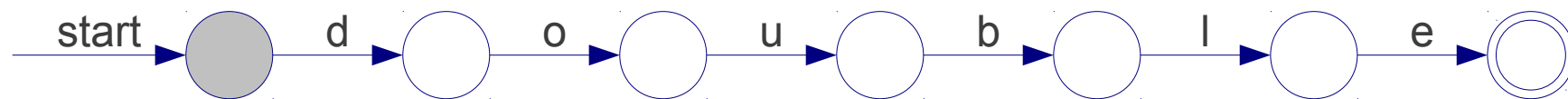
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

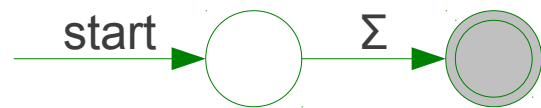
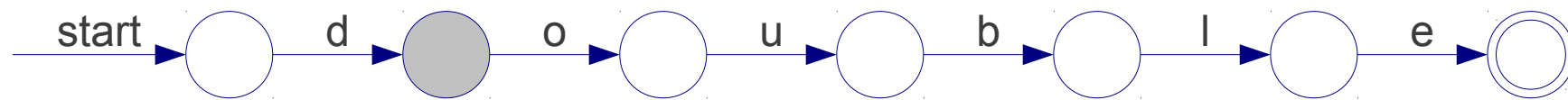
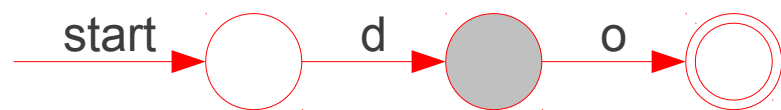
do

T_Double

double

T_Mystery

[A-Za-z]



D O **U B** D O U B L E



Implementing Maximal Munch

T_Do

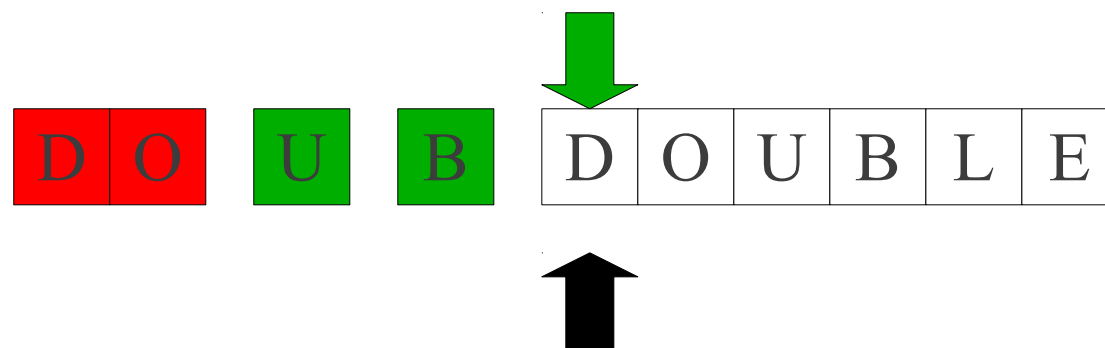
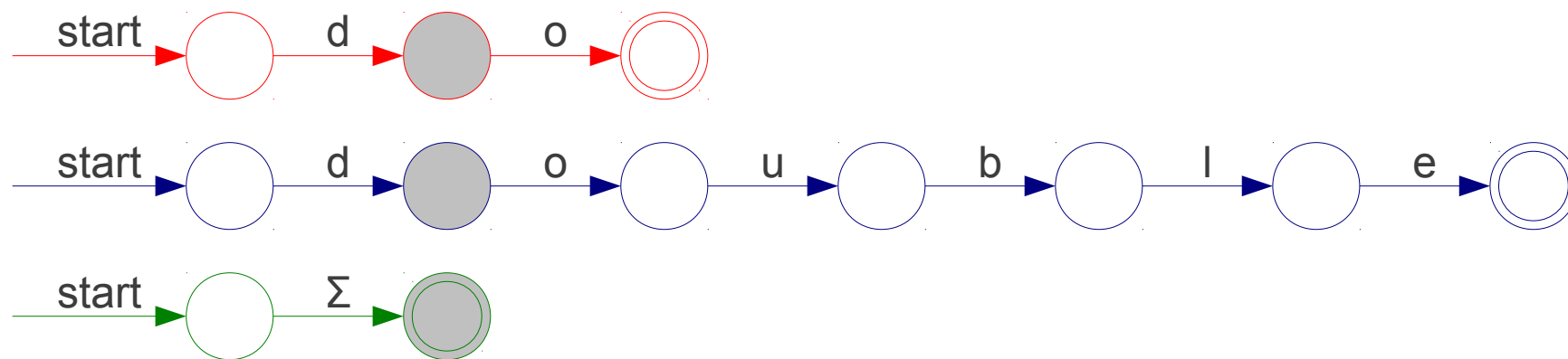
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

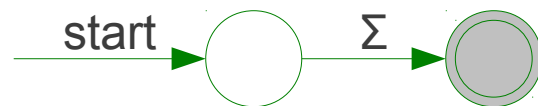
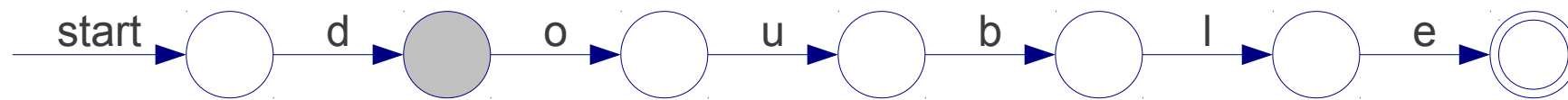
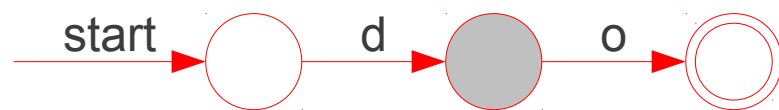
do

T_Double

double

T_Mystery

[A-Za-z]



D O

U

B

D O U B L E



Implementing Maximal Munch

T_Do

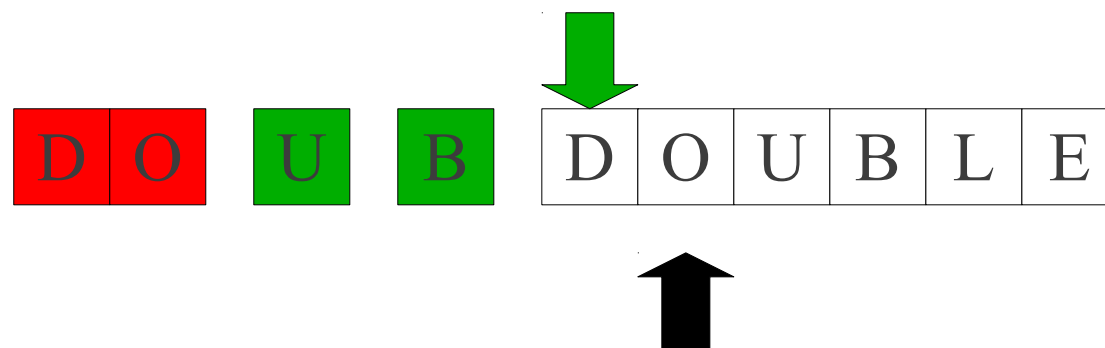
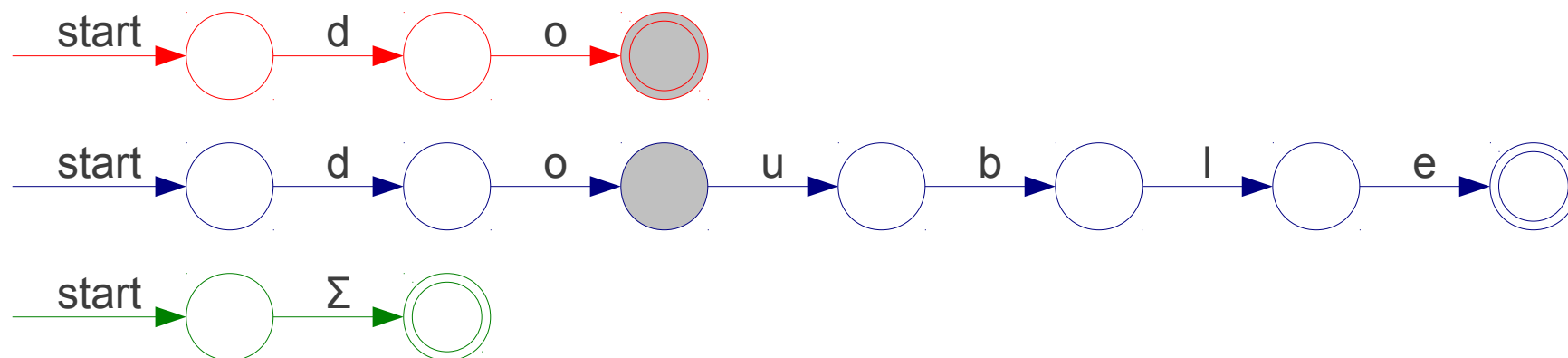
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

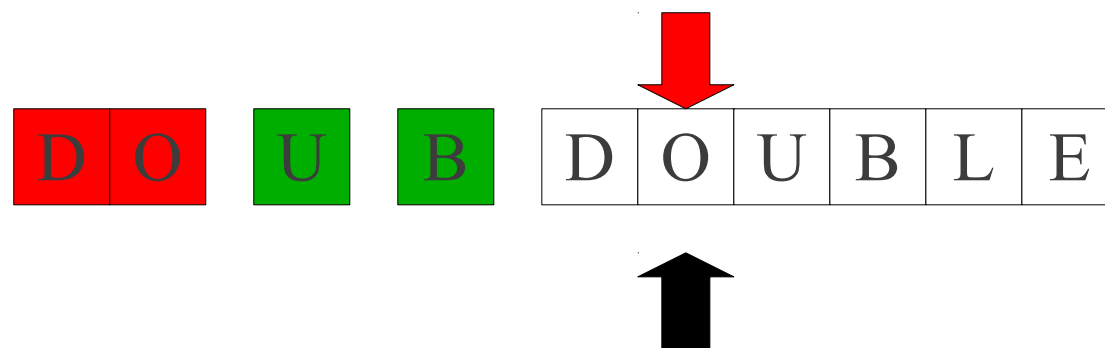
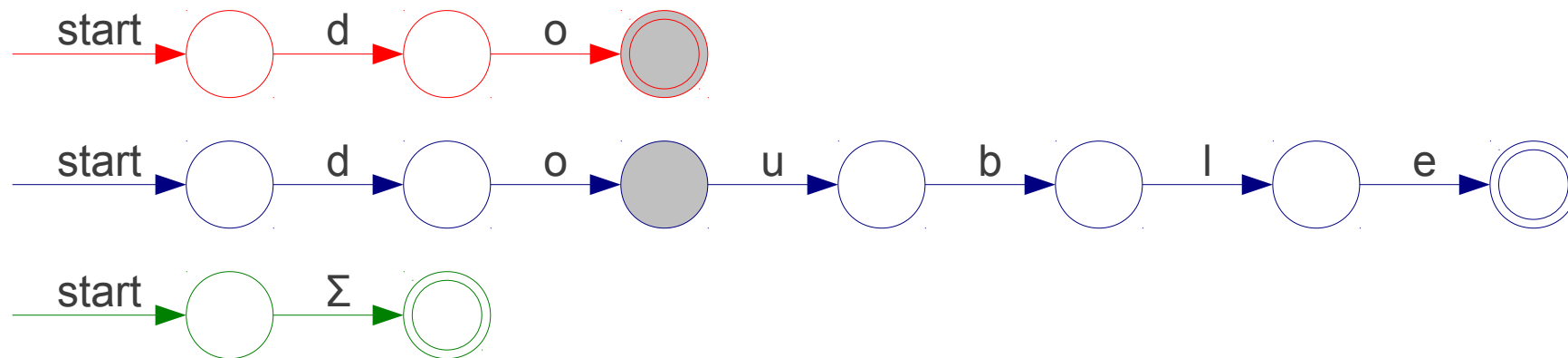
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

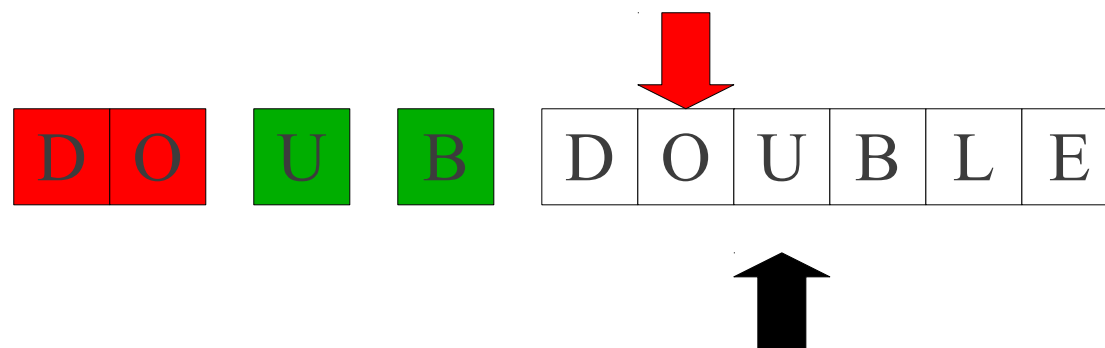
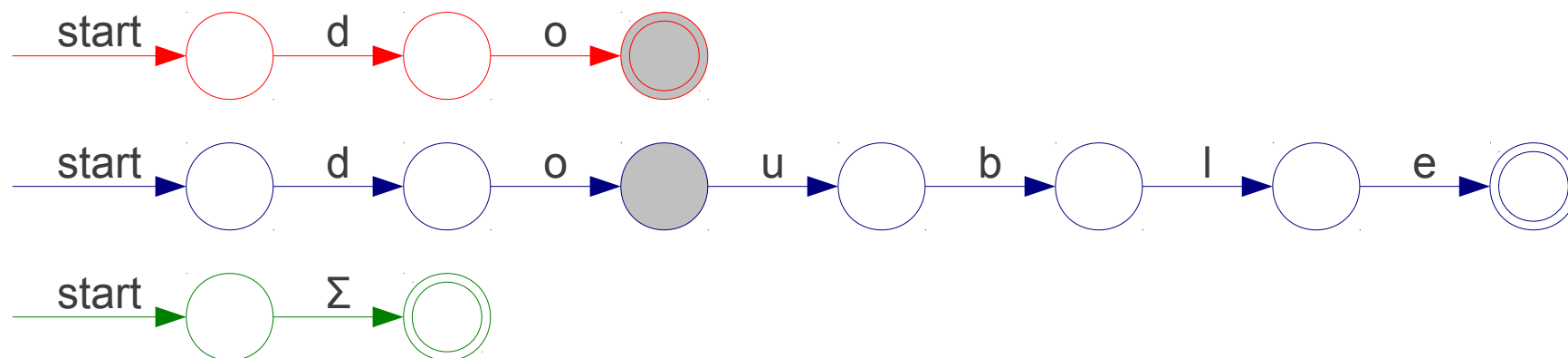
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

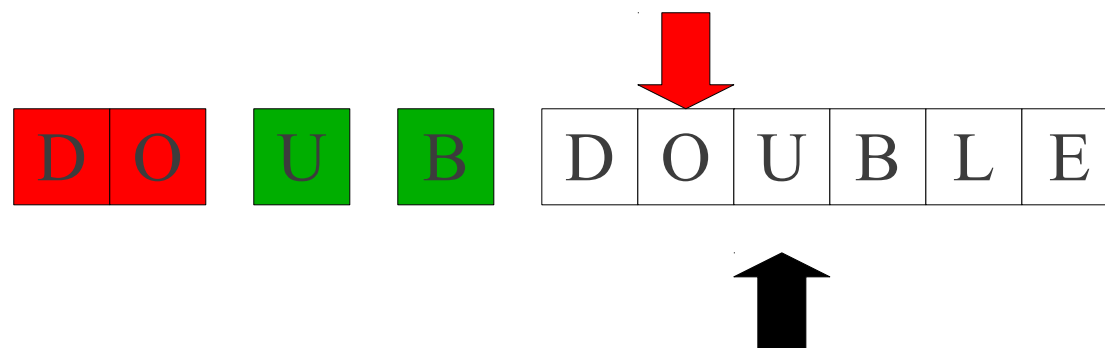
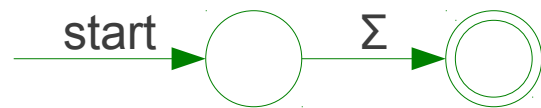
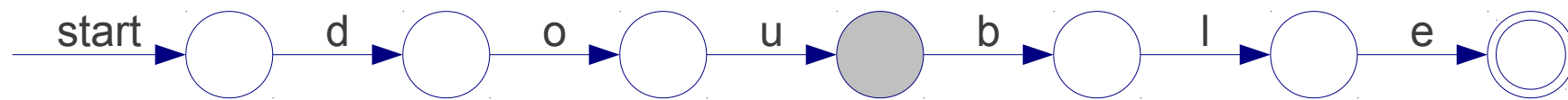
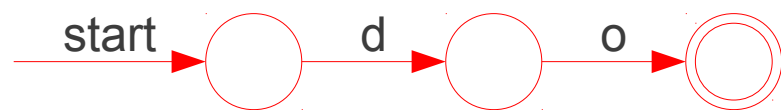
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

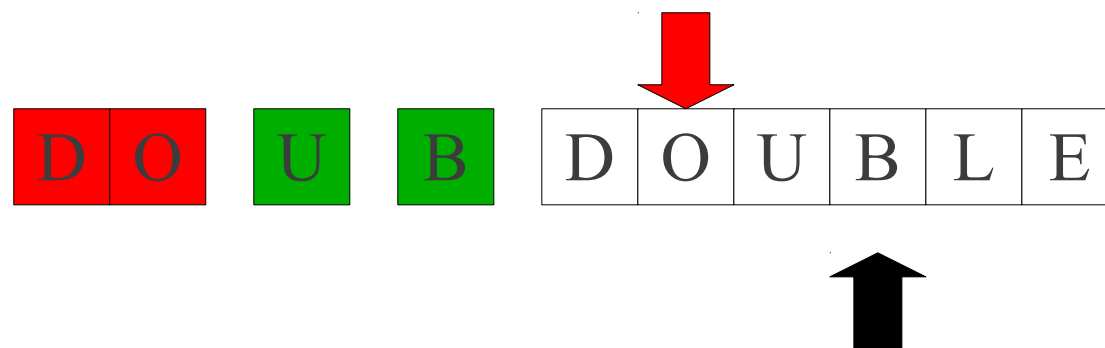
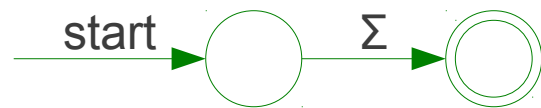
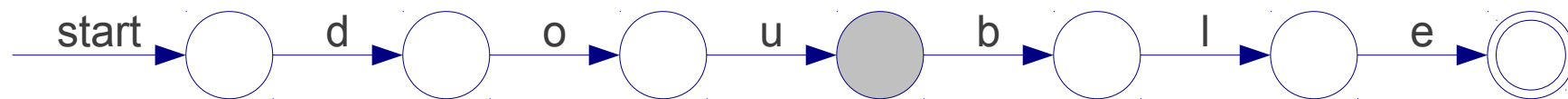
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

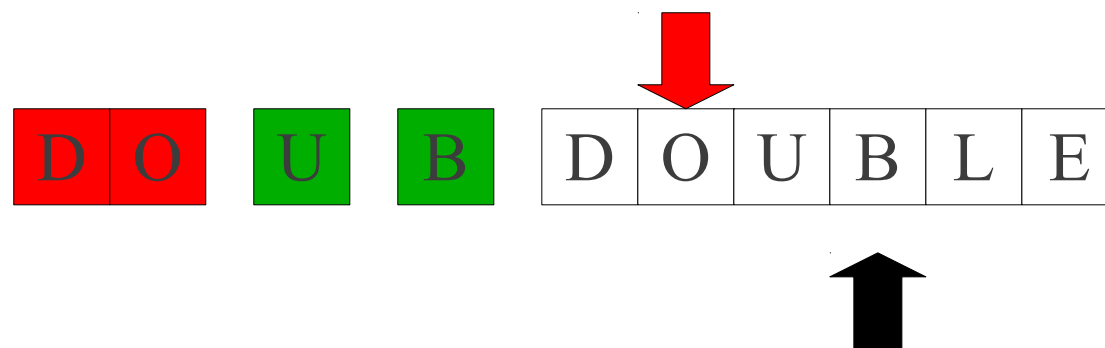
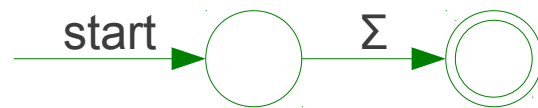
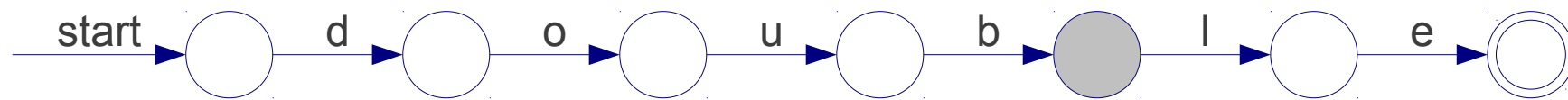
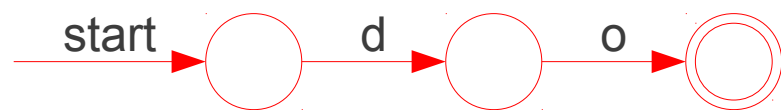
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

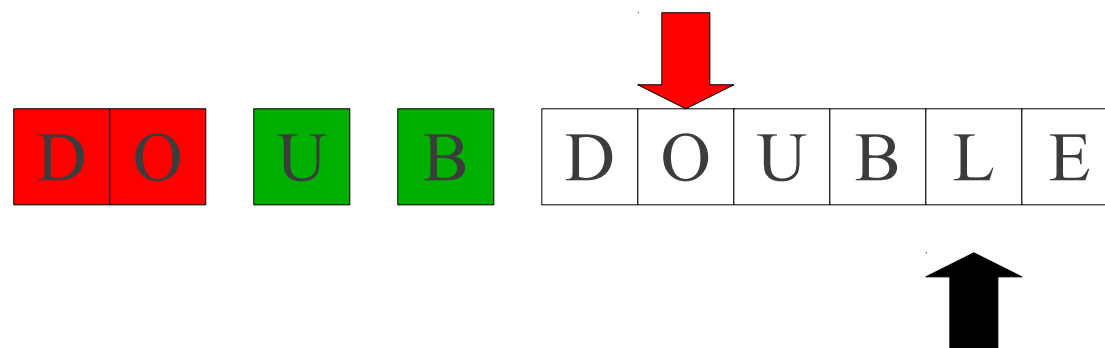
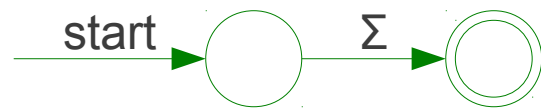
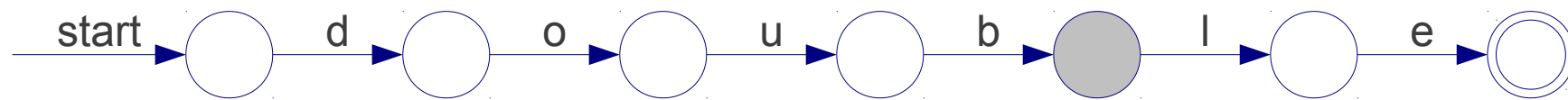
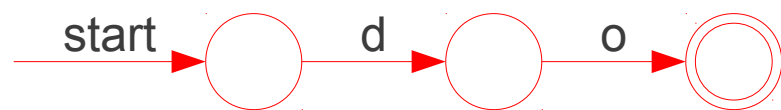
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

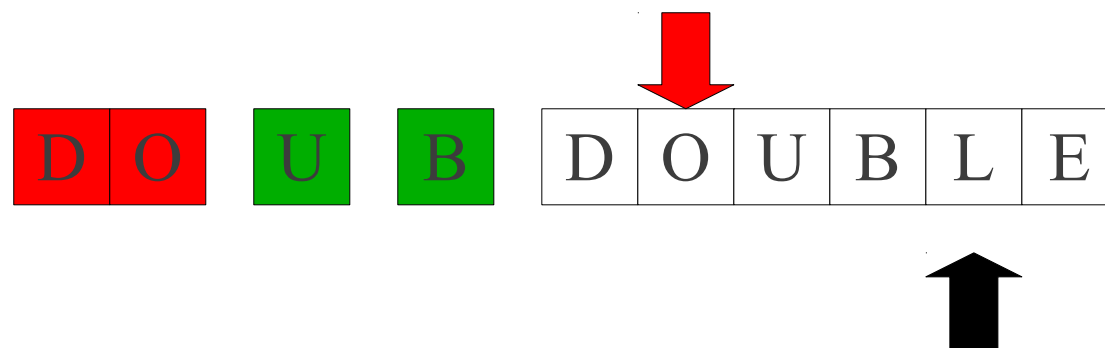
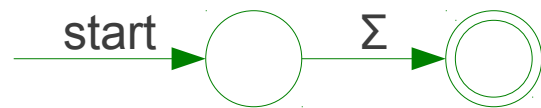
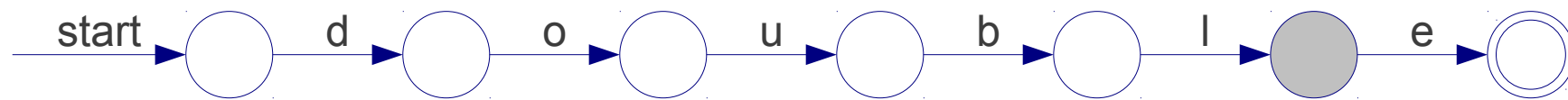
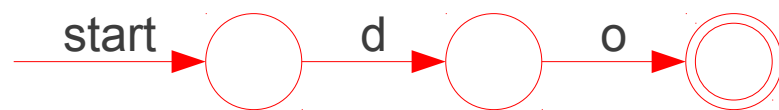
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

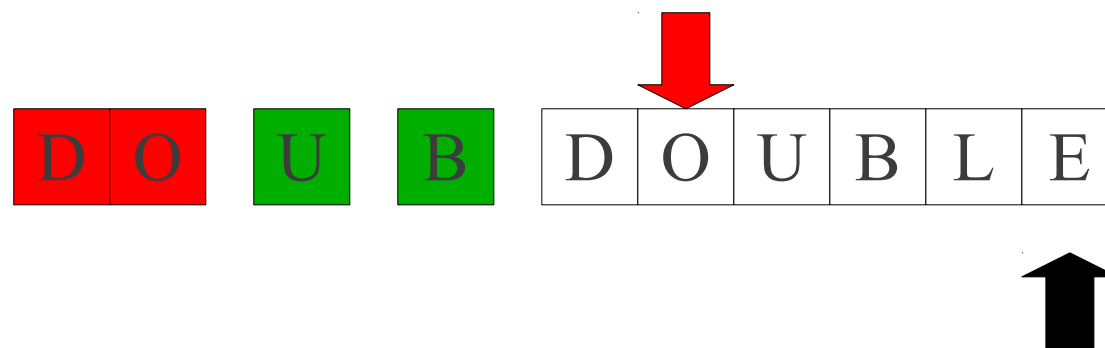
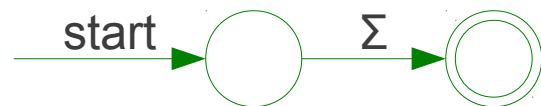
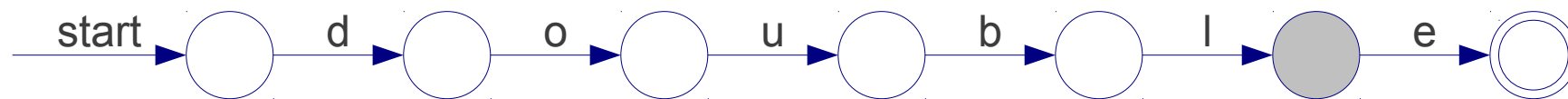
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

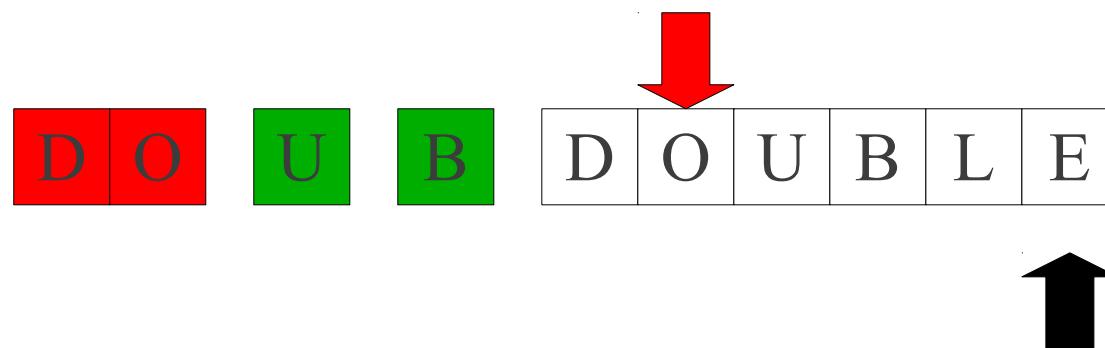
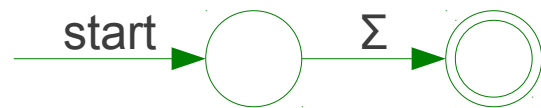
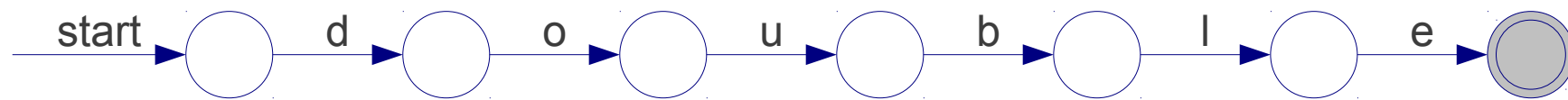
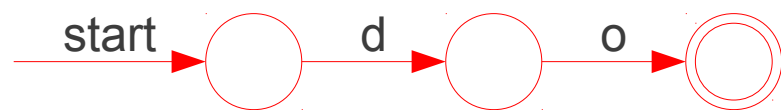
do

T_Double

double

T_Mystery

[A-Za-z]



Implementing Maximal Munch

T_Do

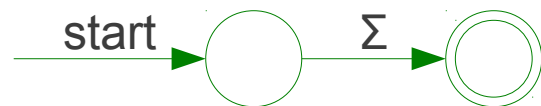
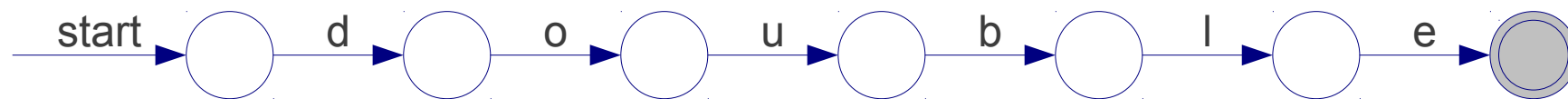
do

T_Double

double

T_Mystery

[A-Za-z]

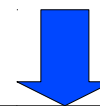


D O

U

B

D O U B L E



Implementing Maximal Munch

T_Do

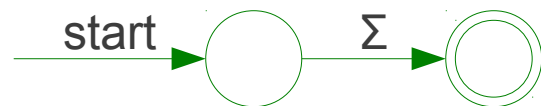
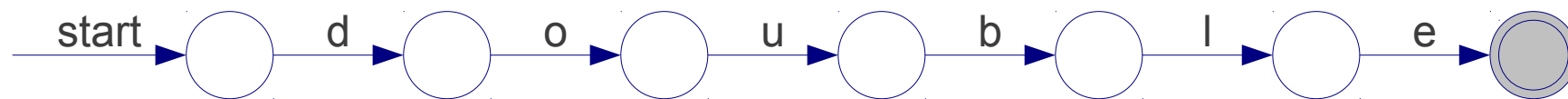
do

T_Double

double

T_Mystery

[A-Za-z]



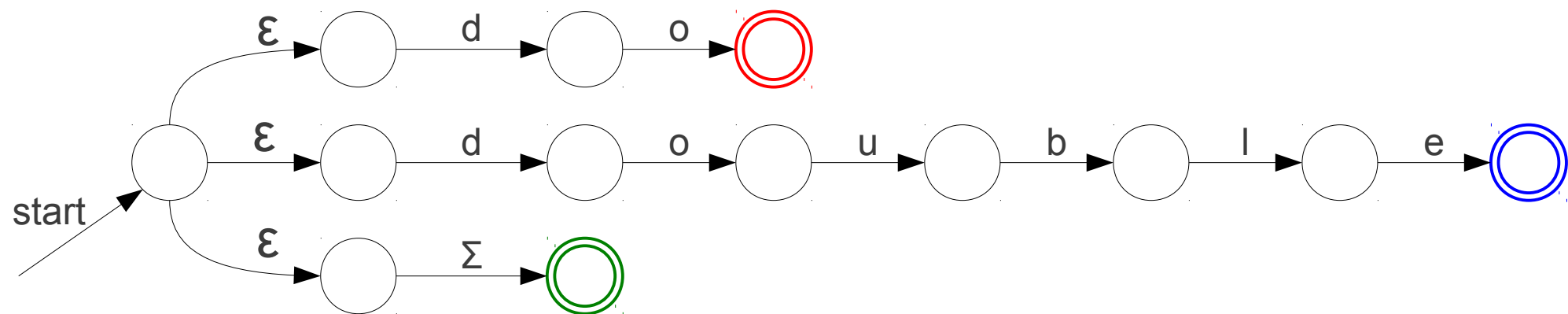
D O

U

B

D O U B L E

A Minor Simplification



Other Conflicts

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

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

More Tiebreaking

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

Other Conflicts

T_Do

do

T_Double

double

T_Identifier [A-Za-z_] [A-Za-z0-9_]*

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

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

Other Conflicts

T_Do

do

T_Double

double

T_Identifier [A-Za-z_] [A-Za-z0-9_]*

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

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

Implement a lexical analyzer

- Use regular expressions to describe token types (keyword, identifier, integer constant..)
- Use DFA/NFA to recognize the regular language
- But...good news. you don't need to implement the algorithms to transform your regular expressions to DFA/NFA to recognize it
- **flex**: given regular expressions -> output c code that does lexical analysis (it internally

Summary

- Lexical Analysis
- Tokens
- Lexemes
- Regular expressions
 - DFA
 - NFA
- Flex: a tool for you to build a lexical analyzer