Chapter 2 Scanning

- Study Goals:
  - Master
    Write regular expression, the transition from regular expression to DFA, the construction of scanner
  - Understand
    Concept of regular expression,NFA,DFA
  - Know

2.1 The Scanning Process

Review

- The task of scanner
  Reading the source program as a file of characters and diving it up into tokens
- Token
  - Token is a sequence of characters that represents a unit of information.
  - Token represents a certain pattern of characters, such as keywords, identifiers, special symbols.

1 The Categories of Tokens

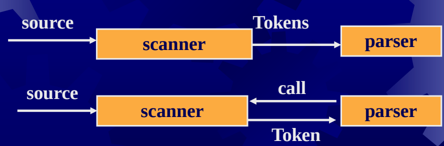
- Categories of Tokens
  - Keywords
    Fixed strings of characters that have special meaning in the language,such as “if” and “then”
  - Special symbols
    Include arithmetic operations,assignment, equality and so on,such as +, -, :=, =

  - Identifies:sequences of letters and digits beginning with a letter
  - Literals:include numeric constants and string literals ,such as 42, 3.14, “hello”, “a”

- Token and lexeme
  - Token is presented as (Kind,Value)
  - Kinds are logical entities,represented as IF,THEN,PLUS,MINUS,NUM,ID and so on
  - The string value represented by a token is called lexeme.
    - Reserved words and special symbol have only one lexeme
    - While number and identifier have infinitely many lexemes
  - Example
    (IF, “if”) (PLUS, “+”) (ID, “x”)

2 Interface of Scanner

- Scanning is a single pass
  Convert the entire source program into a sequence of tokens
- Scanning is a sub function of the parser
  When called by the parser it returns the single next token from the input



- Main Content of Scanning Study
  - Specification of lexical structure : Regular Expression
  - Recognition system:Finite Automata represents algorithms for recognizing strings given by regular expression
  - Practical Methods for writing programs that implement the recognition processes represented by finite automata

2.2 Regular Expressions

- Function
  Represent patterns of strings of characters
- The meaning of regular expression
  - A regular expression r is completely defined by the set of strings that it matches
  - This set is called the language generated by the regular expression, written as L(r)
  - L(r) is defined on a set of symbol called alphabet ∑

2.2.1 String and Language

1. Alphabet
- Definition
  Any finite set of symbols
- Example
  ∑={0,1} A={a,b,c}

2 String

- Definition
  A string over some alphabet is a finite sequence of symbols drawn from that alphabet.
- Examples
  0,00,10 are strings of ∑={0,1}
  a, ab, aaca are strings of A={a,b,c}

- Length of string
  - The length of a string s,usually written as |s|,is the number of occurrences of symbols in s.
  - Example: |abc|=3
- The empty string
  - Denoted byε,is a special string of length zero
  - {ε}is not equal to Φ ({ })

3 Operations on String

- Concatenation
  - If x and y are strings,then the concatenation of x and y,written as xy,is the string formed by appending y to x
  - Example: x=ST，y=abu ,xy=STabu
    εx = xε=x
- Exponentiation
  - If a is a string then a^n = aa…aa
  - Example: a^1=a a^2=aa a^0=ε

4 Language

- Definition
  Any set of strings over some fixed alphabet
- Example
  - { ε }is a language
  - Φ, the empty set is also a language

5 Operations on language

- Concatenation
  - Concatenation of L and M is written as LM LM ={st|s∈L,t∈M}
  - Example:
    L={ab,cde} M = {0,1}
    LM ={ab0,ab1,cde0,cde1}
  - {ε}A=A{ε}=A

- Exponentiation
  The exponentiation of L is defined as:
  - L^0 ={ε }
  - L^1 = L , L^2 = LL
  - L^K = LL……L
  (L^K is L concatenated with itself k-1 times)

- Closure
  - Closure of L (written as L*) denotes zero or more concatenations of L
  - $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$
  - Example：L={0，1}
    L* $=L^0 \cup L^1 \cup L^2 \cup \ldots$
    ={ε,0,1,00,01,10,11,000,…}
  - Positive closure of L(written as L^+) denotes one or more concatenation of L
  - $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$
  - $L^* = L^0 \cup L^+$
    $L^+ = LL^* = L^*L$

2.2.2 Definition of Regular Expressions

- The set of basic regular expression
- Essential set of operations that generate new regular expression from existing ones

A regular expression is one of the following:

1) ε and φ are regular expressions, L(ε)={ε}, L(Φ)=Φ
2) Any a ∈∑ is a regular expression of ∑, L(a)={ a }

3) if e_1 and e_2 are regular expressions of ∑, then the following are all regular expressions of ∑:

regular expression	language generated by the regular expression
(e_1)	$L(e_1)$
$e_1 \mid e_2$	$L(e_1) \cup L(e_2)$
$e_1 e_2$	$L(e_1) L(e_2)$
$e_1{}^*$	$(L(e_1))^*$

- Operations that generate new regular expression from existing ones are:
  - Choice among alternatives (|)
  - Concatenation(.)
  - Repetition or closure (*)
- The precedence of the operations is:

  '*' > '.' > '|'
- Example

  L(a|bc*)={a} ∪({b}{ε ,c,cc,…})

  ={a} ∪{b,bc,bcc…}={a,b,bc,bcc…}

- Example:

  Σ={a，b}, the following are regular expressions and the language they generated

regular expression r	L(r)
a	{a}
a│b	{a,b}
ab	{ab}
(a│b)(a│b)	
L(r)={a,b}{a,b}	={aa,ab,ba,bb}
a *	{ε ,a,aa,…}
(a│b)*	{ε ,a,b,aa,ab ……}

- Names for Regular Expressions
  - To give a name to a long regular expression for convenience
  - Example:

    a sequence of one or more numeric digits

    (0|1|…|9)(0|1|…|9)*

    can be written in

    *digit digit**

    where

    digit = 0|1|…|9

    is a regular definition of the name *digit*

- Example

  Given the description of the strings to be matched and translate the description into a regular expression
- ∑={a,b,c}, regular expression of strings that contain exactly one b is (a|c)*b(a|c)*
- Regular expression of strings that contain at most one b is: (a|c)*|(a|c)*b(a|c)* or (a|c)*(b| ε)(a|c)*

Explanation

- The same language may be generated by many different regular expressions
- Not all sets of strings that we can describe in simple terms can be generated by regular expressions

  Example:

  The set of strings S={b,aba,aabaa,…}={a^nba^n|n≥0} can not be generated by regular expressions

2.2.3 Regular Expression for Programming Language Tokens

1 Typical regular expression for tokens

let l=a|b|…|z d=0|1|…|9

- Identifier: l (l | d)*
- Unsigned integer: dd*
- Real number: dd*(.dd*| ε)
- Reserved word: if|while|do|…

2 Issues related to the recognition of tokens

- Ambiguity
  - Some strings can be matched by several different regular expressions
  - Language definition must give disambiguating rules
    - When a string can be either an identifier or a keyword, keyword interpretation is preferred
    - When a string can be a single token or a sequence of several tokens, the single-token interpretation is preferred(principle of longest substring)

- Token delimiters
  - Characters that are unambiguously part of other tokens are delimiters

    Example: in string “xtemp=ytemp”, ‘=’ is a delimiter
  - Blanks, newlines, tab characters, comment are all token delimiters

    Example:in string “while x…”, two tokens “while” and “x” are separated by a blank

    Scanner discards them after checking for any token delimiting effects

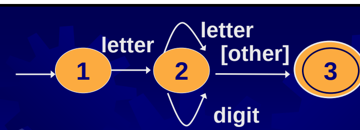
- Lookahead
  - Scanner must deal with the problem of lookahead one or more characters
  - For example: recognizing the special symbol ‘:=‘, when encounter ‘:’, scanner must lookahead to determine whether the token is ‘:’ or ‘:=’
  - Lookahead tokens should not be consumed from the input string

2.3 Finite Automata

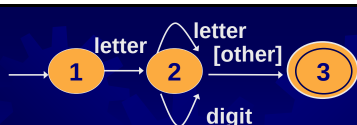
- Function :
  - Finite automata are mathematical ways of describing particular kinds of algorithms.
  - Here they are used to describe the process of recognizing patterns written in regular expressions and so can be used to construct scanners
- Category:
  - Deterministic Finite Automata(DFA)
  - Nondeterministic Finite Automata(NFA)

- Relationship between finite automata and regular expressions

  Example
  - Regular expression for identifier
    let letter=a|b|…|z   digit=0|1|…|9
    identifier=letter(letter|digit)*
  - The process of recognizing such an identifier can be described as finite automata



- States: are locations in the process of recognition recording how much of the pattern has already been seen
- Transitions: record a change from one state to another
- Start state: at which the recognition process begins
- Accepting states: represent the end of the recognition process



- The process of recognizing an actual string can be indicated by listing the sequence of states and transitions in the diagram used in the recognition process.

Example: recognizing process of “xtemp=...”:

$$\rightarrow 1 \xrightarrow{x} 2 \xrightarrow{t} 2 \xrightarrow{e} 2 \xrightarrow{m} 2 \xrightarrow{p} 2 \xrightarrow{=} 3$$

2.3.1 Definition of DFA
2.3.2 Definition of NFA
2.3.3 Implementation of Finite Automata in Code

2.3.1 Definition of DFA

- Definition of DFA

  A DFA M= （S，Σ，T，S_0，A）
  1. S is a set of states
  2. Σis an alphabet
  3. T is a transition function T: S X ∑->S, T(S_i,a)=S_j represents when the current state is S_i and the current input character is a，DFA will transit to state S_j
  4. S_0∈S is a start state
  5. A⊂S is a set of accepting states

- Example

  DFA M=({S，U，V，Q},{a，b},f,S,{Q})

  f is defined as following：

  f（S，a）=U   f（S，b）=V
  f（V，a）=U   f（V，b）=Q
  f（U，a）=Q   f（U，b）=V
  f（Q，a）=Q   f（Q，b）=Q

- The meaning of deterministic

  The next state is uniquely given by the current state and the current input character

Transition Diagram of DFA

- Each state is a node of the diagram
- The start state is indicated by drawing an unlabeled arrowed line to it
- Accepting states are indicated by drawing a double-line border around the state
- If $T(S_i,a)=S_j$, then drawing an arrowed line from the node S_i to node S_j labeled by a

- Example:

  DFA M= ({S, U, V, Q},{a, b},f,S,{Q})

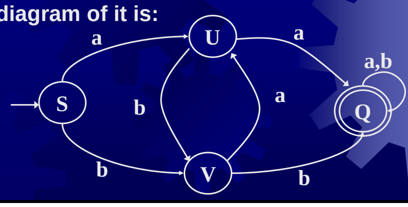
  f (S,a) =U　　f (S,b) =V

  f (V,a) =U　　f (V,b) =Q

  f (U,a) =Q　　f (U,b) =V

  f (Q,a) =Q　　f (Q,b) =Q

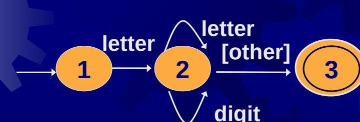
  The diagram of it is:



Notes about the Diagram

- Extension to the definition:

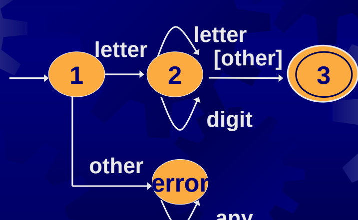
  The transitions can also be labeled with names representing a set of characters



- Convention

  Error transitions are not drawn in the diagram

Diagram for an identifier with error transition



Transition table of DFA

- Transition table is indexed by states and input characters
- It's values express the values of the transition function T
- The first state listed is the start state
- Using a separate column to indicate accepting state

Character / State	C	…	Accepting
S	T(S,C)		yes/no
…			

- Example:

DFA M= ({S, U, V, Q},{a, b},f,S,{Q})

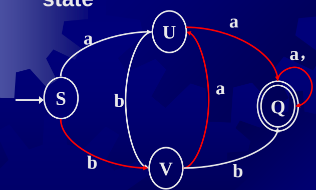
f (S,a) =U	f (S,b) =V
f (V,a) =U	f (V,b) =Q
f (U,a) =Q	f (U,b) =V
f (Q,a) =Q	f (Q,b) =Q

Transition table of the DFA :

State \ Char	a	b	Accepting
S	U	V	no
U	Q	V	no
V	U	Q	no
Q	Q	Q	yes

- L(M): the language accepted by DFA M

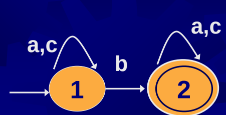
L(M) is the set of strings of characters $c_1c_2\ldots c_n$ with each $c_i \in \Sigma$ such that there exist states $s_1=T(s_0,c_1), s_2=T(s_1,c_2), \ldots S_n=T(s_{n-1},c_n)$ with s_0 is the start state and s_n is an accepting state



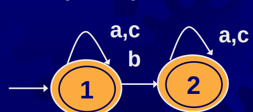
String "baab" is accepted by DFA M

Example of DFA

1 ∑={a,b,c} ,The set of strings that contain exactly one b is accepted by the following DFA:

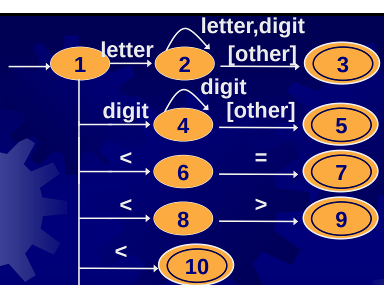


2 The set of strings that contain at most one b is accepted by the following DFA:



2.3.2 Definition of NFA

- The need of NFA
  - Problem of DFA when recognizing all tokens of a program
  - In a programming language there are many tokens, each token will be recognized by its own DFA
  - We should combine all the tokens into one giant DFA.



- This is not a DFA. If there is not a systematic way it will be complex to arrange the diagram to be a DFA

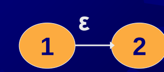
- Solution to this problem
  - Expand DFA to NFA (which includes the case where more than one transition from a state may exist for a particular character)
  - Developing an algorithm for systematically turning NFA to DFA

- Nondeterministic Finite Automaton

An NFA M=(S,Σ,T,S_0,A), where

1. S is a set of states
2. ∑ is an alphabet
3. T is a transition function T:
   S X (∑∪{ε})->subset of S
4. S_0∈S is a start state
5. A⊂S is a set of accepting states

- NFA is similar to DFA except that
  - Expand ∑ to include ε
    NFA may have ε-transition--a transition that may occur without consulting the input string



  - Expand the definition of T
    More than one transition from a state may exist for a particular character. So the value of T is a set of states rather than a single state

- Example

NFA M=({S，P，Z},{0，1},f,S,{Z})

f (S,0) = {P}　f (S,1) = {S，Z}

f (Z,0) = {P}　f (Z,1) = {P}

f (P,1) = {Z}

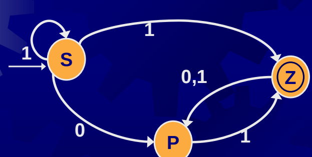
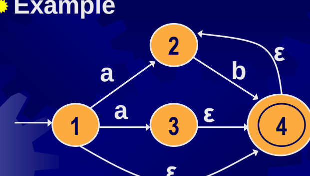


Diagram of NFA

- L(M):the language accepted by M
  L(M) is the set of strings of character c1c2…cn with each ci from ∑∪{ε} such that there exist states s1 in T(S0,c1),s2 in T(S1,c2),…,Sn in T(S_{n-1},c_n) with s_0 is the start state and Sn an element of A
  - Any of the ci in c1c2…cn may be ε
  - The string that is actually accepted is the string c1c2…cn with the ε's removed.

- The meaning of nondeterministic
  The sequence of transition that accepts a particular string is not determined at each step by the state and the next input character

- Example



The string "abb" can be accept by either of the following sequence of transitions:

$$\to 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

$$\to 1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

2.3.3 Implementation of Finite Automata in Code

The process of constructing a scanner :

regular expression → DFA → program for scanner

- Regular expressions represent a pattern, that are used as token descriptions
- DFAs represent algorithms that accept strings according to a pattern described in regular expression

---

- Turn a regular expression to DFA(study in 2.4)
- Translate a DFA into the code for a scanner

---

1 Translate a Diagram of DFA into code

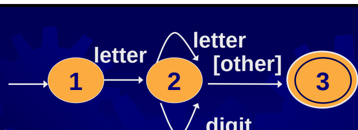
General algorithm that will translate DFA to code

- Use a variable to maintain the current state
- Write the transition as a doubly nested case statement inside a loop
- The first case statement tests the current state
- The nested second level tests the input character, given the state

---

Example:DFA that accepts identifiers


```
state:=1;{start}
while state=1 or 2 do
  case state of
  1:case input character of
      letter:advance the input;
            state:=2;
      else state:=...{error or other};
     end case
```




```
  2:case input character of
      letter,digit:advance the input;
                  state:=2;
      else state:=3;
     end case;
  end case;
end while;
if state=3 then accept else error;
```

---

2 Translate a Transition Table of DFA into code

- Using transition table, we can write code in a form that will implement any DFA

- Variables used in code scheme
  - Transitions are kept in a transition array "T" indexed by states and input characters;
  - Transitions that advance the input are given by the Boolean array "Advance", indexed also by states and input characters;
  - Accepting states are given by the Boolean array "Accept", indexed by states

- Code Scheme

```
state:=1;
ch:=next input character;
while not Accept[state] and not error(state) do
    newstate:=T[state,ch];
    if Advance[state,ch] then ch:=next input char;
    state:=newstate;
end while;
if Accept[state] then accept;
```

- The advantages of table-driven methods
  - The size of code is reduced
  - The same code will work for many different problems
  - The code is easier to change(maintain)
- The disadvantage

  The tables can become very large, causing a significant increase in the space

3 Action of the code

- A typical action when making a transition is to move the character from the input string to a string that accumulates the characters belonging to a single token
- A typical action when reaching an accepting state is to return the token just recognized
- A typical action when reaching an error state is to either back up in the input or generate an error token

2.4 From Regular Expressions to DFAs

- Regular expression is equivalent to DFA
- From regular expression to DFA
  - Translate a regular expression into an NFA(2.4.1)
  - Translate an NFA into a DFA(2.4.2)
  - Minimizing a DFA(2.4.3)

2.4.1 From a Regular Expression to an NFA

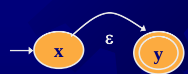
- "Inductive" method

  It follows the structure of the definition of a regular expression

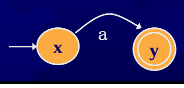
- Construct NFA for each basic regular expression

1 NFA that is equivalent to regular expression ∅

2 NFA that is equivalent to regular expression ε

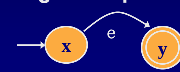


3 NFA that is equivalent to regular expression a, a ∈Σ

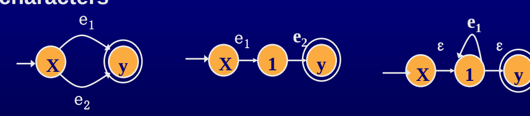


- Construct NFA for complex regular expressions

1 The NFA for regular expression “e” is

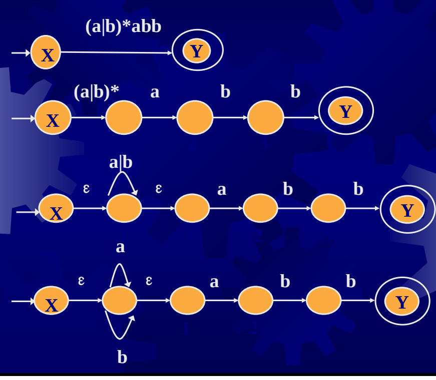


2 Break up the NFA basing on the following three operations until the arrowed line is labeled by only characters



$e=e_1|e_2$ $e=e_1e_2$ $e=e_1{}^*$

Example: translate (a|b)*abb into an NFA

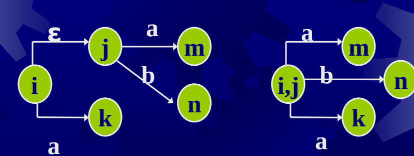


2.4.2 From an NFA to a DFA

1 Two problems need to be solved in translation

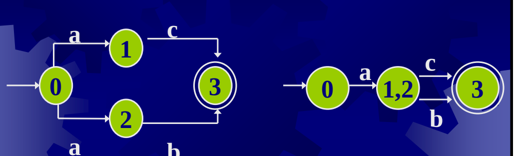
1) Eliminate ε-transition

If $S1 \xrightarrow{\varepsilon} S2$,then S2 is eliminated



(a) (b)

2) Eliminate multiple transitions from a state on a single character



(a) (b)

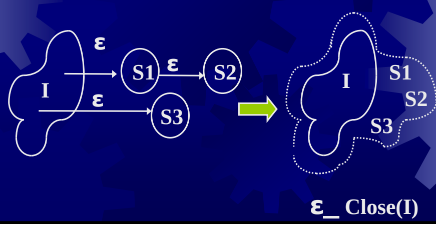
2 Transition Method--Subset Construction

- The states of DFA are the sets of states of the origin NFA
- That is, we use one state of DFA to substitute the set of states of NFA reachable by transition from a state on a single input character

3 Associated computation on the set of states

1) Theε-closure of a set of states

- The set of all states reachable by a series of zero or more ε-transitions from the set of states



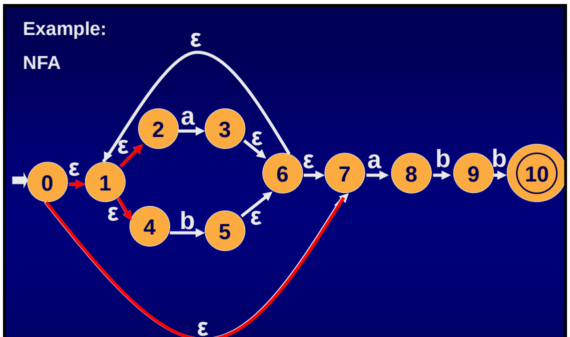
- Write ε_closure(I) as closure(I)

  Closure(I) =I ∪ { S_k | if $S_j \xrightarrow{\varepsilon} S_k$, $S_j \in$ Closure(I) , $S_k \notin$ Closure(I) }

- ε_closure of a set of states always contain the set itself

Example:

NFA



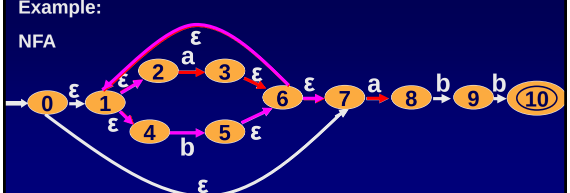
if I={0},the ε_closure(I)={ 0, 1, 2, 4, 7 }

2) I_a Subset

- I is a set of states, a is a character in the alphabet
- Move（I, a）={t|s∈I,and $s \xrightarrow{a} t$}
- I_a= ε_closure (Move(I , a))

Example:

NFA



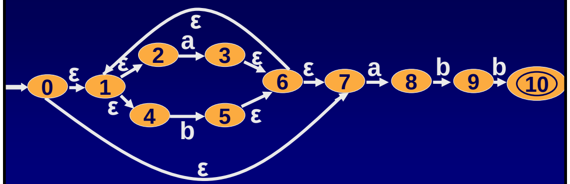
if I={0,1,2,4,7} then

I_a = ε_closure({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 }

I_b = ε_closure({ 5 }) = { 5, 6, 7, 1, 2, 4 }

4 Algorithm for constructing a DFA M' form a given NFA M

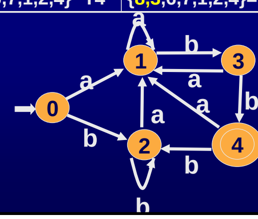
- Compute the ε_closure of the start state of M, this becomes the start state of M'
- For this set, and for each subsequent set S ,we compute transitions S_a on each character a∈Σ, this defines a new state together with a new transition $S \xrightarrow{a} Sa$
- Continue with this process until no new states or transitions are created.
- Mark as accepting those states that contain an accepting state of M



I		I_a	I_b	
{0,1, 7,2,4}	T0	{8,3,6,7,1,2,4}	{5,6,7,1,2,4}	0
{8,3,6,7,1,2,4}	T1	{8,3,6,7,1,2,4}=T1	{9,5,6,7,1,2,4}	0
{5,6,7,1,2,4}	T2	{8,3,6,7,1,2,4}=T1	{5,6,7, 1,2,4}=T2	0
{9,5,6,7,1,2,4}	T3	{8,3,6,7,1,2,4}=T1	{10,5,6,7,1,2,4}	0
{10,5,6,7,1,2,4}	T4	{8,3,6,7,1,2,4}=T1	{5,6,7,1,2,4}=T2	1

Rename the sets of states of DFA,we get

I		I_a	I_b	
{0,1, 7,2,4}	T0	{8,3,6,7,1,2,4}	{5,6,7,1,2,4}	0
{8,3,6,7,1,2,4}	T1	{8,3,6,7,1,2,4}=T1	{9,5,6,7,1,2,4}	0
{5,6,7,1,2,4}	T2	{8,3,6,7,1,2,4}=T1	{5,6,7, 1,2,4}=T2	0
{9,5,6,7,1,2,4}	T3	{8,3,6,7,1,2,4}=T1	{10,5,6,7,1,2,4}	0
{10,5,6,7,1,2,4}	T4	{8,3,6,7,1,2,4}=T1	{5,6,7,1,2,4}=T2	1



2.4.3 Minimizing the Number of States in a DFA

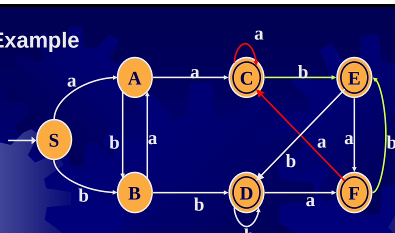
- They are all DFA for regular expression a*, but the later is minimal
- Theory

  Given any DFA, there is an equivalent DFA containing a minimum number of states, and that this minimum-state DFA is unique

- **Equivalent States**

  If s and t are two states, they are equivalent if and only if:
  - s and t are both accepting states or both non-accepting states.
  - For each character a∈Σ, s and t have transitions on a to the equivalent states

Example



C and F are all accepting states. They have transitions on 'a' to C,and have transitions on 'b' to E, so they are equivalent states

S is a non-accepting state and C is an accepting state. They are not equivalent states

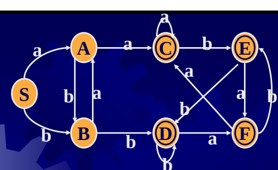
- **Minimizing Algorithm**

  Split the set of states into some un-intersected sets, so states in one set are equivalent to each other, while any two states of different sets are distinguishable.

- First, split the set of states into two sets, one consists of all accepting states and the other consists of all non-accepting states.
- Consider the transitions on each character ‘a’ of the alphabet for each subset, determine whether all the states in the subset are equivalent or the subset should be split.
  - If there are two states s and t in one subset that have transition on ‘a’ that land in different sets, we say that ‘a’ distinguishes the states s and t

---

  - The set of states under consideration must be split according to where their a-transitions land
- Continue this process until either all sets contain only one element (the original DFA is minimal) or until no further splitting of sets occurs.



1.Split into accepting states set and non-accepting states set

{S,A,B} {C,D,E,F}

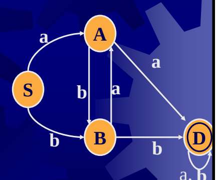
2 Continue to split

{S,A,B}=>{S,B}{A}=>{S}{A}{B}

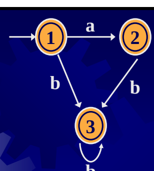
{C,D,E,F}

3 Let D represents {C,D,E,F}

P={S,A,B,D}



- Consider error transitions to an error state that is nonaccepting
  - There are states S and T
  - If S has an a-transition to another states, while T has no a-transition at all (i.e.,an error transition), then ‘a’ distinguishes S and T
  - If S and T both have no a-transition, then they can’t be distinguished by ‘a’

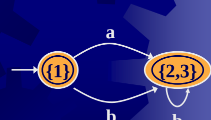


1) All states are accepting:{1,2,3}

2) none of the states are distinguished by b

3) a distinguishes state 1 from states 2 and 3: {1} {2,3}

4) {2,3} cannot be distinguished by either a or b



---

video

- 4-Finite Automata
- 04-01: Lexical Specification
- 04-02: Finite Automata
- 04-03: Regular Expressions into NFAs
- 04-04: NFA to DFA
- 04-05: Implementing Finite Automata