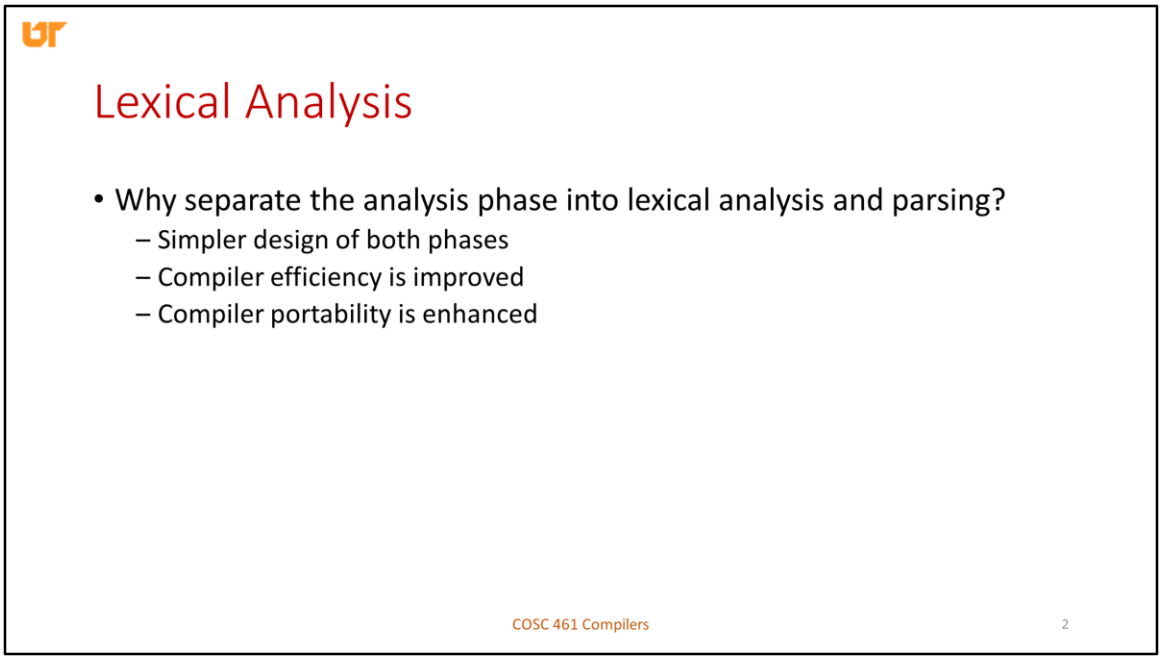
Concepts Introduced in Chapter 3

- Lexical Analysis
- Regular Expressions (RE)
- Lex
- Nondeterministic Finite Automata (NFA)
- Converting an RE to an NFA
- Deterministic Finite Automata (DFA)
- Converting an NFA to a DFA
- Minimizing a DFA

Chapter 3 is our first deep dive into one phase of the compiler. In this chapter, we'll be focused on how to construct a lexical analyzer.

There will be a lot of overlap with topics that might be discussed on a class on computer theory. In particular, we'll be looking at regular expressions and automata – which I actually think are a lot of fun.

We'll go over each of these topics in detail – so you do not have to have taken COSC 312 to understand what's going on. In particular, we'll be looking at:



This slide asks, why might we want to separate the analysis in our compiler into phases that do lexical analysis and parsing?

There are a number of reasons.

Design:
The most important reason is that it makes the design of both phases simpler. Separating these two tasks is likely to make at least one or both of the phases simpler. For instance, a parser that had to deal with whitespace and comments as syntactic units would be considerably more complex. With this design, we can abstract away operations (like skipping whitespace) that may have to be performed repeatedly.

Efficiency:
Next, it's more efficient. Lexical analysis is defined with a regular expression, not a context free grammar. Since this is a less powerful construct, we can write a more efficient algorithm.

Portability:

And, it's also more portable. We can write a lexical analyzer and use it with different parsers. Also, any input-device anomalies can be restricted to the lexical analyzer, and we can write our parser in a more general way.

Lexical Analysis Terms

- A ***token*** is a group of characters having a collective meaning (e.g. id).
- A ***lexeme*** is an actual character sequence forming a specific instance of a token (e.g. `num`).
- A ***pattern*** is the rule describing how a particular token can be formed (e.g. [A-Za-z_][A-Za-z_0-9]*).
- Characters between tokens are called ***whitespace*** (e.g.blanks, tabs, newlines, comments).
- A **lexical analyzer** reads input characters and produces a sequence of tokens as output.

Here are some terms that I'll use throughout our discussion of chapter 3.

A token …

For example, this could be a keyword or some symbol to denote identifiers.

A lexeme …

A pattern …

Whitespace is …

And a lexical analyzer is a tool or algorithm that reads input characters and produces a sequence of tokens as output

Interaction of Lexical Analysis and Parsing

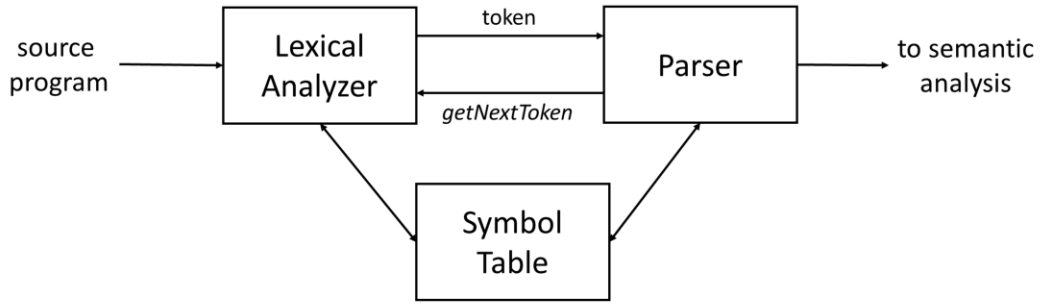


Figure 3.1: Interactions between the lexical analyzer and the parser

Remember, the main task of the lexical analyzer is to read the input source program and produce a sequence of tokens as output. The stream of tokens is sent to the parser for syntax analysis.

So, this figure shows how the lexical analyzer interacts with the parser and symbol table. Commonly, the interaction between the lexical analyzer and parser is implemented by having the parser call the LA directly. That's shown with the getNextToken arrow here.

The LA will also insert and read tokens from the symbol table.

Examples of Tokens

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i,f`	`if`
else	characters `e,l,s,e`	`else`
comparison	< or > or <= or >= or == or !=	`<=, !=`
id	letter followed by letters and digits	`pi, score, D2`
number	any numeric constant	`3.14159, 0, 6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Figure 3.2: Examples of tokens

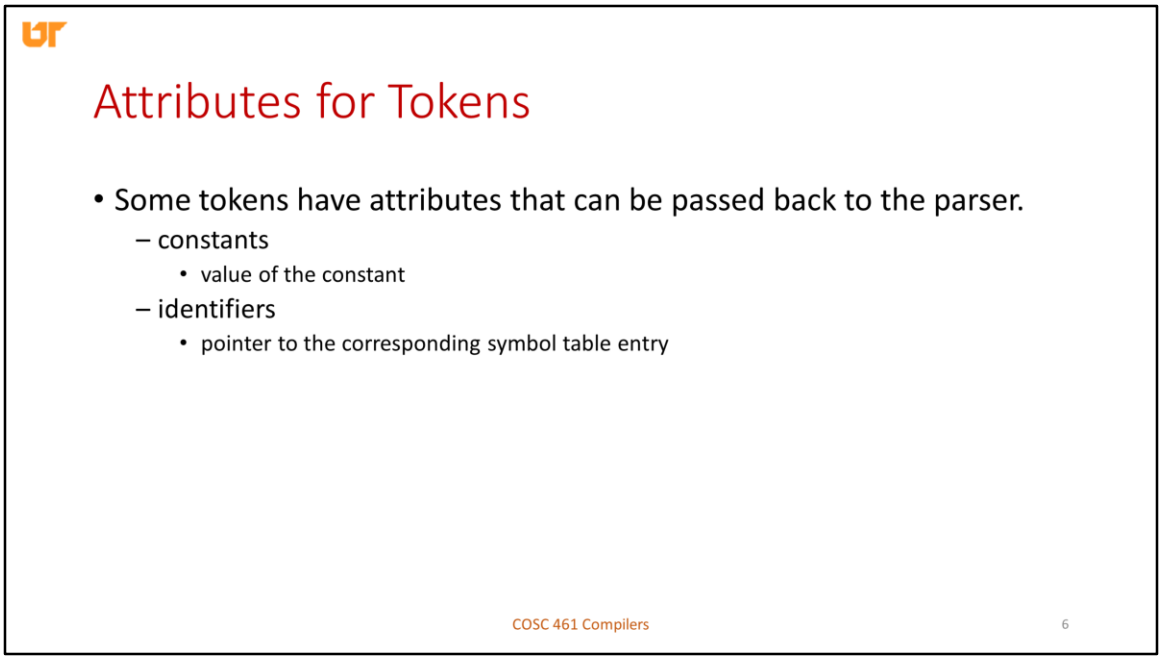
This table shows some typical tokens, with an informal description, and some sample lexemes that might be associated with each token.

As an example, what are the tokens in this statement?

printf("total = %d\n", score);

printf and score are lexemes matching for the token id, "total = %d\n" is a lexeme matching literal.

As we'll see, tokens are almost exclusively made up of: keywords, operators, identifiers, constants (including numerical and string literals), and punctuation symbols.

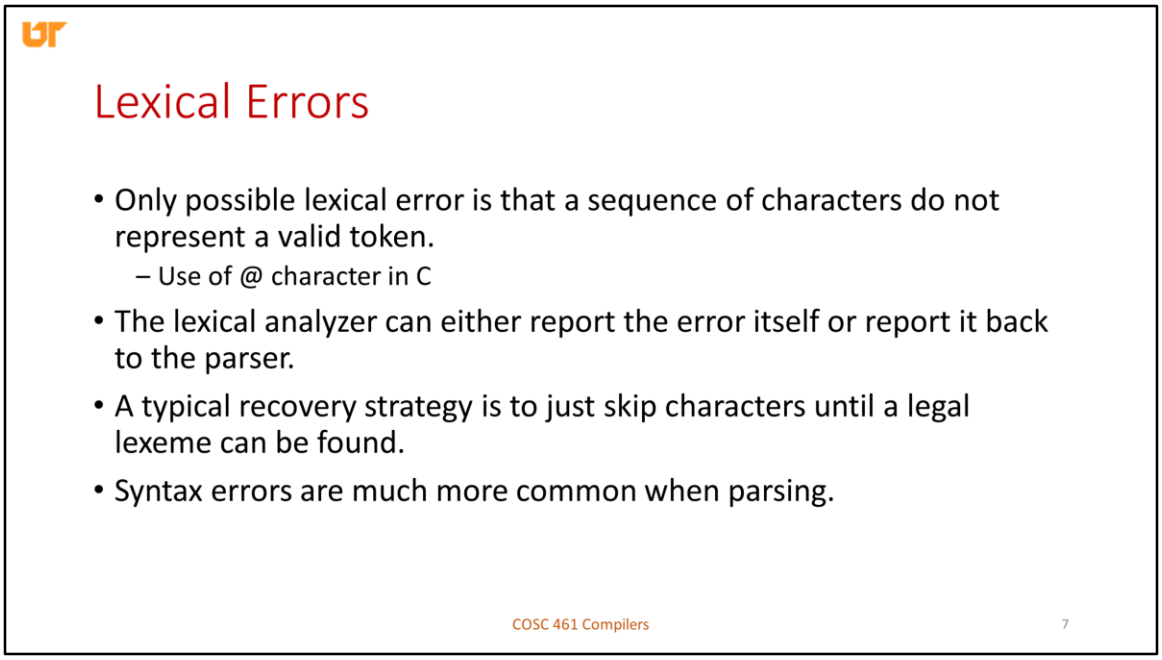


As I had mentioned earlier, some tokens have attributes that can be passed back to the parser.

In particular, when more than one lexeme can match a particular pattern, the subsequent compiler phases need additional information about the particular lexeme that matched.

For instance, if we have a lexeme that matches constants, the LA might associate an attribute that specifies the value of the constant with each matching token.
Or, with identifiers, we might have an attribute that is a pointer to the corresponding symbol table entry for the identifier.

We could have the parser itself create tokens and attributes, but then we would need to pass the actual lexeme to the parser. Later in the course, we'll see how lexical analysis tools, such as Lex, allow us allows us to pass attributes to the parser.



Errors in lexical analyzers do not happen very often.

Really, the only possible lexical error is that a sequence of characters do not represent a valid token.

So, for instance, if we use the @ symbol in C, identifiers cannot start with @, so this would be an example of a lexical error.

The LA can either report the error itself or report it back to the parser, but it needs a recovery strategy if it does find an error.

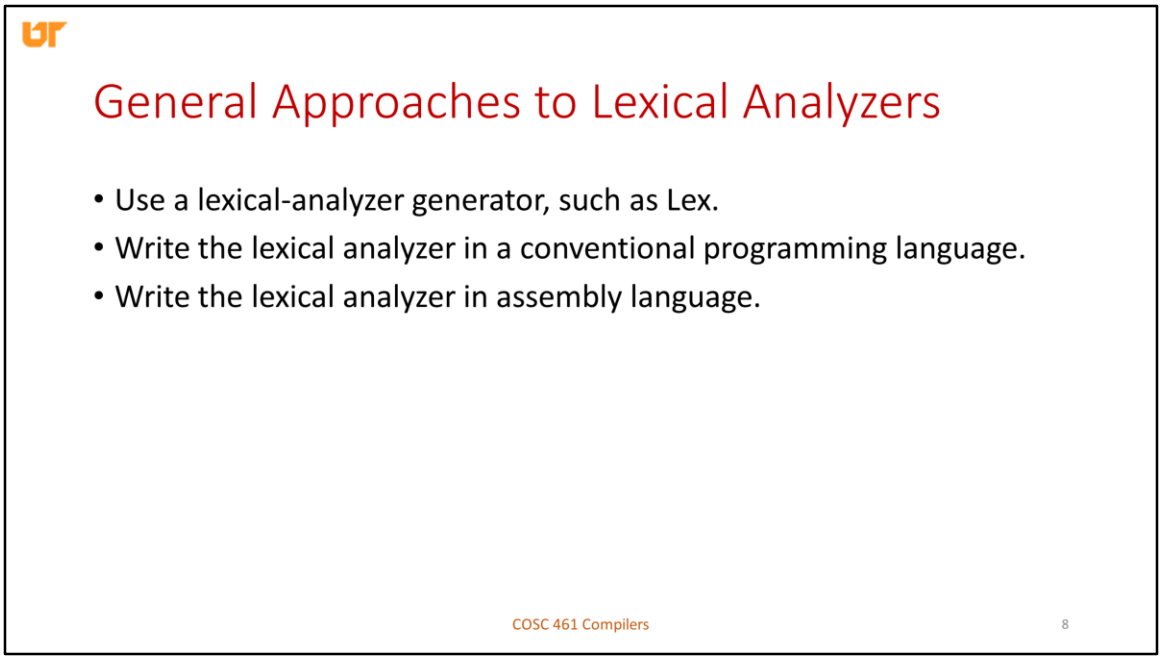
A typical recovery strategy is to just skip characters until a legal lexeme can be found.

Why might we want an error recovery strategy if we find an error while scanning the code?

A. It's useful to detect more than one error if something has gone wrong. We can proceed to detect further errors in the program and not exit after the first error.

As we'll see, syntax errors are much more common during parsing than errors during

LA.



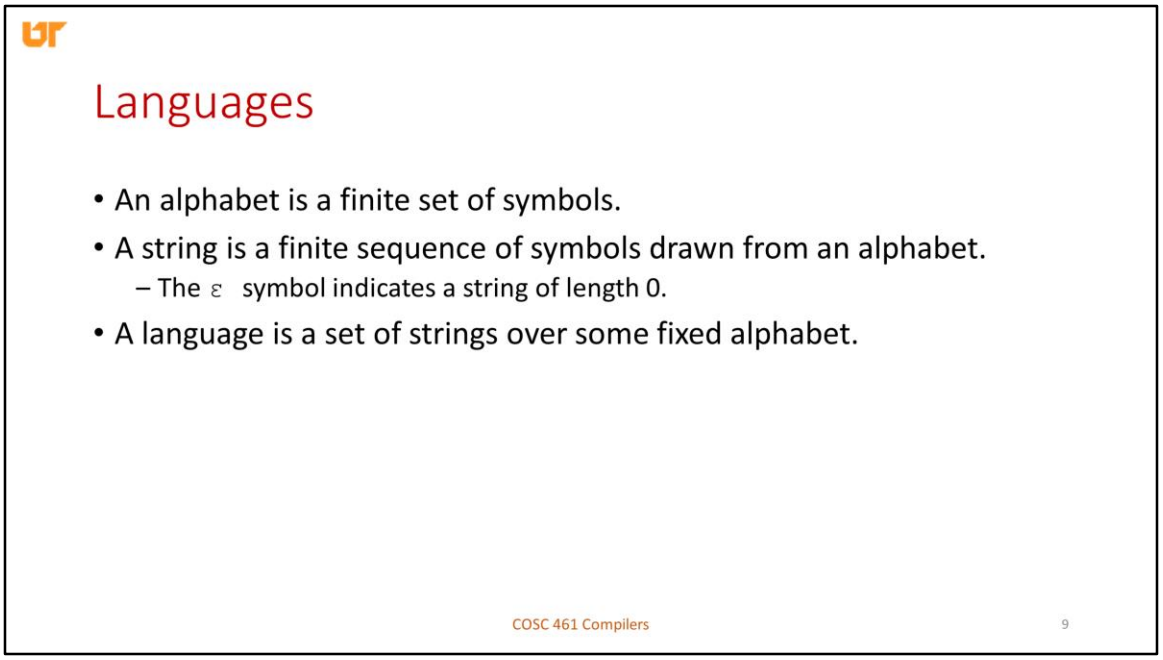
There are three sort-of general approaches we can use to write a lexical analyzer.

This list is shown in increasing order of difficulty, and unfortunately in efficiency. So, the easiest method is also the least efficient.

We can use LA generator tool, such as Lex, which we'll see here soon.

We could, write the LA in a conventional programming language, such as C. LA's are often written in a conventional programming language since they are not that complex and can be made more efficient. We've already seen an example of this in the lecture for the previous chapter.

Or we could write the LA in assembly language. I don't think this is that common anymore because there's not many applications where the potential performance benefits would be worth it.



Now, I want to discuss some of the theory behind how lexical analyzers are actually implemented.

We'll start by discussing terminology for formal languages, and then we'll get into regular expressions. Then, we'll discuss how we go from a regular expression to a lexical analyzer.

So, first, let's talk about what a language is.
An alphabet for some language is just a set of finite symbols.
A string is a finite sequence of symbols drawn from the alphabet.
The epsilon symbol indicates a string of length 0.
And then a language is a set of strings over some fixed alphabet.

In class, when I say language, I'll mostly be referring to some programming language, like C. But this definition could apply to other types of languages. Like, the alphabet for the English language includes the characters A-Z.

Terms for Parts of a String

- A *prefix* of string *s* is any string obtained by removing zero or more symbols from the end of *s*.
  - `ban`, `banana`, and ε are prefixes of `banana`
- A *suffix* of *s* is any string obtained by removing zero or more symbols from the beginning of *s*.
  - `nana`, `banana`, and ε are suffixes of `banana`
- A *substring* of *s* is obtained by deleting any prefix and any suffix from *s*.
  - `banana`, `nan`, and ε are substrings of `banana`
- The *proper* prefixes, suffixes, and substrings of a string *s* are those prefixes, suffixes, and substrings, respectively, of *s* that are not ε or equal to *s* itself.
- A *subsequence* of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s*.
  - `baan` is a subsequence of `banana`

from table on p. 119 in Aho, et. al.

Now, I want to discuss some terms having to do with strings. We will be discussing strings a lot since lexical analyzers will determine how to break strings into tokens.

A prefix of a string is any string obtained by removing zero or more symbols from the end of the string.

So, for instance, if we have the string banana, then ban and banana are prefixes of string. Epsilon is also a prefix which is formed by removing all the symbols from the end of banana. For lexical analyzers, prefix is probably the most important term to know here.

A suffix is just the opposite of a prefix. It is formed by removing zero or more symbols from the beginning of the string.

So, with banana …

A substring of a string is obtained by deleting any prefix and any suffix from s

So, with banana, banana, nan, and the empty string are substrings of s. Notice that

this only removes prefixes and suffixes. The characters that are left are still have to be consecutive.

The proper prefixes, suffixes, and …

And a subsequence of a string is any string formed by deleting zero or more not necessarily consecutive portions of s. So, in this case, a subsequence can remove characters that are not consecutive. So, for instance baan might be a subsequence of banana if we remove the first 'n' and the last 'a'.

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
Concatenation of L and M	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions and operations on languages

The most important operations in lexical analysis are shown in this table with set notation.

First is union of two languages. This says the string s is in the language L U M if s is in L or s is in M.

Next, we have concatenation of two languages. The concatenation of L and M is defined as all the strings of the form st where s is in L and t is in M. So, we just concatenate all the strings from both languages to get the new language.

If we think of concatenation as a product, then we can also define exponentiation. So, L^0 = empty, L^1 = L, L^2 = LL, L^3 = LLL, and so on.

The Kleene closure of a language L is denoted as L^* and it's the set of strings you get by concatenating L zero or more times. So, here's the way to think about Kleene closure. If you're language is just a set containing a single string:

{a}

Then L^0 = empty, L^1 = {a}, L^2 = {aa}, L^3 = {aaa}, …

So, L^* is just the set of strings that have any number of a's.

And we have the Positive closure of L, which is denoted as L^+. This is just like Kleene closure, except we do not include the empty string.

We will see that these operations are useful when specifying the patterns from which a lexical analyzer can be generated.

Example:
Let's do some examples. So, we define L as the set of letters {A,B, … , Z, a, b, … z} and M as the set of digits {0,1,…9}

Q. What is L U D?
A. Set of letters and digits

Q. What is LD?
A. Set of strings of length 2, each consisting of one letter followed by one digit

Q. What is L^4?
A. Set of all 4-letter strings

Q. What is L^*?
A. Set of all strings of letters, including the empty string

Q. L(L U D)*
A. Set of all strings of letters and digits beginning with a letter

Regular Expressions

Given an alphabet Σ

1. ε is a regular expression that denotes {ε}, the set containing the empty string.
2. For each $a \in \Sigma$, **a** is a regular expression denoting {a}, the set containing the string a.
3. If r and s are regular expressions denoting the languages L(r) and L(s), then:
   a) $(r)|(s)$ denotes $L(r) \cup L(s)$
   b) $(r)(s)$ denotes $L(r)L(s)$
   c) $(r)^*$ denotes $(L(r))^*$

Now, let's talk about regular expressions. A regular expression is just a way for describing languages that can be built from these operators.

Here are the rules that define regular expressions over some alphabet sigma. Note, in this slide, we're using the Greek symbol sigma to denote our alphabet.

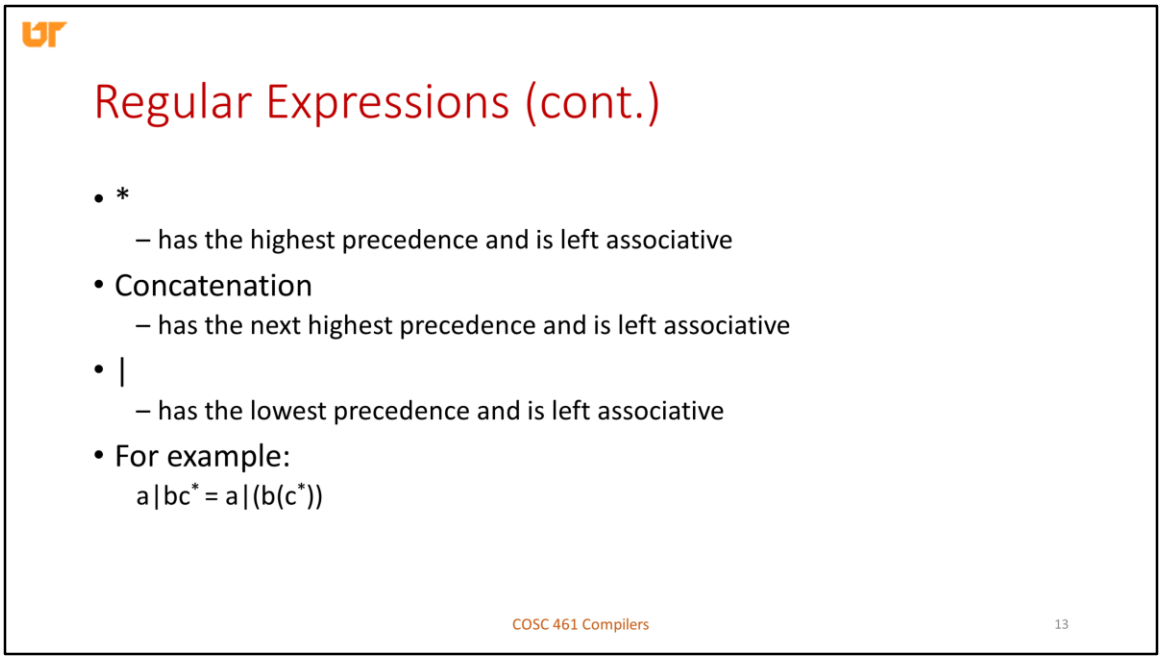
Q. Before going forward, someone tell me what an alphabet is?
A. An alphabet is the set of symbols you can use for a language

Q. So, what would be the alphabet for a language of binary words?
A. {0,1}

So, given an alphabet sigma,
1. The epsilon symbol is a regular expression that denotes {epsilon}, the language whose sole member is the empty string.
2. For each symbol a in the alphabet sigma, a is a regular expression denoting {a}, the language containing the string a
3. If r and s are regular expressions denoting the languages L(r) and L(s), then

r | s denotes the union of L(r) and L(s)
s followed by s denotes the concatenation of L(r) and L(s)
r star denotes the Kleene closure of L(r)

Basically, this is just another notation for using the language operators we just learned about.



There are some more rules for regular expressions I want to go over before doing some examples.

The Kleene closure or "star" has the highest precedence and is left associative

Concatenation has the next highest precedence and is left associative

And union or "pipe" or sometimes I'll just say "or" has the lowest precedence and is left associative

So, in this example, we have:

$a|bc^* = a|(b(c^*))$

Examples of Regular Expressions

Let Σ = {a, b}

a|b	=> {a, b}
(a|b)(a|b)	=> {aa, ab, ba, bb}
a^*	=> {ε, a, aa, aaa, … }
$(a\|b)^*$	=> all strings containing zero or more instances of a's and b's
$a\|a^*b$	=> {a, b, ab, aab, aaab, … }

So, here's some examples of regular expressions over the alphabet {a,b}

Go through each one …

a|b => {a,b}
(a|b)(a|b) => {aa,ab,ba,bb}
a^*
$(a|b)^*$ -- all strings containing zero or more instances of a's and b's

What would this one be?

$a|b^*$ (The letter a or any number of b's)

$a|a^*b$ – a single a, or any number of a's followed by a single b.

Algebraic Laws for Regular Expressions

Law	Description
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|st$	Concatenation distributes over |
$\varepsilon r = r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	ε is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

Finally, here are some algebraic laws for regular expressions

Union is commutative and associative.

Concatenation is associative, but notice that it's not commutative. If you reorder regular expressions with concatenation, you will get a different result.

We also see there is a distributive law:

r(s|t) = rs | rt

The empty regular expression or epsilon is the identity for concatenation. That should make sense because if you concatenate nothing with something – you just get the original thing back.

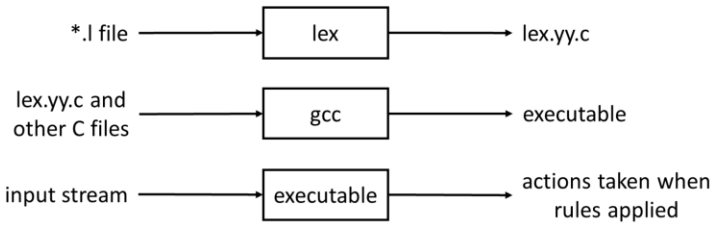
I guess it's convenient to have a law for the empty string being guaranteed in a closure – I am not sure why that one is listed.

Idempotent means that you can apply something more than once and receive the

same result back. So, for example, multiplying by 1 is idempotent because 1 x 1 x 1 = 1 no matter how many times you do it.

Lex – A Lexical Analyzer Generator

- Can link with a lex library to get a main routine.
- Can use as a function called yylex().
- Easy to interface with yacc.



So, that's it for regular expressions for now. We'll come back to those in a little bit. Now, I want to turn to lex.

Lex is a lexical analyzer generator. It is a tool that allows you to automatically produce a lexical analyzer from some specification.

You write your lex specification in a *.l file.
Running the lex tool will produce a file called lex.yy.c. In this file, it will define a function called yylex(). This function is your lexical analyzer.

We will see that yylex() is easy to integrate with yacc.

Lex – A Lexical Analyzer Generator (cont.)

- Lex source

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

- Definitions
  declarations of variables, constants, and regular definitions
- Rules
  regular expression action
- Regular Expressions
  operators "\ [ ] ^ -? . * + | ( ) $ / { }
  actions C code

A lex source file consists of three parts – and each part is separated using two percent signs.

The first part is definitions. Sometimes we refer to variables or constants in the actions that we've defined – that is the fragments in C code. These declarations can be placed in the definitions part.
We might also put regular definitions in this section. Regular definitions are ways to define a shorthand for a regular expression you want to refer to later (like a #define in C). We'll see some examples of these a little later.

Your rules in the Lex file are just regular expressions and their associated actions.

The regular expressions use a number of operators that I'll describe in just a moment, and the actions are defined as C code.

The final section is the user subroutines section. This section is only if you want to use the lexical analyzer as a standalone program. So, the final section is often omitted.

The first section can be omitted as well. So, you might just have a collection of regular expressions and their associated actions for your lex program.

The smallest lex program is just %%, and all that will do is take the input from stdin and copy it back out to stdout.

Lex Regular Expression Operators

- "s" string *s* literally
- \c character *c* literally (used when *c* would normally be used as a lex operator)
- [s] for defining *s* as a character class
- ^ to indicate the beginning of a line
- [^s] means to match characters *not* in the *s* character class
- [a-b] used for defining a range of characters (*a* to *b*) in a character class
- r? means that *r* is optional

This slide shows some of the operators lex uses to define regular expressions.

"" are for defining a literal.
\ is used to escape a character when that character might be used as a lex operator.
[] are used for defining a character class. A character class is a way of saying match any one of the characters in the class.
^ is used to indicate the beginning of a line.
[^s] means to match characters that are not in the s character class
[-] is used for defining a range of characters in a character class.
? means that the preceding character is optional.

Lex Regular Expression Operators (cont.)

- . means any character but a newline
- r^* means zero or more occurrences of *r*
- r^+ means one or more occurrences of *r*
- $r_1|r_2$ r_1 or r_2
- (r) r (used for grouping)
- $ means the end of the line
- r_1/r_2 means r_1 when followed by r_2
- r{m,n} means m to n occurrences of *r*

. means any character but a newline.
r* means zero or more occurrences.
r+ means one or more occurrences.
r1|r2 mean match r1 or r2
() are used for grouping.
$ means the end of the line
r1/r2 means r1 when followed by r2
r{m,n} means m to n occurrences of r

Example Regular Expressions in Lex

• a*	zero or more *a*'s
• a+	one or more *a*'s
• [abc]	*a* or *b* or *c*
• [a-z]	lower case letter
• [a-zA-Z]	any letter
• [^a-zA-Z]	any character that is not a letter
• a.b	*a* followed by any character followed by *b*
• ab|cd	*ab* or *cd*
• a(b|c)d	*abd* or *acd*
• ^B	*B* at the beginning of a line
• E$	*E* at the end of a line

see Figure 3.8 (p. 127) in Aho

These are examples of some of the commonly used operations.

Example Regular Expressions in Lex

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a[1,5]`
r_1r_2	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: Lex regular expressions

These are in your book (Figure 3.8).

Lex (cont.)

- Actions

  Actions are C source fragments. If it is compound or takes more than one line, then it should be enclosed in braces.

- Example Rules

```
[a-z]+          printf("found word\n");
[A-Z][a-z]*     { printf("found capitalized word\n");
                  printf(" %s\n", yytext);
                }
```

- Definitions

```
name            translation
```

- Example Definition

```
digits          [0-9]
```


Draw:

[definitions]
%%
[rules]
%%
[user subroutines]

Indicate we are talking about rules.

An action in a rule is a C source fragment that is executed when the rule is taken. If the action has more than one statement or takes more than one line, then it should be enclosed in braces.

[Read rules, ask students what each rule will do.]

We have a few special variables that Lex defines for your convenience.
yytext contains the character string of the lexeme for token matching the pattern in

the RE.
And yyleng contains the number of characters (non-null) in yytext.

Definitions are like #defines in C.
They are used for defining abstractions that make reading and writing the lex specification easier.

Example Lex Program

```
digits          [0-9]
ltr             [a-zA-Z]
alpha           [a-zA-Z0-9]
%%

[-+]{digits}+           |
{digits}+               printf("number: %s\n", yytext);
{ltr}(_|{alpha})*       printf("identifier: %s\n", yytext);
"'"."'"                 printf("character: %s\n", yytext);
.                       printf("?: %s\n", yytext);
```

Prefers longest match and earlier of equals.

Here is an example lex program

We could have defined alpha in terms of the other definitions.

alpha ({ltr}|{digits})

A rule consists of a regular expression and an action.

The regular expression and the action are separated by one or more blanks, tabs.

A newline means that it is the end of a rule, unless the action is enclosed in curly braces.

Note use of | action, which means that the action for this rule is the action for the next rule (can read as "or")

[-+] means either - or +.
Note that we could have shortened the first rule by putting ? after the [-+].

The default action if there is no match on a character is to just print the character.

So this simple lex program will list numbers, identifiers, character literals, and other character symbols one per line.

If two rules both match then it will apply the action for the rule that consumes the greatest number of input characters.

If two rules consume the same number of characters, then it will apply the earlier of the two rules.

Given the rules:

```
abc     { printf("found abc\textbackslash n");
[a-z]+ { printf("found identifier\textbackslash n"); }
```

So if input was:

abc

It would use the first rule.

abcd

Would use the second rule.

If I switched the order of the rules, then we would use [a-z]+ and the abc rule would never be used.

Say we have the rule:

```
'.*'
```

and we have the input:

```
'compilers' is an interesting 'subject'
```

then lex would match the entire line since . says any character but a
newline and the entire line is a longer match than 'compilers'.
[Enter examples in .l file and show demo in class.]

Tokens, Patterns, and Attributes

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| ε
expr → *term* **relop** *term*
| *term*
term → **id**
| **number**

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

This table shows all the tokens for an example grammar.

The left column shows the lexeme or collection of lexemes that match a particular token, the middle is the token name, and the right column shows the attribute value associated with each token.

The ws in the first row stands for whitespace.

Example Lex Program

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}        {/* no action and no return */}
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);}

%%

install_id() {
    /* procedure to install the lexeme, whose
    first character is pointed to by yytext and
    whose length is yyleng, into the symbol table
    and return a pointer thereto */
}

install_num() {
    /* similar procedure to install a lexeme that
    is a number */
}
```

Fig. 3.23. Lex program for the tokens of Fig. 3.12.

Here is the example lex program for that grammar.

At the top, you see we can define constants in between the %{ and %}. These constants can be returned to represent classes of tokens.

Note that regular definitions are referred to inside curly braces. Also, notice that regular definitions can build upon one another.

Whitespace is a sequence of one or more blanks, tabs, and newlines.
We have letters and digits – pretty self-explanatory.

An 'id' is made up of a letter followed by any number of letters and digits.

For numbers, notice the use of '?' in the declarations to make the decimal point and fractional digits optional. '?' means zero or one occurrence of the preceding item.

Here are the rules:

Note there is no action when whitespace is encountered. This means that the lexical

analyzer will just skip over it.

yylval is a global variable whose value is also available to the parser. The purpose is to hold the lexical value since the return value can only return a code for the token class.

install_id() puts an identifier into the symbol table. For this function, we need to make use of the global variable yytext yytext to get the actual lexeme associated with the identifier.

Likewise, install_num() will use yytext to install the lexeme associated with the number.

Nondeterministic Finite Automata

- A nondeterministic finite automaton (NFA) consists of
  - A set of states *S*
  - A set of input symbols Σ (the input alphabet)
  - A *transition function* move that maps state-symbol pairs to sets of states
  - A state s_0 that is distinguished as the *start state* or *initial state*
  - A set of states, *F*, distinguished as *accepting* or *final states*

OK – that's it for our lex examples. Now, we're going to look at how lex turns its input program into a lexical analyzer.

Internally, Lex needs to build a machine or an algorithm to recognize the RE. Internally, it uses a formalism known as a nondeterministic finite automaton or NFA.

An NFA is a machine that can recognize the language specified by a regular expression.

NFA's consist of …

Operation of an Automaton

- An automaton operates by making a sequence of moves.
- A move is determined by a current state and the symbol under the read head.
- A move is a change of state and may advance the read head.

So, here's an example finite automaton:

(0) -- a --> (1) – b --> ((2)) a)

The NFA works by making a series of moves

And the move it makes is determined by the current state and the symbol under the read head.

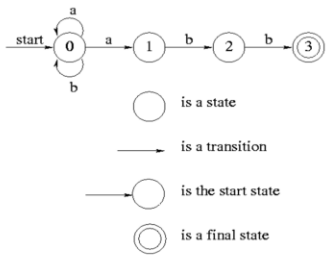
A move is just a change of state and might advance the read head on the input string.

Try with strings:
abaa (accept)
aab (reject)
abab (reject)

If we ever are in a non-final state from which there are no transitions on a given input symbol, then the string is not in the language accepted by the automata.

Representations of Automata

- Example: (a|b)*abb



Transition Diagram

State	input symbol a	b
0	{0,1}	{0}
1	–	{2}
2	–	{3}
3	–	–

Transition Table

Here's another example of an NFA. This one accepts the RE (a|b)*abb

This slide shows two different ways we can represent an NFA. The first is with a transition diagram, which just shows the states as circles and an arrow for the transitions. A double circle denotes a final or accepting state.

Another representation is the transition table. Each row in the table shows the set of states we might transition to on each input symbol.

Q. Look at this figure and tell me why this automata is called non-deterministic automata?
A. This automata is nondeterministic since 2 transitions from state 0 on input a.

A transition table is a convenient way to implement an automata in software.

Operations on NFA States

OPERATION	DESCRIPTION
ε-*closure*(s)	Set of NFA states reachable from NFA state s on ε-transitions alone.
ε-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ε-transitions alone; $= \cup_{s \text{ in } T} \varepsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Figure 3.31: Operations on NFA states

So, part of writing a lexical analyzer is being able to create an finite automaton that recognizes the language for a given regular expression. Then, you can simulate the automaton with the input to see if it matches the regular expression.

However, because an NFA often has a choice of multiple moves on an input symbol, or even a choice of making a move without consuming any input, simulating an NFA is often less straightforward than simulating a DFA. Thus, it is often important to convert an NFA to a DFA that accepts the same language.

For your next assignment, you will need to implement an algorithm that translates a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA).

This table shows some basic operations on the states in an NFA. You will have to understand these operations for your assignment.

In this table, s is a single NFA state and T is a set of NFA states.

An epsilon-transition in an NFA is just going from one state to another without consuming any input. To translate from an NFA to a DFA, you will need to deal with all

the epsilon-transitions in the automaton properly.

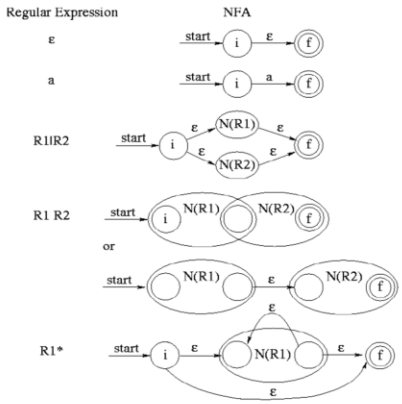
The epsilon-closure of a state is just the set of states that are reachable from s on epsilon-transitions alone. The epsilon closure from a given state always includes staying at the same state.

The second rule just defines epsilon-closure for a set of states. The epsilon-closure for a set of states is just the union of the epsilon-closure for all the states in the set.

And the third rule is move. Given a set of states and an input symbol, the move function returns the set of states to which there is a transition on the input symbol from state in the set of states.

These rules are the basic rules that allow movement from one state to another in an NFA.

Regular Expression to an NFA



OK – now let's talk about how to construct an NFA from a regular expression. This method is called Thompson's construction.

Given a regular expression R. we want to construct N(R), which is the NFA accepting L(R).

The algorithm works by breaking R down into its constituent subexpressions and building the NFA from a set of simple rules. These figures show how to construct an NFA given some simple regular expressions.

In each figure, state (i) is the start state of the NFA representing that regular expression.
And state ((f)) is a final state of the NFA representing that regular expression.

The -- epsilon --> are episilon transitions. This means that the NFA can make this transition without processing an input character.

Explain NFA for epsilon
Explain NFA for 'a'

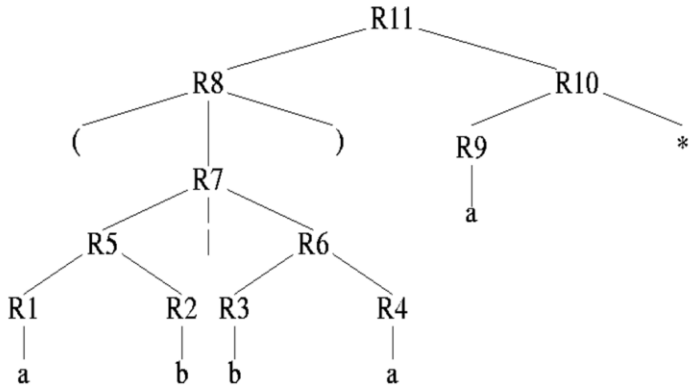
Explain NFA for R1|R2 – If we accept either R1 or R2, then we accept R1|R2

Explain R1R2 – For this one, the accepting state of R1 becomes the start state of R2. It might be easiest to use the second alternative for R1R2, especially when there is more than one accepting state of N(R1).

For R1*, we can accept the NFA for R1 an arbitrary number of times to accept R1*.

How could $R1^+$ be translated to an NFA?
A. Remove the bottom epsilon transition from $R1^*$

Decomposition of (ab|ba)a*



There was a little confusion last class about constructing this parse tree.

First of all, someone asked, well if all you have is finite automaton, you cannot parse the expression (ab|ba)a*. You need some way of matching the parantheses, and finite automaton cannot match arbitrarily nested patterns like parantheses. For this example, we first construct a parse tree using a parsing algorithm to make it easier to translate this expression into an NFA. Parsing this expression into a tree will use a separate algorithm – not related to finite automatons. We will cover parsing algorithms in the next chapter.

For now, you can use this procedure to construct your parse tree. Look at the expression and determine the order of operations in the expression. The operations that happen last will go highest in the tree. And the operations that happen first go lowest in the tree. So, we'll do a top-down parse because these are easier to construct by hand. This example in the slides uses a bottom-up parse – so the nodes will be labeled differently – but the tree will look the same.

So, from the expression, what will the last operation be? (union, concatenation, or star)

Concatenation of (ab|ba) and a*
Make a node with children as the two regular expressions we are concatenating (don't need to label the concat operator).
Now, we can choose whichever side we want to go down first. We'll go down the left side. We'll make a node for the parantheses. Now, what's the last operation that would be applied inside the parantheses (union)
K – so we make a node whose children are the union operator and the two regular expressions that we are unioning together.
Continue … finish off the left side.
Now, we go to the right of the top-level concat. We have a*, so we make a node with children as the * operator and the regular expression we are applying the * to.

OK – so let's try an example:

Say, we are given the regular expression:

(ab|ba)a*

We can parse the regular expression into this tree. (this tree is similar to the parse trees we saw in the last chapter). In this example, the parse tree has two operands for the (outer nodes) and one operator for the (middle node). We do not draw the concatenation operator. To make the construction easier, we can write the RE like this:

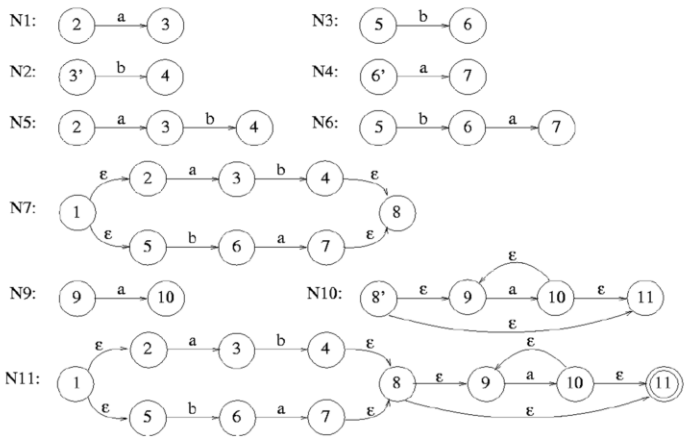
((ab)|(ba))(a*)

I won't go over how to construct this parse tree – but it's similar to what we saw earlier with arithmetic operations. So let's work our way up the parse tree.

R1 and R2 correspond to the leftmost a and b and R5 concatenates them. Similarly, R3 and R4 and R6 correspond to the ba expression. Then, we have the node R7 to do the union operation for these trees. We use another node for parantheses, and then we have R9 and R10 for the a* expression, and R11 ties the two parts ((ab)|(ba)) and (a*) together.

To construct the NFA, we start at the leaves and work our way up.

Decomposition of (ab|ba)a* (cont.)



OK – now we just construct NFA's for each node in the parse tree corresponding to the rules. So, for each Ri in the parse tree, we construct a corresponding NFA Ni.

For R1 and R2, we construct N1 and N2. And similarly, we construct N3 and N4.

Next, for R5, which concatenates R1 and R2, we construct N5 using the rule for concatenation. We construct N6 for R5 and R6 in a similar way.

For R7, we apply the rule for union (show the rule), and we get N7.

Then for R9 and R10, we create N9 and N10. N10 is the rule for the * operator.

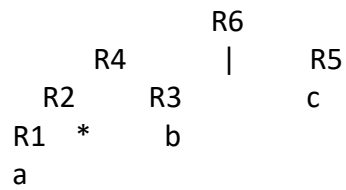
And finally, for R11, we are again concatenating two NFA's, so we concatenate N7 and N10, and we get N11. Notice, there is no N8, because the {} operators do not require any special action.

OK, let's try another example:

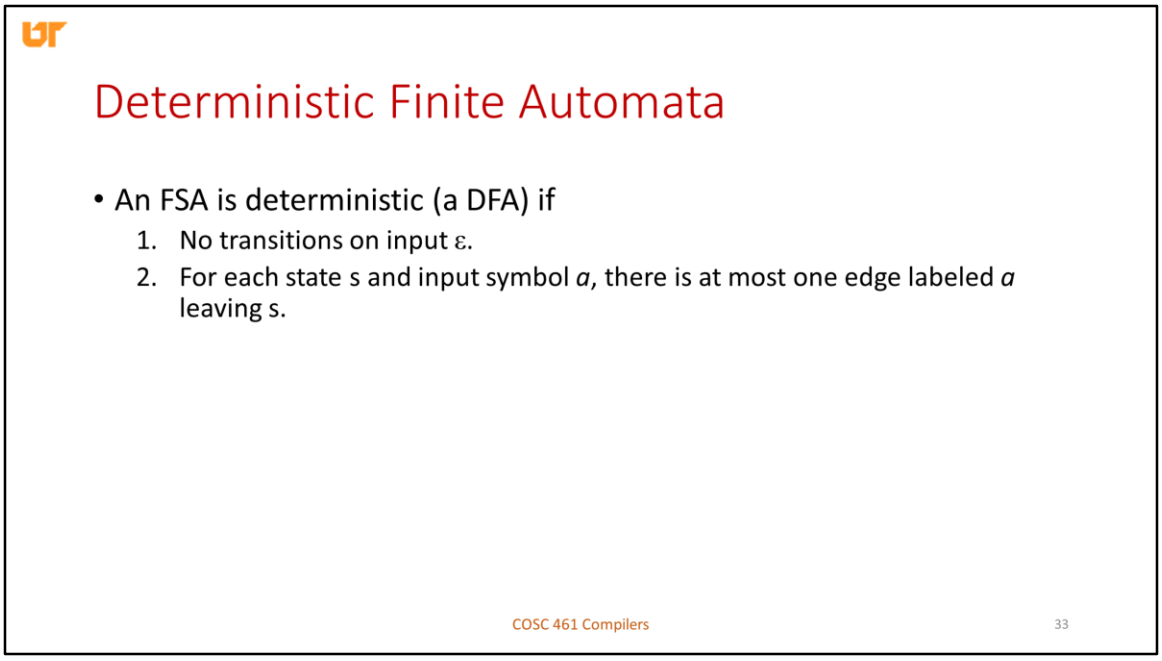
a*b|c

Q. How would this be parenthesized?
A. ((a*)b)|c

First, make a parse tree:



Then work through the example.



Now, let's talk about deterministic finite automata or DFA's. A DFA is just like an NFA, but with two exceptions:

1) No transitions on epsilon
2) For each state, at most one transition with each symbol

It's easier to use DFA's to implement a recognizer because one knows exactly which state to move to on the next input symbol.

Multiple paths (transitions) on a single input symbol would require backtracking.

So DFAs are easier to implement and faster but may require more states (space).

Operations on NFA States

OPERATION	DESCRIPTION
ε-*closure*(s)	Set of NFA states reachable from NFA state s on ε-transitions alone.
ε-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ε-transitions alone; $= \cup_{s \text{ in } T} \varepsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Figure 3.31: Operations on NFA states

Remember when we talked about closure? The closure operations can be used to convert an NFA to a DFA.

Remember that the closure operation always includes the current state.

Converting NFA's to DFA's

initially, ϵ-$closure(s_0)$ is the only state in *Dstates*, and it is unmarked;
while (there is an unmarked state T in *Dstates*) {
    mark T;
    for (each input symbol a) {
        $U = \epsilon$-$closure(move(T, a))$;
        if (U is not in *Dstates*)
            add U as an unmarked state to *Dstates*;
        $Dtran[T, a] = U$;
    }
}

Figure 3.32: The subset construction

OK – so here is the algorithm for converting an NFA to a DFA. You will implement this for your first assignment.

S0 is the start state in the NFA, and Dstates is the set of states in the DFA that we're constructing.

The algorithm starts by computing the epsilon-closure of s0 into Dstates.

Remember that (write out on board):
epsilon-closure(s) = (s) U { t | s --> t on an epsilon transition }

It works by iteratively adding states to the DFA by following transitions in the NFA. It marks states it has already seen so that it does not go into an infinite loop.

So, when it gets an unmarked state, the algorithm, goes through each input symbol, and computes the set of states that are reachable from that Dstate with the input symbol and epsilon transitions. Then, if that set of states is not already a state in Dstates, then it adds that set as a new DFA state to the DFA. It also updates the transition table to indicate that T goes to U on input symbol a.

The final states in the DFA are those that hold at least one final state in the NFA.

Computing ε-closure(T)

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);
                        push u onto stack;
                }
}
```

Figure 3.33: Computing ε-closure(T)

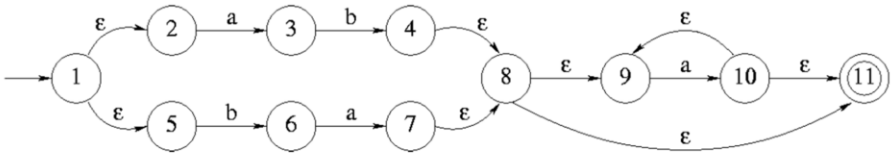
This is the algorithm for computing the epsilon-closure for a set of states T.

This is a simple stack-based algorithm. We initialize epsilon-closure(T) to T because remember that an epsilon transition from a state always includes the same state.

We put the states in T onto the stack, and then until the stack is empty, pop elements off the stack. When we pop a state off the stack called t, we cycle through all the states that are reachable from t on epsilon-transitions. If those states are not already in epsilon-closure(T), then we add that state to the epsilon-closure, and push it onto the stack.

Pretty straightforward – just uses a stack to traverse the NFA outward.

Example of Converting an NFA to a DFA



OK – now we're going to do an example where we convert the NFA to a DFA. We'll try with this example.

Draw this up on the board – then put the algorithm up on the screen. Go through the algorithm – and check your answer with the next slide.

Remember to draw a stack, the set of dstates in a transition table, and a structure noting which states have been marked.

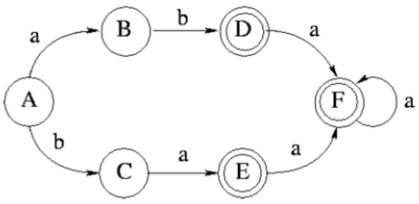
Leave algorithm on screen, copy transition diagram, and go through the example using the algorithm.

Example of Converting an NFA to a DFA

A = *ε-closure*({1}) = {1, 2, 5}
mark A
{1, 2, 5} --- a ---> {3}
B = *ε-closure*({3}) = {3}
{1, 2, 5} --- b ---> {6}
C = *ε-closure*({6}) = {6}
mark B
{3} --- b ---> {4}
D = *ε-closure*({4}) = {4, 8, 9, 11}
mark C
{6} --- a ---> {7}
E = *ε-closure*({7}) = {7, 8, 9, 11}
mark D
{4, 8, 9, 11} --- a ---> {10}
F = *ε-closure*({10}) = {9, 10, 11}
mark E
{7, 8, 9, 11} --- a ---> {10}
mark F
{9, 10, 11} --- a ---> {10}

Put up slide to see if done correctly.

Example of Converting an NFA to a DFA



Transition Diagram

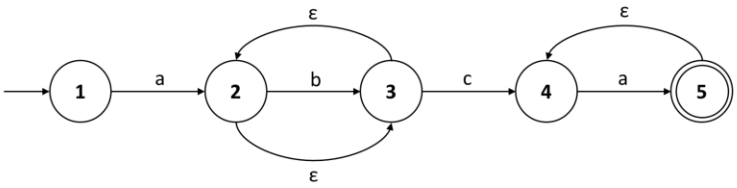
	a	b
A	B	C
B		D
C	E	
D	F	
E	F	
F	F	

Transition Table

Produce transition table and diagram and then put slide up to check if done correctly.

Any DFA state containing a final state from the NFA is a final state.

Another Example of NFA to DFA Conversion



A = epsilon-closure({1}) = {1}
mark A
{1} -a-> {2}
B = epsilon-closure({2}) = {2,3}
mark B
{2,3} -b-> {3}, epsilon-closure({3}) = {2,3} = B
{2,3} -c-> {4}
C = epsilon-closure({4}) = {4}
mark C
{4} -a-> {5}
D = epsilon-closure({5}) = {4,5}
mark D
{4,5} -a-> {5}
epsilon-closure({5}) = {4,5} = D

(A) -- a --> (B))b -- c --> (C) -- a --> (D))a

Minimizing a DFA

Given a DFA M

If some states of M ignore some inputs, add transitions to a "dead" state
Let P = { M's final states, M's non-final states }
Let P' = {}

loop:

For each group G ∈ P do
Partition G into subgroups so that s, t ∈ G are in the same subgroup iff each input *a* moves s and t to states of the same P-group
Put these new subgroups in P'
If P ≠ P'
assign P' to P
goto loop

These subgroups denote the states of the minimized DFA.

Remove any dead or unreachable states.

OK – now let's discuss an algorithm for minimizing a DFA.

Q. What is the one big disadvantage of DFA as compared to NFA?
A. Its size, or number of states are often greater than a corresponding NFA.

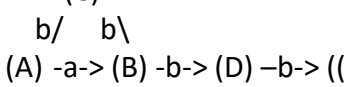
Q. Why would we want a smaller DFA?
A. If we can reduce the number of states in the DFA, the table used by the recognizer or lexical analyzer will be smaller.

The algorithm works by partitioning the states of the DFA into groups of states that cannot be distinguished. The book provides a formal definition of what it means for two states to be distinguishable:

"the string x distinguishes the state s from state t if exactly one of the states reached from s and t by following the path with x is an accepting state. The state s is distinguishable from state t if there is some string that distinguishes them." So, for example:

Example:

In the DFA:



The string bb distinguishes A from B because starting at A, if we follow bb, we wind up in B, but if we're in B and we do bb, we end in an accepting state.

So, the algorithm is essentially trying to prove that two states are distinguishable, then grouping them into different sets.

The algorithm starts by modifying the DFA M so that if some inputs are ignored, we add transitions on those inputs to a dead state.

Next, we initialize P and P'. P and P' are partitions of subgroups of states that have not yet been distinguished. So, if two states are in the same group, that means we do not know if they are distinguishable. If we put two states into different groups, then we know they can be distinguished.

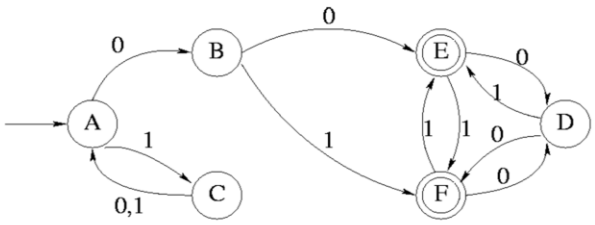
First, we know that the accepting states and non-accepting states are distinguishable, so we create two groups, one with all the final states, and one with all the non-final states, and put them into the partition P.

Then, for each group G in P, partition G into subgroups such that, for each pair of states s and t in G, s and t are in the same subgroup if and only if each input symbol moves s and t to states of the same P-group. Then, we assign the new partition as P'. If there were no changes on the last iteration, then we break out. Otherwise, we assign P to be P' and loop again.

So the general idea is keep states together if they have the same behavior on each input symbol.

When we're done, the subgroups denote the states of the minimized DFA. And we have to be sure to remove any dead or unreachable states.

Example of Minimizing a DFA



	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

So, let's do an example. Here is a DFA represented by both a transition table and a transition diagram.

Draw this up on the board, then go back to the algorithm and walk through it.

Note, this all states have transitions on 0 and 1, so we do not need to add a dead state.

a b
{ {A,B,C,D}, {E,F} }

A B C D E F
aa bb aa bb ab ab

a b c
{ {A, C}, {B, D}, {E, F} }

A C B D E F
ba aa cc cc bc bc

a b c d
{ {A}, {C}, {B, D}, {E, F} }

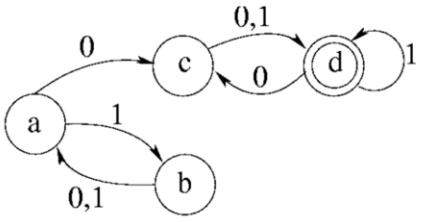
A	C	B	D	E	F
ca	aa	dd	dd	cd	cd

Example of Minimizing a DFA

a	b
{A, B, C, D}	{E, F}
A B C D	E F
aa bb aa bb	ab ab
a	b	c
---	---	---
{A, C}	{B, D}	{E, F}
A C B D	E F
ba aa cc cc	bc bc
a	b	c	d
---	---	---	---
{A}	{C}	{B, D}	{E, F}
_ _ B D	E F
dd dd	cd cd

[Don't show the slide, just keep the previous slide on the board and then minimize it.]
Final RE is (1(0|1))*0((0|1)1*0)*(0|1)1* if the students ask.

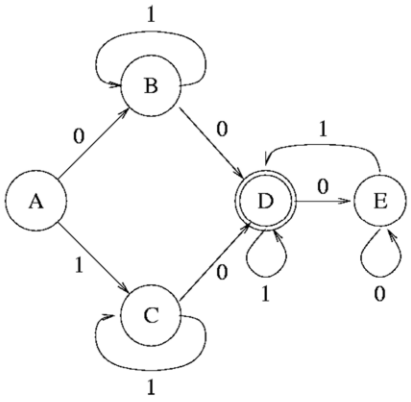
Example of Minimizing a DFA



	0	1
a	c	b
b	a	a
c	d	d
d	c	d

Put up slide to check results.
Note we now have 4 states instead of 6.
Ask if they could convert this to a regular expression.

Another Example of Minimizing a DFA



Have students work through the example.

a b
{ A, B, C, E } { D }

A B C E D
aa ba ba ab ab

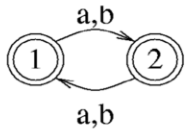
a b c d
{ A } { B, C } { E } { D }

A B C E D
bb db db cd cd

0	1
a | b	b
b | d	b
c | c	d
D | b	c

Example of Minimizing a DFA with All Accepting States and No Dead States

(a | b)*



	a	b
1	2	2
2	1	1

A B

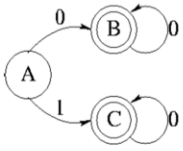
{ } {1, 2}

BB BB

	a	b
B	B	B

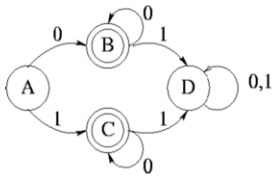
If there are no dead states and all accepting states, then the algorithm will accept any combination of input symbols that are legal and the DFA will be expressed with a single state.

Example of Minimizing a DFA w/ a Dead State



	0	1
A	B	C
B	B	–
C	C	–

Original Transition Diagram



	0	1
A	B	C
B	B	D
C	C	D
D	D	D

After Adding a Dead State

Here's an example where we need to add a dead state before starting the algorithm.

Notice we add the state D. Transitions out of D just go back to the dead state D.

Also, note that D is initially grouped with the non-accepting states since dead states are not final states.

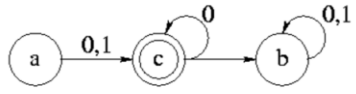
Example of Minimizing a DFA w/ a Dead State

	a	b
{A, D}	{B, C}	
bb	aa	ba ba

	a	b	c
{A}	{D}	{B, C}	
cc	bb	cb	cb

	0	1
a	c	c
b	b	b
c	c	b

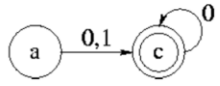
or



Remove dead state

	0	1
a	c	c
c	c	–

or



Working through the algorithm gives us this DFA.

Notice, b is still a dead state – we can never get back to an accepting state if we go to b – so we can remove it.

Structure of the Generated Analyzer

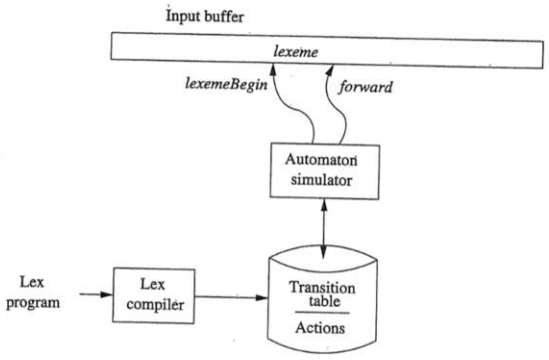
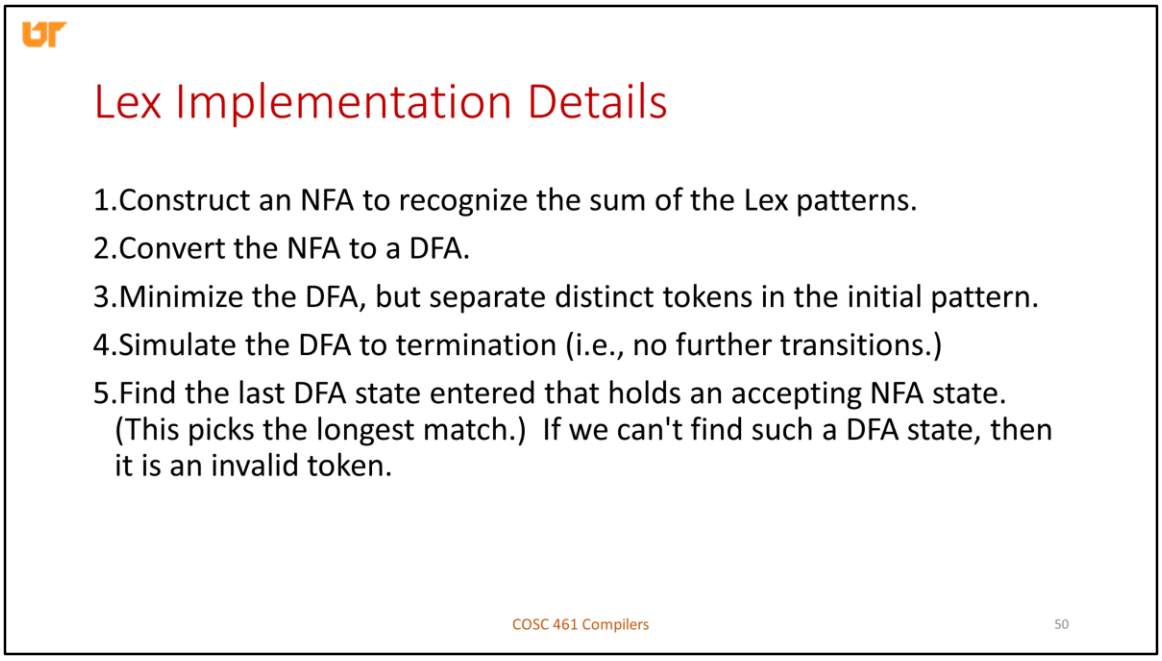


Figure 3.49: A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

OK – so we've talked a lot about how to create DFA's from NFA's and how to minimize DFA's. We also talked about how to construct an NFA from a regular expression.

Given the regular expressions in your lex source code, lex produces a transition table representing a finite automaton.

It also produces code that simulates the finite automata using the transition table that was generated. Using this code, it parses the lexemes in your program.



Putting it all together, this is how lex creates a lexical analyzer from the regular expressions in your source code.

First, it creates an NFA to recognize the sum, or union, of the lex patterns.
Then, it converts this NFA to a DFA using the subset construction algorithm we learned and that you are implementing for project 1.

Next, it minimizes the DFA using the DFA minimization algorithm, but anything that's a distinct token is separated into different groups. This allows Lex to keep track of the accepting states for individual regular expressions.

Next, it simulates the DFA on your input program until termination (i.e. until there are no more transitions).

Finally, it finds the last DFA state that it visited that holds an accepting NFA state. In this way, it always picks the longest matching token. If there is no such state, then it's an invalid token.

Example Lex Program

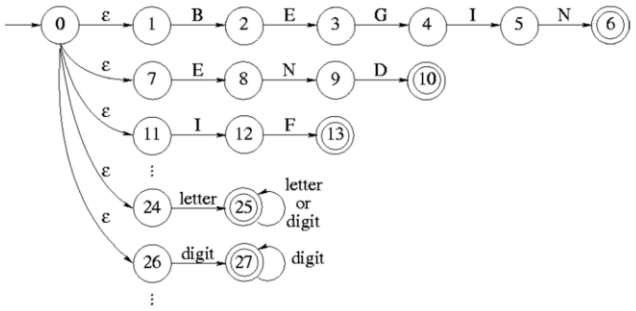
```
%%
BEGIN                { return (1); }
END                  { return (2); }
IF                   { return (3); }
THEN                 { return (4); }
ELSE                 { return (5); }
letter(letter|digit)* { return (6); }
digit+               { return (7); }
<                    { return (8); }
<=                   { return (9); }
=                    { return (10); }
<>                   { return (11); }
>                    { return (12); }
>=                   { return (13); }
```


Here's another example lex program. In this example, we return a unique integer to indicate the type of token that was matched.

Go through a few tokens.

We will examine how this is actually translated into a lexical analyzer in the next few slides.

Lex Implementation Details



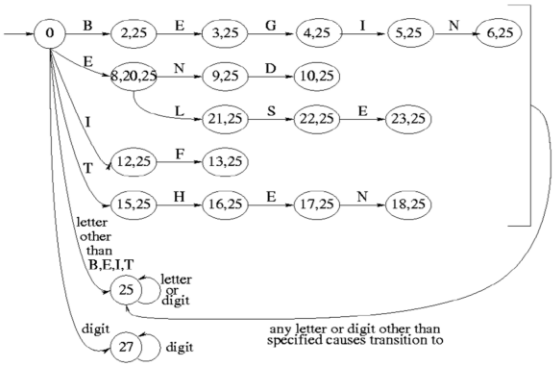
NFA for Example Lex Program

First, we create an NFA for each pattern in the file.
Notice we have NFA's for the reserved words, and then an NFA for identifiers and an NFA for literal numbers.

Then, we add a start state with epsilon transitions to the start state of each NFA.

Really 2-5, 8-9, and 12 should be final states as well since they are legal identifiers. But we will see this in the next slide after minimizing.

Lex Implementation Details



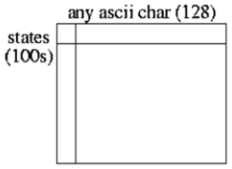
DFA for Example Lex Program

Here is the NFA after converting to a DFA and minimzing.

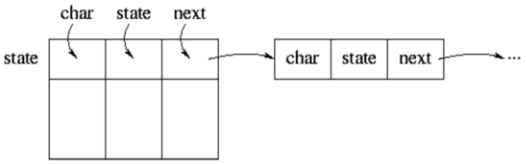
We don't show final states here – but anything with 25 or 27 is a final state.

Note that in this DFA, any DFA state that contains an NFA final state is a DFA final state.

Representing the Transition Diagram



2D array
(fastest, but too much space)



Linked list to store transitions
out of each state

There's a couple ways we could represent the transition diagram

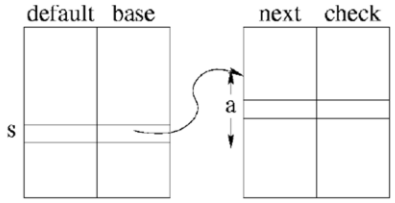
The simplest is to use a two-dimensional array. The array is accessed by the current state, and the ascii character for the next input symbol -- given a state and the next input character, we can compute the next state using the table.
This is the fastest, but it also takes up the most space. Space doesn't matter on most machines – because the table usually takes up less than 1MB – but it might matter for some embedded devices.

In these cases, it might make sense to represent each state as a list of transitions – that is, character-state pairs. On every input character, we search the list of transitions until we find a match – and then follow a pointer to the corresponding state. If we get an input character that's not in the list, then we go to some default state that indicates failure. This is much more compact, but it's slower since we have to traverse the list at each state for every input character.

Representing the Transition Diagram

- Hybrid Approach
  - Combines fast array access with linked list compactness

```
procedure nextstate(s, a):
    if check[base[s] + a] == s then
        return next[base[s] + a];
    else
        return nextstate(default[s], a);
```



There is a hybrid approach that combines fast array access with linked list compactness.
It provides the fast access of our table technique and the compactness of the linked list approach.

This approach uses four different arrays, each indexed by state numbers.

The base array gives us the base location of the entries in next and check array.
The next array contains the next state for some input symbol.
The check array is used to check if the input is a valid transition for our current state.
The default array is used as a fallback if the input is not valid. So, the default array will point to an alternate base location that we can use to try again.

Here's the procedure for nextstate. Given a state s, and an input symbol a, we calculate an index into the check array by doing base[s] + a. If that entry is equal to our current state, that means this is a valid entry for this state, and return next indexed by base[s] + a.

If it's not then we return nextstate on the state pointed to by the default array.

As an example, say we have a machine that accepts identifiers, but it also has a reserved word called ‘then’ (from an if, then, else statement). Say you’ve already matched a ‘th’, and s is the state that says you’ve matched a ‘th’ so far. You have another state that says you’ve matched ‘the’, and you also have a state that just matches identifiers. In this case, you set check[base[s] + ‘e’] to the state s to indicate that ‘e’ is a valid input on state ‘s’ and you should keep going. You also set next[base[s] + 'a'] to the state that remembers ‘the’. And you set default[s] to be the state that just matches identifiers. Then, if you get input other than an ‘e’ in state ‘s’, you know that you’re just matching an identifier, so you continue from default[s] – the state that matches identifiers.

So, the intention of this design is that the next and check can be fairly short because we can overlap a lot of similarities for each state.