CMPT 379
Compilers

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
- Example:

```
double f = sqrt(-1);
```

T_DOUBLE	("double")
T_IDENT	("f")
T_OP	("=")
T_IDENT	("sqrt")
T_LPAREN	("(")
T_OP	("-")
T_INTCONSTANT	("1")
T_RPAREN	(")")
T_SEP	(";")

Token Attributes

- Some tokens have attributes
  - T_IDENT "sqrt"
  - T_INTCONSTANT 1
- Other tokens do not
  - T_WHILE
- *Token*=T_IDENT, *Lexeme*="sqrt", *Pattern*
- Source code location for error reports

Lexical errors

- What if user omits the space in "doublef"?
  - No lexical error, single token T_IDENT ("doublef") is produced instead of sequence T_DOUBLE, T_IDENT("f")!
- Typically few lexical error types
  - E.g., illegal chars, opened string constants or comments that are not closed

Lexical errors

- Lexical analysis should not disambiguate tokens,
  - e.g. unary op + versus binary op +
  - Use the same token T_PLUS for both
  - It's the job of the parser to disambiguate based on the context
- Language definition should not permit crazy long distance effects (e.g. Fortran)

DO 5 I = 1,5	T_DO T_INT(5) T_ID(I)
DO 5 I = 1.5	T_ID(DO5I) T_EQ

Ad-hoc Scanners

Implementing Lexers: Loop and switch scanners

- Ad hoc scanners
- Big nested switch/case statements
- Lots of getc()/ungetc() calls
  - Buffering; Sentinels for push-backs; streams
- Can be error-prone, use only if
  - Your language's lexical structure is very simple
  - The tools do not provide what you need for your token definitions
- Changing or adding a keyword is problematic
- Have a look at an actual implementation of an ad-hoc scanner

Implementing Lexers: Loop and switch scanners

- Another problem: how to show that the implementation actually captures all tokens specified by the language definition?
- How can we show correctness
- Key idea: separate the definition of tokens from the implementation
- Problem: we need to reason about patterns and how they can be used to define tokens (recognize strings).

Specification of Patterns using Regular Expressions

Formal Languages: Recap

- Symbols: a, b, c
- Alphabet : finite set of symbols $\Sigma = \{a, b\}$
- String: sequence of symbols bab
- Empty string: ε Define: $\Sigma^{\varepsilon} = \Sigma \cup \{\varepsilon\}$
- Set of all strings: Σ^* cf. *The Library of Babel*, Jorge Luis Borges
- (Formal) Language: a set of strings
  $\{ a^n b^n : n > 0 \}$

Regular Languages

- The set of regular languages: each element is a regular language
- Each regular language is an example of a (formal) language, i.e. a set of strings

  e.g. { $a^m b^n$: m, n are +ve integers }

Regular Languages

- Defining the set of all regular languages:
  - The empty set and {a} for all a in Σ^ε are regular languages
  - If L_1 and L_2 and L are regular languages, then:

    $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ (concatenation)

    $L_1 \cup L_2$ (union)

    $L^* = \cup_{i=0}^{\infty} L^i$ (Kleene closure)

    are also regular languages
  - There are no other regular languages

Formal Grammars

- A formal grammar is a concise description of a formal language
- A formal grammar uses a specialized syntax
- For example, a **regular expression** is a concise description of a regular language

  *(a|b)*abb* : is the set of all strings over the alphabet {*a*, *b*} which end in *abb*
- We will use regular expressions (regexps) in order to define tokens in our compiler,
  - e.g. lexemes for string tokens are \" (Σ\")* \"

Regular Expressions: Definition

- Every symbol of $\Sigma \cup \{ \varepsilon \}$ is a regular expression
  - E.g. if Σ = {a,b} then 'a', 'b' are regexps
- If r_1 and r_2 are regular expressions, then the core operators to combine two regexps are
  - Concatenation: r_1r_2, e.g. 'ab' or 'aba'
  - Alternation: $r_1|r_2$, e.g. 'a|b'
  - Repetition: r_1*, e.g. 'a*' or 'b*'
- No other core operators are defined
  - But other operators can be defined using the basic operators (as in lex regular expressions) e.g. a+ = aa*

Lex regular expressions

<table>
<tr><th>Expression</th><th>Matches</th><th>Example</th><th>Using core operators</th></tr>
<tr><td>c</td><td>non-operator character c</td><td><code>a</code></td><td></td></tr>
<tr><td>\c</td><td>character c literally</td><td><code>*</code></td><td></td></tr>
<tr><td>"s"</td><td>string s literally</td><td><code>"**"</code></td><td></td></tr>
<tr><td>.</td><td>any character but newline</td><td><code>a.*b</code></td><td></td></tr>
<tr><td>^</td><td>beginning of line</td><td><code>^abc</code></td><td>used for matching</td></tr>
<tr><td>$</i></td><td>end of line</td><td><code>abc$</code></td><td>used for matching</td></tr>
<tr><td>[s]</td><td>any one of characters in string s</td><td><code>[abc]</code></td><td>(a|b|c)</td></tr>
<tr><td>[^s]</td><td>any one character not in string s</td><td><code>[^a]</code></td><td>(b|c) where Σ = {a,b,c}</td></tr>
<tr><td>r*</td><td>zero or more strings matching r</td><td><code>a*</code></td><td></td></tr>
<tr><td>r+</td><td>one or more strings matching r</td><td><code>a+</code></td><td>aa*</td></tr>
<tr><td>r?</td><td>zero or one r</td><td><code>a?</code></td><td>(a|ε)</td></tr>
<tr><td>r{m,n}</td><td>between m and n occurences of r</td><td><code>a{2,3}</code></td><td>(aa|aaa)</td></tr>
<tr><td>r_1r_2</td><td>an r_1 followed by an r_2</td><td><code>ab</code></td><td></td></tr>
<tr><td>$r_1|r_2$</td><td>an r_1 or an r_2</td><td><code>a|b</code></td><td></td></tr>
<tr><td>(r)</td><td>same as r</td><td><code>(a|b)</code></td><td></td></tr>
<tr><td>r_1/r_2</td><td>r_1 when followed by an r_2</td><td><code>abc/123</code></td><td>used for matching</td></tr>
</table>

Regular Expressions: Definition

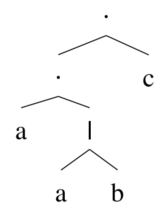
- Note that operators apply recursively and these applications can be ambiguous
  - E.g. is aa|bc equal to a(a|b)c or ((aa)|b)c?
- Avoid such cases of ambiguity - provide explicit arguments for each regexp operator
  - For convenience, for examples on this page, let us use the symbol '·' to denote the operator for concatenation
- Remove ambiguity with an explicit regexp tree
  - a(a|b)c is written as (·(·a(|ab))c) or in postfix: aab|·c·
  - ((aa)|b)c is written as (·(|(·aa)b)c) or in postfix: aa·b|c·

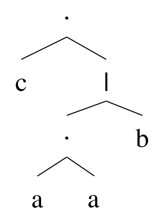
Regular Expressions: Definition

- Remove ambiguity with an explicit regexp tree

  a(a|b)c is written as
  (·(·a(|ab))c)
  or in postfix: aab|·c·

  ((aa)|b)c is written as
  (·(|(·aa)b)c)
  or in postfix: aa·b|c·
- Does the order of concatenation matter?





Equivalence of Regexps

- (R|S)|T == R|(S|T) == R|S|T
- (RS)T == R(ST)
- (R|S) == (S|R)
- R*R* == (R*)* == R* == RR*| **ε**
- R** == R*
- (R|S)T = RT|ST
- R(S|T) == RS | RT
- (R|S)* == (R*S*)* == (R*S)*R* == (R*|S*)*
- RR* == R*R
- (RS)*R == R(SR)*
- R = R|R = R**ε**

Equivalence of Regexps

- 0(10)*1|(01)*
- (01)(01)*|(01)*
- (01)(01)*|(01)(01)*|ε
- (01)(01)*|ε
- (01)*

- (RS)*R == R(SR)*
- RS == (RS)
- R* == RR*|**ε**
- R == R|R
- R* == RR*| **ε**

Regular Expressions

- To describe all lexemes that form a token as a *pattern*
  - (0|1|2|3|4|5|6|7|8|9)+
- Need decision procedure: to which token does a given sequence of characters belong (if any)?
  - Finite State Automata
  - Can be deterministic (DFA) or non-deterministic (NFA)

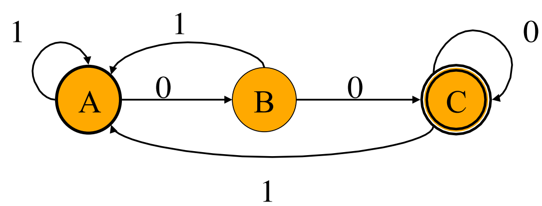
Implementing Regular Expressions with Finite-state Automata

Deterministic Finite State Automata: DFA

- A set of states S
  - One start state q_0, zero or more final states F
- An alphabet $\sum$ of input symbols
- A transition function:
  - δ: S x $\Sigma \Rightarrow$ S
- Example: $\delta(1, a) = 2$

DFA: Example

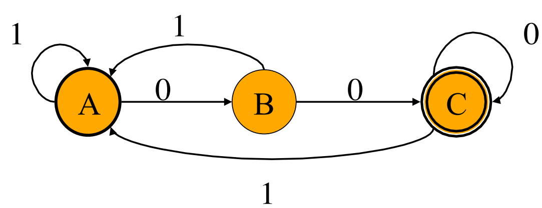
- What regular expression does this automaton accept?



A: start state
C: final state

Answer: (0|1)*00

DFA simulation

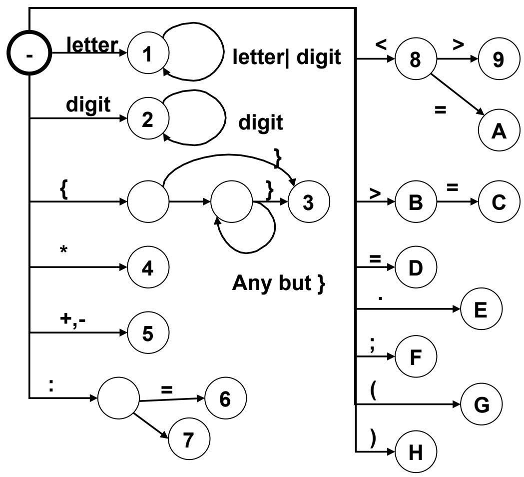


Input string: 00100

DFA simulation takes at most *n* steps for input of length *n* to return accept or reject

- Start state: A
  1. δ(A,0) = B
  2. δ(B,0) = C
  3. δ(C,1) = A
  4. δ(A,0) = B
  5. δ(B,0) = C
- no more input and C is final state: **accept**

FA: Pascal Example



Building a Lexical Analyzer

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
- NFA $\Rightarrow$ DFA
- DFAs or NFAs for all the tokens $\Rightarrow$ **Lexical Analyzer**
- Two basic rules to deal with multiple matching: **greedy match** + **regexp ordering**

Note that **greedy** means *longest leftmost match*

Lexical Analysis using `Lex`

```
%{
#include <stdio.h>
#define NUMBER     256
#define IDENTIFIER 257
%}

/* regexp definitions */
num [0-9]+

%%

{num}           { return NUMBER; }
[a-zA-Z0-9]+    { return IDENTIFIER; }

%%

int
main () {
  int token;
  while ((token = yylex())) {
    switch (token) {
      case NUMBER: printf("NUMBER: %s, LENGTH:%d\n", yytext, yyleng); break;
      case IDENTIFIER: printf("IDENTIFIER: %s, LENGTH:%d\n", yytext, yyleng); break;
      default: printf("Error: %s not recognized\n", yytext);
    }
  }
}
```


NFAs

- NFA: like a DFA, except
  - A transition can lead to more than one state, that is, δ: S x $\Sigma \Rightarrow 2^S$
  - One state is chosen non-deterministically
  - Transitions can be labeled with ε, meaning states can be reached without reading any input, that is,

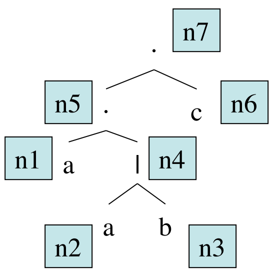
$$\delta: S \times \Sigma \cup \{ \varepsilon \} \Rightarrow 2^S$$

Thompson's construction

Converts regexps to NFA

Build NFA recursively from regexp tree

Build NFA with left-to-right parse of postfix string using a stack



Input = aab|·c·
• read a, push n1 = nfa(a)
• read a, push n2 = nfa(a)
• read b, push n3 = nfa(b)
• read |, n3=pop(); n2=pop(); push n4 = nfa(or, n2, n3)
• read ·, n4 = pop(); n1 = pop(); push n5 = nfa(cat, n1, n4)
• read c, push n6 = nfa(c)
• read ·, n6 = pop(); n5 = pop(); push n7 = nfa(cat, n5, n6)

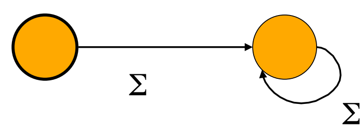
Thompson's construction

- Converts regexps to NFA
- Six simple rules
  - Empty language
  - Symbols
  - Empty String
  - Alternation (r_1 or r_2)
  - Concatenation (r_1 followed by r_2)
  - Repetition (r_1*)

Used by Ken Thompson for pattern-based search in text editor QED (1968)
To keep things simple our version is more verbose

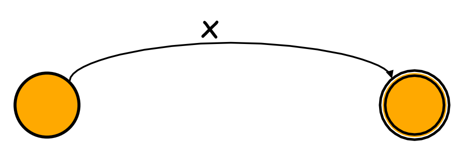
Thompson Rule 0

- For the empty language ϕ (optionally include a *sinkhole* state)



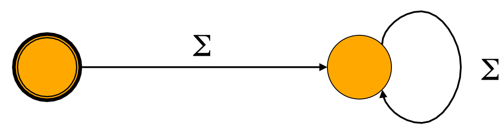
Thompson Rule 1

- For each symbol *x* of the alphabet, there is a NFA that accepts it (include a *sinkhole* state)



Thompson Rule 2

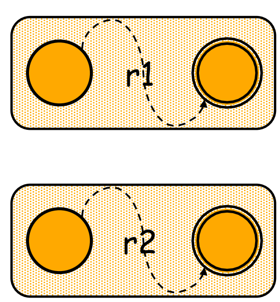
- There is an NFA that accepts only ε



9/22/11 33

Thompson Rule 3

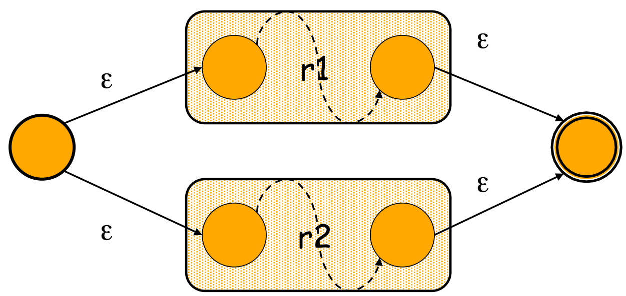
- Given two NFAs for r_1, r_2, there is a NFA that accepts $r_1|r_2$



9/22/11 34

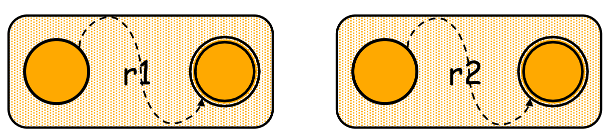
Thompson Rule 3

- Given two NFAs for r_1, r_2, there is a NFA that accepts $r_1|r_2$



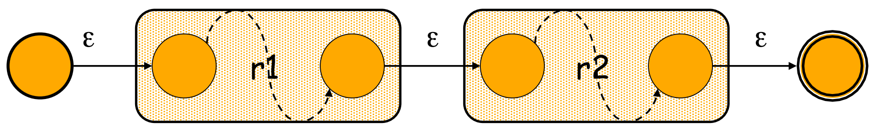
Thompson Rule 4

- Given two NFAs for r_1, r_2, there is a NFA that accepts r_1r_2



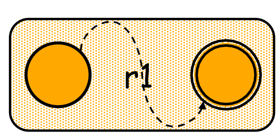
Thompson Rule 4

- Given two NFAs for r_1, r_2, there is a NFA that accepts r_1r_2



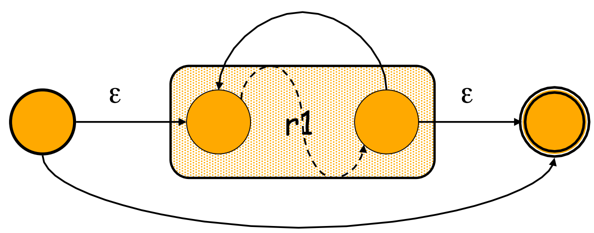
Thompson Rule 5

- Given a NFA for r_1, there is an NFA that accepts r_1*



Thompson Rule 5

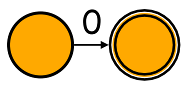
- Given a NFA for r_1, there is an NFA that accepts r_1*

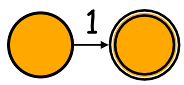


Example

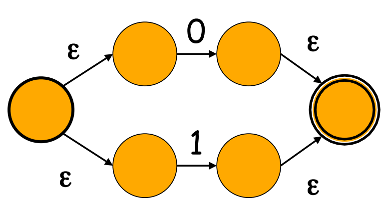
- Set of all binary strings that are divisible by four (include 0 in this set)
- Defined by the regexp: ((0|1)*00) | 0
- Apply Thompson's Rules to create an NFA

Basic Blocks 0 and 1

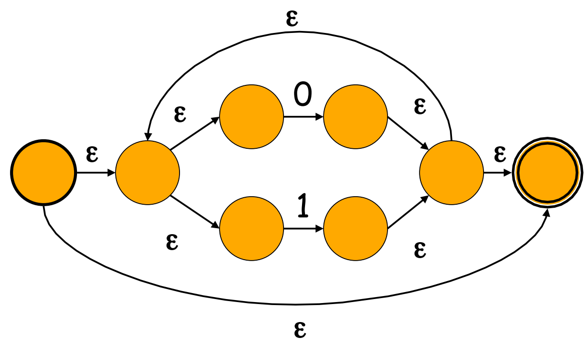
- 0 

- 1 

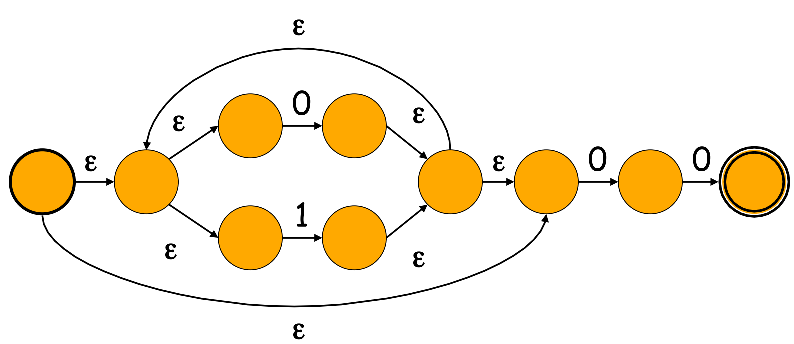
(this version does not report errors: no *sinkholes*)



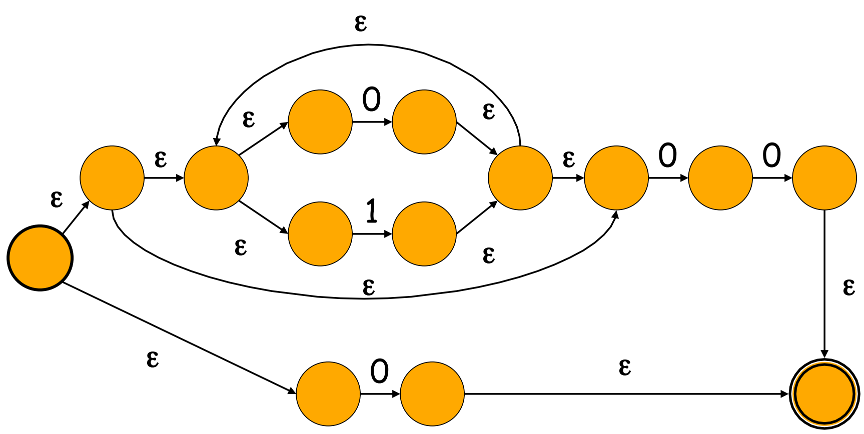
0|1



(0|1)*



(0|1)*00



((0|1)*00)|0

Simulating NFAs

- Similar to DFA simulation
- But have to deal with ε transitions and multiple transitions on the same input
- Instead of one state, we have to consider *sets* of states
- Simulating NFAs is a problem that is closely linked to converting a given NFA to a DFA

NFA to DFA Conversion

- Subset construction
- Idea: subsets of set of all NFA states are *equivalent* and become one DFA state
- Algorithm simulates movement through NFA
- Key problem: how to treat ε-transitions?

ε-Closure

- Start state: q_0
- ε-closure(S): S is a set of states

$$
\begin{aligned}
&\textbf{initialize: } S \leftarrow \{q_0\}\\
&T \leftarrow S\\
&\textbf{repeat } T' \leftarrow T\\
&\quad T \leftarrow T' \cup [\cup_{s \in T'} \textbf{move}(s, \epsilon)]\\
&\textbf{until } T = T'
\end{aligned}
$$

ε-Closure (T: set of states)

```
push all states in T onto stack
initialize ε-closure(T) to T
while stack is not empty do begin
     pop t off stack
     for each state u with u ∈ move(t, ε) do
        if u ∉ ε-closure(T) do begin
           add u to ε-closure(T)
           push u onto stack
        end
end
```


NFA Simulation

- After computing the ε-*closure* move, we get a set of states
- On some input extend all these states to get a new set of states

$$\mathbf{DFAedge}(T, c) = \epsilon\textbf{-closure}\left(\cup_{q \in T} \mathbf{move}(q, c)\right)$$

NFA Simulation

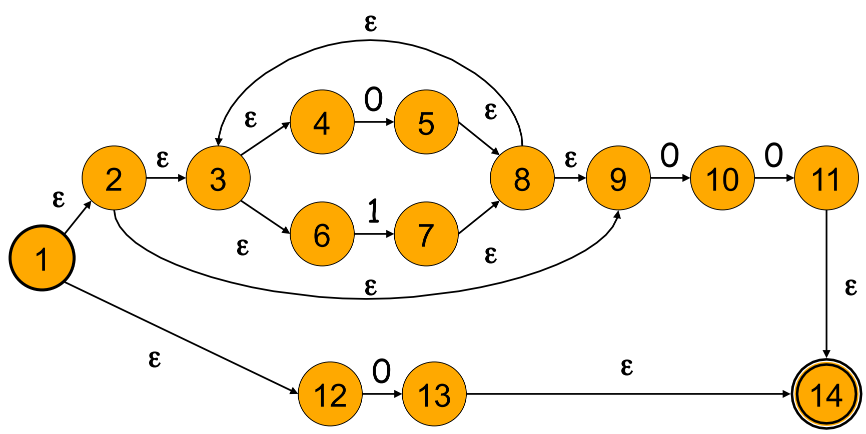
- Start state: q_0
- Input: $c_1, \ldots, c_k$

$$T \leftarrow \epsilon\textbf{-closure}(\{q_0\})$$
$$\textbf{for } i \leftarrow 1 \textbf{ to } k$$
$$\quad T \leftarrow \textbf{DFAedge}(T, c_i)$$

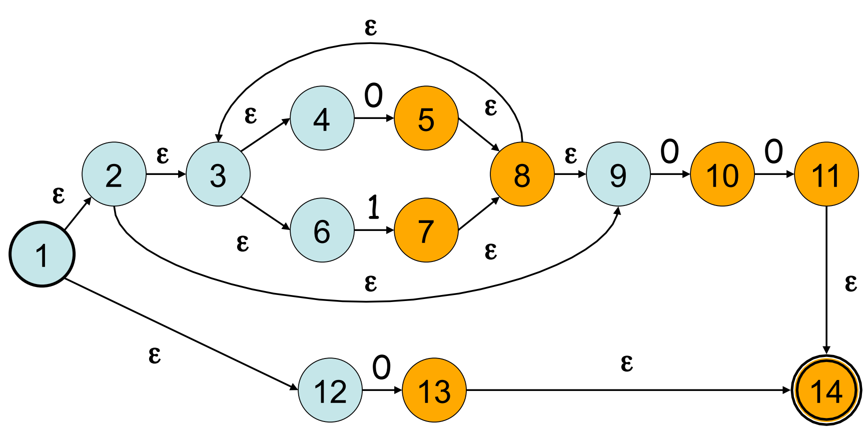
Conversion from NFA to DFA

- Conversion method closely follows the NFA simulation algorithm
- Instead of simulating, we can collect those NFA states that behave identically on the same input
- Group this set of states to form one state in the DFA

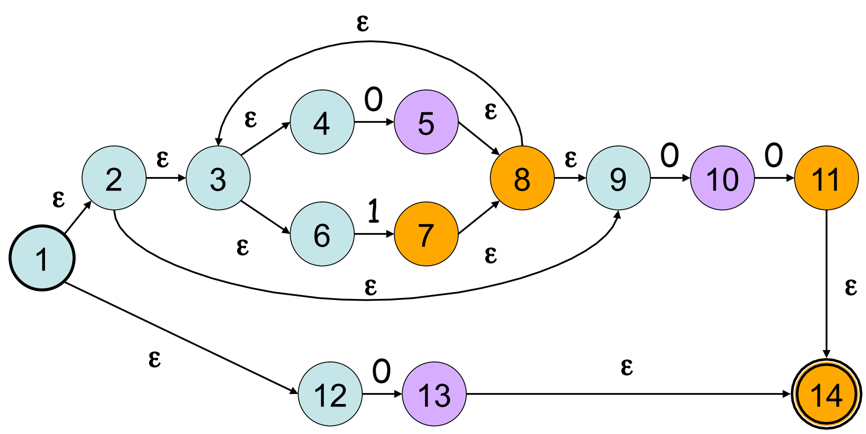
Example: subset construction



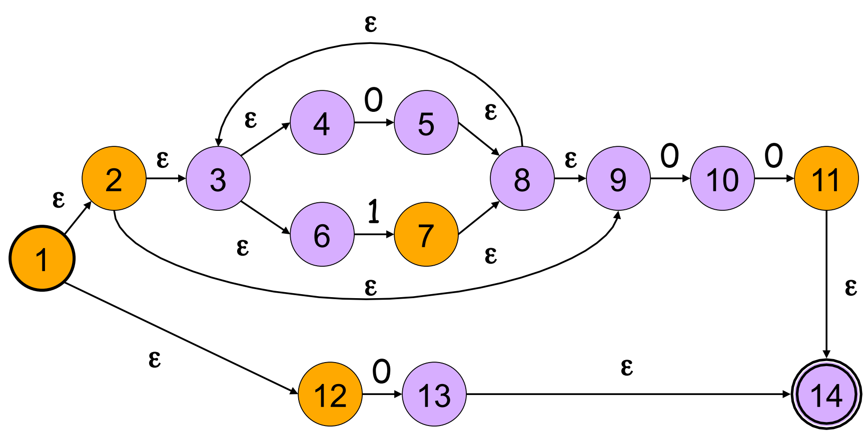
ε-*closure*(q_o)



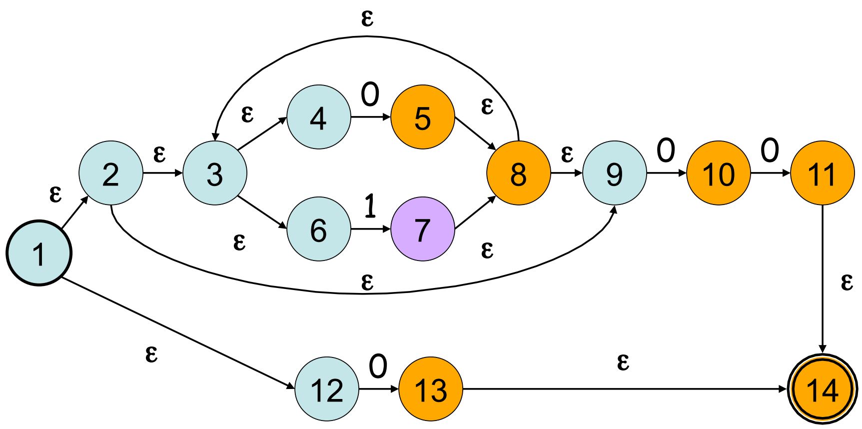
move(ε-*closure*(q_0), 0)



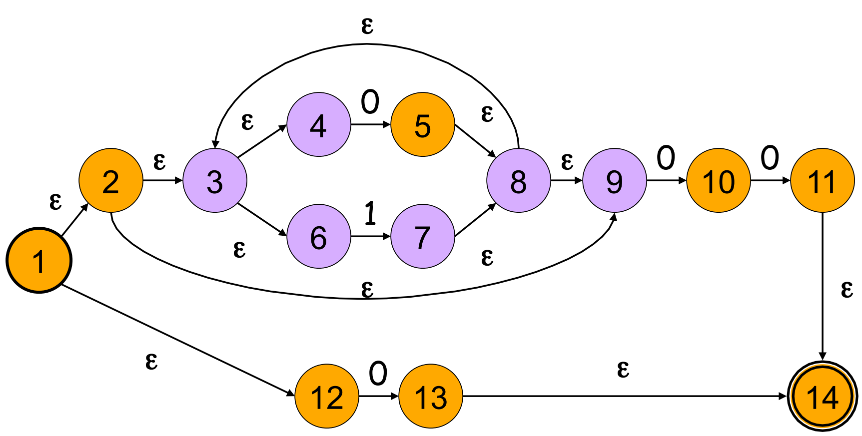
ε-*closure*(move(ε-*closure*(q_0), 0))



move(ε-*closure*(q_0), 1)



ε-*closure*(move(ε-*closure*(q_0), 1))



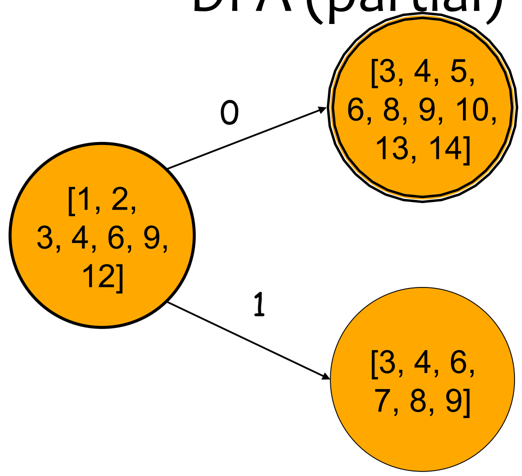
Subset Construction

```
add ε-closure(q0) to Dstates unmarked
while ∃ unmarked T ∈ Dstates do begin
    mark T;
    for each symbol c do begin
        U := ε-closure(move(T, c));
        if U ∉ Dstates then
            add U to Dstates unmarked
        Dtrans[d, c] := U;
    end
end
```

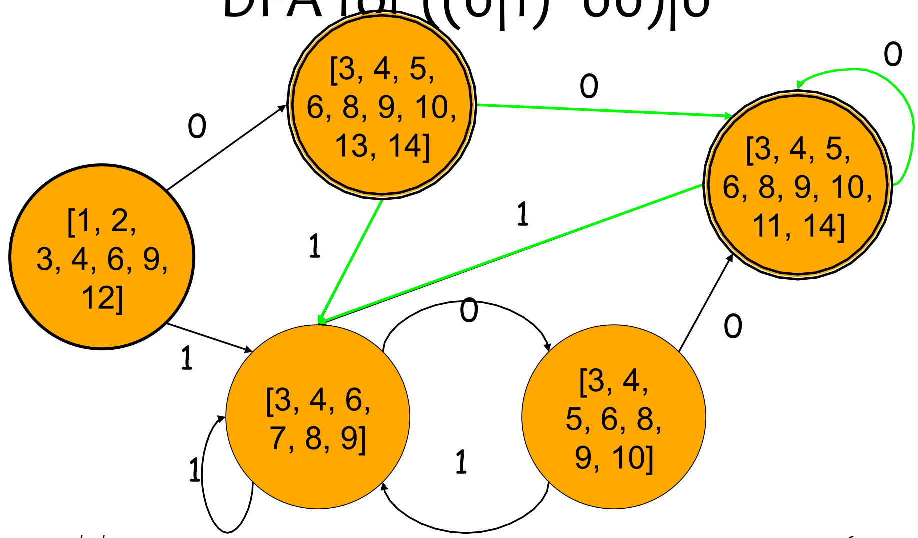

Subset Construction

```
states[0] = ε-closure({q0})
p = j = 0
while j ≤ p do begin
        for each symbol c do begin
                e = DFAedge(states[j], c)
                if e = states[i] for some i ≤ p
                then    Dtrans[j, c] = i
                else    p = p+1
                        states[p] = e
                        Dtrans[j, c] = p
        j = j + 1
        end
end
```

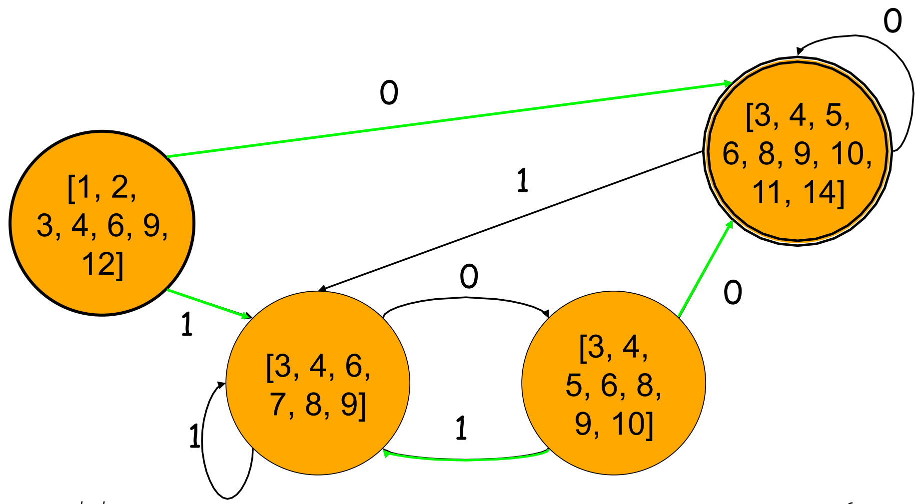
DFA (partial)



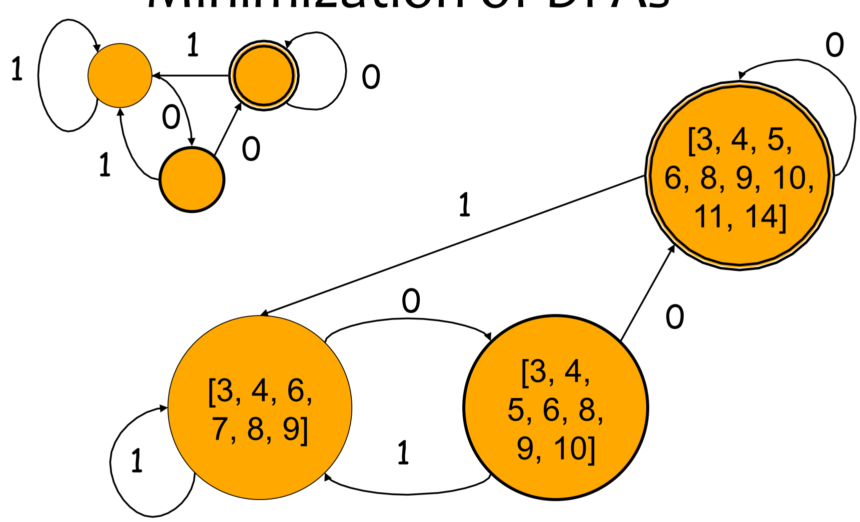
DFA for ((0|1)*00)|0



Minimization of DFAs

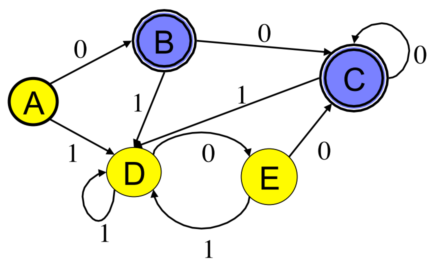


Minimization of DFAs



Minimization of DFAs

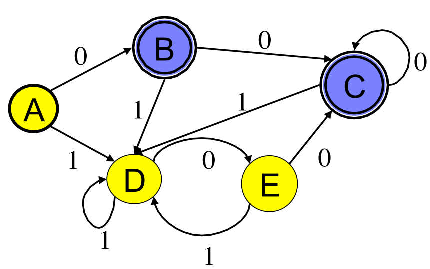
- Algorithm for minimizing the number of states in a DFA
- Step 1: partition states into 2 groups: accepting and non-accepting



Minimization of DFAs

- Step 2: in each group, find a sub-group of states having property P
- P: The states have transitions on each symbol (in the alphabet) to the *same* group

A, 0: blue
A, 1: yellow
E, 0: blue
E, 1: yellow
D, 0: yellow
D, 1: yellow

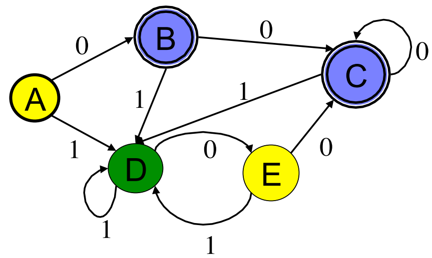


B, 0: blue
B, 1: yellow
C, 0: blue
C, 1: yellow

Minimization of DFAs

- Step 3: if a sub-group does not obey P split up the group into a separate group
- Go back to step 2. If no further sub-groups emerge then continue to step 4

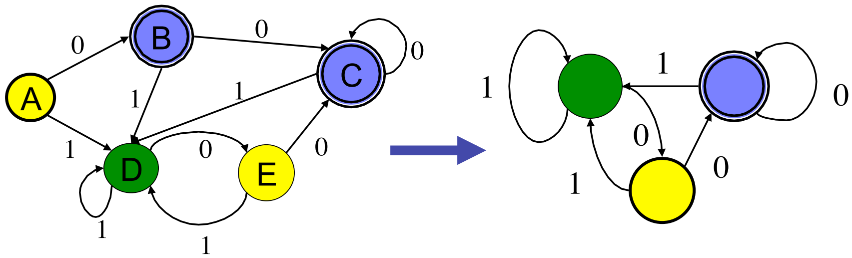
A, 0: blue
A, 1: green
E, 0: blue
E, 1: green
D, 0: yellow
D, 1: green



B, 0: blue
B, 1: green
C, 0: blue
C, 1: green

Minimization of DFAs

- Step 4: each group becomes a state in the minimized DFA
- Transitions to individual states are mapped to a single state representing the group of states



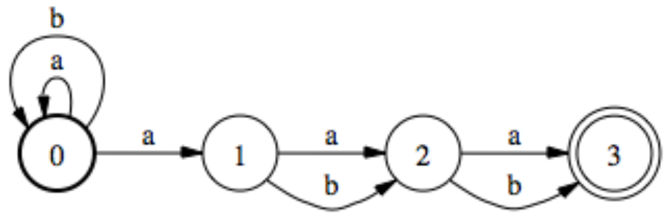
NFA to DFA

- Subset construction converts NFA to DFA
- Complexity:
  - For FSAs, we measure complexity in terms of initial cost (creating the automaton) and per string cost
  - Let r be the length of the regexp and n be the length of the input string
  - NFA, Initial cost: $O(r)$; Per string: $O(rn)$
  - DFA, Initial cost: $O(r^2s)$; Per string: $O(n)$
  - DFA, common case, $s = r$, but worst case $s = 2^r$

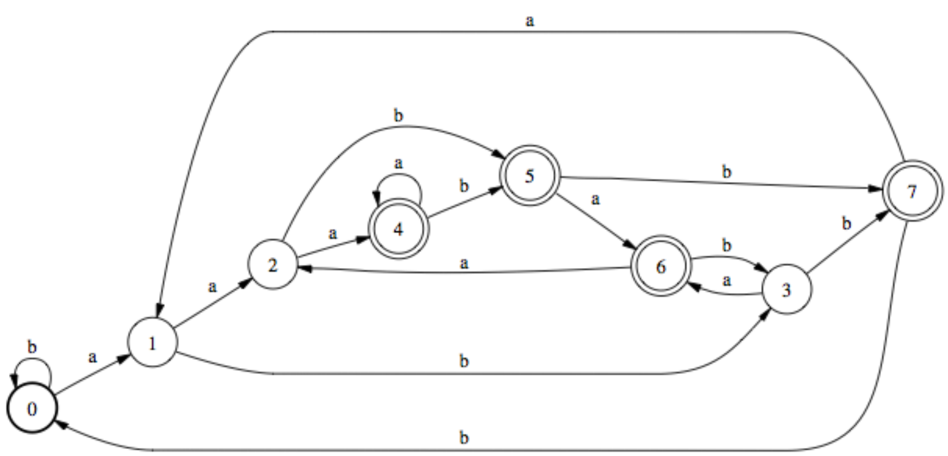
NFA to DFA

- A regexp of size r can become a 2^r state DFA, an exponential increase in complexity
  - Try the subset construction on NFA built for the regexp $\mathbf{A^*aA^{n-1}}$ where **A** is the regexp **(a|b)**
- Note that the NFA for regexp of size r will have r states
- Minimization can reduce the number of states
- But minimization requires determinization

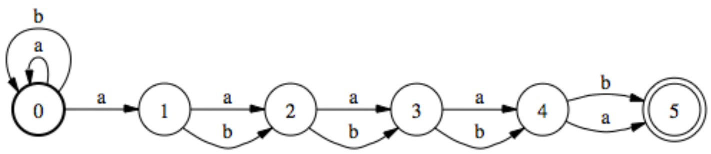
NFA to DFA

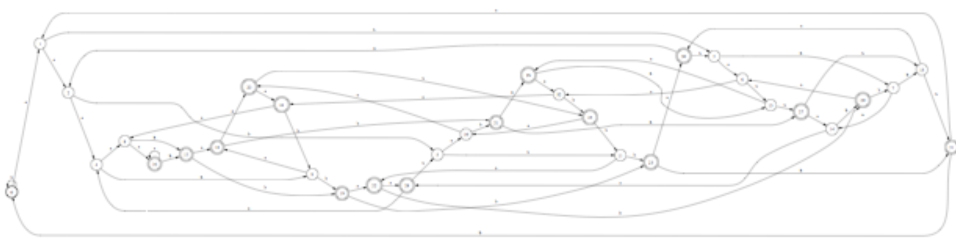


NFA to DFA



NFA to DFA





$2^5 = 32$ states

NFA vs. DFA in the wild

Engine Type	**Programs**
DFA	*awk* (most versions), *egrep* (most versions), *flex*, *lex*, MySQL, Procmail
Traditional NFA	GNU *Emacs*, Java, *grep* (most versions), *less*, *more*, .NET languages, PCRE library, Perl, PHP (pcre routines), Python, Ruby, *sed* (most versions), vi
POSIX NFA	*mawk*, MKS utilities, GNU *Emacs* (when requested)
Hybrid NFA/DFA	GNU *awk*, GNU *grep*/*egrep*, Tcl

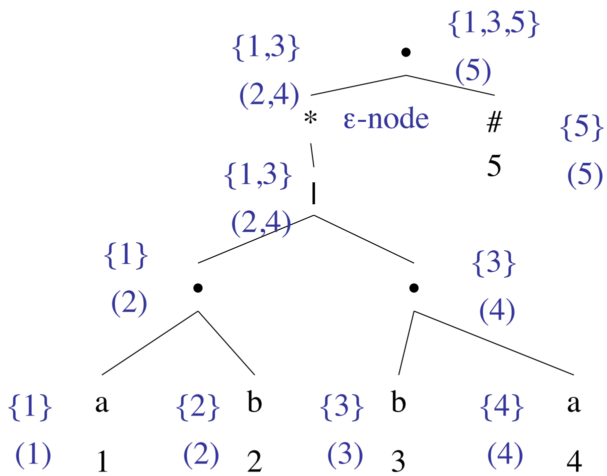
Extensions to Regular Expressions

- Most modern regexp implementations provide extensions:
  - matching groups; \1 refers to the string matched by the first grouping (), \2 to the second match, etc.,
    - e.g. ([a-z]+)\1 which matches abab where \1=ab
  - match and replace operations,
    - e.g. s/([a-z]+)/\1\1/g which changes ab into abab where \1=ab
- These extensions are no longer "regular". In fact, extended regexp matching is NP-hard
  - Extended regular expressions (including POSIX and Perl) are called REGEX to distinguish from regexp (which *are* regular)
- In order to capture these difficult cases, the algorithms used even for simple regexp matching run in time exponential in the length of the input

Converting Regular Expressions directly into DFAs

This algorithm was first used by Al Aho in `egrep`, and used in `awk`, `lex`, `flex`

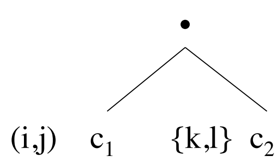
Regexp to DFA: `((ab)|(ba))*#`



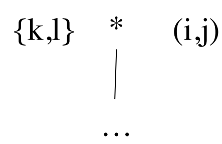
firstpos = {}
lastpos = ()

Regexp to DFA: *followpos*

- *followpos(p)* tells us which positions can follow a position *p*
- There are two rules that use the *firstpos* {} and *lastpos* () information

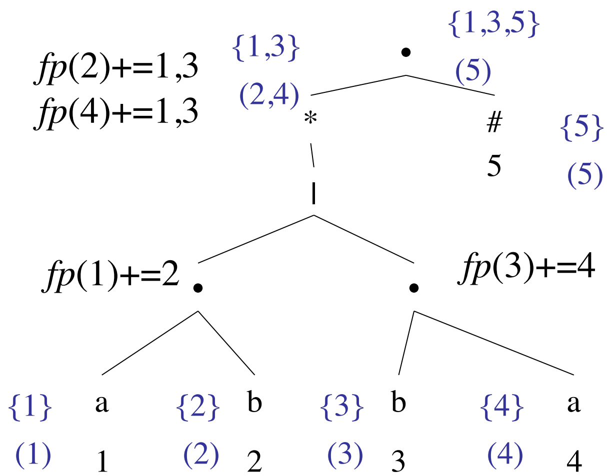


followpos(i)+=k,l
followpos(j)+=k,l



followpos(i)+=k,l
followpos(j)+=k,l

Regexp to DFA: `((ab)|(ba))*#`



root={1,3,5}
fp(1)=2
fp(3)=4
fp(2)=1,3,5
fp(4)=1,3,5

Regexp to DFA: `((ab)|(ba))*#`

root={1,3,5}
fp(1)=2
fp(3)=4
fp(2)=1,3,5
fp(4)=1,3,5

{1,3,5} A

A: fp(1),a {2},a B,a

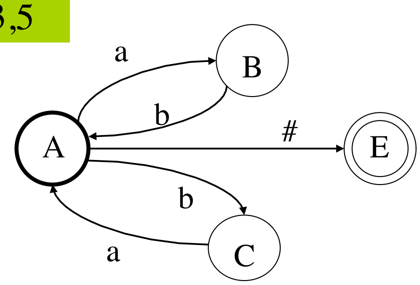
A: fp(3),b {4},b C,b

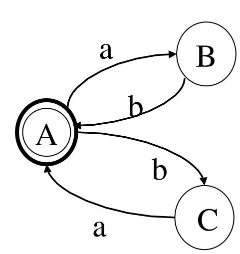
A: fp(5),# {},# E,#

B: fp(2),b {1,3,5},b A,b

C: fp(4),a {1,3,5},a A,a

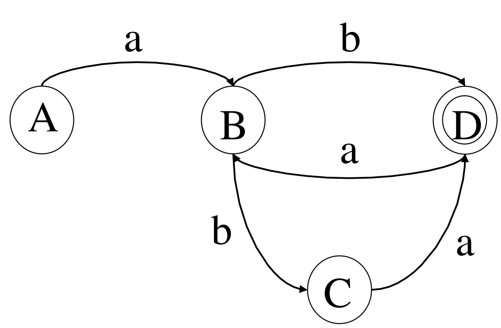
1:a
2:b
3:b
4:a
5:#





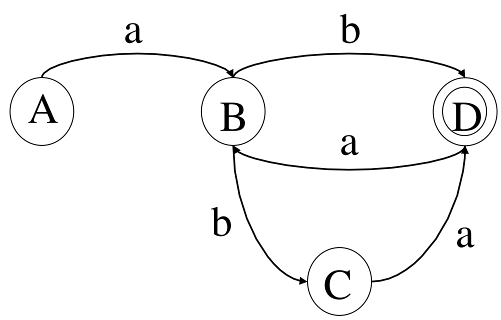
Converting an NFA into a Regular Expression

NFA to RegExp



What is the regular expression for this NFA?

NFA to RegExp



- A = a B
- B = b D | b C
- D = a B | ε
- C = a D

NFA to RegExp

- Three steps in the algorithm (apply in any order):

1. Substitution: for B = X pick every A = B | T and replace to get A = X | T
2. Factoring: (R S) | (R T) = R (S | T) and (R T) | (S T) = (R | S) T
3. Arden's Rule: For any set of strings S and T, the equation X = (S X) | T has X = (S*) T as a solution.

NFA to RegExp

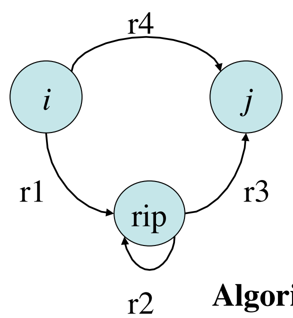
- A = a B
  B = b D | b C
  D = a B | ε
  C = a D
- Substitute:
  A = a B
  B = b D | b a D
  D = a B | ε
- Factor:
  A = a B
  B = (b | b a) D
  D = a B | ε
- Substitute:
  A = a (b | b a) D
  D = a (b | b a) D | ε

NFA to RegExp

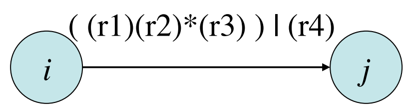
A = a (b | b a) D
D = a (b | b a) D | ε

- Factor:
  A = (a b | a b a) D
  D = (a b | a b a) D | ε
- Arden:
  A = (a b | a b a) D
  D = (a b | a b a)* ε
- Remove epsilon:
  A = (a b | a b a) D
  D = (a b | a b a)*
- Substitute:
  A = (a b | a b a)
  (a b | a b a)*
- Simplify:
  A = (a b | a b a)+

NFA to Regexp using GNFAs

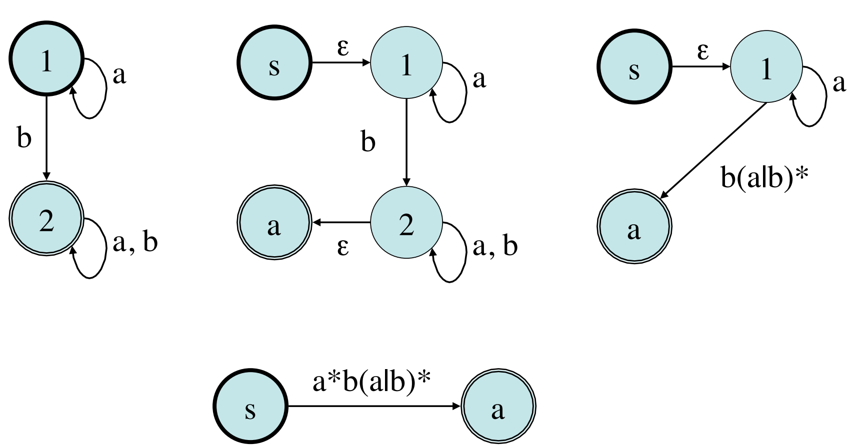


Generalized NFA: transition function takes state and regexp and returns a set of states

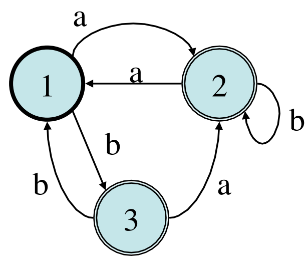


Algorithm:
1. Add new start & accept state
2. For each state *s*: rip state *s* creating GNFA, consider each state *i* and *j* adjacent to *s*
3. Return regexp from start to accept state

NFA to Regexp using GNFAs



NFA to Regexp using GNFAs



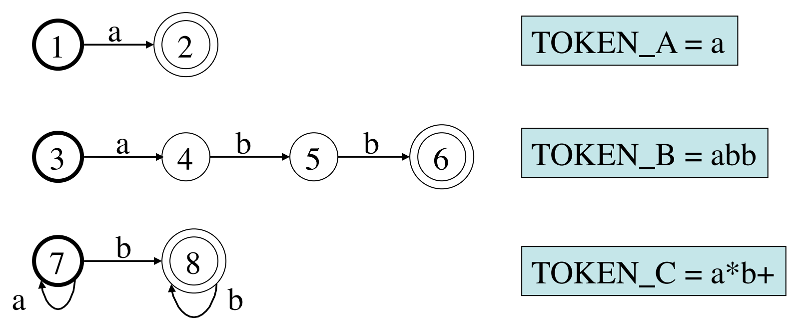
Rip states 1, 2, 3 in that order, and we get: (a(aa|b)*ab|b)((ba|a)(aa|b)*ab|bb)*((ba|a)(aa|b)*|ε)|a(aa|b)*

Implementing a Lexical Analyzer

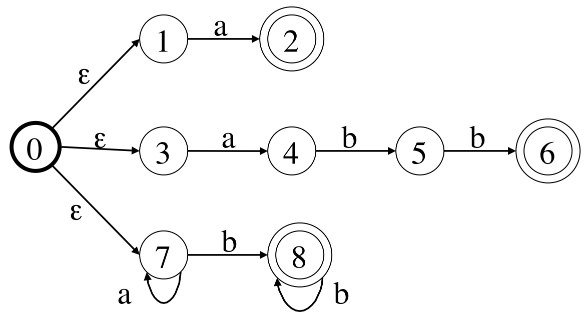
Lexical Analyzer using NFAs

- For each token convert its regexp into a DFA or NFA
- Create a new start state and create a transition on ε to the start state of the automaton for each token
- For input $i_1, i_2, \ldots, i_n$ run NFA simulation which returns some final states (each final state indicates a token)
- If no final state is reached then raise an error
- Pick the final state (token) that has the longest match in the input,
  - e.g. prefer DFA #8 over all others because it read the input until i_{30} and none of the other DFAs reached i_{30}
  - If two DFAs reach the same input character then pick the one that is listed first in the ordered list

Lexical Analysis using NFAs



Lexical Analysis using NFAs



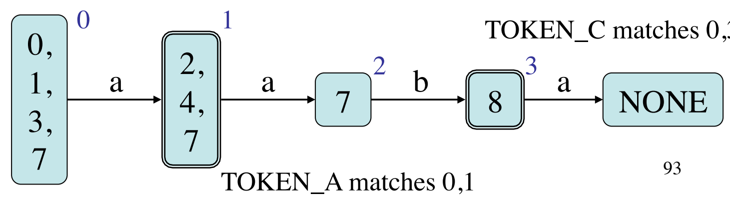
TOKEN_A = a

TOKEN_B = abb

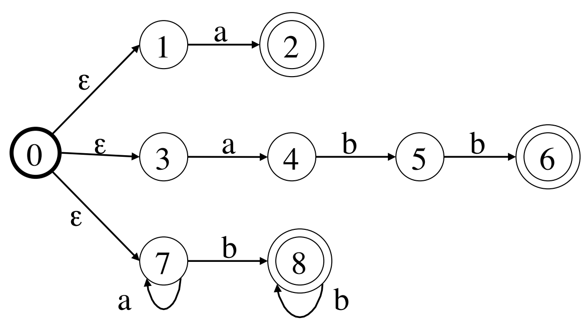
TOKEN_C = a*b+

Input: aaba

${}_0a_1a_2b_3a_4$



Lexical Analysis using NFAs



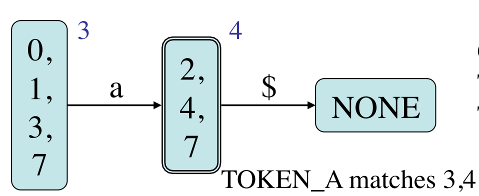
TOKEN_A = a

TOKEN_B = abb

TOKEN_C = a*b+

Input: aaba

${}_0a_1a_2b_3a_4$



Output:
TOKEN_C aab [0,3]
TOKEN_A a [3,4]

Lexical Analyzer using DFAs

- Each token is defined using a regexp r_i
- Merge all regexps into one big regexp
  - $R = (r_1 \mid r_2 \mid \ldots \mid r_n)$
- Convert R to an NFA, then DFA, then minimize
  - remember orig NFA final states with each DFA state

Lexical Analyzer using DFAs

- The DFA recognizer has to find the *longest leftmost match* for a token
  - continue matching and report the last final state reached once DFA simulation cannot continue
  - e.g. longest match: *<print>* and not *<pr>, <int>*
  - e.g. leftmost match: for input string *aabaaaaab* the regexp a^+b will match *aab* and not *aaaaab*
- If two patterns match the same token, pick the one that was listed earlier in R
  - e.g. prefer final state (in the original NFA) of r_2 over r_3

Lookahead operator

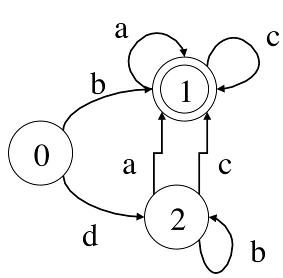
- Implementing r_1/r_2 : match r_1 when followed by r_2
- e.g. *a*b+/a*c* accepts a string *bac* but not *abd*
- The lexical analyzer matches $r_1 \varepsilon r_2$ up to position *q* in the input
- But remembers the position *p* in the input where r_1 matched but not r_2
- Reset to start state and start from position *p*

Efficient data-structures for DFAs

Implementing DFAs

- 2D array storing the transition table
- Adjacency list, more space efficient but slower
- Merge two ideas: array structures used for sparse tables like DFA transition tables
  - base & next arrays: Tarjan and Yao, 1979
  - Dragon book (default+base & next+check)

Implementing DFAs



	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

Implementing DFAs

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

			-	1	-	2		
					1	-	1	-
	1	2	1	-				
next	1	2	1	1	1	2	1	-
	0	1	2	3	4	5	6	7
check	2	2	2	0	1	0	1	-

base

0	2
1	4
2	0

nextstate(s, x) :
L := base[s] + x
return next[L] **if** check[L] **eq** s

Implementing DFAs

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

		-	1	-	2		
				1	-	1	-
	-	2	-	-			
next	-	2	1	1	2	1	-
	0	1	2	3	4	5	6
check	-	2	0	1	0	1	-

base / default

	base	default
0	1	-
1	3	-
2	0	1

nextstate(s, x) :
L := base[s] + x
return next[L] **if** check[L] **eq** s
else return *nextstate*(default[s], x)

Summary

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
  - Thompson's Rules
- NFA $\Rightarrow$ DFA
  - Subset construction
- DFA $\Rightarrow$ minimal DFA
  - Minimization

$\Rightarrow$ Lexical Analyzer (multiple patterns)