بسم الله الرحمن الرحيم

Lexical Analysis

Compilers - 002 - S02

نکات صنفی

۱ــ پیش از حذف و اضافه نهایی کنیم!

— مثلا میان‌ترم

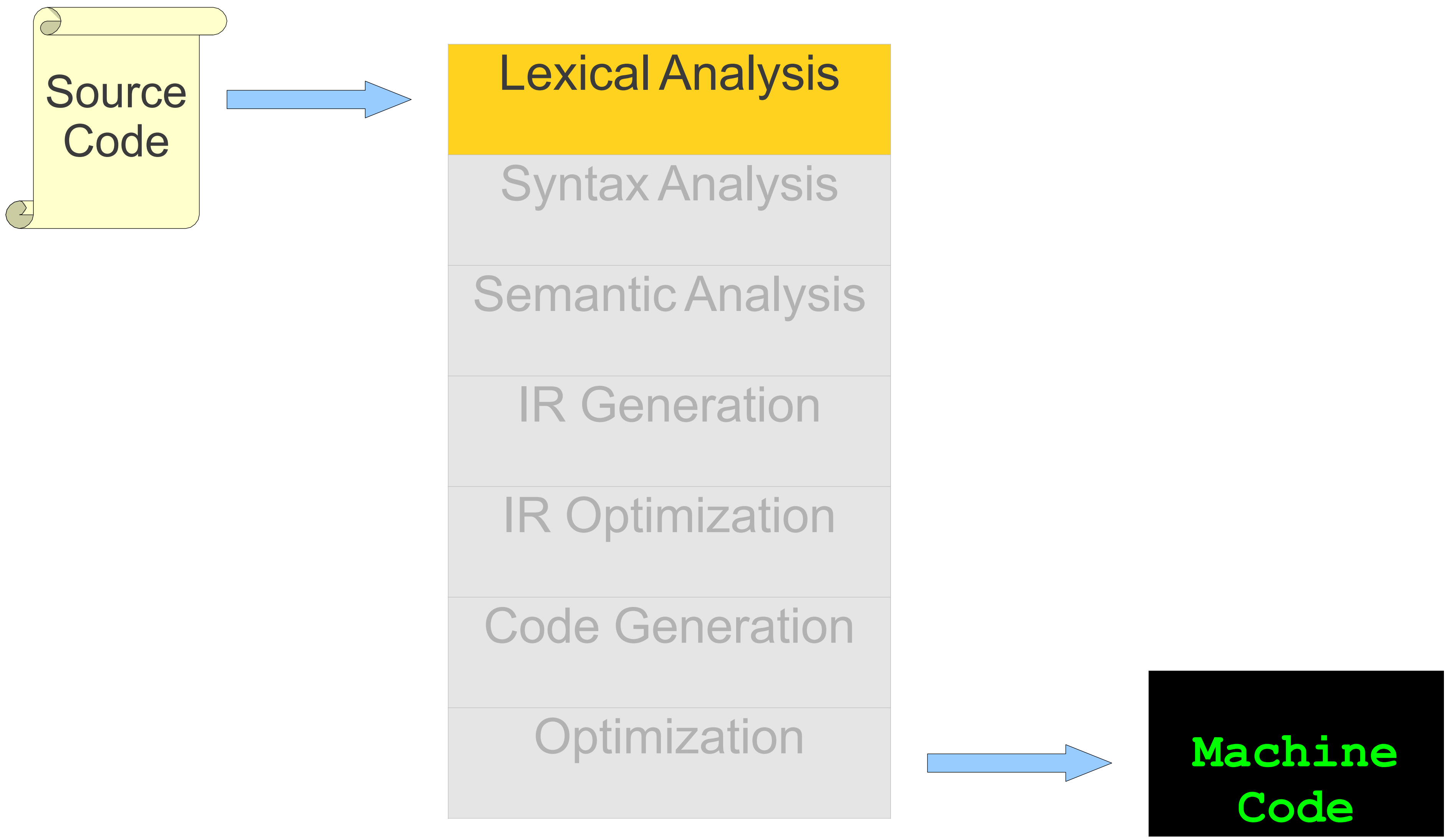
— پروژه‌ها دو نفره، تمرین‌ها تک‌نفره

۲ــ زندگی خوب، حال خوب

۳ــ گروه درس،

۴ــ در کوئرا عضو شوید

Where We Are


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

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

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

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

T_While	(	T_Ident	<	T_Ident	)	++	T_Ident
		ip		z			ip

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

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

```
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;
```

Lexical Analyzer and Parser Communication

Lexical Analyzer and Parser Communication

- Usually parser (Syntax Analyzer) initiates compilation process.

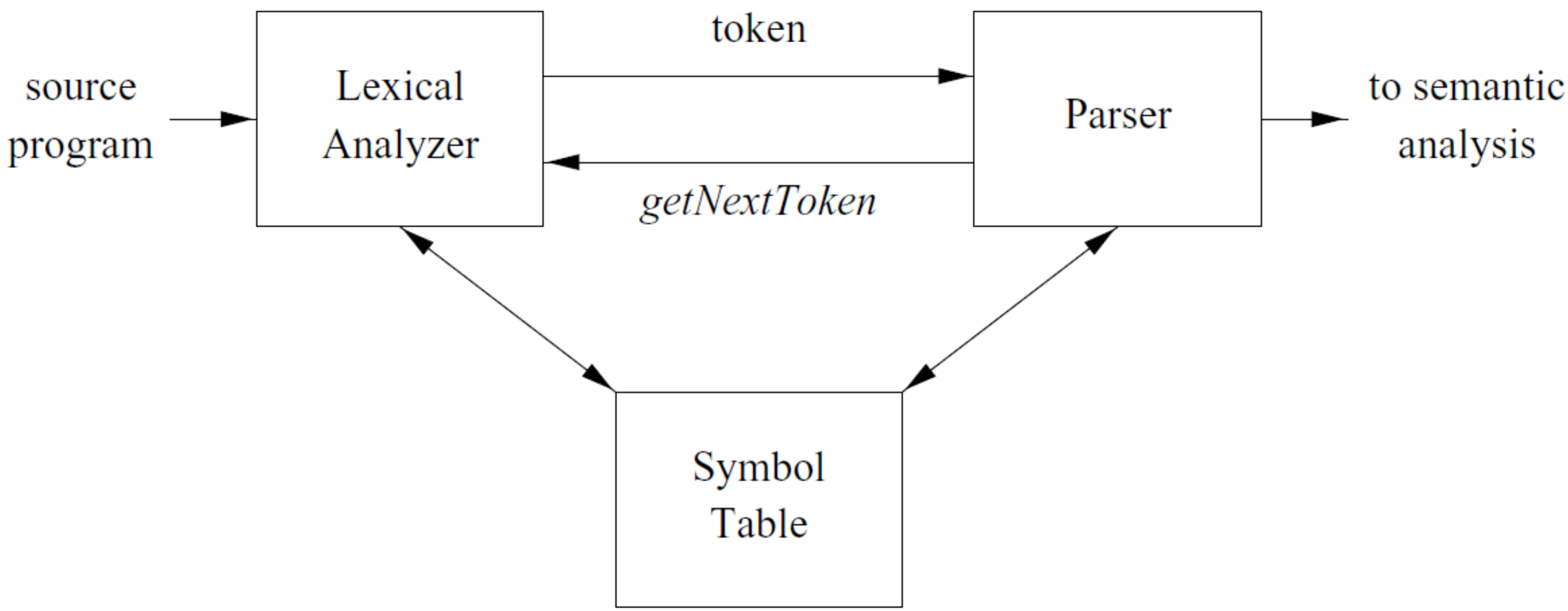
Lexical Analyzer and Parser Communication

- Usually parser (Syntax Analyzer) initiates compilation process.
- With this assumption we have a **passive** scanner.
    - i.e. parser **calls** scanner's function and scanner **returns** a token.

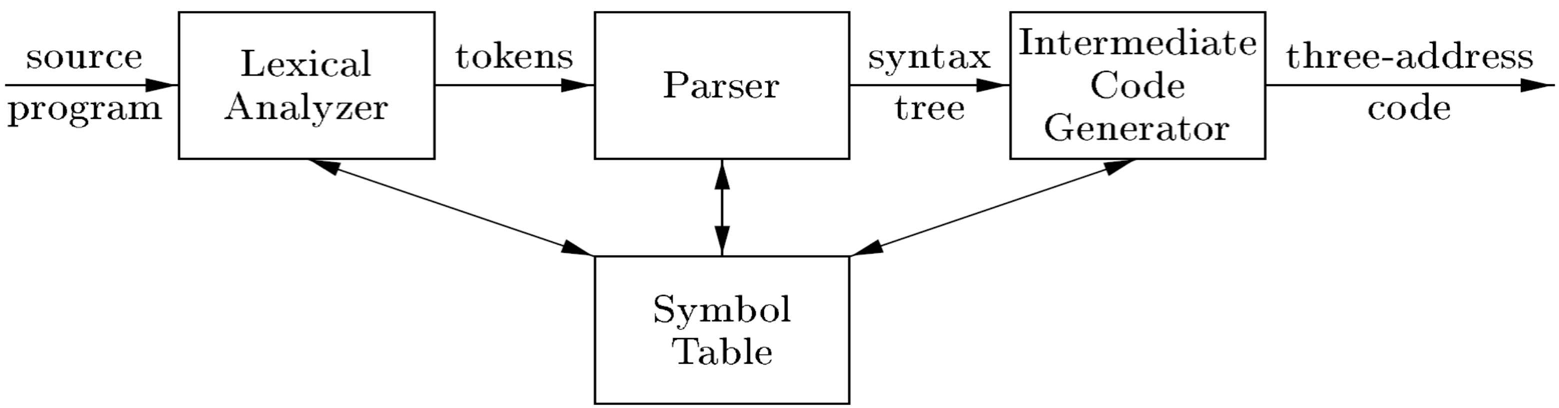
Lexical Analyzer and Parser Communication

- Usually parser (Syntax Analyzer) initiates compilation process.
- With this assumption we have a **passive** scanner.
  - i.e. parser **calls** scanner’s function and scanner **returns** a token.
- How scanner should report tokens?
  - **Coding**: e.g. T_plus = 1, T_id = 2, T_int = 3, T_if =4 ,and etc.

Passive Scanner in Compiler Structure



Lexical Analyzer: Another Look



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	;

The piece of the original program from which we made the token is called a **lexeme**.

`T_While`

This is called a **token**. You can think of it as an enumerated type representing what logical entity we read out of the source code.

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

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`

Sometimes we will discard a lexeme rather than storing it for later use.
Here, we ignore whitespace, since it has no bearing on the meaning of the program.

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

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` `(`

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	(

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` `(`

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` `(` `T_IntConst`

`137`

Scanning a Source File

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

`T_While` `(` `T_IntConst`

`137`

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

Token and Lexeme

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Goals of Lexical Analysis

Goals of Lexical Analysis

Convert from physical description of a program into sequence of of **tokens**.

- Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.

Goals of Lexical Analysis

Convert from physical description of a program into sequence of of **tokens**.

- Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.

Each token is associated with a **lexeme**.

- The actual text of the token: “137,” “int,” etc.

Each token may have optional **attributes**.

- Extra information derived from the text – perhaps a numeric value.

The token sequence will be used in the parser to recover the program structure.

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;
}
```

```
for         {
int         }
<<          ;
=           <
(           [
)           ]
++
```

What Tokens are Useful Here?

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

```
for             {
int             }
<<              ;
=               <
(               [
)               ]
++

Identifier
IntegerConstant
```

Choosing Good Tokens

- Very much dependent on the language.
- Typically:

Choosing Good Tokens

- Very much dependent on the language.
- Typically:

  Give keywords their own tokens.

Choosing Good Tokens

- Very much dependent on the language.
- Typically:

    Give keywords their own tokens.

    Give different punctuation symbols their own tokens.

Choosing Good Tokens

- Very much dependent on the language.
- Typically:

  Give keywords their own tokens.

  Give different punctuation symbols their own tokens.

  Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.

Choosing Good Tokens

- Very much dependent on the language.
- Typically:

    Give keywords their own tokens.

    Give different punctuation symbols their own tokens.

    Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.

    Discard irrelevant information (whitespace, comments)

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

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

Read this!

Thanks to Prof. Alex Aiken

Scanning is Hard

- FORTRAN: Whitespace is irrelevant

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

Thanks to Prof. Alex Aiken

Scanning is Hard

- 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

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)))
```

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

Scanning is Hard

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

Scanning is Hard

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

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

Thanks to Prof. Alex Aiken

Scanning is Hard

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

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

Scanning is Hard

- 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

Challenges in Scanning

-
-
-

Challenges in Scanning

- How do we determine which lexemes are associated with each token?
-
-

Challenges in Scanning

- 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?
-

Challenges in Scanning

- 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?
- How do we address these concerns efficiently?

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what **information** they provide.
- Some tokens might be associated with only a single lexeme:
    - Tokens for keywords like **`if`** and **`while`** probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes:
    - All variable names, all possible numbers, all possible strings, etc.

Sets of Lexemes

Sets of Lexemes

Idea: Associate a set of lexemes with each token.

Sets of Lexemes

Idea: Associate a set of lexemes with each token.

We might associate the “number” token with the set { `0`, `1`, `2`, …, `10`, `11`, `12`, … }

Sets of Lexemes

Idea: Associate a set of lexemes with each token.

We might associate the “number” token with the set { **`0`**, **`1`**, **`2`**, …, **`10`**, **`11`**, **`12`**, … }

We might associate the “string” token with the set { **`""`**, **`"a"`**, **`"b"`**, **`"c"`**, … }

Sets of Lexemes

Idea: Associate a set of lexemes with each token.

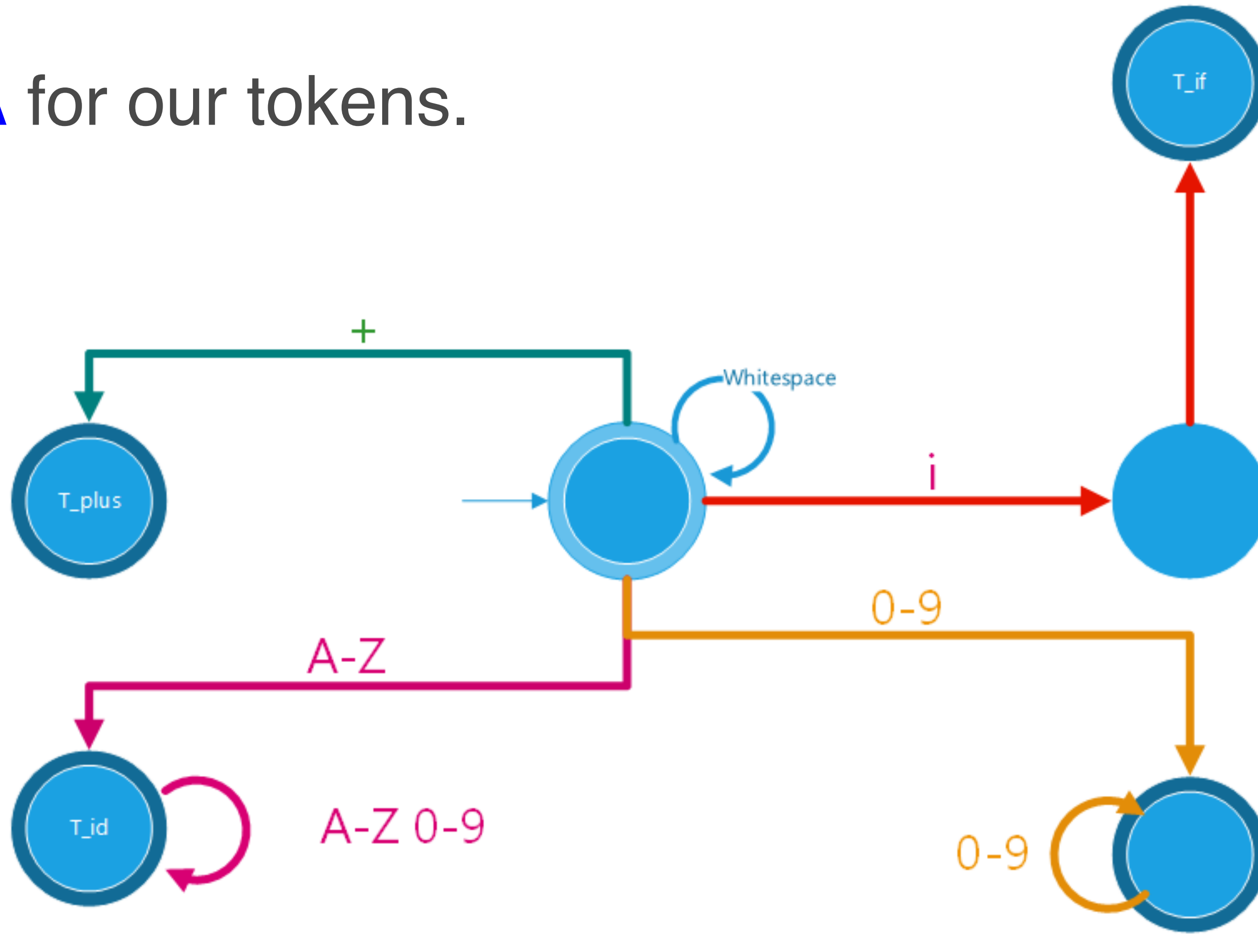
We might associate the “number” token with the set { `0`, `1`, `2`, …, `10`, `11`, `12`, … }

We might associate the “string” token with the set { `""`, `"a"`, `"b"`, `"c"`, … }

We might associate the token for the keyword `while` with the set { `while` }.

Lexical Analyzer: First Idea

- We design a **DFA** for our tokens.



We investigate a hand-made scanner

Code the DFA

- Structure would be:

```
//Define codes
function scanner()
{
switch (char)
        case 1:
                .. .. ..
                return code;
        .
        .
        .
        case n:
}
```

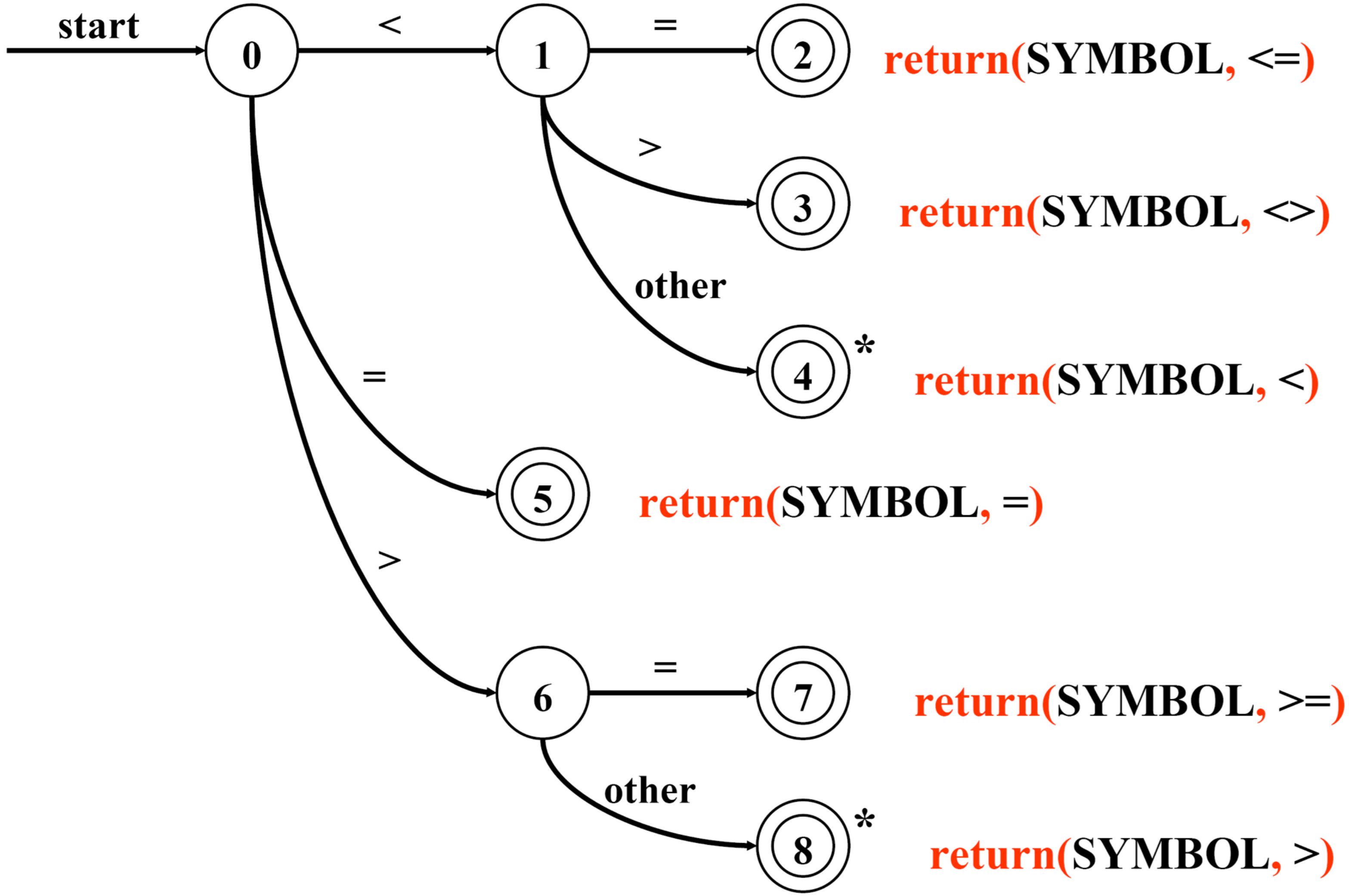
```
enum class TokenCode
{
    T_plus,
    T_id,
    T_int,
    T_if
}

char ch; // to read chars
int picv; // to store Positive Integer Constatn Value
string idval; // to store id
```

```
TokenCode scanner()
{
_LS:
    switch (ch)
    {
        case '+':           // plus found
            ch = getchar();
            return TokenCode::T_plus;
            break;
```

```
        case 'i':            // mybe it is if
             ch = getchar();
             if(ch == 'f') {
                 ch = getchar();
                 return TokenCode::T_if;
             }
             break;
         case '0' ... '9':   //integer constant
             picv = 0;
             while(ch <= '9' && ch >= '0') {
                 picv = picv * 10 + (ch - '0');
                 ch = getchar();
             }
             return TokenCode::T_int;
             break;
```

```
        case 'A' ... 'Z':   // ID is starting!
            idval = ch;
            ch = getchar();
            while (('A' <= ch && ch <= 'Z') || (ch <= '9' && ch >= '0')) {
                idval += ch;
                ch = getchar();
            }
            return TokenCode::T_id;
            break;
        case ' ' :  case '\f':  case '\n': // whitespaces
        case '\r':  case '\t':  case '\v':
            ch = getchar();
            goto _LS;
            break;
        default:
            cout << "Error: Undefined pattern.\n";
            break;
    }
}
```

Operator Diagram



A Better Way: Find a way to describe which 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.

Regular Expressions

Regular Expressions

- **Regular expressions** are a family of descriptions that can be used to capture certain languages (the *regular languages*).

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.

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, **R_1R_2** is a regular expression represents the **concatenation** of the languages of R_1 and R_2.
- If R_1 and R_2 are regular expressions, **R_1 | R_2** is a regular expression representing the **union** of R_1 and R_2.
- If R is a regular expression, **R*** is a regular expression for the **Kleene closure** of R.
- If R is a regular expression, **(R)** is a regular expression with the same meaning as R.

Operator Precedence

Regular expression operator precedence is

(R)

R^*

R_1R_2

$R_1 \mid R_2$

So **ab*c|d** is parsed as **((a(b*))c)|d**

Simple Regular Expressions

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

(0 | 1)*00(0 | 1)*

Simple Regular Expressions

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

(0 | 1)*00(0 | 1)*

11011100101
0000
1111101111011111

Simple Regular Expressions

- Suppose the only characters are `0` and `1`.
- Here is a regular expression for strings of length exactly four:

(0|1){4}

0000
1010
1111
1000

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

(+|-)?[0-9]*[02468]

42
+1370
-3248
-9999912

Another view

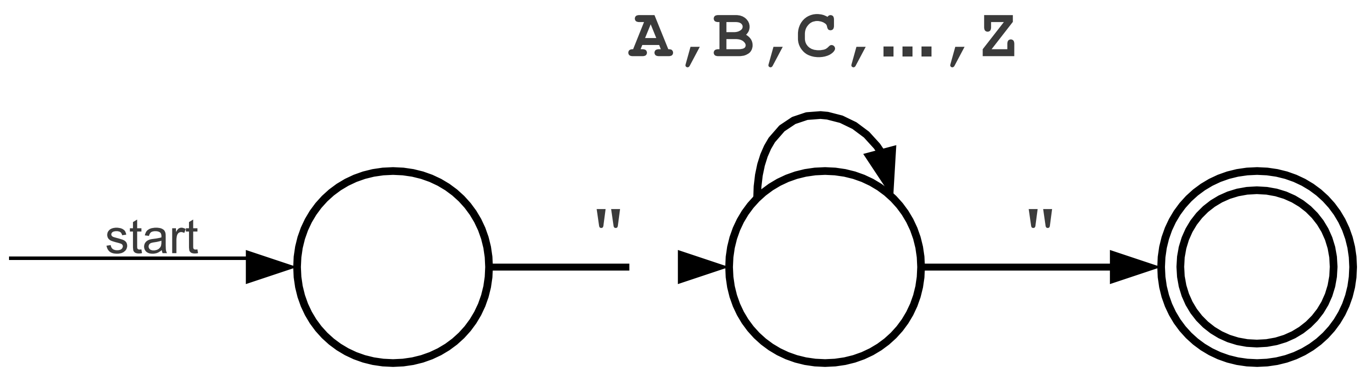
- In the view of a lexical analyzer a program p is **admissible** if $p \in \left\{ \left(t_1 + t_2 + \ \ldots t_n \right)^* \right\}$.
- Each t_i is also a regular expression.
- You can assume that we have relaxed the scanning problem!
- So lexical analyzer, check an announce the constituent elements.
- Returns certificates.

Matching Regular Expressions

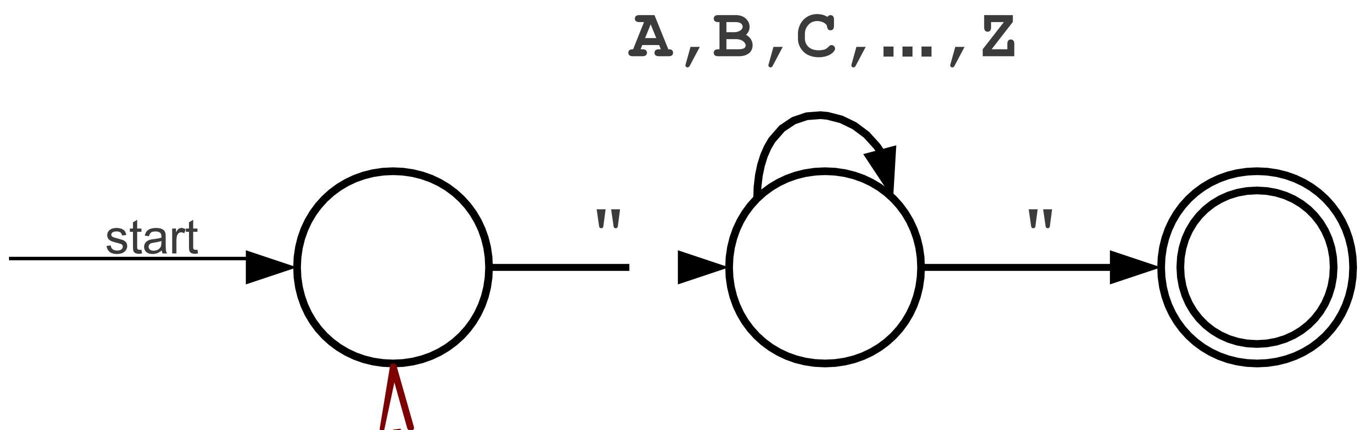
Implementing Regular Expressions

- Regular expressions can be implemented using **finite automata**.
- There are two main 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 Automaton

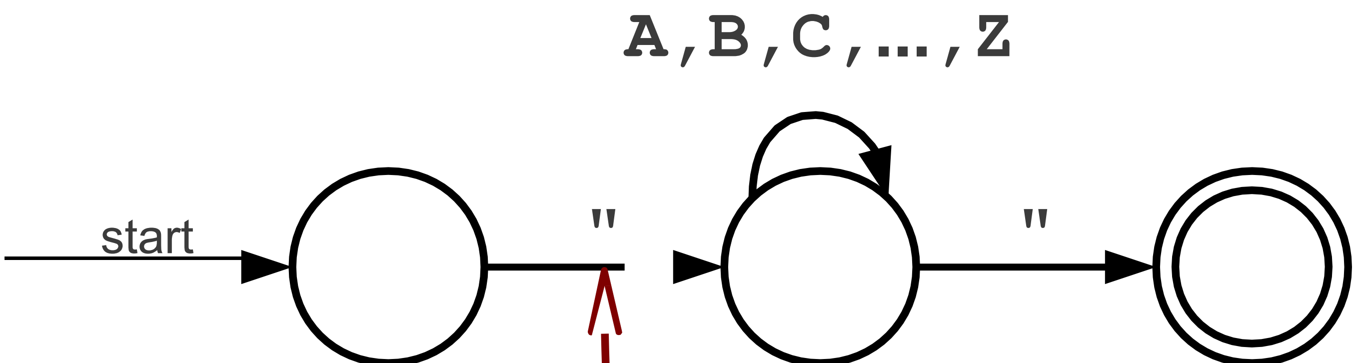


A Simple Automaton



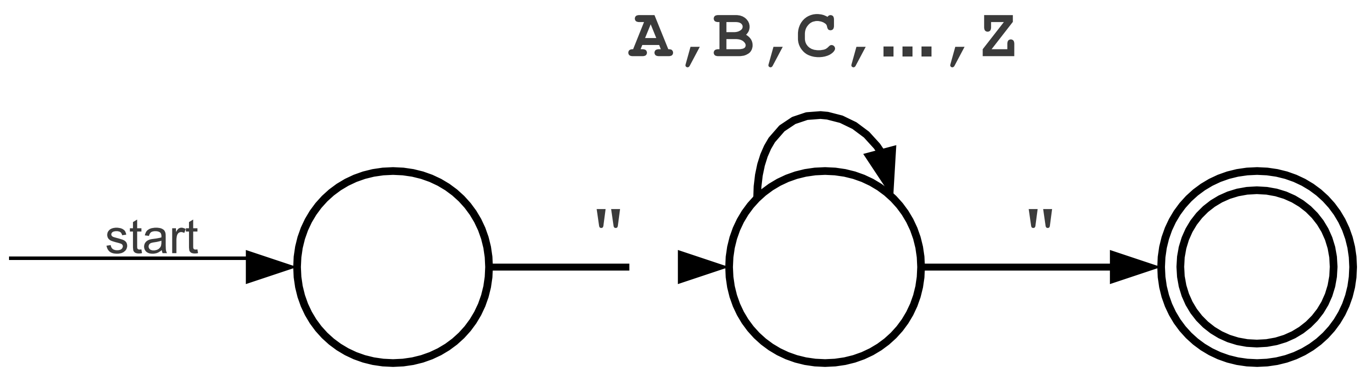
Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

A Simple Automaton



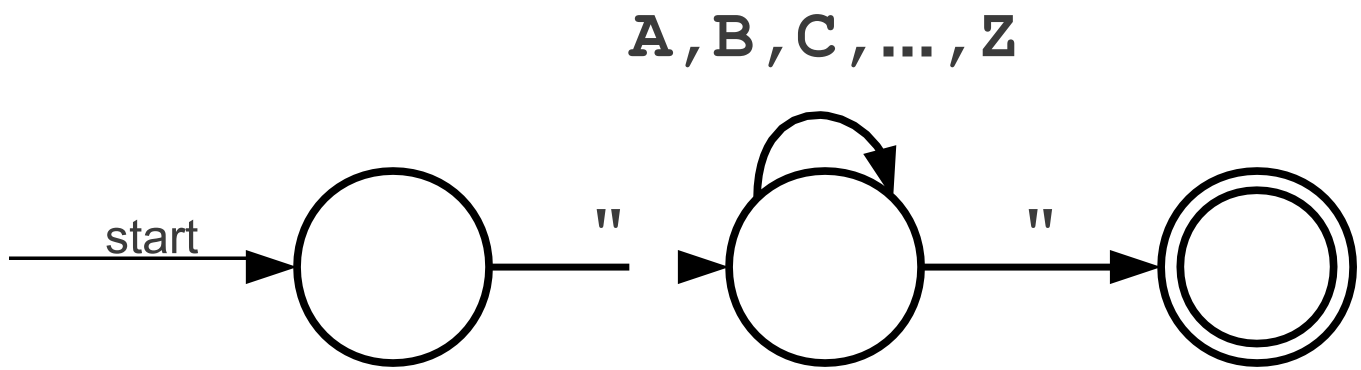
These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

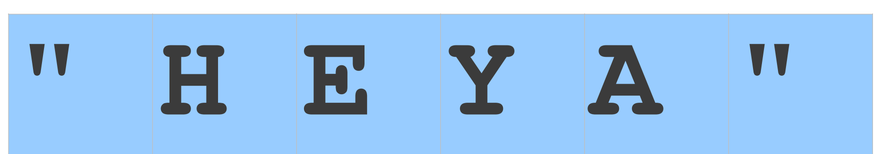
A Simple Automaton



"	H	E	Y	A	"

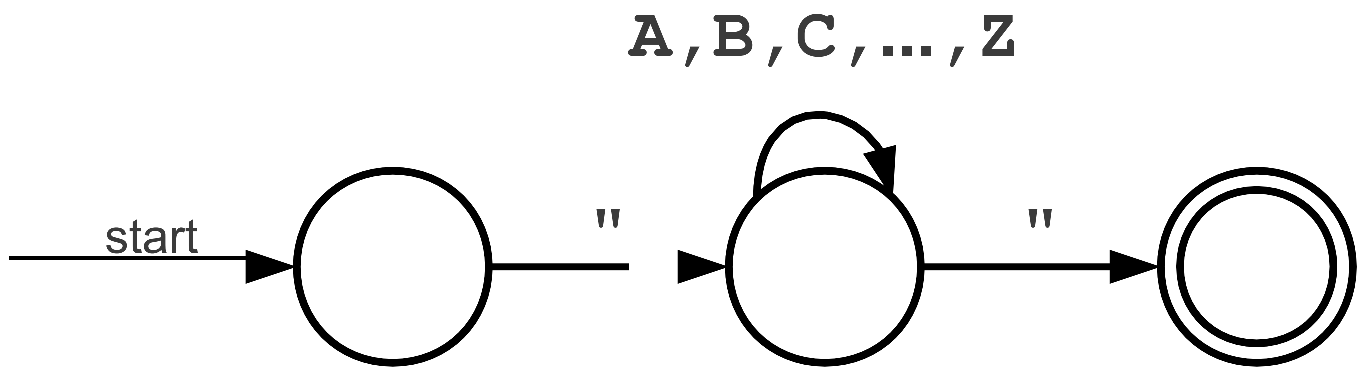
A Simple Automaton

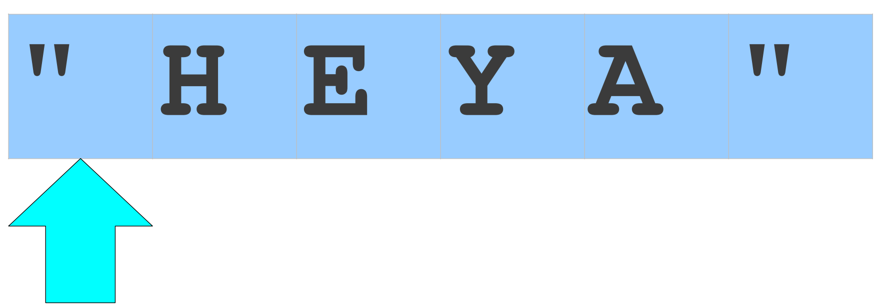




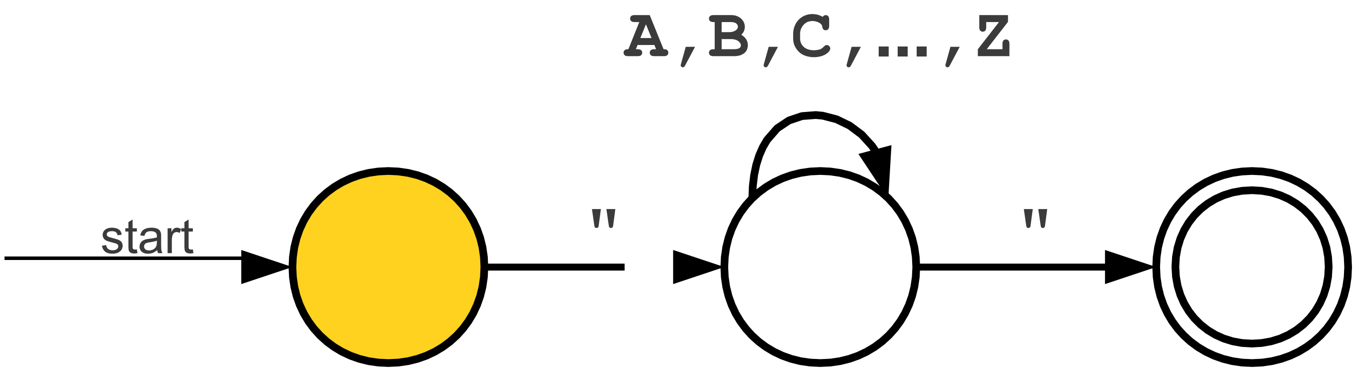
The automaton takes a string as input and decides whether to accept or reject the string.

A Simple Automaton

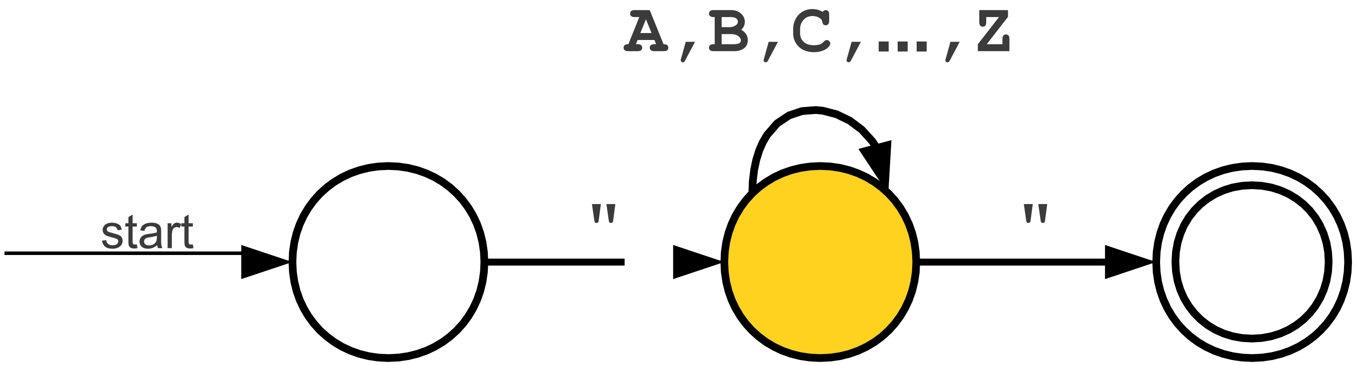


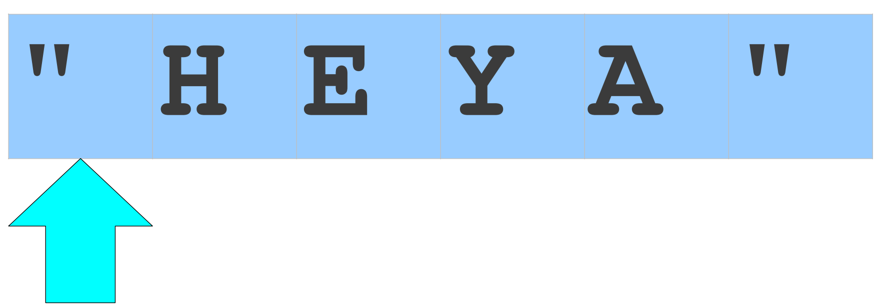


A Simple Automaton

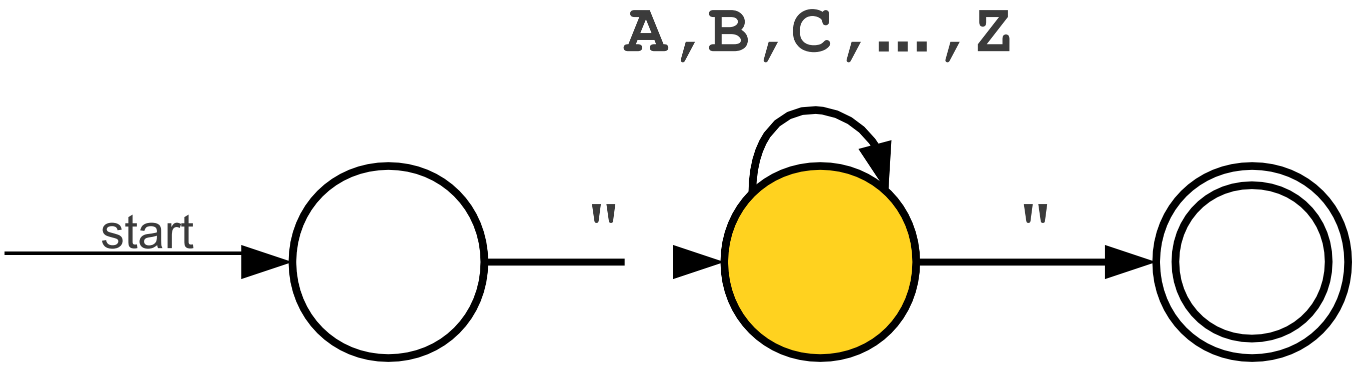


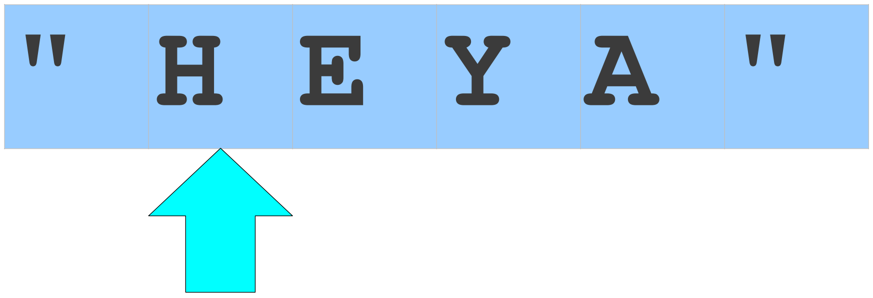
A Simple Automaton



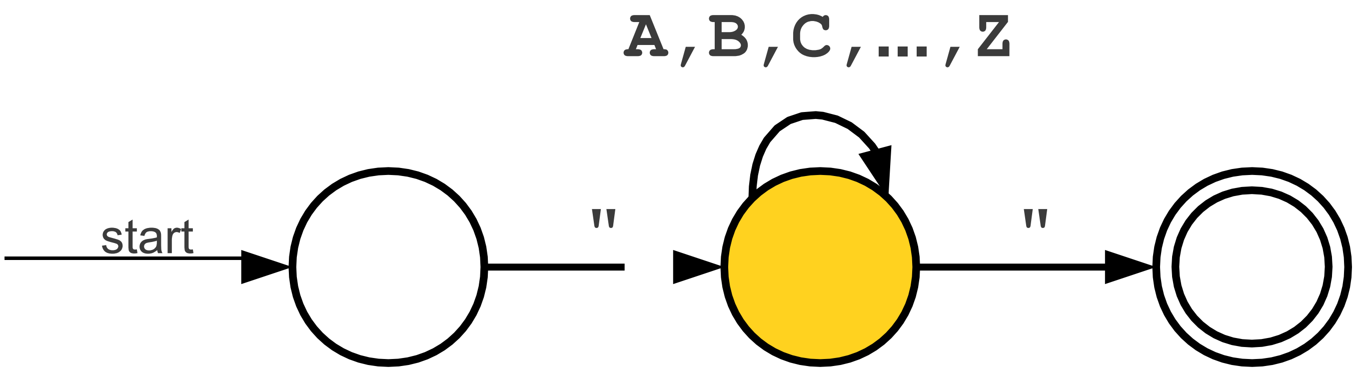


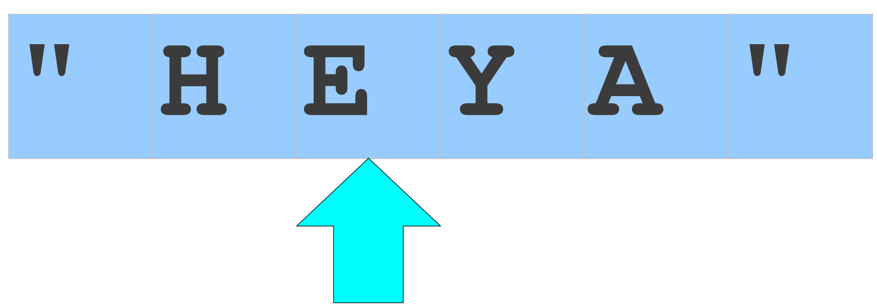
A Simple Automaton



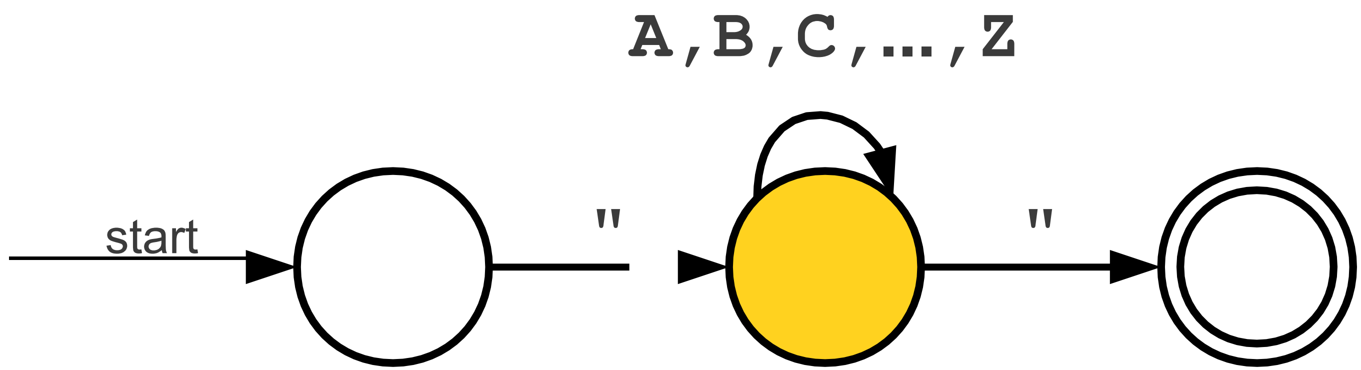


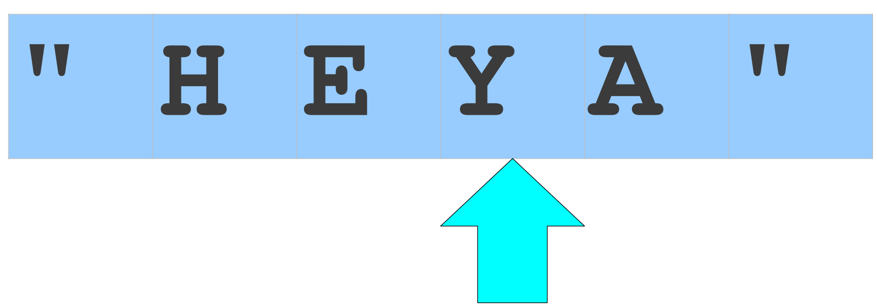
A Simple Automaton



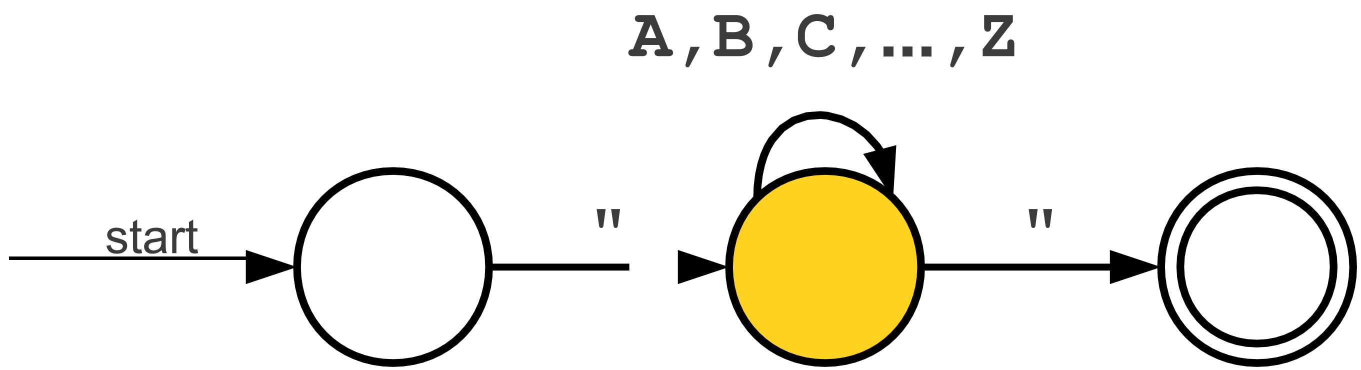


A Simple Automaton

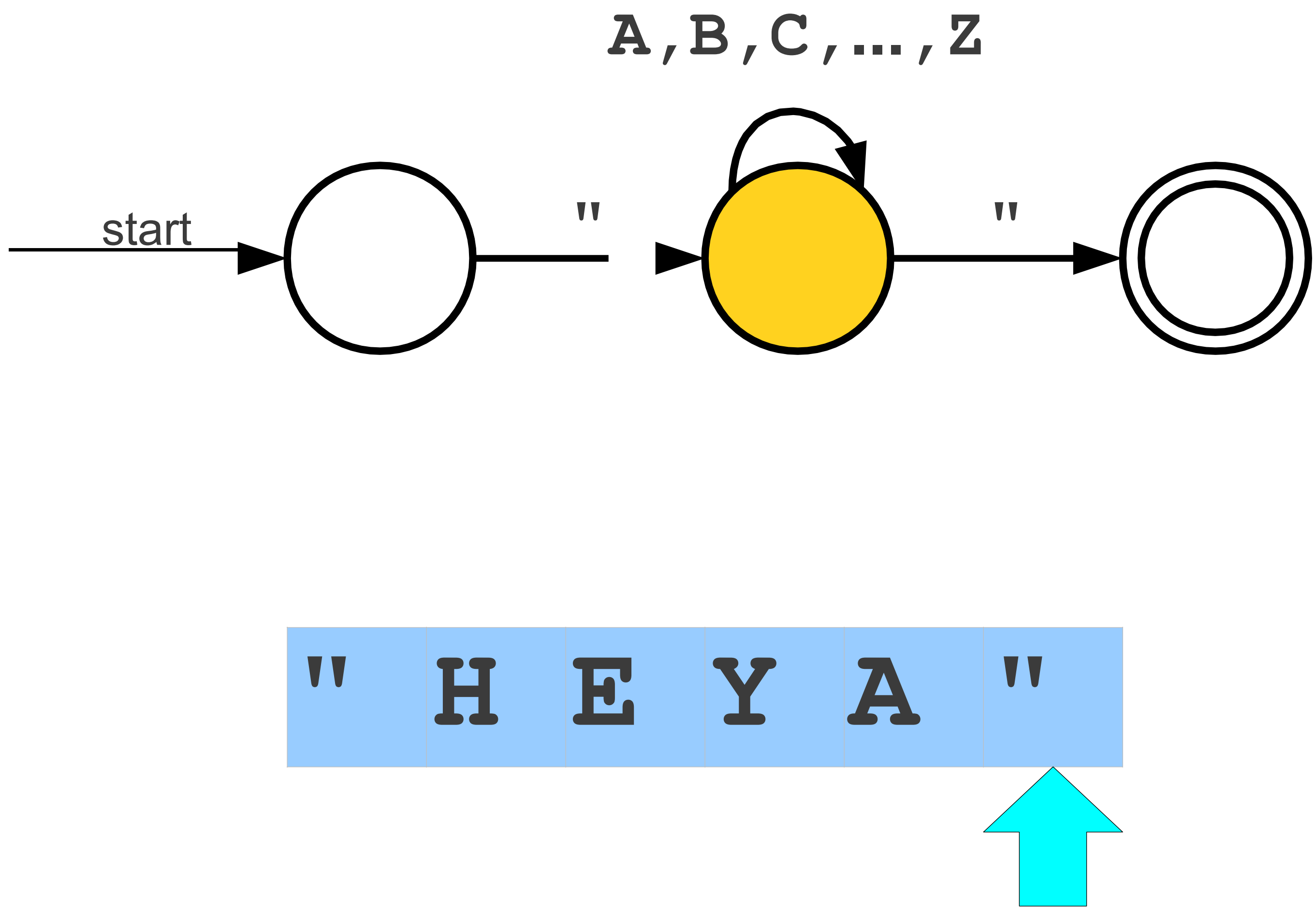




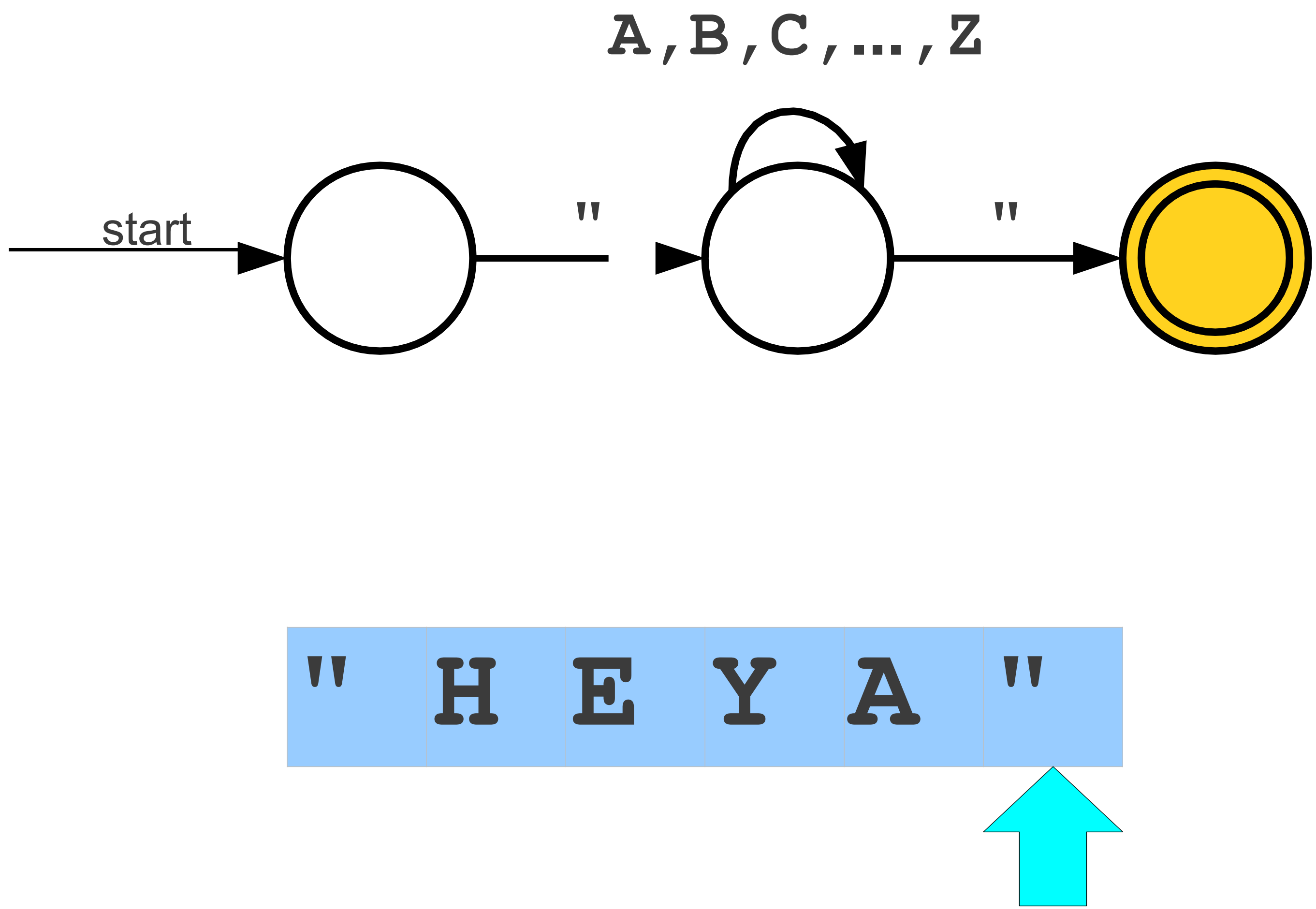
A Simple Automaton



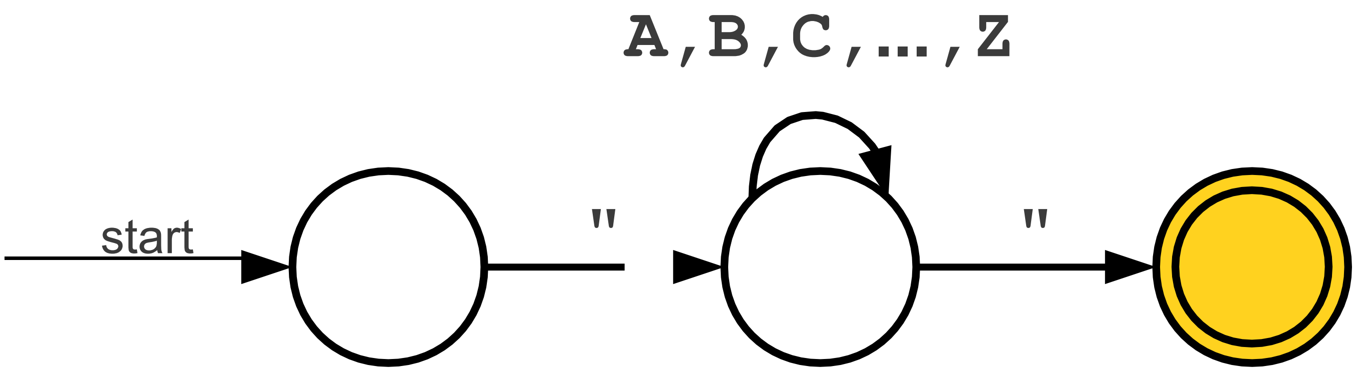
A Simple Automaton



A Simple Automaton

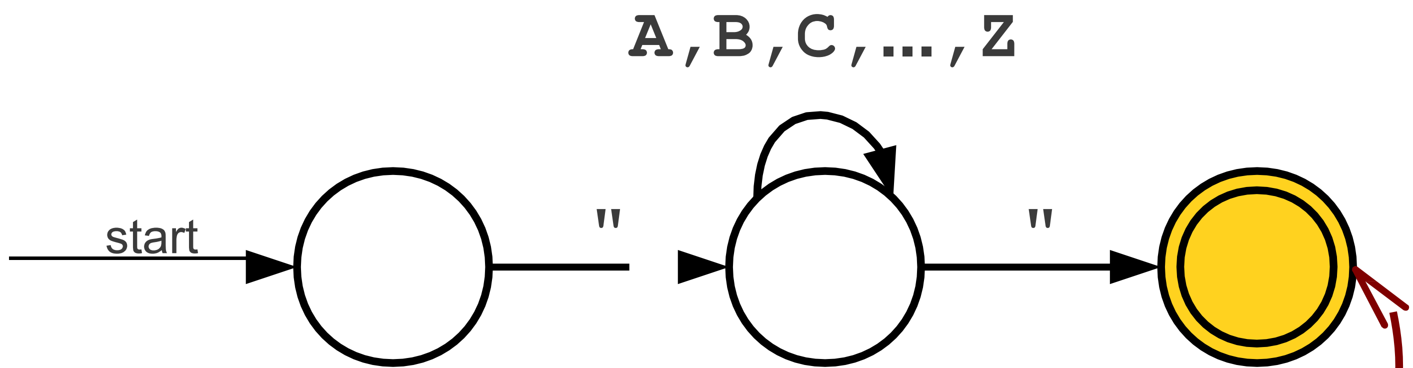


A Simple Automaton



"	H	E	Y	A	"

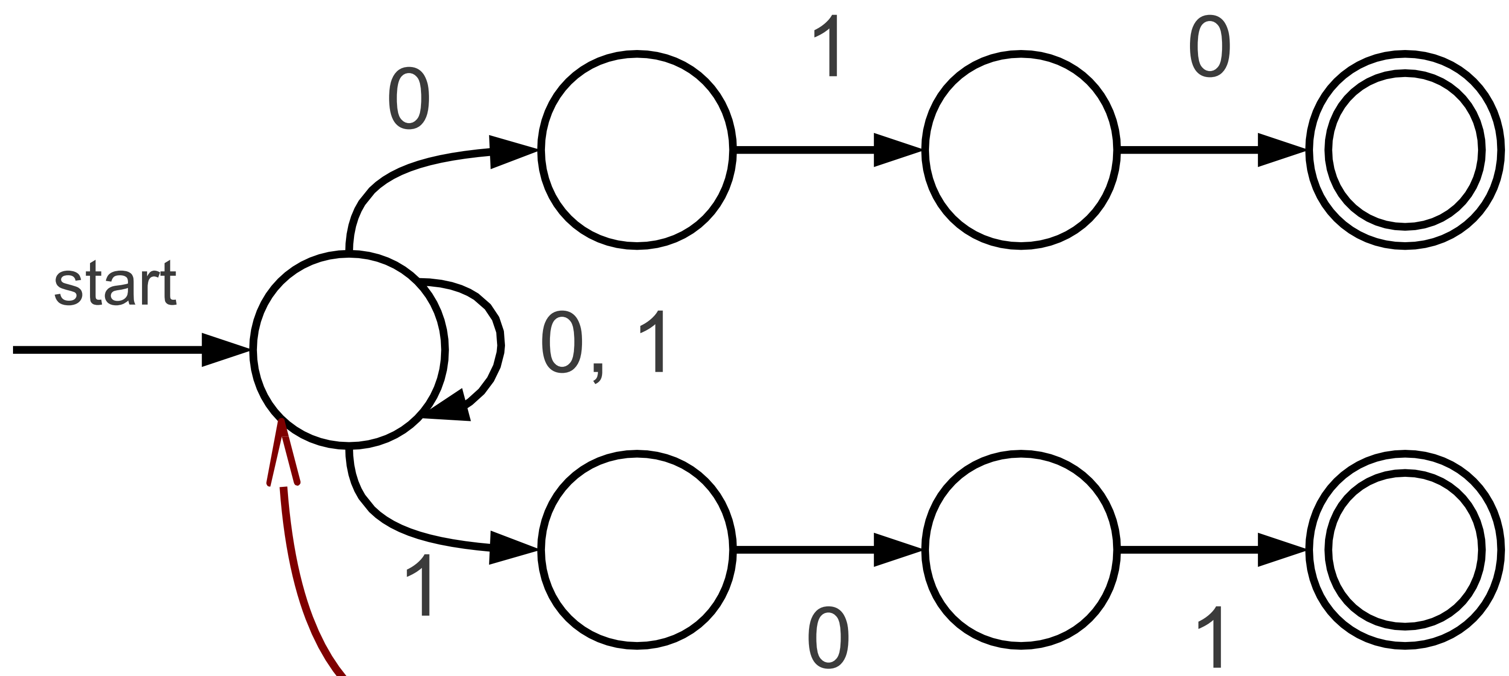
A Simple Automaton



" H E Y A "

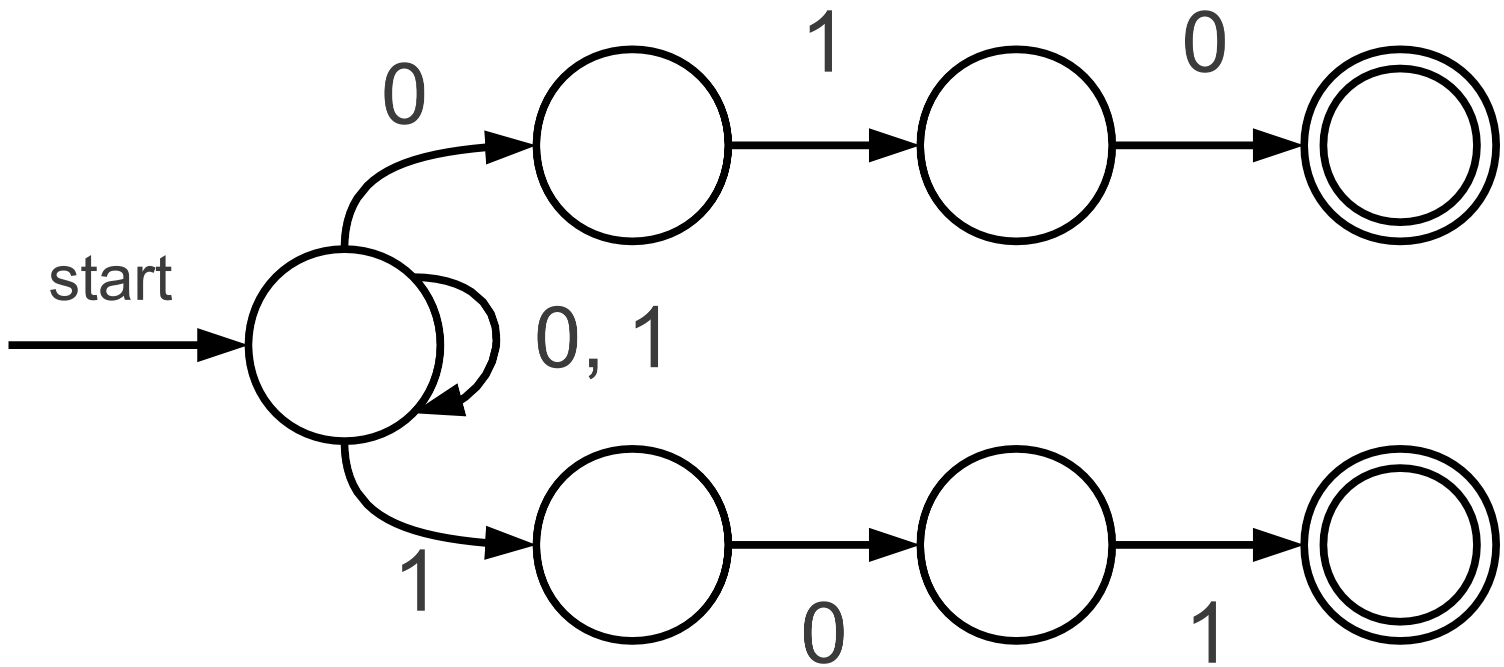
The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.

A More Complex Automaton

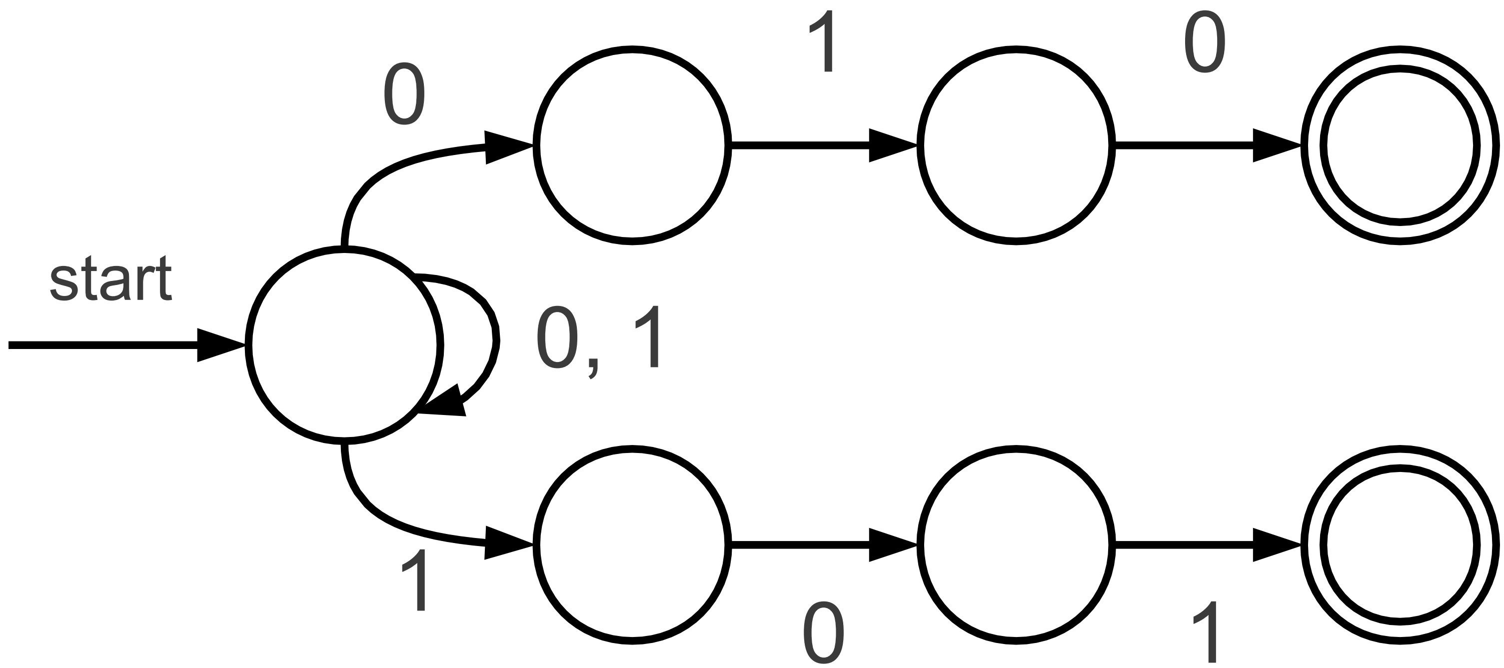


Notice that there are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow *both* transitions and enter multiple states.

A More Complex Automaton

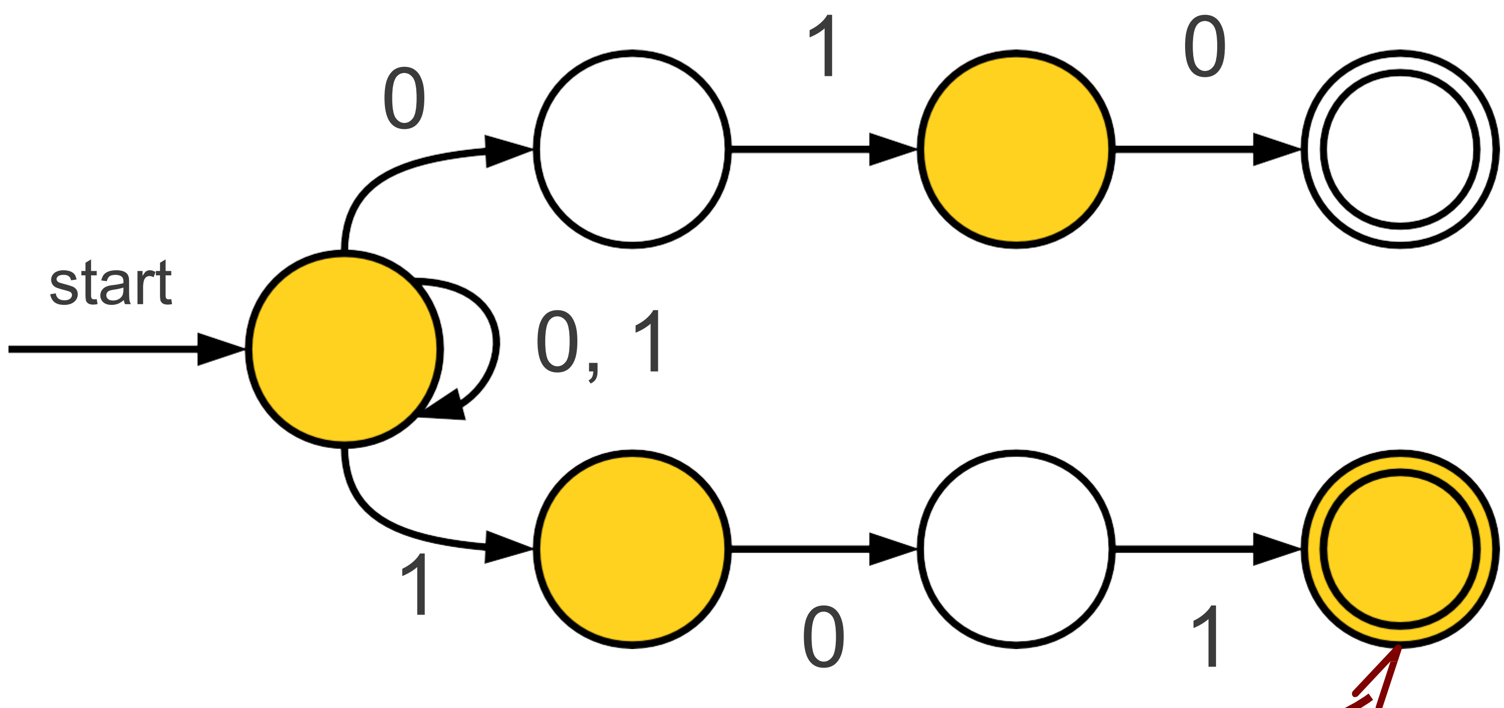


A More Complex Automaton



0	1	1	1	0	1

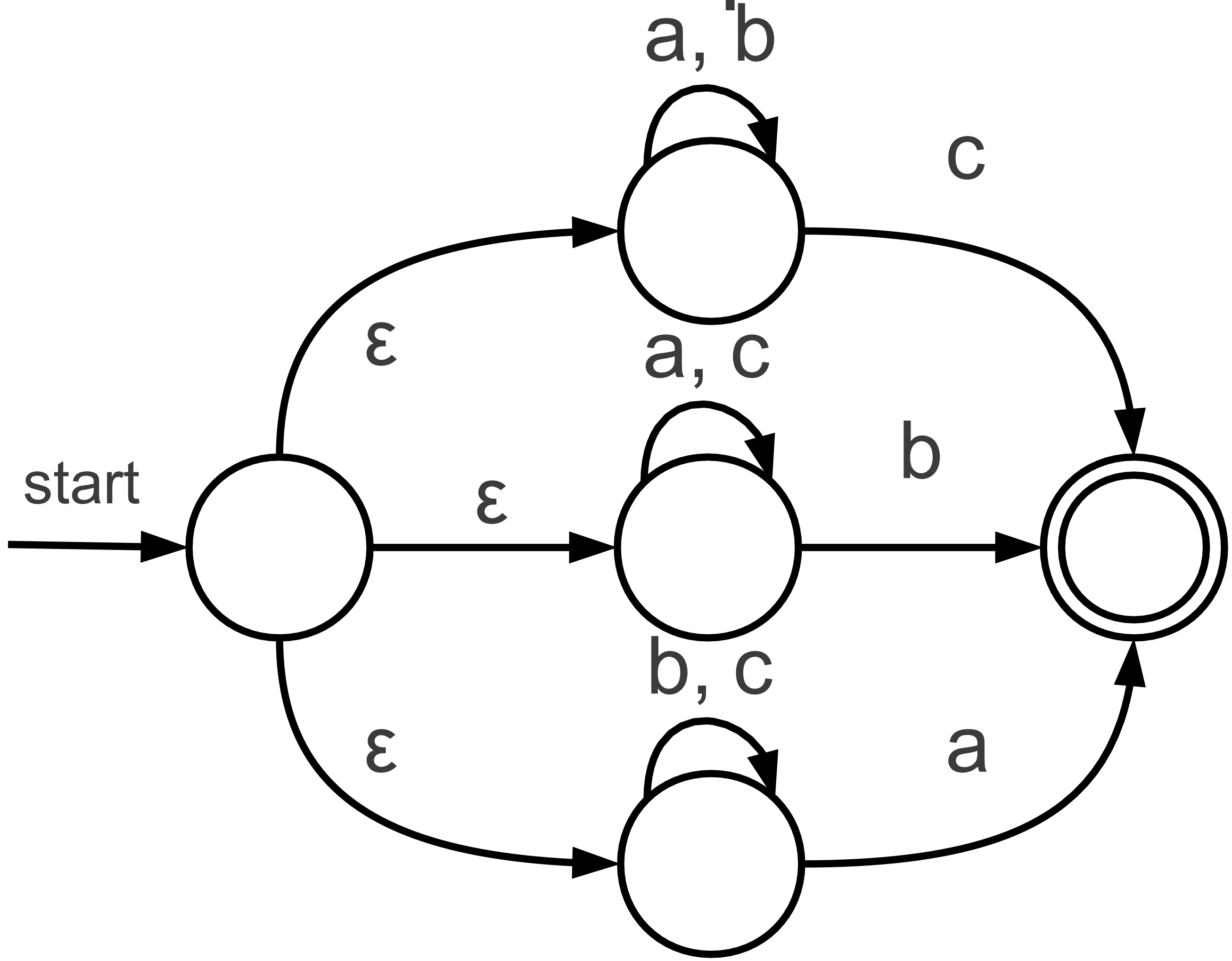
A More Complex Automaton



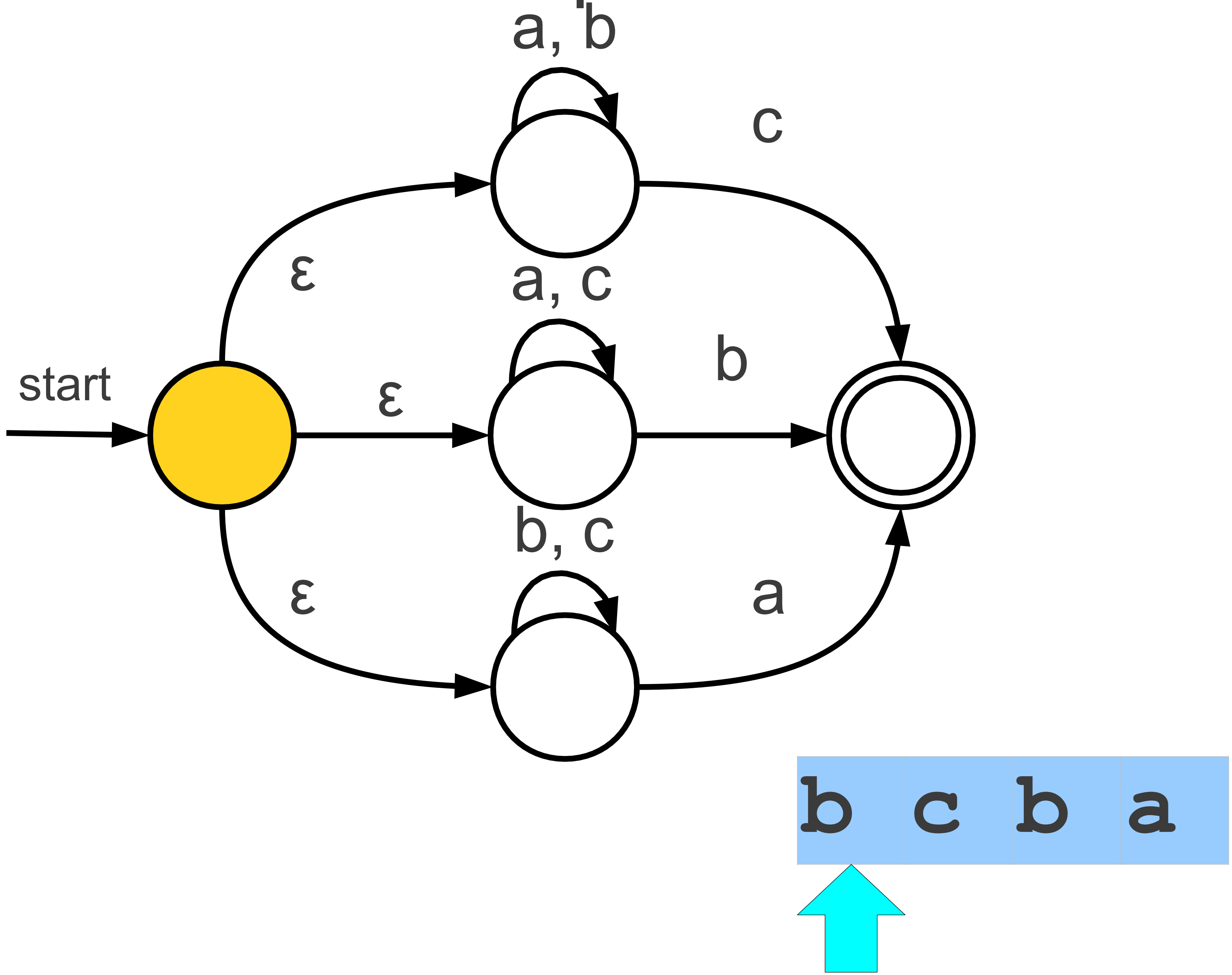
0 1 1 1 0 1

Since we are in at least one accepting state, the automaton accepts.

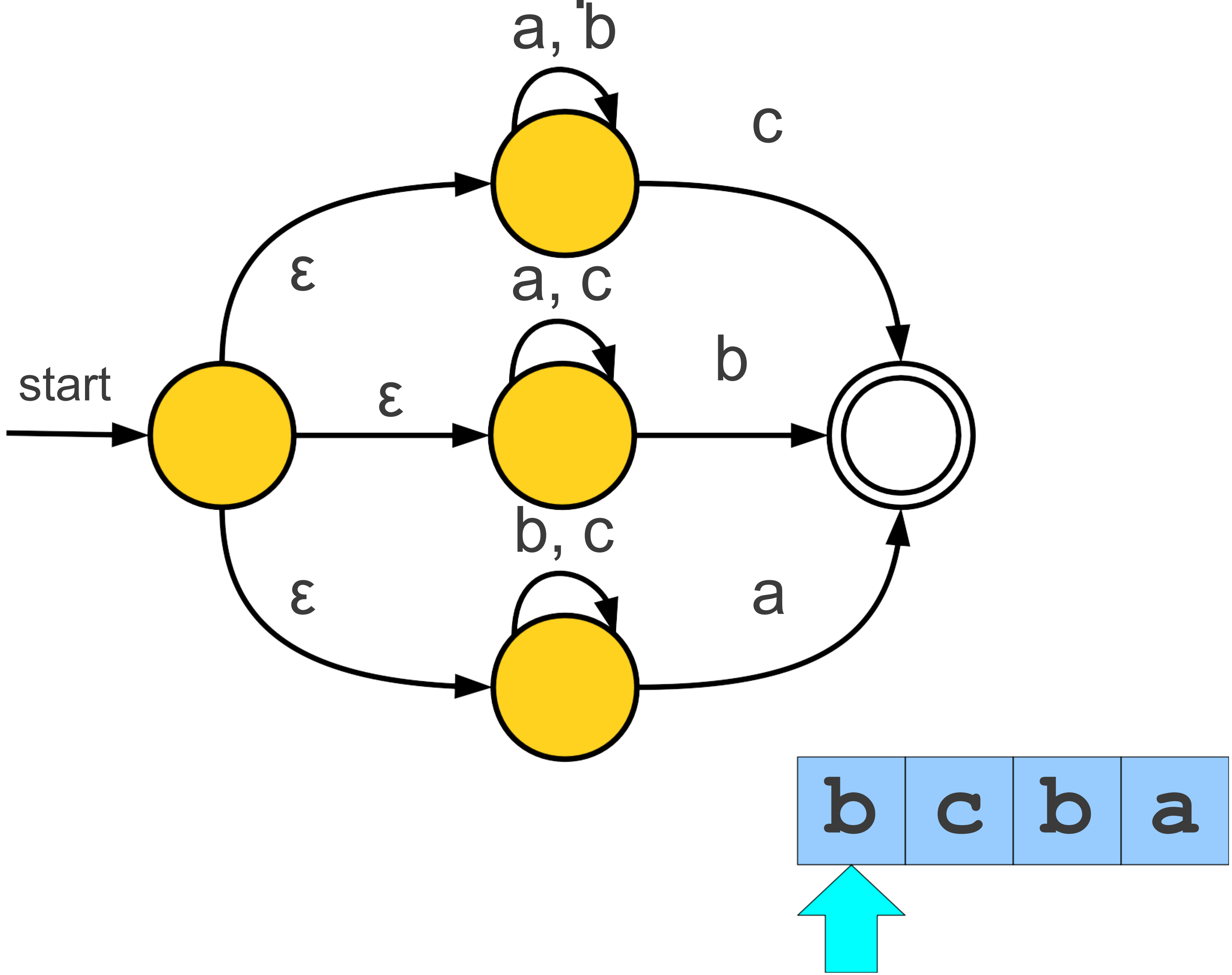
An Even More Complex Automaton



An Even More Complex Automaton

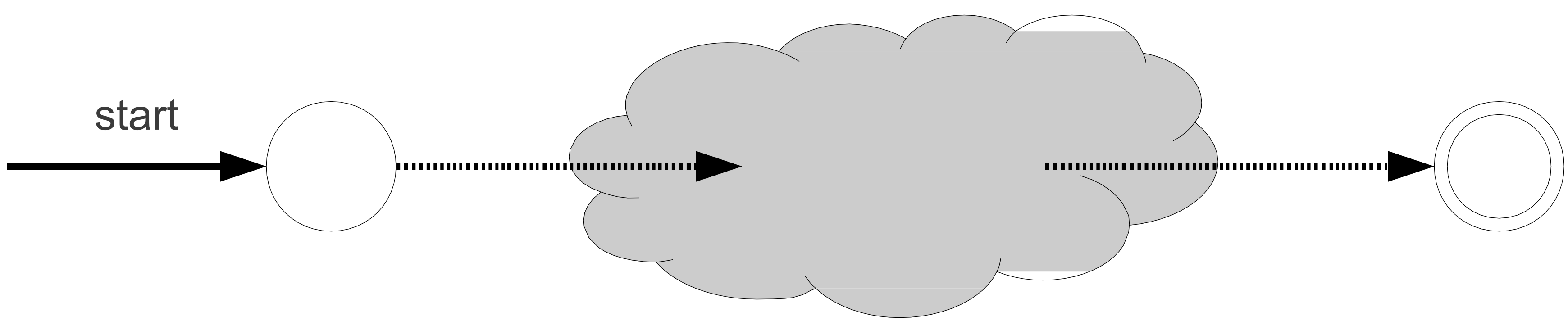


An Even More Complex Automaton

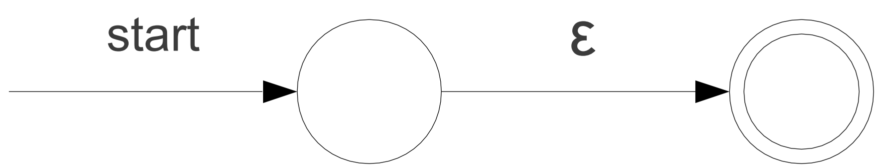


From Regular Expressions to NFAs

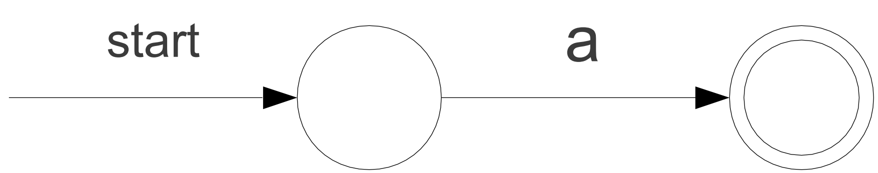
- There is a (beautiful!) procedure from converting a regular expression to an NFA.
- Associate each regular expression with an NFA with the following properties:
  - There is exactly one accepting state.
  - There are no transitions out of the accepting state.
  - There are no transitions into the starting state.
- These restrictions are stronger than necessary, but make the construction easier.



Base Cases



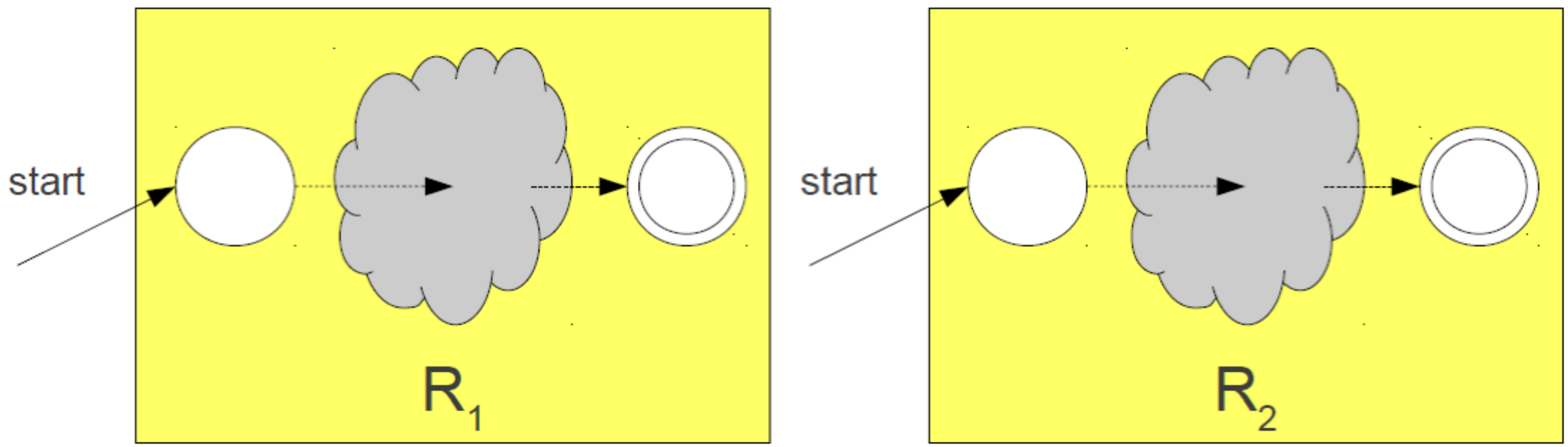
Automaton for ε



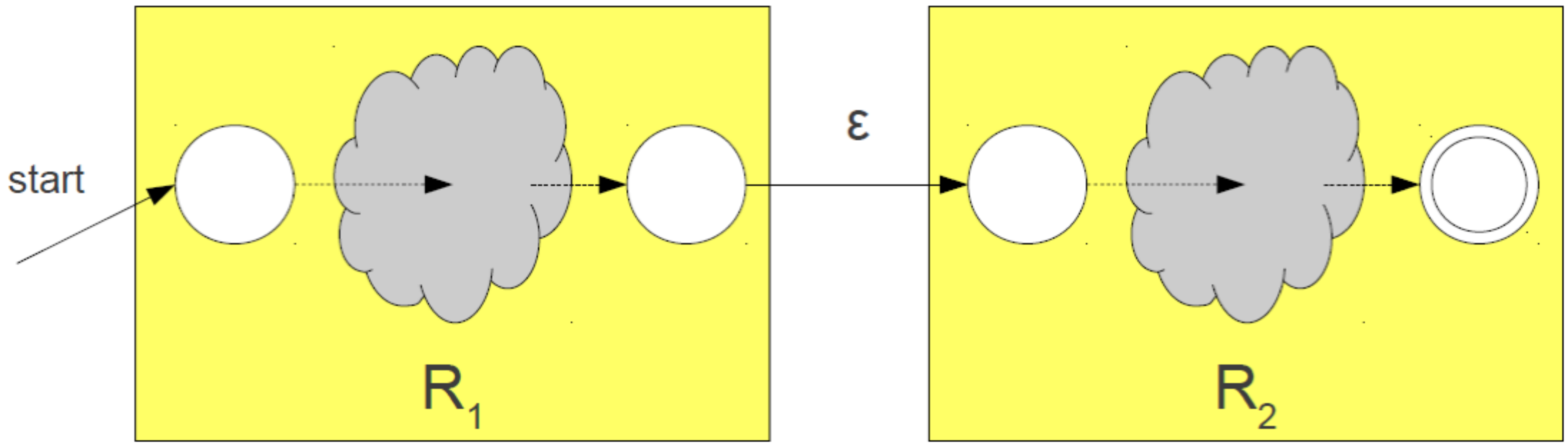
Automaton for single character **a**

Construction for R_1R_2

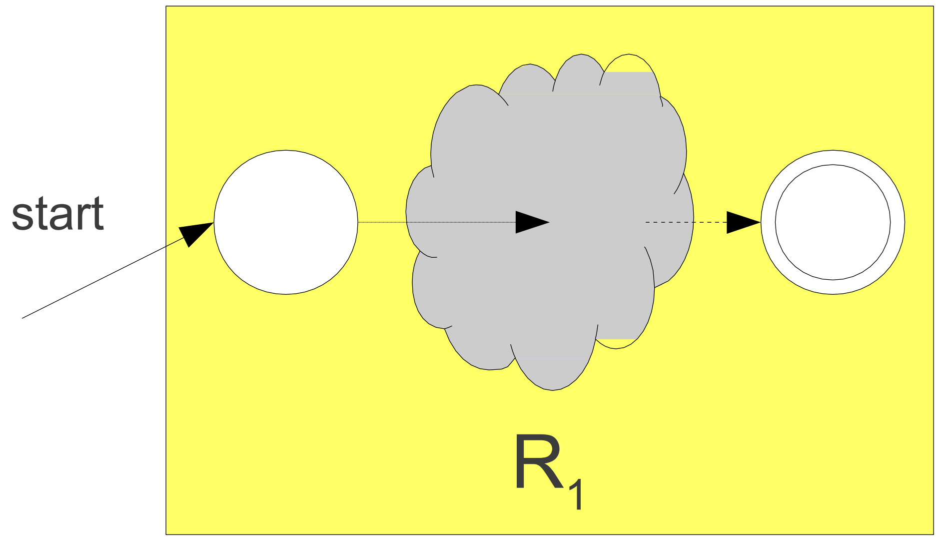
Construction for R_1R_2

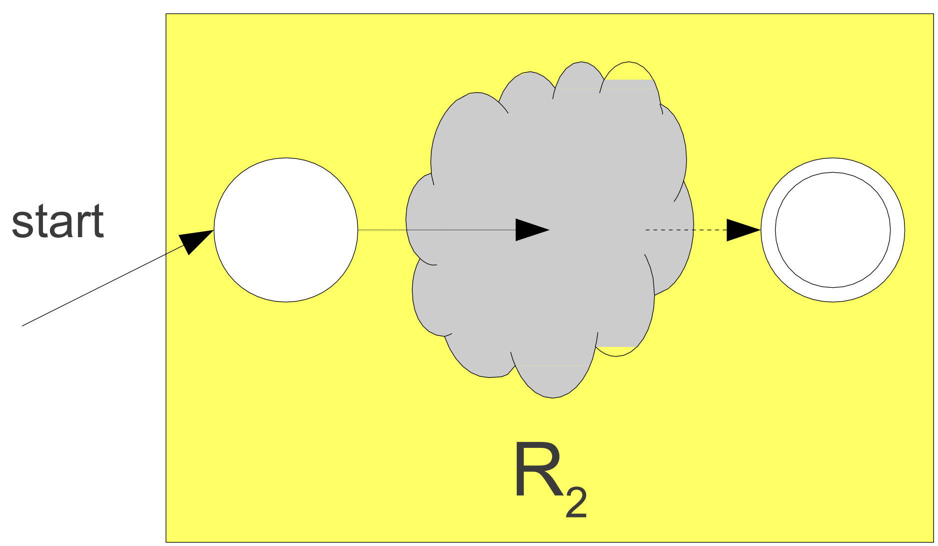


Construction for R_1R_2

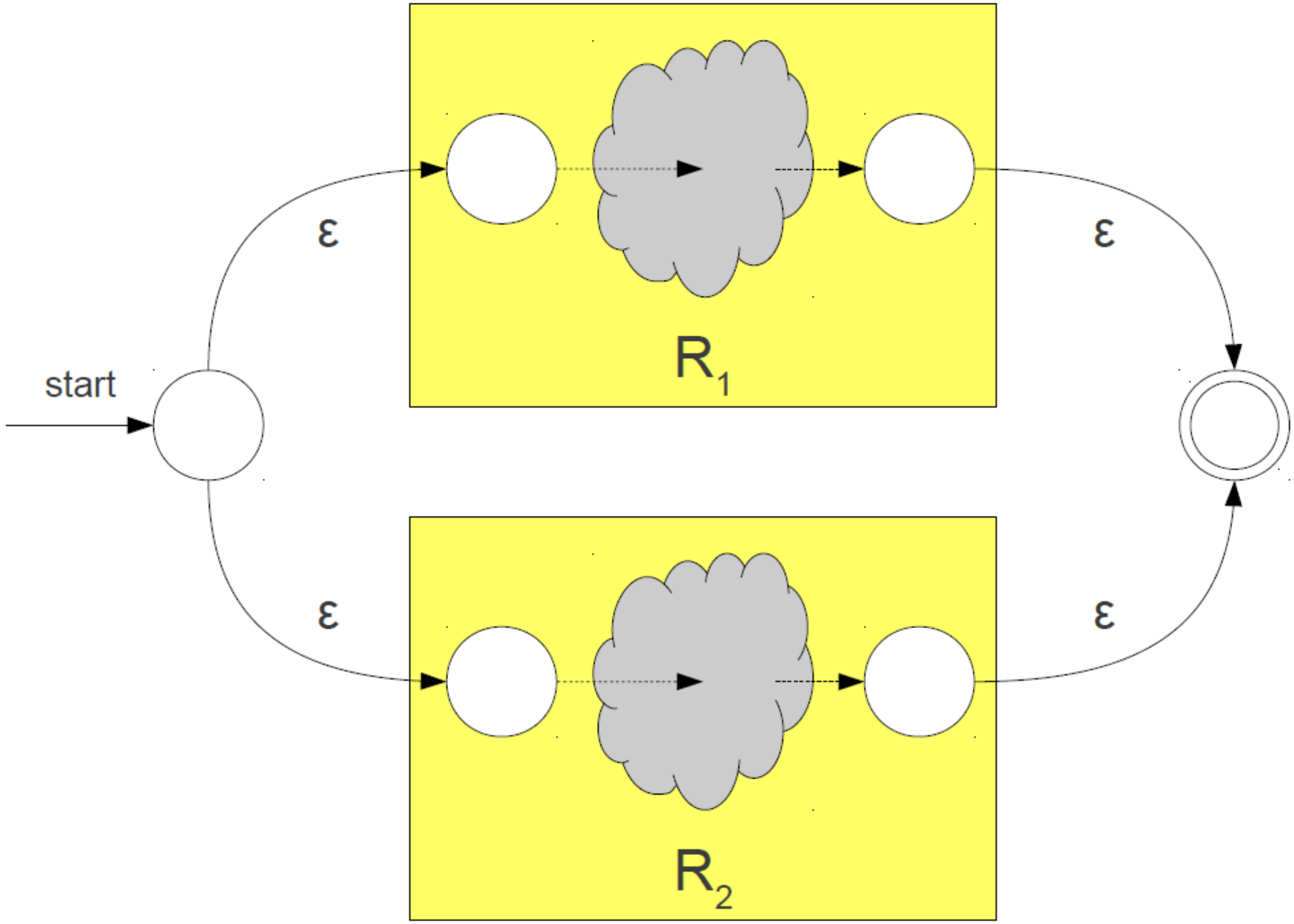


Construction for R_1 | R_2

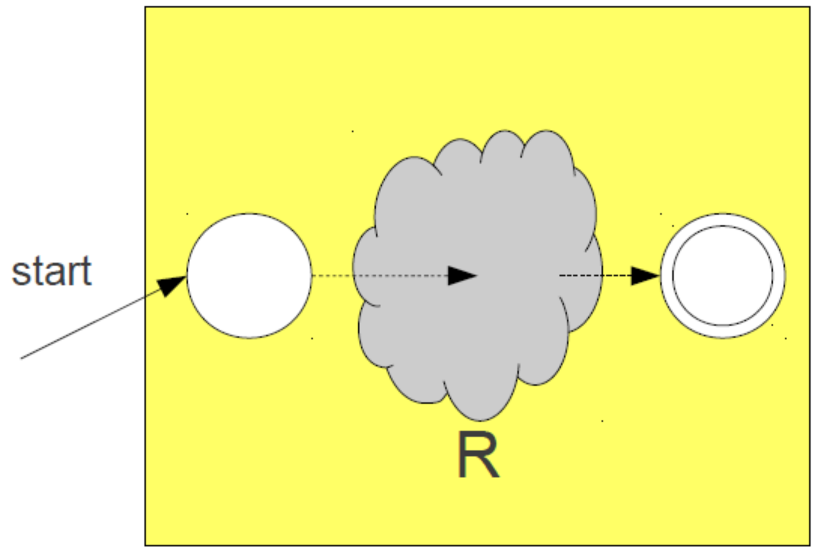




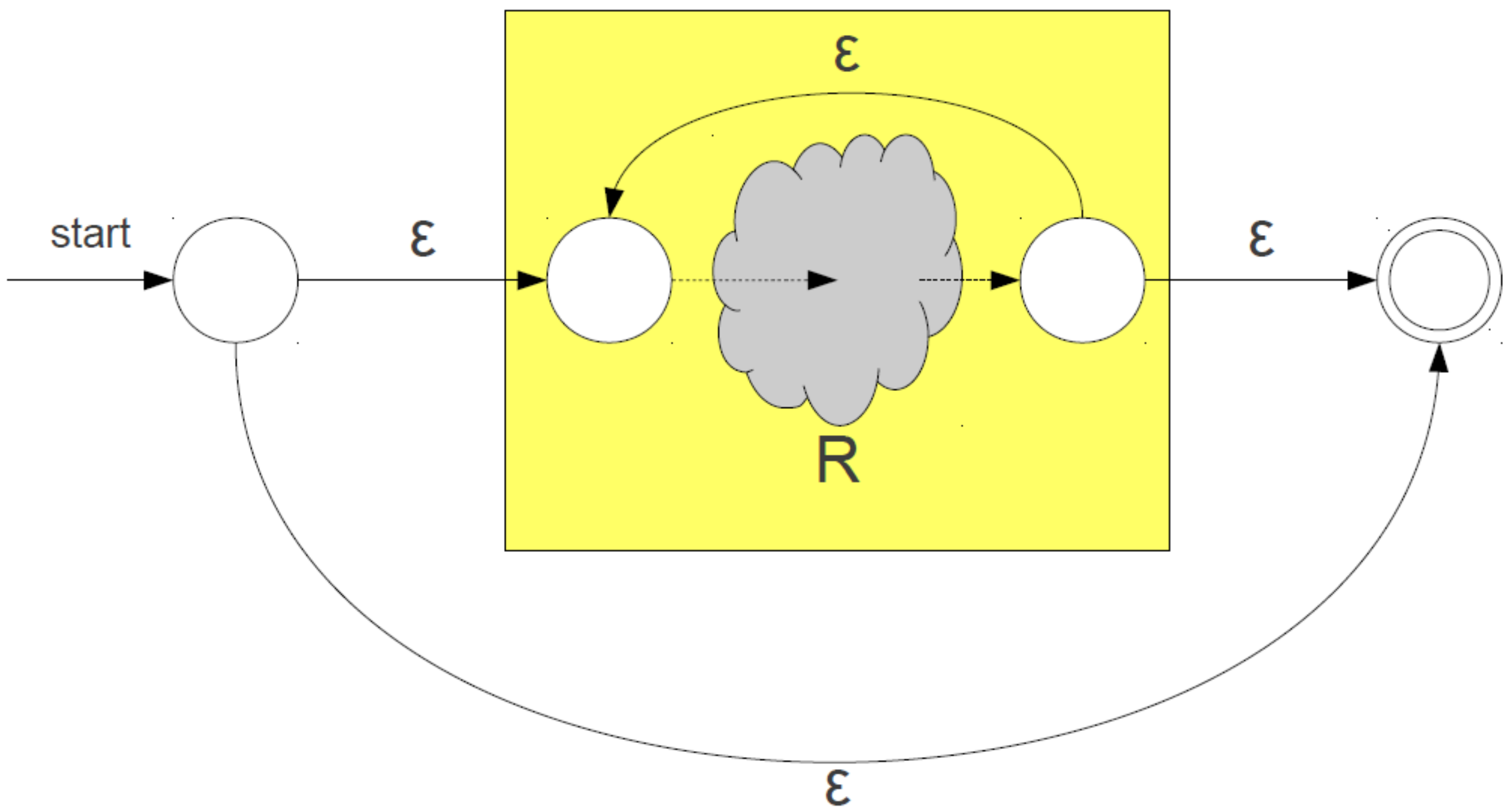
Construction for $R_1 \mid R_2$



Construction for R*



Construction for R*



Overall Result

McNaughton–Yamada–Thompson Algorithm

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