

CSE 131A
“Compiler Construction I”

Lecture 4
Using JFlex
Lexical scanning theory

Announcements

- A1 has been posted; it's due Friday 1/26
- Office hours schedule **next** week
 - Monday 1/22 are CANCELLED
 - Weds 1/24
Moved to Thursday 1/25, 5 to 6pm

Recognizing tokens

- So far we've talked about how to specify tokens
 - Tokens are regular languages...
 - ... specified with regular expressions
- We next consider how to recognize them in a text stream and respond with actions
- We'll use a lexer generated automatically by JFlex

Some practical details

- We recall that the parser calls the scanner, which in turn delivers tokens
- The parser is really in charge
- The tools we use—JFlex and CUP—expect this relationship between parser and scanner
- The scanner must conform to an interface specification assumed by the parser
- We need some information about the parsing process in order to build a scanner

Jflex in context

- Jflex is a *source-to-source translator*
- A program that translates source code from one language into another language
- In our case, Jflex translates Jflex source code into java

A simple language

- Modified Example 3.6 on pp. 98 of the text
(../public Examples/Fig_3_18)
- Terminal symbols are in **bold face**

stmt → **if** *expr* **then** *stmt* **else** *stmt* | *term*

expr → *term* **relop** *term* | *term*

term → **id** | **num**

relop → < | <= | = | <> | > | >=

id → **letter** (**letter** | **digit**)*

num → **digit**⁺ ("." **digit**⁺)?

Example expressions

- Some expressions in that language:

3

foo

if x then 987.6 else z123

if a>4 then if b<=7 then 2323 else c

How do we build the flex file?

- The lexical analyzer must recognize the Terminal symbols in the language
- These are defined by the context-free grammar for the language
- Declared in the CUP file (more on parsing later)
- JFlex supports regular expression macros that define tokens:

```
if = "if"  
then = "then"  
else = "else"  
LT      = "<"  
LE      = "<="   
LETTER  = [A-Za-z]  
ID      = {LETTER}+  
DIGIT   = [:digit:]  
NUMBER  = {DIGIT}+("."{DIGIT}+)?
```

What's in that Jflex file?

- Jflex files have 3 parts

- User code
- Options and declarations
- Lexical Rules

```
%{  
private Symbol genToken(int ndxToken) {  
    return new Symbol(ndxToken, ychar,  
                      ychar+yylength()-1); }  
private Symbol genToken(int ndxToken,  
                        Object val) {  
    return new Symbol(ndxToken, ychar,  
                      ychar+yylength()-1,val);  
}  
%}
```

```
import java_cup.runtime.*;  
%%  
// JFlex Options Section ***  
// Output class  
%class Scanner  
%public  
// Generator options  
%unicode  
%cup  
%char  
%line  
%column
```

Lexical Rules

```
%%  
// JFlex Rules Section **  
{WS}      { /* Ignore whitespace */ }  
"if"  
  { return genToken(sym.IF); }  
"then"  
  { return genToken(sym.THEN); }  
"else"  
  { return genToken(sym.ELSE); }  
{ID}  
  { return genToken(sym.ID,yytext()); }  
{NUMBER}  
  { return genToken(sym.NUMBER,yytext()); }  
{LT}  
  { return genToken(sym.LT,yytext()); }  
...
```

How does Jflex translate the flex file?

- It generates a **class** declared in the options section
- It interpolates (includes) all **user code** verbatim
- It translates the **regexp/action pairs** into java code

```
import java_cup.runtime.*;
%%
// JFlex Options Section ***

// Output class
%class Scanner
%%
// JFlex Rules Section **
{WS}      { /* Ignore whitespace */ }
"if"      { return genToken(sym.IF); }
```

Translating the regexp/action pairs

```
/* The following code was generated by JFlex 1.4.1 on 1/16/06 8:00 PM */
import java_cup.runtime.*;
public java_cup.runtime.Symbol next_token() throws java.io.IOException {
while (true) {
    switch (zzAction < 0 ? zzAction : ZZ_ACTION[zzAction]) {
        case 6:
            { return genToken(sym.ID,yytext()); }
        case 2:
            { /* Ignore whitespace */ }
        case 7:
            { return genToken(sym.NUMBER,yytext()); }
        case 11:
            { return genToken(sym.IF);}
    }
}
```

```
// JFlex Rules Section **
{WS}      { /* Ignore whitespace */ }
"if"
    { return genToken(sym.IF); }
{ID}
    { return genToken(sym.ID,yytext()); }
{NUMBER}
    { return
    genToken(sym.NUMBER,yytext()); }
```

Multiple matches

- We always extract the lexeme which is the longest possible
- Thus, we recognize \leq rather than $<$
- Otherwise, how would we recognize \leq

Errors

- The final rule is a “catch all”
 - Any input not matched by a previous rule will match this last “any character but end-of-line” pattern
 - This rule matching may indicate a problem
 - Action can throw exception, print message, etc.; or it might not be an error at all

```
[^] { System.err.println("Illegal " +  
      "character: "+yytext()); }
```

Matching
lexeme for a
rule is returned
as a Java String

The form of JFlex rules

- The left-hand-side of a rule is a regexp token pattern
- Code for the rule's action is within curly braces { }
- The definition of constants `sym.SEMI`, etc. are `public final ints` defined in the `sym` class
- Action usually includes a return statement
 - this will return from `get_token`, so must provide a `Symbol` to return
- Action is up to you

Symbol class

- The `get_token` method returns a `Symbol` object, representing a token
- `Symbol` class might be defined something like:

```
public class Symbol {  
  
    public Object val; // value of the token  
    public int id;    // ID of the token  
    public int line; // line # of token in input  
    public int col;  // col # of token in input  
  
    public Symbol(Object val, int id) {  
        this.val=val; this.id=id;  
    }  
}
```

Defining token ID's

- Token ID's may be defined as named int constants

```
public class sym {  
    public final int IF = 2;  
    public final int THEN = 3;  
    public final int ELSE = 4;  
    public final int GT = 1;  
    public final int ID = 5;  
    public final int NUMBER = 6;  
    public final int EOF = 0;  
    ...  
}
```

Identifiers and keywords

- Identifier names are entered into the symbol table, which is accessed in later stages of compilation
- The attribute of an identifier is a pointer to the symbol table
- Keywords are reserved. Rather than put them into the symbol table, we can assign them a non-terminal
- This is reasonable if the number of keywords is small

Constants

- We can return the value of the constant as an attribute of the lexeme
- Or, we could put all constants into a table of constants, and return a pointer as with an identifier
- This is especially useful for character strings or other long-winded literals

Useful builtin JFlex methods and variables

- If the **%line** and **%column** options are set in the JFlex Options Section, then the int variables **yyline**, **yycolumn** hold the line and column in the input stream of the start of the matching lexeme

CUP Interface

- CUP exports two classes via the package `java_cup.runtime.*`
 - `Symbol` (i.e. tokens)
 - `Scanner`
- It generates symbolic names for all the tokens and puts them in a file `sym.java`
- The scanner generates instances of `Symbol`
- Our scanner driver invokes `next_token()` to extract tokens
- There will be no parser at this point... later on!
- In assignment #1, we'll give you `sym.java`

Specifying the Lexical Rules

- Consulting the lexical spec for A1, we write a JFlex file that specifies the two components of each lexical rule
 - A token pattern, specified by a regular expression
 - An action to be taken (code to be run) when the pattern is matched

Recognizing tokens

- JFlex generates the method **get_token()**
- When called:
 - finds the token that matches the current initial segment of the stream and consumes that segment, and
 - returns some useful information such as the value of the matching lexeme, the ID of the token, etc., and perhaps perform other actions
- Our scanner driver (or the parser) will call it

Towards a lexical scanner

- The **get_token()** method performs the specified action when the corresponding regular expression is recognized in the input
- By convention
 - If one or more tokens match initial segments of different length in the input, the token with the longest lexeme is recognized
 - If two tokens match the same lexeme in the input, the one appearing first in the specification is recognized

Tokens and actions

- A specification for a lexical analyzer will be a list of regexp/action rules

"if" "IF"	{ return new Symbol(sym.IF); }
777	{return new Symbol(sym.LUCKY);}
[:digit:]+	{ return new Symbol(sym.INT,ytext()); }
"<"	{ return new Symbol(sym.LT); }
"<<"	{ return new Symbol(sym.LLT); }
" "	{ /* do nothing */ }

An example

- With those regexp/action rules and those conventions, how will this text be tokenized?

if 7778 << 777 IF <<< 8777 iF

"if" | "IF"

777

[:digit:]+

"<"

"<<"

" "

An effect of a convention

- C/C++ lexical analysis follows this convention, so the following code contains a syntax error that will be detected by the parser:

```
int v, w=3, x=4, y, z;  
int *p, *q;  
p = &w;   q = &x;  
y = *p+*q; /* add */  
z = *p/*q; /* divide */  
v = *p-*q; /* subtract */
```

- ... do you see it?

A JFlex example

- A simple calculator language example provided with the JFlex distribution
- The calculator language permits expressions involving + and *, possibly including parentheses, and terminated by a semicolon
`5*(1+2);`
- You can also examine the lexer output
- Check it out in [~/../public/Examples/DC](#)
- Will be discussed in section on Friday

Lookahead

- A token is matched and its action taken when no longer match is possible
- Recognizing a match may involve reading many more characters beyond the end of the matching lexeme
- Consider the token patterns "x" "y" "xxxxxxxxxy" and the input string "xxxxxxxxxy"
- Sometimes you want to specify particular words that must follow a token for the token to match
- This requires explicit *lookahead*, supported by some regular expression systems

Returning to Fortran

- One is an assignment statement, the other a loop

`do i = 1.25`

`do i = 1,25`

- But white space is irrelevant in Fortran

`doi=1.25`

`doi=1,25`

- But the following is a legal FORTRAN assignment statement:

`doi=1`

- A lexer needs to look ahead to the comma to know how to tokenize these statements

Specifying a lookahead pattern

- In JFlex, explicit lookahead is specified using the "/" slash operator in a regexp
- **R1 / R2** means: Match regular expression **R1**, but only if followed by "trailing context" text that matches **R2**
- If the first part of **R2** is the same as the last part of **R1** and **R1** has unbounded length, there may be problems!

Solving the FORTRAN problem

- Let's restrict the expressions before and after the comma to integers and variable names

id = [[:letter:]]([[:letter:]] | [[:digit:]])*

num = [[:digit:]]+

- Match the keyword **DO** only with the correct trailing context:

"DO" / id "= " num ", "

- The scanner will recognize **DO** as the lexeme, but not until it looks ahead and matches the pattern to the right of the /
- Only the **DO**, not the lookahead, is consumed when this matches

The Theory Behind Lexer Generators

Lexical rules

- A lexical rule has two parts:
 - Specification of a **token pattern**
 - A regular expression (regexp)
 - Specification of **actions** the lexer must take when that pattern is matched
 - A block of code to be executed
- In A1, after specifying your set of rules in a file, you let JFlex build your scanner (lexer) – in particular, your `get_token()` function
 - But what really goes on behind the scenes? That is, what is JFlex doing for you?

A cartoon

- Starting with the token pattern/action rule specification...
- Construct an NFA
 - recognizes the regular language (RL) specified by the regexps
 - associate actions with accepting states
 - simulate the NFA
- But NFAs can be slow
- Transform the NFA into a DFA using the subset construction, and simulate the DFA
- Uses lots of space, transformation takes time, but shorter simulation time
- So, we then compact the states into less space
- Tradeoffs between these approaches: see Sec. 3.7

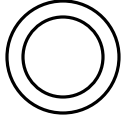
Non-deterministic finite automata

- Multiple transitions out of a single state using the same symbol
- Transitions on no symbol at all: ϵ transitions
- Simulation is expensive
 - At any point in the scanning process.. we can be in multiple states, and we need to determine transitions from those states
 - With R rules and N tokens, $O(R)$ states, $O(RN)$ worst case time

NFA from regular expressions

- Regular languages can be built up from simpler RLs
 - Operations: union, concatenation and Kleene closure (star)
 - Regexps are a way of writing those operations
- To build an NFA from any regexp
 - Start with NFAs recognizing the simplest RLs which are part of the expression: the empty string, single characters, and character sets
 - Then build NFAs that recognize the union, concatenation and Kleene star of the simple RLs, and continue from there
 - The algorithm is called “**Thompson’s construction**”
- Each rule action is associated with an accepting state
- Runs the action code when in an accepting state

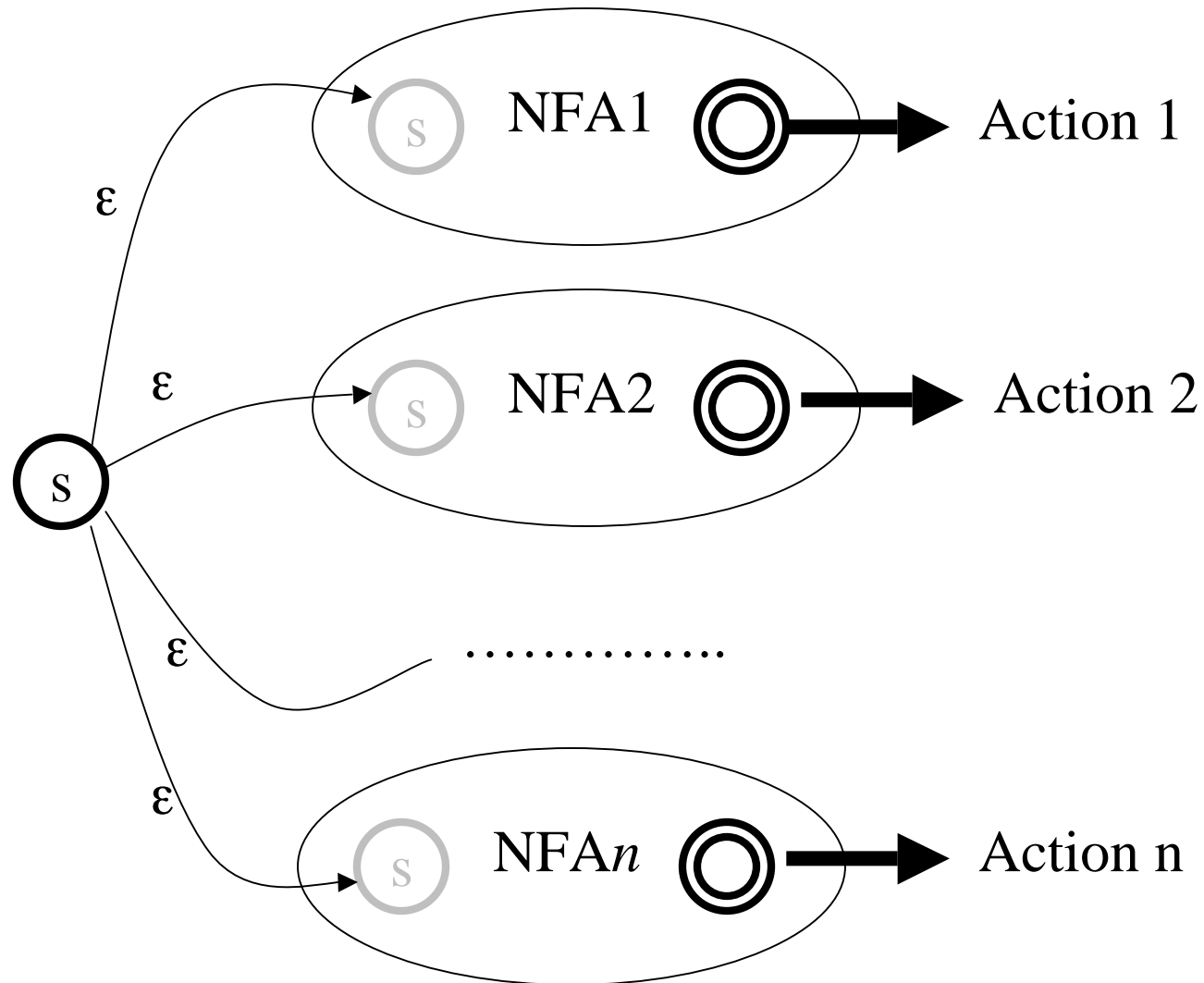
NFA notation

- NFA start state labeled with “s”
- NFA accept state marked with concentric circles: 
- Arrowed arcs represent transitions
- Label on arc represents character to match, or ϵ
- In this construction, a regular language will be recognized by a NFA where:
 - There is exactly one accept state (and one start state)
 - Start state has no incoming transitions, accept state has no outgoing transitions
 - Every state has either one character transition, or a most two ϵ -transitions, leaving it (cf pp. 124)

One NFA for all of the rules

- Each lexical rule has a token pattern given as a regexp
- Each of these regexps has a corresponding NFA
- All of these NFAs (one for each lexical rule) can be joined into a single NFA ...

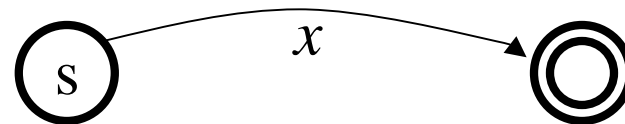
NFA for lexical rules 1,...,n



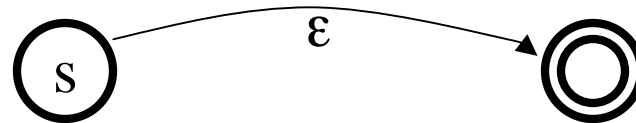
NFAs for simple languages

- For each regexp in the lexical rules, consider occurrences of single characters, character sets, and the empty string, and construct NFAs for these:

- Character or char set x :

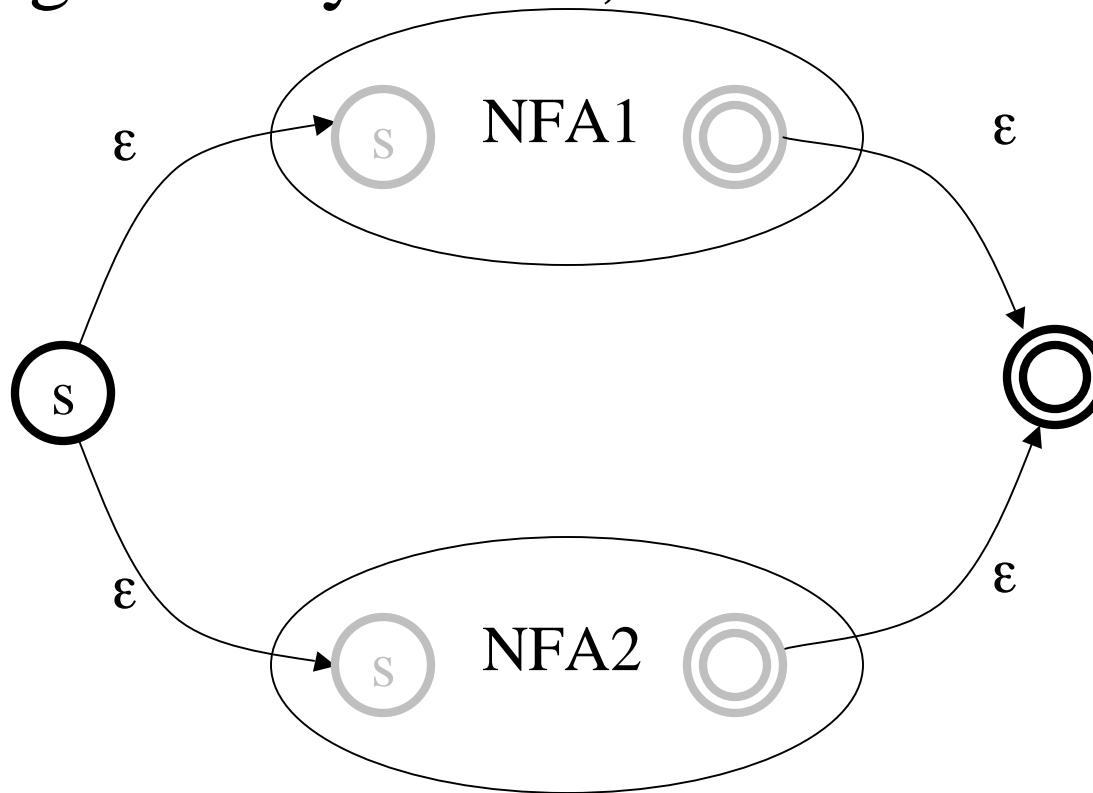


- Empty string (epsilon transition):



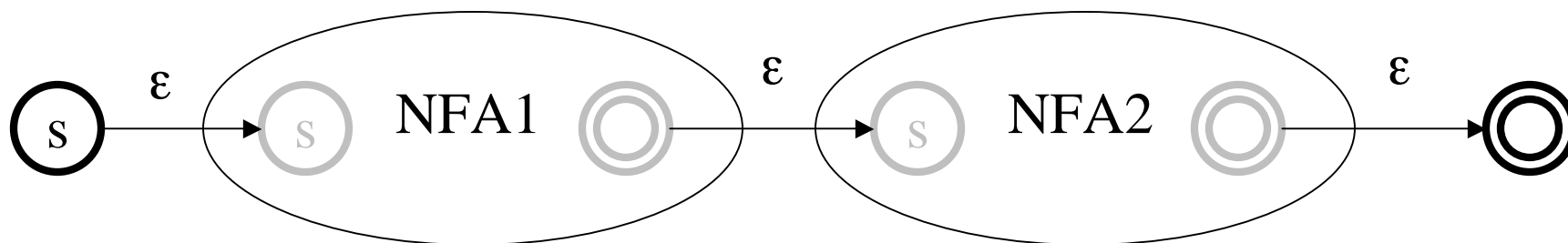
NFA for union of languages

- Create NFA for union of languages recognized by NFA1, NFA2



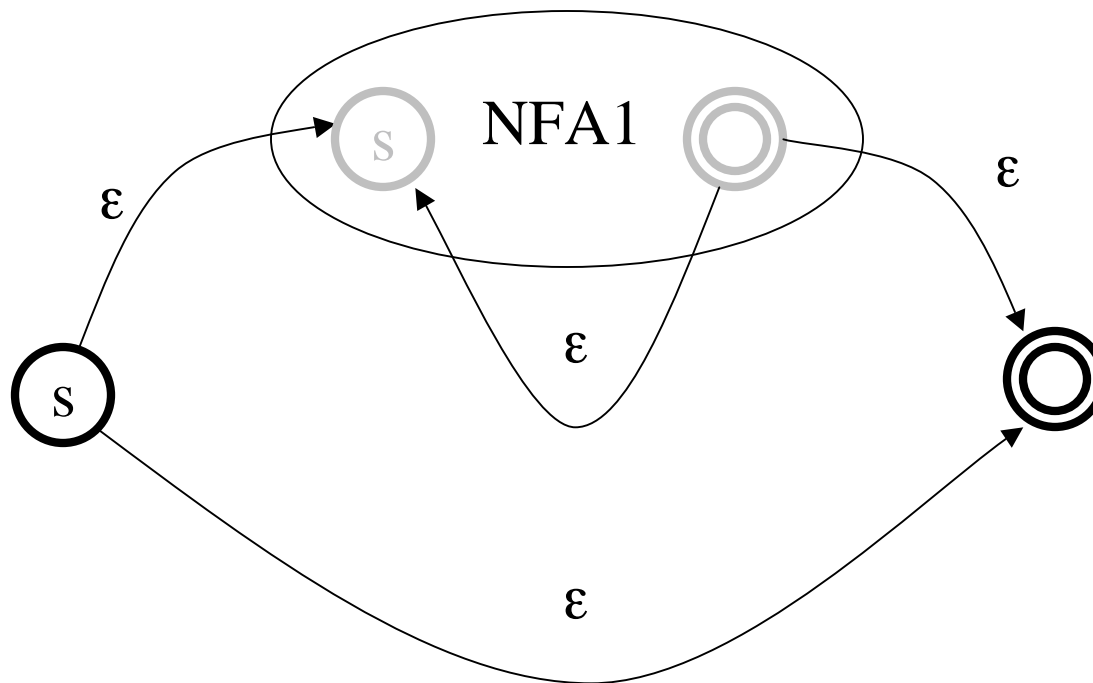
NFA for concatenation of languages

- Create NFA for concatenation of languages recognized by NFA1, NFA2



NFA for Kleene star of a language

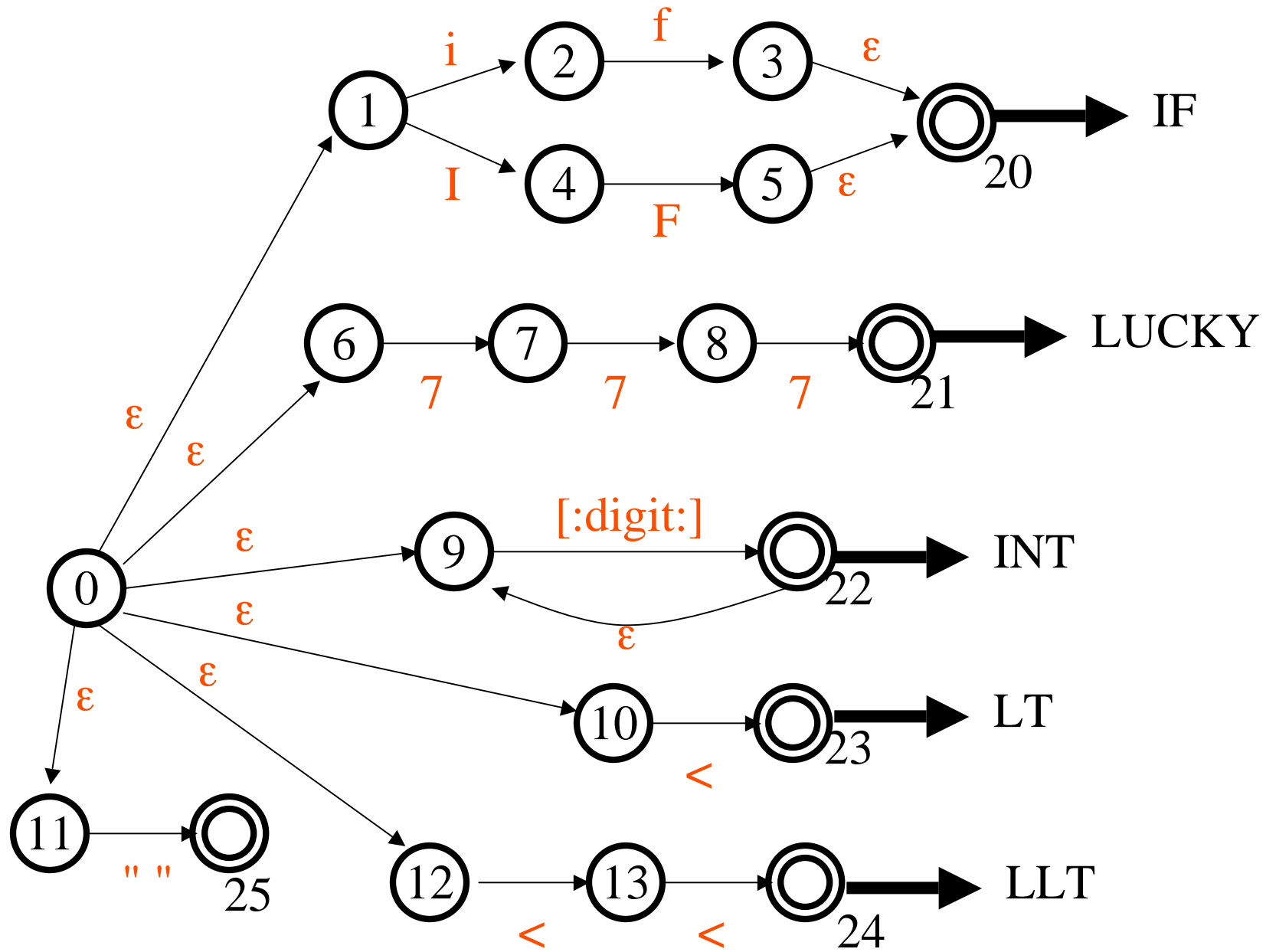
- Create NFA for Kleene closure (star) of a language recognized by NFA1



Example

- A NFA recognizing the RLs specified in these rules is shown on the next slide...

<code>“if” “IF”</code>	<code>{ return new Symbol(sym.IF) }</code>
<code>777</code>	<code>{ return new Symbol(sym.LUCKY) };</code>
<code>[:digit:]+</code>	<code>{ return new Symbol(sym.INT,yytext()); }</code>
<code>“<”</code>	<code>{ return new Symbol(sym.LT); }</code>
<code>“<<”</code>	<code>{ return new Symbol(sym.LLT); }</code>
<code>[\t\n\r]</code>	<code>{ /* do nothing if encounter whitespace */ }</code>



Complexity of Thompson's construction

- Begin with 2-state NFA for each single character that appears in reg expressions
- Each regexp operation adds at most 2 states
- Each state (except the overall start state) has at most 3 transitions leaving it
- If total length of all regexp rules is R , the construction takes time $O(R)$ and space $O(R)$