CSE 131A
"Compiler Construction I"

Lecture 6

Lexical scanning theory

Announcements

- Groups have been set up for users in the file ~/../public/Tools/groups/groups_1
- A2 will be posted by Saturday morning
- New tests for lexical errors

New “error tests”

- err1.xqy

  ^ ^ ^

```
<?xml version="1.0" encoding="UTF-8"?>
<OnyxSource filename="err1.xqy">
  <token column="1" id="64" line="1">^</token>
  <token column="3" id="64" line="1">^</token>
  <token column="5" id="64" line="1">^</token>
  <token column="-1" id="0" line="-1"/>
</OnyxSource>
```

Lexically correct, syntactically incorrect

if (3 > 1) thev 4 elsf 5

```
<OnyxSource filename="err2.xqy">
  <token column="1" id="54" line="1"/>                  IF
  <token column="4" id="5" line="1"/>                   LPAREN
  <token column="5" id="43" line="1">3</token>          INTEGERLITERAL
  <token column="7" id="22" line="1"/>                  GREATER
  <token column="9" id="43" line="1">1</token>          INTEGERLITERAL
  <token column="10" id="6" line="1"/>                  RPAREN
  <token column="12" id="4" line="1">thev</token>       QNAME
  <token column="17" id="43" line="1">4</token>         INTEGERLITERAL
  <token column="19" id="4" line="1">elsf</token>       QNAME
  <token column="24" id="43" line="1">5</token>         INTEGERLITERAL
  <token column="-1" id="0" line="-1"/>                 EOF
</OnyxSource>
```

Lexical contexts

- Some patterns may have different meanings in different contexts
- For example, character entities inside strings

```
“< > & &amp;”
```

```
<token column="1" id="45" line="1">&lt; &gt; &amp; &amp;</token>
```

- Tag start symbol in a string

```
"3 < 4"
```

```
<token column="1" id="45" line="1">3 &lt; 4</token>
```

- Less than operator in an expression

```
3 < 4
<token column="1" id="43" line="1">3</token>
<token column="3" id="21" line="1"/>
<token column="5" id="43" line="1">4</token>
<token column="-1" id="0" line="-1"/>
```

Multiple automata

- We can build our lexical analyzer with a single state machine
- To handle multiple context we can have specialized state machines, one for each context
- The rule actions switch between state machines
- JFlex provides *named lexical states* to this end

Jflex named states

- The initial (default) state: YYINITIAL
- Other states are defined in the file options section of the specification

```
%state STRING
```

- Named blocks in the rules section define the named state
- We switch states with yybegin( )

```
<YYINITIAL> { \"     { string.setLength(0); yybegin(STRING); } }
<STRING> {
 \"                  { yybegin(YYINITIAL);
                       return symbol(sym.STRING_LITERAL,
                                     string.toString()); }
[^\n\r\"\\]+         { string.append( yytext() ); }
// more rules
 }
```

Under the hood of a scanner generator

- Starting with the token pattern/action rule specification…
- Construct an NFA that
  - recognizes the regular language (RL) specified by the regexps
  - associates actions with accepting states
  - simulates the NFA
- The scanner generator produces code that implements the NFA

EXTRA TEXT

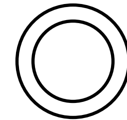
- NFAs can be slow
- Transform the NFA into a DFA using the subset construction, and simulate the DFA
- Uses lots of space, transformation takes time, but shorter simulation time
- So, we then compact the states into less space

Recapping from last time

- 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

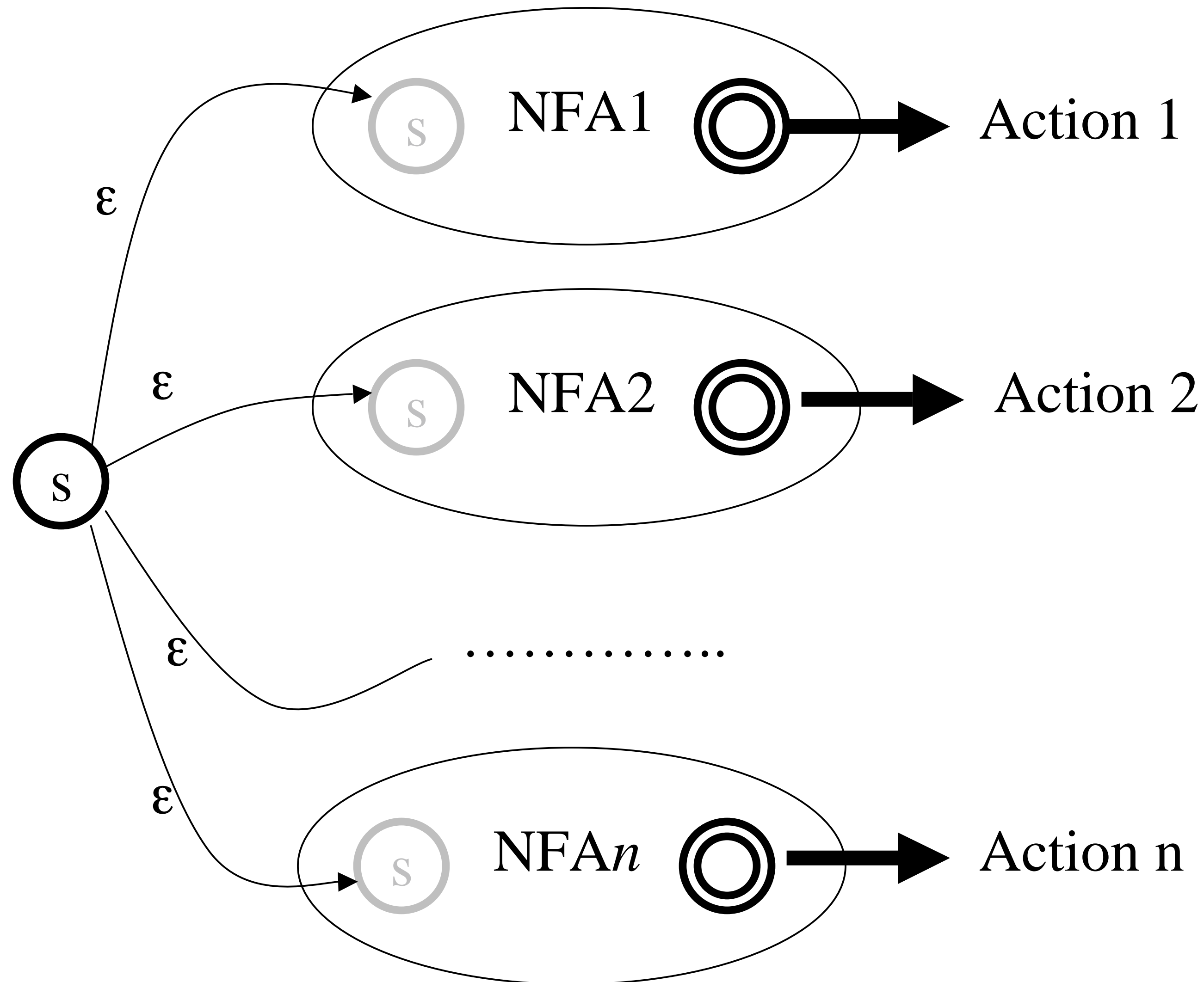
NFA notation

- 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)

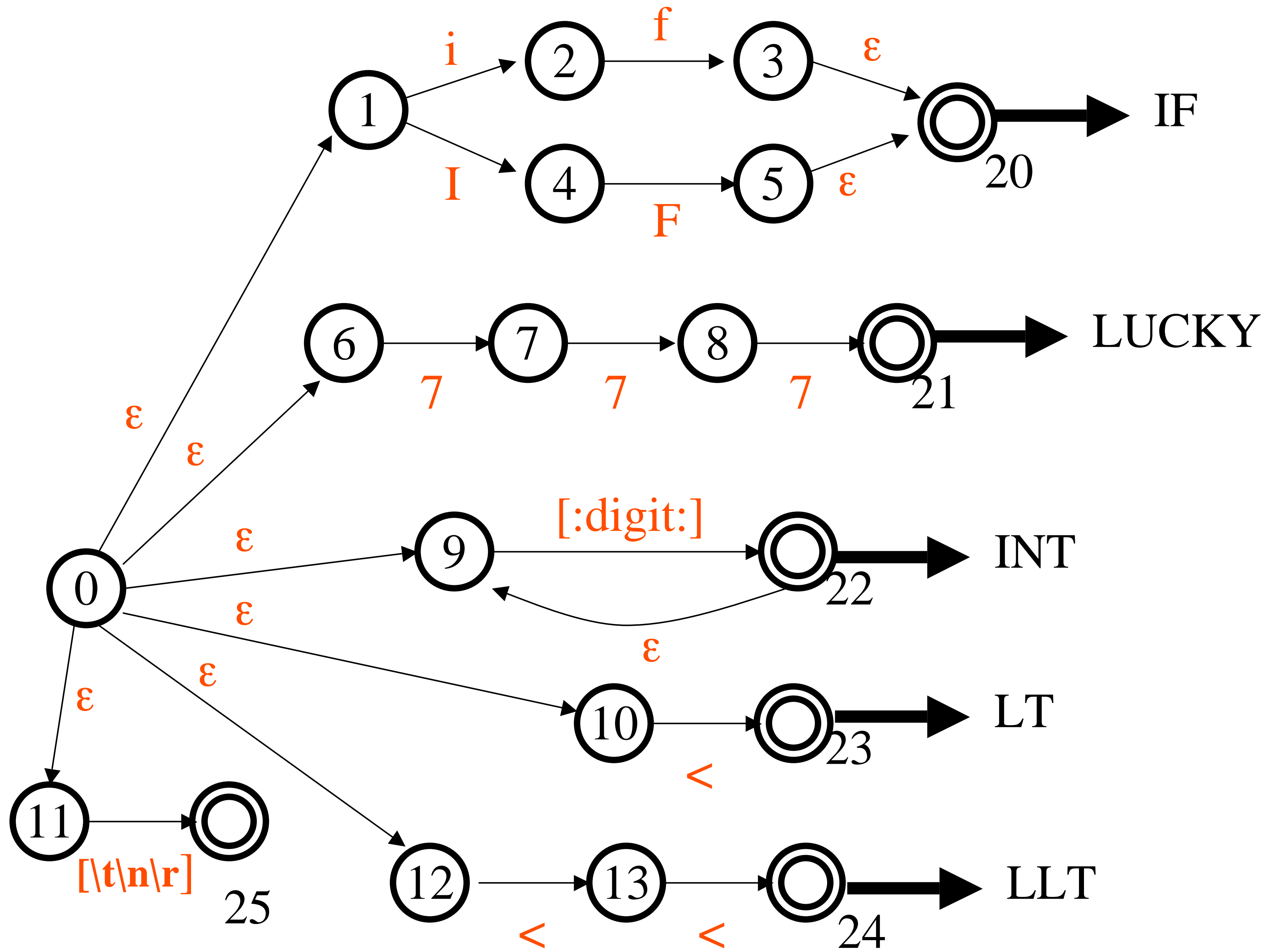
NFA for lexical rules 1,…,n



Example

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

“if” | “IF”	{ return new Symbol(sym.IF) }
777	{ return new Symbol(sym.LUCKY) };
[:digit:]+	{ return new Symbol(sym.INT,yytext()); }
“<”	{ return new Symbol(sym.LT); }
“<<”	{ return new Symbol(sym.LLT); }
[\t\n\r]	{ /* do nothing if there’s whitespace */ }


0
1
2
3
4
5
6
7
8
9
10
11
12
13
20
21
22
23
24
25
i
f
I
F
ε
7
7
7
[:digit:]
<
<
<
[\t\n\r]
IF
LUCKY
INT
LT
LLT

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)

Implementing an NFA for lexical analysis

- NFA may have more than one transition on a given character input, and even the possibility of transitions on *no* input (epsilon transitions)
- To implement an NFA, must keep track of set of *all* states that can be reached from the start, given the input read so far
- Assume text input has length N, and that the total length of regexp rules is R
  - Must consider O(R) states when each character is read, so scanning the input takes O(RN) time

NFA simulation algorithm – get_token

- The function get_token simulates the NFA
- It scans the input and carries out the required state transitions corresponding to the specified patterns of input
- It carries out the action for the recognized pattern when the NFA is an accept state, and there exists no rule for a longer pattern

Bookkeeping

- Assume there is an input buffer holding the input text
- We will need a variable left, pointing to the first character in the input not yet tokenized
- We will need a variable right, pointing to the next character to be read from the input, and a method nextChar() that accesses this character
- Since we may reach termination in a state set that contains no accepting states, we may need to backtrack:
- lastchar and laststate refer to the rightmost character associated with an accepting state, together with that state

Helper functions

- In simulating an NFA, it is useful to define 3 helper functions
- The ε-closure: given a set of states, what states are reachable following ε-transitions only
- Move set: The set of states that can be reached from given a set of states, obtained by following the transitions labeled by a given character
- Termination: Are any new states reachable from a given set of states, by following transitions labeled by a given character?

Epsilon-closure

- In simulating an NFA, it is useful to be able to compute the ε-closure of a set of states
- The ε-closure of a set of states S is a set of states T such that a state t is in T if and only if
  - t is in S, or
  - t can be reached from a state in S by following only ε-transitions
- Intuitively, the ε-closure gives you the states a NFA can be in next without reading any input
- The ε-closure is not hard to compute...

Move set

- Similar to, but simpler than, the ε-closure of a set of states is the move set of a set of states given a character in the input
- The move set of a set of states S with respect to character c is a set of states T such that a state t is in T if and only if t is reachable from a state in S by following a transition labeled with c
- Intuitively, the move set gives you the states a NFA can be in next by consuming a character from the input

Computing the move set

```
Set<Integer> move(Set<Integer> S, char c) {

    Set<Integer> move = new Set<Integer> ();
    //add all c-next states for every state in s to the result
    for(Iterator I = S.iterator(); i.hasNext(); ) {
        Integer st =   i.next();
        move.addAll(  next(c, st.intValue()) );
    }

    return move;
}
```

- States t
  - *t* is reachable from a state in *S* by following a transition labeled with *c*
  - Possible next states when it consumes a single character from the input

Termination

- Lexical analyzers typically use the convention that the longest regexp match wins
- This means that it is not enough just to reach a set of states that contain an accept state; we have to make sure there is no longer match possible
- We can be sure of this if the state set has reached termination, i.e., the ε-closure of the set has empty move set with respect to the next input character
- If ε-closure of move set is not empty, must continue… but keep track of the configuration in which we found an accept state, *just in case we need to backtrack*

Computing termination

```
boolean isTerminated (Set<Integer> s, char c) {
    Set<Integer> next = move(eClose(s),c);
    return next.isEmpty();
}
```

NFA simulation algorithm (get_token)

1. Initial conditions: left pointer and right pointer at start of input text, and NFA in start state
2. Increment right pointer and read next char from input as current char
3. Augment current set of states with states reachable with epsilon transitions from current states
4. If current set of states contains an accept state and no transition from current set of states matches current character (requires function to map state × character →state)
   a. Take action of first rule among accept states in the current set; Matching lexeme is between left and right pointers.
   b. Move left pointer to right pointer.
   c. Make start state current state and go to (3).
5. If current set of states does not contain an accept state and no transition matches current character, rewind pointer to last state set that contained an accept state and go back to (4b).
   ERROR if there is no such last accept state.
6. Otherwise, set the current set of states to be those reachable by a transition on the current character; go to 2

get_token

```
int left=0, right =0, lastchar, laststate; char ch;
 Symbol get_token() {
   Set<Integer> s = eClose(0);            // compute epsilon closure of start state
   lastchar=laststate=-1;                 // no accepts yet
   left = right;                          // move left pointer over to right
   while(true) {
        if(hasAccept(s)) {                // remember these
                lastchar = right; laststate = firstAccept(s);
        }
        if(isTerminated(s)) {  // go back to last accept, take action
                right = lastchar + 1;
                // now lexeme is text between left and right
                return doAction(laststate);             //error if laststate < 0
        }
        ch = nextChar(); right++;         // read next char from buffer
        s = eClose(move(s,ch));           // advance the NFA state set
   }
 }
```

Simulation

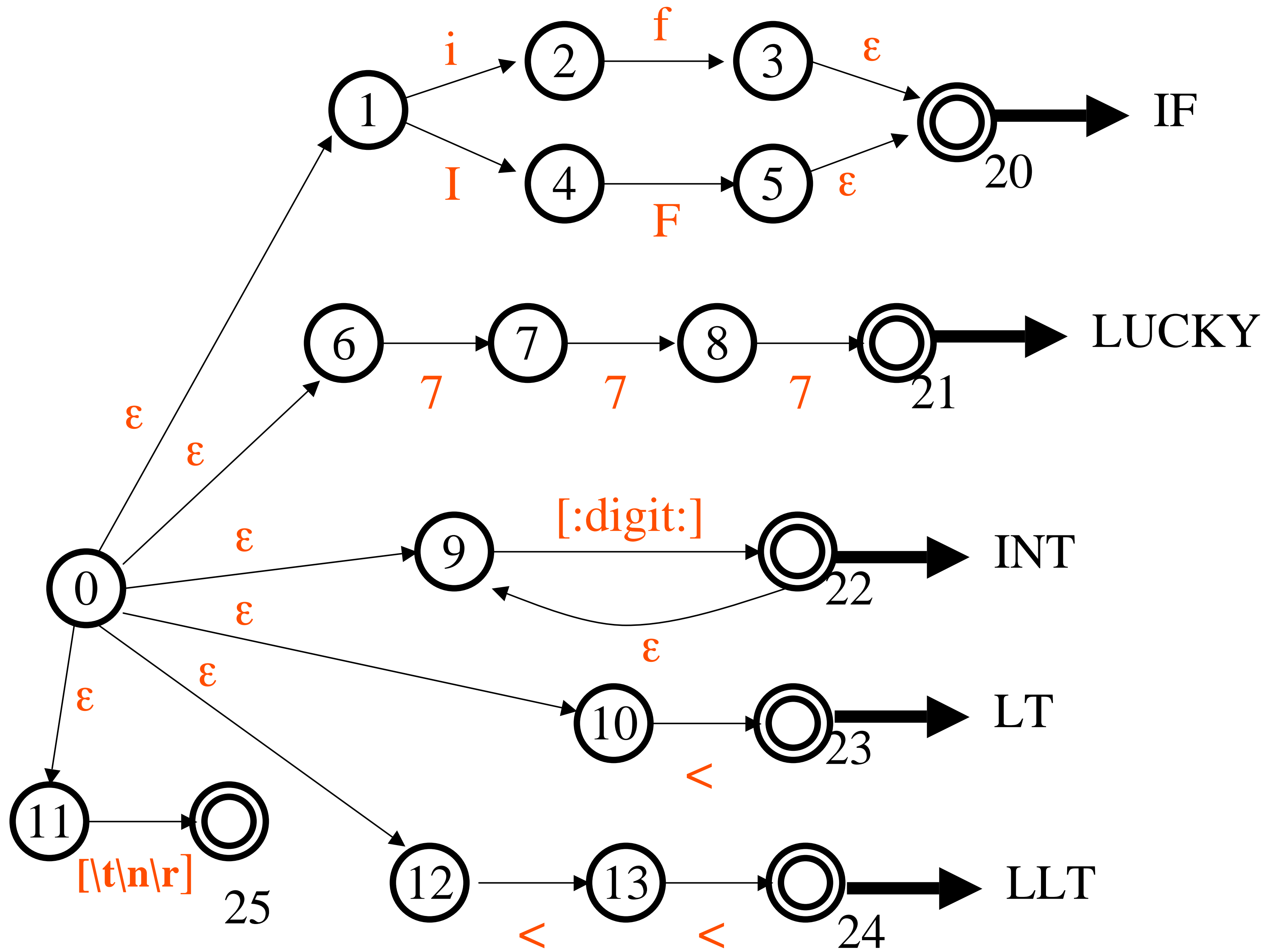
```
Set<Integer> S = eClose(0);  // compute epsilon closure
                             //  of start state
 a = nextchar;

while(! EOF) {
       S = eClose(move( S , a ));
        a = nextchar;
 }
  return (S contains an accept state)
```

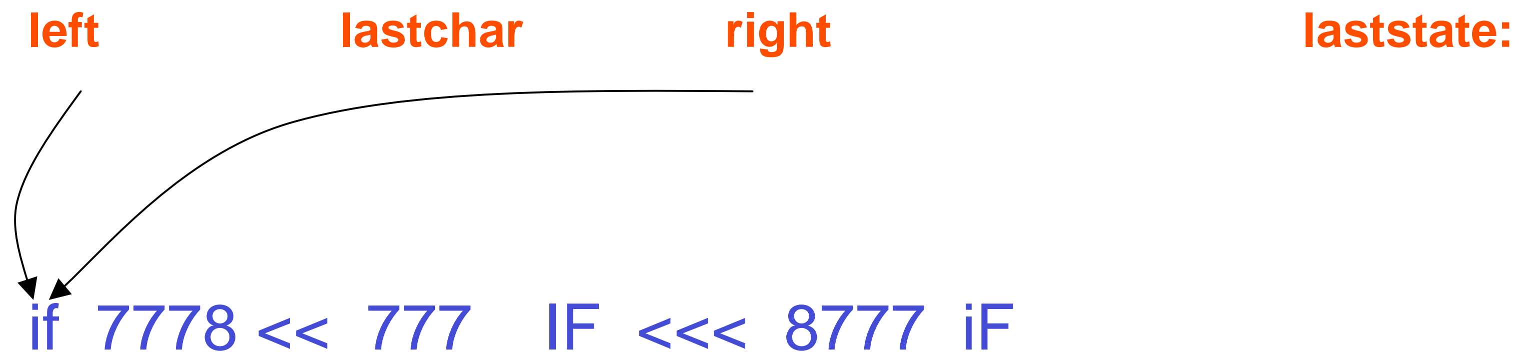
Example

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

<table>
<tr><td>“if” | “IF”</td><td>{ return new Symbol(sym.IF) }</td></tr>
<tr><td>777</td><td>{ return new Symbol(sym.LUCKY) };</td></tr>
<tr><td>[:digit:]+</td><td>{ return new Symbol(sym.INT,yytext()); }</td></tr>
<tr><td>“<”</td><td>{ return new Symbol(sym.LT); }</td></tr>
<tr><td>“<<”</td><td>{ return new Symbol(sym.LLT); }</td></tr>
<tr><td>[\t\n\r]</td><td>{ /* do nothing if there’s whitespace */ }</td></tr>
</table>

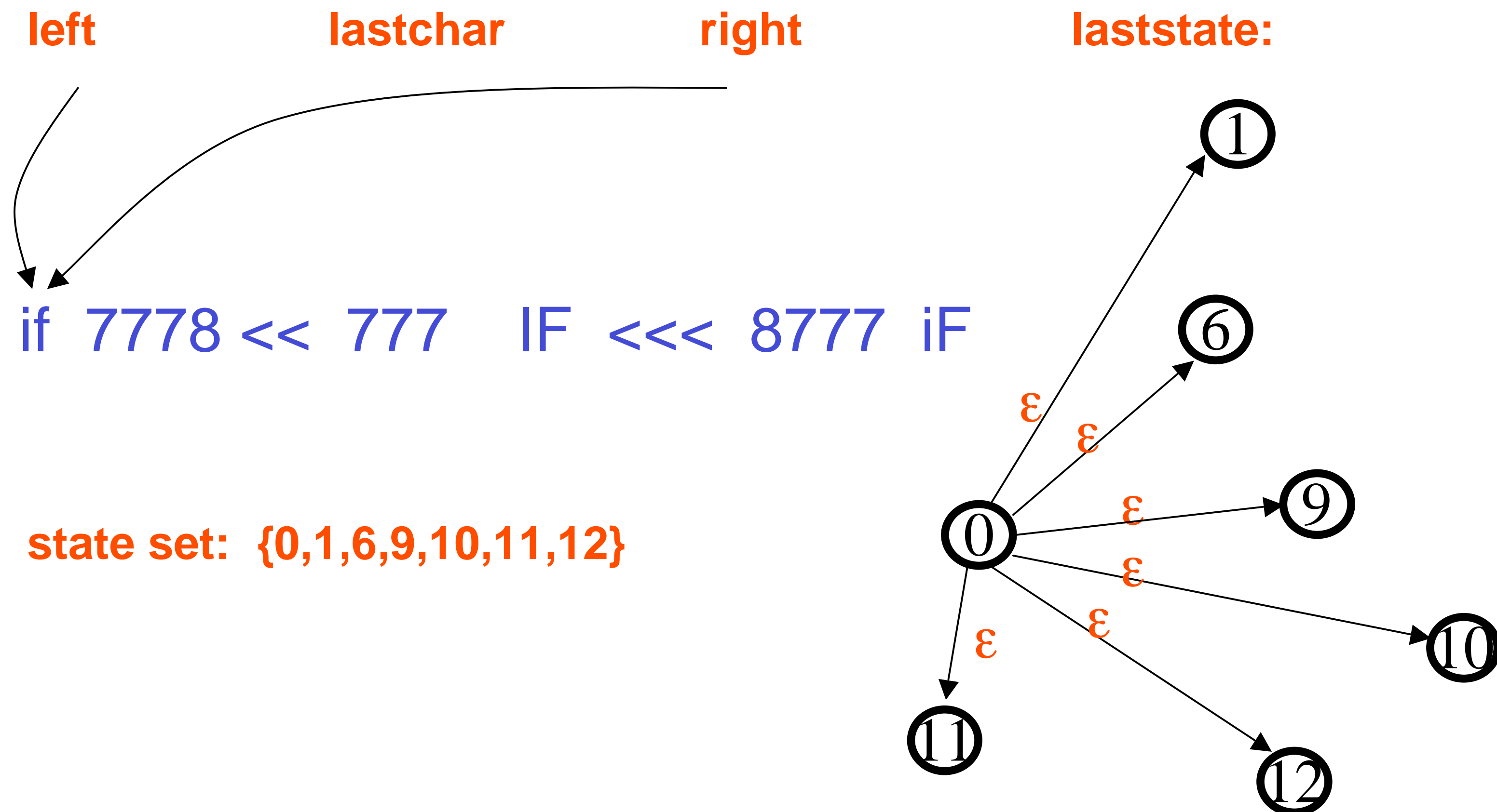

i
f
ε
IF
I
F
ε
20
7
7
7
LUCKY
21
ε
ε
ε
[:digit:]
INT
22
ε
ε
ε
ε
LT
23
<
[\t\n\r]
25
<
<
LLT
24
0
1
2
3
4
5
6
7
8
9
10
11
12
13

Tokenize this input text

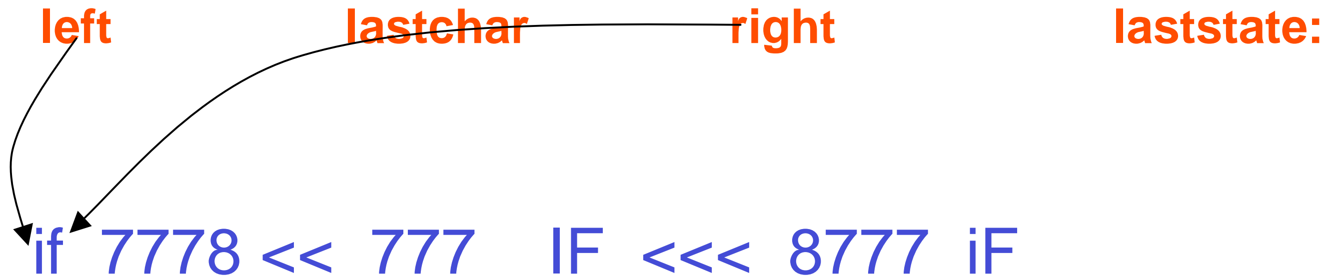


state set: {0}

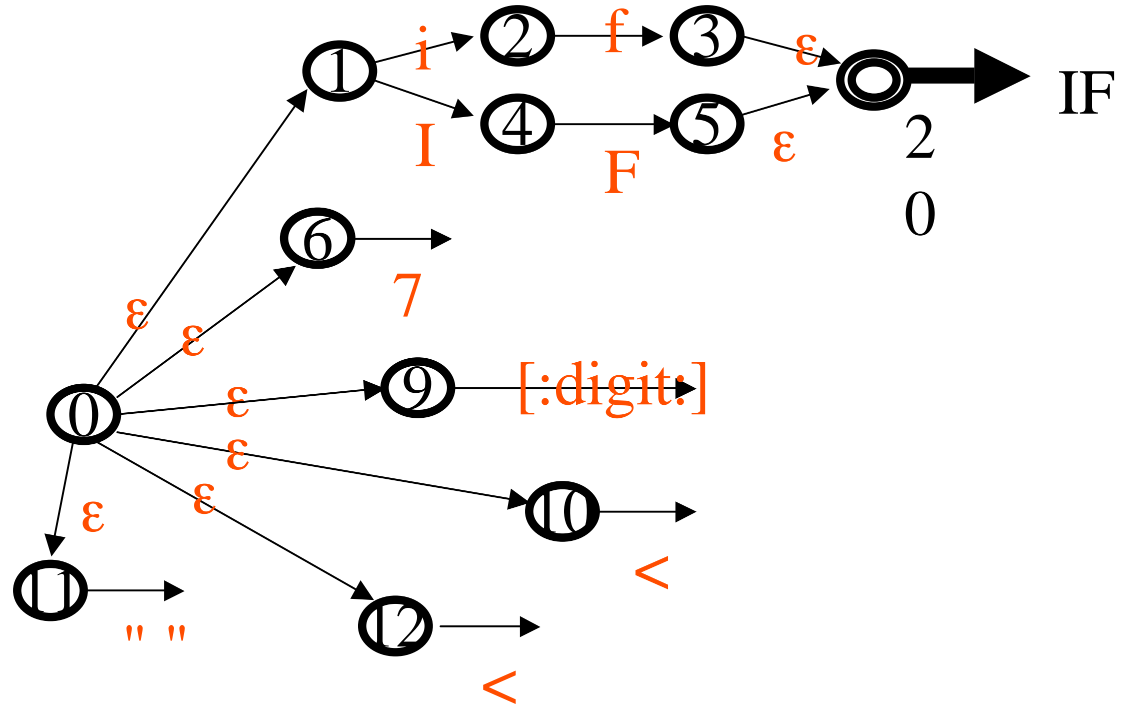
Tokenize this input text



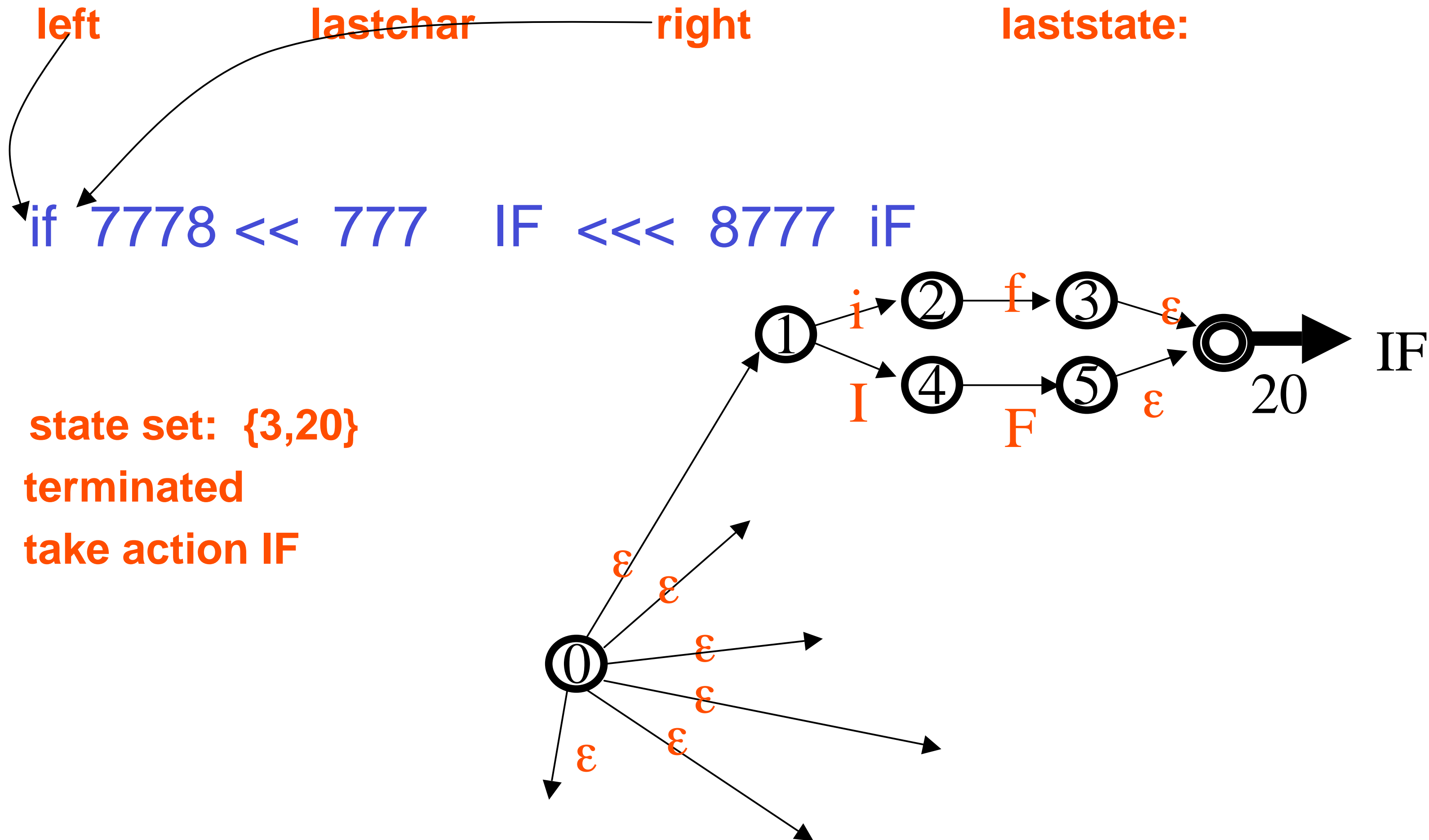
Tokenize this input text



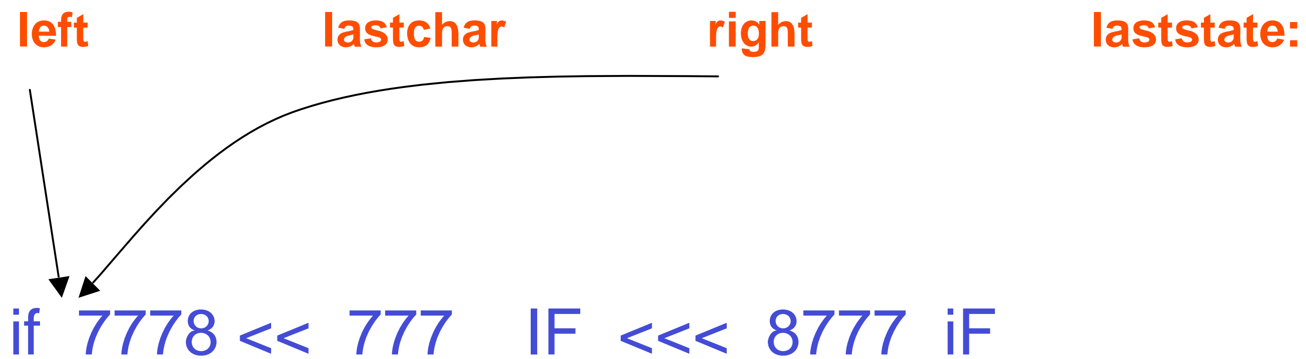
state set: {2}



Tokenize this input text

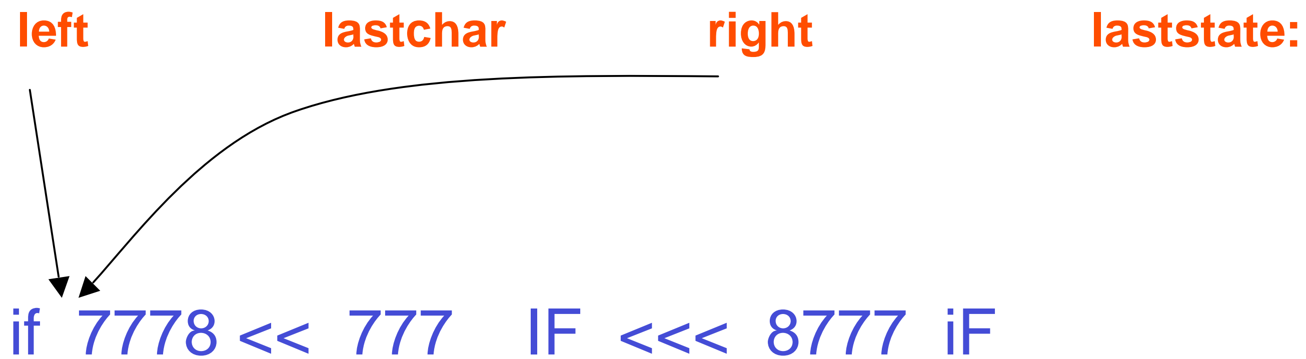


Tokenize this input text



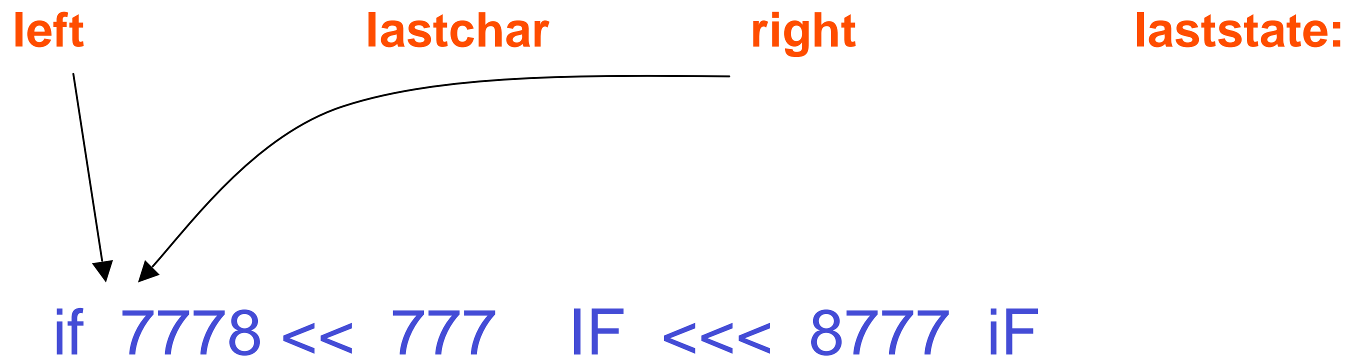
state set: {0}

Tokenize this input text



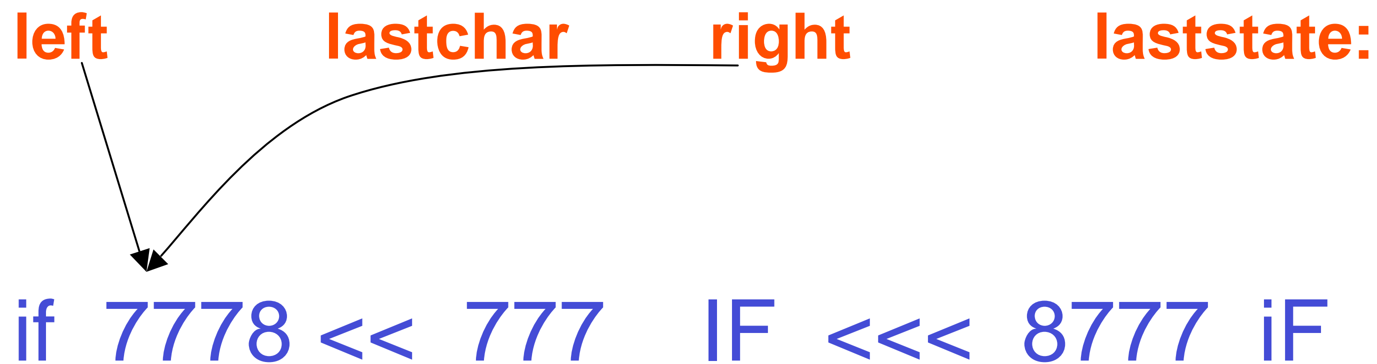
state set: {0,1,6,9,10,11,12}

Tokenize this input text



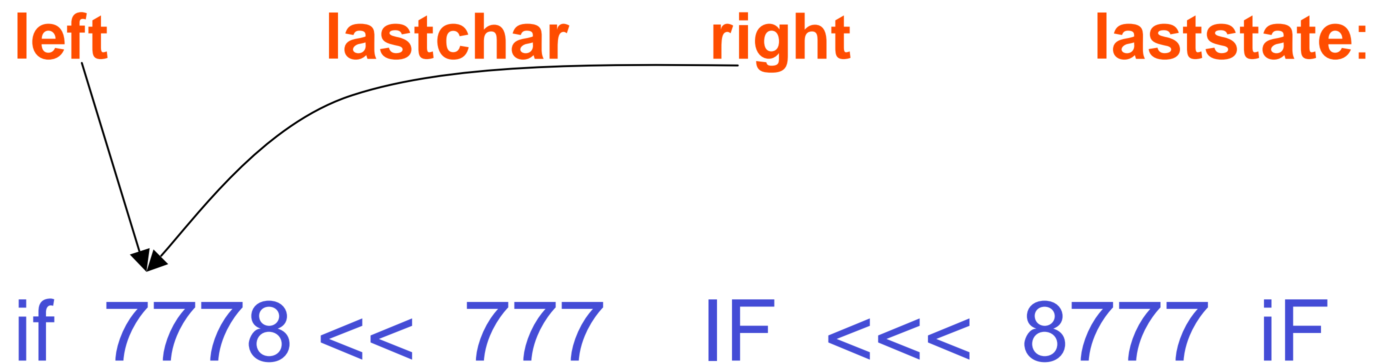
state set: {25}, terminated, no action specified

Tokenize this input text



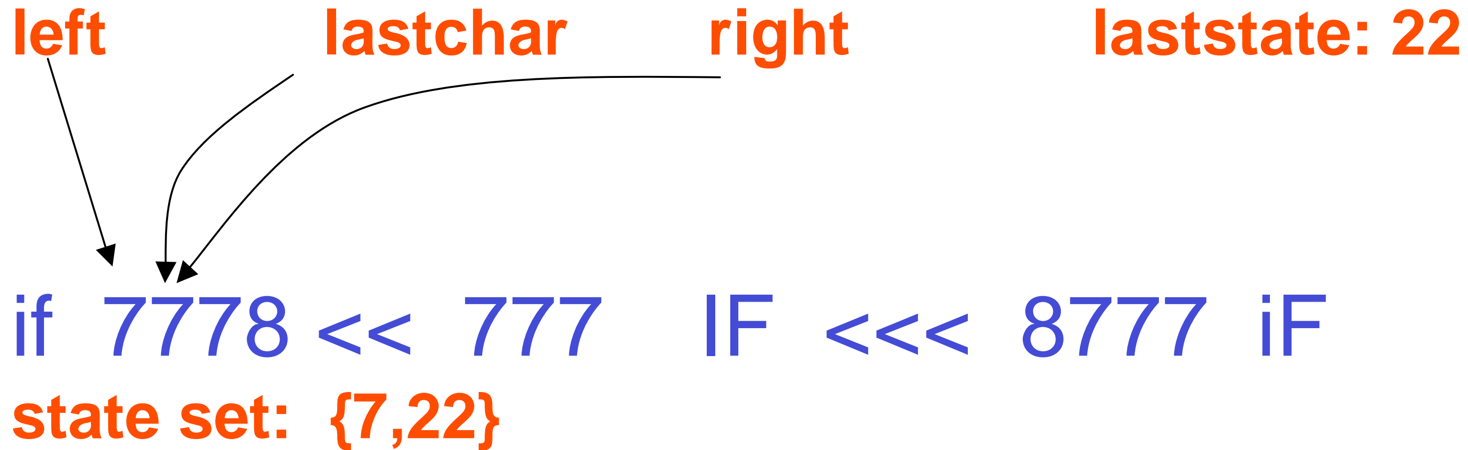
state set: {0}

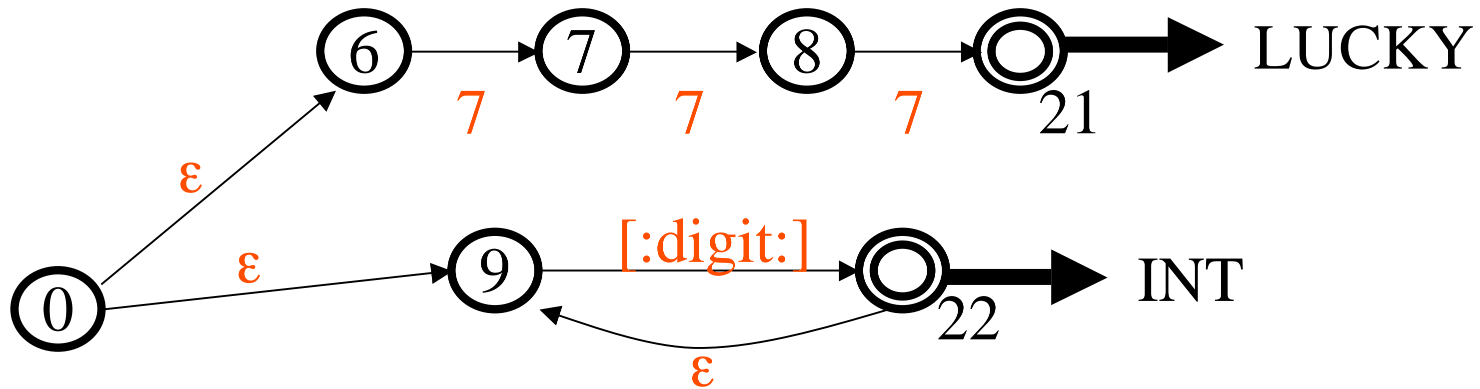
Tokenize this input text



state set: {0,1,6,9,10,11,12}

Tokenize this input text



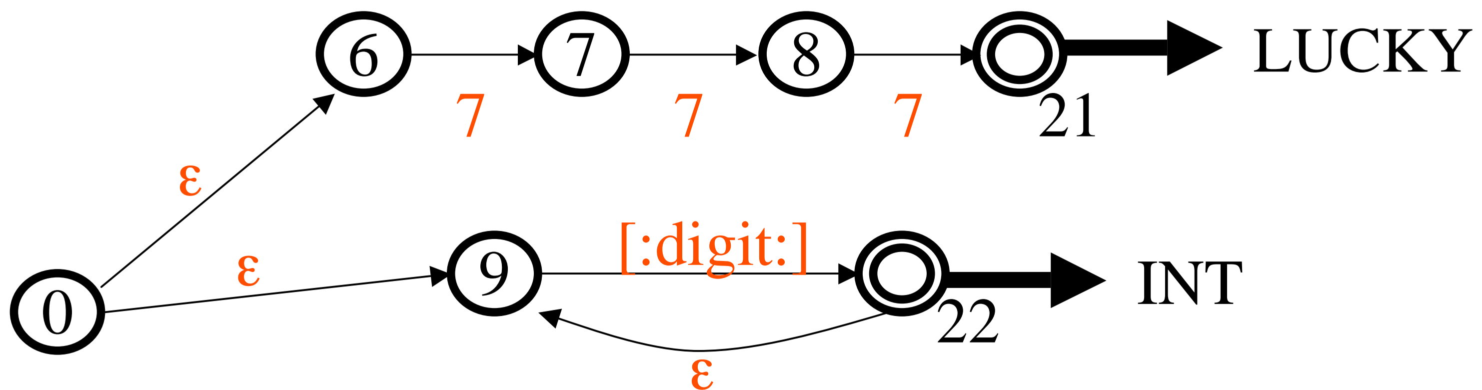


Tokenize this input text

left **lastchar** **right** **laststate: 22**

if 7778 << 777 IF <<< 8777 iF

state set: {8,22}

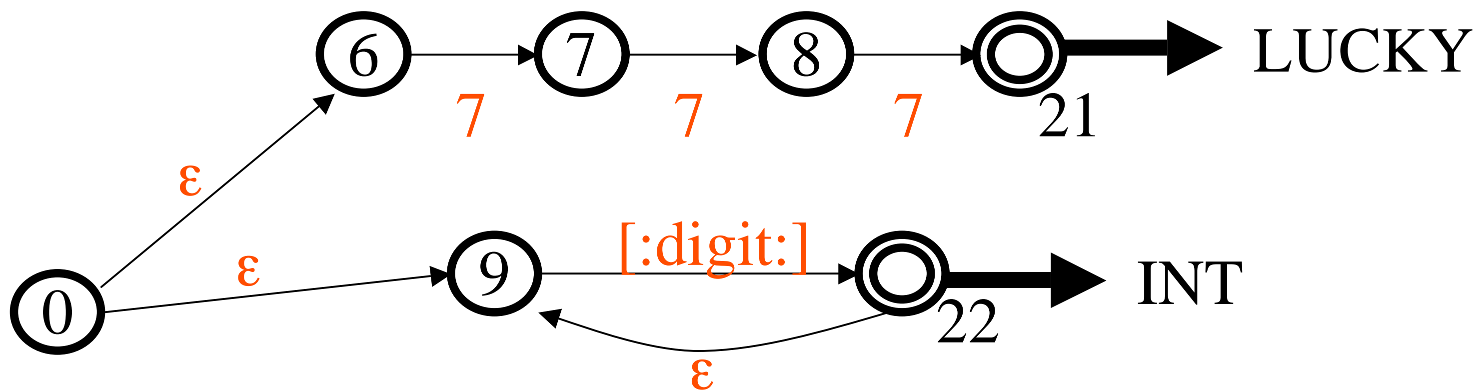


Tokenize this input text

left **lastchar** **right** **laststate: 21**

if 7778 << 777 IF <<< 8777 iF

state set: {21,22}



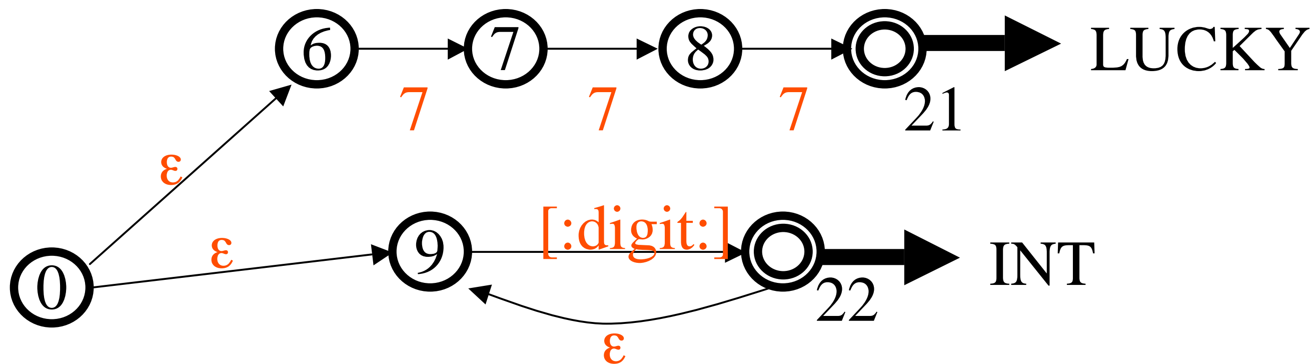
Tokenize this input text

left **lastchar** **right** **laststate: 22**

if 7778 << 777 IF <<< 8777 iF

state set: {22}, terminated, action INT

Continue this on your own…



Next

- Syntax analysis and parsing

Parsing

- So far we've talked about how to perform lexical analysis: how to recognize tokens
  - Tokens are regular languages…
  - … specified with regular expressions
- We next look at how to perform syntactic analysis: how to *parse*
  - Syntactically correct token patterns are context-free languages...
  - ... specified with a context-free grammar

The role of a parser

- A parser works in concert with the scanner
- From parser's perspective, the scanner is a function that returns the next token
- The parser also interacts with other modules including **error recovery, symbol table manager,** and **code generation**
- We'll look at all of these eventually

Differences between scanning and parsing

- Scanning is a linear process
  - The scanner reads its input one character at a time
  - Output from the scanner is a linear sequence of tokens, that are members of a specified regular language
- Parsing is a hierarchical process
  - The parser reads tokens from the scanner one at a time
  - The parser builds a hierarchical structure (**parse tree** or **abstract syntax tree**) from the tokens, according to the specification of a context free grammar

Differences in computational power

- Scanners can do things like lookahead
- But they don't support recursive definitions of structure
  - A scanner can't tell if parentheses are balanced
  - We need a parser for that

An analogy

- Text can be made up of paragraphs, which are made up of sentences, which are in turn made up of words
- If we scan the characters one at a time, we can detect many spelling errors
- But even if all the words are spelled correctly, the structure they form may not make sense or may not be grammatically correct
  - When in the course of human events (http://www.law.indiana.edu/uslawdocs/declaration.html)
  - We the People of the United States, in Order to form (http://www.archives.gov/national-archives-experience/charters/constitution_transcript.html)

Why Context Free Grammars?

- Many programming language constructs are inherently recursive
- For example, in Java if-statements are statements that can contain statements within them
- So the definition of what a statement is in Java must mention Java statements!
- This kind of "center embedded" recursive definition cannot be expressed with regular expressions
- We need **context-free grammars**

Next time

- Context free grammars
- Grammar productions and derivations
- Syntax trees
- Ambiguity

Some terminology

- A **context free grammar** G consists of:
  - A finite set Σ of *terminal symbols*.
    The alphabet of the language defined by the grammar
    Also referred to as tokens
  - A finite set N of *nonterminal symbols*. These act as syntactic variables designating sets of strings defined by the grammar
  - A distinguished *start symbol* $S \in N$.
  - A finite set of *rules* or *productions* of the form
    $$A \rightarrow \alpha$$
    where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$

Example

statement → **if** "(" *expr* ")" **then**
statement **else** *statement*

Non-terminals in *italics*

Terminals in **bold face** or
as a "quoted string"

A simple arithmetic expression grammar

- Productions

  expr → *expr op expr*
  expr → (*expr*)
  expr → **-** *expr*
  expr → **id**
  op → **+ | - | * | / | ^**

- Start symbol

  expr

- Terminals

  id + - * / ^ ( )

- Nonterminals

  expr op

Notational conventions

- Use UPPER CASE for designating non-terminal variables
- If there are productions with the same left hand side, use '|' to designate alternatives
- Two notational variants of the same grammar

$E \rightarrow E \ \ OP \ \ E \mid (\ E \ ) \mid \ - E \mid \textbf{id}$

$\boldsymbol{OP} \ \ \rightarrow + \mid \textbf{-} \mid * \mid / \mid \wedge$

expr → expr op expr
expr → (expr)
expr → **-** expr
expr → **id**
op → **+** | **-** | ***** | **/** | **^**

Derivations

- By convention, lower case Greek letters $\alpha\ \beta\ \gamma$ represent strings of grammar symbols
  (terminals or nonterminals)
- $\Rightarrow$ means a one-step derivation
- Starting with a sequence of grammar symbols, replace one nonterminal with its definition
- For example $\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$
- Where $A \rightarrow \gamma$ is a production in the grammar
- A derivation is a sequence: $\alpha_1 \Rightarrow \alpha_2 \ldots \Rightarrow \alpha_n$

  We say that "α_1 derives α_n"

Notation for derivations

- $\Rightarrow$ means "derive in one step"
- $\overset{*}{\Rightarrow}$ means "derive in zero or more steps"
- $\overset{+}{\Rightarrow}$ means "derive in one or more steps"

Some properties of derivations

- Reflexivity:

  $\alpha \overset{*}{\Rightarrow} \alpha$ for all strings α

- Transitivity:
  If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$

Languages and grammars

- L(G) is the language generated by grammar G
- The alphabet Σ for L(G) is the set of terminal symbols in G
- A string of terminal symbols $w \in \Sigma^*$ is in L(G) iff we can derive w from S, the start symbol of G
- That is, $w \in$ L(G) if and only if w consists of terminals only, and

$$S \overset{+}{\Rightarrow} w$$

Sentences, sentential forms, equivalence

- Two grammars are **equivalent** if they generate the same language
- If $w \in L(G)$, we say that w is a **sentence** of L(G)
- If $S \stackrel{+}{\Rightarrow} \alpha$, where α may contain non-terminals, then α is a **sentential form** of *G*
- A sentence is the special case of a sentential form with no nonterminals

Derivations in action

- The string **–(id + id)** is a sentence of the grammar below…
- It contains only terminals, and it can be derived from the start symbol:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E\ OP\ E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

- All the strings in the above derivation are ***sentential forms***

$$E \rightarrow E\ OP\ E \mid (\ E\ ) \mid -E \mid \textbf{id}$$

$$OP \rightarrow + \mid - \mid * \mid / \mid \wedge$$

Ordering of derivations

- In a derivation, you can choose which nonterminal to replace at each step
- The above derivation was a **leftmost derivation** because it replaced the leftmost nonterminal at each step
- Here is a **rightmost derivation**:

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E\ OP\ E) \Rightarrow$
$-(E\ OP\ \mathrm{id}) \Rightarrow -(E + \mathrm{id}) \Rightarrow -(\mathrm{id} + \mathrm{id})$

Context-freedom

- What makes a grammar context-free?
- The left-hand-side of every production consists of exactly one symbol, which is a nonterminal symbol
- This means that each production applies no matter what the context of its left-hand-side is
- A language that can be produced by a context-free grammar (CFG) is called a **context-free language** (CFL)
- Not all languages are context-free, but most programming languages are (or nearly so)

Generating and parsing languages

- We may be interested in how to generate strings in a language given a grammar for the language
- But in parsing, we are interested in going the other way: given a string and a grammar,
  - Is the string in the grammar's language, and if so...
  - How could it have been generated, i.e. what is its syntactic structure
- We will consider several different kinds of context free parsers

Different types of parsers

- ***Universal*** parsers: these algorithms can parse any context-free language, but they are too slow to be used in practice for parsing programming languages
  - Cocke-Younger-Kasimi , Earley
- ***Top down*** parsers: build the parse tree starting from the root and work down to the leaves
  - Predictive recursive descent parsing, LL parsing
  - We construct these manually or automatically
- ***Bottom up*** parsers: build the parse tree starting from the leaves and work up to the root
  - Shift-reduce parsing, LR parsing
  - Can only be constructed automatically
- Top down and bottom up parsers cannot recognize all context-free languages, but the limitations are reasonable for parsing typical programming languages

Parse trees

- We may view a parse tree as a representation of a derivation
- The root of a parse tree is the start symbol
- Any internal node of a parse tree is a nonterminal symbol; the children of an internal node are the symbols on the right-hand-side of a production for their parent symbol
- Sentential forms have parse trees whose leaves contain terminals and non-terminals
- What can we say about the leaves of a parse tree for a sentence in the grammar's language?

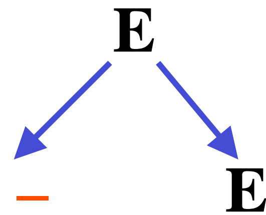
Deriving a parse tree, top-down

- Consider **– (id + id)**

E

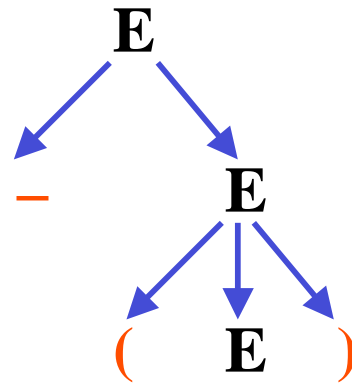
Deriving a parse tree

– (id + id)



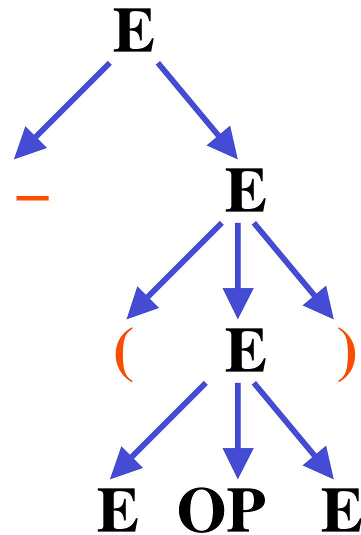
Deriving a parse tree

– (id + id)



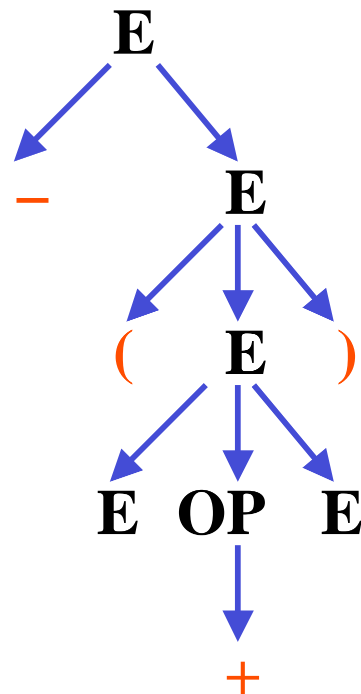
Deriving a parse tree

– (id + id)



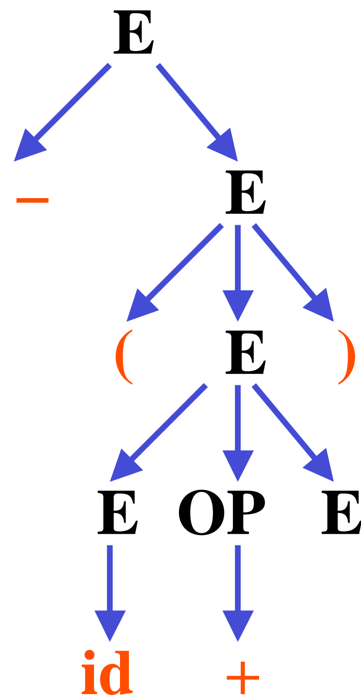
Deriving a parse tree

– (id + id)



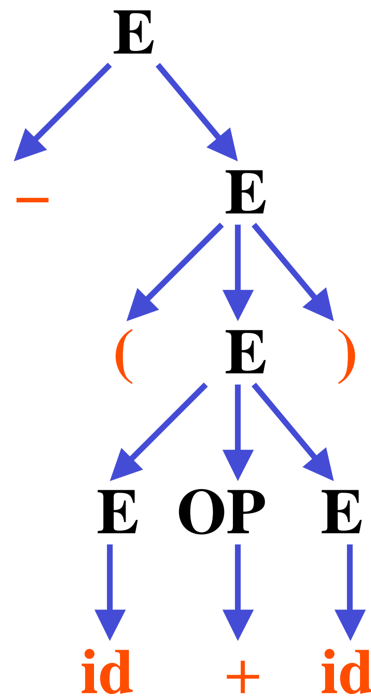
Building a parse tree

- Parse: **–(id + id)**



Building a parse tree

- Parse: **–(id + id)** ... done.



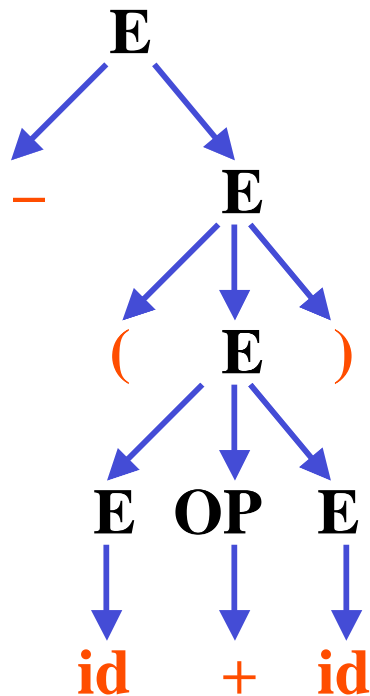
Parse trees and abstract syntax trees

- A parse tree provides the complete syntactic structure of a sentence, with internal nodes labeled with nonterminals
- The parse tree contains all the details: it is sometimes called the *concrete syntax tree*
- But often we suppress some details of the derivation process, e.g. a tree in which internal nodes are operators and their children are operands
- Such a tree is called an *abstract syntax tree* (AST) and is more suitable as input to the interpreter or code generator

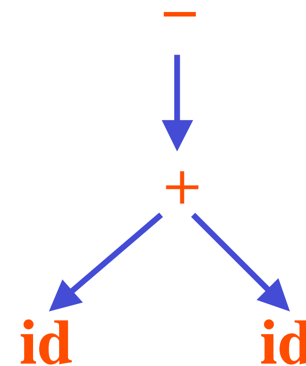
Parse tree vs AST

input: **–(id + id)**

parse tree



abstract syntax tree



An example for home

- Consider the parse tree for the following grammar

  E **→** ***T*** **|** ***T*** **+** ***E***
  T **→** ***F*** **|** ***F*** ***** ***T***
  F **→ tknInteger**

  tknInteger → [: number :]

- Derive the AST