محلل المفردات

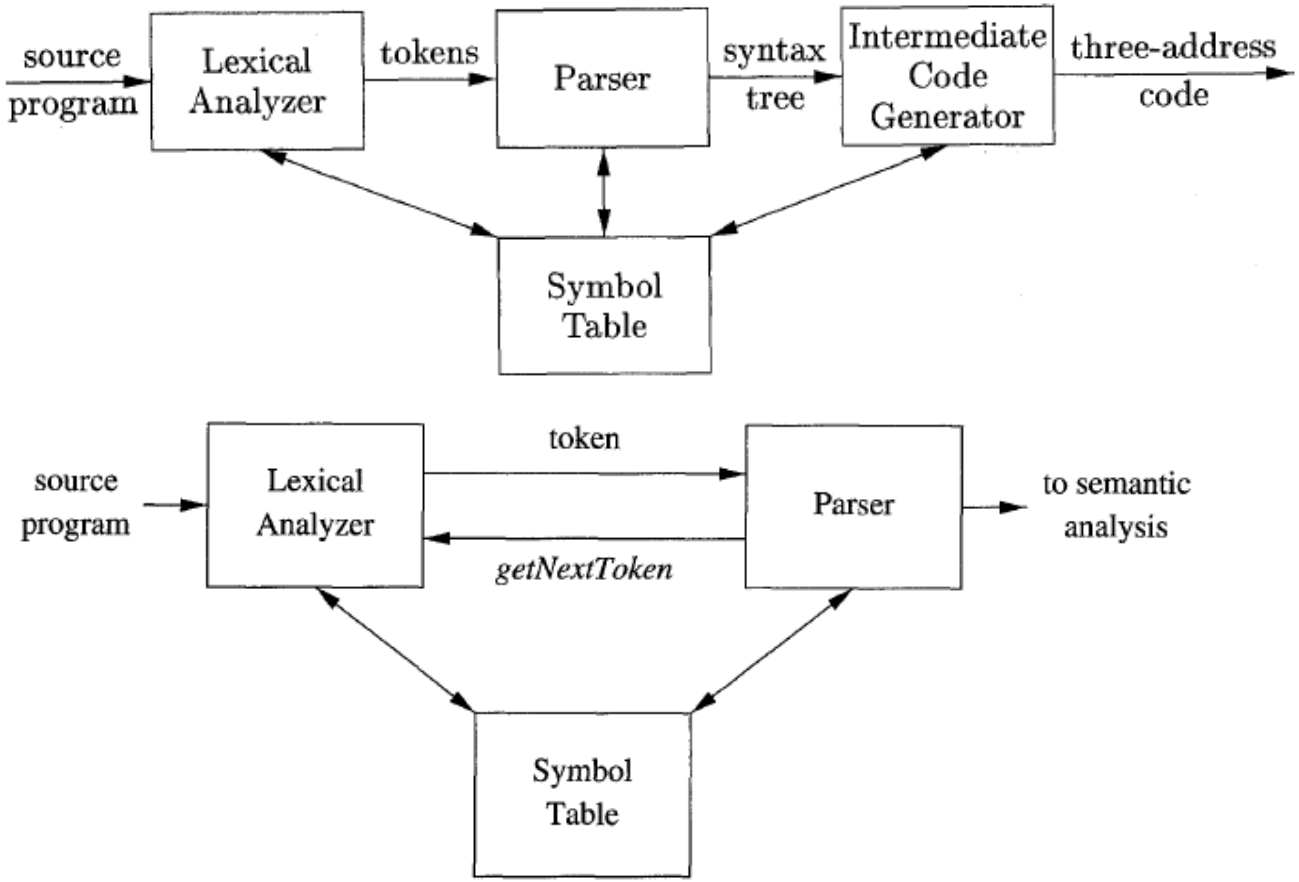
المحلل اللفظي

Lexical Analyzer

طرق بناء محلل المفردات

- الطريقة اليدوية: يجب كتابة كود للتعرف على كل مفردة من اللغة وإعادة معلومات عنها.
- توليد المحلل اللفظي: من خلال توصيف لنماذج مفردات اللغة. يوجد العديد من التطبيقات التي تعتمد على اكتشاف النماذج
  - أسهل لتعديل المحلل.
  - أسرع كون توصيف مفردات اللغة يتم في مستوى تجريد عال بينما تفاصيل الكود تقع على عاتق مولد الكود.

المحلل اللفظي في سلسلة الترجمة



دور محلل المفردات

- Scanninig: حذف التعليقات والفراغات.
- Lexical Analysis: إنتاج سلسلة من مفردات اللغة الموجودة في نص البرنامج.
  - Token: زوج من اسم المفردة (نوع الوحدة اللفظية) وقيمتها (إختيارية).
  - Patterm: كون توصيف مفردات اللغة يتم في مستوى تجريد عال بينما تفاصيل الكود تقع على عاتق مولد الكود.
    - من أجل الكلمات المحجوزة: النموذج هو سلسلة المحارف التي تشكل الكلمة.
    - من أجل المعرفات: النموذج هو بنية أعقد تتطابق مع مجموعة كبيرة من الكلمات.
  - Lexeme: سلسلة محارف من الكود المصدري تطابق نموذج مفردة، يكتشفها المحلل اللفظي كغرض من تلك المفردة.

Examples of tokens

Token	Sample Lexemes	Description
const	const	const
if	if	if
else	else	else
relation	<, <=, ==, >, >=	< or <= or == or …..
id	pi, count, D5	letter followed by letters and digits
unm	3.14, 0, 6.02E23	any numeric constant
literal	“hello world”	any characters between “ and “ except “

- Ex: const pi = 3.14;

Token = id Lexeme = pi

Tokens

Token	pattern
Keywords	the keyword itself
Operators	relation operators themselves.
Identifiers	regular expression
Constants: Numbers & Literals	regular expression
Punctuation symbols as parentheses, commas, semicolons	Symbols themselves

Regular Expressions

- ***Ex1***: identifiers are strings of letters, digits, and underscores:
  - *letter_ → A|B|…|Z|a|b|…|z|_*
  - *digit → 0|1|2|…|9*
  - *Id → letter_(letter_|digit)**
- ***Ex2***: Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.
  - *digit → 0|1|2|…|9*
  - *digits → digit digit**
  - *optionalFraction → . digits | ε*
  - *optionalExponent → (E (+ | - | ε) digits) | ε*
  - *number → digits optionalFraction optionalExponent*

Extensions of Regular Expressions
Lex (Unix)

- *Operator* $^+$: one or more instances.
- *Operator* $^?$: zero or one instance.
- *Character classes*:
  - *[abc] = a|b|c*
  - *[a-z] = a|b|…|z*

  - ***Ex:*** *Id → [A-Za-z] [A-Za-z0-9]**
  - ***Ex:*** *Id → [A-Za-z_] [A-Za-z0-9_]**

Extensions of Regular Expressions

- ***Ex1***: identifiers are strings of letters, digits, and underscores:
  - *letter_ → [A-Za-z_]*
  - *digit → [0-9]*
  - *Id → letter_(letter_|digit)**
- ***Ex2***: Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.
  - *digit → [0-9]*
  - *digits → digit^{+}*
  - *number → digit^{+} (. Digits)$^{?}$ (E[+-]$^{?}$digits)$^{?}$*

Extensions (Flex) of Regular Expressions

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}\,s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a[1,5]`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Recognition of Tokens

- ***Language Rules ….***

- *stmt → if expr then stmt*
  | if *expr then stmt else stmt*
  | ε
- *expr → term relop term*
  | *term*
- *term → id*
  | number

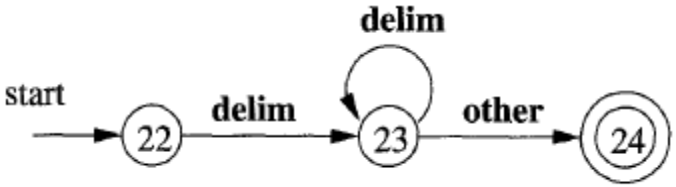
- ***Patterns …***

- *ws → (blank I tab | newline)$^{+}$*
- *digit → [0-9]*
- *digits → digit^{+}*
- *number → digit^{+} (. Digits)$^{?}$ (E[+-]$^{?}$digits)$^{?}$*
- *letter → [A-Za-z]*
- *id →letter(letter|digit)**
- *if → if*
- *then → then*
- *else → else*
- *relop → < | > | >= | <= | = | <>*

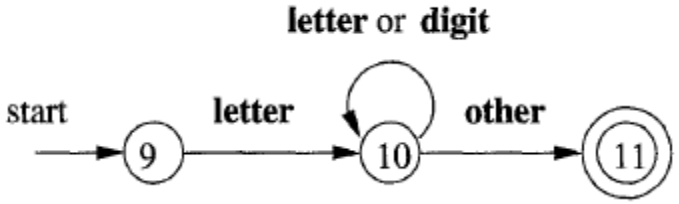
Recognition of Tokens

Lexemes	Tokens	Token attributes
Any ws	-	-
if	if	-
then	then	-
else	else	-
Any id	id	Pointer to entry table
<	relop	LT
<=	relop	LE
>	relop	GT
>=	relop	GE
=	relop	EQ
<>	relop	NE

Recognition of Tokens

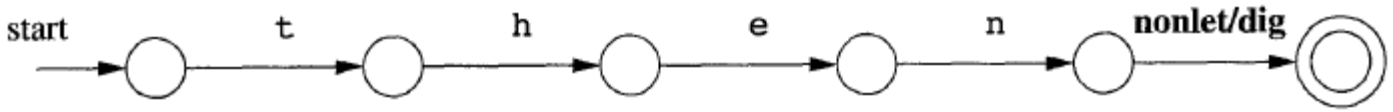


Recognition of white spaces

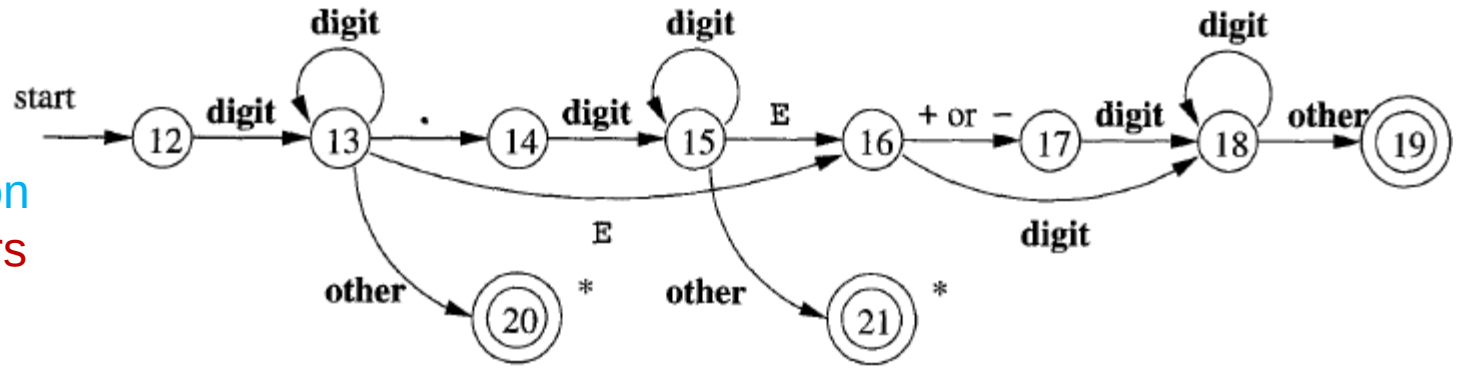


Recognition of identifiers

Recognition Of then



Recognition Of numbers

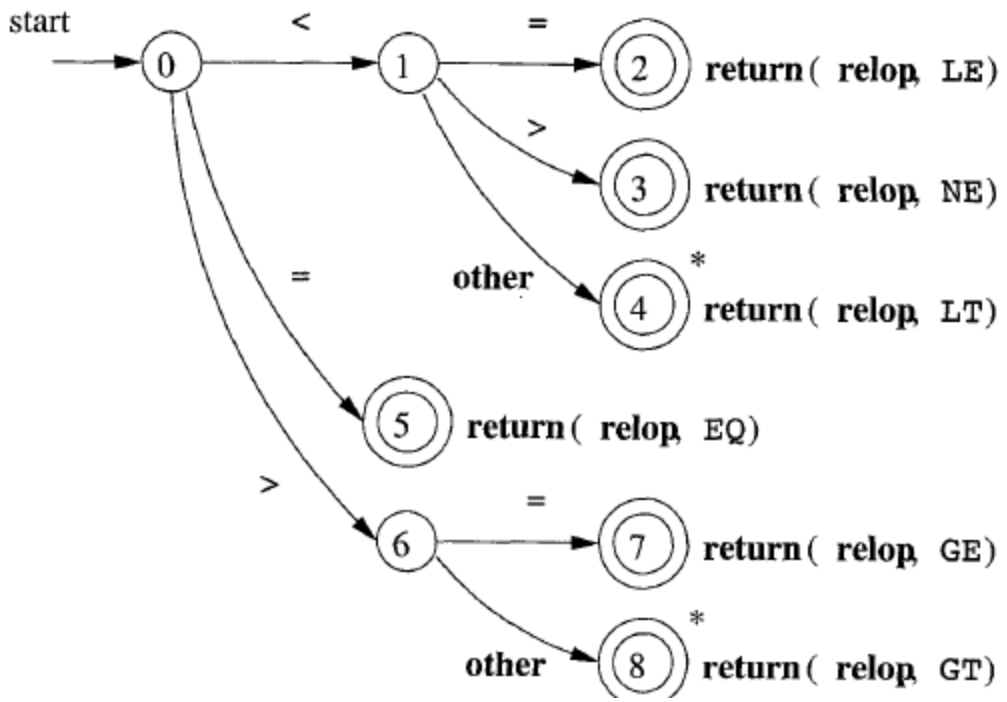


Recognition of Tokens

```
TOKEN getRelop() {
   TOKEN retToken = new(REL0P);
    while(1) {
        switch(state) {
             case 0: c = nextchar();
              if ( c == '<' ) state = 1;
              else if ( c == '=' ) state = 5;
              else if ( c == '>' ) state = 6;
              else  fail( ) ;
              break;

              case 1: . . .

              case 8: retract();
                      retToken.attribute = GT;
                      return(retToken);
               break;
          }
     }
}
```



Recognition of relation

Next start state

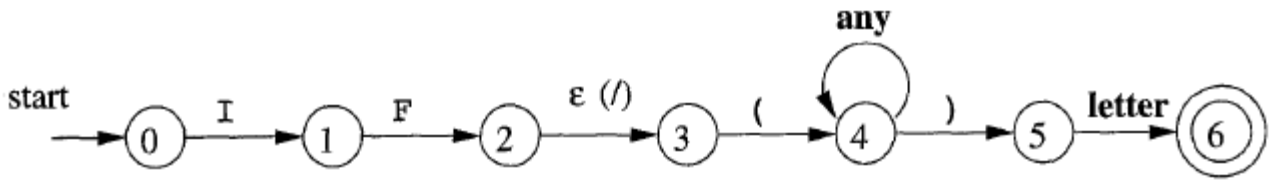
```
state ← 0; start ←0;
lexcal_value;
int fail() {
    forward = token_beginning;
    switch(start) {
        case 0: start ← 9; break;
        case 9: start ← 12; break;
        case 12: start ← 20; break;
        case 20: start ← 25; break;
        case 25: recover;
     }
}
```

Code of Lexical Analyzar …

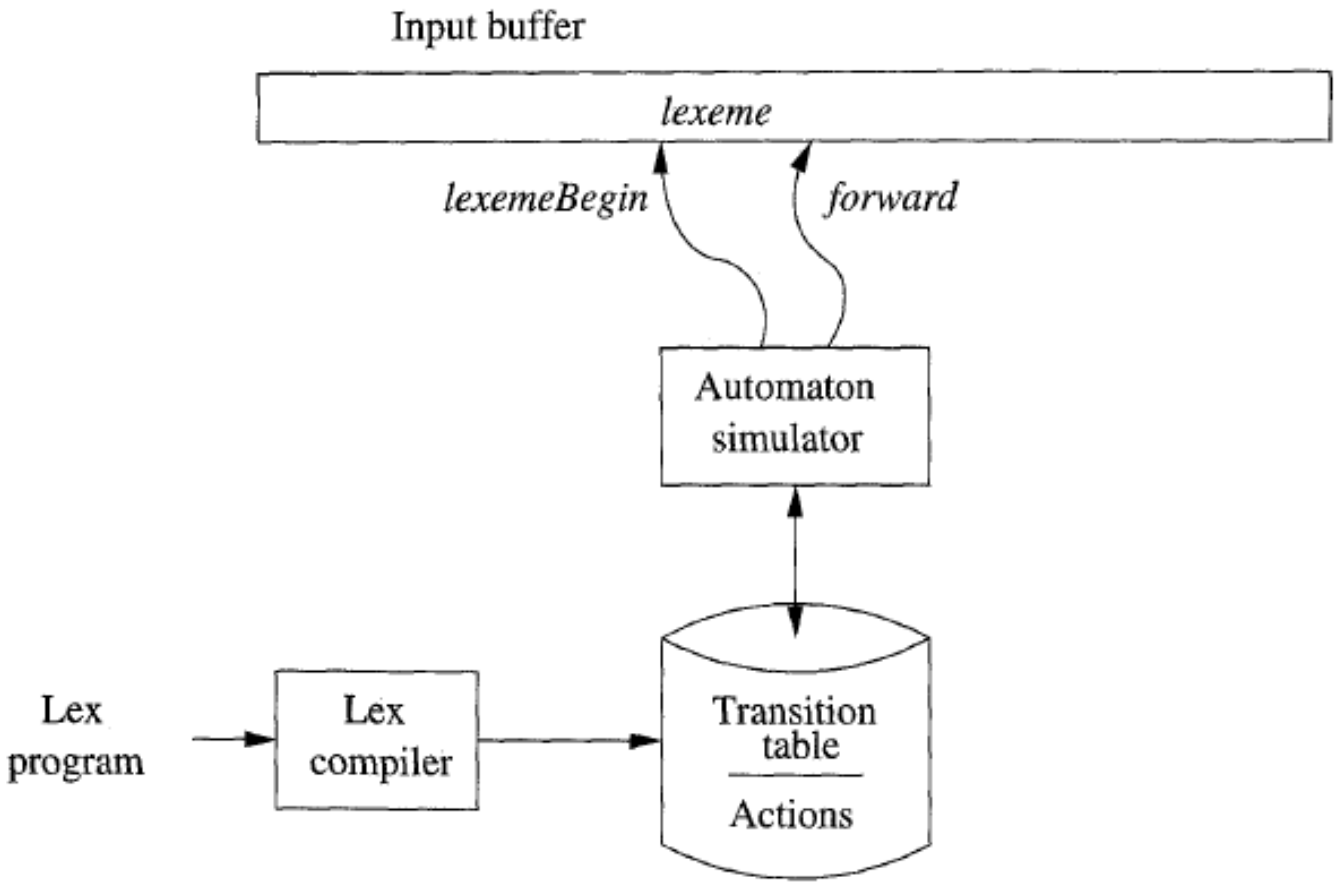
```
token nexttoken() {
    while(1) {
        switch(state) {
            case 0 : c ← nextchar();
             if (c== blank || c == tab || c == newline) {
                state ← 0
                lexeme_beginning++;
             }
             elseif (c == '<') state ← 1;
             elseif (c == '=') state ← 5;
             elseif (c == '>') state ← 6;
             else state = fail();
             break;
             //….
             case 9: c← nextchar();
              //…
```

- في حال وجود عدة حالات تطابق:
  - نختار الـ lexeme الأطول.
  - وعند تساوي طول أكثر من lexeme نختار المعرف أولاً...

Conditional recognition of lexeme

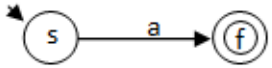
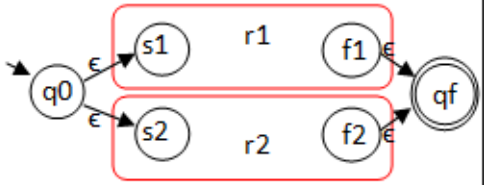
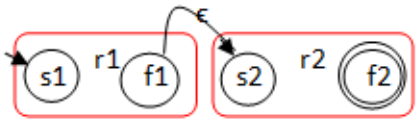
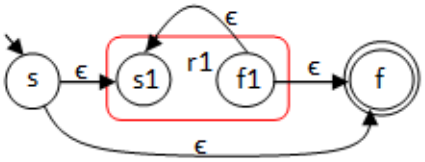


Generation of Lexical Analyzer



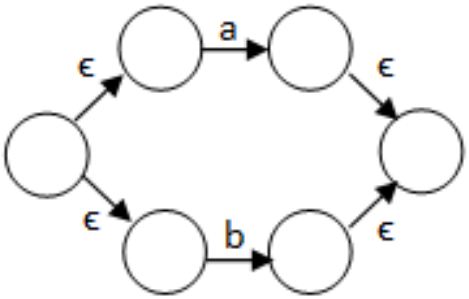
Transformation
Regular expressions to ε-NFA

- RE operations

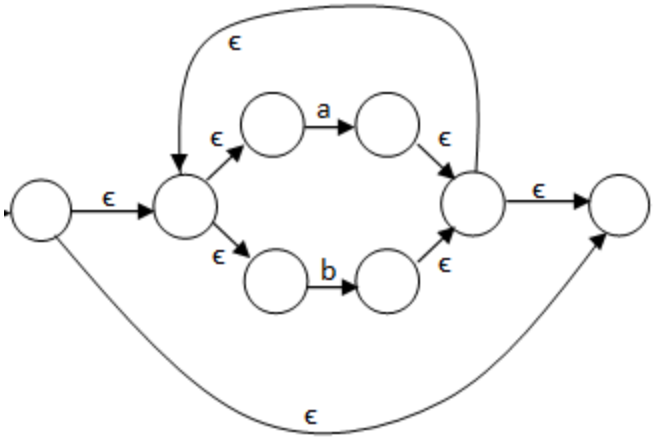
Regular Expression	Language	Finite automata
Φ	{}	
ϵ	{ϵ}	
a	{a}	
r1 + r2	L(r1+r2)=L(r1) + L(r2)	
r1.r2	L(r1.r2)=L(r1).L(r2)	
r1*	L(r1*) = L(r1)*	

RE ➔ ε-NFA

- Transform a.(a+b)*.b.b to ε-NFA
- (a+b)

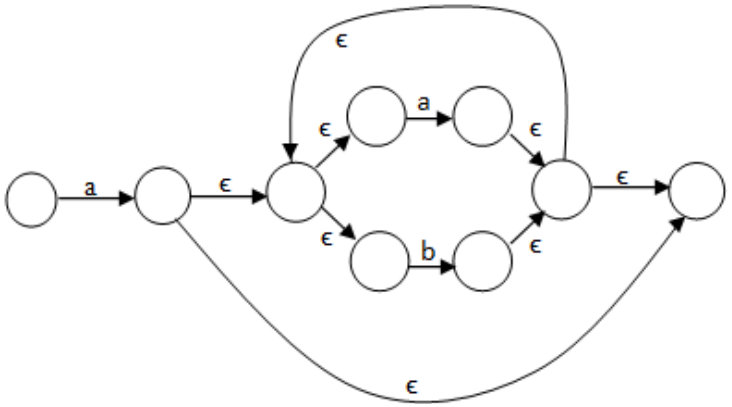


- (a+b)*

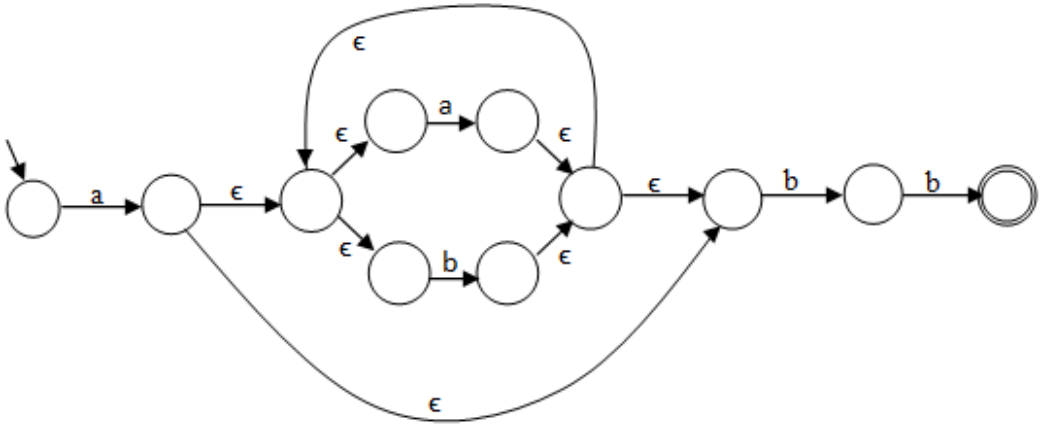


RE ➔ ε-NFA

- Transform a.(a+b)*.b.b to ε-NFA
- a.(a+b)*



- a.(a+b)*.b.b



Simulation ε-NFA …

s ← ε-closure({s0})
a ← nextchar
while(a != eof) do
    s ← ε-closure (move(s, c))
    c ← nextchar
end while
if s in finite states then
    return ‘yes’
else
    return ‘no’

move	a	b	ε
0	{0, 1}	{0}	Φ
1	Φ	{1}	Φ
2	Φ	Φ	Φ
3	Φ	Φ	Φ

Transition Table

Simulation NFA … (ε-closure)

```
push all states of T onto stack
initialize E- closure(T) to T
while ( stack is not empty ) {
    pop t, the top element of stack;
    for ( each state u with an edge from t to u labeled ε ){
        if ( u is not in e-closure(T) ) {
            add u to e-closure(T);
            push u onto stack
        }
    }
}
```

Transform ε-NFA to DFA

initially, e-closure(so) is the only state in Dstates, and it is unmarked;
while (there is an unmarked state T in Dstates) {
    mark T;
    for (each input symbol a) {
        U = ***E- closure(move(T, a));***
        if (*U is not in Dstates*)
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}

simulation DFA …

```
s ← s0
c ← nextchar
while(c != eof) do
    s ← move(s, c)
    c ← nextchar
end while

if s in finite states then
    return ‘yes’
else
    return ‘no’
```

Thanks