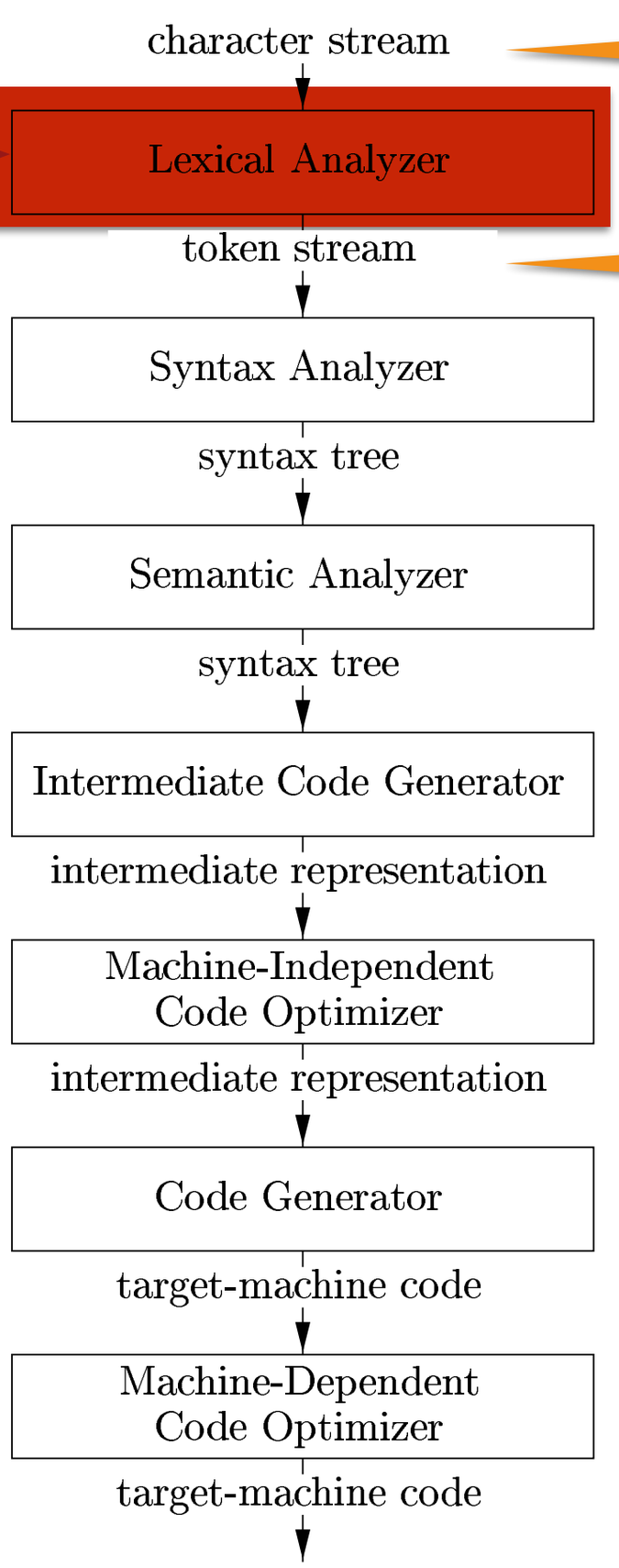
بسم الله الرحمن الرحیم

واژه‌یابی

درس کامپایلر

بهار ۱۳۹۴-۹۵

اینجا هستیم



position = initial + rate * 60

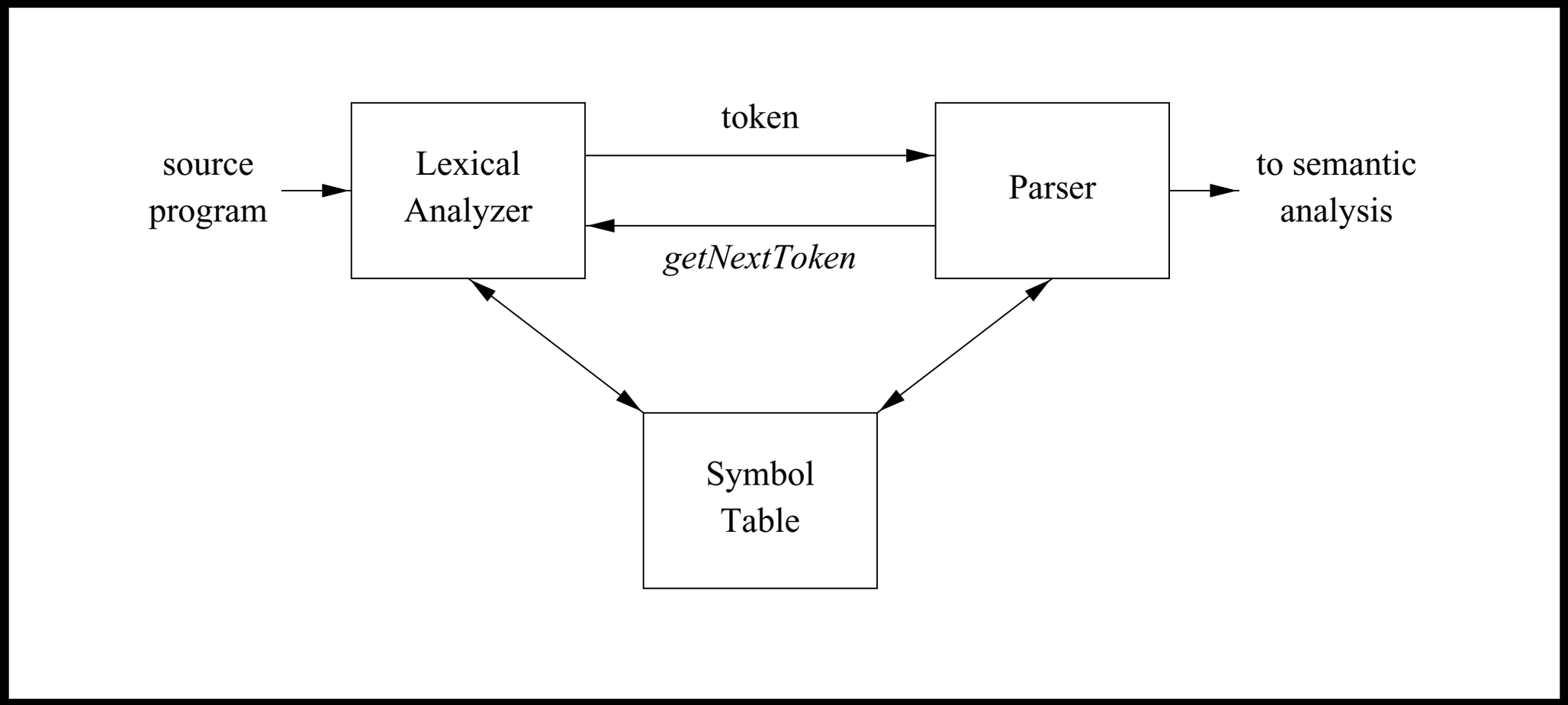
$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

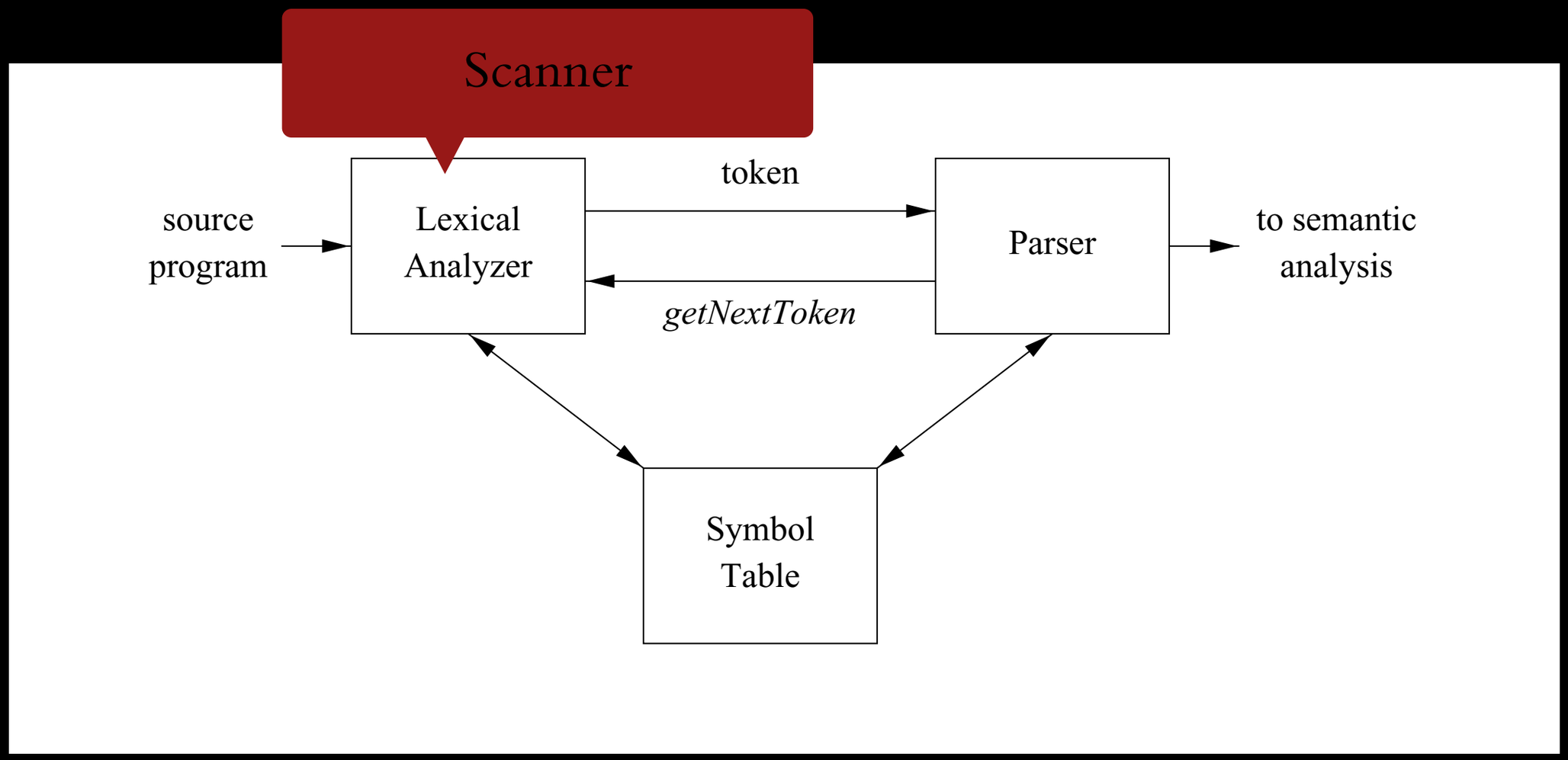
مقدمه

جایگاه واژه‌یاب در ساختار کامپایلر



- تولید نمونه به هنگام نیاز

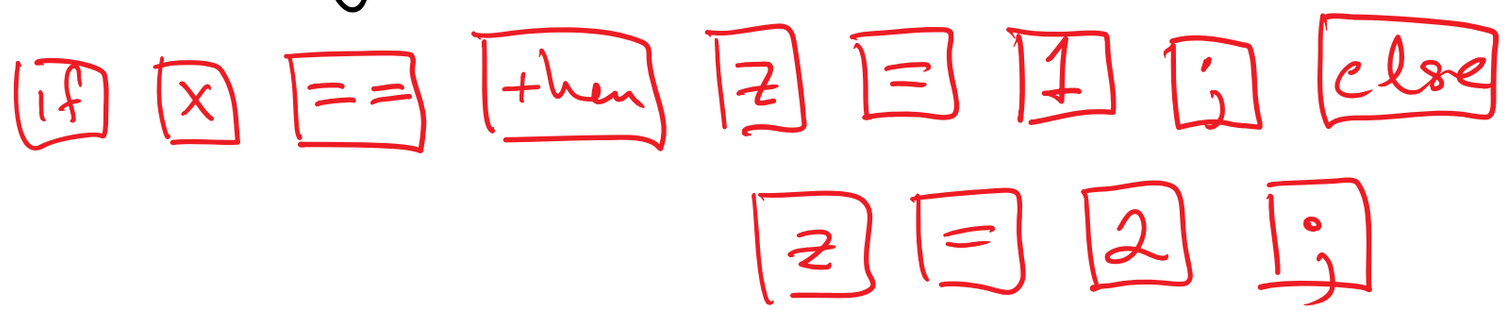
جایگاه واژه‌یاب در ساختار کامپایلر



- تولید نمونه به هنگام نیاز

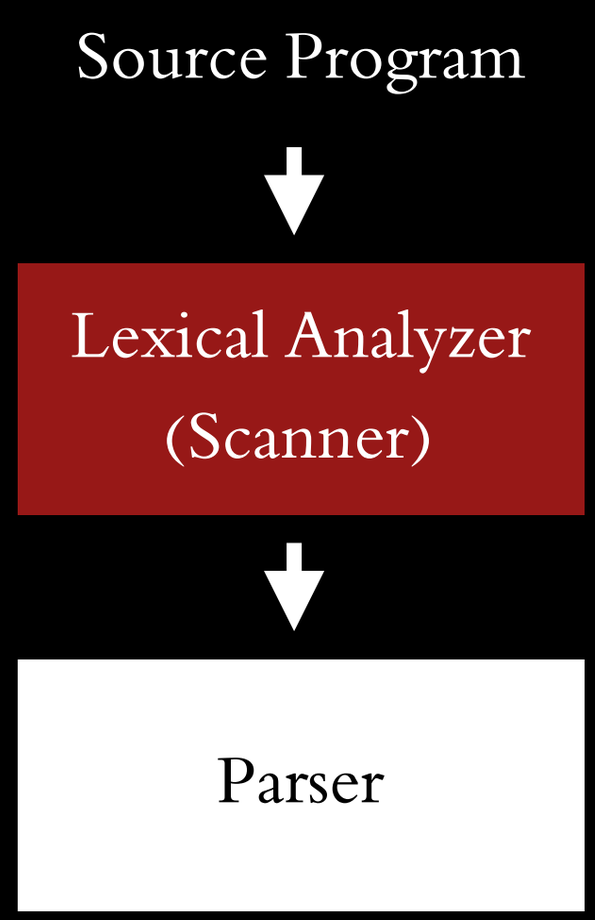
مثالی از خروجی واژه‌یاب

```
if x == y then z = 1 , else z = 2;
```

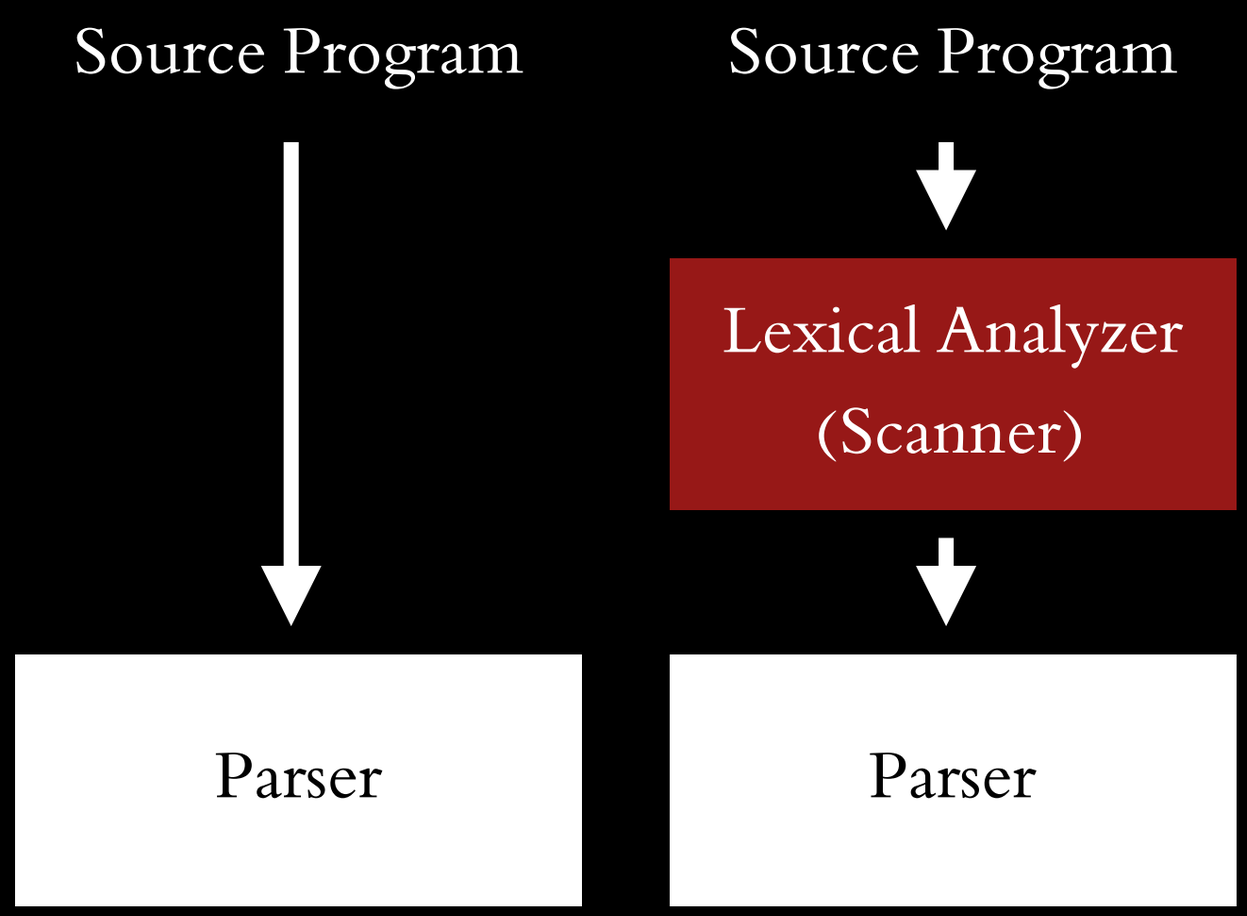


- اجرای مرحله به مرحله ...
- فاصله را رد می‌کند

چرا واژه‌یاب

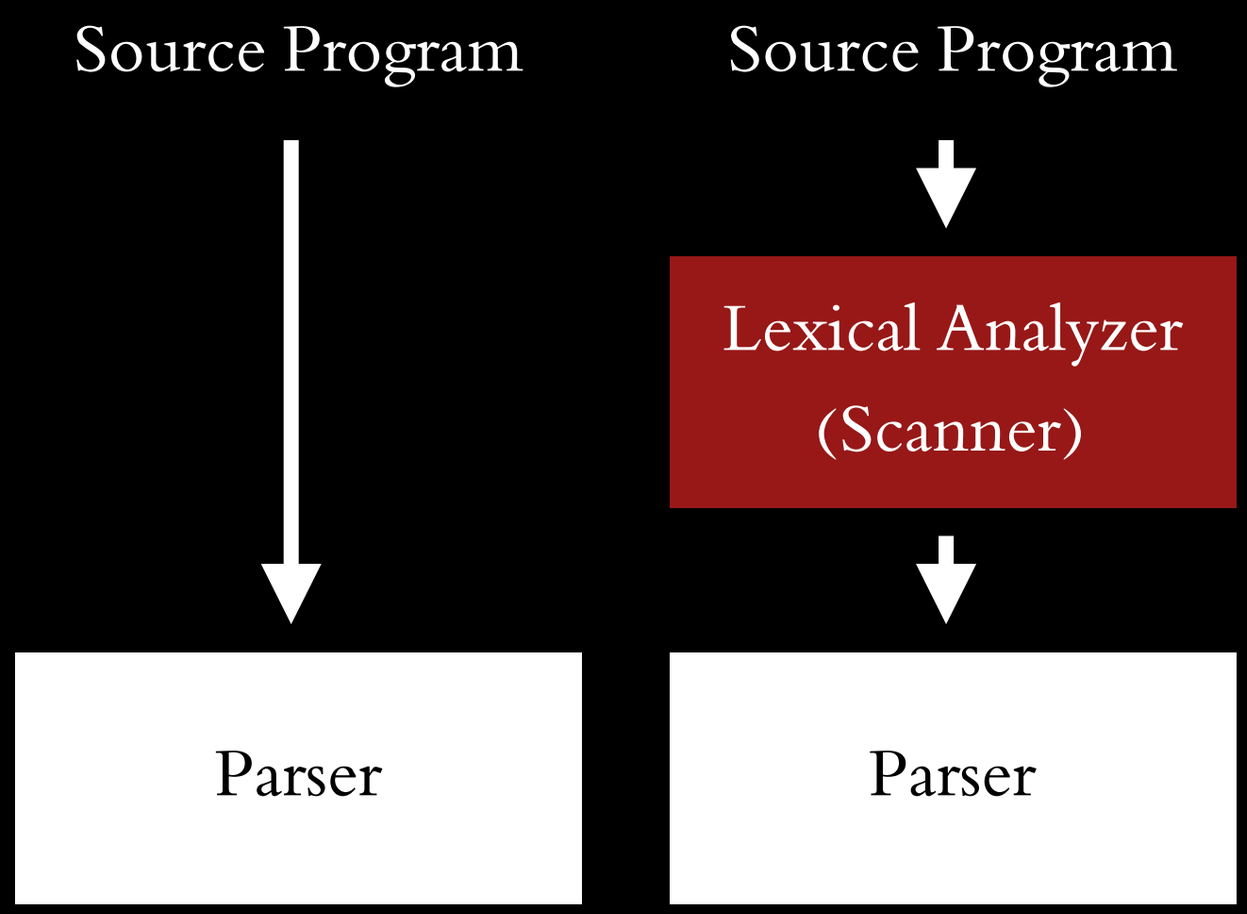


چرا واژه‌یاب


Source Program
Source Program
Lexical Analyzer
(Scanner)
Parser
Parser

چرا واژه‌یاب

- مزایای واژه‌یاب:
  - سریع‌تر
  - قسمت‌بندی
  - فکر نکردن به فاصله‌ها، یادداشت، در مرحله‌های بعدی



برخی تعریف‌ها

- نمونه (token): خروجی واژه‌یاب
- الگو (pattern): الگو
- تکواژه (lexeme): منتسب شده به الگو

برخی تعریف‌ها

- نمونه (token): خروجی واژه‌یاب
- الگو (pattern): الگو
- تکواژه (lexeme): منتسب شده به الگو

```
position = initial + rate * 60
```

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

برخی تعریف‌ها

تک‌واژه

position = initial + rate * 60

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$

- نمونه (token): خروجی واژه‌یاب
- الگو (pattern): الگو
- تکواژه (lexeme): منتسب شده به الگو

برخی تعریف‌ها

تک‌واژه

position = initial + rate * 60

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$

نمونه

- نمونه (token): خروجی واژه‌یاب
- الگو (pattern): الگو
- تکواژه (lexeme): منتسب شده به الگو

برخی تعریف‌ها

- نمونه (token): خروجی واژه‌یاب
- الگو (pattern): الگو
- تکواژه (lexeme): منتسب شده به الگو

تک‌واژه

position = initial + rate * 60

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$

نمونه

حروف متوالی

الگو

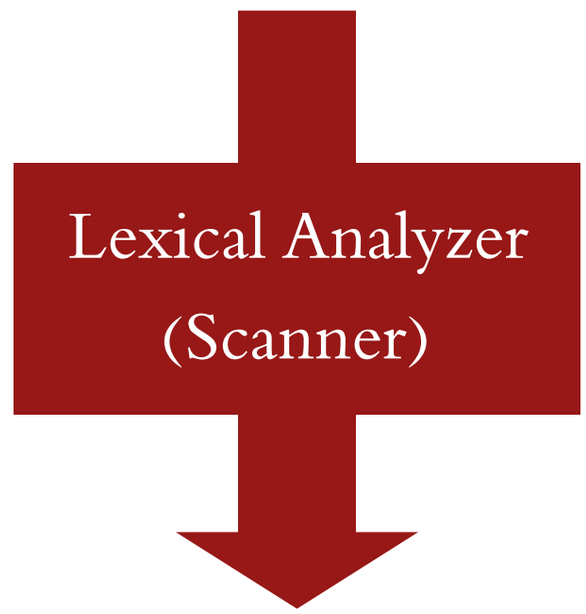
مثالی از برخی نمونه‌ها (token)

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"`

توصیف تک‌واژه‌ها
یا الگوها

چگونه یک برنامه بنویسیم؟

```
position = initial + rate * 60
```



$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60 \rangle$$

یک راه حل: توصیف تک‌واژه‌ها

`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

FIGURE 2.2. Regular expressions for some tokens.

زبان‌های منظم

Specifying Lexical Structure Using Regular Expressions

- Have some alphabet Σ = set of letters
- Regular expressions are built from:
  - ε - empty string

    Any letter from alphabet Σ
  - $r_1 r_2$ – regular expression r_1 followed by r_2 (sequence)
  - $r_1 | r_2$ – either regular expression r_1 or r_2 (choice)
  - r^* - iterated sequence and choice ε | r | rr | …
  - Parentheses to indicate grouping/precedence

Concept of Regular Expression Generating a String

Rewrite regular expression until have only a sequence of letters (string) left

General Rules

1) $r_1 | r_2 \rightarrow r_1$
2) $r_1 | r_2 \rightarrow r_2$
3) $r^* \rightarrow rr^*$
4) $r^* \rightarrow \varepsilon$

Example

(0 | 1)*.(0|1)*
(0 | 1)(0 | 1)*.(0|1)*
1(0|1)*.(0|1)*
1.(0|1)*
1.(0|1)(0|1)*
1.(0|1)
1.0

Nondeterminism in Generation

- Rewriting is similar to equational reasoning
- But different rule applications may yield different final results

Example 1

(0|1)*.(0|1)*
(0|1)(0|1)*.(0|1)*
1(0|1)*.(0|1)*
1.(0|1)*
1.(0|1)(0|1)*
1.(0|1)
1.0

Example 2

(0|1)*.(0|1)*
(0|1)(0|1)*.(0|1)*
0(0|1)*.(0|1)*
0.(0|1)*
0.(0|1)(0|1)*
0.(0|1)
0.1

Examples of Languages and Regular Expressions

- Σ = { 0, 1, . }
  - (0|1)*.(0|1)* - Binary floating point numbers
  - (00)* - even-length all-zero strings

    1*(01*01*)* - strings with even number of zeros
- Σ = { a,b,c, 0, 1, 2 }
  - (a|b|c)(a|b|c|0|1|2)* - alphanumeric identifiers
  - (0|1|2)* - trinary numbers

Alternate Abstraction
Finite-State Automata

- Alphabet Σ
- Set of states with initial and accept states
- Transitions between states, labeled with letters

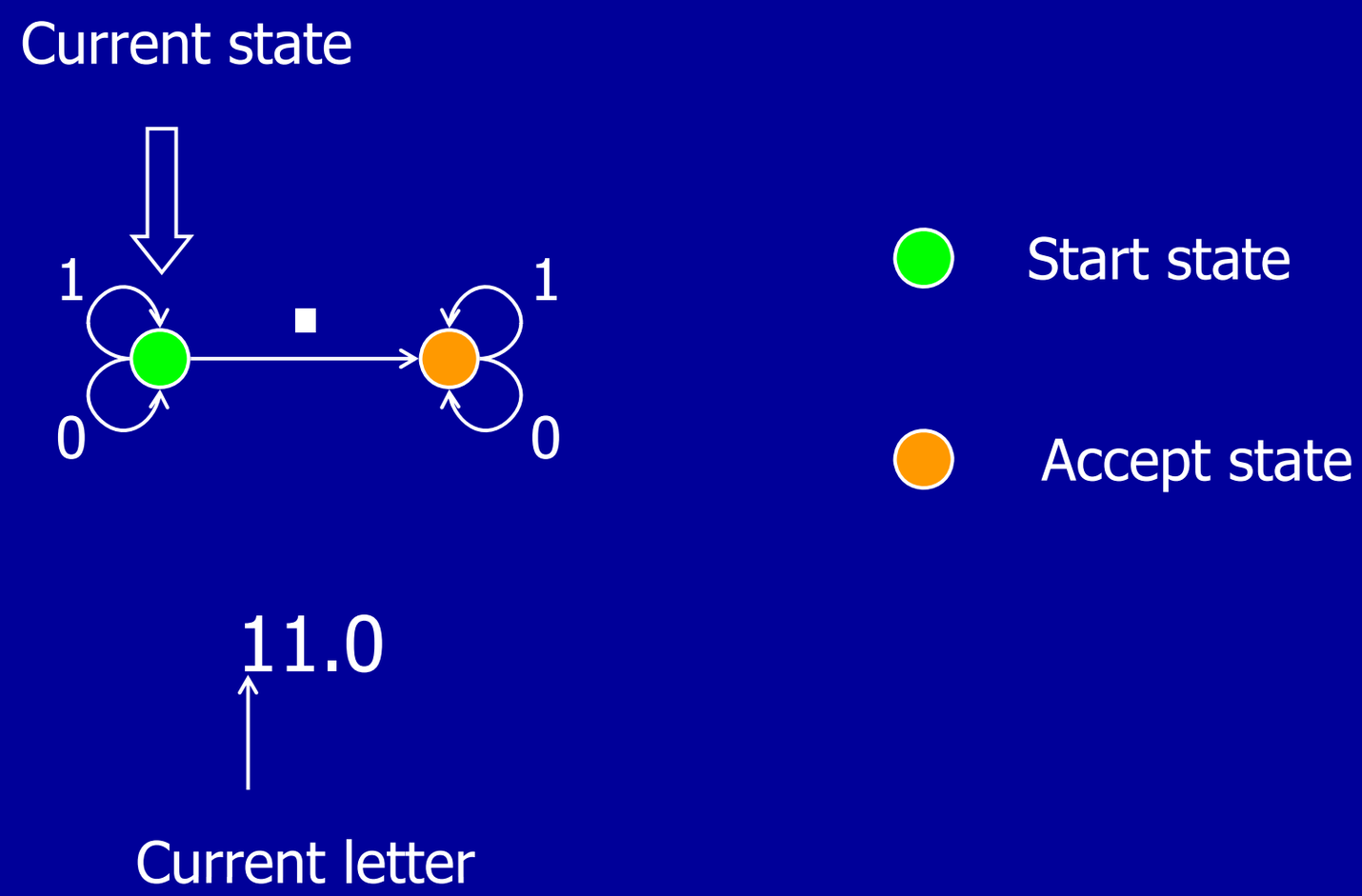
(0|1)*.(0|1)*

Automaton Accepting String

Conceptually, run string through automaton

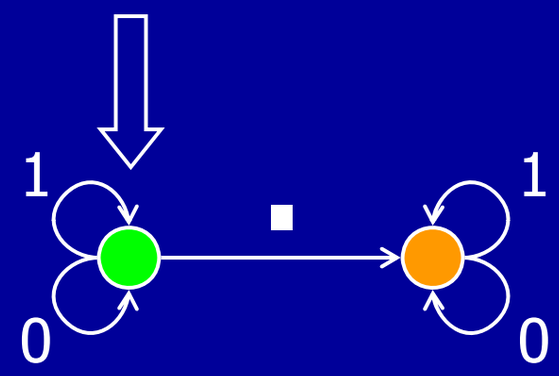
- Have current state and current letter in string
- Start with start state and first letter in string
- At each step, match current letter against a transition whose label is same as letter
- Continue until reach end of string or match fails
- If end in accept state, automaton accepts string
- Language of automaton is set of strings it accepts

Example


Current state
1
0
.
1
0
Start state
Accept state
11.0
Current letter

Example

Current state



Start state

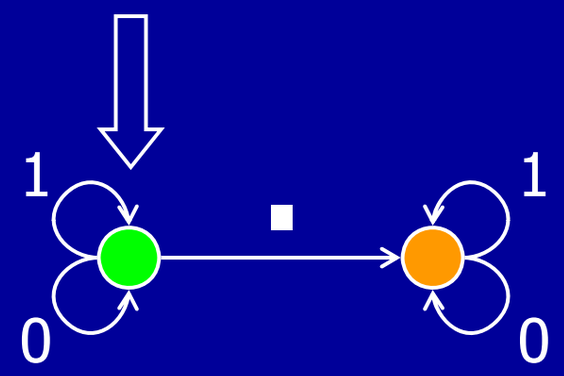
Accept state

11.0

Current letter

Example

Current state



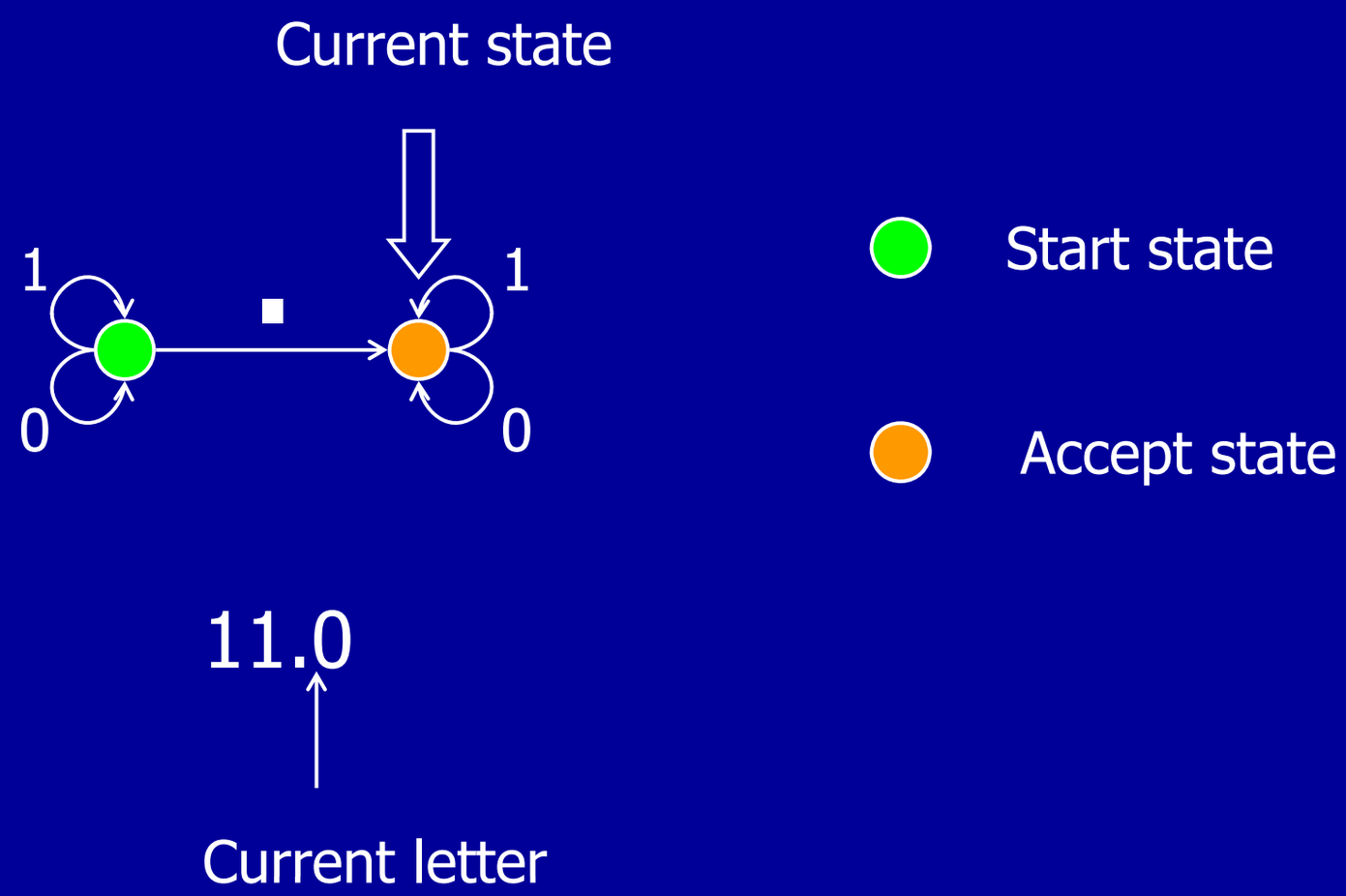
Start state

Accept state

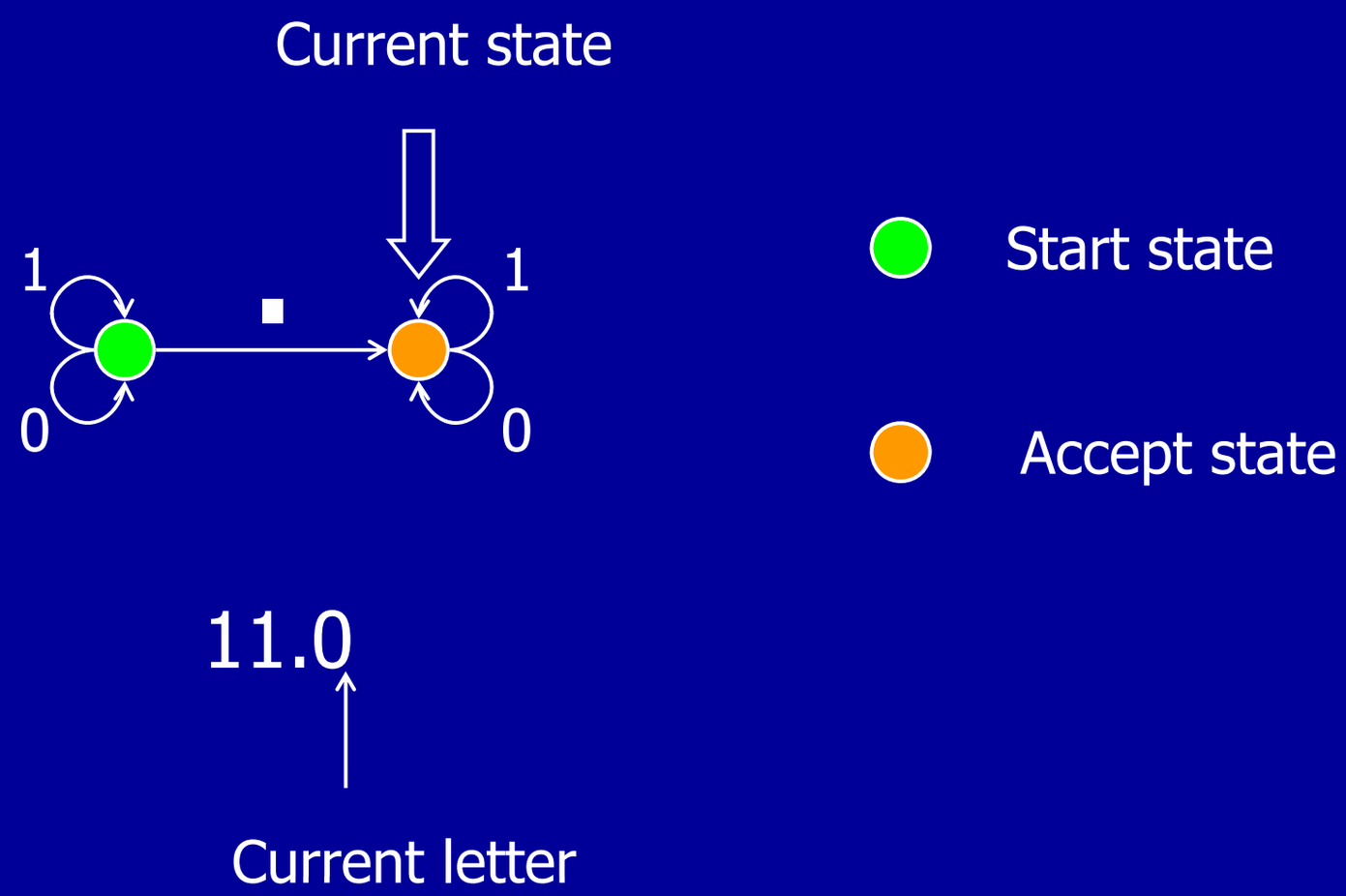
11.0

Current letter

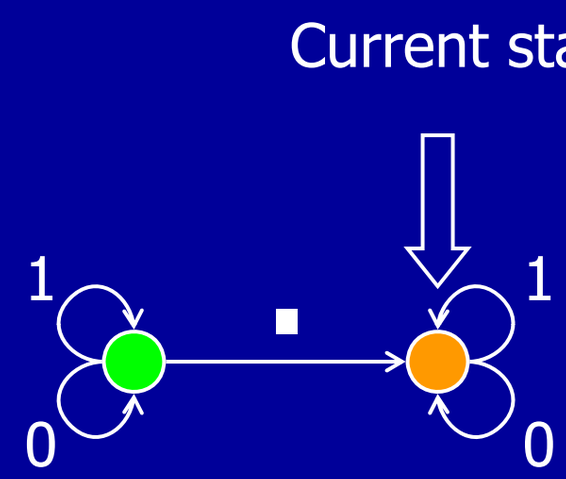
Example


Current state
1
0
.
1
0
Start state
Accept state
11.0
Current letter

Example


Current state
1
0
.
1
0
Start state
Accept state
11.0
Current letter

Example



Start state

Accept state

11.0

Current letter

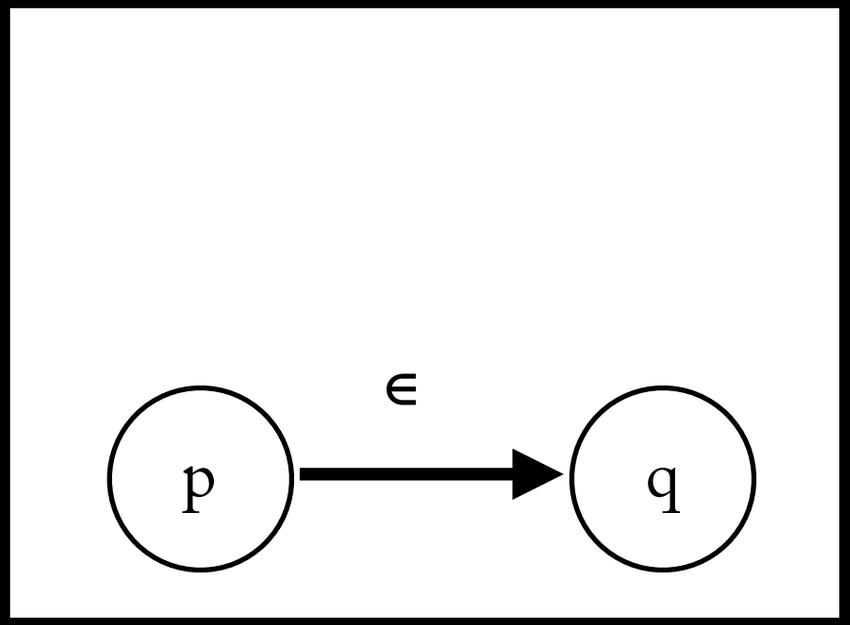
String is accepted!

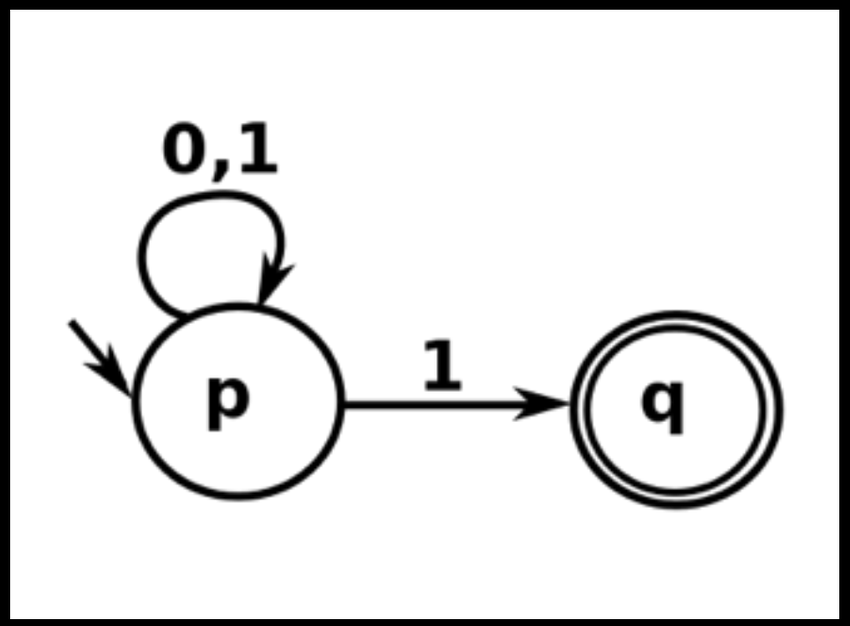
Generative Versus Recognition

- Regular expressions give you a way to generate all strings in language
- Automata give you a way to recognize if a specific string is in language
  - Philosophically very different
  - Theoretically equivalent (for regular expressions and automata)
- Standard approach
  - Use regular expressions when define language
  - Translated automatically into automata for implementation

تبدیل زبان منظم به اتوماتا

تبدیل به اتوماتای غیرقطعی





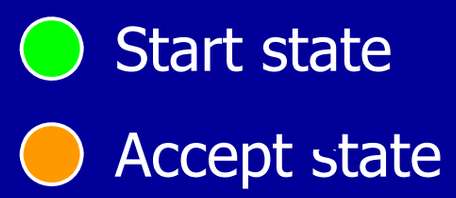
- یال ∈
- یال‌های هم‌حرف

From Regular Expressions to Automata

- Construction by structural induction
- Given an arbitrary regular expression r
- Assume we can convert r to an automaton with
  - One start state
  - One accept state

  Show how to convert all constructors to deliver an automaton with
  - One start state
  - One accept state

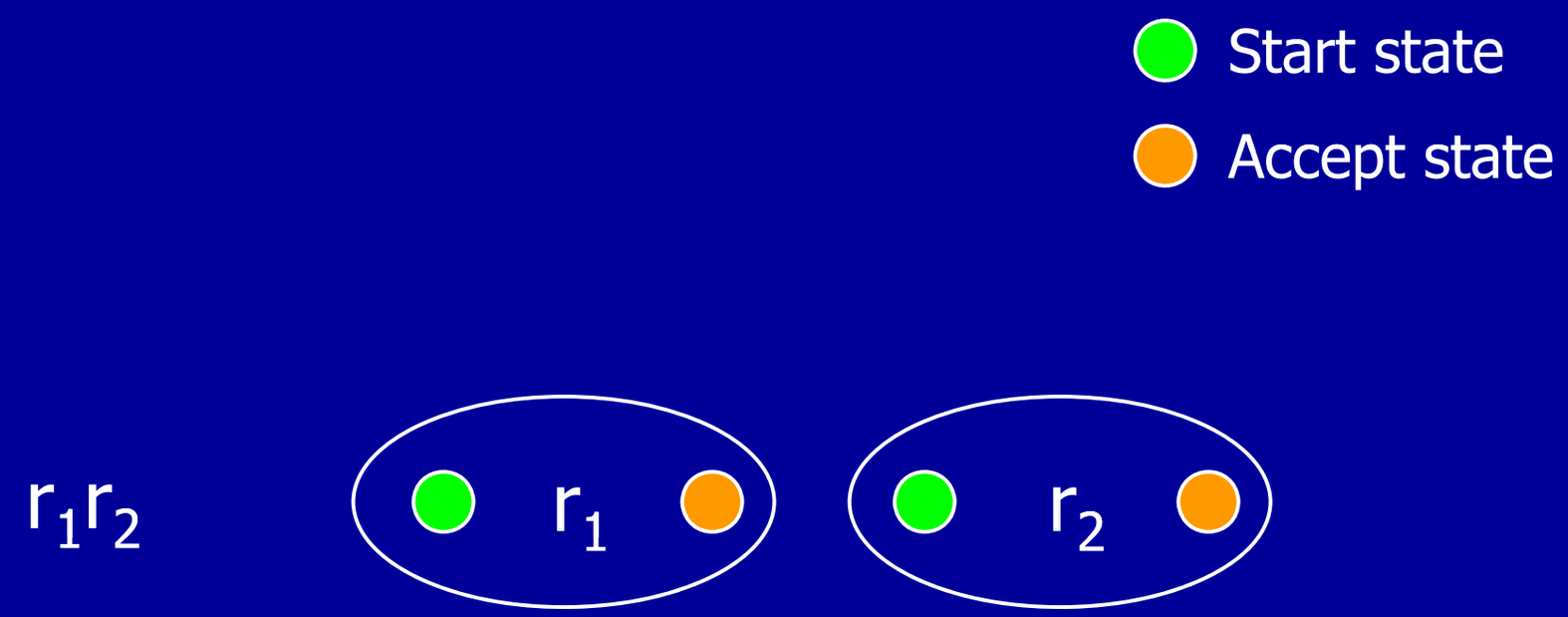
Basic Constructs



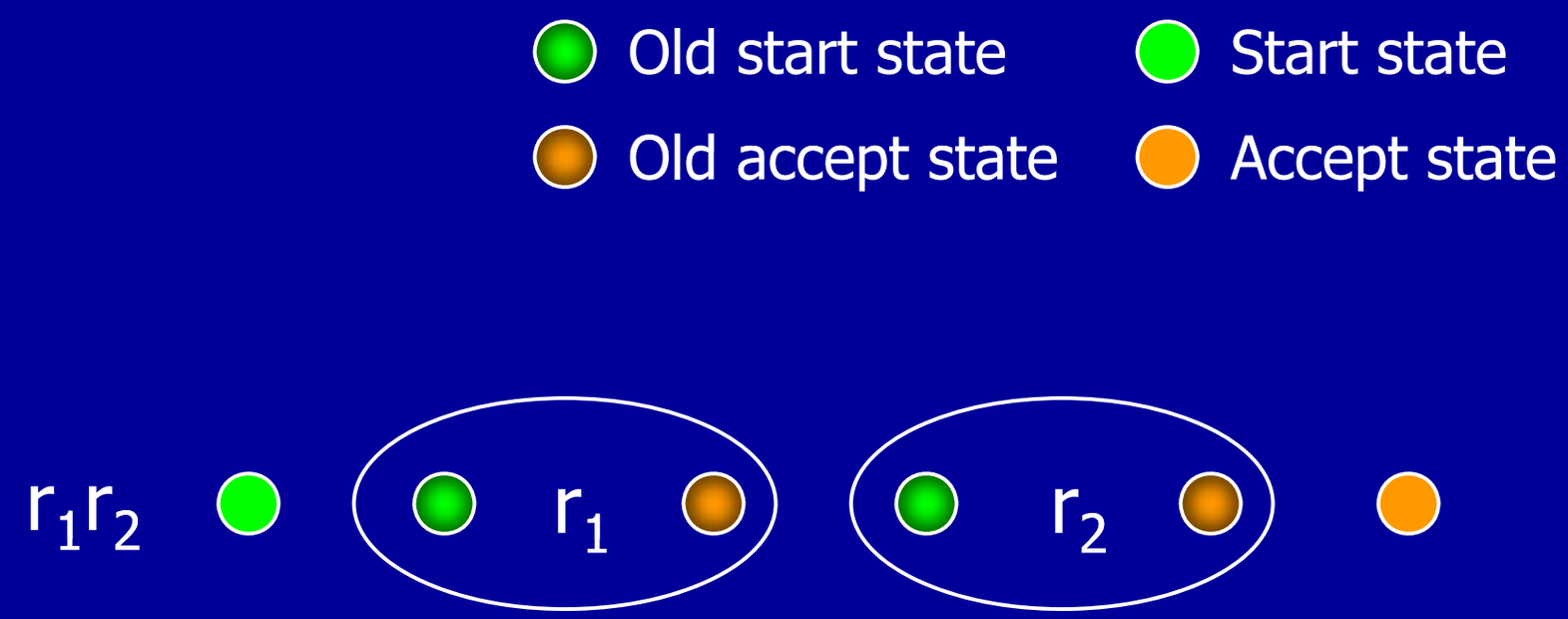
ε

$a \in \Sigma$

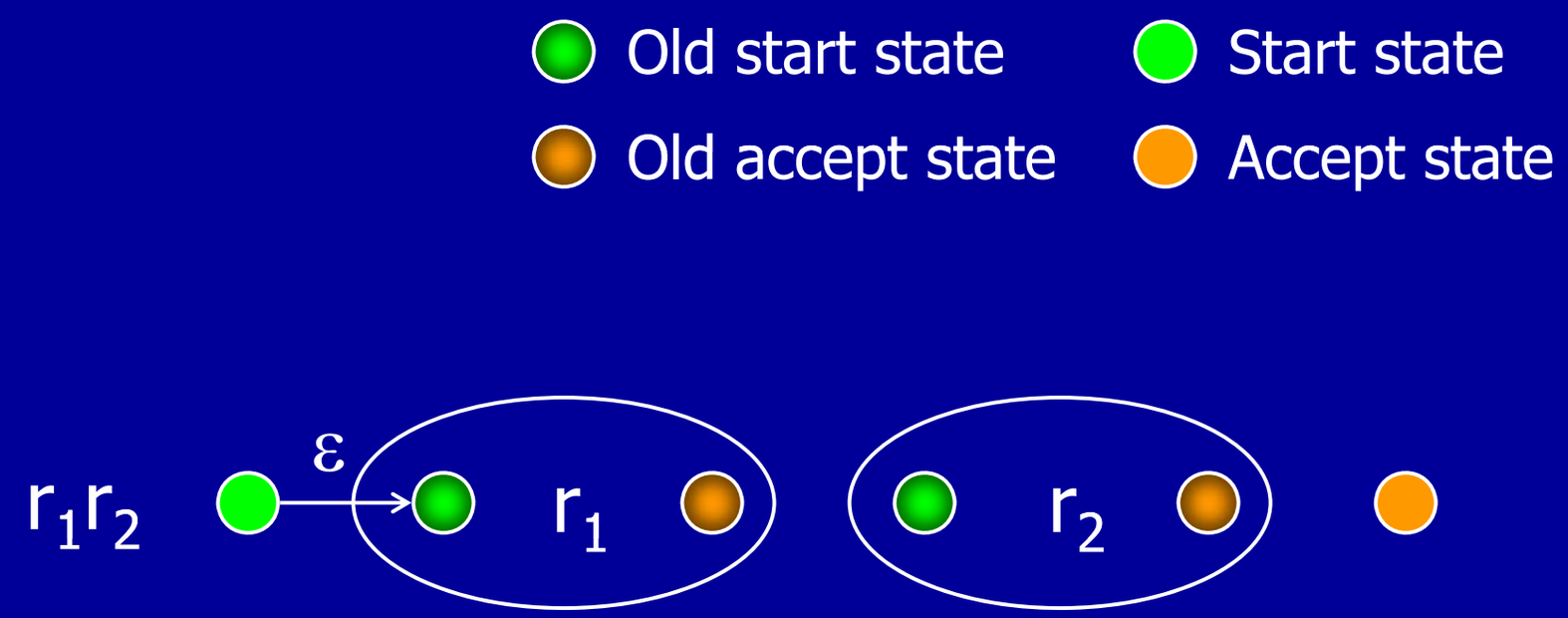
Sequence



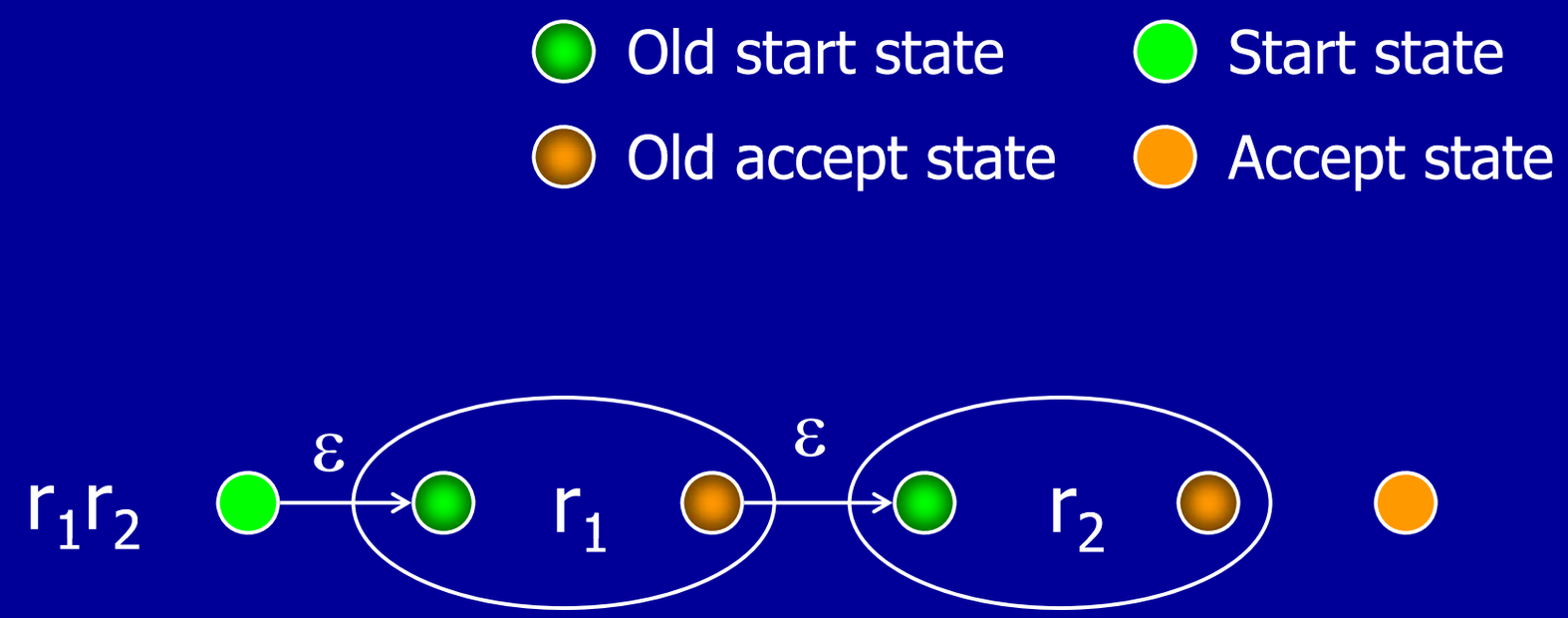
Sequence


Old start state
Start state
Old accept state
Accept state
r_1r_2
r_1
r_2

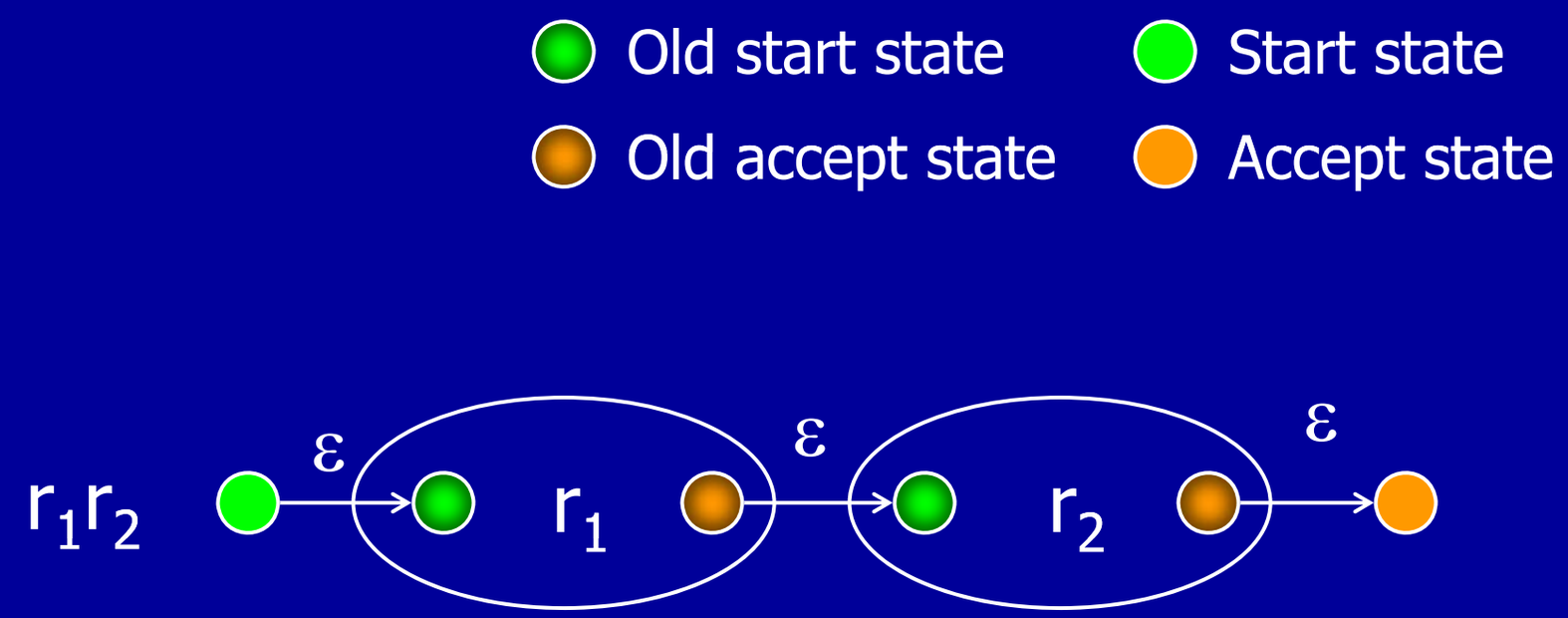
Sequence


Old start state
Start state
Old accept state
Accept state
r_1r_2
ε
r_1
r_2

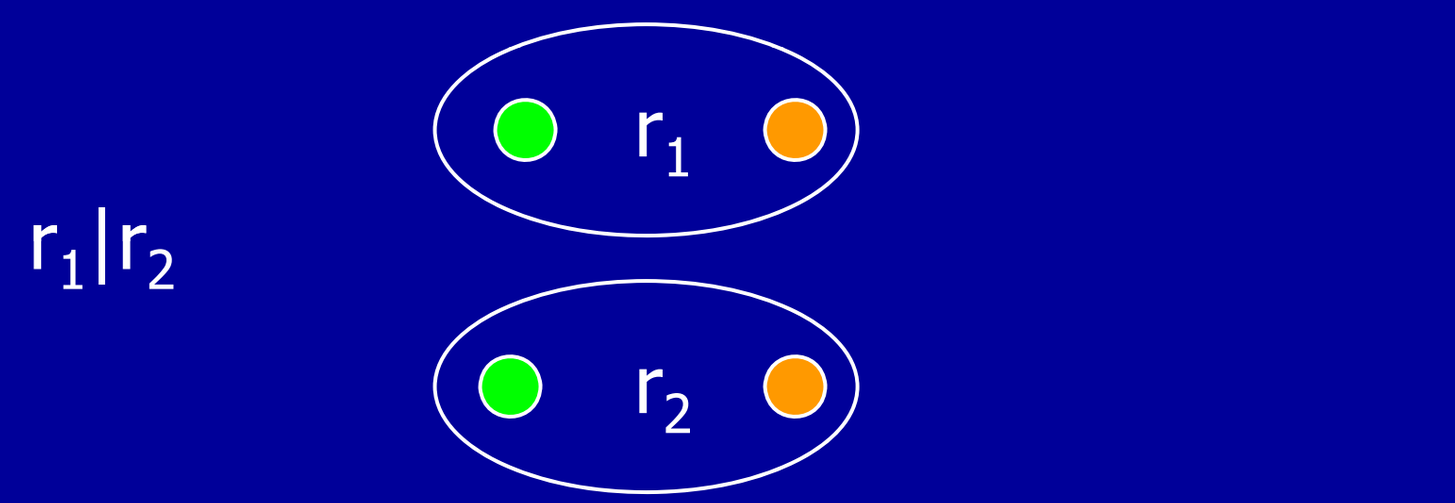
Sequence


Old start state
Start state
Old accept state
Accept state
r_1r_2
ε
r_1
ε
r_2

Sequence

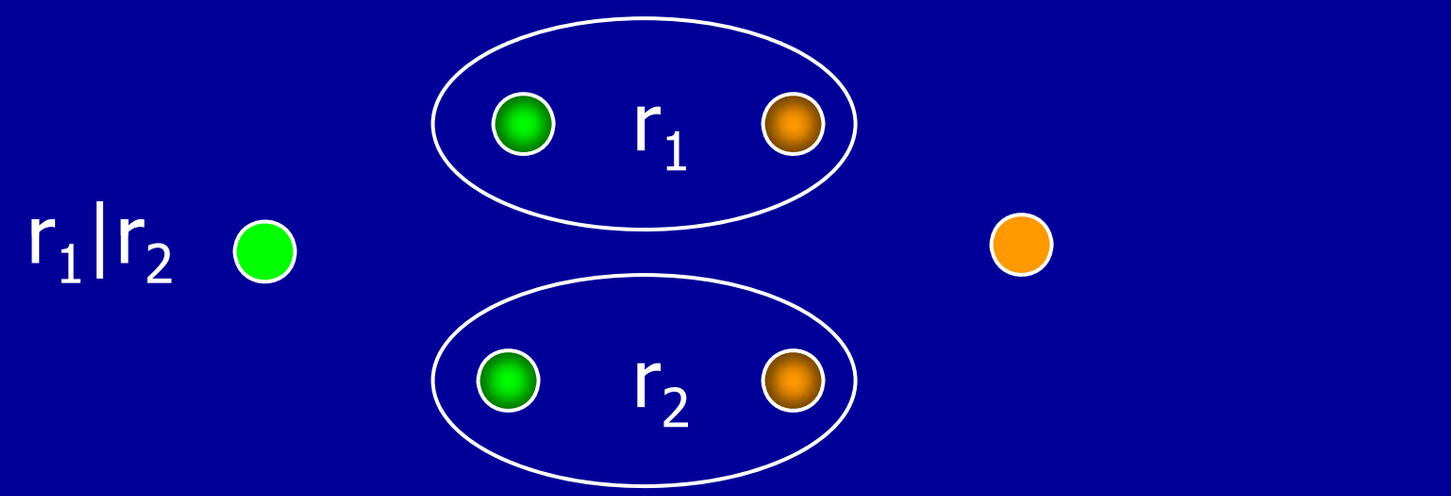

Old start state
Start state
Old accept state
Accept state
r_1r_2
ε
r_1
ε
r_2
ε

Choice

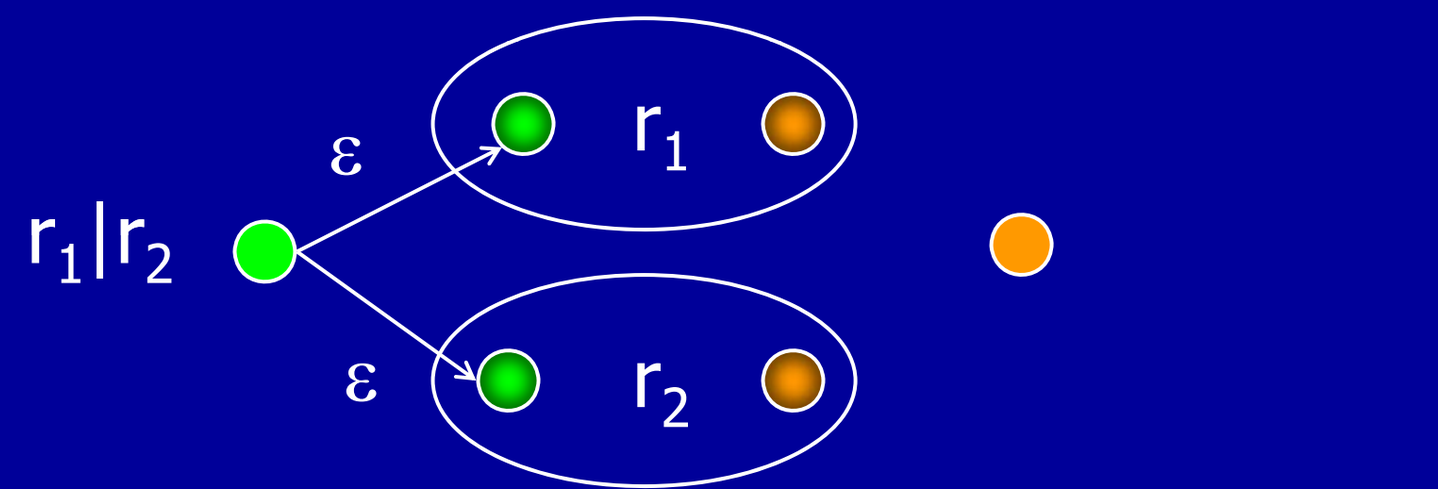


$r_1|r_2$

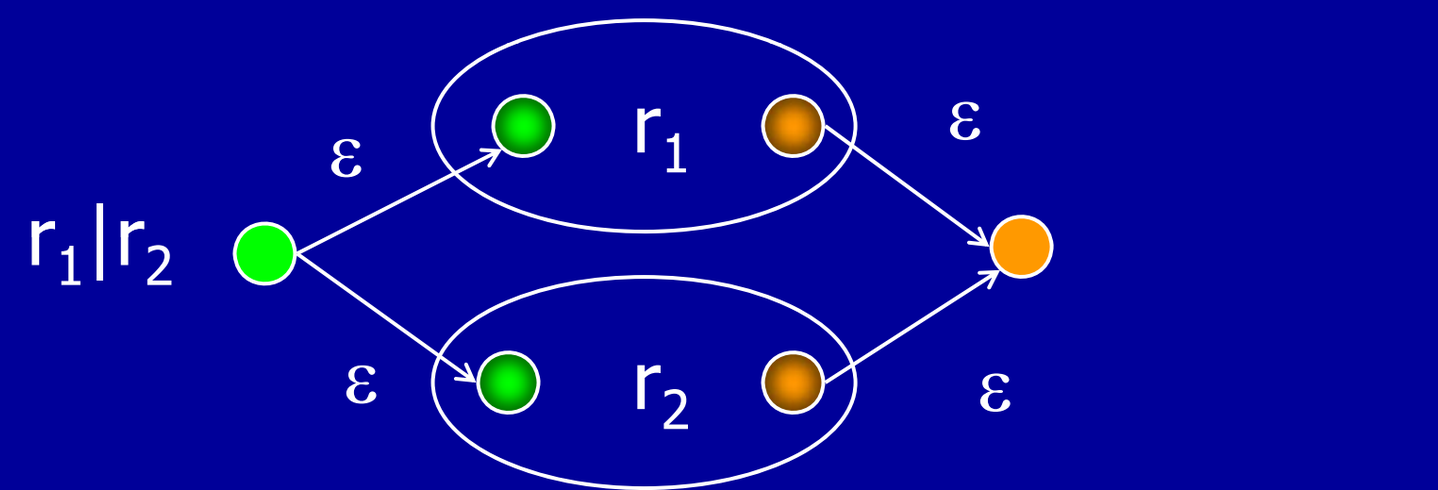
Choice


Old start state
Start state
Old accept state
Accept state
r_1
$r_1|r_2$
r_2

Choice

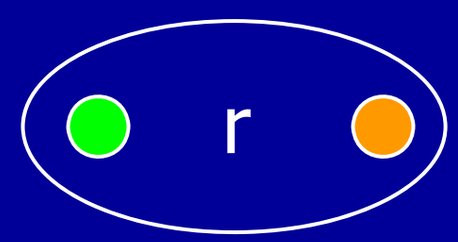

Old start state
Start state
Old accept state
Accept state
$r_1|r_2$
ε
r_1
ε
r_2

Choice

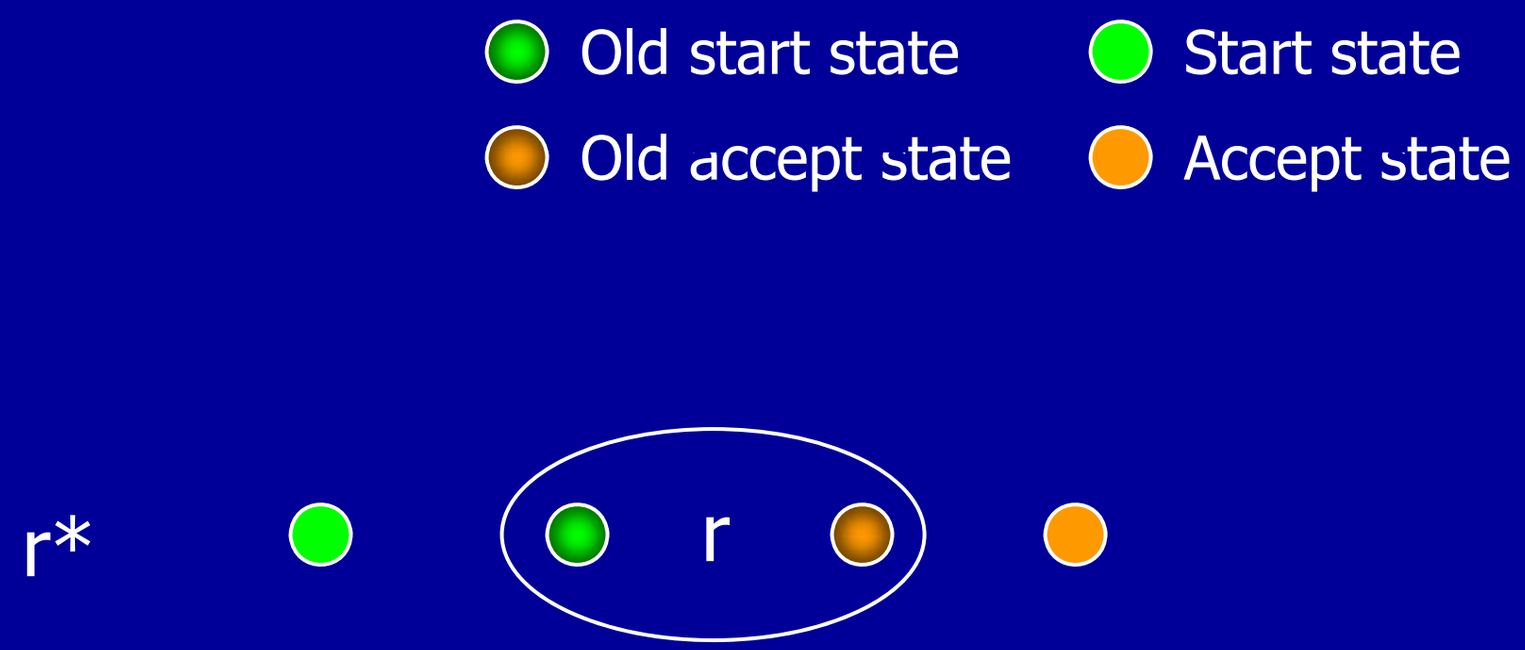

Old start state
Start state
Old accept state
Accept state
$r_1|r_2$
ε
r_1
ε
ε
r_2
ε

Kleene Star

r*



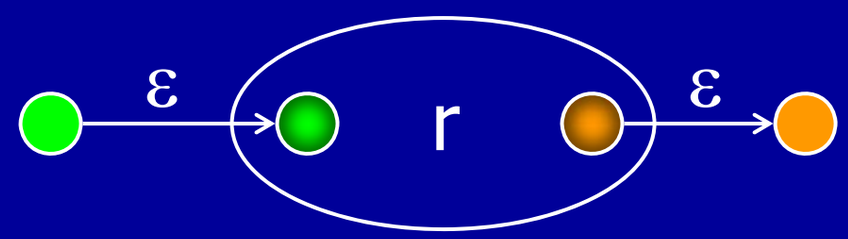
Kleene Star


Old start state
Start state
Old accept state
Accept state
r*
r

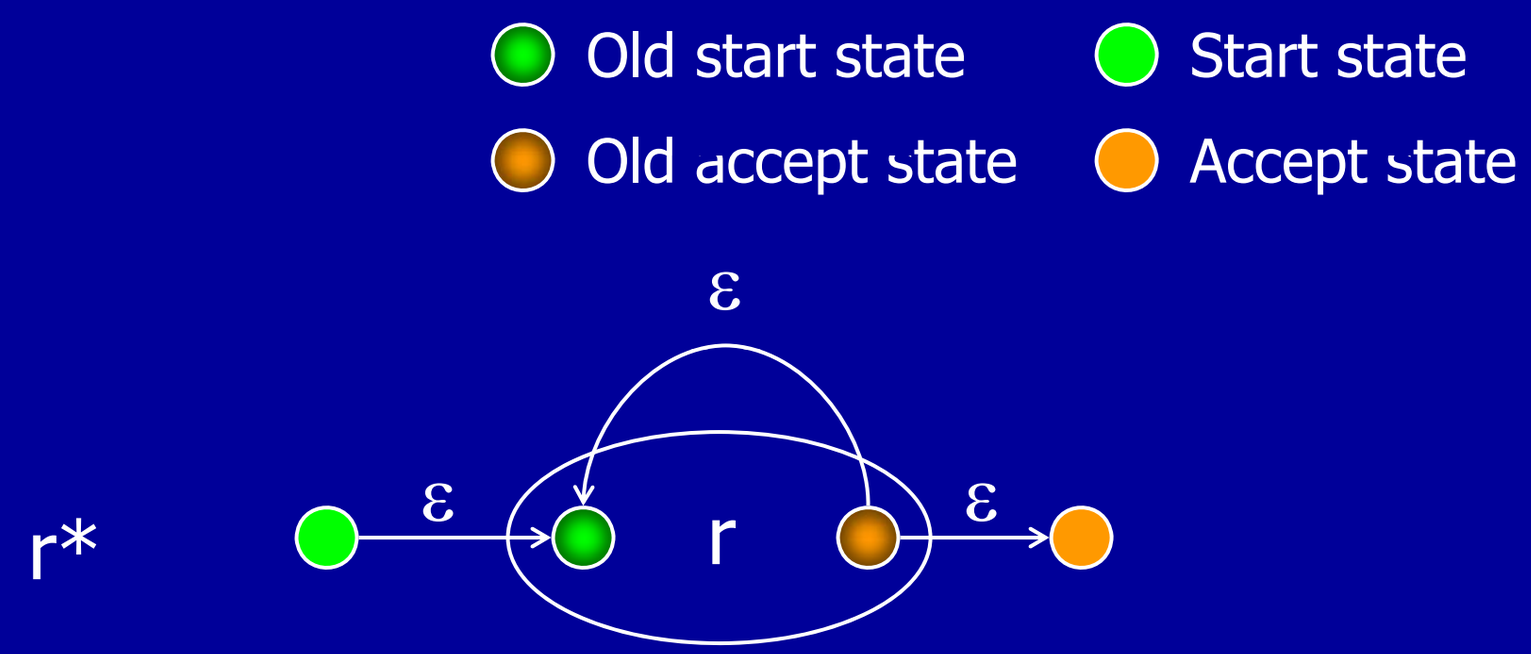
Kleene Star

- Old start state
- Start state
- Old accept state
- Accept state

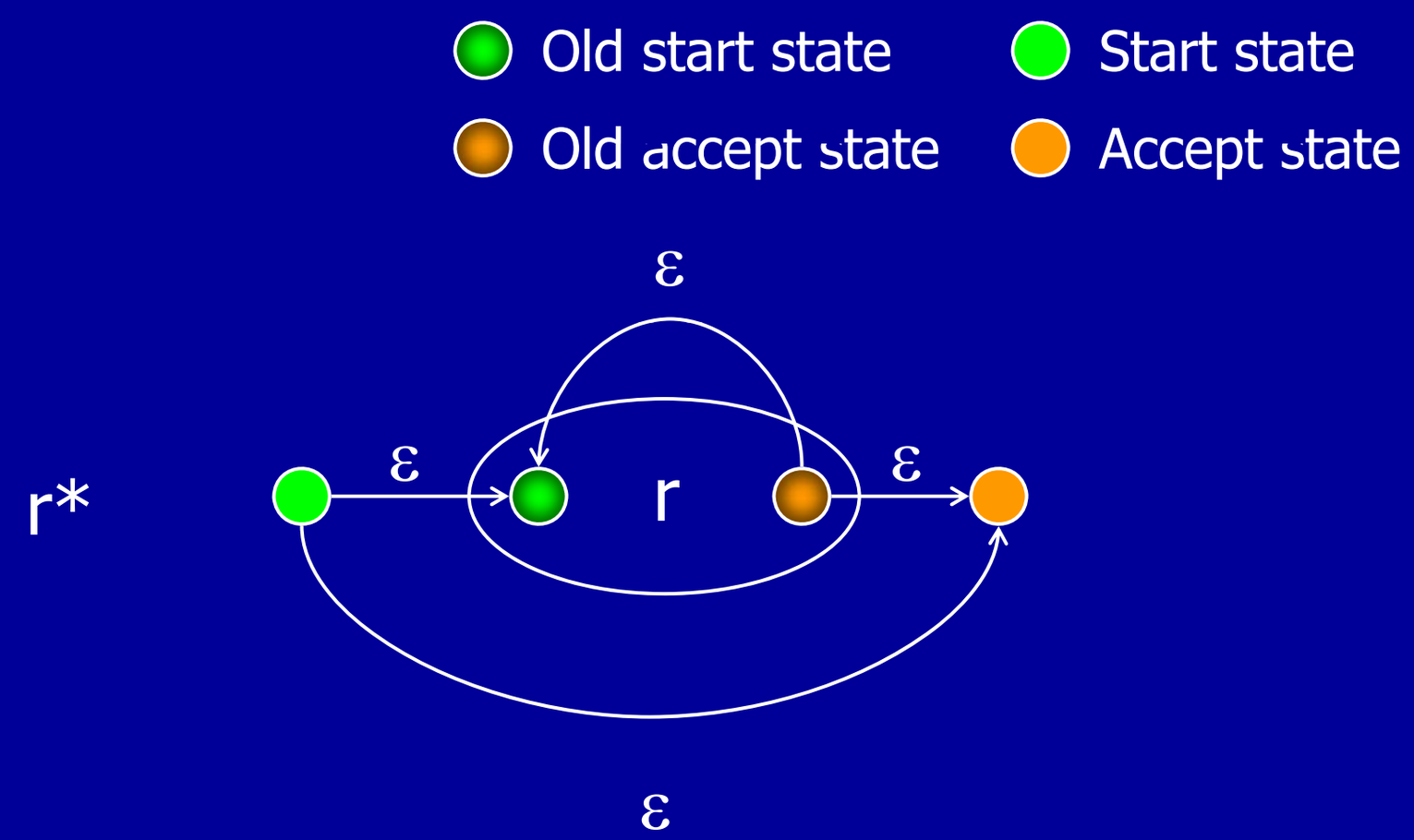
r*



Kleene Star

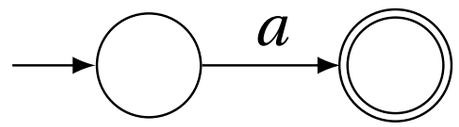

Old start state
Start state
Old accept state
Accept state
ε
r*
ε
r
ε

Kleene Star


Old start state
Start state
Old accept state
Accept state
ε
r*
ε
r
ε
ε

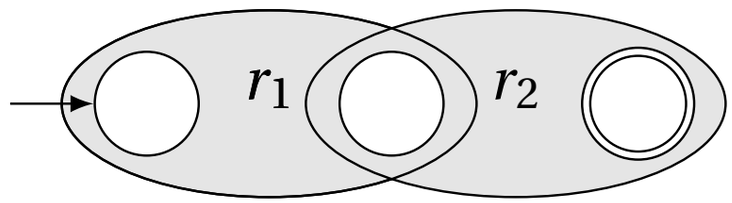
Translating REs into NFAs

a



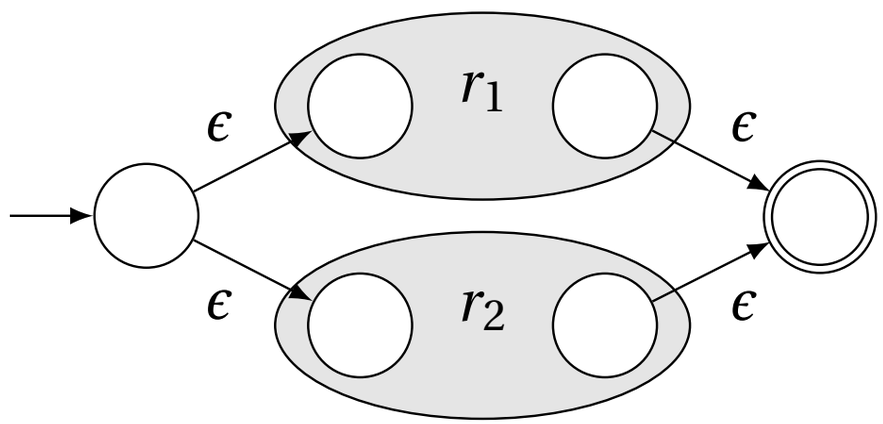
Symbol

$r_1 r_2$



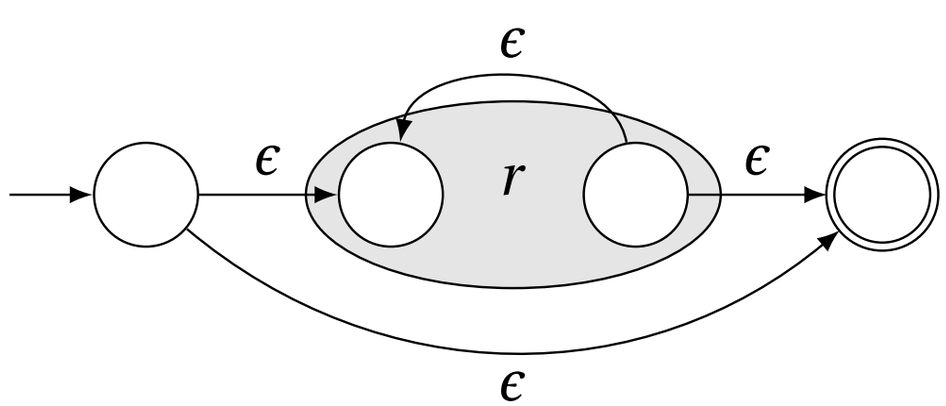
Sequence

$r_1 \mid r_2$



Choice

$(r)^*$



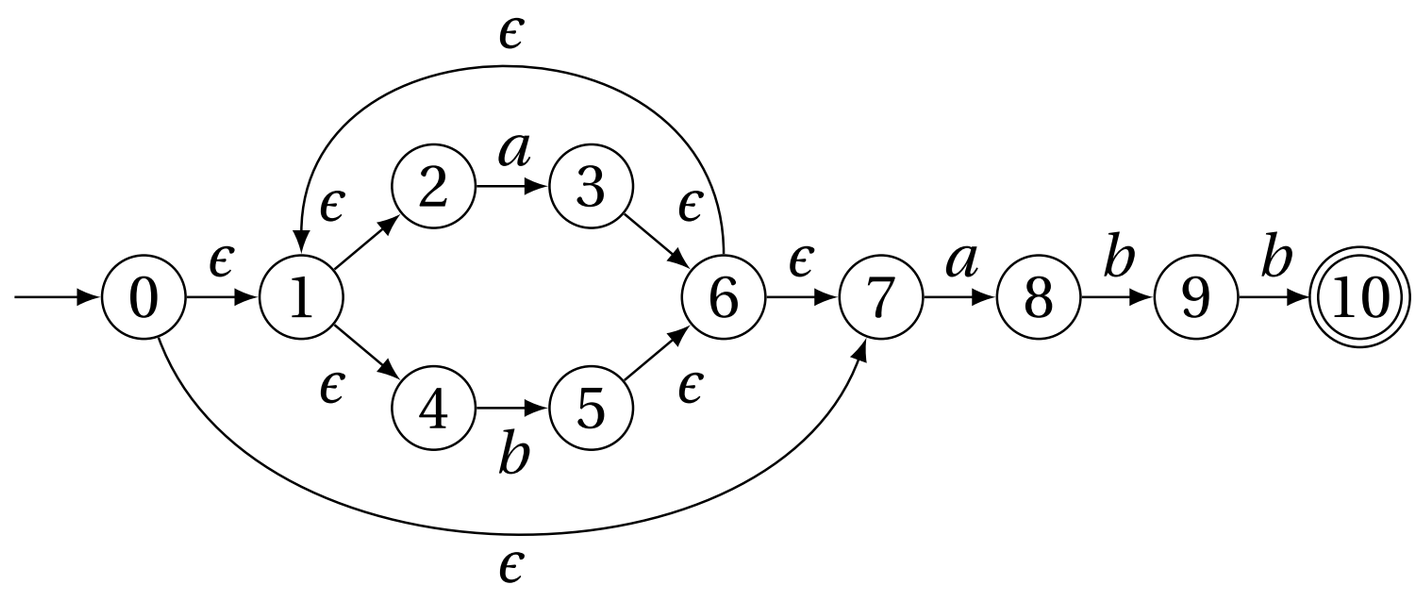
Kleene Closure

مثال تبدیل عبارت منظم به NFA

عبارت منظم: (a | b)*abb

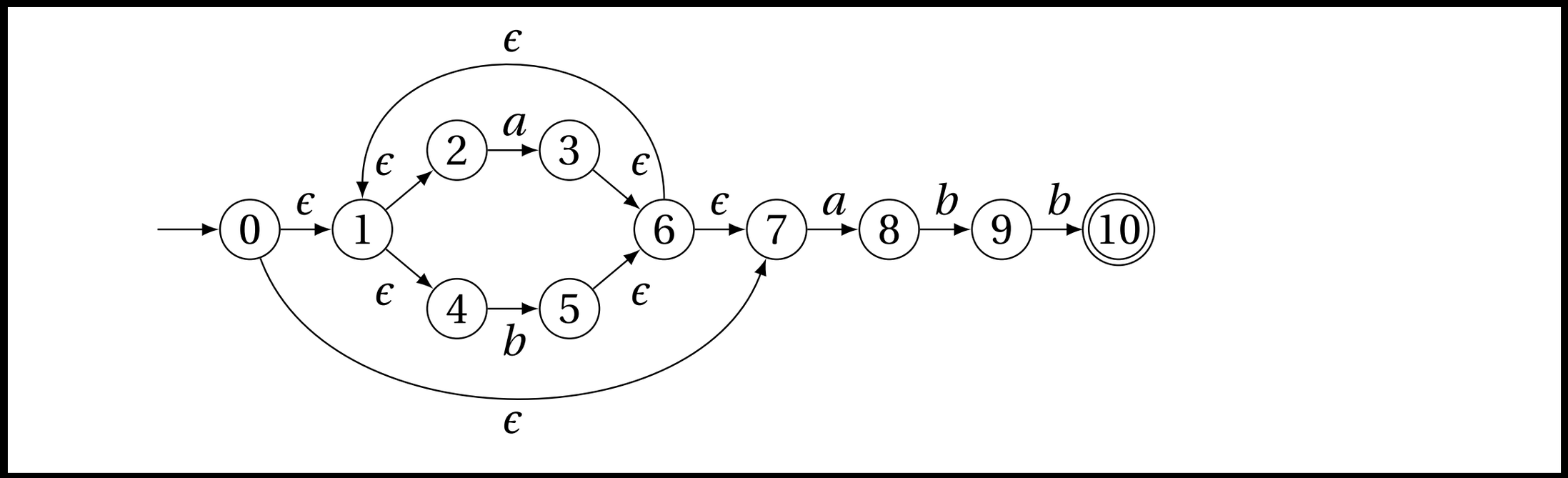
مثال تبدیل عبارت منظم به NFA

عبارت منظم: (a | b)*abb



مثال تبدیل عبارت منظم به NFA

عبارت منظم: (a | b)*abb

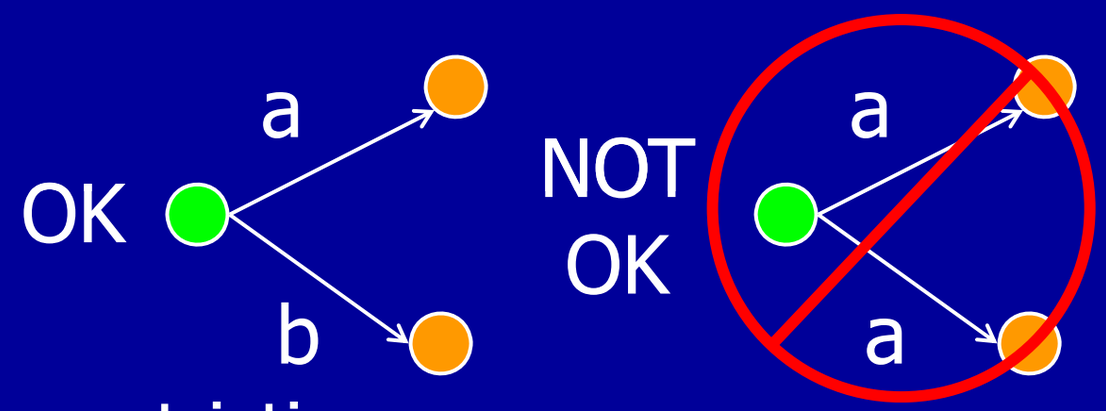


آیا aabb را می‌پذیرد؟

تبدیل NFA به DFA

NFA vs. DFA

- DFA
  - No ε transitions
  - At most one transition from each state for each letter



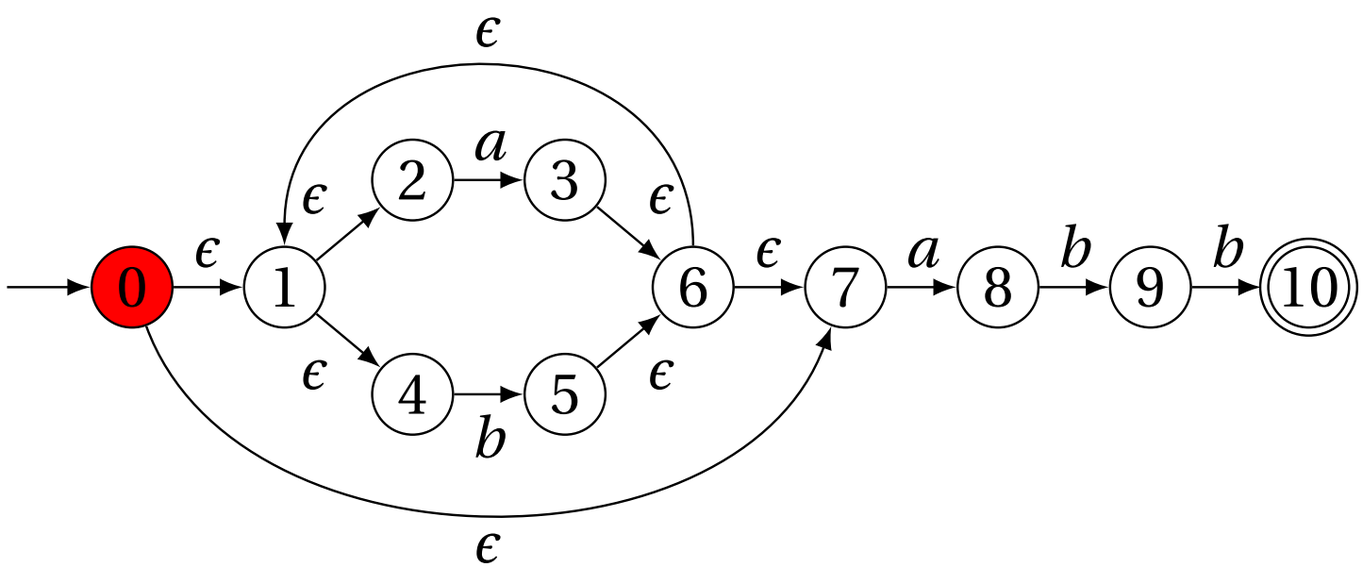
- NFA – neither restriction

Conversions

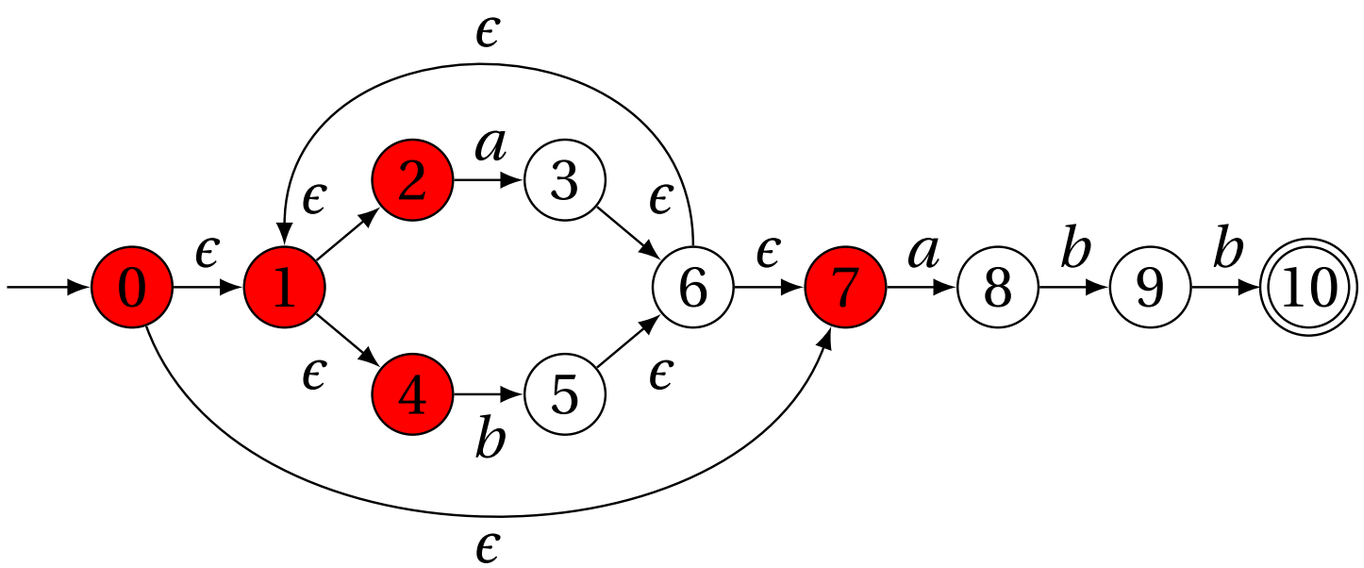
- Our regular expression to automata conversion produces an NFA
- Would like to have a DFA to make recognition algorithm simpler
- Can convert from NFA to DFA (but DFA may be exponentially larger than NFA)

مثالی از شبیه‌سازی NFA

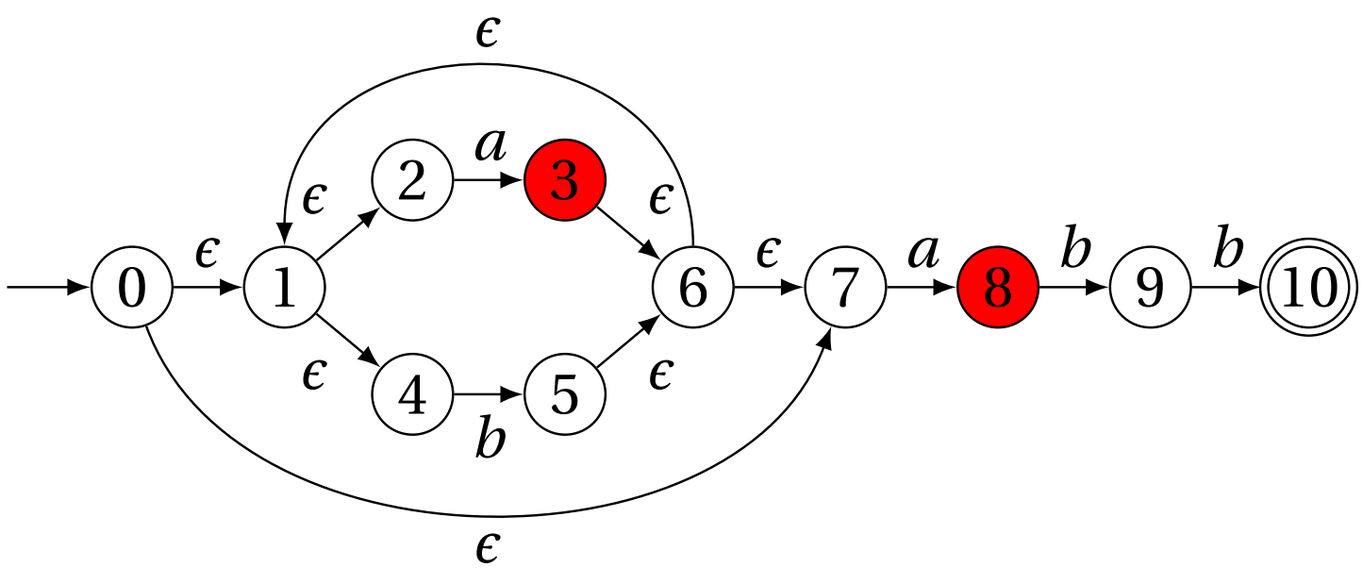
Simulating an NFA: $\cdot aabb$, Start



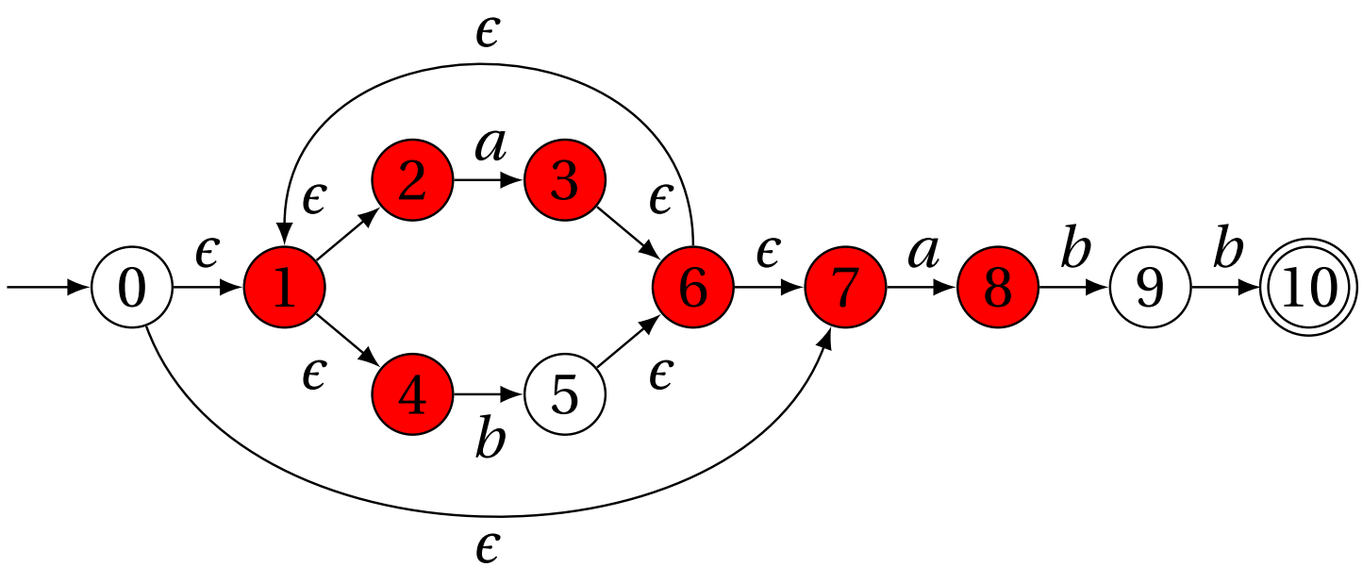
Simulating an NFA: $\cdot aabb$, ϵ-closure



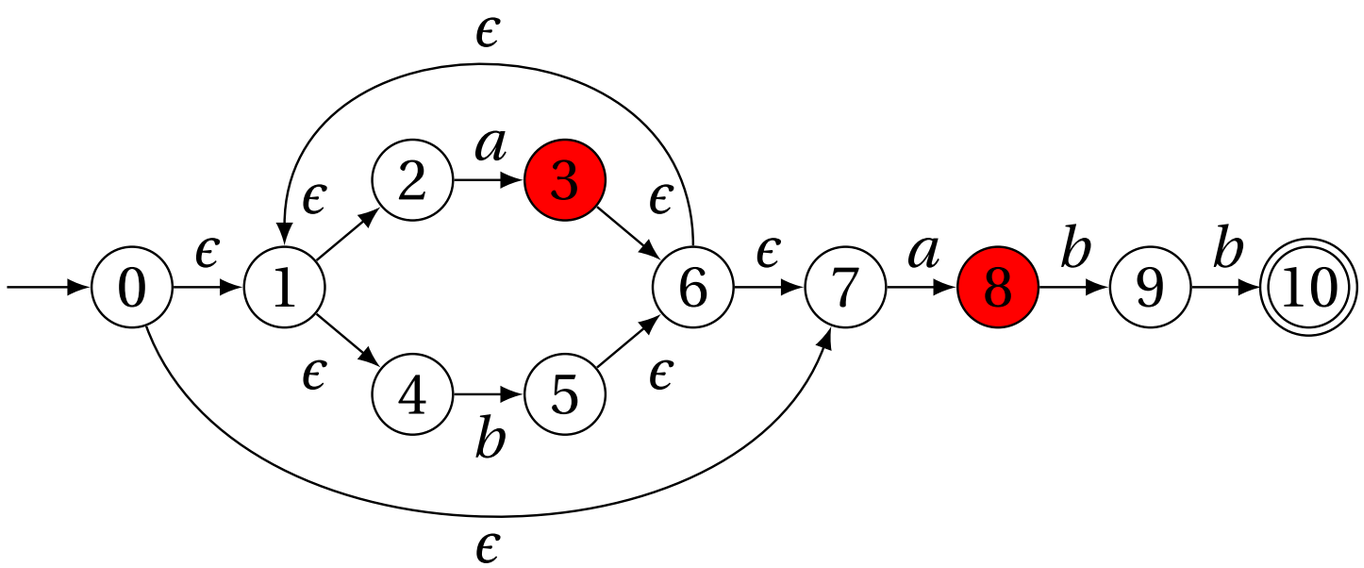
Simulating an NFA: $a{\cdot}abb$



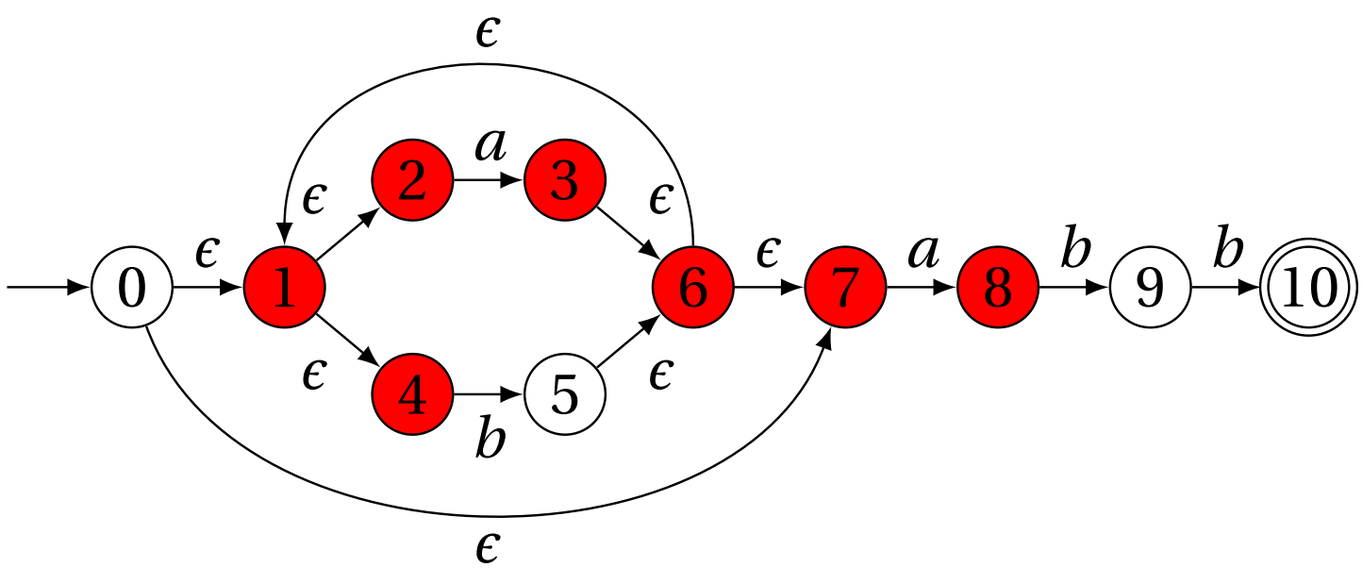
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



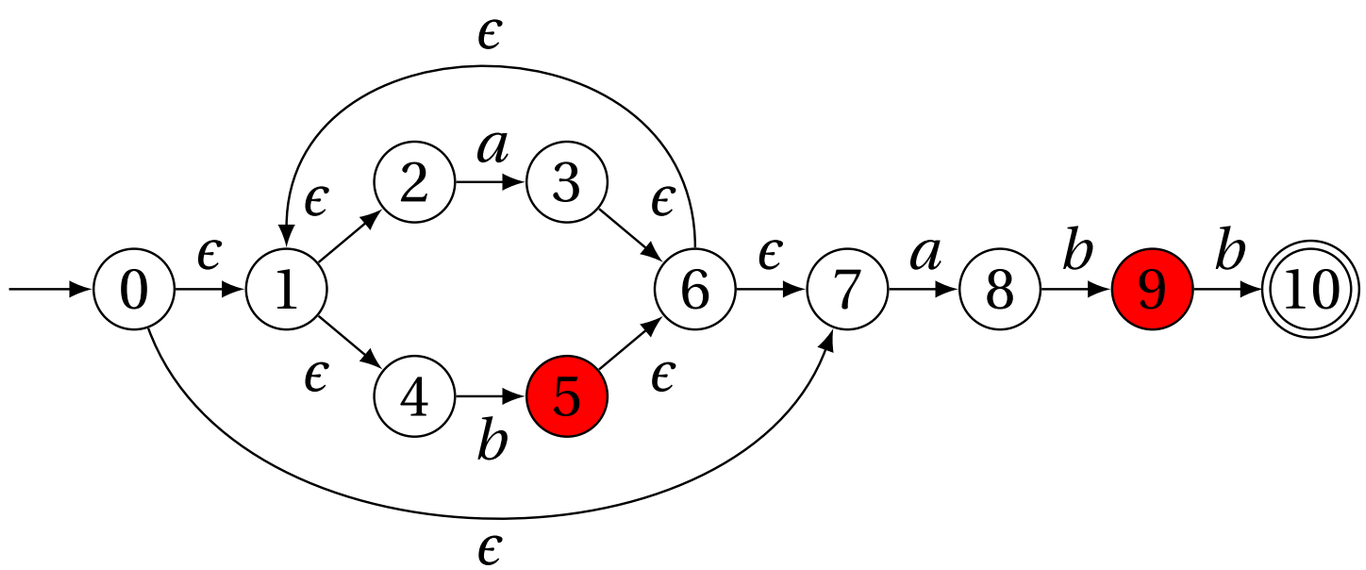
Simulating an NFA: $aa{\cdot}bb$



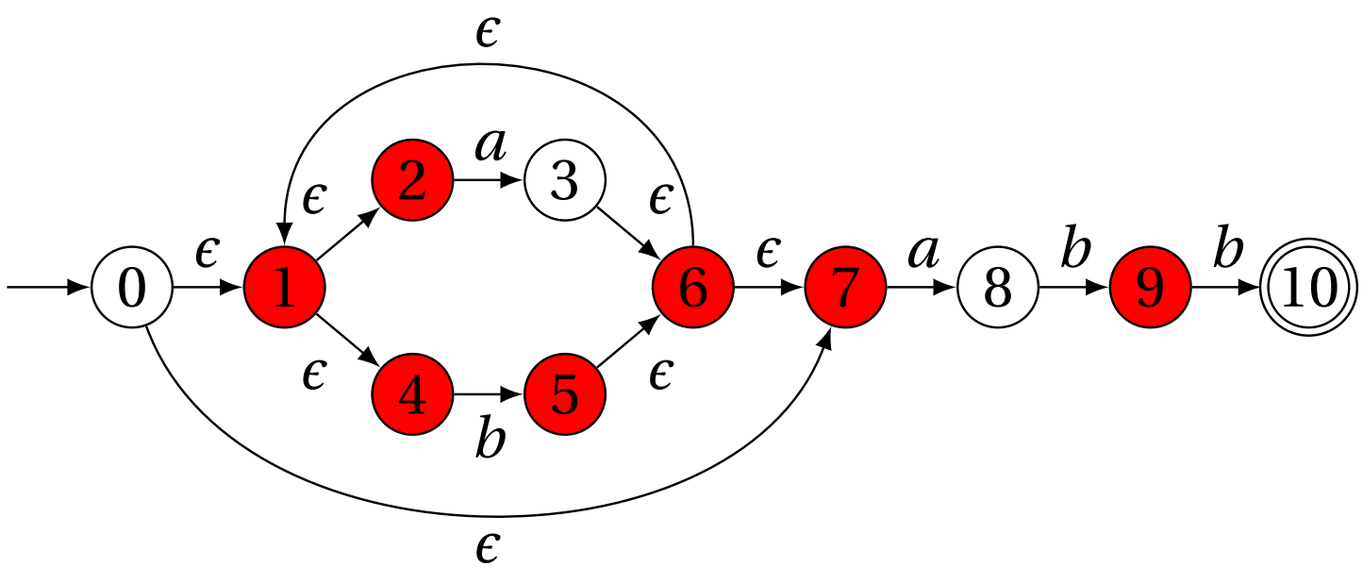
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



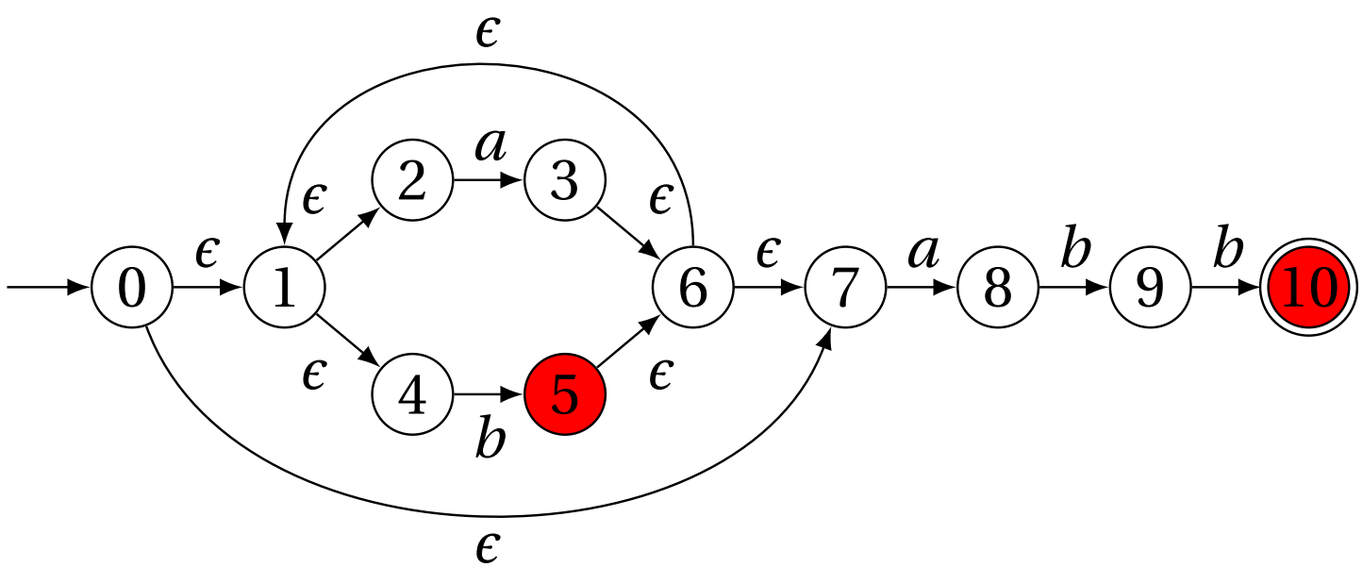
Simulating an NFA: $aab{\cdot}b$



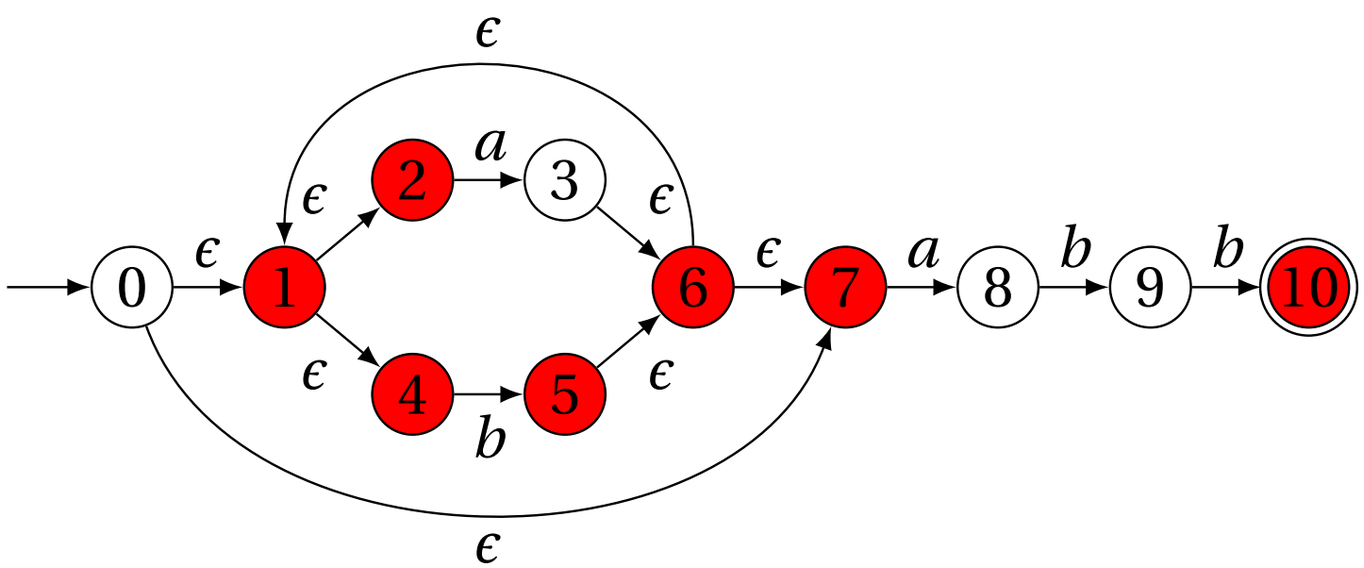
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$

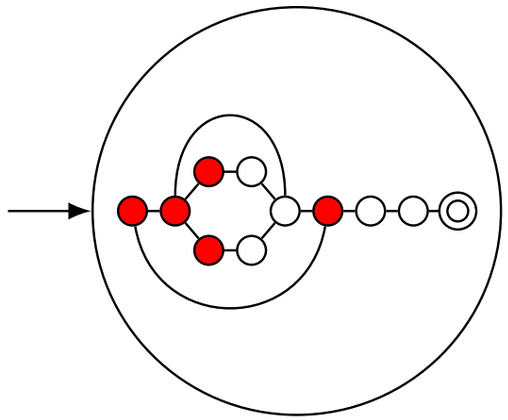


Simulating an NFA: $aabb\cdot$, Done

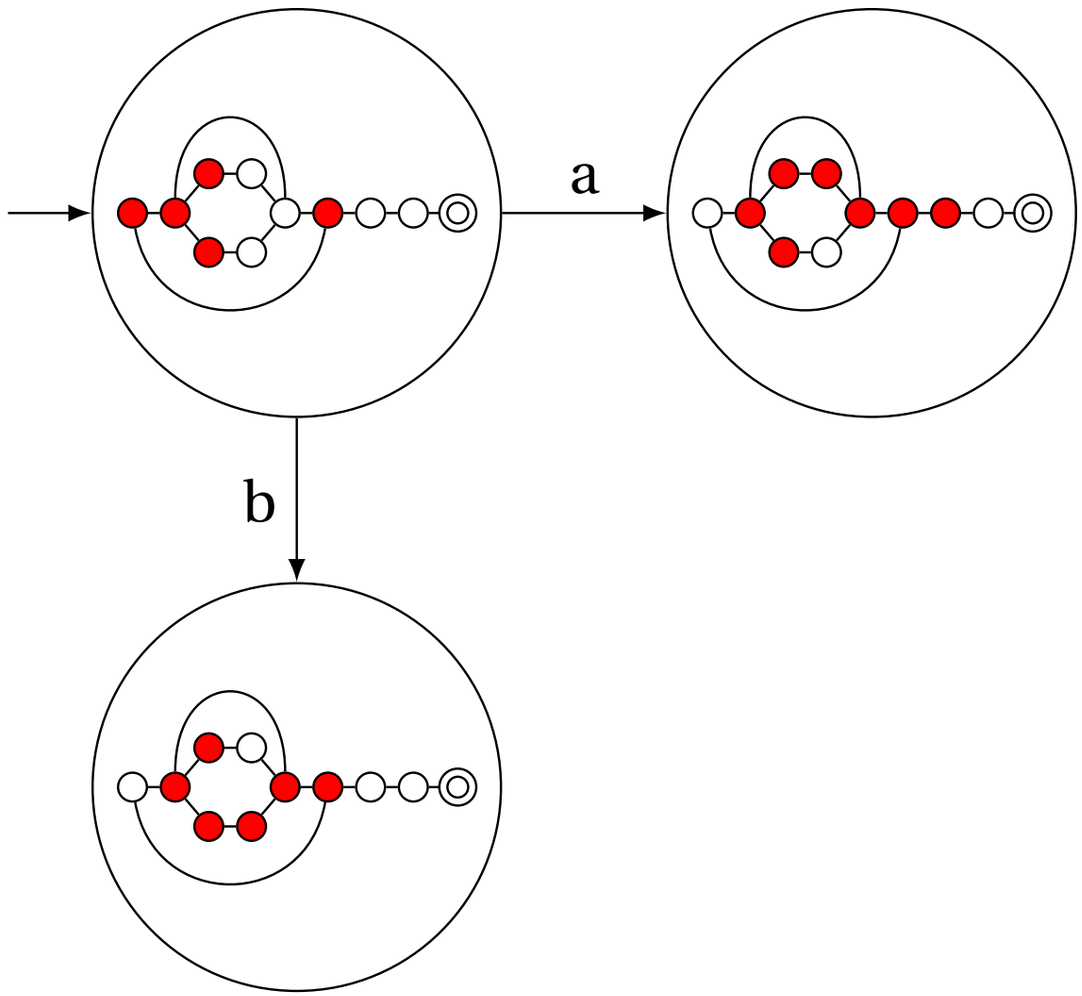


روش تبدیل

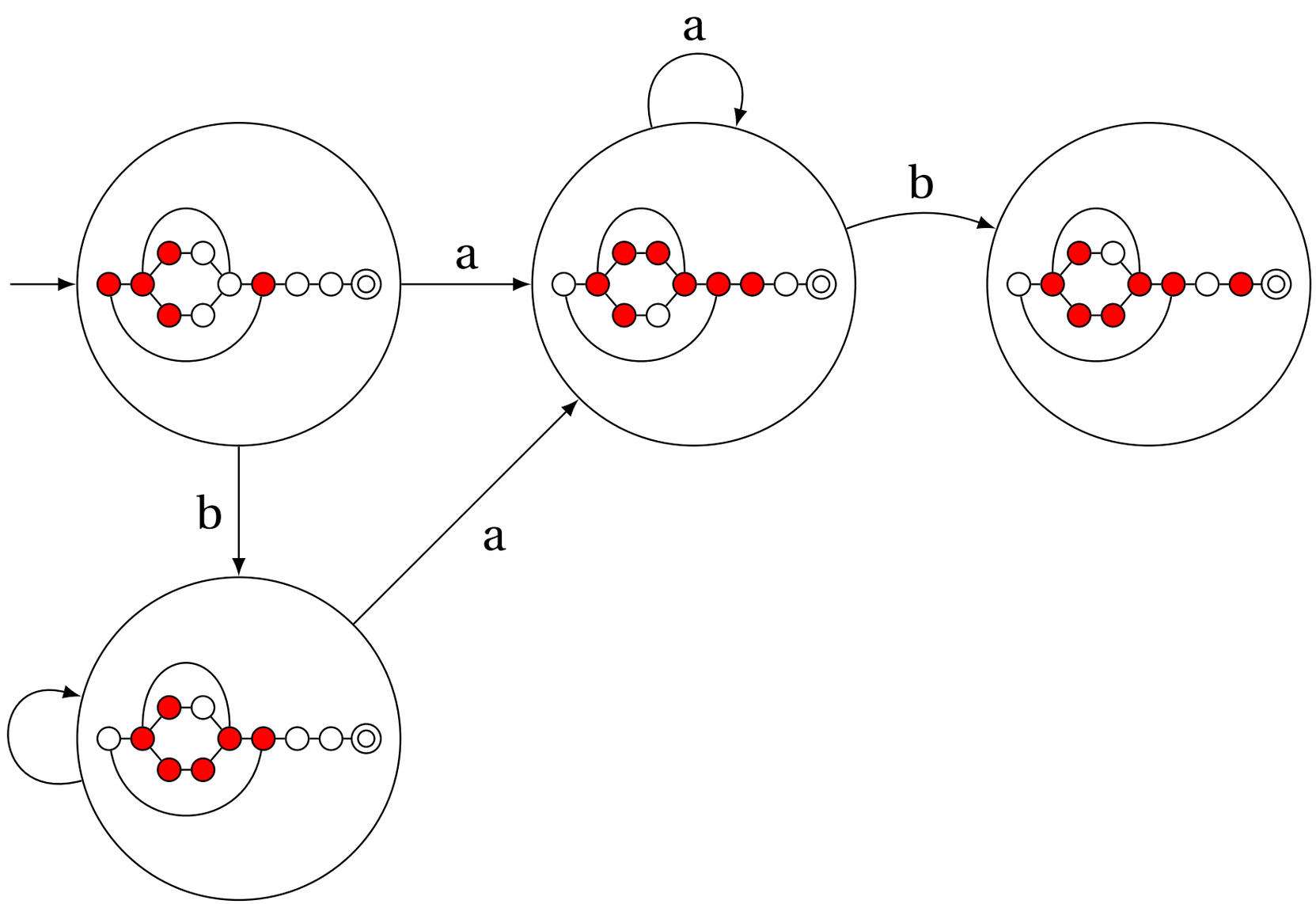
Subset construction for $(a \mid b)^* abb$



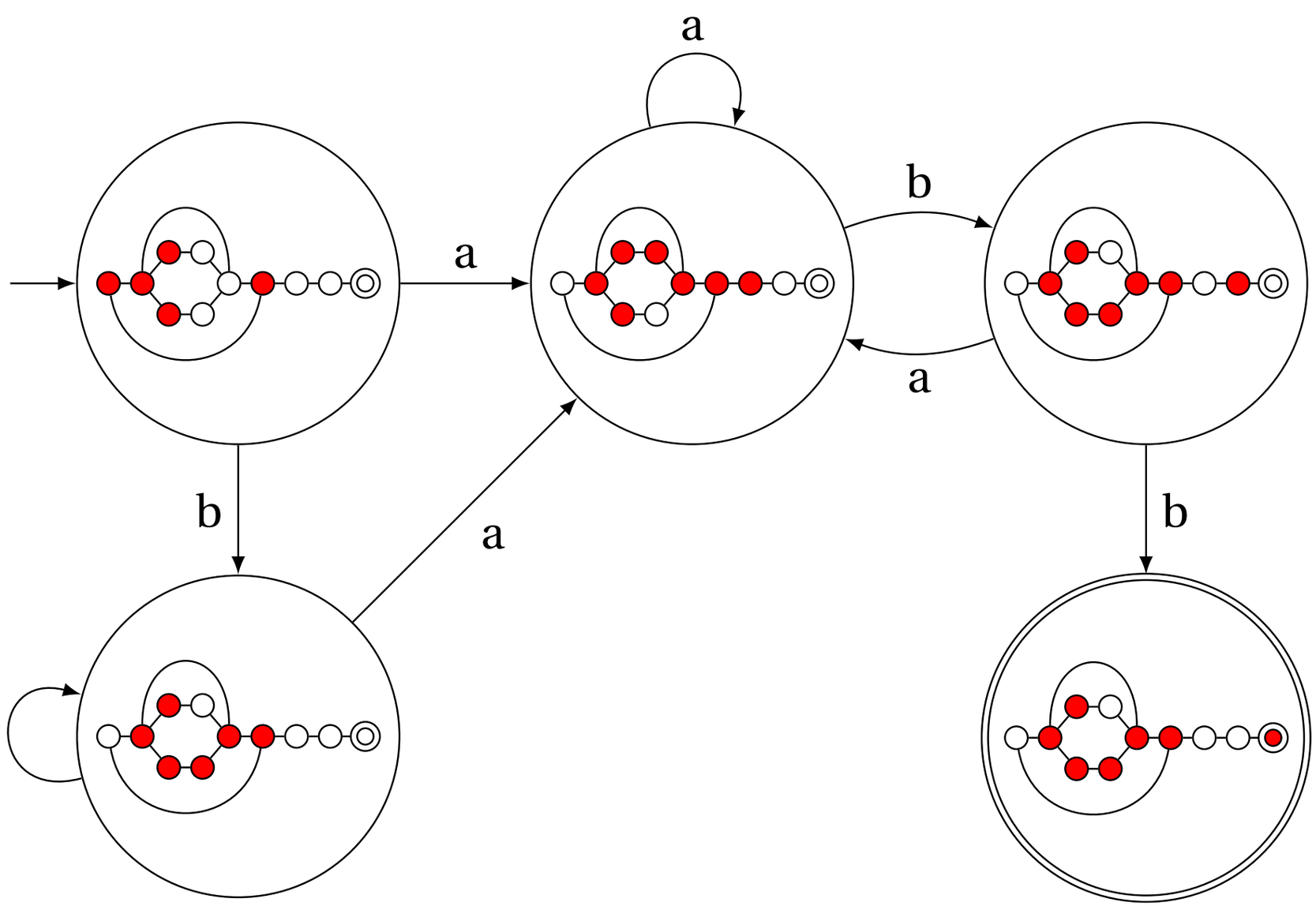
Subset construction for $(a \mid b)^* abb$



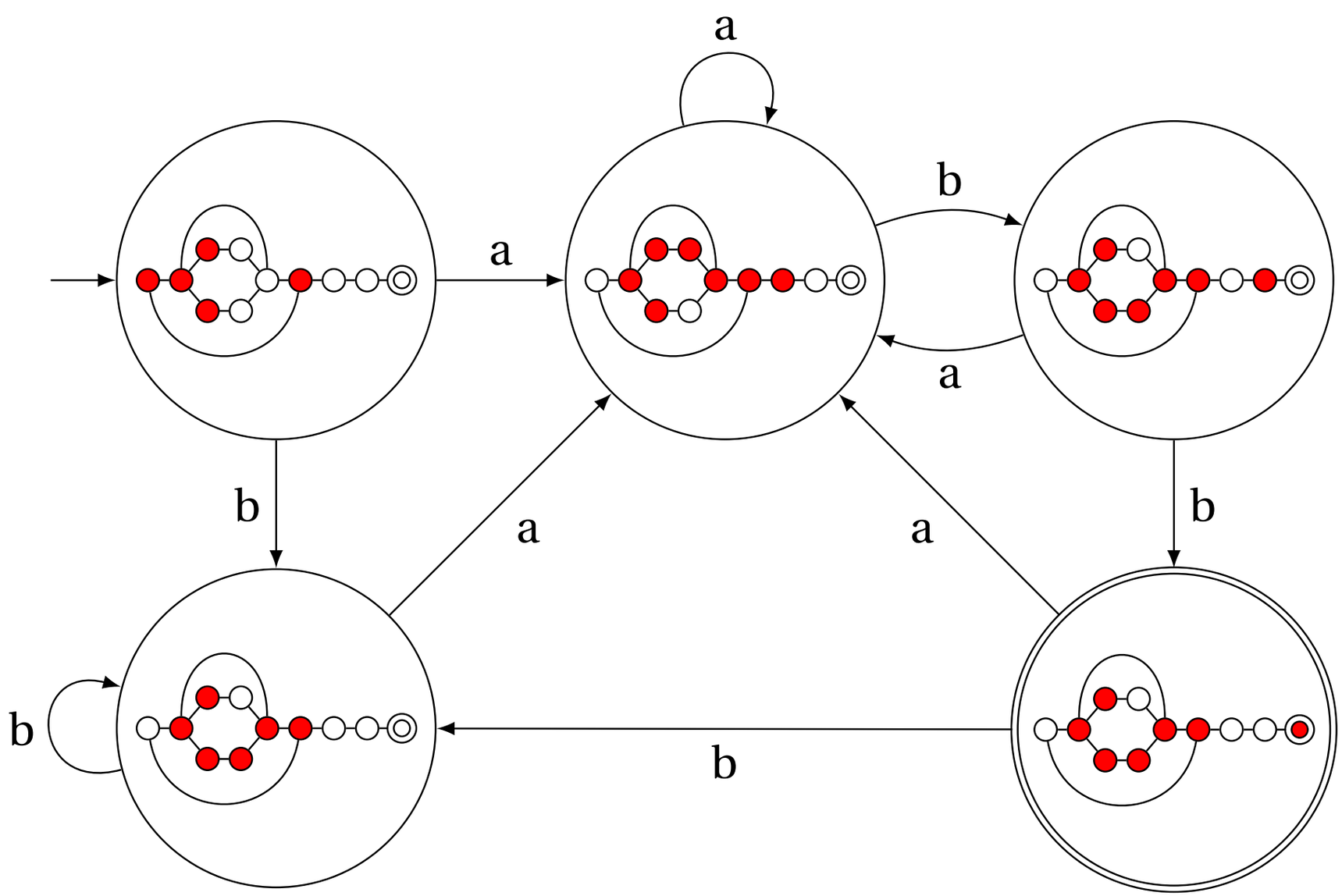
Subset construction for $(a \mid b)^* abb$



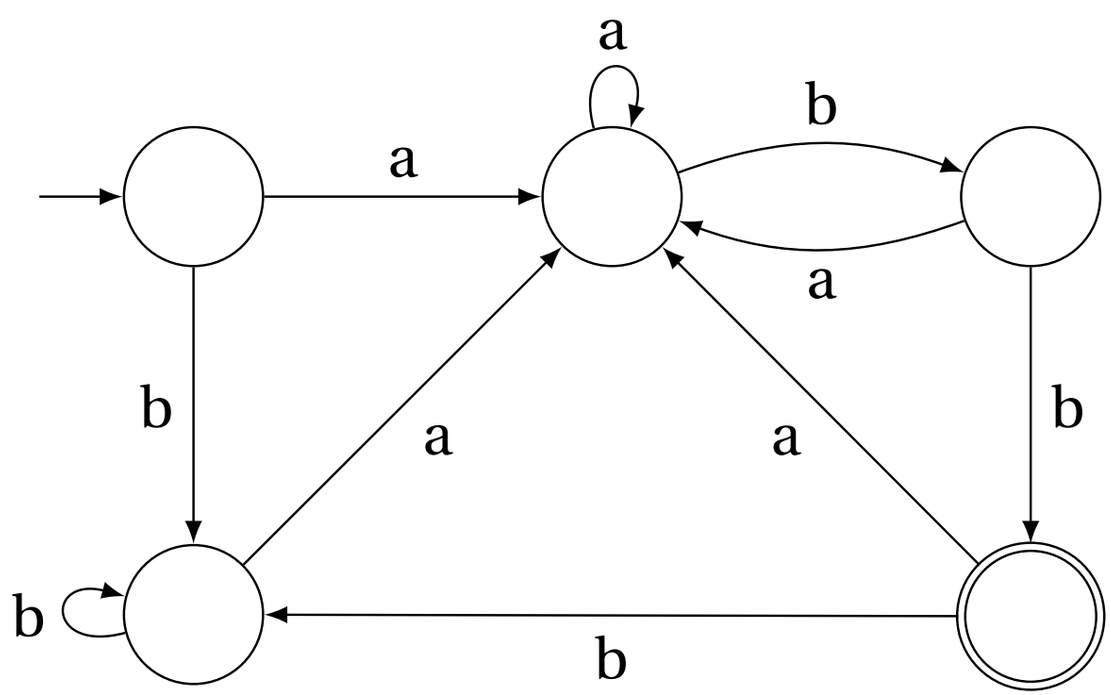
Subset construction for $(a \mid b)^* abb$



Subset construction for $(a \mid b)^* abb$



Result of subset construction for $(a \mid b)^* abb$



Is this minimal?

مثال از تک‌واژه‌های یک زبان

برخی نمونه‌های زبان

`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

FIGURE 2.2. Regular expressions for some tokens.

اتوماتای نمونه‌های زبان

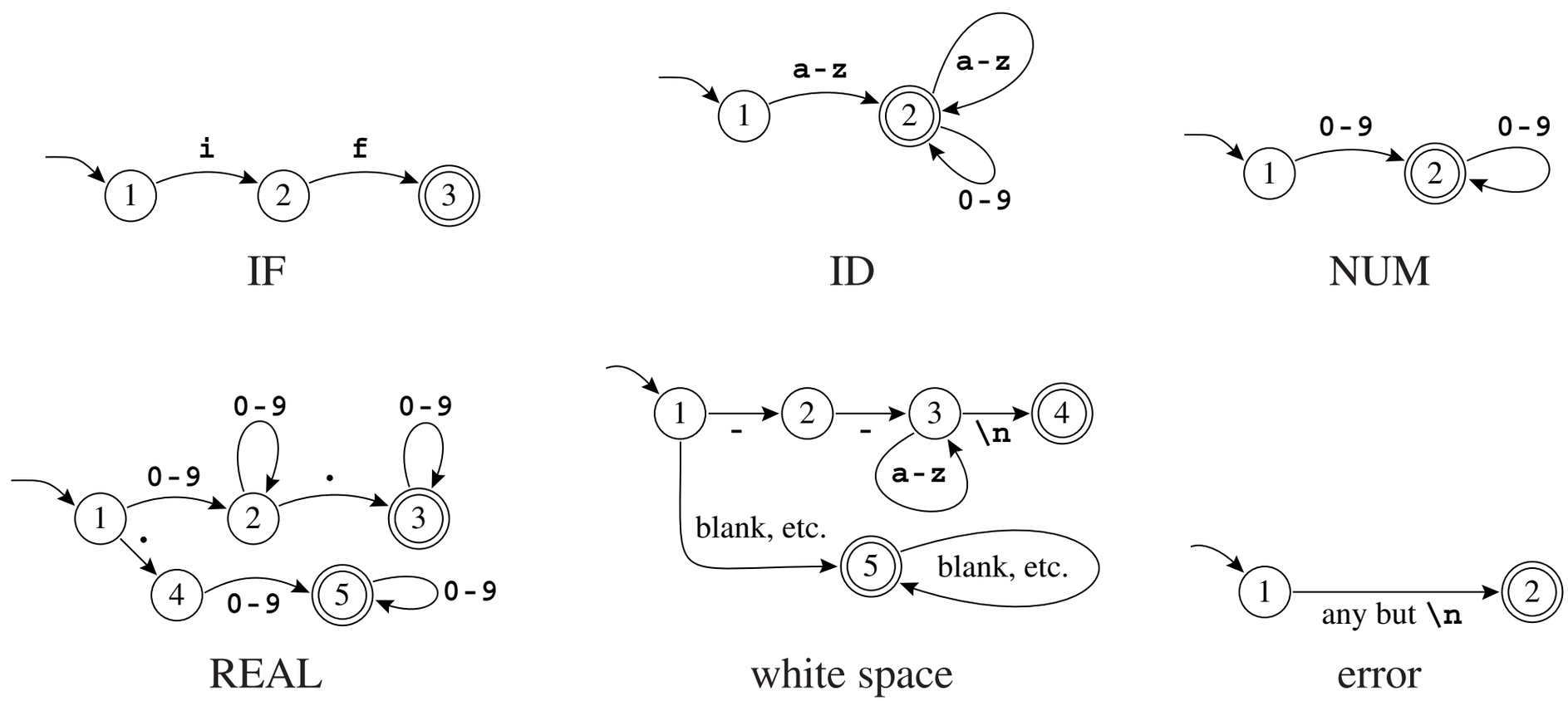


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

ادغام نمونه‌های زبان

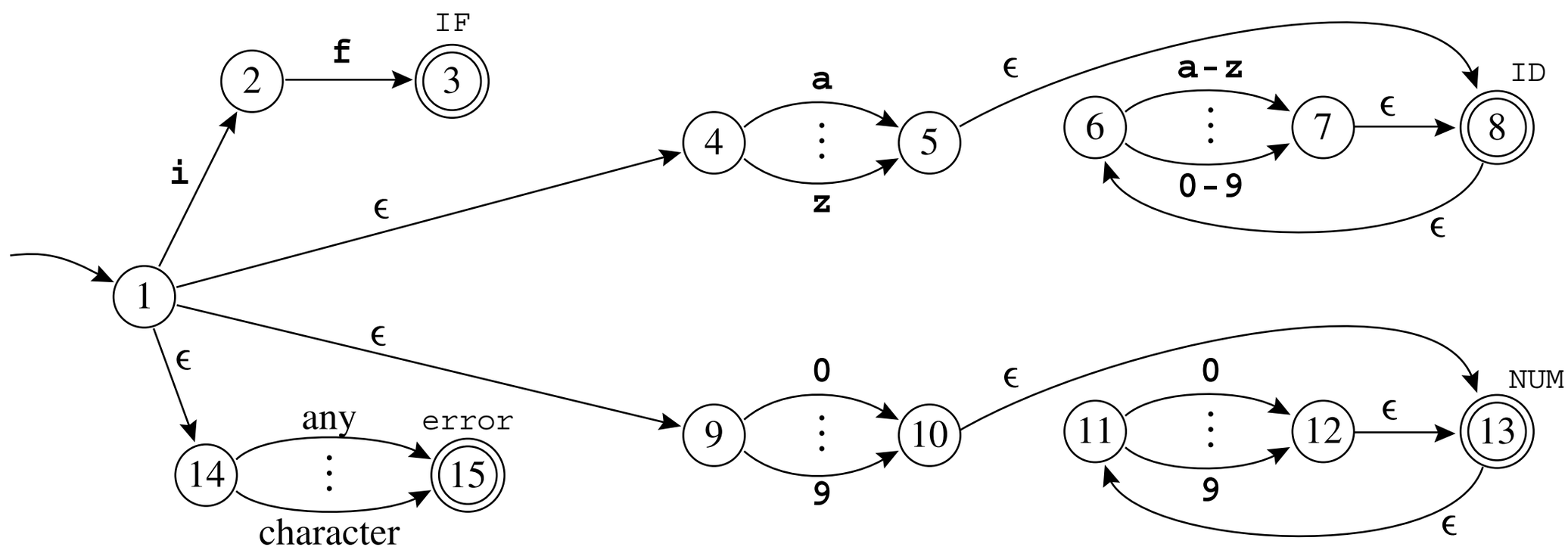


FIGURE 2.7. Four regular expressions translated to an NFA.

تبدیل NFA به DFA زبان

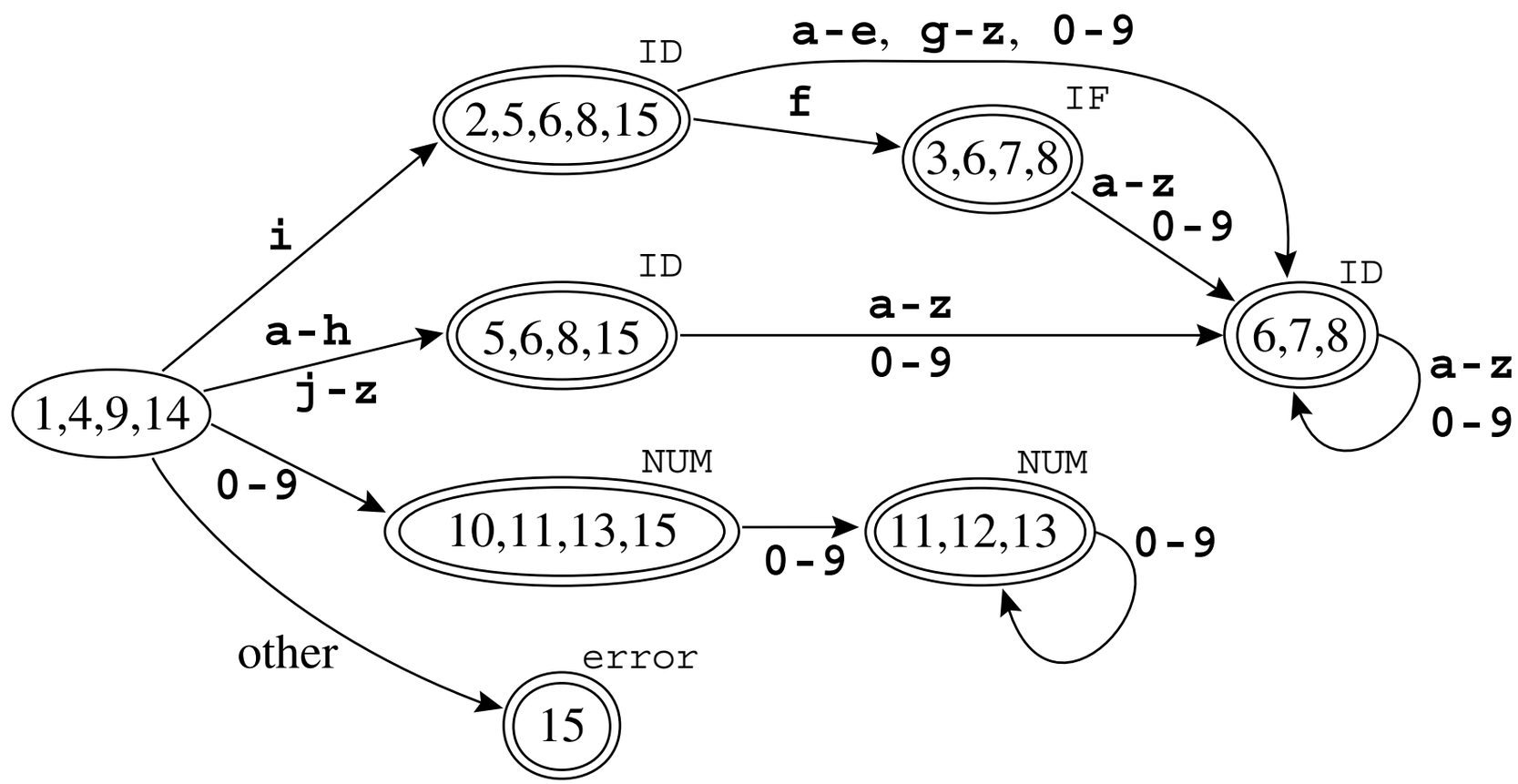
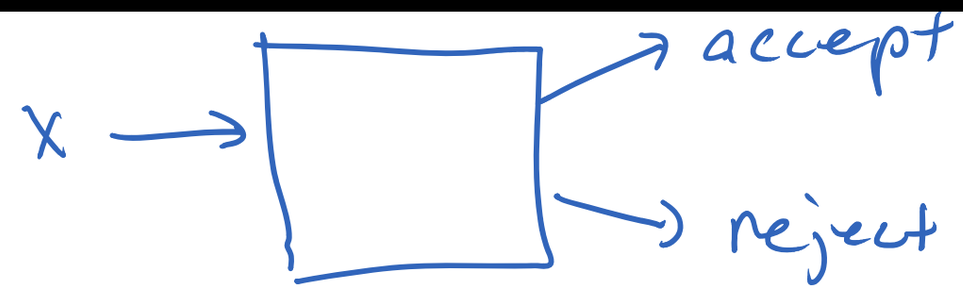


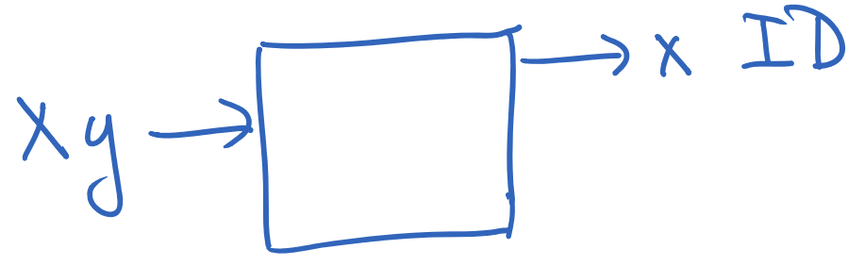
FIGURE 2.8. NFA converted to DFA.

در عمل

Theory



Practice

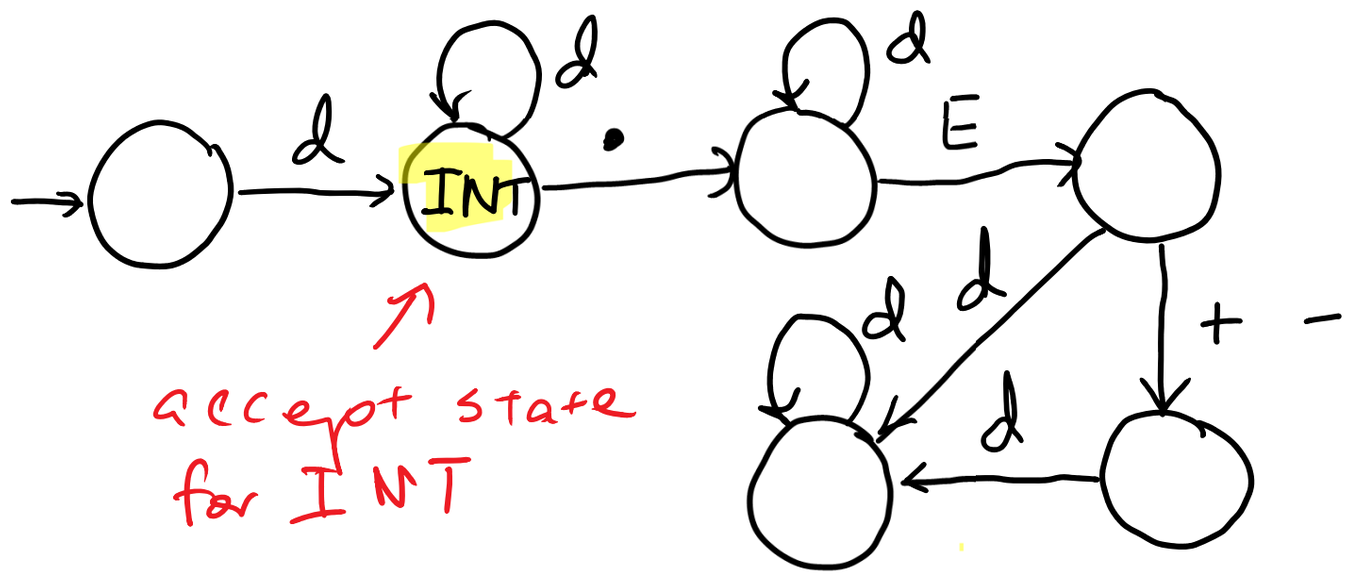


1. Lexeme is <u>prefix</u> of input
2. Multiple token types

Combined Automaton

Real: digit$^+$. digit* (E (+|−)? digit$^+$).

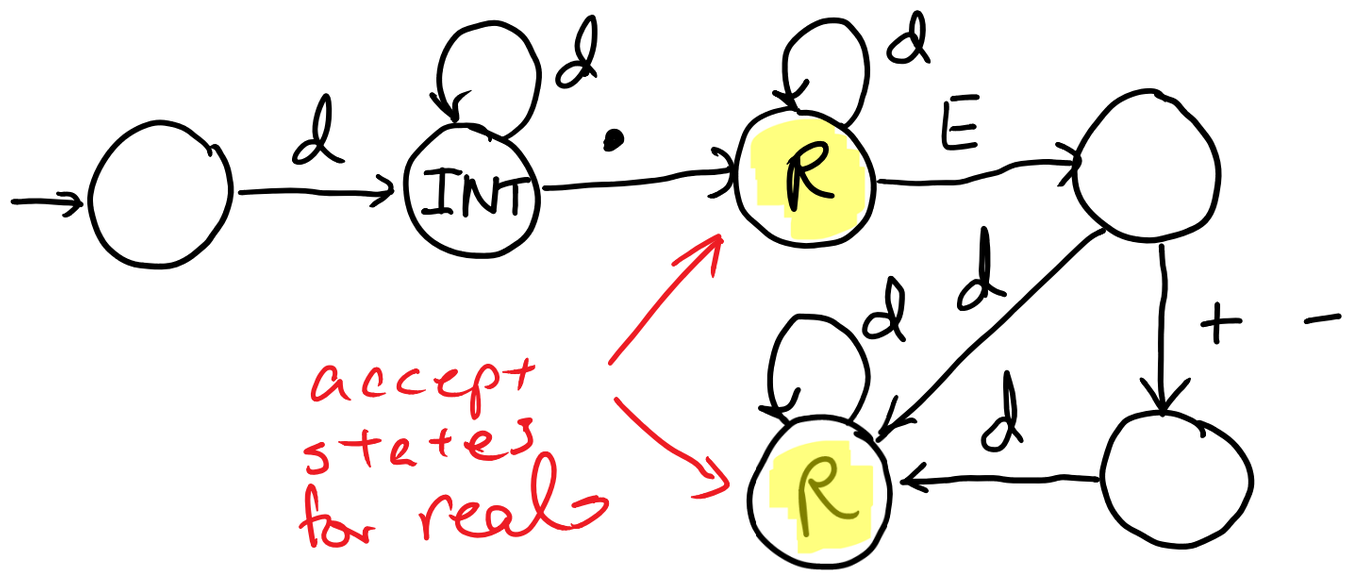
Int: digit$^+$



Combined Automaton

Real: digit$^+$. digit* (E (+|−)? digit$^+$).

Int: digit$^+$

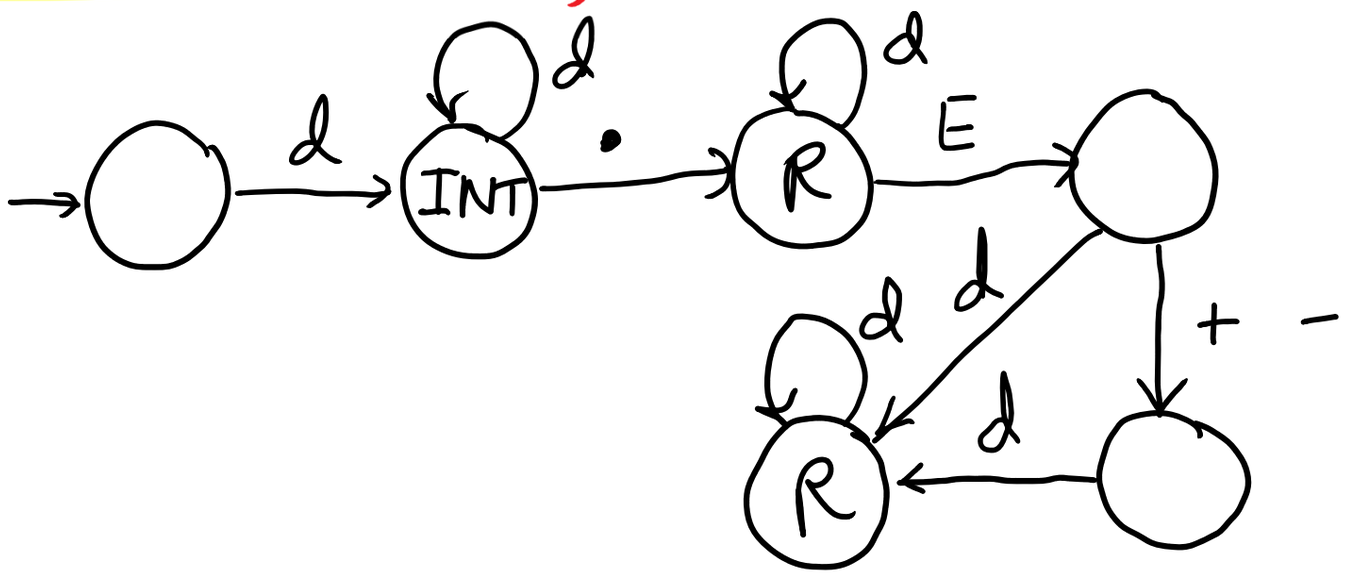


Input: 12.34+6

Tokens $\boxed{1}$ $\boxed{2.3}$ $\boxed{4}$?

Tokens $\boxed{12.34}$?

Longest lexeme rule ("maximal munch")

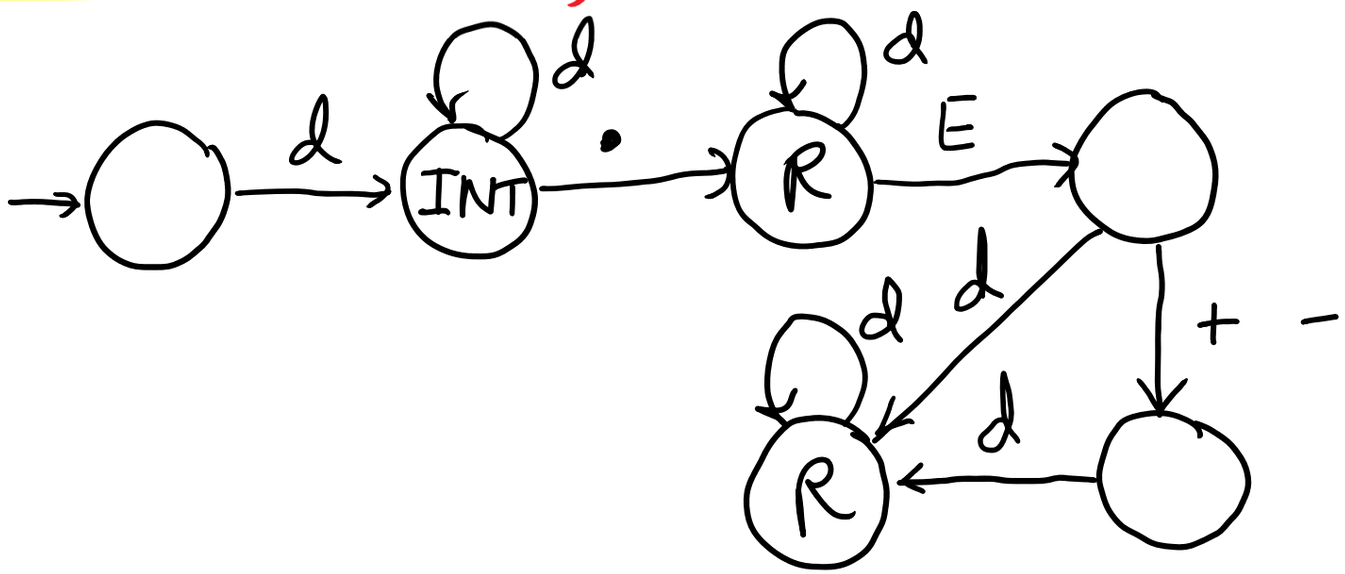


Input: 12.34+6

Tokens $\boxed{1}$ $\boxed{2.3}$ $\boxed{4}$?

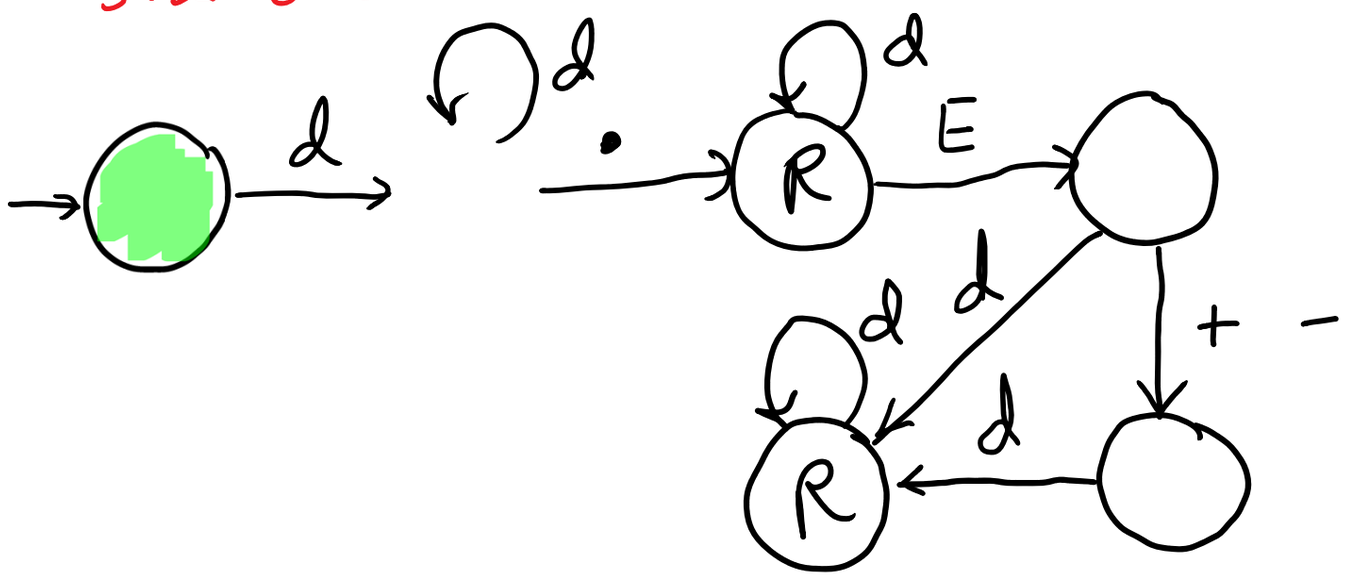
Tokens $\boxed{12.34}$?

Longest lexeme rule ("maximal munch")



Input: 12.34+6

startchar nextchar (next char to read)

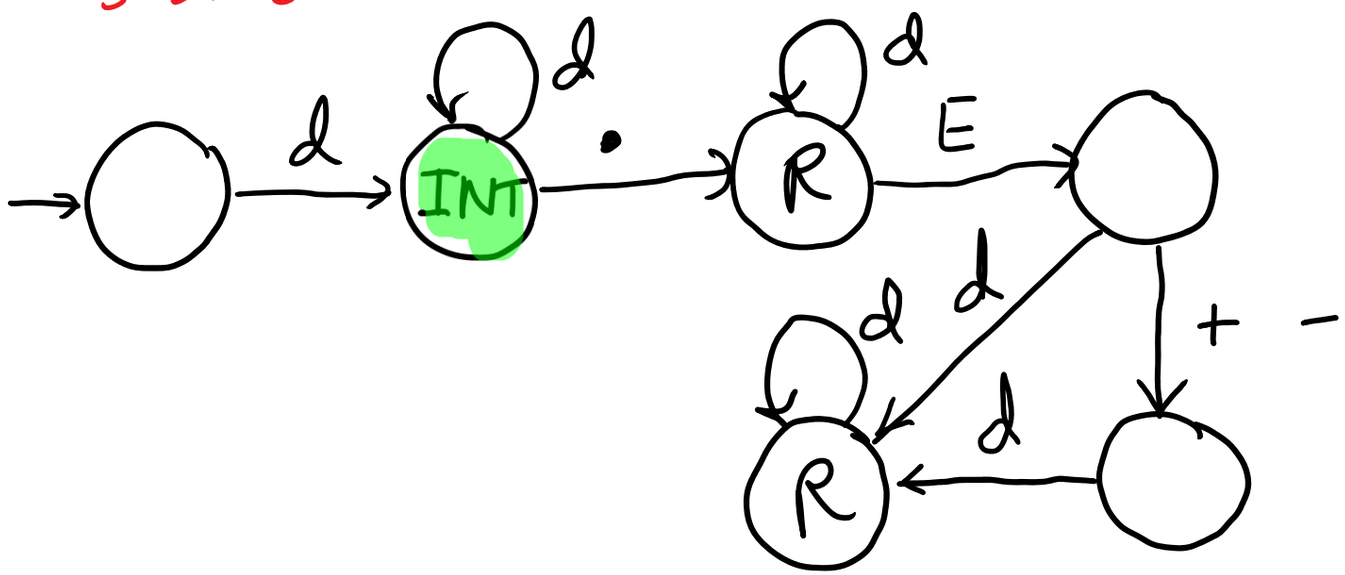


Input: 12.34+6

startchar

nextchar

(advanced)



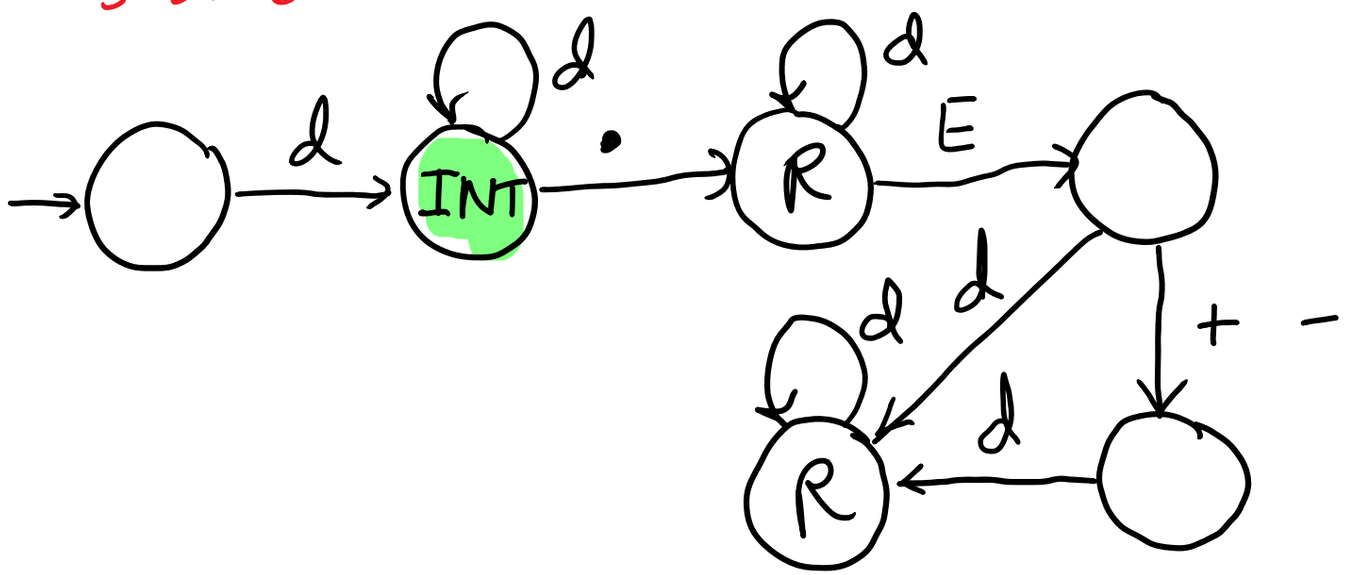
Input: 12.34+6

lastchar

startchar

nextchar

(end of longest lexeme so far – "1" in this case)



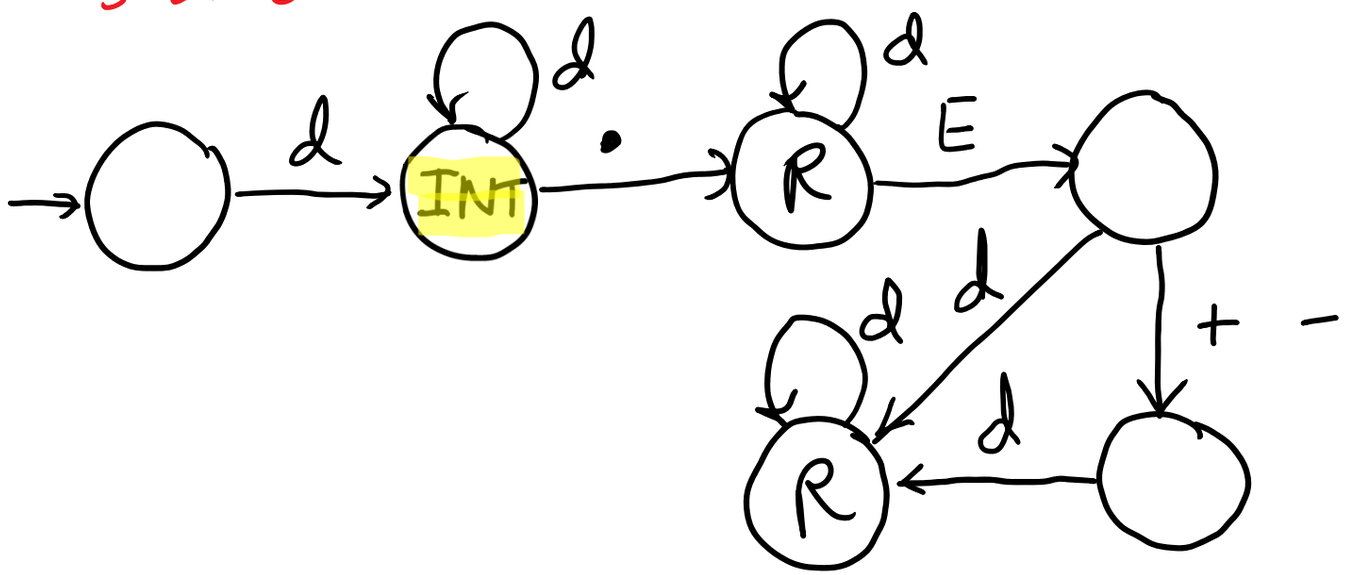
Input: 12.34+6

lastchar

startchar

nextchar

longest so far:
"12"



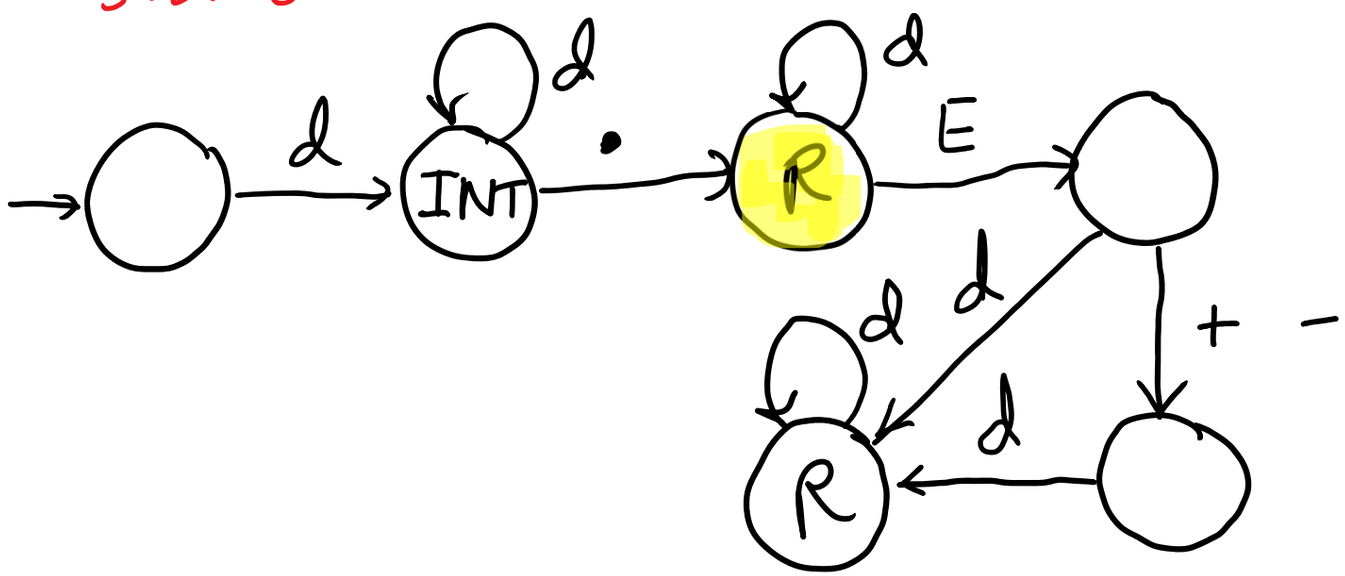
Input: 12.34+6

startchar

nextchar

lastchar

longest so far:
"12."



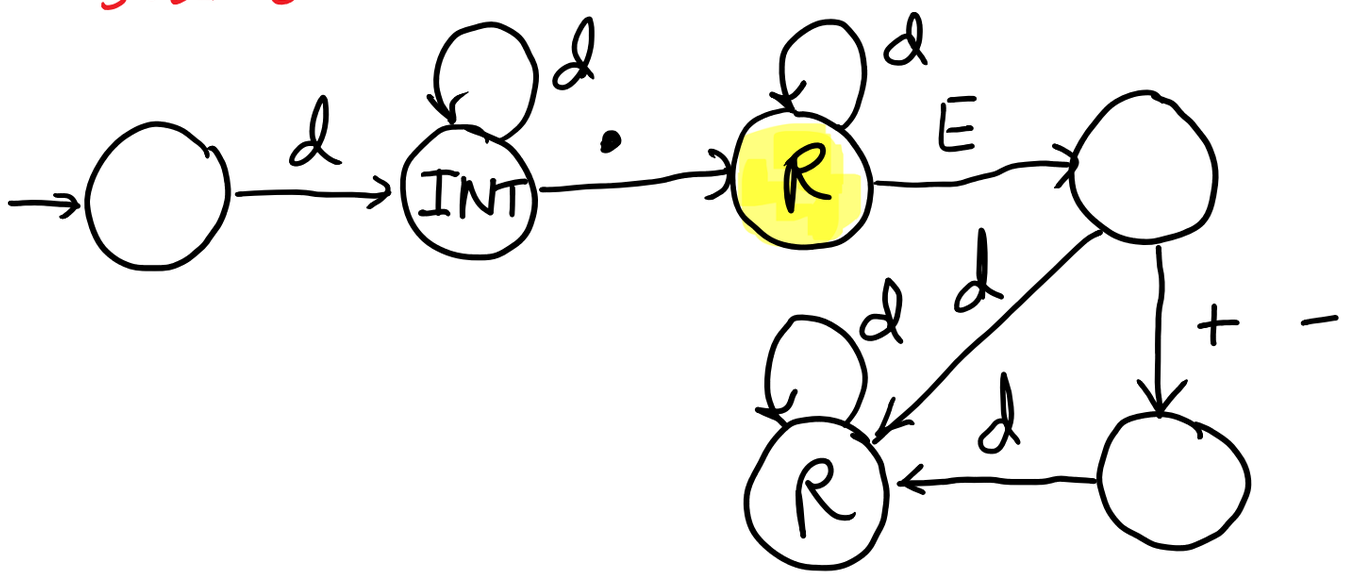
Input: 12.34+6

lastchar

startchar

nextchar

longest so far:
"12.3"



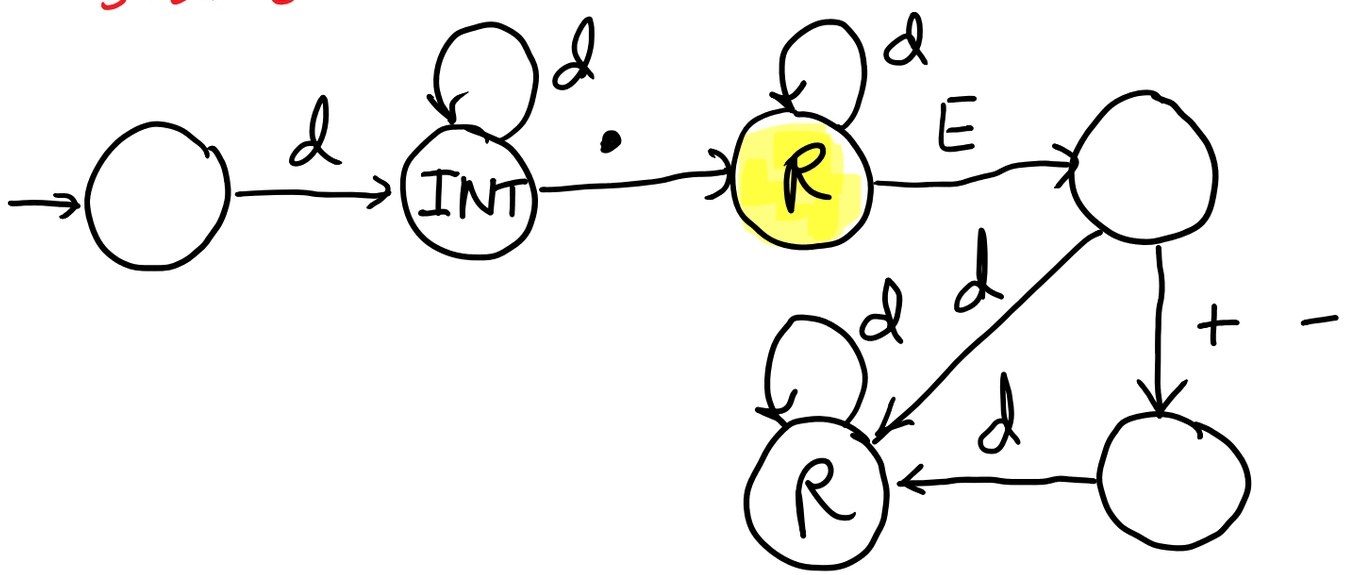
Input: 12.34+6

lastchar

startchar

nextchar

longest so far:
"12.34"



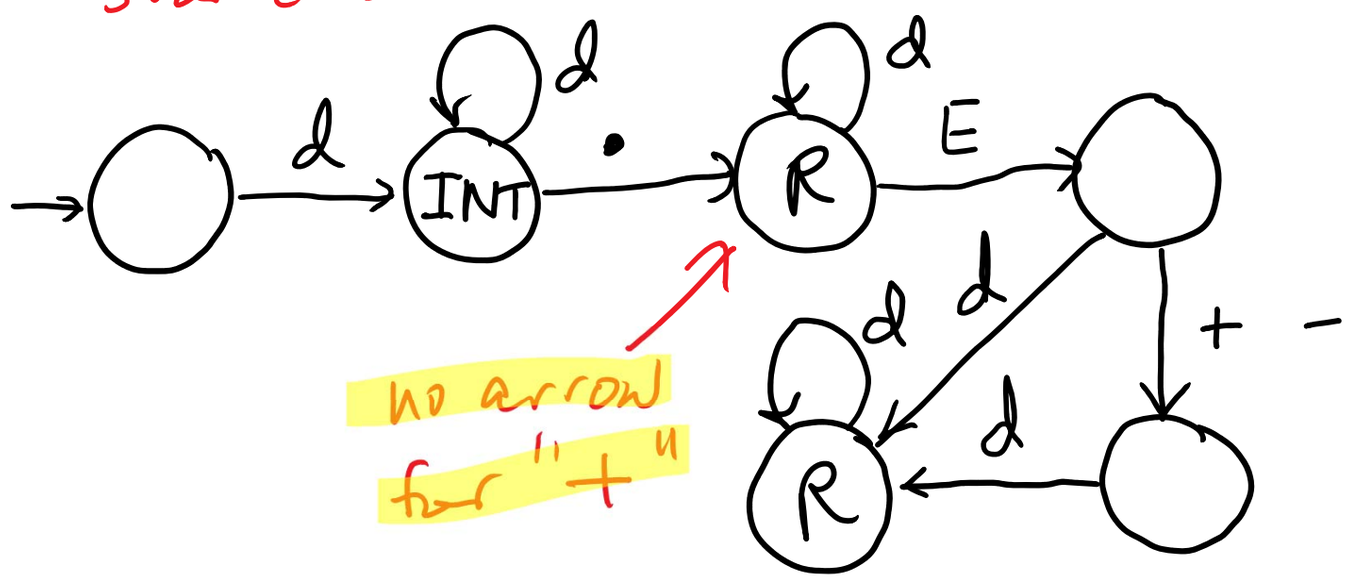
Input: 12.34+6

lastchar

startchar

nextchar

longest so far: "12.34"



no arrow for "+"

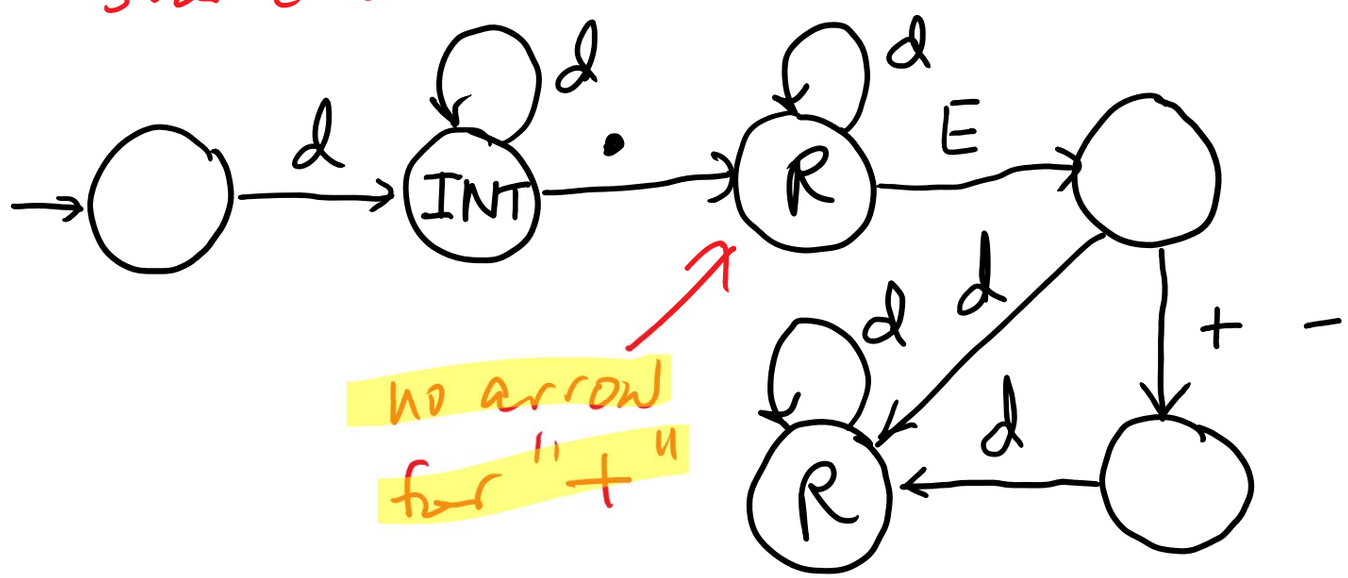
Input: 12.34+6

lastchar

startchar

nextchar

longest ~~so far~~: "12.34"



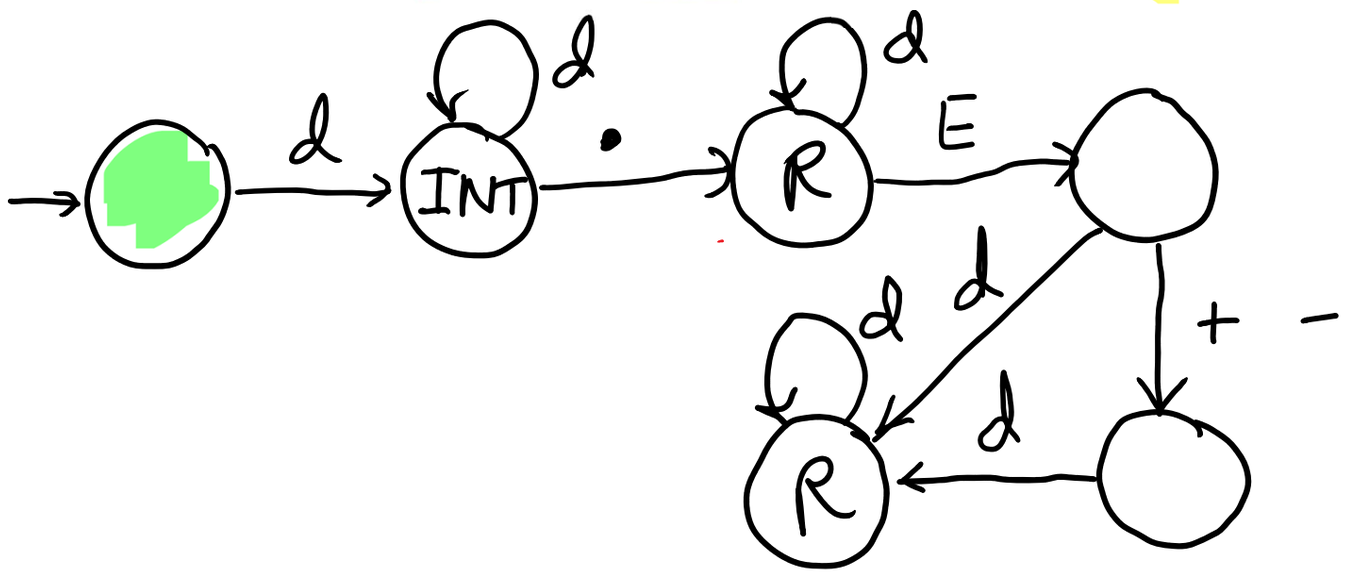
no arrow for "+"

Ready for next lexeme

Input: 12.34+6

startchar

nextchar



lastchar

longest ~~so far~~: "12.34"

Input: 12.34+6

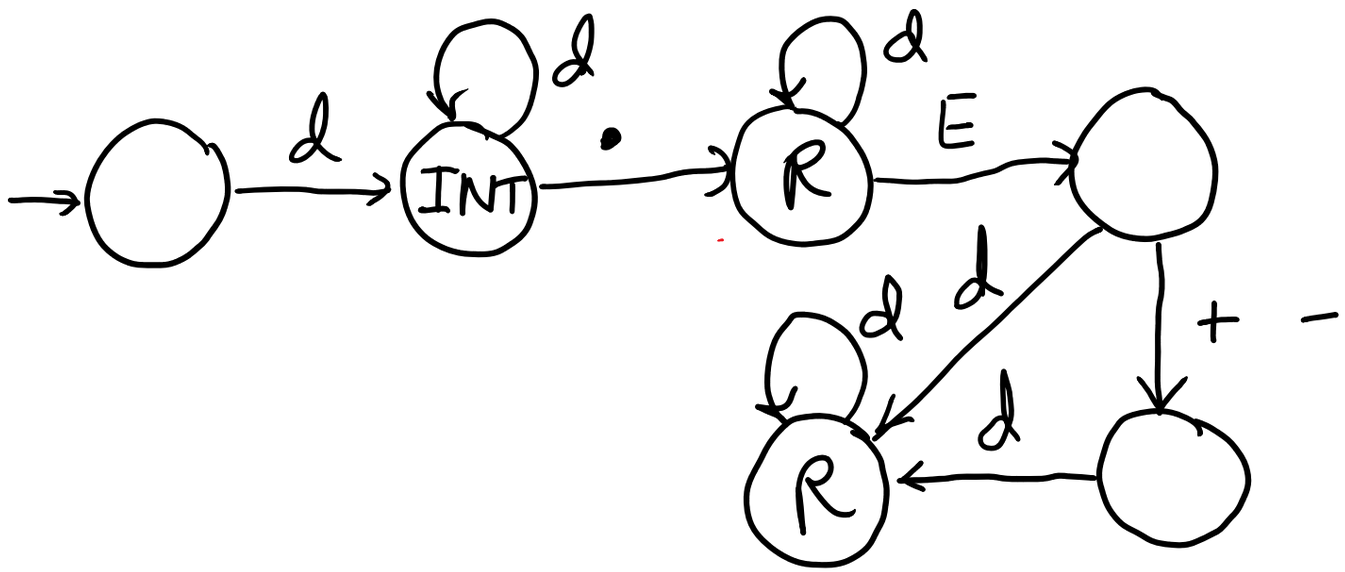
startchar

nextchar

"Look ahead" – had to read "+" to see that end of lexeme was previous char.

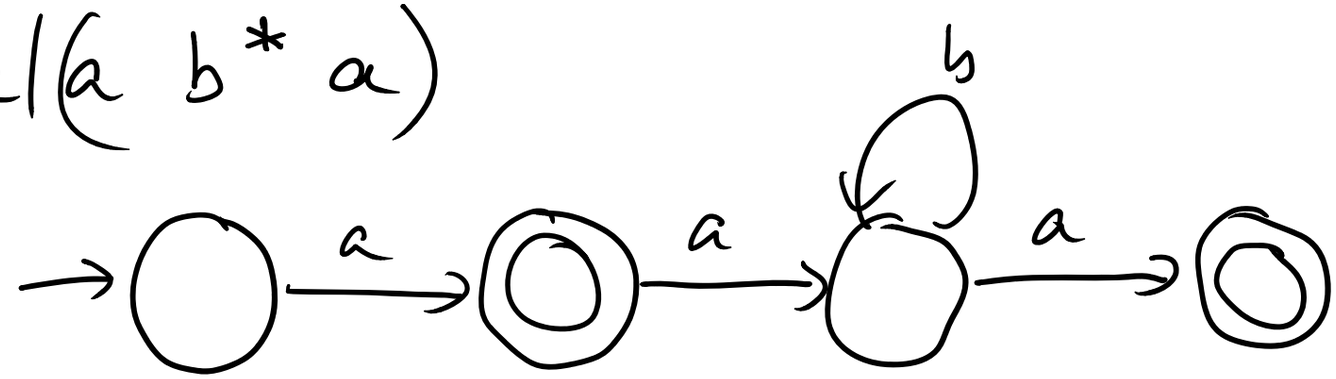
Lookahead = max(nextchar – lastchar)

Lookahead for this example is at least 1. Is it more?



Unbounded lookahead?

$a|(a\ b^*\ a)$



Input: a a b b b b b b b . . . . b

last char

next char

May need to read whole file to find first lexeme!

C Comments

/* ... */

Has unbounded lookahead if handled with a single pattern.

If closing "*/" is missing, tokenizes the comment: $\boxed{/}$ $\boxed{*}$ ....

Saves string unnecessarily

Start Conditions

A better way to handle comments

When you see "/*"

Use new DFA

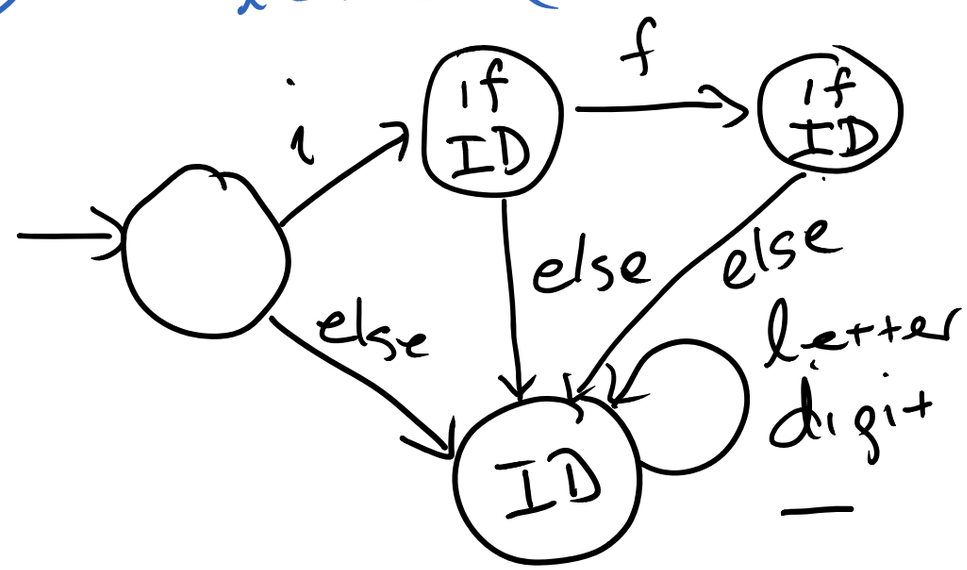
Patterns:

- any single char — discard
- "*/" - discard, return to main DFA

Rule Ordering

if if

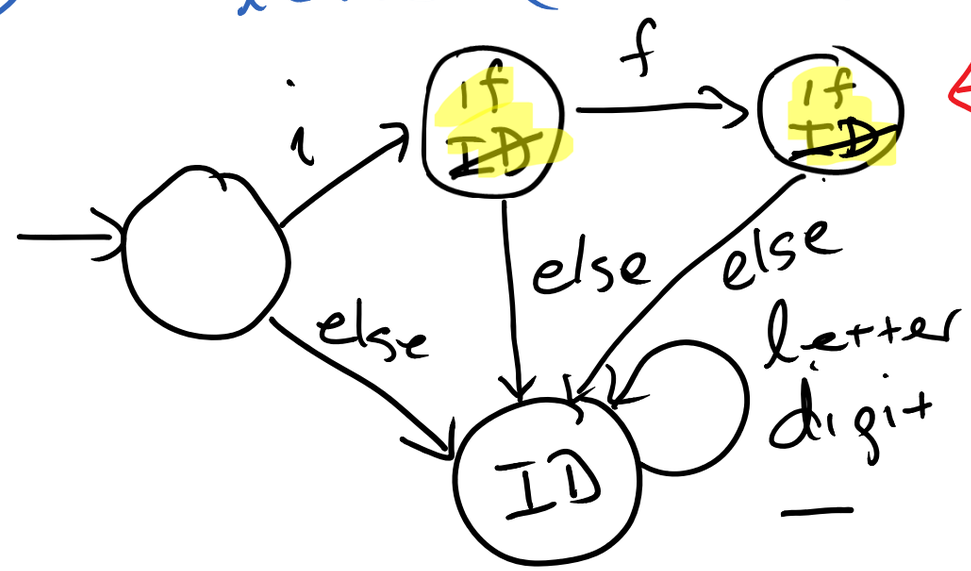
ID letter(letter | digit | _)



Rule Ordering

if if

ID letter(letter | digit | _)

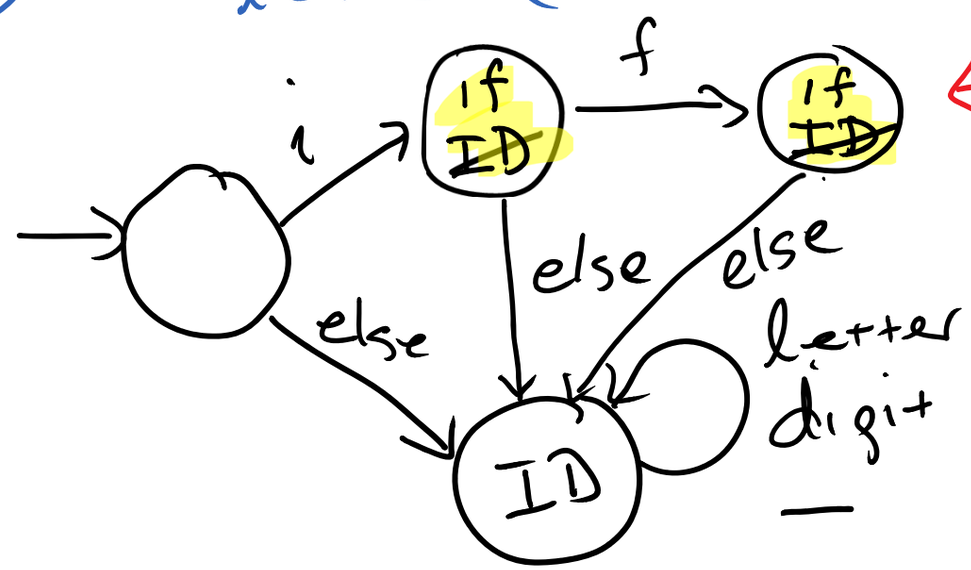


← label for first rule in file wins.

Rule Ordering

if if ← special case first

ID letter(letter | digit | _) ← default



← label for first rule in file wins.

JFLex

تولید خودکار

- Writing scanners is a rather “robotic” activity which can be automated.
- JLex (JFlex)
  - input:
    - a set of REs and action code
  - output
    - a fast lexical analyzer (scanner): based on a DFA
- Or the lexer is built into the parser generator as in JavaCC

ساختار فایل JFlex

user code (added to start of generated file)

User code is copied directly into the output class

%%

options

JLex directives allow you to include code in the lexical analysis class, change names of various components, switch on character counting, line counting, manage EOF, etc.

%{

user code (added inside the scanner class declaration)

%}

macro definitions

Macro definitions gives names for useful regular expressions (regexps)

%%

lexical declaration

Regular expression rules define the tokens to be recognised and actions to be taken

شروع فایل

```
import java_cup.runtime.*;

%%

%class Lexer
%unicode
%cup
%line
%column
%state STRING

%{
...
```

تعریف یک حالت

```
%state STRING

%{
  StringBuffer string = new StringBuffer();

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }

%}

...
```

ماکروها

```
%}
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} |

TraditionalComment = "/*" {CommentContent} "*"+ "/"
EndOfLineComment= "//"{InputCharacter}* {LineTerminator}
CommentContent       = ( [^*] | \*+ [^/*] )*

Identifier = [:jletter:] [:jletterdigit:]*
DecIntegerLiteral = 0 | [1-9][0-9]*

%%
...
```

کد اولیه

```
%state STRING

%{
  StringBuffer string = new StringBuffer();

  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }

%}

...
```

کلیدواژه‌ها

```
...
%%

<YYINITIAL> "abstract"   { return
symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"    { return symbol(sym.BOOLEAN); }

<YYINITIAL> "break"      { return symbol(sym.BREAK); }


<YYINITIAL> {
  /* identifiers */
  {Identifier}         { return symbol(sym.IDENTIFIER);}

  /* literals */
  {DecIntegerLiteral} { return symbol(sym.INT_LITERAL);}
  ...
```

کلیدواژه‌ها (ادامه)

```
...
/* literals */
{DecIntegerLiteral} { return symbol(sym.INT_LITERAL);}
\"                  { string.setLength(0);
                      yybegin(STRING); }

/* operators */
"="                 { return symbol(sym.EQ); }
"=="                { return symbol(sym.EQEQ); }
"+"                 { return symbol(sym.PLUS); }

  /* comments */
  {Comment}         { /* ignore */ }

  /* whitespace */
  {WhiteSpace}      { /* ignore */ }
}
...
```

حالت STRING

```
...
<STRING> {
  \"                  { yybegin(YYINITIAL);
                        return symbol(sym.STRINGLITERAL,
                                      string.toString()); }
  [^\n\r\"\]+         { string.append( yytext() ); }
  \\t                 { string.append('\t'); }
  \\n                 { string.append('\n'); }

  \\r                 { string.append('\r'); }
  \\"                 { string.append('\"'); }
  \\                  { string.append('\'); }

}
```

با تشکر

محدودیت‌های زبان‌های منظم

Programming Language Syntax

- Regular languages suboptimal for specifying programming language syntax
- Why? Constructs with nested syntax
  - (a+(b-c))*(d-(x-(y-z)))
  - if (x < y) if (y < z) a = 5 else a = 6 else a = 7
- Regular languages lack state required to model nesting
- Canonical example: nested expressions
- No regular expression for language of parenthesized expressions