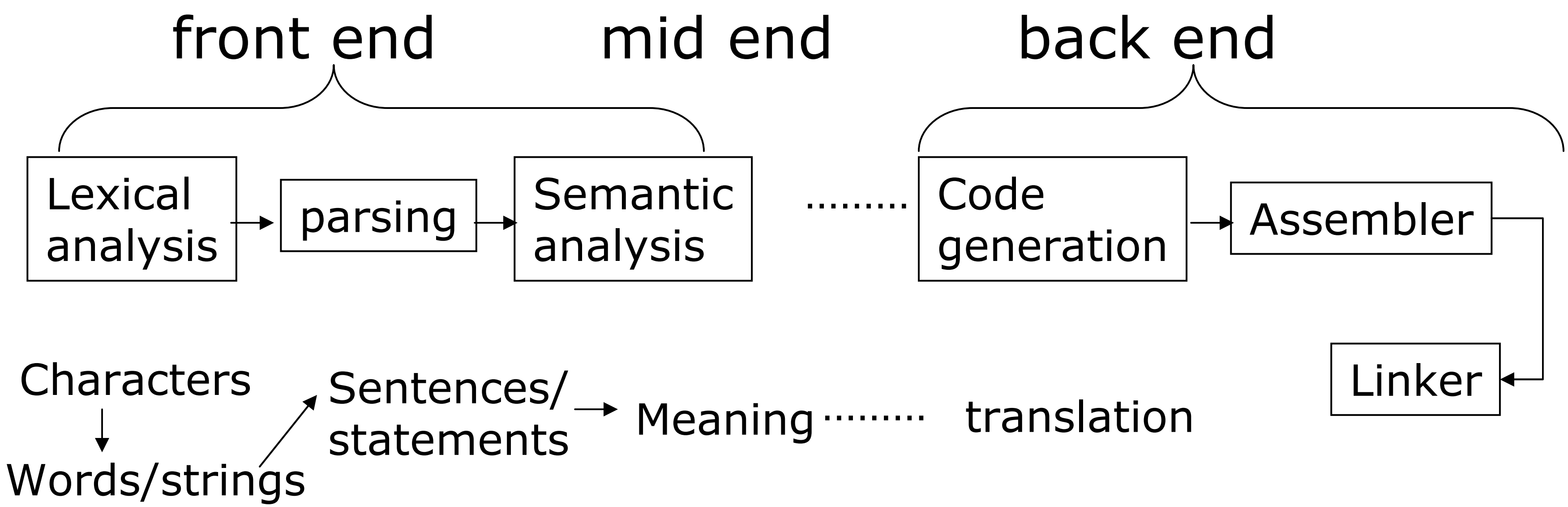
Lexical Analysis

Scanners, Regular expressions, and Automata

Phases of compilation

Compilers

Read input program → optimization → translate into machine code



Lexical analysis

- The first phase of compilation
  - Also known as lexer, scanner
  - Takes a stream of characters and returns tokens (words)
  - Each token has a “type” and an optional “value”
  - Called by the parser each time a new token is needed.

- if (a == b) c = a;

IF
LPARAN
<ID “a”>
EQ
<ID “b”>
RPARAN
<ID “c”>
ASSIGN
<ID “a”>

Lexical analysis

- Typical tokens of programming languages
  - Reserved words: class, int, char, bool,…
  - Identifiers: abc, def, mmm, mine,…
  - Constant numbers: 123, 123.45, 1.2E3…
  - Operators and separators: (,), <, <=, +, -, …
- Goal
  - recognize token classes, report error if a string does not match any class

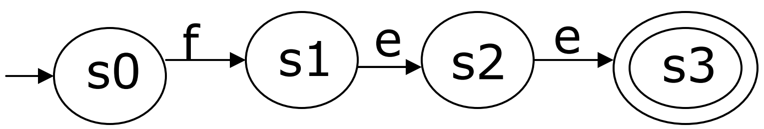
Each token class could be

A single reserved word: **CLASS, INT, CHAR,…**
A single operator: **LE, LT, ADD,…**
A single separator: **LPARAN, RPARAN, COMMA,…**
The group of all identifiers: **<ID “a”>, <ID “b”>,…**
The group of all integer constant: **<INTNUM 1>,…**
The group of all floating point numbers **<FLOAT 1.0>…**

Simple recognizers

- Recognizing keywords
  - Only need to return token type

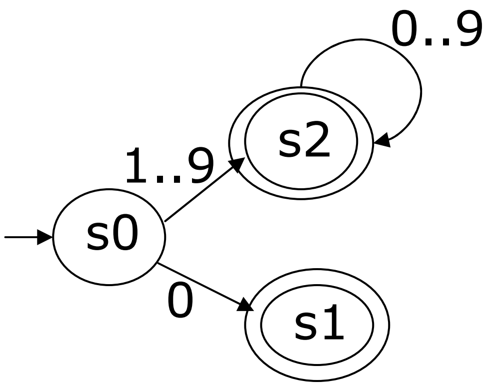
```
c ← NextChar()
if (c == 'f') {
      c ← NextChar()
      if (c == 'e') {
            c ← NextChar()
             if (c=='e')  return <FEE>
      }
}
report syntax error
```



Recognizing integers

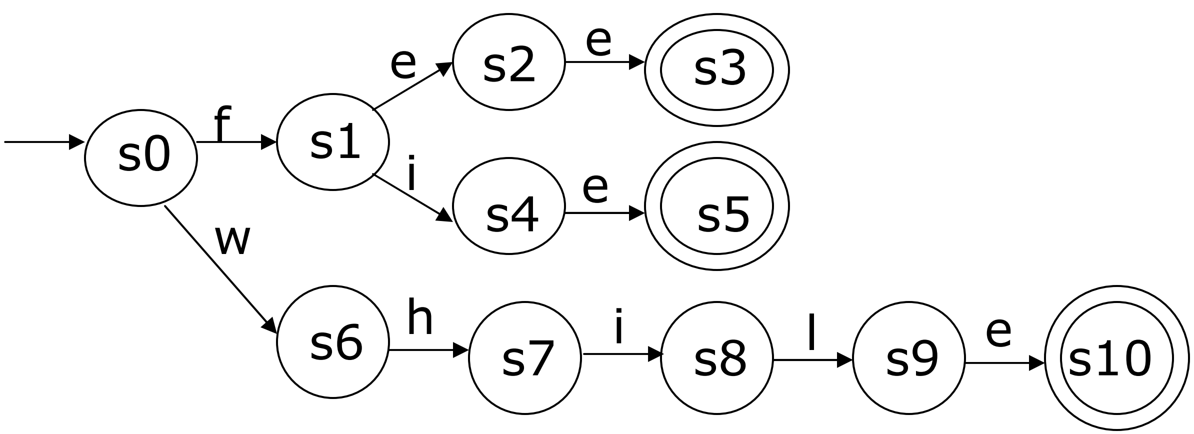
- Token class recognizer
  - Return <type,value> for each token

```
c ← NextChar();
if (c = '0') then return <INT,0>
else if (c >= '1' && c <= '9')  {
          val = c – '0';
          c ← NextChar()
          while (c >= '0' and c <= '9') {
                    val = val * 10 + (c – '0');
                    c ← NextChar()
          }
          return <INT,val>
 }
 else  report syntax error
```



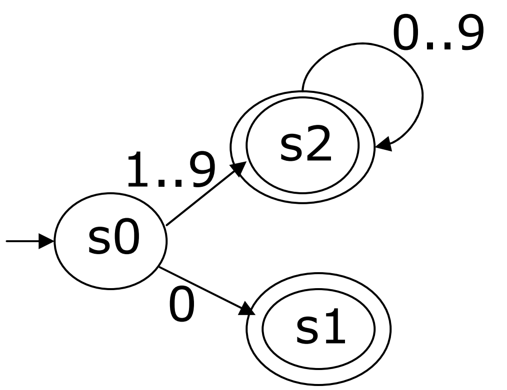
Multi-token recognizers

```
 c ← NextChar()
 if (c == 'f') { c ← NextChar()
                   if (c == 'e') { c ← NextChar()
                                     if (c == 'e')  return <FEE>    else report error }
                   else if (c == 'i') { c ← NextChar()
                                     if (c == 'e')  return <FIE> else report error }
                 }
else if (c == 'w') { c ← NextChar()
                     if (c ==`h') { c ← NextChar(); …}
                     else report error; }
else report error
```



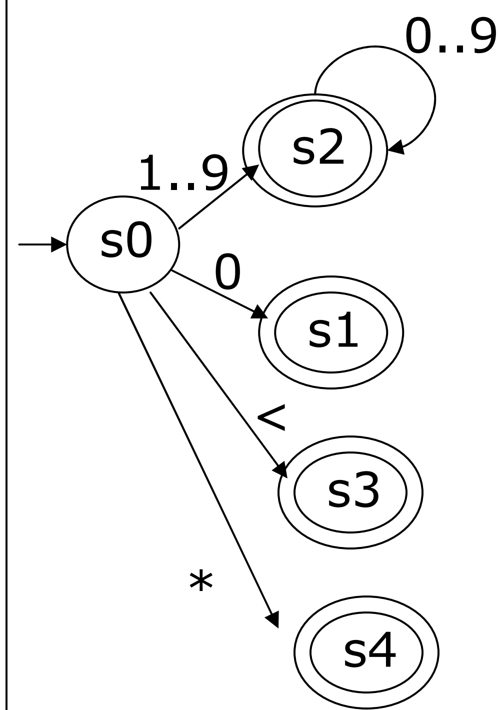
Skipping white space

```
c ← NextChar();
while (c==‘ ’ || c==‘\n’ || c==‘\r’ || c==‘\t’)
      c ← NextChar();
if (c = ‘0’) then return <INT,0>
else if (c >= ‘1’ && c <= ‘9’)  {
          val = c – ‘0’;
          c ← NextChar()
          while (c >= ‘0’ and c <= ‘9’) {
                 val = val * 10 + (c – ‘0’);
                 c ← NextChar()
          }
          return <INT,val>
  }
  else  report syntax error
```



Recognizing operators

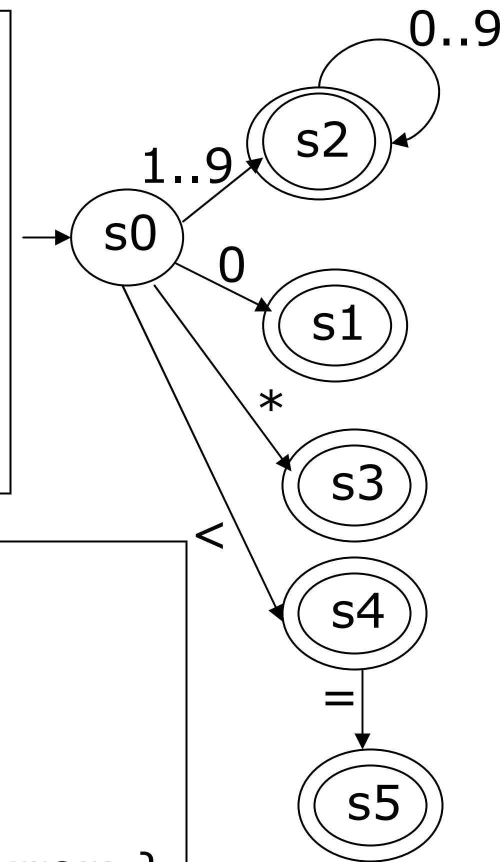
```
c ← NextChar();
while (c==‘ ’ || c==‘\n’ || c==‘\r’ || c==‘\t’)
      c ← NextChar();
if (c = ‘0’) then return <INT,0>
else if (c >= ‘1’ && c <= ‘9’)  {
            val = c – ‘0’;
            c ← NextChar()
            while (c >= ‘0’ and c <= ‘9’) {
                     val = val * 10 + (c – ‘0’);
                     c ← NextChar()
            }
            return <INT,val>
  }
  else if (c == ‘<’) return <LT>
  else if (c == ‘*’) return <MULT>
  else …
  else  report syntax error
```



Reading ahead

- What if both "<=" and "<" are valid tokens?

```
c ← NextChar();
 ……
else if (c == '<') {
     c ← NextChar();
     if (c == '=') return <LE>
     else {PutBack(c); return <LT>; }
}
else …   else  report syntax error
```

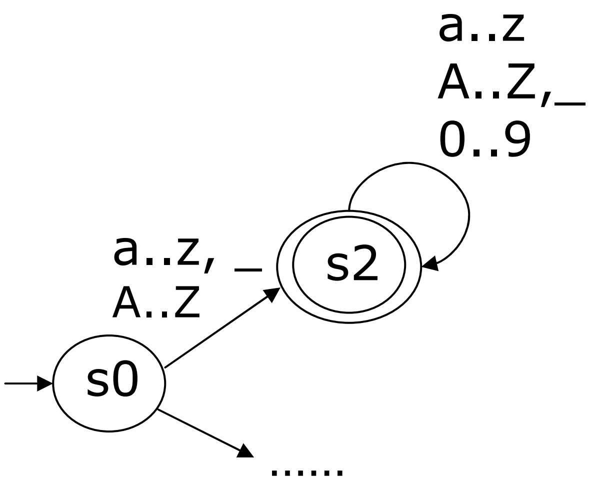



```
static char putback=0;
NextChar() {
   if (putback==0)  return GetNextChar();
   else { c = putback; putback=0; return c; }
}
Putback(char c) { if (putback==0) putback=c; else error; }
```

Recognizing identifiers

- Identifiers: names of variables <ID,val>
  - May recognize keywords as identifiers, then use a hash-table to find token type of keywords

```
 c ← NextChar();
 if (c >= 'a' && c <= 'z' || c>='A' && c<='Z'
|| c == '_')  {
         val = STR(c);
         c ← NextChar()
         while (c >= 'a' && c <= 'z' ||
                 c >= 'A' && c <='Z' ||
                 c >= '0' && c <= '9' ||
                 c=='_') {
                val = AppendString(val,c);
                c ← NextChar()
         }
         return <ID,val>
  }
  else  ......
```



Describing token types

- Each token class includes a set of strings

CLASS = {"class"}; LE = {"<="}; ADD = {"+"};
ID = {strings that start with a letter}
INTNUM = {strings composed of only digits}
FLOAT = { … }

- Use formal language theory to describe sets of strings

An alphabet $\sum$ is a finit set of all characters/symbols
e.g. {a,b,…z,0,1,…9}, {+, -, * ,/, <, >, (,)}
A string over $\sum$ is a sequence of characters drawn from $\sum$
e.g. "abc" "begin" "end" "class" "if a then b"
Empty string: ε
A formal language is a set of strings over $\sum$
{"class"} {"<+"} {abc, def, …}, {…-3, -2,-1,0, 1,…}
The C programming language
English

Operations on strings and languages

- Operations on strings
  - Concatenation: "abc" + "def" = "abcdef"
    - Can also be written as: s1s2 or s1 · s2
  - Exponentiation: $s^i = \underbrace{sssssssss}_{i}$
- Operations on languages
  - Union: $L1 \cup L2 = \{ x \mid x \in L1 \text{ or } x \in L2\}$
  - Concatenation: $L1L2 = \{ xy \mid x \in L1 \text{ and } x \in L2\}$
  - Exponentiation: $L^i = \{ x^i \mid x \in L\}$
  - Kleene closure: $L^* = \{ x^i \mid x \in L \text{ and } i >= 0\}$

Regular expression

- Compact description of a subset of formal languages
  - L(α): the formal language described by α

- Regular expressions over Σ,

  the empty string ε is a r.e., L(ε) = {ε}

  for each s $\in$ Σ, s is a r.e., L(s) = {s}

  if α and β are regular expressions then

  (α) is a r.e., L((α)) = L(α)

  αβ is a r.e., L(αβ) = L(α)L(β)

  α | β is a r.e., L(α | β) = L(α) ∪ L(β)

  α^i is a r.e., L(α^i) = L(α)^i

  α* is a r.e., L(α*) = L(α)*

Regular expression example

- Σ={a,b}

  a | b ➔ {a, b}

  (a | b) (a | b) ➔ {aa, ab, ba, bb}

  a* ➔ {ε, a, aa, aaa, aaaa, …}

  aa* ➔ { a, aa, aaa, aaaa, …}

  (a | b)* ➔ all strings over {a,b}

  a (a | b)* ➔ all strings over {a,b} that start with a

  a (a | b)* b ➔ all strings start with and end with b

Describing token classes

letter = A | B | C | … | Z | a | b | c | … | z
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ID = letter (letter | digit)*
NAT = digit digit*
FLOAT = digit* . NAT | NAT . digit*
EXP = NAT (e | E) (+ | - | ε) NAT
INT = NAT | - NAT

What languages can be defined by regular expressions?

alternatives (|) and loops (*)

each definition can refer to only previous definitions

no recursion

Shorthand for regular expressions

- Character classes
  - [abcd] = a | b | c | d
  - [a-z] = a | b | … | z
  - [a-f0-3] = a | b | … | f | 0 | 1 | 2 | 3
  - [^a-f] = Σ - [a-f]
- Regular expression operations
  - Concatenation: $\alpha \circ \beta = \alpha\, \beta = \alpha \cdot \beta$
  - One or more instances: $\alpha^{+} = \alpha\, \alpha^{*}$
  - i instances: $\alpha^{i} = \alpha\, \alpha\, \alpha\, \alpha\, \alpha$
  - Zero or one instance: $\alpha? = \alpha \mid \varepsilon$
  - Precedence of operations

    $* >> \circ >> |$ **when in doubt, use parenthesis**

What languages can be defined by regular expressions?

letter = A | B | C | … | Z | a | b | c | … | z
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ID = letter (letter | digit)*
NAT = digit digit*
FLOAT = digit* . NAT | NAT . digit*
EXP = NAT (e | E) (+ | - | ε) NAT
INT = NAT | - NAT

What languages can be defined by regular expressions?

alternatives (|) and loops (*)

each definition can refer to only previous definitions

no recursion

Writing regular expressions

- Given an alphabet $\Sigma=\{0,1\}$, describe
  - the set of all strings of alternating pairs of 0s and pairs of 1s
  - The set of all strings that contain an even number of 0s or an even number of 1s
- Write a regular expression to describe
  - Any sequence of tabs and blanks (white space)
  - Comments in C programming language

Recognizing token classes from regular expressions

- Describe each token class in regular expressions
- For each token class (regular expression), build a recognizer
  - Alternative operator (|) ➔ conditionals
  - Closure operator (*) ➔ loops
- To get the next token, try each token recognizer in turn, until a match is found

```
if (IFmatch()) return IF;
else if (THENmatch())  return THEN;
else if (IDmatch())   return ID;
……
```

Building lexical analyzers

- Manual approach
  - Write it yourself; control your own file IO and input buffering
  - Recognize different types of tokens, group characters into identifiers, keywords, integers, floating points, etc.
- Automatic approach
  - Use a tool to build a state-driven LA (lexical analyzer)
    - Must manually define different token classes
- What is the tradeoff?
  - Manually written code could run faster
  - Automatic code is easier to build and modify

Finite Automata --- finite state machines

- Deterministic finite automata (DFA)
  - A set of states S
    - A start (initial) state s0
    - A set F of final (accepting) states
  - Alphabet Σ : a set of input symbols
  - Transition function δ : S x Σ ➔ S
    - Example: δ (1, a) = 2
- Non-deterministic finite automata (NFA)
  - Transition function δ: S x ($\Sigma \cup \{\varepsilon\}$)➔ 2^S
    - Where ε represents the empty string
    - Example: δ (1, a) = {2,3}, δ (2, ε) = 4,
- Language accepted by FA
  - All strings that correspond to a path from the start state s0 to a final state f $\in$ F

Implementing DFA

```
Char ← NextChar()
state ← s0
while (char ≠ eof and state ≠ ERROR)
     state ←δ (state, char)
     char ← NextChar()
if (state ∈ F) then report acceptance
else report failure
```

$S = \{s0, s1, s2\}$

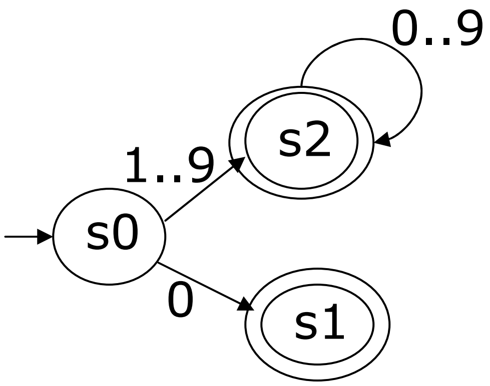
$\Sigma = \{0,1,2.3,4,5,6,7,8,9\}$

$\delta(s0,0) = s1$

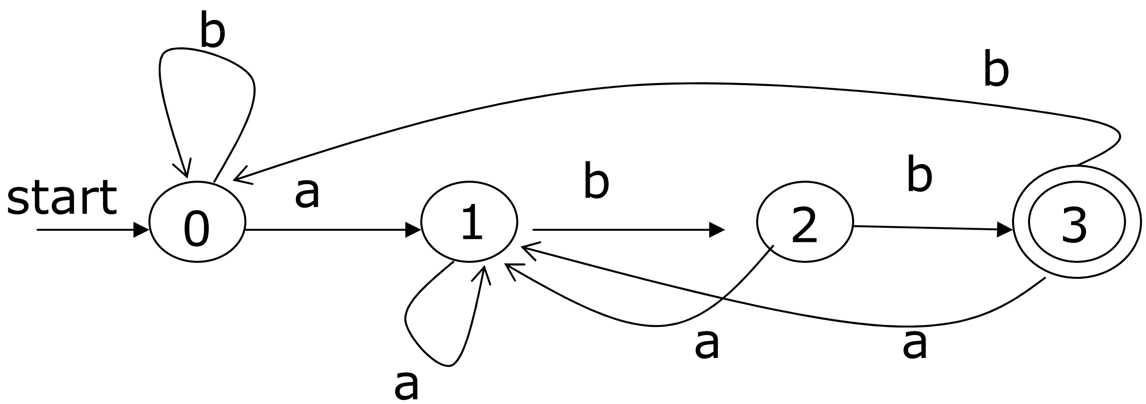
$\delta(s0,1\text{-}9) = s2$

$\delta(s2,0\text{-}9) = s2$

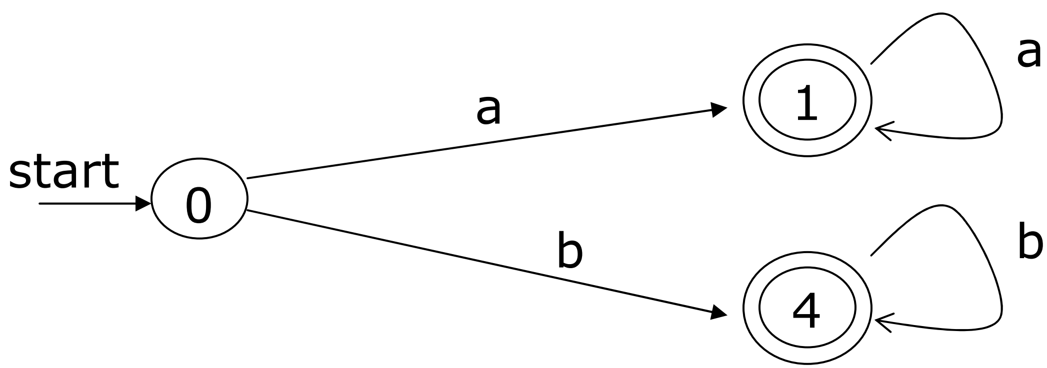
$F = \{s1, s2\}$



DFA examples

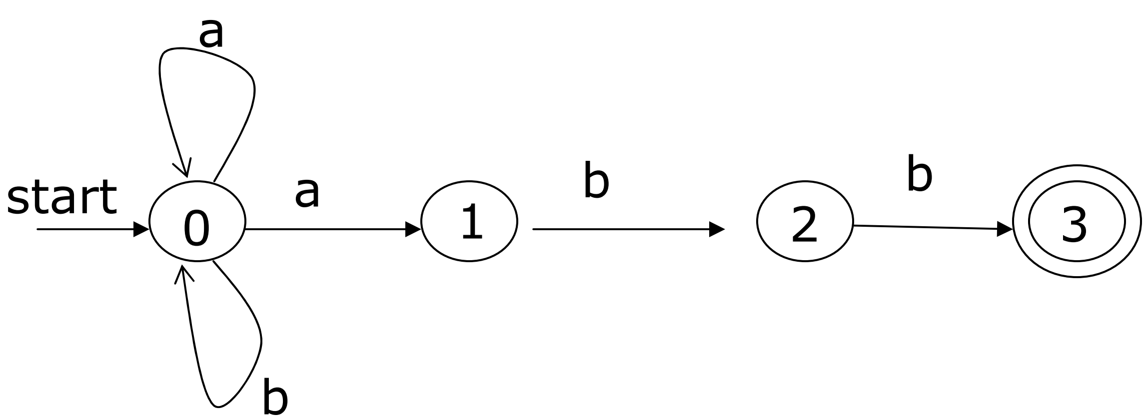


Accepted language: (a|b)*abb

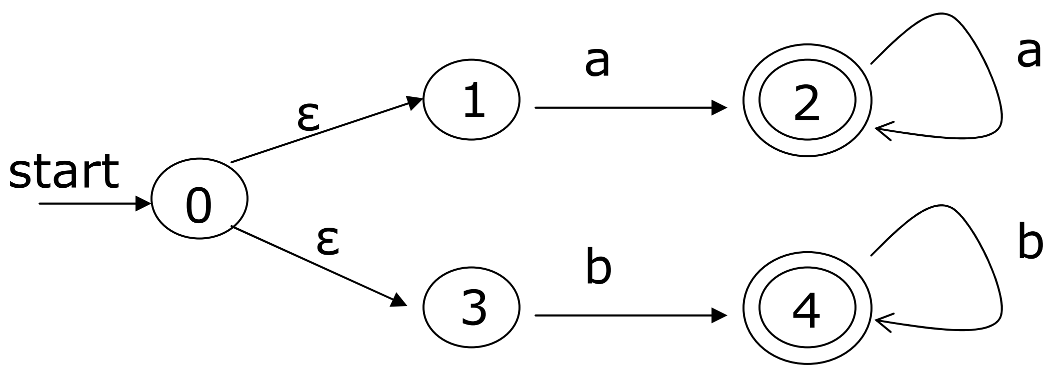


Accepted language: a+ | b+

NFA examples



Accepted language: (a|b)*abb



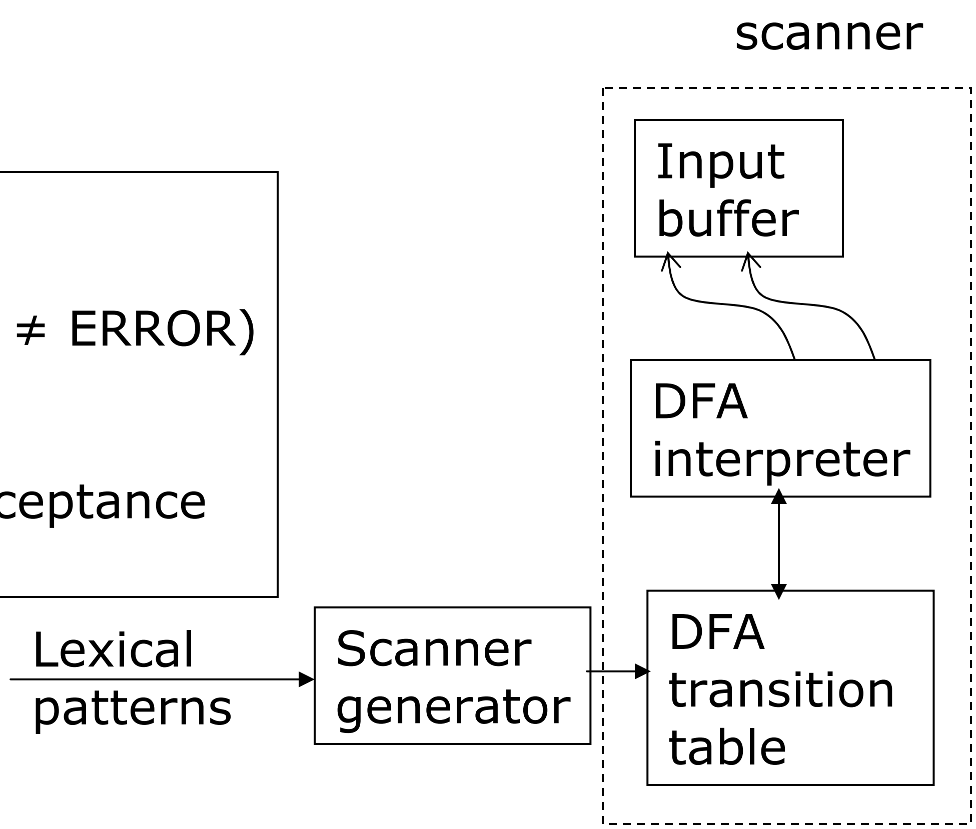
Accepted language: a+ | b+

Automatically building scanners

- Regular Expressions/lexical patterns ➔ NFA
- NFA ➔ DFA
- DFA ➔ Lexical Analyzer

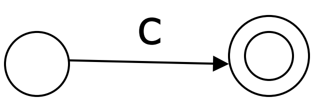
DFA interpreter:

```
Char ← NextChar()
state ← s0
While (char ≠ eof and state ≠ ERROR)
    state ←δ (state, char)
    char ← NextChar()
if (state ∈ F) then report acceptance
Else report failure
```

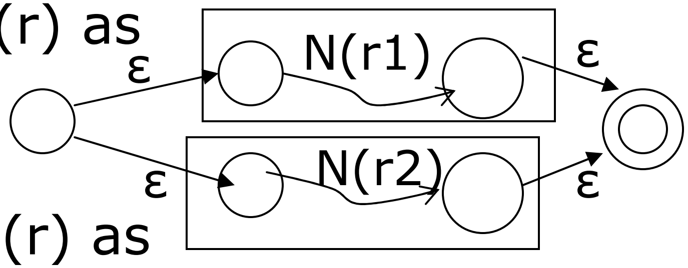


Converting RE to NFA

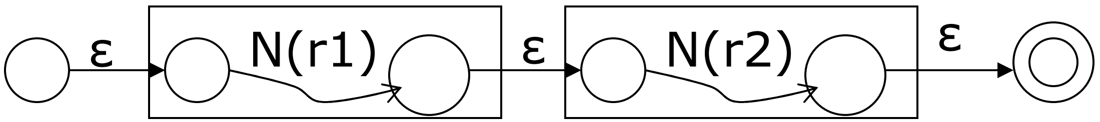
- Thompson's construction
  - Takes a regexp r and returns NFA N(r) that accepts L(r)
- Recursive rules
  - For each symbol c ∈ ∑ ∪{ε}, define NFA N(c) as



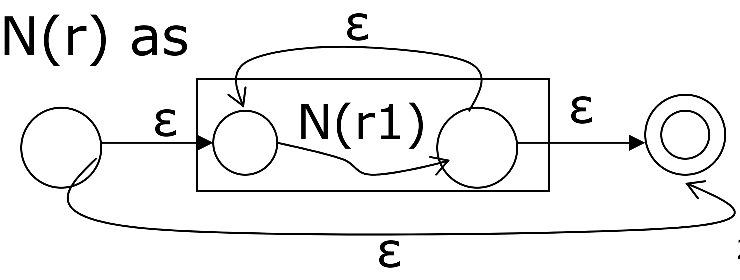
  - Alternation: if (r = *r1* | *r2*) build N(r) as



  - Concatenation: if (r = r1r2) build N(r) as

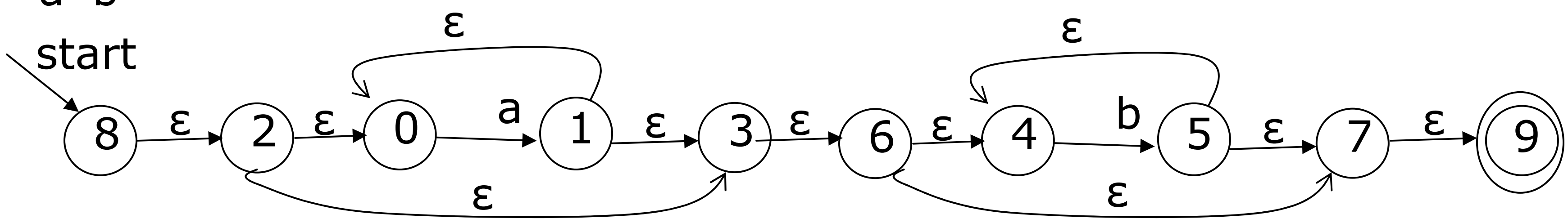


  - Repetition: if (r = *r1**) build N(r) as

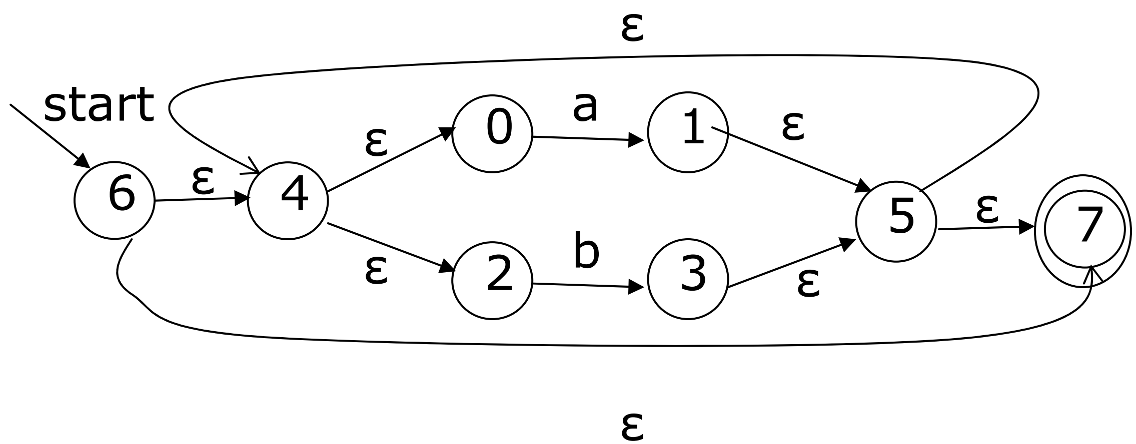


RE to NFA examples

a*b*

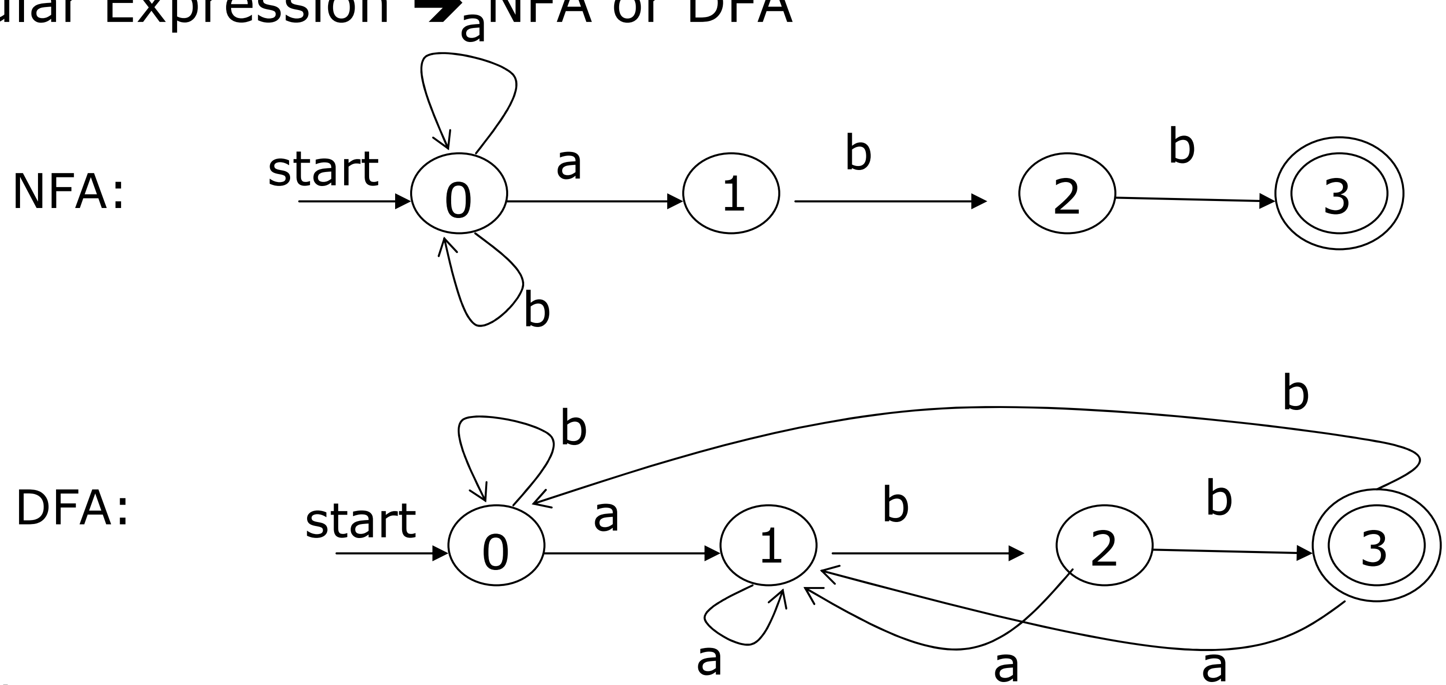


(a|b)*



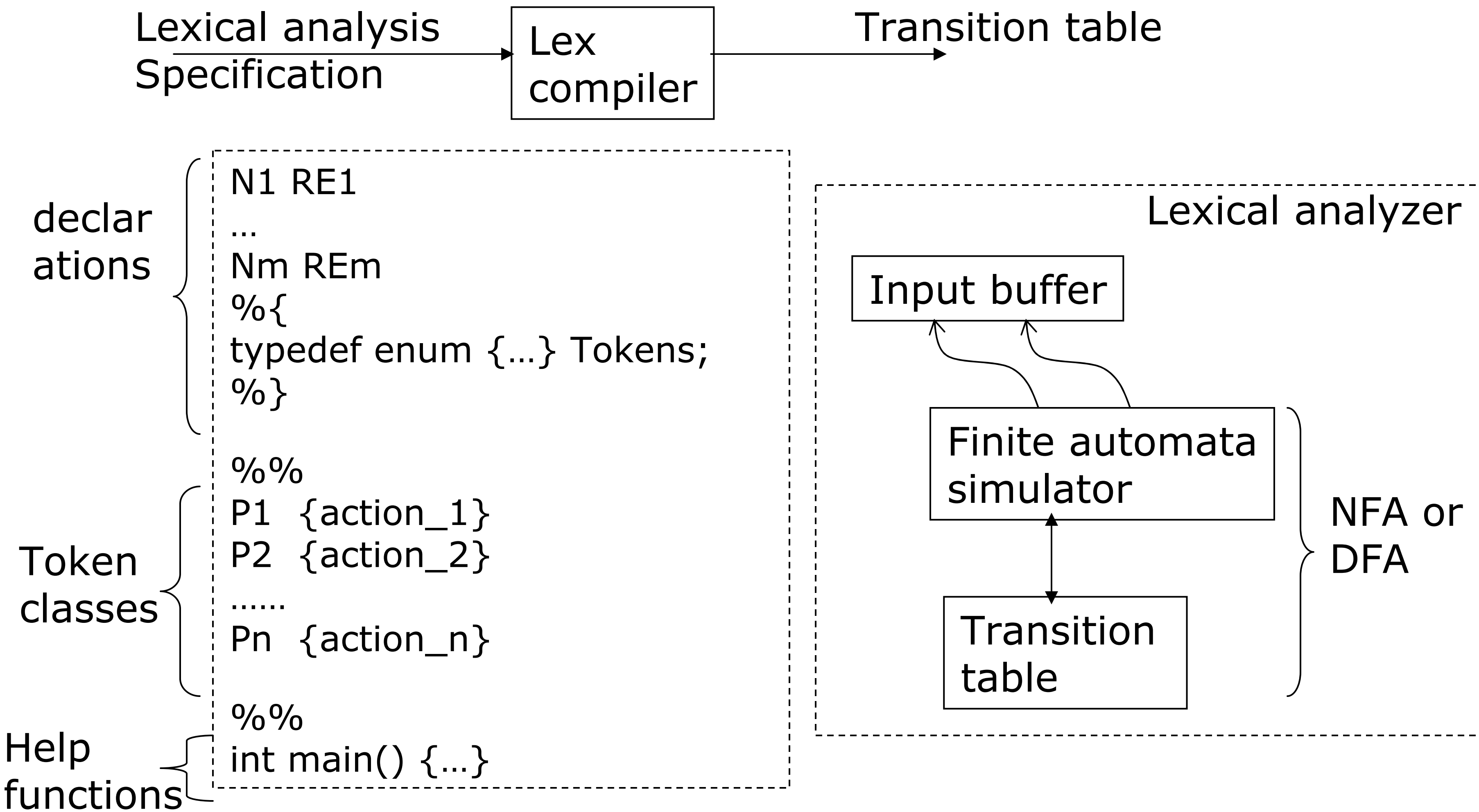
Automatically building lexical analyzer

- Token ➔ Pattern
- Pattern ➔ Regular Expression
- Regular Expression ➔ NFA or DFA



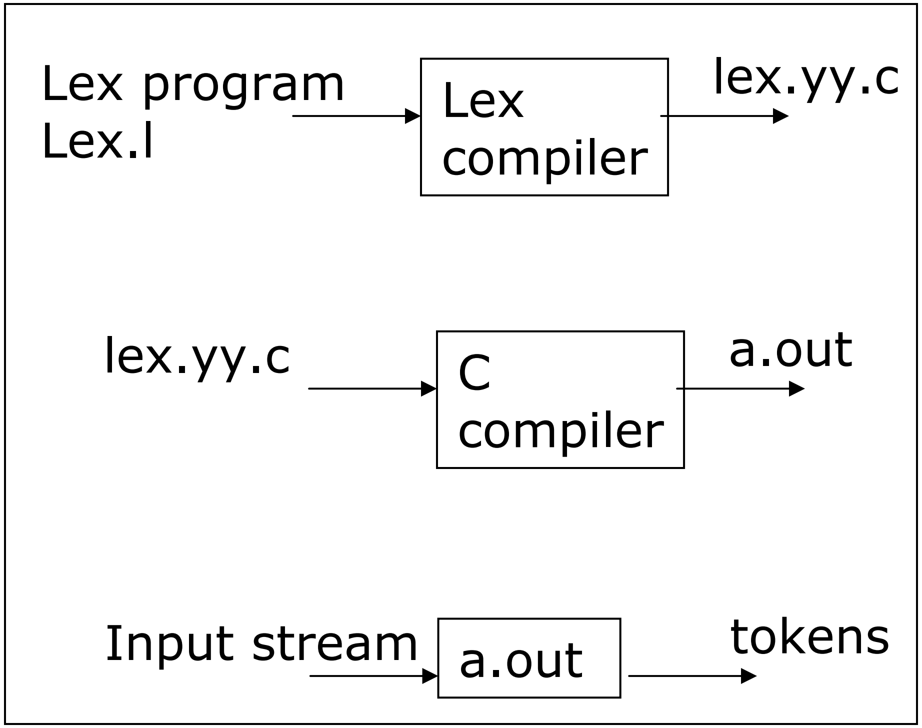
- NFA/DFA ➔ Lexical Analyzer

Lexical analysis generators



Using Lex to build scanners

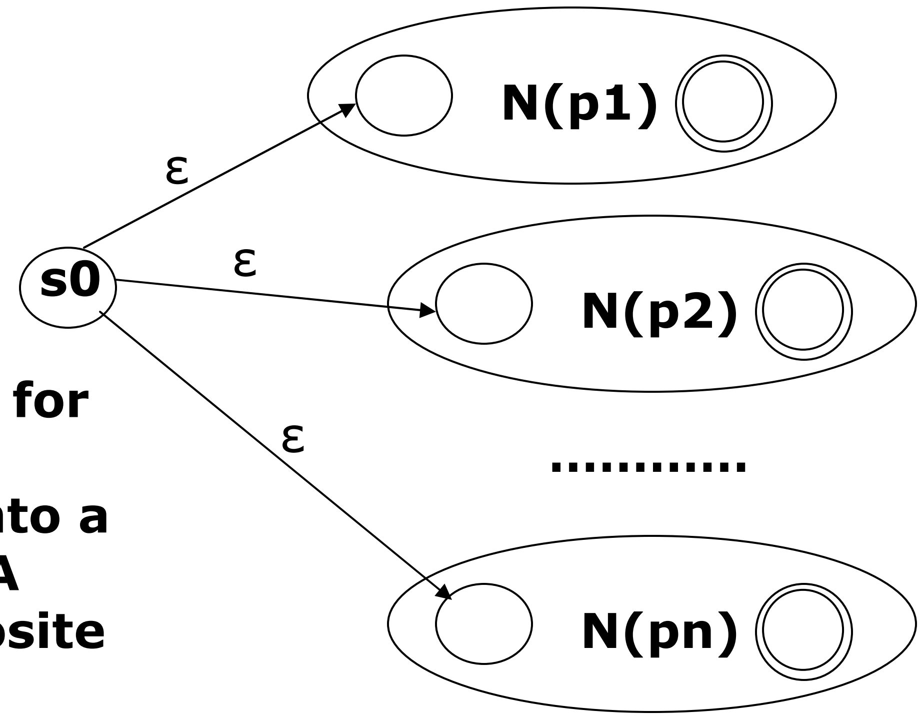
```
cconst   '([^\']+|\\\')'
sconst  \"[^\"]*\"

%pointer

%{
  /* put C declarations here*/
%}

%%
foo { return FOO; }
bar { return BAR; }
{cconst} { yylval=*yytext;
                 return CCONST; }
{sconst} { yylval=mk_string(yytext,yyleng);
                 return SCONST; }
[ \t\n\r]+    {}
.        { return ERROR; }
```



NFA-based lexical analysis

Specifications

```
P1  {action_1}
P2  {action_2}
……
Pn  {action_n}
```



(1) Create a NFA N(pi) for each pattern
(2) Combine all NFAs into a single composite NFA
(3) Simulate the composite NFA: must find the longest string matched by a pattern ➔ continue making transitions until reaching termination

Simulate NFA

- Movement through NFA on each input character
  - Similar to DFA simulation, but must deal with multiple transitions from a set of states
- Idea: each DFA state correspond to a set of NFA states
  - s is a single state

    ε-closure(t) = {s | s is reachable from t through ε-transitions}
  - T is a set of states

    ε-closure(T) = $\{s \mid \exists\, t \in T$ s.t. $s \in$ ε-closure(t) $\}$

```
S = ε-closure(s0); a = nextchar();
while (a != eof)
      S = ε-closure( move(S,a) );
      a = nextchar();
If (S ∩ F != ∅) return "yes"; else return "no"
```

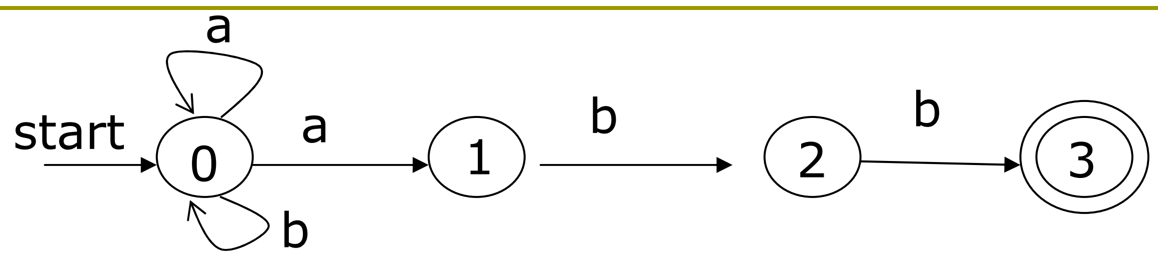
DFA-based lexical analyzers

- Convert composite NFA to DFA before simulation
  - Match the longest string before terminiation
  - Match the pattern specification with highest priority

add ε-*closure*(s0) to *Dstates* unmarked
while there is unmarked T in *Dstates* **do**
    mark T;
    for each symbol *c in* ∑ **do begin**
        U := ε-*closure*(**move**(T, *c*));
        Dtrans[T, *c*] := U;
        if U is not in *Dstates* **then**
            add U to *Dstates* unmarked

Convert NFA to DFA example

NFA:



Dstates = {ε-*closure*(s0)} = { {s0} };
Dtrans[{s0},a] = ε-*closure*(**move**({s0}, *a*)) = {s0,s1};
Dtrans[{s0},b] = ε-*closure*(**move**({s0}, *b*)) = {s0};

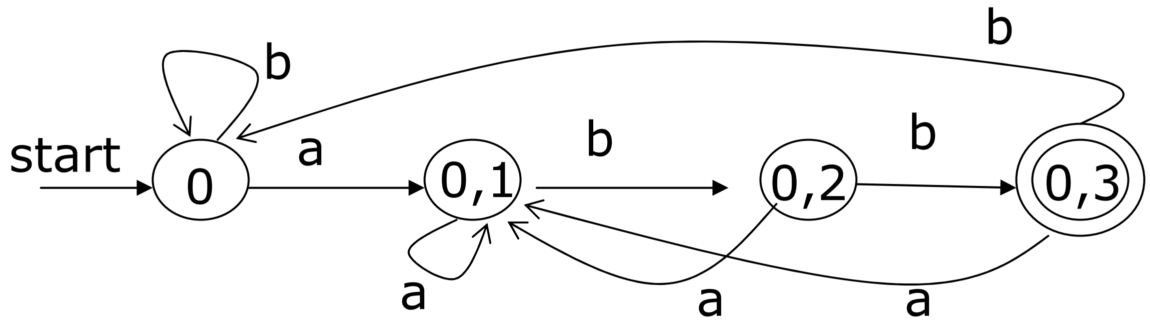
Dstates = {{s0} {s0,s1} };
Dtrans[{s0,s1},a] = ε-*closure*(**move**({s0,s1}, *a*)) = {s0,s1};
Dtrans[{s0,s1},b] = ε-*closure*(**move**({s0,s1}, *b*)) = {s0,s2};

Dstates = {{s0} {s0,s1} {s0,s2} };
Dtrans[{s0,s2},a] = ε-*closure*(**move**({s0,s2}, *a*)) = {s0,s1};
Dtrans[{s0,s2},b] = ε-*closure*(**move**({s0,s2}, *b*)) = {s0,s3};

Dstates = {{s0}, {s0,s1}, {s0,s2}, {s0,s3}};
Dtrans[{s0,s3},a] = ε-*closure*(**move**({s0,s3}, *a*)) = {s0,s1};
Dtrans[{s0,s3},b] = ε-*closure*(**move**({s0,s3}, *b*)) = {s0};

Convert NFA to DFA example

DFA:



Dstates = {{s0}, {s0,s1}, {s0,s2}, {s0,s3}};
Dtrans[{s0},a] = {s0,s1};
Dtrans[{s0},b] = {s0};
Dtrans[{s0,s1},a] = {s0,s1};
Dtrans[{s0,s1},b] = {s0,s2};
Dtrans[{s0,s2},a] = {s0,s1};
Dtrans[{s0,s2},b] = {s0,s3};
Dtrans[{s0,s3},a] = {s0,s1};
Dtrans[{s0,s3},b] = {s0};