Pattern Matching

Some of these lecture slides have been adapted from:

- *Algorithms in C*, Robert Sedgewick.

Pattern Matching

Goal. Generalize string searching to incompletely specified patterns.

Applications.

- Test if a string or its substring matches some pattern.
  - validate data-entry fields (dates, email, URL, credit card)
  - text filters (spam, NetNanny, Carnivore)
  - computational biology
- Parse text files.
  - given web page, extract names of all links (web crawling, indexing, and searching)
  - Javadoc: automatically create documentation from comments
- Replace or substitute some pattern in a text string.
  - text-editor
  - remove all tags in web page, leaving only content

Pattern Matching

Goal. Generalize string searching to incompletely specified patterns.

Text. N characters.

Pattern. M character REGULAR EXPRESSION.

- Compact and expressive notation for describing text patterns.
- Algorithmically interesting.
- Easy to implement.

Matching. Does the text match the pattern?

Search. Find a substring of the text that matches the pattern.

Search all. Find all substrings of the text that match the pattern.

Review of Regular Expressions

Theoretician. Language accepted by FSA.

Programmer. Compact description of multiple strings.

You. Practical application of core CS principles.

Concatenate.

- `abcda` `abcda`

Logical OR.

- `a + b` `a, b`
- `(a + cc)(b + d)` `ab, ad, ccb, ccd`

Closure.

- `a*` `ε, a, aa, aaa, aaaa, aaaaa, …`
- `ca*b` `cb, cab, caab, caaab, caaaab, …`
- `c(a + bb)* d` `cd, cad, cbbd, caad, cabbd, caaad, …`

Pattern Matching and You

Broadly applicable programmer's tool.

- Many languages support extended regular expressions.
- Built into Perl, PHP, Python, JavaScript, emacs, egrep, awk.

Find any 11+ letter words in dictionary that can be typed by using only top row letters, followed by bottom row letters.

```
egrep '^[qwertyuiop]*[zxcvbnm]*$' /usr/dict/words |
egrep '...........'
```

```
perl -ne 'print if /^[qwertyuiop]*[zxcvbnm]*$/' /usr/dict/words |
perl -ne 'print if /.........../'
```


FSA and RE

Kleene's theorem (1956). FSA and RE describe same languages.

Possible grep implementation.

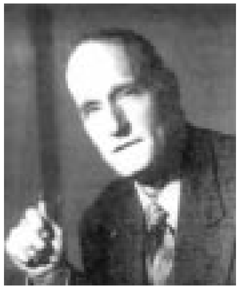
- Build FSA from RE.
- Write C program to simulate FSA.
- Performance barrier: FSA can be exponentially large.

Actual grep implementation.

- Build nondeterministic FSA from RE.
- Write C program to simulate NFSA.

Essential paradigm in computer science.

- Build intermediate abstractions.
- Pick the right ones!



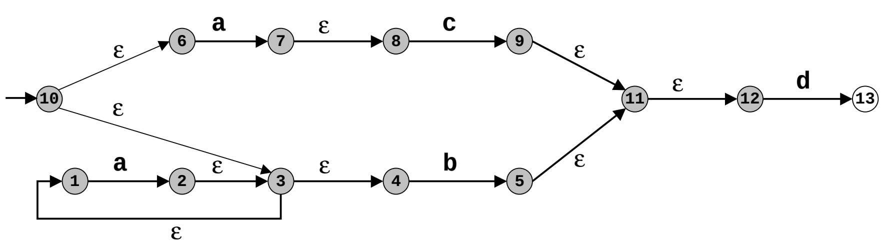
Stephen C. Kleene
(1909 - 1994)

Review of NFSA

A nondeterministic FSA.

- 0, 1, or 2 arcs leaving a state, each with same label.
- ε - transitions allowed, but no ε - cycles.

Note: this restricted form is no loss of generality.



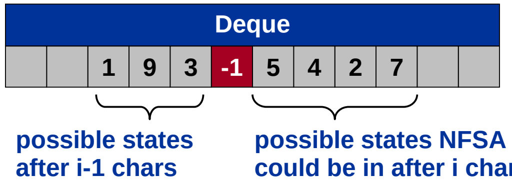
`(a*b + ac)d`

Simulating an NFSA

Brute force. Try all possible paths ⇒ exponential time.

Better idea. Keep track of all possible states NFSA could be in after reading in first i characters.

- Use a deque (double-ended queue).
  - can push/pop like stack, enqueue like queue



- Pop state v.
  - if label of arc v→w is ε, push state w
  - if current character matches label, enqueue state w
  - if mismatch, ignore

NFSA Simulator

nfsa()

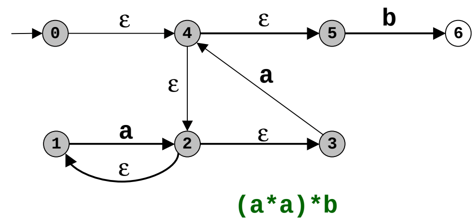
```
#define SCAN -1
#define EPS  ' '
#define MATCHSTATE 0

int match(char a[]) {
   int j = 0, state = next1[0];
   DQinit();
   DQput(SCAN);
   while(state != MATCHSTATE) {
      if (state == SCAN)          { DQput(scan); j++;     }
      else if (ch[state] == a[j]) { DQput(next1[state]);  }
      else if (ch[state] == EPS)  { DQpush(next1[state]);
                                    DQpush(next2[state]); }
      if (DQisempty() || a[j] == '\0') return 0;
      state = DQpop();
   }
   return j;
}
```


Performance Gotcha

Major performance bug if not careful.

- Simulate input aaaaaaaaaab on NFSA.



Deque

	4	-1								
	2	-1								
	2	-1								
1	3	-1								
	3	-1	2							
		-1	2	4						
			2	4	-1					
		1	3	4	-1					
			3	4	-1	2				
				4	-1	2	4			
					...					
					-1	2	4	2	4	
						2	4	2	4	-1

- Duplicate states allowed on deque ⇒ exponential growth!

Easy fix.

- Disallow duplicate states on same side of deque.
- Keep "existence array" of states currently on each side of deque.

Build NFSA from RE

Goal: build NFSA from RE.

First challenge: Is expression a legal RE?

- Use context free language to describe RE.

```
Start : <expr>

<expr>  ←  <term>
<expr>  ←  <term> + <expr>

<term>  ←  <fctr>
<term>  ←  <fctr><term>

<fctr>  ←  c
<fctr>  ←  c*
<fctr>  ←  (<expr>)
<fctr>  ←  (<expr>)*
```


Parse Tree

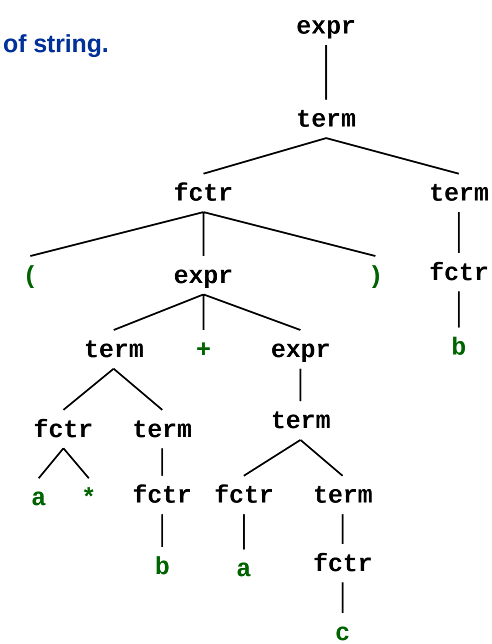
Parse tree: grammatical structure of string.
Parser: construct tree.
Example: `(a*b + ac)d`.

```
Start : <expr>

<expr>  ←  <term>
<expr>  ←  <term> + <expr>

<term>  ←  <fctr>
<term>  ←  <fctr><term>

<fctr>  ←  c
<fctr>  ←  c*
<fctr>  ←  (<expr>)
<fctr>  ←  (<expr>)*
```



Recursive Descent Parser for RE

Top-down recursive descent parser: Recursive program directly derived from CFL.

main()

```
int j = 0;              // current index
char p[MAXN + 1];       // RE pattern

void parserror(void) {
    printf("%s is not a RE.\n", p);
    exit(EXIT_FAILURE);
}

int main(void) {
    scanf("%s", p);
    expr();
    if (j != strlen(p)) parserror(p);
    return 0;
}
```


Recursive Descent Parser for RE

Definition of expression in CFL.

- <expr> ← <term>
- <expr> ← <term> + <expr>

expr()

```
void expr() {
    term();
    if (p[j] == ’+’) {
        j++;
        expr();
    }
}
```

Definition of term in CFL.

- <term> ← <fctr>
- <term> ← <fctr> <term>

term()

```
void term() {
    fctr();
    if ((p[j] == ’(’) || islower(p[j]))
        term();
}
```


Recursive Descent Parser for RE

Definition of factor in CFL.

- <fctr> ← c
- <fctr> ← c*
- <fctr> ← (<expr>)
- <fctr> ← (<expr>)*

factor()

```
void fctr() {
    if (islower(p[j])) {
        j++;
    }
    else if (p[j] == ’(’) {
        j++;
        expr();
        if (p[j] == ’)’) j++;
        else parserror();
    }
    else parserror();

    if (p[j] == ’*’) j++;
}
```


Left Recursive Parsers

Not as trivial as it first seems.

Alternate definition of expr in CFL.

- <expr> ← c
- <expr> ← <expr> + <term>

badexpr()

```
void badexpr() {
   if (islower(p[j]) j++;
   else {
      badexpr();
      if (p[j] == ’+’) {
         j++;
         term();
      }
      else parserror();
   }
}
```

Fix: use left recursive CFL.

- Avoiding infinite recursive loops is fundamental difficulty in recursive-descent parsers.
- Problem can be more subtle than example above.

Left Recursive Parsers

Example. (a*b + ac)d.

- Corresponds to parse tree.

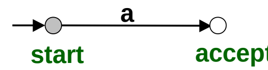
Unix

```
expr()
   term()
      fctr()
         (
         expr()
            term()
               fctr()  a  *
               term()
                  fctr() b
            +
            expr()
               term()
                  fctr() a
                  term()
                     fctr() c
         )
      term()
         fctr() d
```

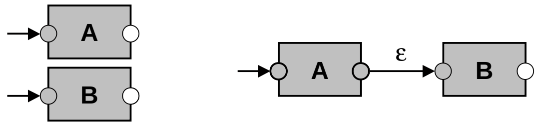

Building NFSA from RE

Each RE construct corresponds to a piece of NFSA.

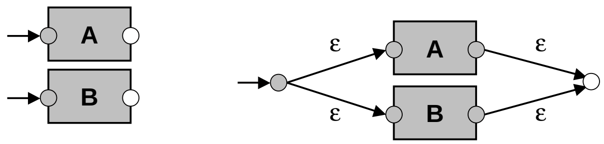
- Single character.
  - a



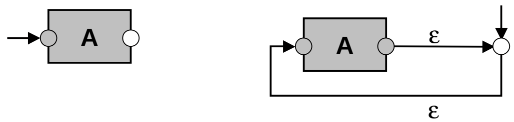
- Concatenation.
  - AB



- OR.
  - A + B



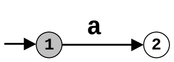
- Closure.
  - A*



Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- `(a*b + ac)d`

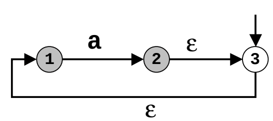


a

Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- `(a*b + ac)d`

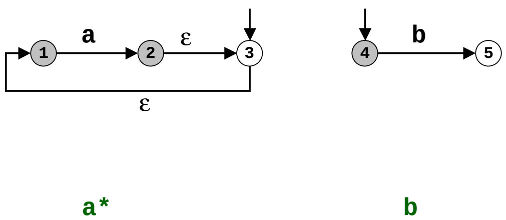


a*

Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

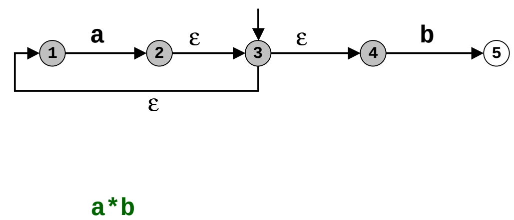
- `(a*b + ac)d`



Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

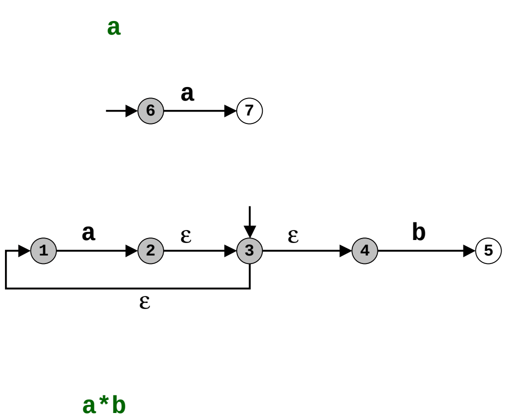
- `(a*b + ac)d`



Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

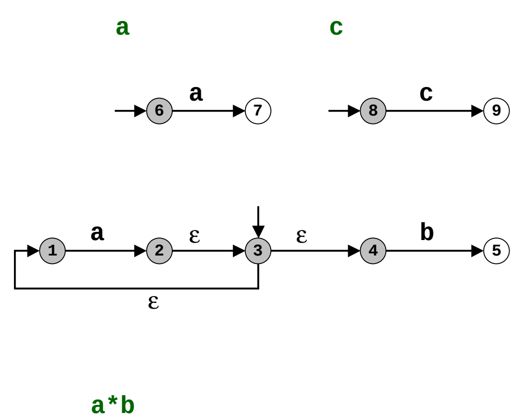
- `(a*b + ac)d`



Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- `(a*b + ac)d`

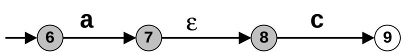


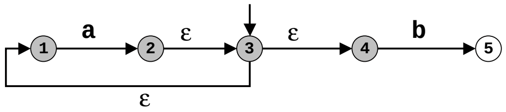
Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- (a*b + ac)d

ac



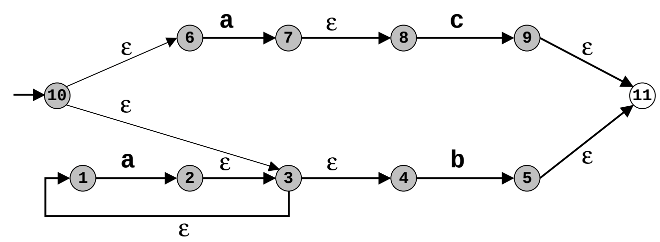


a*b

Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- (a*b + ac)d

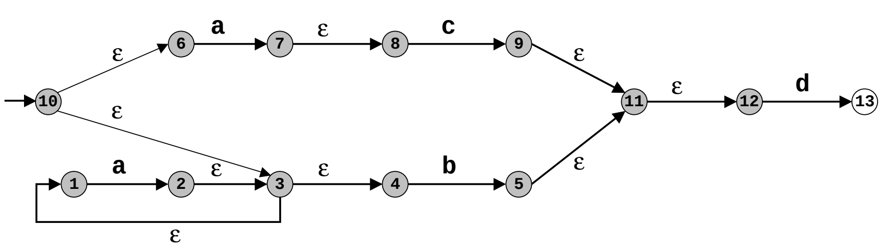


a*b + ac

Building NFSA from RE: Example

Each RE construct corresponds to a piece of NFSA.

- (a*b + ac)d

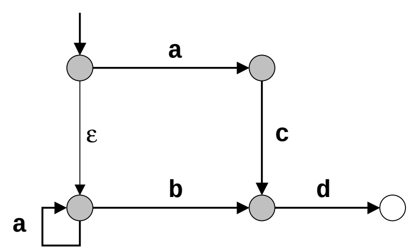


(a*b + ac)d

Building NFSA from RE: Example

Note. This construction doesn't yield simplest NFSA.

- (a*b + ac)d



Building NFSA from RE: Theory

For any RE of length M, our construction produces an NFSA with the following properties.

- No more than two arcs leave any state.
  - if two arcs, they both have label ε
- No ε - cycles.
- Exactly 1 start state, has 1 incoming arc.
- Exactly 1 accept state, has at most 1 leaving arc.
- Number of states ≤ 2M.

Proof: Apply 3 composition rules and use induction on length of RE.

- For number of states.
  - single character: 2
  - concatenation AB: |A| + |B|
  - closure A*: |A| + 1
  - OR A + B: |A| + |B| + 2

Building NFSA from RE: Practice

To build NFSA, augment parser to generate state table.

- For details: Sedgewick, Chapter 21 (Algorithms in C, 2nd edition).
  - recursive routines return index of start state
  - state = next state to be filled in
  - setstate() fills in NFSA table

expr()

```
int expr() {
    int s1, s2, start;
    start = s1 = term();
    if (p[j] == '+') {
        j++;
        start = s2 = ++state;
        state++;
        setstate(s2,   EPS, expr(), s1);
        setstate(s2-1, EPS, state,  state);
    }
    return start;
}
```


Complexity Analysis

Text. N characters.
Pattern. M character regular expression.

Matching: Does the text match the pattern?

- Build NFSA.
  - at most 2M states ⇒ O(M) time, O(M) space
- Simulate NFSA.
  - O(M) time per text character because of ε-transitions
- O(MN) time, O(M) space.

Search: Find a substring of the text that matches the pattern.

- For each offset of text, solve matching problem.
- $O(MN^2)$ time, O(M+N) space.

Perspective

Compiler. A program that translates from one language to another.

- Grep: RE ⇒ NFSA.
- C compiler: C language ⇒ machine language.

Abstract Machine	NFSA	Computer
Pattern	Word in CFL	Word in CFL
Parser	Check if legal RE	Check if legal C program
Compiler	Output NFSA	Output machine executable
Simulator	Find match	Run program in hardware