Regular Pattern Matching

Maxime Crochemore

King’s College London
http://www.dcs.kcl.ac.uk/staff/mac/

Regular Pattern Matching

Problem

locate segments of a text *t*
described by a regular expression *r*

text *t*

segment *x*
described by *r*

Applications

text editing : vi, emacs , sed, ed, **...**
search : grep, egrep , **...**
translation : lex, sed , **…**
languages : awk, perl, **…**
compression : compress, gzip, **…**
virus detection, etc.

GREP

```
unix > grep toto t.txt
```

produces lines of **`t.txt`**
containing the word **`toto`**

Applications

searching files by contents
checking the content of a file

Versions

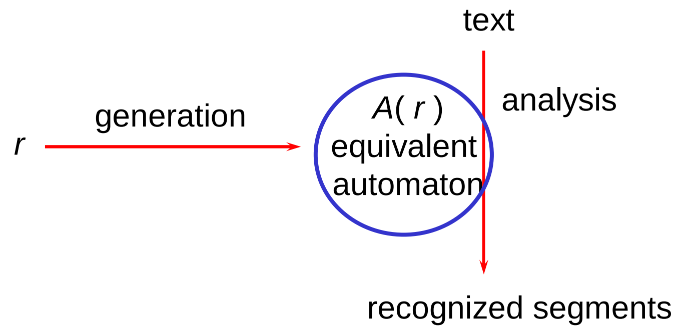
`egrep, fgrep, ...`

Schema of algorithm

Two phases

1 generation of automaton *A*(*r*)
2 text analysis with *A*(*r*)

phases can be integrated (pipelining)



Expression / language

Regular expression r	Corresponding language $L(r)$
a , **b** , …	{**a**} , {**b**} , … **a**, **b** $\in$ **A**
$\emptyset$, ε	$\emptyset$, {empty word}
[u , v	$L(u)$, $L(v)$]
(u **)**	$L(u)$
u **+** v	$L(u) \cup L(v)$
uv	$\{ xy \mid x \in L(u), y \in L(v) \}$
u*****	$\{ x_1x_2 \ldots x_k \mid k \geq 0, x_i \in L(u) \}$

`1(0+1)*0`	{binary representations of even integers}
`(a*b)*a*`	{finite strings of **a** and **b**}
`(c*l)*c*f`	{textual files}

Description of expressions

A grammar of regular expressions

$E \to T \mid T$ '+' E
$T \to F \mid F\,T$
$F \to S\,G$
$G \to \varepsilon \mid$ '*' G
$S \to$ '**a**' | '**b**' | '$\emptyset$' | 'ε' | '(' E ')'

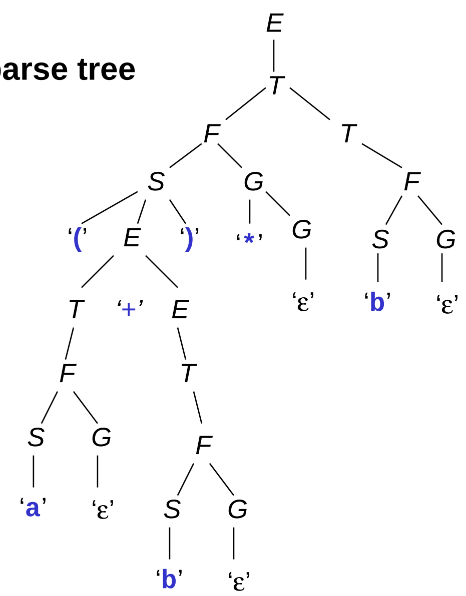
Variables

E expression   *T* term
F factor   *S* simple factor   *G* for stars

`(a+b)*b` {finite strings of **a** and **b** ending with **b**}

Analysis of (a+b)*b

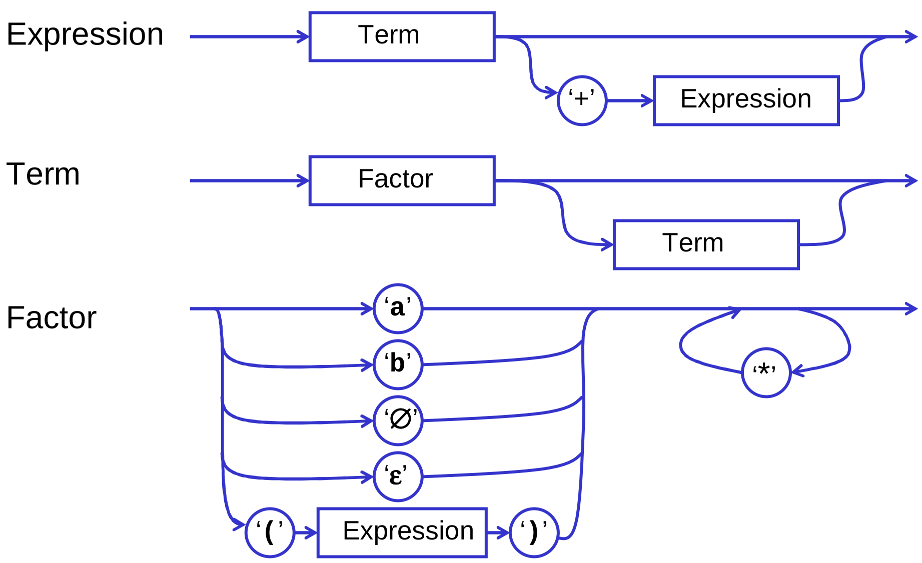
parse tree



Grammar

$E \rightarrow T$ | T '+' E
$T \rightarrow F$ | $F\ T$
$F \rightarrow S\ G$
$G \rightarrow \varepsilon$ | '*' G
$S \rightarrow$ '**a**' | '**b**' | '∅' | 'ε' | '(' E ')'

Syntactic diagrams



Parsing algorithm

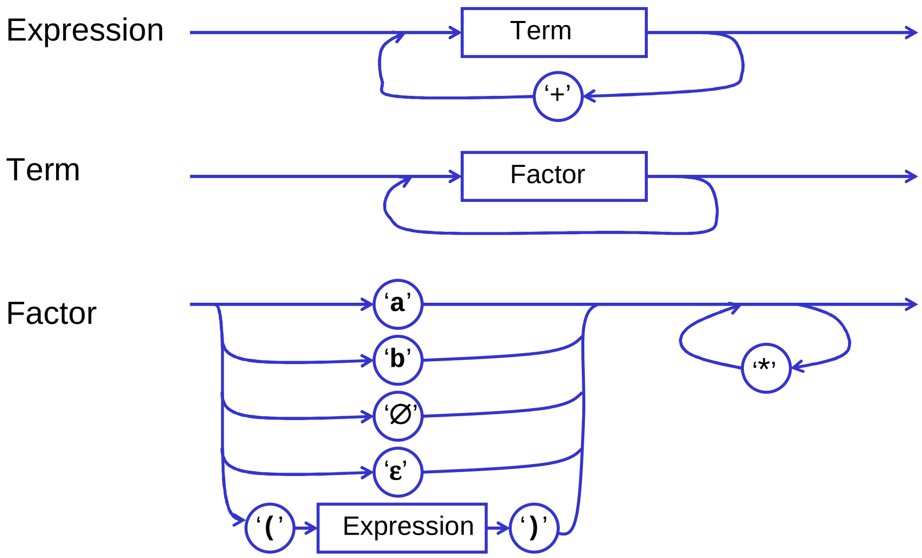
```
character car;  /* next character, global */

Parser(regular expression r){
      car ← first character of r ;
      Expression();
      if(car ≠ end of expression)
            erreur();
}
Expression(){
      Term();
      if(car = '+'){
            car ← next character;
            Expression();
      }
}
Term(){
      Factor();
      if(car ∈ {'a', 'b', '∅', 'ε', '('})
            Term();
}
```


Parsing algorithm (followed)

```
Factor(){
      if(car ∈ {'a', 'b', '∅', 'ε'}){
            car ← next character;
      }else if(car = '('){
            car ← next character;
            Expression();
            if(car = ')')
                  car ← next character;
            else
                  error();
      }else
            error();
      while(car = '*')
            car ← next character;
}
```

Syntactic diagrams (2)



Parsing algorithm (2)

```
character car;  /* next character, global */

Parser(regular expression r){
      car ← first character of r ;
      Expression();
      if(car ≠ end of expression)
            error();
}
Expression(){
      Term();
      while(car = '+'){
            car ← next character;
            Term();
      }
}
Term(){
      repeat
            Factor();
      while(car ∈ {'a', 'b', '∅', 'ε', '('})
}
```

Expression / automaton

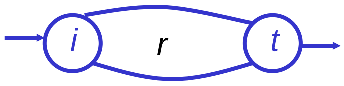
$A(\, r\,)$ automaton associated with regular expression r on the alphabet **A**

Invariants of the construction

$A(\, r\,)$ has only one initial state i
only one terminal state t
no arc enters i
no arc leaves t

Graphic representation

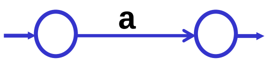
$A(\, r\,)$:



Transformation

Reg expr.	Equiv. aut.	Reg expr.	Equiv. aut.

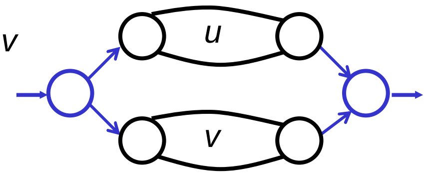
a



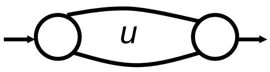
ε

∅

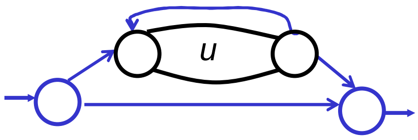
$u + v$



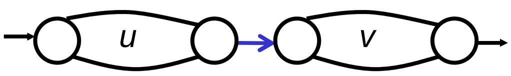
$(\, u\,)$



u*



uv



Generated automaton

Proposition

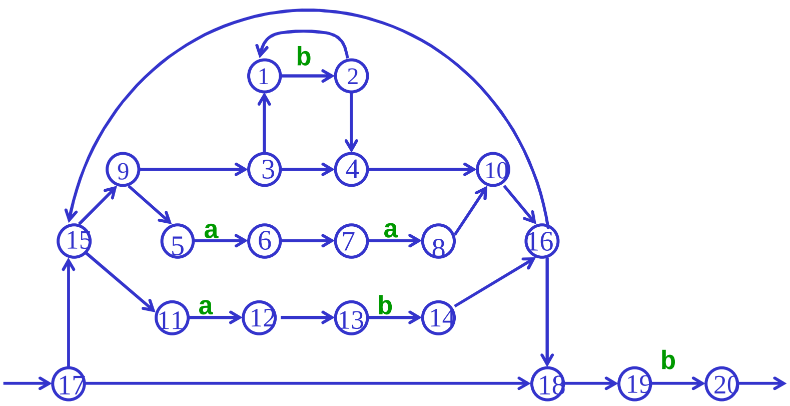
$A(r)$ accepts the language $L(r)$

Additional properties

$A(r)$ has $2.|r|$ states [not counting '(' nor ')']
output degrees are at most 2
if exactly 2: arcs have empty labels
on each arc arrive at most 2 arcs

Example

Expression : `(b* + aa + ab)*b`



is **bab** accepted?
no, but infinitely many paths starting at 17 and labelled by **bab**

Special implementation

state = index on table
table of arcs

```
struct{
      char lettre;
      state succ1, succ2;
} Trans[MAX];
```

automaton (identified to 2 states)

```
struct{
      state initial;
      state terminal;
} automaton;
```

Trans

p	**'a'**	p1	
q		q1	q2

1	'b'	2	
2		1	4
3		1	4
4		10	
5	'a'	6	
6		7	
7	'a'	8	
8		10	
9		3	5
10		16	
11	'a'	12	
12		13	
13	'b'	14	
14		16	
15		9	11
16		15	18
17		15	18
18		19	
19	'b'	20	
20			

initial = 17
terminal = 20

Translation algorithm

Production of automaton *A*(*r*)
associated with the regular expression *r*

```
Translation(regular expression r){
      car ← first character of r ;
      A ← Expression();
      if(car ≠ end of expression)
            erreur();
      else
            return A;
}
```

Translation algorithm (followed)

```
Expression(){
      A ← Term();
      while(car = '+'){
            car ← next character;
            B ← Term();
            A ← Union(A,B);
      }
      retour A;
}
Term(){
      A ← Factor();
      while(car ∈ {'a', 'b', '∅', 'ε', '('}){
            B ← Factor();
            A ← Product(A,B);
      }
      return A;
}
```


Translation algorithm (followed)

```
Factor(){
      if(car ∈ {'a', 'b', '∅', 'ε'}){
            A ← Elementary-Automaton(car);
            car ← next character;
      }else if(car = '('){
            car ← next character;
            A ← Expression();
            if(car = ')')
                  car ← next character;
            else
                  error();
      }else
            error();
      while(car = '*'){
            A ← Star(A);
            car ← next character;
      }
      return A ;
}
```

Recognition (with non-deterministic automaton)

Function

`boolean accept( automaton A, string x )`

Value is true iff there exists a path labelled by *x* from *A.initial* to *A.terminal*

Programming:

backtracking (mind cycles in the automaton!)

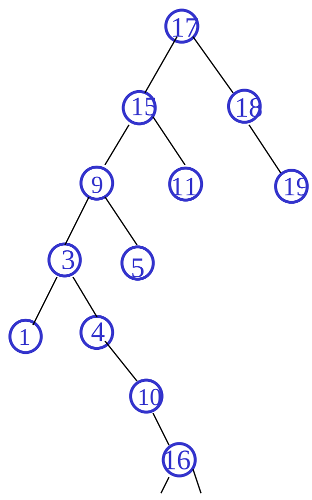
dynamic programming (storing possible states)

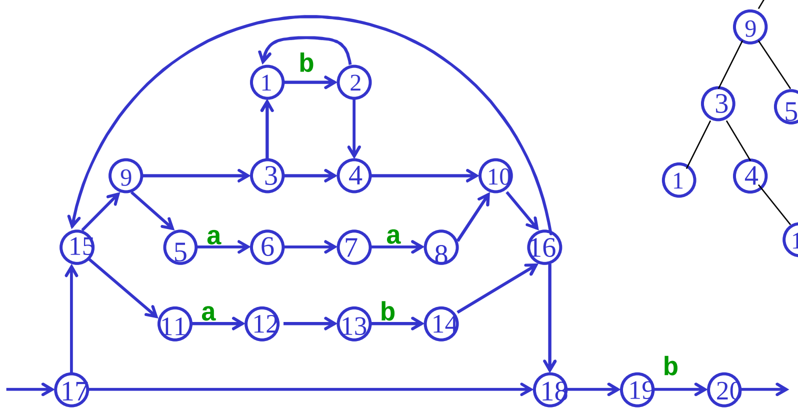
Closure

closure(*p*) = { states accessible from *p* using paths having empty labels }

Closure

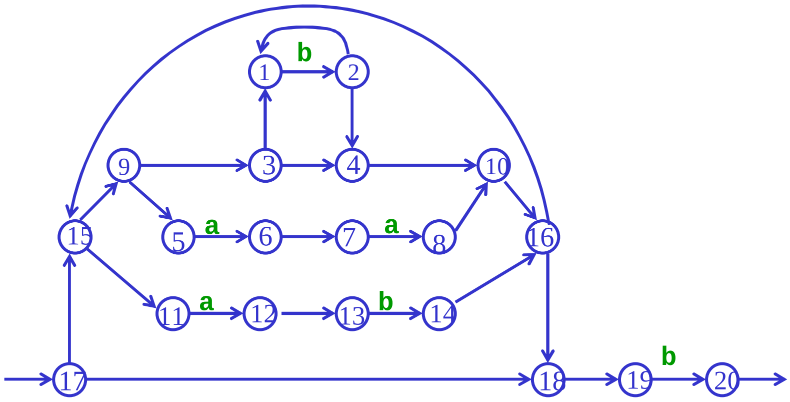
closure(17) = { 1,3,4,5,9,10,11,15,16,17,18,19 }





Recognition

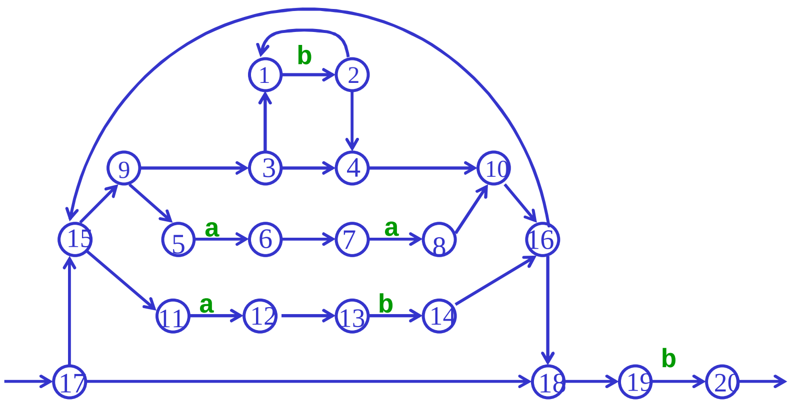
Expression : **(b* + aa + ab)*b** **Recognition of bab**



Initial state 17
Closure {1, 3, 4, 5, 9, 10, 11, 15, 16, 17, 18, 19}

Recognition (followed)

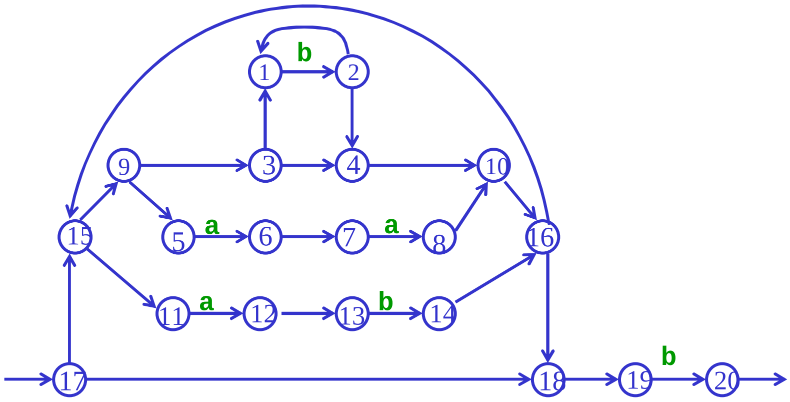
Expression : **(b* + aa + ab)*b** **Recognition of bab**



{1, 3, 4, 5, 9, 10, 11, 15, 16, 17, 18, 19}
Transition by b {2, 20}
Closure {1, 2, 3, 4, 5, 9, 10, 11, 15, 16, 18 , 19, 20}

Recognition (followed)

Expression : **(b* + aa + ab)*b** **Recognition of bab**



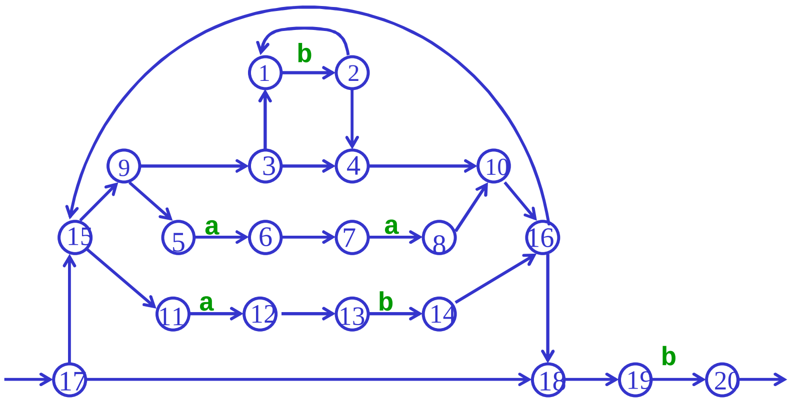
{1, 2, 3, 4, 5, 9, 10, 11, 15, 16, 18, 19, 20}

Transition by a {6, 12}

Closure {6, 7, 12, 13}

Recognition (followed)

Expression : **(b* + aa + ab)*b** **Recognition of bab**



{6, 7, 12, 13}

Transition by b {14}

Closure {1, 3, 4, 5, 9, 10, 11, 14, 15, 16, 18, 19}

bab not recognized since 20 not reached

Recognition algorithm

```
boolean accept(automaton A, string x){
     set E;
     E ← closure(A.initial);
     while(not end of x){
          car ← next letter of x;
          E ← closure(transition(E,car));
     }
     if (A.terminal ∈ E) return(true);
     else return(false);
}
```

On-line treatment of x: buffer of only one letter on x.
Running time: $O(\#\{\text{states of } A\} \,.\, |x|)$
(with adequate implementation of E)

Regular pattern matching

Pattern specified by regular expression r
$A(r)$ automaton associated with r

Pattern in text

text t

segment x
accepted by $A(r)$, $x \in L(r)$

Searching principles

like recognition except:
- initial state is added at each step
- search terminates as soon as terminal state is reached

Matching algorithm

```
boolean regular-matching(automaton A, text t){
     set E;
     E ← closure(A.initial);
     while(not end of t){
          car ← next letter of t;
          E ← closure({A.initial}∪transition(E,car));
          if(A.terminal ∈ E) return(true);
     }
     return(false);
}
```

On-line treatment of the text: buffer of only one letter on *t*

Running time: O(#{states of *A*} . | *t* |)

(with adequate implementation of *E*)

Time-space tradeoff

regular expression *r*

non-deterministic automaton *A*
(state number $\leq 2\,|\,r\,|$)

determinisation

deterministic automaton
(state number = $O(2^{|r|})$)

search *t*
In time O(| *t* |)

simulation of deterministic automaton

search *t*
In time O(| *r* | . | *t* |)

	determinisation	*simulation*
Search time	$O(\mid t \mid)$	$O(\mid r \mid . \mid t \mid)$
space	$O(2^{\mid r \mid})$	$O(\mid r \mid)$

LEX

production of lexical parsers or translators

`lex.l` specifications in Lex

Lex compiler

`lex.yy.c` lexical parser in C

C compiler

`a.out` lexical parser

source → lexical parser → lexical units (translation)

Specifications : declarations
%%
translation rules
%%
auxiliary procedures

```
% {/* definitions of constants */
         LT, LE, EQ, NEQ, GT, GE,
         IF, THEN, ELSE, ID, NB, RELOP
% }
/* definitions of regular expressions */
spacer      [ \t\n]
bl          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {lettre}+({lettre}|{chiffre})*
number      {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?
%%
{bl}        {/* no action, no return */ }
if          {return (IF) ; }
then        {return (THEN) ; }
else        {return (ELSE) ; }
{id}        {yylval = RangerId ( ) ; return (ID) ; }
{number}    {yylval = RangerNb ( ) ; return (NB) ; }
"<"         {yylval = LT ; return (RELOP) ; }
"<="        {yyval = LE ; return (RELOP) ; }
"="         {yyval = EQ ; return (RELOP) ; }
"<>"        {yyval = NEQ ; return (RELOP) ; }
">"         {yyval = GT ; return (RELOP) ; }
">="        {yyval = GE ; return (RELOP) : }
```



```
%%
StoreId(){
      /* procedure that store in the table of symbols
      a lexical unit which first letter is pointed by
      yytext and which length is yyleng;
      returns a pointer on table entry */
...
}

StoreNb(){
      /* similar procedure to store a number (constant) */
...
}
```