CS 4300: Compiler Theory

Chapter 3
Lexical Analysis

Dr. Xuejun Liang

Outlines (Sections)

1. The Role of the Lexical Analyzer
2. Input Buffering (Omit)
3. Specification of Tokens
4. Recognition of Tokens
5. The Lexical -Analyzer Generator Lex
6. Finite Automata
7. From Regular Expressions to Automata
8. Design of a Lexical-Analyzer Generator
9. Optimization of DFA-Based Pattern Matchers*

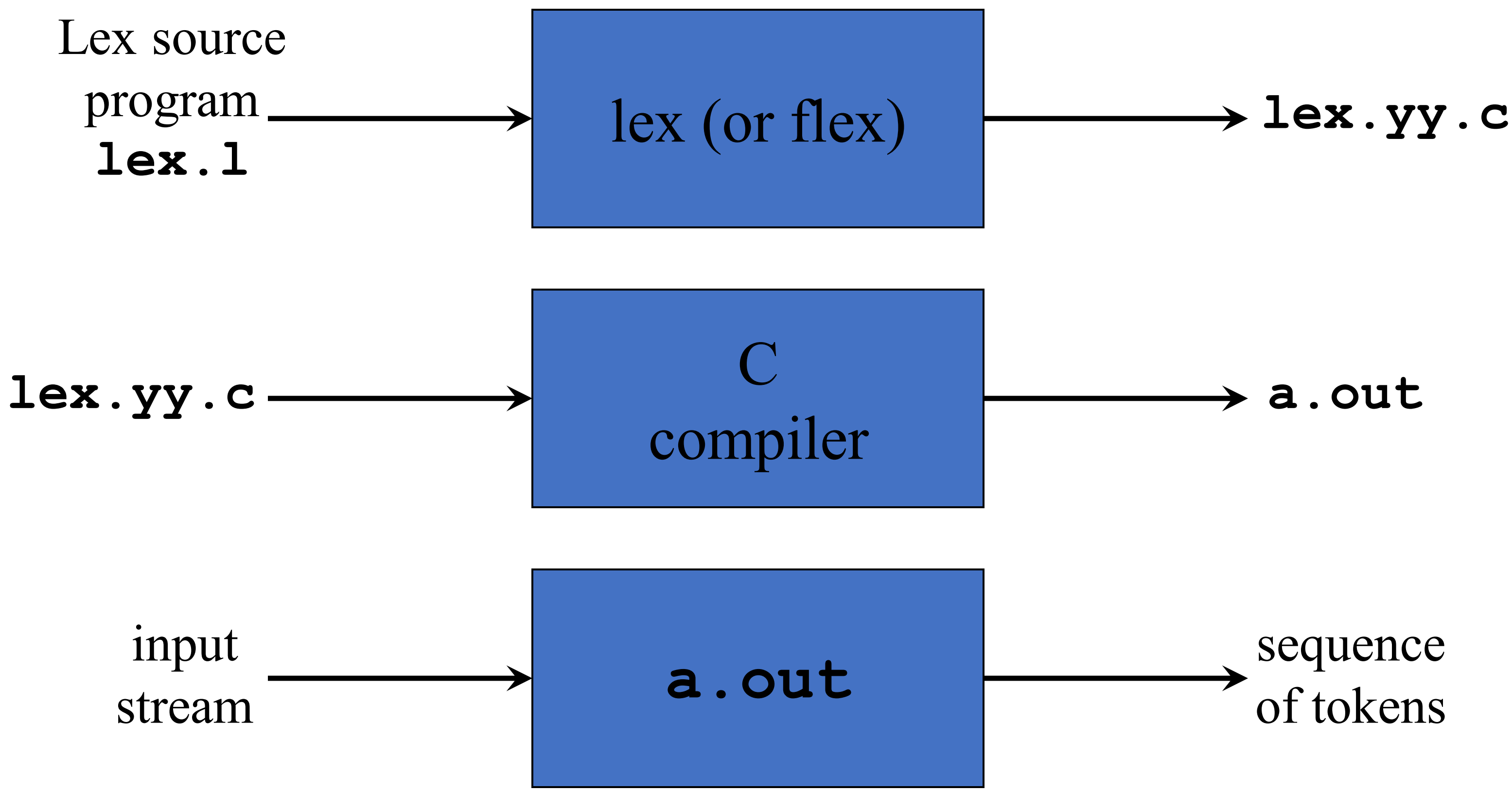
Quick Review of Last Lecture

- The Role of the Lexical Analyzer
  - What a lexical analyzer (scanner) does?
  - Tokens, Patterns, and Lexemes
  - Attributes for Tokens
- Specification of Tokens
  - String operations and language operations
  - Regular expression, its operations, and examples
  - Regular definitions, extensions, and examples
- Recognition of Tokens
  - Patterns for tokens, lexemes, attribute values,
  - Transition diagrams for each regular definition

5. Lexical-Analyzer Generator: Lex and Flex

- *Lex* and its newer cousin *flex* are *scanner generators*
- Scanner generators systematically translate regular definitions into C source code for efficient scanning
- Generated code is easy to integrate in C applications

Creating a Lexical Analyzer with Lex and Flex



Structure of Lex Programs

- A Lex program consists of three parts:

  declarations
  `%%`
  translation rules
  `%%`
  user-defined auxiliary procedures

- declarations
  - *C declarations in* `%{   %}`
  - *regular definitions*
- The translation rules are of the form:

  $pattern_1$ { $action_1$ }
  $pattern_2$ { $action_2$ }
  …
  $pattern_n$ { $action_n$ }

Regular Expressions in Lex

`x`	match the character **`x`**
`\.`	match the character **`.`**
"*string***"**	match contents of string of characters
`.`	match any character except newline
`^`	match beginning of a line
`$`	match the end of a line
`[xyz]`	match one character **`x`**, **`y`**, or **`z`** (use **`\`** to escape **`-`**)
`[^xyz]`	match any character except **`x`**, **`y`**, and **`z`**
`[a-z]`	match one of **`a`** to **`z`**
*r***`*`**	closure (match zero or more occurrences)
*r***`+`**	positive closure (match one or more occurrences)
*r***`?`**	optional (match zero or one occurrence)
$r_1 r_2$	match r_1 then r_2 (concatenation)
r_1**`|`**r_2	match r_1 or r_2 (union)
`(` *r* **`)`**	grouping
r_1**`\`**r_2	match r_1 when followed by r_2
`{`*d***`}`**	match the regular expression defined by *d*

Example Lex Specification 1

Contains the matching lexeme

Translation rules

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("%s\n", yytext); }
.|\n    { }
%%
main()
{ yylex();
}
```

Invokes the lexical analyzer

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```

Example Lex Specification 2

Regular definition

Translation rules

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}
delim      [ \t]+
%%
\n          { ch++; wd++; nl++; }
^{delim}    { ch+=yyleng; }
{delim}     { ch+=yyleng; wd++; }
.           { ch++; }
%%
main()
{ yylex();
  printf("%8d%8d%8d\n", nl, wd, ch);
}
```

Example Lex Specification 3

Translation rules

Regular definitions

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main()
{ yylex();
}
```

Lex Specification: Example 3.8

```
%{ /* definitions of manifest constants */
#define LT (256)
…
%}
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
%%
{ws}        { }
if          {return IF;}
then        {return THEN;}
else        {return ELSE;}
{id}        {yylval = install_id(); return ID;}
{number}    {yylval = install_num(); return NUMBER;}
“<“         {yylval = LT; return RELOP;}
“<=“        {yylval = LE; return RELOP;}
“=“         {yylval = EQ; return RELOP;}
“<>“        {yylval = NE; return RELOP;}
“>“         {yylval = GT; return RELOP;}
“>=“        {yylval = GE; return RELOP;}
%%
int install_id()
…
```

Return token to parser

Token attribute

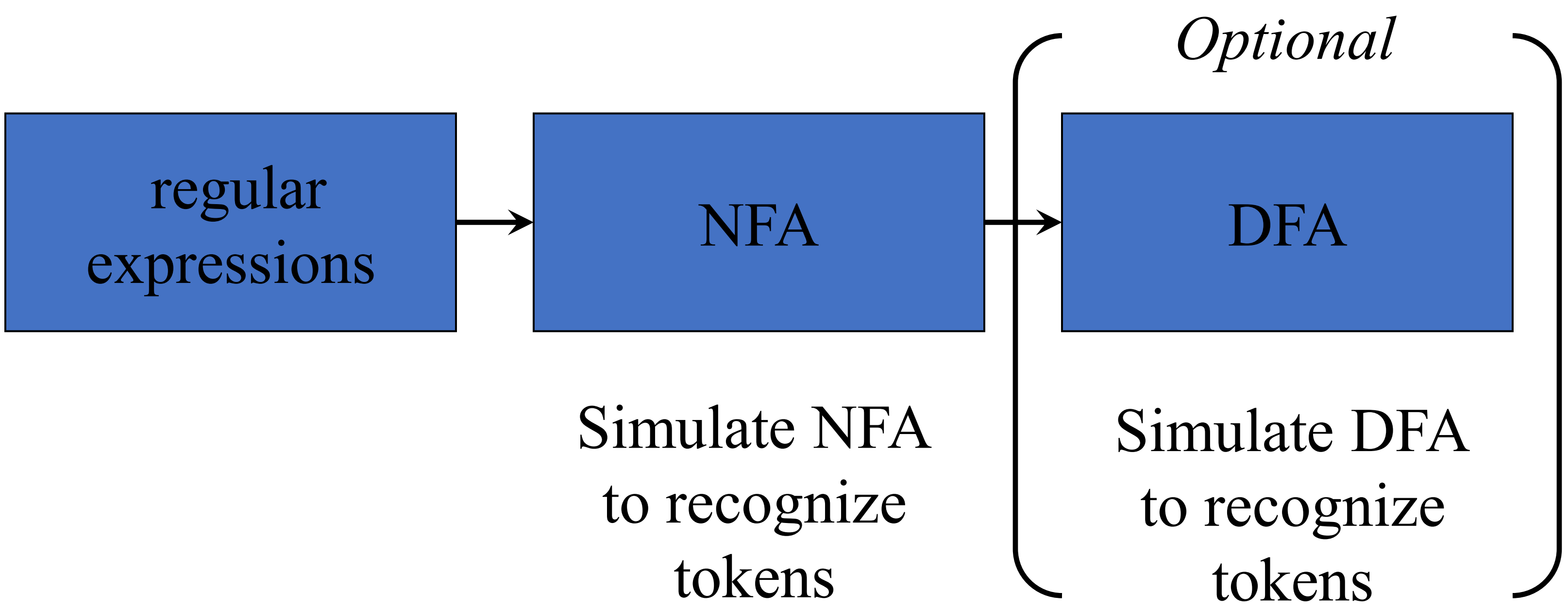
Install **`yytext`** as identifier in symbol table

Conflict Resolution in Lex

- Two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:
  1. Always prefer a longer prefix to a shorter prefix.
  2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

6. Finite Automata

- Design of a Lexical Analyzer Generator
  - Translate regular expressions to NFA
  - Translate NFA to an efficient DFA



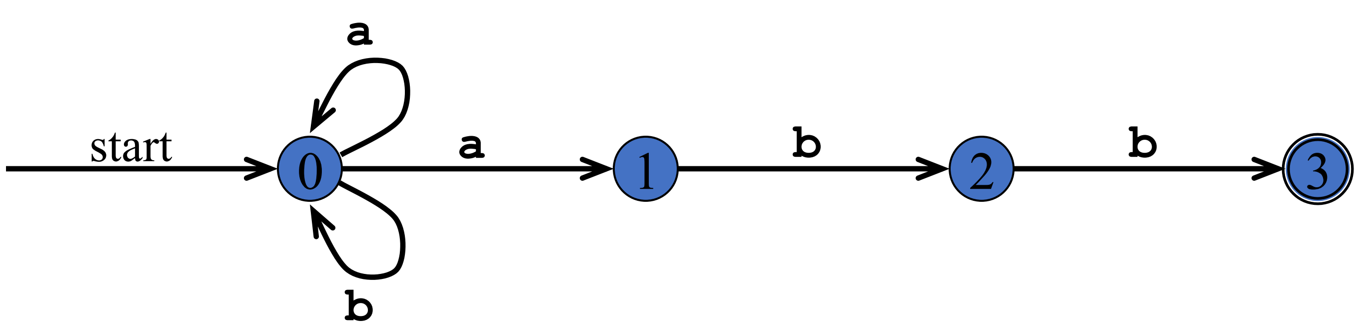
Nondeterministic Finite Automata

- An NFA is a 5-tuple (S, Σ, δ, s_0, F) where

  S is a finite set of *states*
  Σ is a finite set of symbols, the *alphabet*
  δ is a *transition function* from $S \times (\Sigma \cup \{\varepsilon\})$ to a set of states
  $s_0 \in S$ is the *start state*
  $F \subseteq S$ is the set of *accepting* (or *final*) *states*

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a *transition graph*
- Example
  - an NFA recognizing the language of regular expression **(a|b) * abb**



$$S = \{0,1,2,3\},\ \Sigma = \{\mathbf{a},\mathbf{b}\},\ s_0 = 0,\ F = \{3\}$$

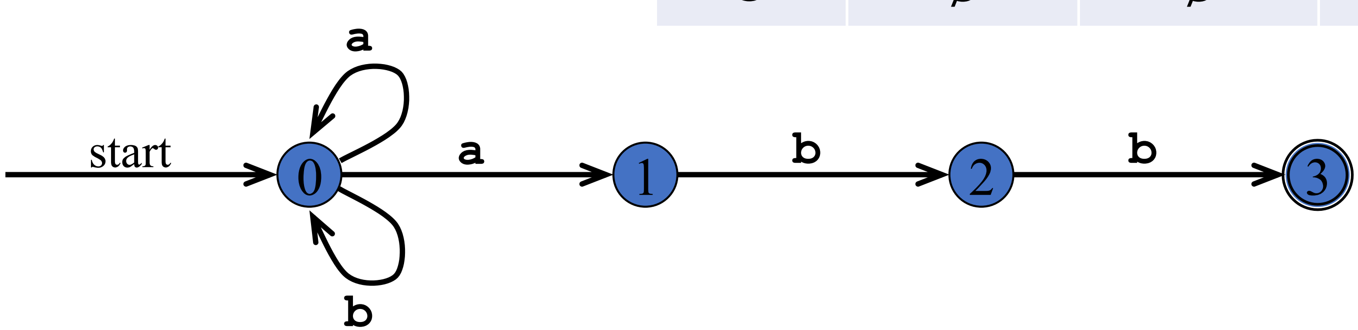
Transition Table

- The mapping δ of an NFA can be represented in a *transition table*

$\delta(0, \mathbf{a}) = \{0,1\}$
$\delta(0, \mathbf{b}) = \{0\}$
$\delta(1, \mathbf{b}) = \{2\}$
$\delta(2, \mathbf{b}) = \{3\}$

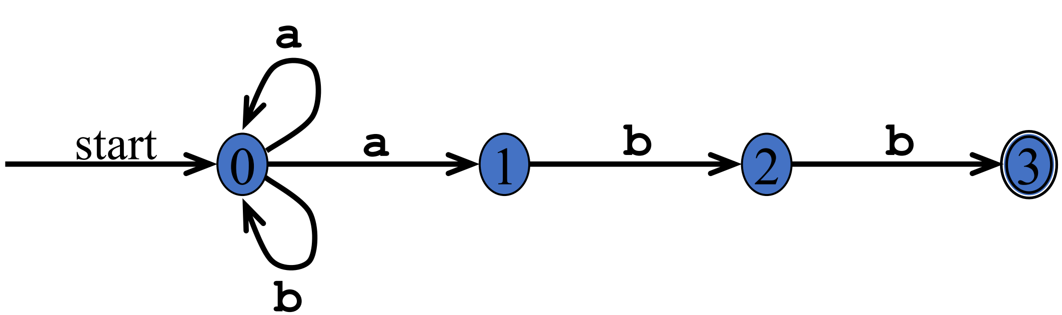
$\longrightarrow$

State	Input a	Input b	Input ε
0	{0, 1}	{0}	∅
1	∅	{2}	∅
2	∅	{3}	∅
3	∅	∅	∅



The Language Defined by an NFA

- An NFA *accepts* an input string *x* if and only if there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
- A state transition from one state to another on the path is called a *move*
- The *language defined by* an NFA is the set of input strings it accepts, such as (**a** | **b**)***abb** for the example NFA



Deterministic Finite Automata

- A deterministic finite automaton (DFA) is a special case of NFA
  - No state has an ε-transition
  - For each state *s* and input symbol *a* there is exactly one edge out of *s* labeled *a*
- Each entry in the transition table is a single state
  - At most one path exists to accept a string
  - Simulation algorithm is simple

Simulating a DFA

$s = s_0;$
$c = nextChar();$
while (c != **eof**) {
    $s = move(s, c);$
    $c = nextChar();$
}
if (s is in F) **return "yes";**
else return "no";

Example: A DFA that accepts (**a** | **b**)***abb**