Compiler Technology of Programming Languages

Chapter 3

Scanning

Theory and practice

Prof. Farn Wang

Quick Review of Chapter 2

- AC language
- Formal definition of AC
  - RE and CFG
- CFG
  - Recursion
    - Allows nested structures and repetitions
- Recursive Descent Parsing and Predictive Parsing
- Left recursion elimination (for top-down parsing)
- Build AST
- Code generation

Recursive Descent Parsing

Stmt → ID = Val Expr
| Print ID

```
Stmt() {
   token t=next_token();
   if (t == ID) { match(ID);
                  match(ASSIGN);
                  Val();
                  Expr();}
   else if (t==Print) { match(Print); match(ID);} }
```

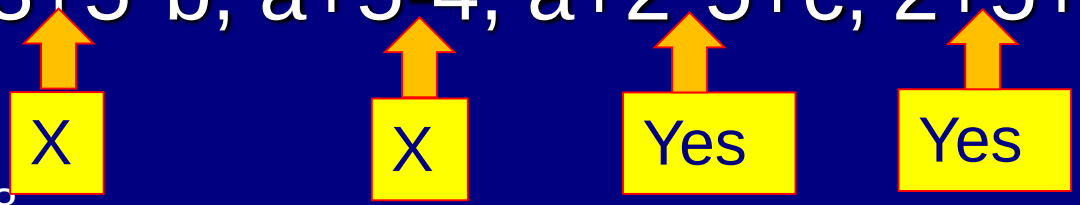
Assignment#1 Q&A

- Overlapped RE's
  - Reserved word: i, f, p
  - Variables names: iab, fcd, pxy, ifp, …

  Here, reserved words are part of variable names. Scanner should obtain the longest match: e.g. iab is an ID, not reserved word i, followed by ID ab.
- Constant folding

  What to fold, what not to ?

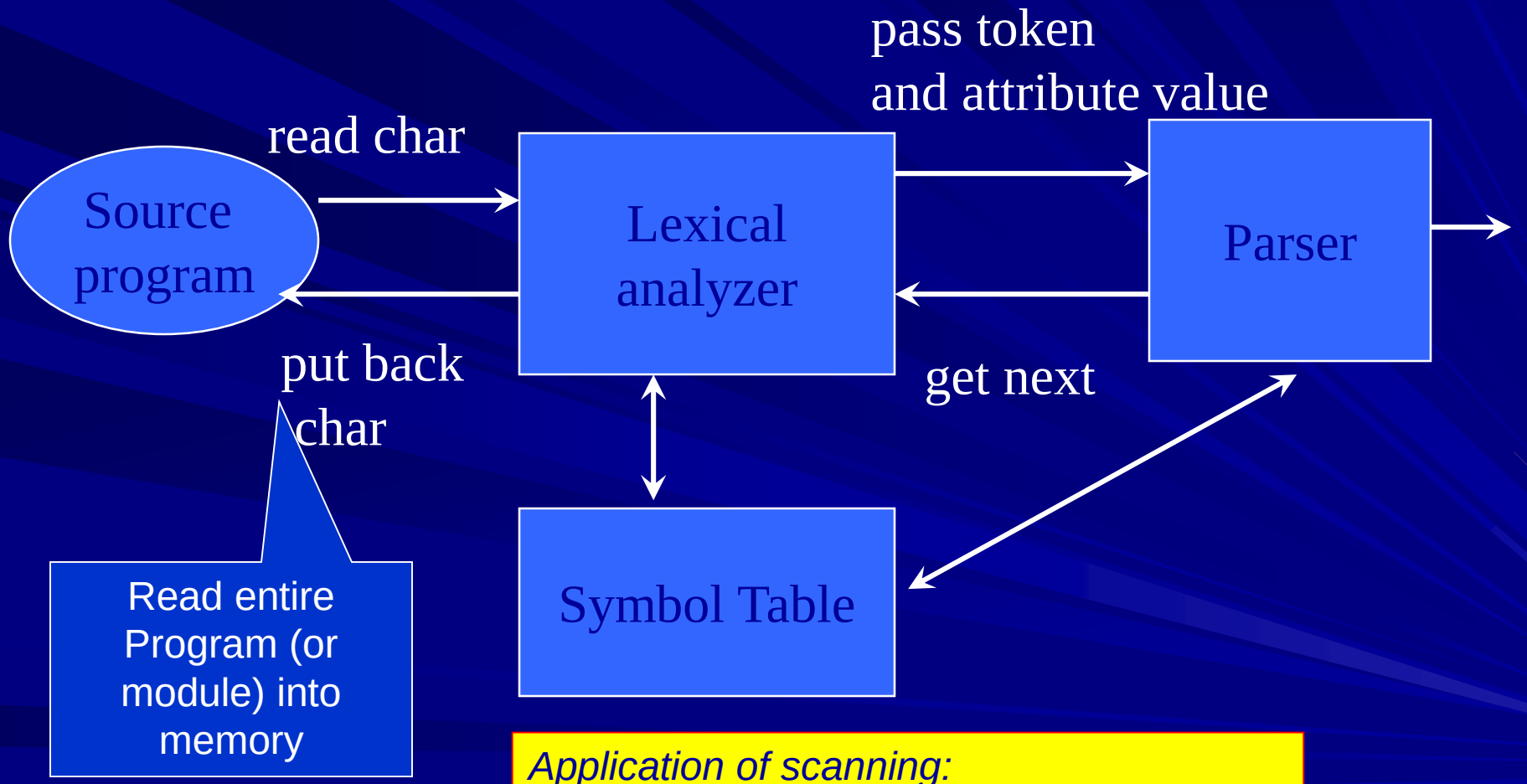
  3+5*b, a+5-4, a+2*5+c, 2+5+c



Ch 3. Scanning – Theory and Practice

- How to **specify** and **implement** a scanner (i.e. lexical analyzer)
  - Using regular expressions (RE) to define tokens
  - Using a scanner generator: Lex/Flex
- Practical considerations
- Theory behind scanner generator:
  - converting RE into DFA and transition table

The Role of a Lexical Analyzer



Application of scanning:
network packets, display of web pages, Interpretation of digital video/audio, text search, ... etc.

Lexical Analyzer

- Functions
  - Grouping input characters into tokens
  - Stripping out comments and white spaces
  - Correlating error messages with the source program
  - Preprocessing (e.g. macro expansion, ..)
- Issues (why separating lexical analysis from parsing)
  - Simpler design
  - Compiler efficiency
  - Compiler portability

Typical Tokens in common PL

- Symbols, operators

  +, -, *, /, =, <, >, ->, …
- Keywords or Reserved words

  if, while, struct, float, int, …
- Integer and Real (floating point) literals

  123, 123.45
- Char (string) literals
- Identifiers
- Comments
- White space

Case Study

- String Literals
  - Quoted strings are most commonly used
  - How about delimiter collision (e.g. want a quotation mark in the string)
  - Can newline characters appear in string?
    - C allows them if they have been escaped (\)
    - Pascal forbids it
    - Ada forbids unprintable characters in strings
  - Are null strings (of length 0) allowed?
    - Pascal forbids them
    - Ada and C allow null strings

Case Study (cont.)

- When blanks are not significant (as in Fortran and Algol68)

DO 5 I = 1.25

DO5I is an ID

DO 5 I = 1, 25

This is a DO loop
So 7 tokens will get generated

Case Study (cont.)

- Should 1. And .10 be legal constant?

  If yes, then how to tell

  Is 1..10 a range ? or two constants 1. and .10?

  Note that 1. And .10 are both allowed in Fortran and C, but not allowed in Pascal and Ada – because Pascal and Ada support ranges.

Why Formal Specification?

- Precise definition of tokens is necessary to ensure lexical rules are properly enforced.
- Contract between programmers and compiler implementers.
- Allowing flaws to be discovered before the design is completed.
- Allowing automatic scanner generation.
- A trend towards nonprocedural programming.

Regular Expressions (RE)

- Why RE?
  - Suitable for specifying the structure of tokens in programming languages
- Basic concept

  A RE defines a set of strings (called *regular set*)
  - Vocabulary/Alphabet : a finite character set V
  - Strings are built from V via catenation
  - Three basic operations: concatenation, alternation (|) and closure (*).
  - ∅ is a RE denoting the empty set
  - λ is a RE denoting the set that contains only the empty string.

Examples

- a | b denotes the set {a,b}
- (a|b) (a|b) denotes the set {aa, ab, ba, bb}
- a* denotes { λ, a, aa, aaa, …}
- (a|b)* denotes all strings of a's and b's

Regular Definition

- We may give names to regular expressions and to define regular expressions using these names

  letter = A | B | C …. | a | b | c …. | z

  digit = 0 | 1 | 2 | … | 9

  id = letter (letter | digit) *

  digits = digit digit*

- The identifier in Pascal can be defined as

  letter (letter | digit) *

Regular Definition

- Question: How is regular definition different from CFG?
- Answer:

  CFG allows recursion – a non-terminal can show up in the right hand side of itself.

  Regular Definition does not allow recursion. It is like Macro definition.

Common notations and Metacharacter

- * means repeat zero or more times
- + means repeat one or more times
- ? means repeat zero or one time

P+ == PP*

Not(A) == V – A /* A is a set of char

Not(S) == V* - S /* S is a set of strings

A^K == AA …A (k copies)

P? == P | λ

Common notations (cont.)

- [abc] means a | b | c
  - [] forms a character class which matches any char listed
  - A negate class can be specified by an upper arrow e.g. [^ab] matches any char except a and b.
- [a-z] denotes the RE a | b | c …. | z
- () used for grouping, e.g. (a|bc)
- A dot (.) matches any character, except a new line

Common notations (cont.)

- [abc] means a | b | c
  - [] form a character class which matches any char listed
  - A negate class can be specified by an upper arrow e.g. [^ab] matches any char except a and b.
- [a-z] denotes the RE a | b | c | … | z
- () used for groupin
- A dot (.) matches a
  except a new line

How about precedence?
ab|c → ab or c
b|ac* → b or a or ac+
*** > catenation > alternation**

Special Characters

- The following characters have special meaning when they are inside a char class.
  - { start of macro name
  - } end of macro name
  -] end of a char class
  - - range of characters
  - ^ negative char class
  - \ take away special meaning of the next char
- Other metacharacters, such as *,+,? are not special in a char class. For instance, [*?] matches * or ?.

Notation Examples

- [a-z]
- [z-a9-0] this is ok
- [a\-z]
- [a^b]
- [^a-z]
- [0-z] non-portable warning
- ({letter})*
- ab*

Exercise

D= (0 | 1 | 2 …| 9) L=(A |… | Z)

D=[0-9]
L=[A-Z]

a) A Java or C++ single line comment that begins with // and ends with Eol

comment = // [^\n]*\n

// (.)*\n

a) A fixed decimal literal

decimal = D+ "•" D+

D* "•" D*

b) An identifier, composed of L,D, and _ that begins with a letter, ends with a letter or digit

ident = L (L | D)* (_+ (L | D)$^{+}$)*

How about
L (L|D|_)*(L|D)+

c) A comment delimited by ## that allows # in the body

comments = ##([#]?[^#])* ##

How about
C identifiers?

(_|L)(_|L|D)*

Exercise (cont.)

- Token While

  while = “while”

- White space

  note: newline is often treated differently to track line number

  WhiteSpace = [\t]+

- Define floating point constants. It can be any of the following forms

  1.25e12 .25 .12e+10 12.5e-5 12e6

  (D+ (\. D+)? (e (+|-)?D+)?) |
  (D+)? (\. D+) (e (+|-)?D+)?

How about (D+)?[\.](D+)?(e(+|-)?D+)?

How about (D+)?(\.D+)?(e(+|-)?D+)?

How about 125 ?

Exercise (cont.)

- Token While

  while = “while”

- White space

  note: newline is often treated differently to track line number

  WhiteSpace = [\t]+

- Define floating point constants. It can be any of the following forms

  1.25e12 .25 .12e+10 12.5e-5 12e6

(D+)(e(+|-)?D+) | (D+ (\. D+) (e (+|-)?D+)?) | (D+)? (\. D+) (e (+|-)?D+)?

Non-regular set

- RE can denote an unspecified number of repetitions of a given construct. Its best use is for describing identifiers, constants, … etc.
- RE can not be used to describe balanced or nested structures, such as nested loops, nested if-then-else.
- Example:

  $\{ [^i]^i \mid i \geq 1 \}$ ➔ [[[ ]]], [[[[ ]]]], …

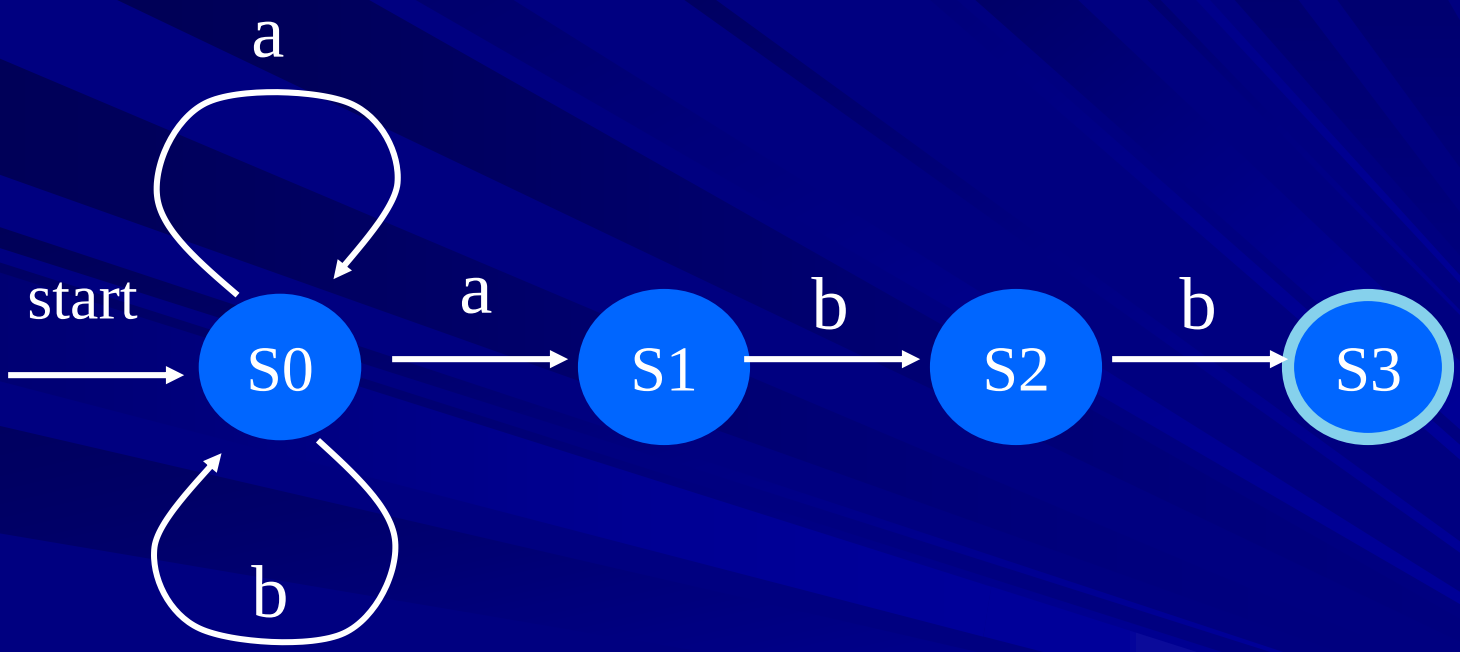
  can we define it as

  R = (“[“ R* “]”) ?

Finite Automata and Scanners

- A Finite Automation (FA) can be used to recognize the tokens specified by a RE
- Nondeterministic Finite Automation (NFA)
  - A set of states
  - A set of input symbols
  - A transition function, *move()*, that maps state-symbol pairs to sets of states.
  - A start state S0
  - A set of states F as accepting (Final) states.

Example (*RE=(a|b)*abb*)



The set of states = {S0,S1,S2,S3}
Input symbol = {a,b}
Start state is S0, accepting state is S3

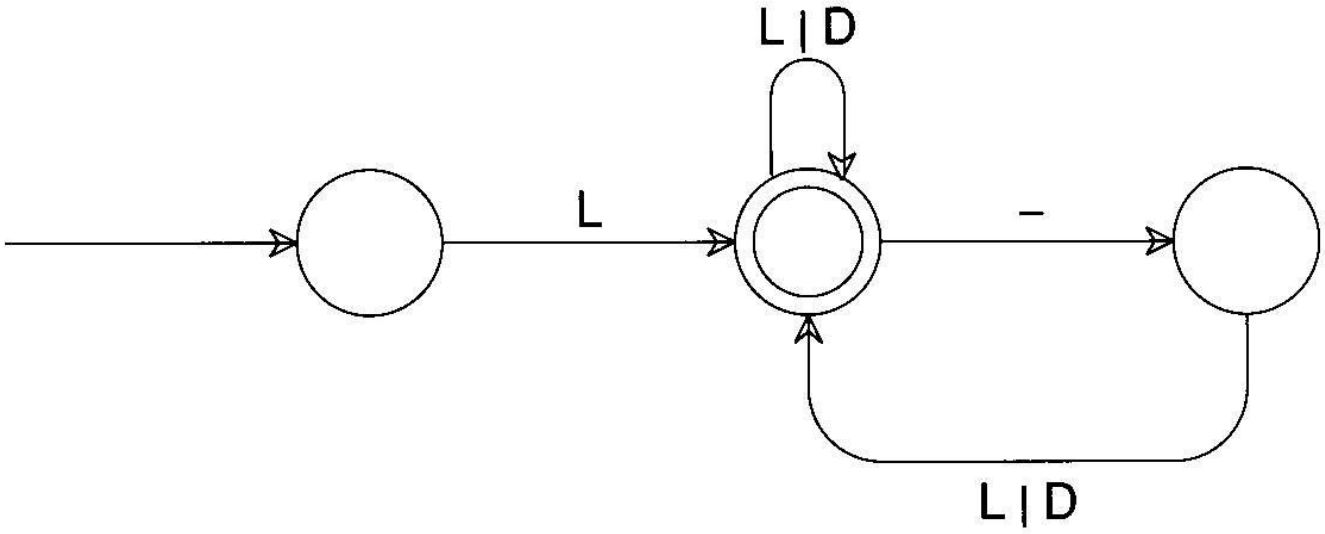
Transition Function

- Transition function can be implemented as a transition table.

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	--	{2}
2	--	{3}
3	--	--

- **Example**

$$ID = L(L|D)^*(\_(L|D)^+)^*$$



- Non-deterministic Finite Automata (NFA)
  - An NFA accepts an input string *x* iff there is a path in the transition graph from the start state to some accepting (final) states.
  - The language defined by an NFA is the set of strings it accepts.
- Deterministic Finite Automata (DFA)

  A DFA is a special case of NFA in which
  - There is no λ-transition
  - Always have unique successor states.

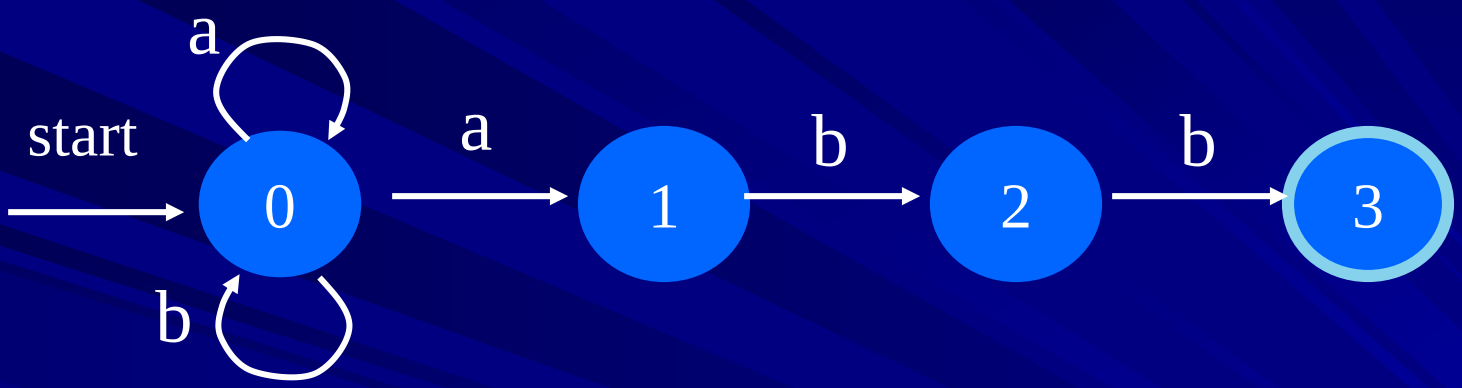
– How to simulate a DFA

```
s := s0;
c := nextchar;
while ( c <> eof) do
        s := move(s, c);
        c := nextchar;
end
if (s in F) then return “yes”
```

Conversion of NFA to DFA

- Why?
  - DFA is difficult to construct directly from RE's
  - NFA is difficult to represent in a computer program and inefficient to compute
- Conversion algorithm: subset construction
  - The idea is that each DFA state corresponds to a set of NFA states.
  - After reading input a1, a2, …, an, the DFA is in a state that represents the subset T of the states of the NFA that are reachable from the start state.

NFA to DFA conversion



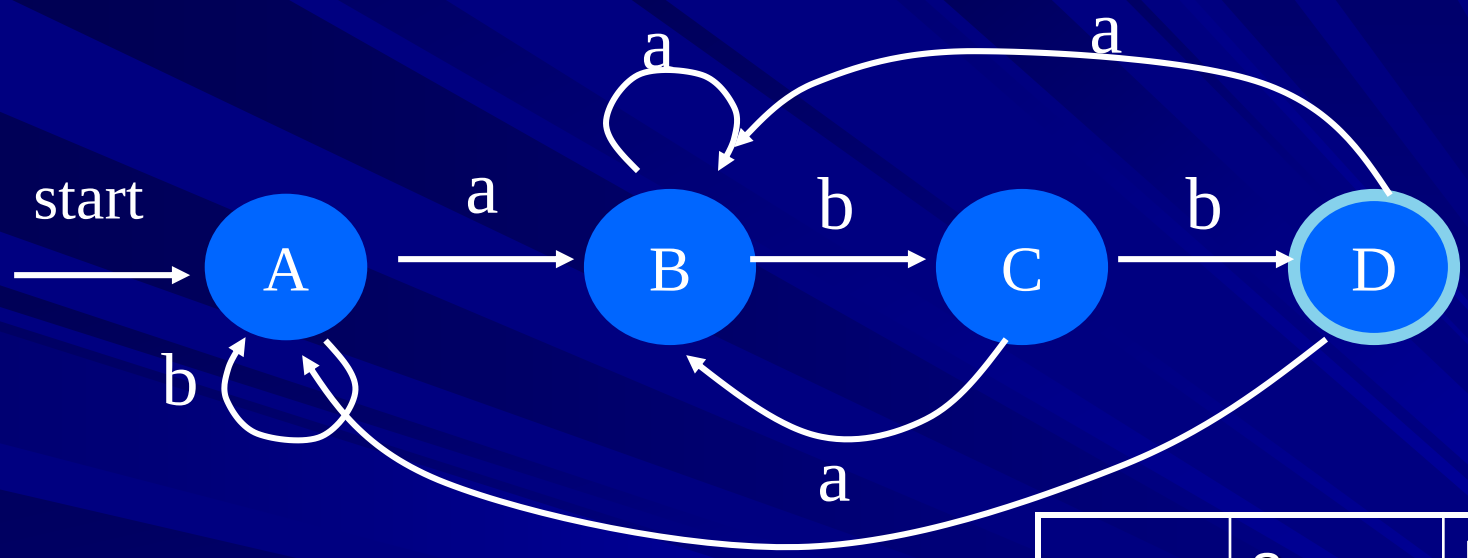
(0,a) = {0,1}
(0,b) = {0}
({0,1}, a) = {0,1}
({0,1}, b) = {0,2}
({0,2}, a) = {0,1}
({0,2}, b) = {0,3}

New states

A = {0}
B = {0,1}
C = {0,2}
D = {0,3}

	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)

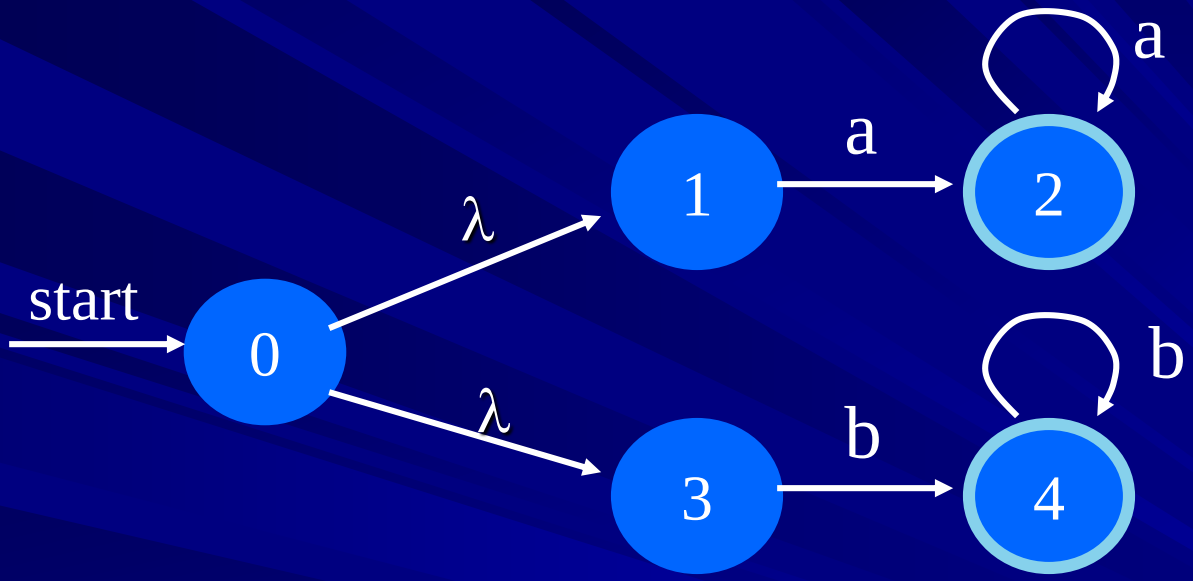


RE = =*(a|b)*abb*

abababb
aabbabb
abbbbaabb

	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)

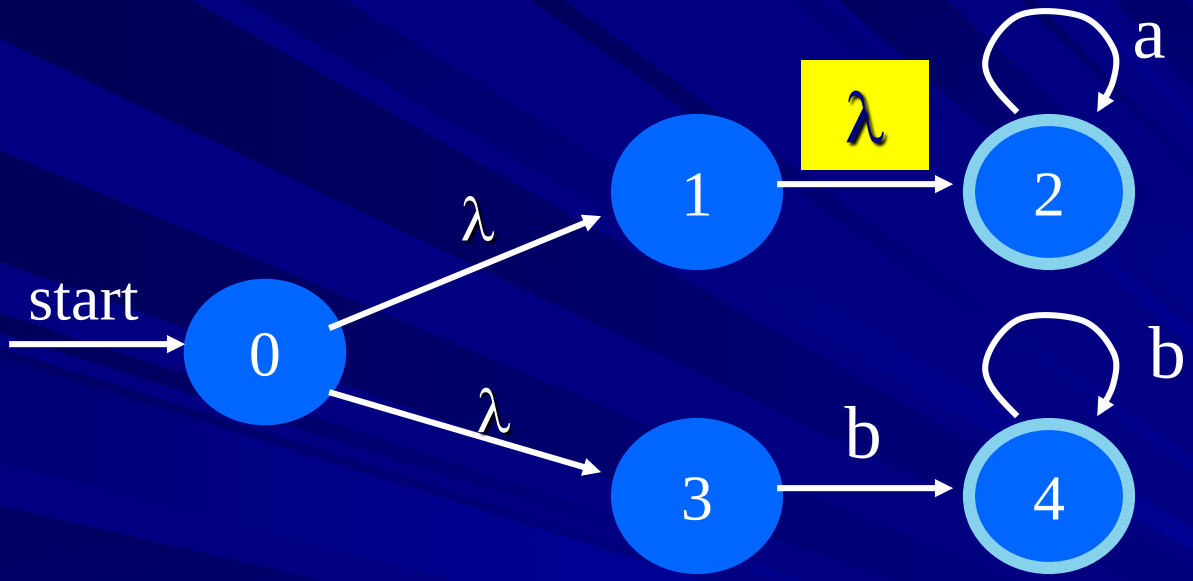


How about λ-transition?

Due to λ-transitions, we must compute λ-closure(S) which is the set of NFA states reachable from NFA state S on λ-transition, and λ-closure(T) where T is a set of NFA states.

Example: λ-closure (0) = {0,1,3}

NFA to DFA conversion (cont.)

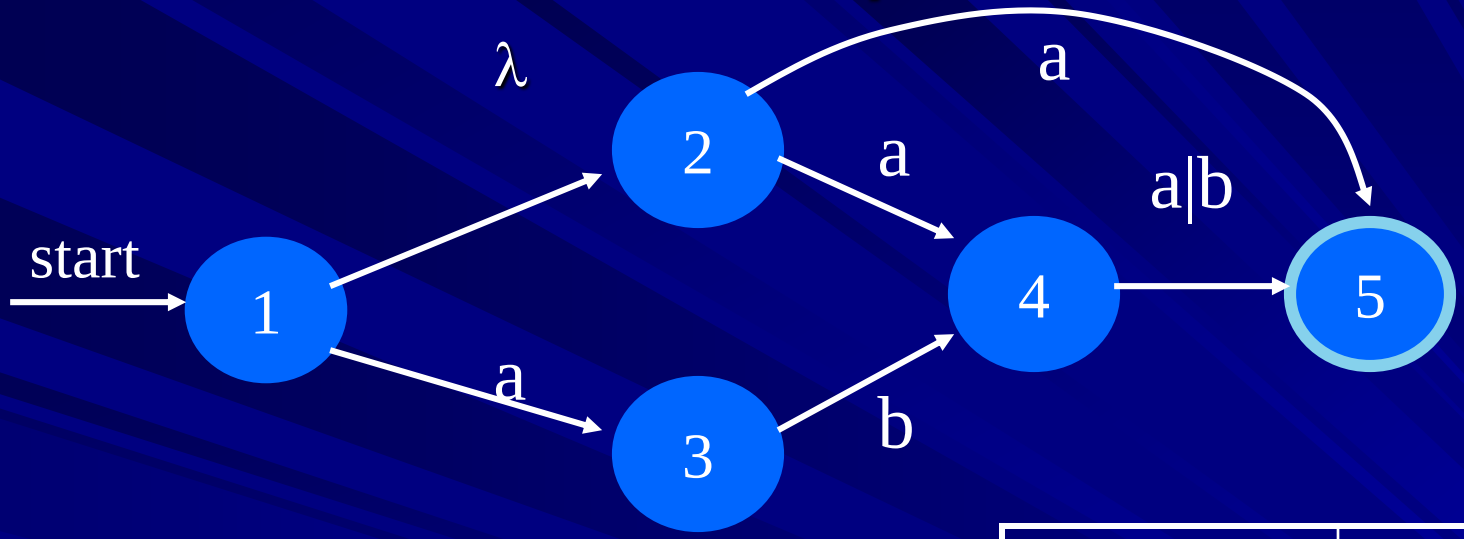


λ-closure (0) = {0,1,2,3}

Subset Construction Algorithm

```
Dstates := λ-closure (s0)
While there is an unmarked state T in Dstates do
begin
    mark T;
    for each input symbol a do
    begin
        U := λ-closure ( move(T,a) );
        if  U is not in Dstates then
              add U as an unmarked state to Dstates;
        Dtran [T, a] := U;
    end
end
```

Example

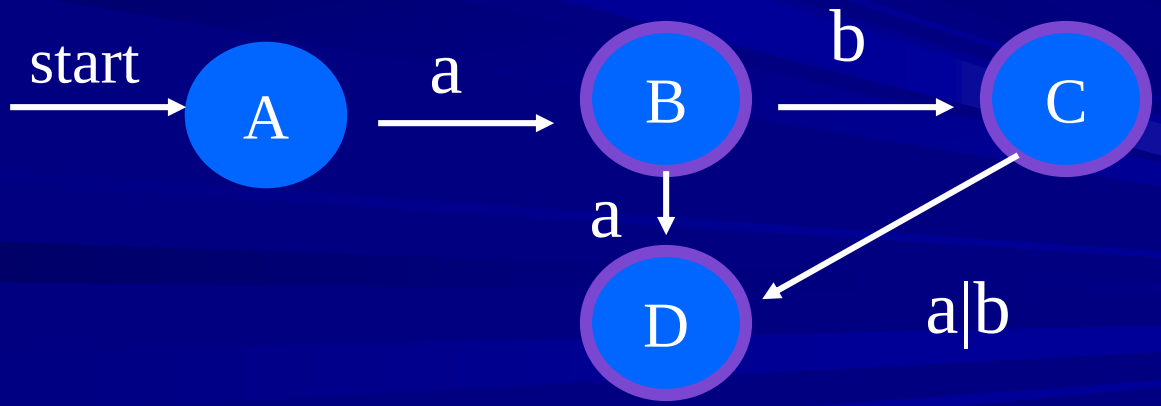


Dstates := λ-closure(1) = {1,2}
U:= λ-closure (move({1,2}, a)) = {3,4,5}
Add {3,4,5} to Dstates
U:= λ-closure (move({1,2}, b)) = {}
λ-closure(move({3,4,5}, a)) = {5}
λ-closure (move({3,4,5}, b)) = {4,5}
λ-closure (move({4,5}, a)) = {5}
λ-closure (move({4,5}, b)) = {5}

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--

DFA after conversion

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--



Transition Table and Driver

- The transition diagram can be naturally implemented by a transition table.

  Table[state] [c]

  The driver looks as follows:

```
While (not eof) {
    new_state = Table [current_state ] [c ]
    c = nextchar();
    current_state = new_state;
}
```

Practical Considerations

- Reserved words
- Compiler Directives and Listing Source Lines
- Entry of Identifiers into the Symbol Table
- Scanner Termination
- Lexical Error Recovery

Reserved Words

- When key words are not reserved words (such as in PL/1)

example 1:

IF THEN THEN THEN = ELSE ELSE ELSE = THEN;

Which THEN is an identifier?

Which THEN is the key word?

example 2:

Declare (arg1, arg2, arg3, …)

Is Declare a subroutine name or is it the key word?

Reserved Words (cont.)

- Assume begin and end are not reserved in Pascal.

example 3:

```
begin
  begin;
  end;
  end;
  begin;
end
```

How to parse this code fragment?

For example 1 and 2, ambiguity can be solved by multiple characters look ahead.

Compiler Directives

- Compiler options e.g. optimization, profiling, etc.
  - handled by scanner or semantic routines
  - Complex pragmas are treated like other statements.
- Source inclusion
  - e.g. #include in C
  - handled by preprocessor or scanner
- Conditional compilation
  - e.g. #if, #endif in C
  - useful for creating program versions

Entry of ID into the Symbol Table

- For simple language with only global variables, it is common to have the scanner enter an ID into the symbol table.
- How about block-structured languages?

```
{ int a;
        …
    { int a; }
}
```

Postpone and wait for the semantic routine to enter.

Scanner Termination

- Return **Eof** when the end of the input file is reached. (may need to define the **Eof** token)
- What if a scanner is called after the end of file is reached?
  - We may continue to return **Eof.** This is better than crash with a fatal error since it allows the parser to handle termination cleanly.

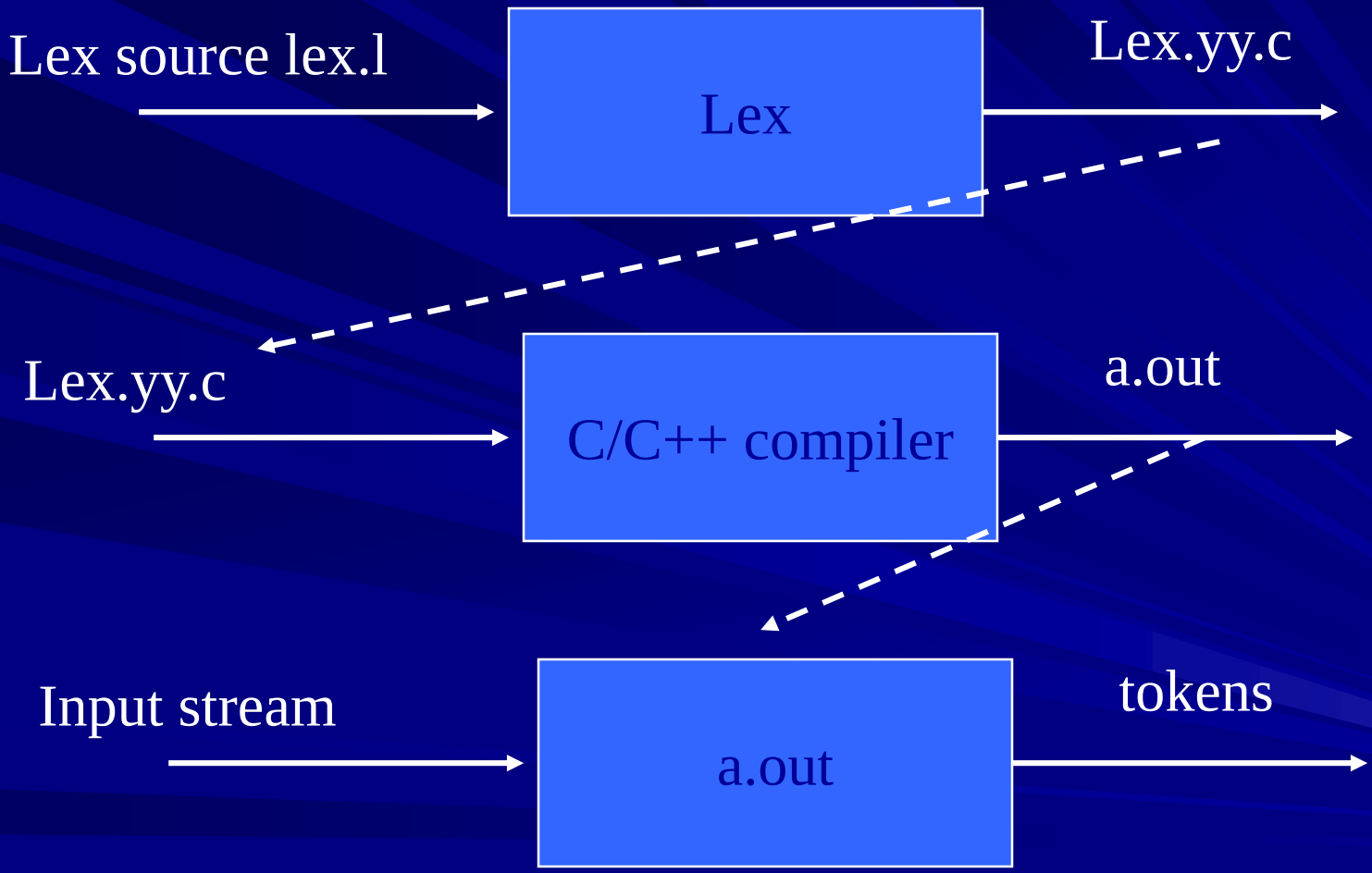
Lexical Error and Recovery

- Error detection
- Error reporting
- Error recovery
  - Delete the current character and restart scanning at the next character
  - Delete the first character read by the scanner and resume scanning at the character following it.
  - How about runaway strings and comments?

LEX – A Language for Specifying Lexical Analyzers

- Lex is a lexical analyzer generator
  - Implemented by Lesk and Schmidt of Bell Lab initially for Unix
  - Not only a table generator, but also allows “actions” to associate with RE’s.
  - Lex is widely used in the Unix community
  - Lex is not efficient enough for production compilers, however, Flex (Fast Lex) is an improved version.

Lex Usage


Lex source lex.l
Lex
Lex.yy.c
Lex.yy.c
C/C++ compiler
a.out
Input stream
a.out
tokens

Lex Specification

- Declaration
  - Variables, constants
  - Regular definitions to define character class and auxiliary regular expressions
- Translation rules
  - Regular expression-1 {action 1}
- Auxiliary procedures
  - Routines needed by actions
  - Symbol table routines

Example

```
%{
#define THEN 5
#define ID    100
%}
WS      [ \t]+
letter  [A-Za-z]
digit   [0-9]
id      {letter}( {letter} | {digit})*
%%
{WS}    {}
{id}    {yylval = insert_id(); return(ID);}
%%
Insert_id() {…}
```

Definition section
%%
Rules section
%%
Subroutine section

To pass attributes to the parser, a global variable yylval is often used.

Overlapped Regular Expressions

Lex allows RE's to overlap. In the case of overlapped RE's, two rules apply:

- Longest possible match, for example, how to get "<=" token rather than "<" and "=".
- Order of rules. If two RE's match the same string, the earlier rule is preferred.

So ***if8*** is an identifier while ***if*** is a reserved word. (assuming if is defined as a token by RE).

Special Variables/Procedures

- **yytext** where token text is stores
- **yyleng** length of the token text
- **yylineno** the current line number
- **yylex()** name of procedure for the lex generated scanner
- **yywrap** A user supplied function. It returns 1 when no more input to process, otherwise, return 0
- **yyin, yyout** input and output files

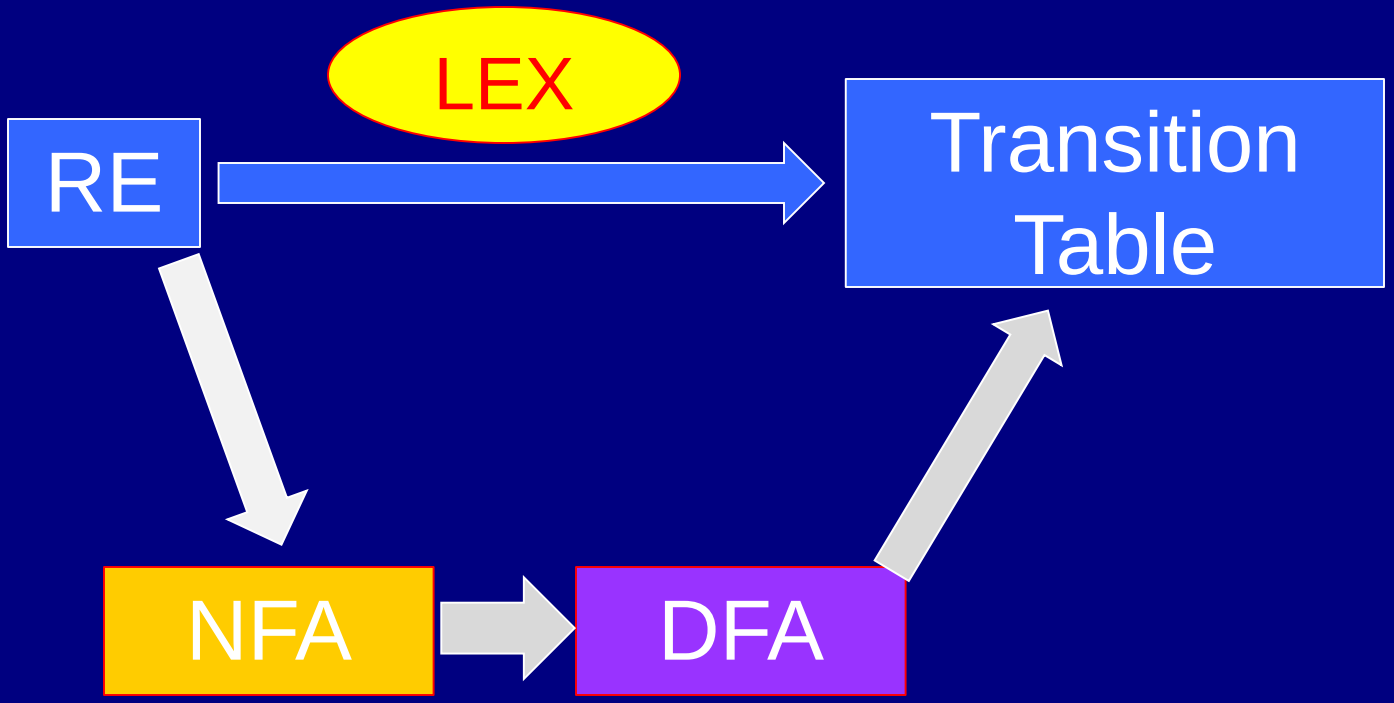
Additional Notes

- [^a-z] negate the char class will also match a new line “\n”.
  - True in Lex and Flex, but not necessarily true in other implementation of RE.
- ECHO copies yytext to output
- If you use Lex, link with lex library –lf, if you use Flex, link with library –lfl.
- EOF is not handled by RE, it is signaled by having yylex() to return integer 0.

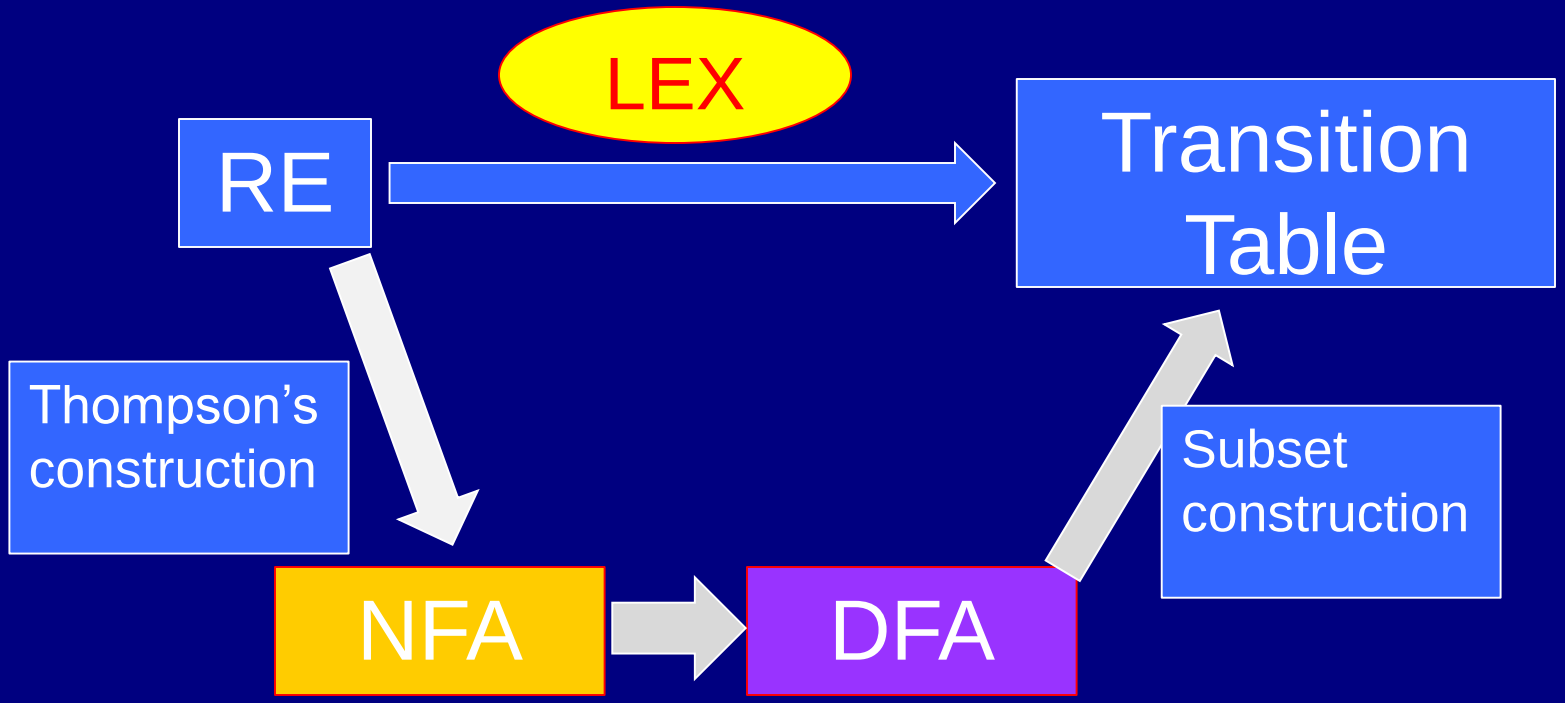
Lexical Analyzer Generator

- Lexical analyzer generator is to transform RE into a state transition table (i.e. Finite Automaton)
- Theory of such transformation
- Some practical consideration

Lexical Analyzer Generator

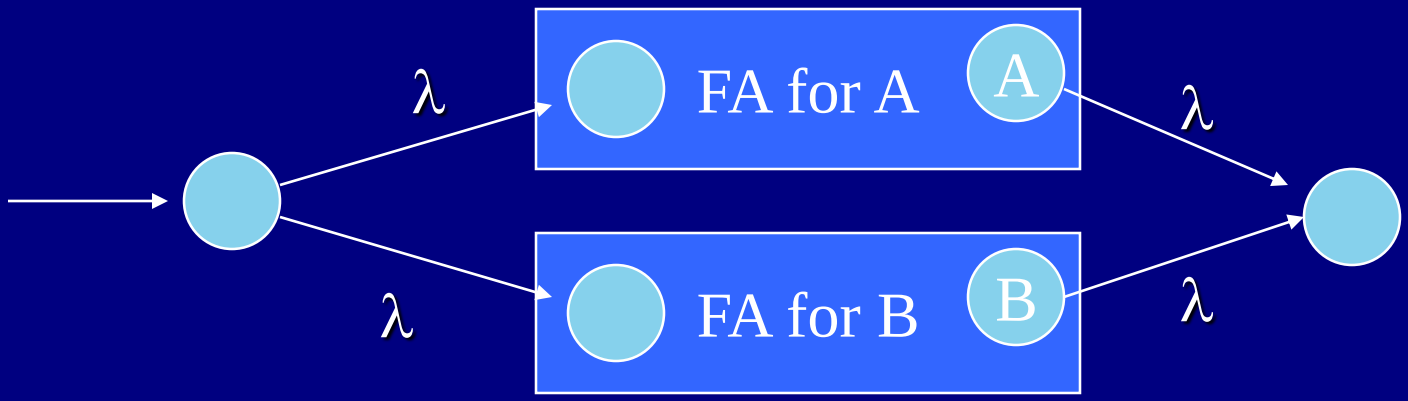


Lexical Analyzer Generator



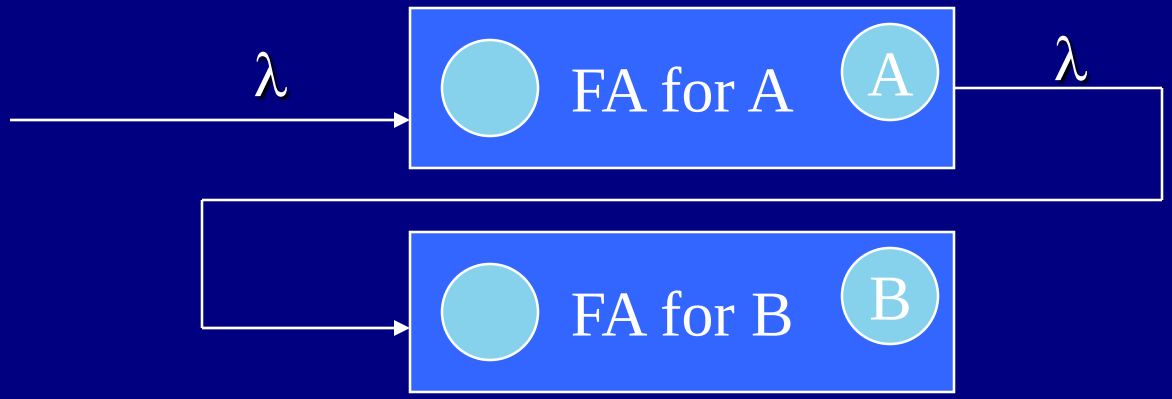
Map RE to NFA

- Three basic operations
  - All RE's are built out of atomic regular expressions by using concatenation, alternation and closure.
- Constructing A|B



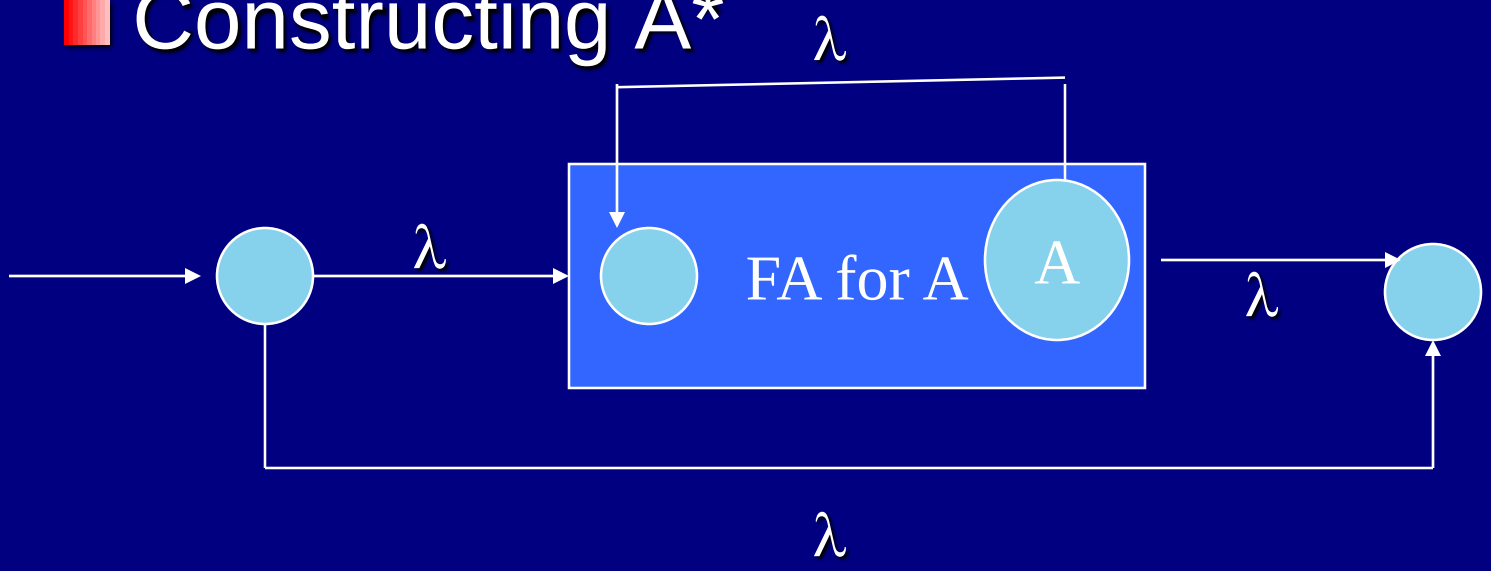
Map RE to NFA

- Constructing AB



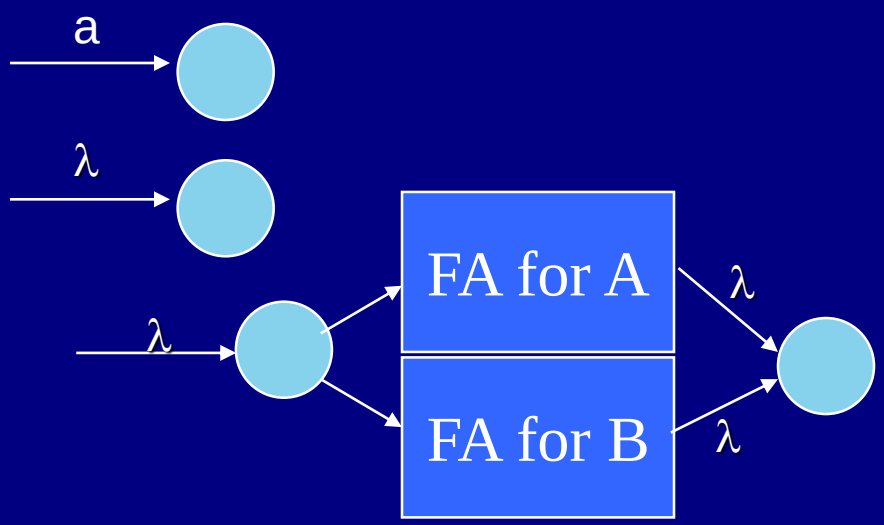
Map RE to NFA

- Constructing A*

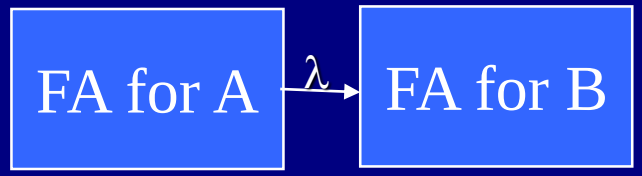


Summary: Map RE to NFA

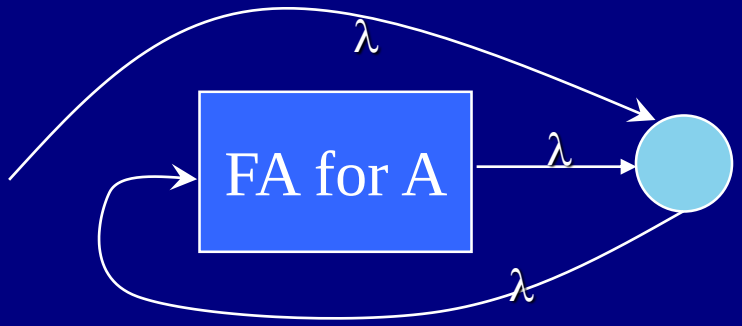
1) a
2) λ
3) A|B



4) AB



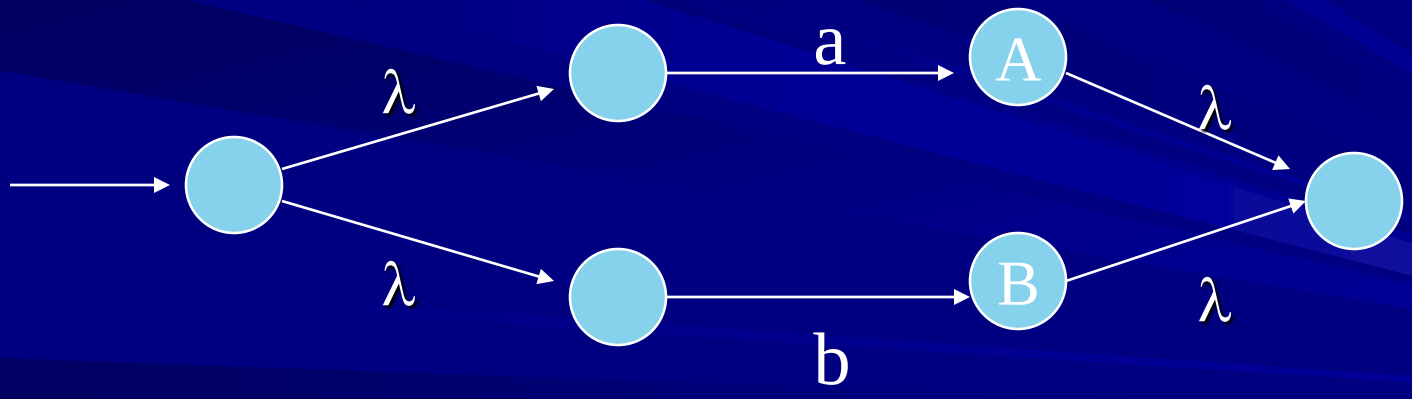
5) A*



Example

Constructing NFA for regular expression
r = (a|b)*abb

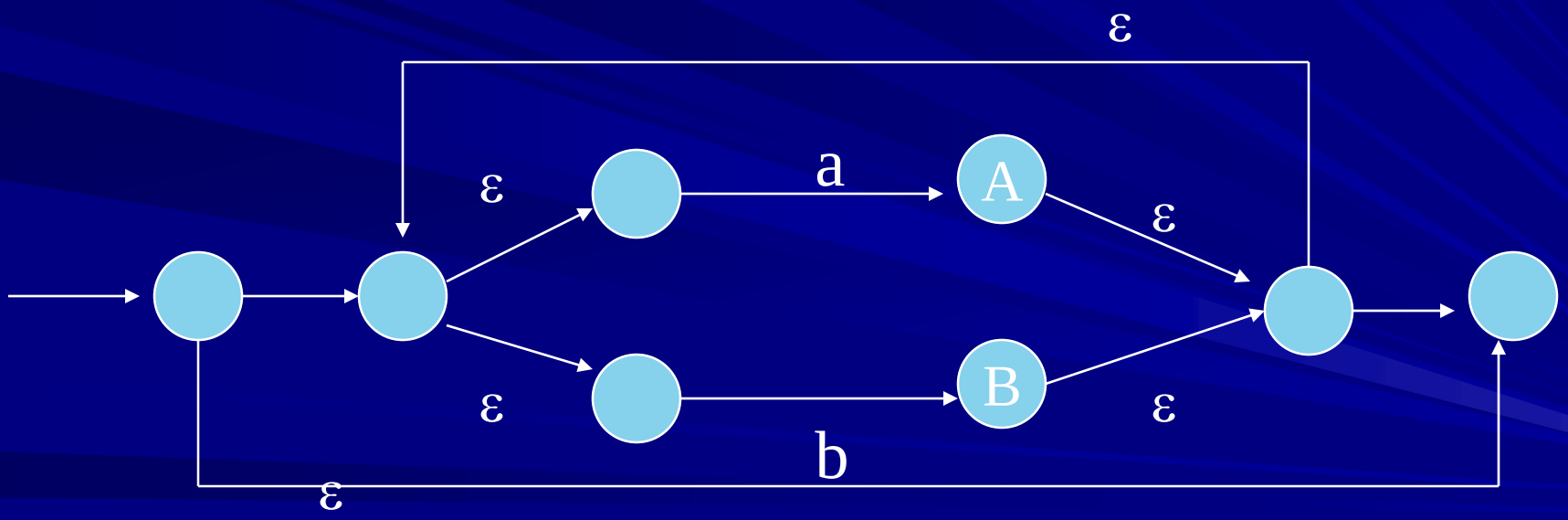
Step 1: constructing a | b



Example

Constructing NFA for regular expression
r = (a|b)*abb

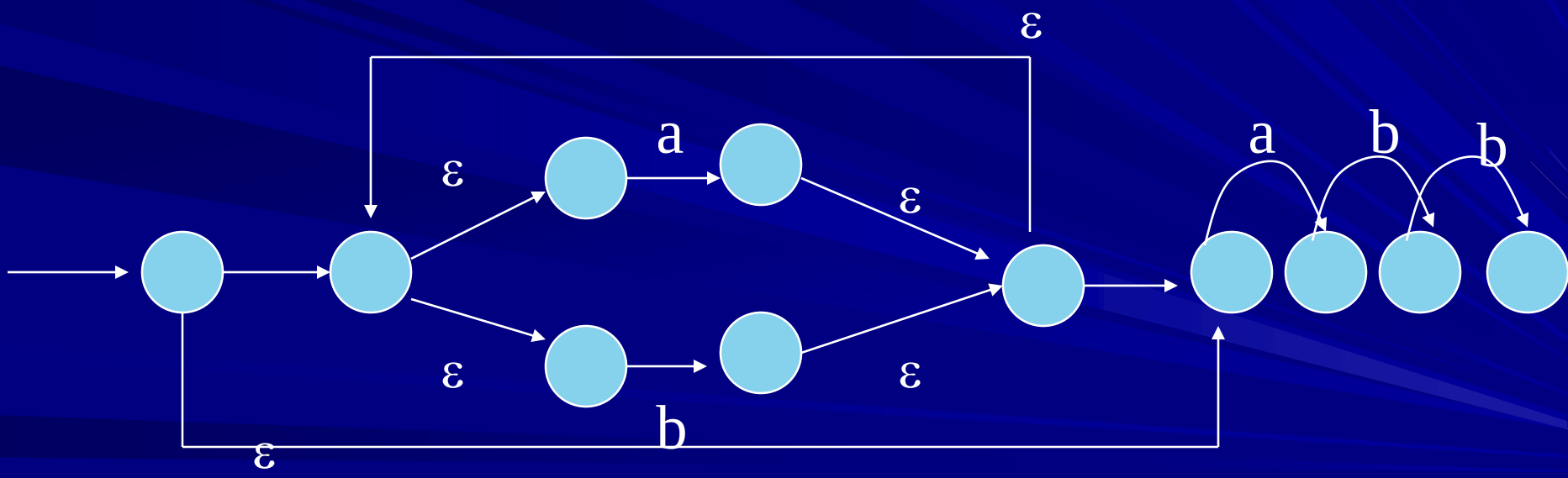
Step 2: constructing (a | b)*



Example

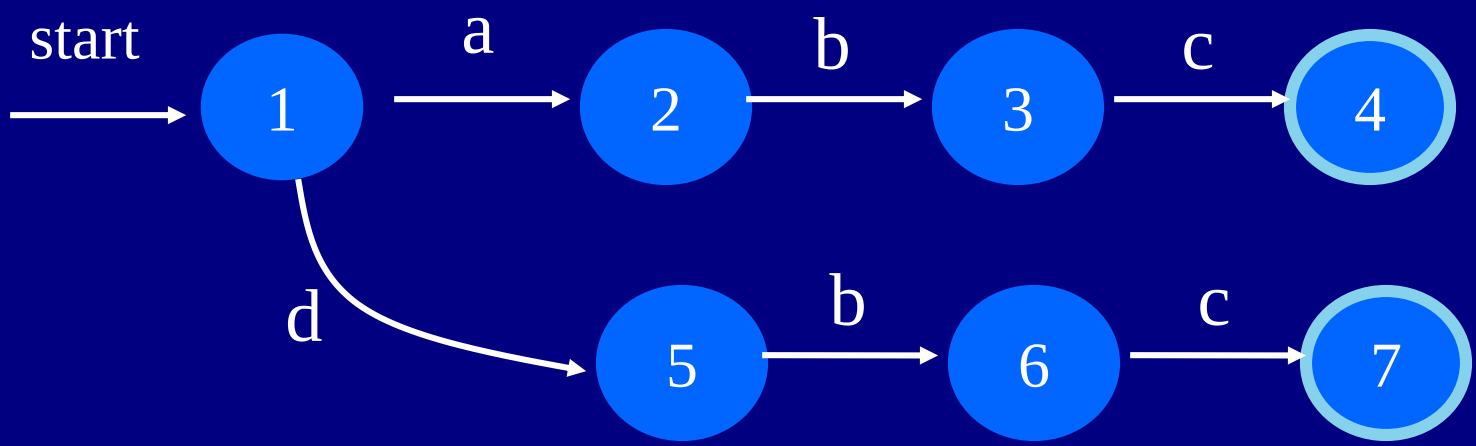
Constructing NFA for regular expression
r = (a|b)*abb

Step 3: catenate with abb

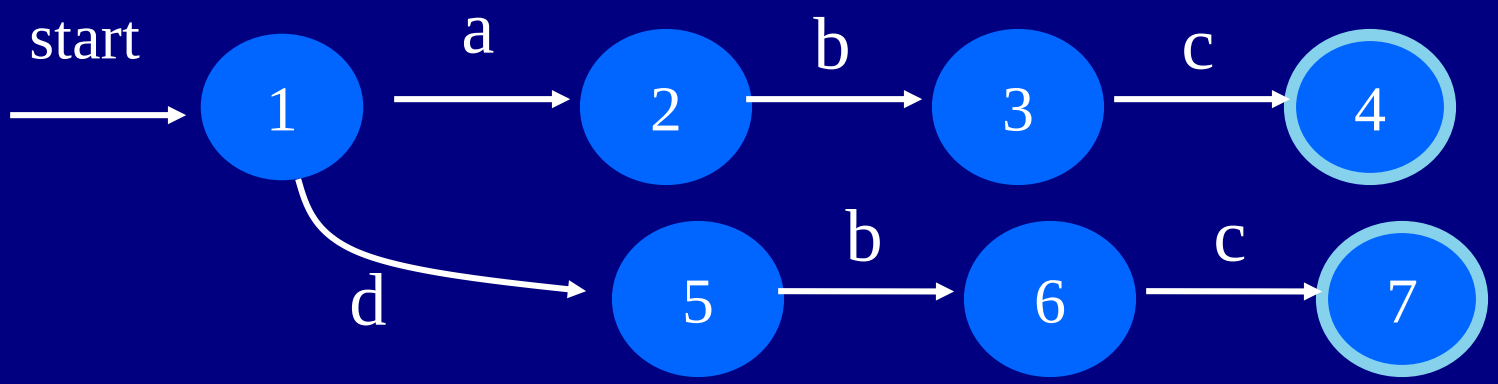


Optimizing Finite Automata

- Minimizing the number of states
  - For every DFA, there is a unique smallest equivalent DFA (that accept the same set of strings).



Optimizing Finite Automata



	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

State 2 and state 5
State 3 and 6
State 4 and 7

Are equivalent!

Optimizing Finite Automata

- We may begin by trying the most aggressive (or optimistic) merge by creating only two states: final state and non-final state. We then split the states.

- Algorithm:

1. Repeat
   2. Let S be any merged state {s1, s2, ..,sn}, and c be any input symbol.
   3. Let t1, t2, … tn, be the successor state to {s1, .. sn} under c, if t1, .. tn do not all belong to the same merged state then split S into new states so that si and sj remain in the same merged state iff ti and tj are in the same merged state.
4. Until no more splits are possible.

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

Start with two states
Non-final = {1,2,3,5,6}
Final = {4,7}

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

Start with two states
Non-final = {1,2,3,5,6}
Final = {4,7}

Split non-final as
{1} {2,3,5,6}
Final = {4,7}

For input b, the successor state for 2, and 3 are not in the same merged state, so we nee to split more

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				

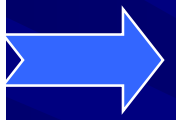
Non-final is
{1} {2,3,5,6}
Final = {4,7}

Non-final is
{1} {2,5} {3,6}
Final = {4,7}

No more split! Job done.

Optimizing Finite Automata

	a	b	c	d
1	2			5
2		3		
3			4	
4				
5		6		
6			7	
7				



	a	b	c	d
A	B			B
B		C		
C			D	
D				

A = {1}
B = {2,5}
C = {3,6}
D = {4,7}

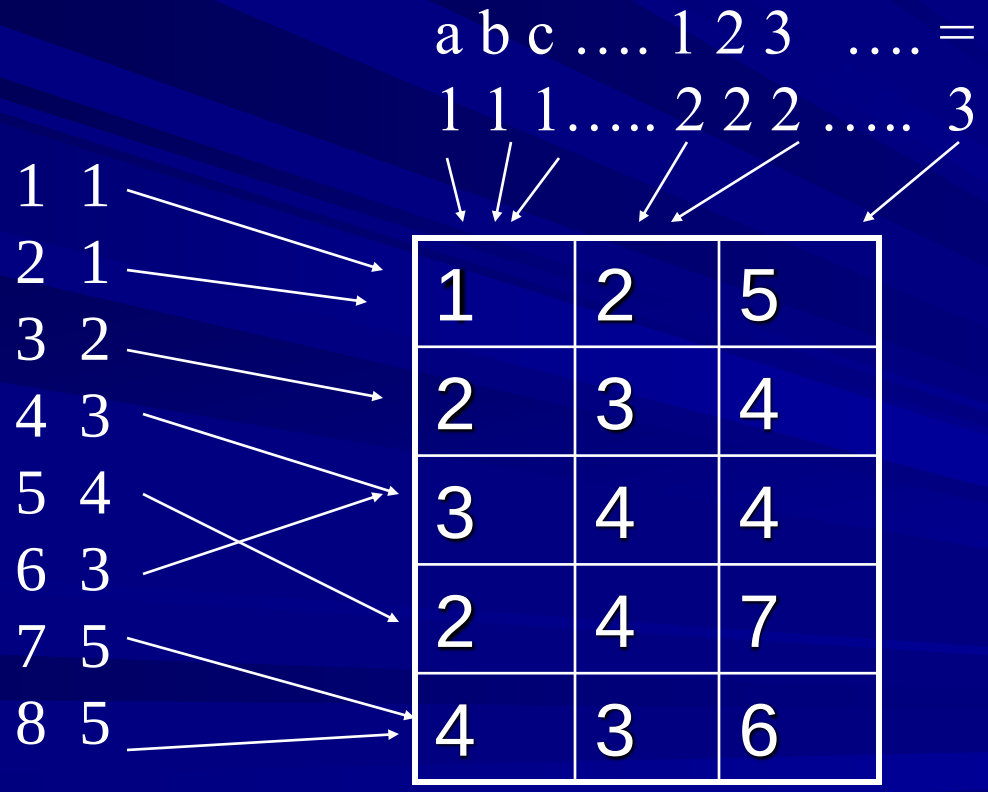
Optimizing Finite Automata

- Table Compaction
  - Two dimensional arrays provide fast access
  - Table size may be a concern (10KB to 100KB)
  - Table compression techniques
    - Compressing by eliminating redundant rows
    - Pair-compressed transition tables

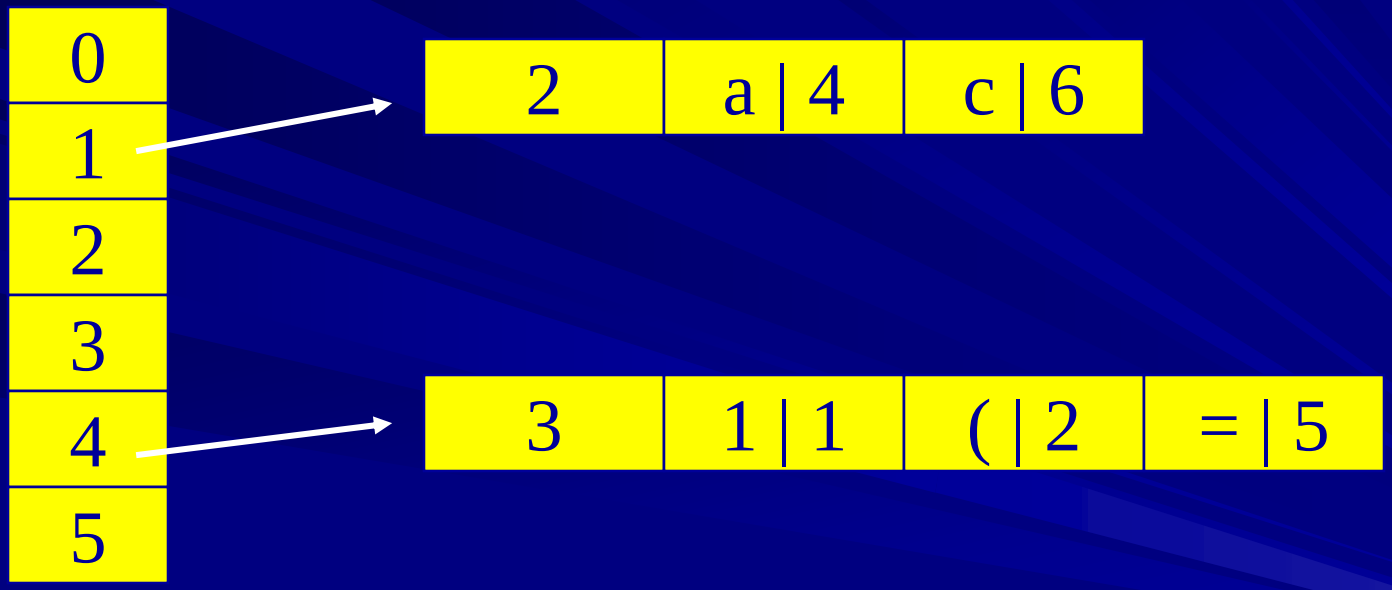
- A typical transition table has many identical columns and some identical rows.

	a	b	c	…	1	2	…	=
1	1	1	1		2	2		5
2	1	1	1		2	2		5
3	2	2	2		3	3		4
4	3	3	3		4	4		4
5	2	2	2		3	3		4
6	3	3	3		4	4		4
7	4	4	4		3	3		6
8	4	4	4		3	3		6

We may create a much smaller transition table with indirect row and column maps. Table is now accessed as T[rmap[s], cmap[c]].



Sparse table techniques



Summary

- Finite Automata, NFA, DFA
- Converting NFA to DFA – subset construction
- From RE to NFA – 3 basic operations
- Time space tradeoffs
- Optimizations: minimize the number of states and table compression

Summary

- Learn how to specify and implement a lexical analyzer
- Precise specification: RE
- Map RE to transition diagram, and map transition diagrams to code
- LEX – A pattern-directed language and a scanner generator

Homework / Assignment#2

- Using Lex/Flex to write a scanner for a subset of language C
- Define tokens in RE's
- Handle identifiers and interact with a symbol table