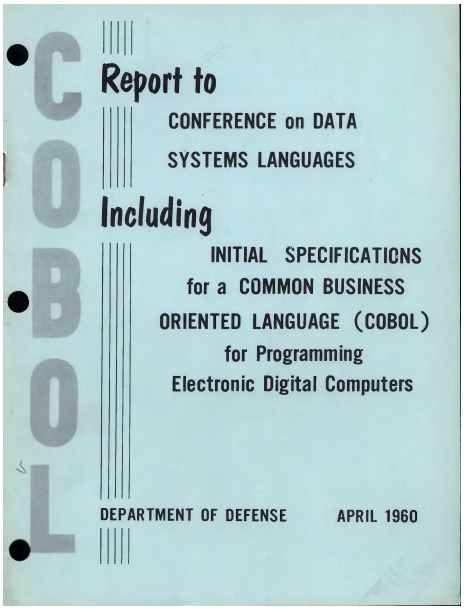
Compilers

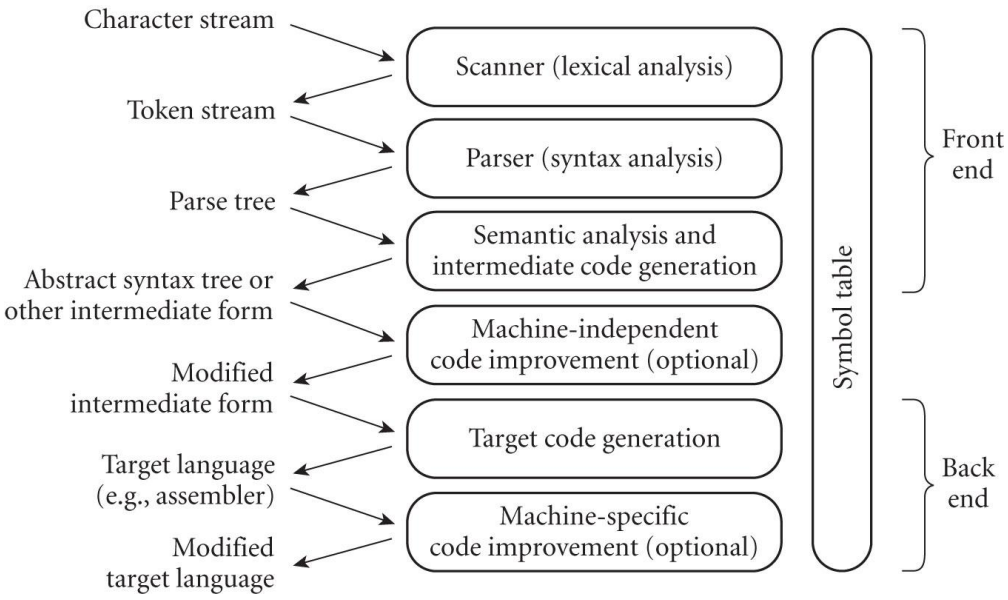
CSE 4305 / CSE 5317
M01 Lexical Analysis
Fall 2020



M01

Lexical Analysis

Phases of Compilation



Program Analysis

- A compiler's *Front End* is concerned with the *Analysis* of the source program.
  - Determining the *Meaning* of the source program.
- Generally thought of as breaking apart into three *phases*:
  - *Lexical Analysis*: Convert a text source program into *tokens*.
  - *Syntactic Analysis*: Convert a token stream into a *parse tree*.
  - *Semantic Analysis*: Enforce semantic rules and convert a parse tree into an *intermediate form*.
    - Two activities, but often happen in an interleaved fashion.

Lexical Analysis

- Take a stream of individual characters and convert it into a stream of *tokens* (possibly with *attributes*).
- Detect *lexical* errors (badly formed literals, illegal characters, etc.)
  - But *not* misspelled keywords. (Why not?)
- Discard whitespace.
  - Useful only inside literals and to separate otherwise ambiguous constructs.
- Discard comments.
  - Some toolsets interpret comments as directives.

Tokens

- A *token* is the basic building block of a program.
  - The shortest strings of characters in the source program that have individual meaning.
- Tokens come in various types, according to the programming language's specification.
  - Common types include *numbers*, *identifiers*, *keywords*, *operators*, *punctuation marks*, and so forth.

Some Generic Token Categories …

Pattern	*Category*	*Attribute*
letter or underscore possibly followed by letters, underscores, or decimal digits	`ID`	symbol table entry
decimal digits	`INTEGER_LITERAL`	int number
" characters "	`STRING_LITERAL`	string
`if`	`IF`	
`<` *or* `<=` *or* `==` *or* `>=` *or* `>`	`RELATIONAL_OPERATOR`	enum value
`[0-9]*(([0-9][.])|([.][0-9]))[0-9]*`	`REAL_LITERAL`	FP number
`(`	`LEFT_PARENTHESIS`	

Example …

```
if ( a <= 17.34 ) {
 b = b + 1;
} else {
 // Oops!
 print( "too big!" );
}
```

Characters	*Token*	*Attribute*
if	IF	
(	LEFT_PARENTHESIS	
a	ID	"a"
<=	RELATIONAL_OPERATOR	LE
17.34	REAL_LITERAL	17.34
)	RIGHT_PARENTHESIS	
{	LEFT_BRACE	
b	ID	"b"
=	ASSIGN_OPERATOR	ASSIGN
b	ID	"b"
+	ADD_OPERATOR	PLUS
1	INTEGER_LITERAL	1
;	SEMICOLON	
}	RIGHT_BRACE	
else	ELSE	
{	LEFT_BRACE	
print	ID	"print"
(	LEFT_PARENTHESIS	
"too big!"	STRING_LITERAL	"too big!"
)	RIGHT_PARENTHESIS	
;	SEMICOLON	
}	RIGHT_BRACE	

Lexical Analysis

- So how to do this conversion?
- We could just hand-code a routine …
- Cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
void tokenize( char *inStr ) {
  int ptr = 0;

  while ( inStr[ ptr ] ) {
    if ( isspace( inStr[ ptr ] ) ) {
      ptr += 1;

    } else if ( isdigit( inStr[ ptr ] ) ) {
      int n = 0;

      while ( isdigit( inStr[ ptr ] ) ) {
        n = n*10 + inStr[ ptr ]-'0';
        ptr += 1;
      }

      printf( "    INTEGER_LITERAL %d\n", n );

    } else if ( isalpha( inStr[ ptr ] ) || inStr[ ptr ] == '_' ) {
      int ptrBegin = ptr;

      while ( isalpha( inStr[ ptr ] ) ||
              isdigit( inStr[ ptr ] ) ||
              inStr[ ptr ] == '_' ) {
        ptr += 1;
      }

      printf( "    ID '%.*s'\n", ptr-ptrBegin, &inStr[ptrBegin] );

    } else {
      printf( "Illegal character '%c'\n", inStr[ ptr ] );
      ptr += 1;
    }
  }
```

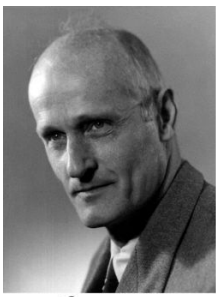
Lexical Analysis

- Hand-coded lexical analyzers are fun for about 10 seconds and then one realizes that they're …
  - Tedious to write.
  - Difficult to extend.
    - Imagine adding `REAL_LITERAL` to this tokenizer.
  - Error-prone.
    - Imagine adding `REAL_LITERAL` to this tokenizer and not screwing up `INTEGER_LITERAL` along the way.
  - Hard to separate into *patterns* and *processing*.

Lexical Analysis

- We want to express the *patterns* of token classes as *Regular Expressions*.
  - Easy to write, easy to update, reduced chance of errors, lots of theory to help with the processing.
- We want to express the *processing* of token classes independently of the *pattern*.
  - Reduced chance of error, easier to write, easier to update.

Regular Expression (RE)



Stephen C. Kleene

/ˈkleɪni/ KLAY-nee

1. A *character*
2. The *empty string*, denoted as ε
3. The *concatenation* of two REs

   Meaning one RE after the other RE
4. The *alternation* of two REs, denoted by |

   Meaning one RE *or* the other RE
5. An RE followed by the *Kleene star*, denoted as *

   Meaning *zero* or *more* repetitions of the RE

https://www.nap.edu/read/9649/chapter/10

Regular Expressions

- Aside from the basic five items, there are some common notation extensions that make writing regular expressions more convenient.
- An RE followed by the *Kleene Plus*, denoted by `+`
  - Meaning *one* or *more* repetitions of the RE.
  - `a+` is equivalent to `aa*`, so no extra power.
- An RE followed by `?`
  - Meaning *zero* or *one* instances of the RE.
  - `a?` is equivalent to `(a|ε)`, so no extra power.
- A set or range of characters in brackets, `[]`
  - Meaning any *one* of the indicated characters.
  - `[0-9]` is equivalent to `0|1|2|3|4|5|6|7|8|9`, so no extra power.

Regular Expressions

- Regular Expressions are just one category of *formal* languages.
- These categories can be organized in a hierarchy according to the kinds of languages they can describe (and their parsing complexity, required resources, etc.)
- Different levels have different rules for their *productions*.

Grammars and the Chomsky Language Hierarchy

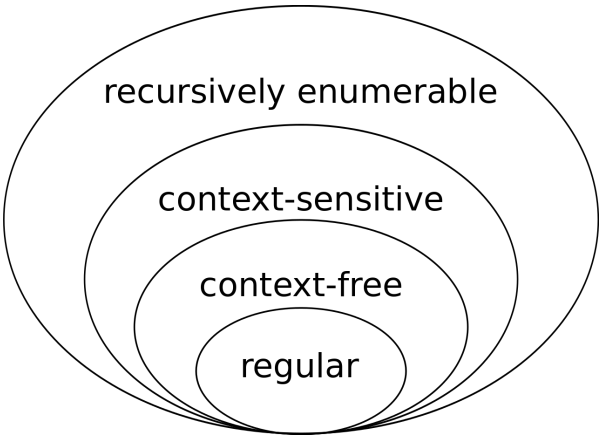
- A *grammar* specification includes
  - A set of *terminal* symbols (T)
  - A set of *non-terminal* symbols (N)
    - $V = T \cup N$
  - A set of *production rules*
  - A *start symbol*

Terminals	a, b
Non-terminals	S
Production rules	$S \rightarrow aSb$ $S \rightarrow \varepsilon$
Start symbol	S

- Restrictions on the form of the production rules can be used to categorize grammars into classes, each of which is more *expressive* than the previous.
- The example expresses the language $a^n b^n$, for $n \geq 0$.

Grammars and the Chomsky Language Hierarchy

- The Chomsky Hierarchy shows four levels of grammar expressivity.
- Each properly includes the previous.
- The levels have increasing expressiveness, but also parsing complexity, resource requirements, etc.



https://commons.wikimedia.org/wiki/File:Chomsky-hierarchy.svg

[*Grammars and the Chomsky Language Hierarchy*]

Type	*Name*	*Allowed Productions*	*Example Language*	*Example Grammar*	*Example Use*	*Recognizing Automaton*	*Storage Required*	*Parsing Complexity*
0	Recursively Enumerable	$\alpha \rightarrow \beta$ $\alpha \in V+$ $\beta \in V^*$			(*Theoretical Interest*)	Turing Machine	Infinite Tape	Undecidable
1	Context Sensitive	$\alpha \rightarrow \beta$ $S \rightarrow \varepsilon$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$ $\alpha \in V^*NV^*$ $\beta \in V+$ *S not on any RHS*	$a^n b^n c^n$ $n > 0$	$S \rightarrow aSBC$ $S \rightarrow aBC$ $CB \rightarrow BC$ $aB \rightarrow ab$ $bB \rightarrow bb$ $bC \rightarrow bc$ $cC \rightarrow cc$	(*Theoretical Interest*)	Linear Bounded Automaton	Tape a linear multiple of input length	NP Complete
2	Context Free	$A \rightarrow \alpha$ $A \in N$ $\alpha \in V^*$	$a^n b^n$ $n \geq 0$	$S \rightarrow aSb$ $S \rightarrow \varepsilon$	Arithmetic Expressions, Programming Languages	Pushdown Automaton	Pushdown stack	$O(n^3)$
3	Regular	$A \rightarrow xB$ $A \rightarrow x$ $A, B \in N$ $x \in T^*$	$a^n b$ $n > 0$	$S \rightarrow aS$ $S \rightarrow ab$	Token Formats, String Recognition	Finite Automaton	Finite	$O(n)$

[*"Recursive" Production Rules …*]

- Later we'll learn that *recursion* is what separates *Context-Free Grammars* from *Regular Expressions*. No recursion in REs.
- On the previous chart, $S \rightarrow aS$ appears as a rule for an RE.
  - Hey, isn't that *recursive*? After all, *S* refers to itself, right?

- ***No!*** It just ***looks*** as if it's recursive. :)
- Because the *S* appears on the far right (there's nothing after *S* in the rule), this is just a way of expressing a Kleene-* operation (a*).
- A proof that this isn't true recursion requires going deeper in formal language theory than required for this class, so just accept it as true.

Lexical Analysis

Token Class	*Regular Expression*
ID	[_a-zA-Z][_a-zA-Z0-9]*
INTEGER_LITERAL	[0-9]+
STRING_LITERAL	"[^"\n]*"
IF	if
RELATIONAL_OPERATOR	<|<=|==|>=|>
REAL_LITERAL	[0-9]*(([0-9][.])|([.][0-9]))[0-9]*
LEFT_PARENTHESIS	(

Processing Regular Expressions

- How to use Regular Expressions to do lexical scanning?
  - That is, how do we convert an RE into a program?
- Recognize that there's a correspondence between an RE and a *Finite Automaton*.
  - And a *Finite Automaton* is convertible into *scanning* (or *recognizing*) program in a mechanical way.

Finite Automaton (FA)

- A *Finite Automaton* consists of …
  - A finite set of *symbols*, known as its *alphabet*.
  - A finite set of *states*.
  - A finite set of *edges* each of which …
    - … goes from one state to another (possibly the same) state.
    - … is labelled with a *symbol*.
  - One identified state known as the *start state*.
    - Usually state 1.
  - A subset of states identified as *final* (or *accepting*) states.

Finite Automaton (FA)

- A finite automaton *scans* a *finite* input string by …
  - Starting in the *start* state.
  - For each successive *symbol* in the input string, transiting along an edge from the current state that is labelled with the symbol.
  - *Or* at any time transiting along an accessible edge labelled with ε.
- After all symbols are used up (and all ε moves are made), if the current state is a *final* state, the FA *accepts* the input, otherwise it *rejects* the input.

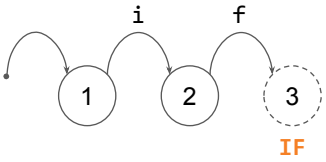
Finite Automaton (FA)

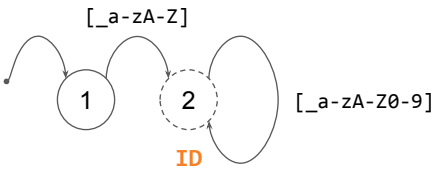
- The set of strings *accepted* by an FA is said to be its *language*.
- Though all FA are *finite*, the *language* accepted by an FA can be *infinite*.
  - Each *string* in the language, however, is itself *finite*.

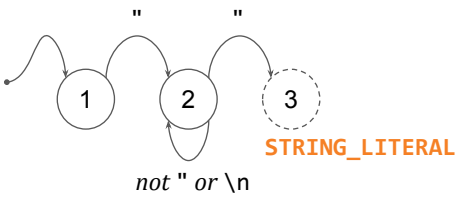
Deterministic Finite Automaton (DFA)

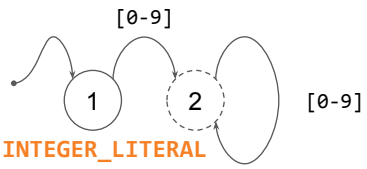
- If a Finite Automaton complies with the following two criteria, it is said to be *Deterministic* …
  - No two edges from a state can have the *same* symbol.
    - Each symbol valid for a state appears on only *one* edge.
  - No edge is labelled with ε.
    - ε is the *empty* transition, i.e., no symbol is consumed.
- When a FA is *deterministic*, the acceptance or rejection of an input string will take no more transitions than the length of the input.
  - Why?

Deterministic Finite Automaton Examples



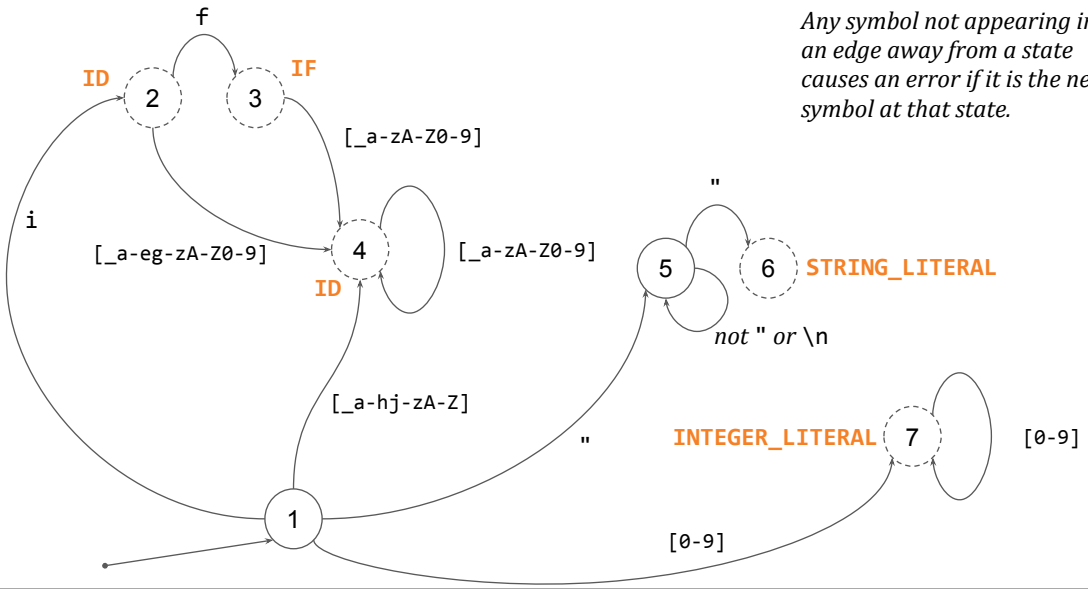






Any symbol not appearing in an edge away from a state causes an error if it is the next symbol at that state.

Combining Deterministic Finite Automata



Any symbol not appearing in an edge away from a state causes an error if it is the next symbol at that state.

Lexical Analysis

- What a pain in the butt!
- Imagine having to do that kind of construction for a programming language with dozens of overlapping token class definitions.
- However, we can express this complex Finite Automaton in a more convenient form …

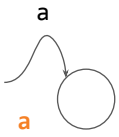
Nondeterministic Finite Automata (NFA)

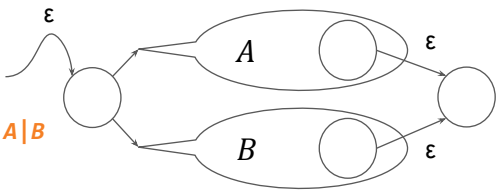
- If an FA satisfies either of the two following criteria, it is *Non-Deterministic* …
  - At least one state has at least two edges going to different other states and bearing the the same symbol.
  - At least one edge is labelled with ε.
- Does being *non-deterministic* increase the descriptive power of a Finite Automaton?
  - ***No!*** Why not?

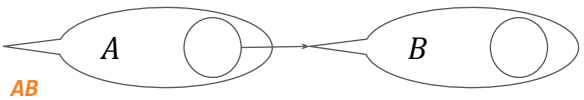
Making a Regular Expression into an NFA

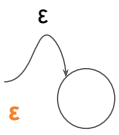
- Turns out that it's trivial to convert a Regular Expression into an NFA.
- We'll show this *Constructively* by giving ways to convert each of the five kinds of Regular Expressions into an NFA.
- In the following, a is any symbol, ε is the empty string, and *A* and *B* are any NFAs representing regular expressions.

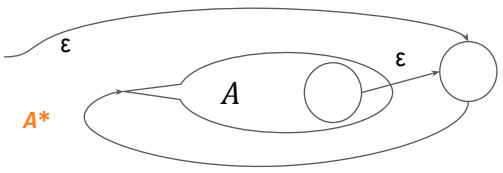
Making an RE into an NFA



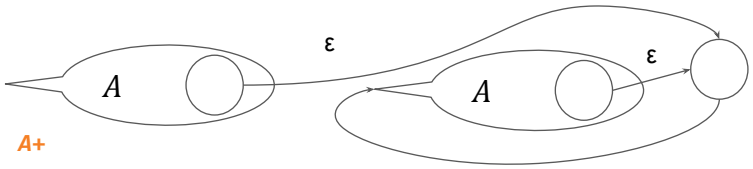




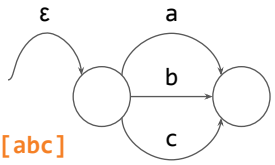




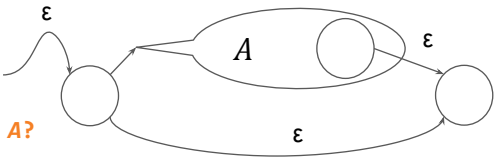
Making an RE into an NFA



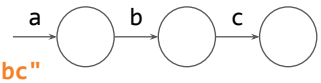
A+



[abc]

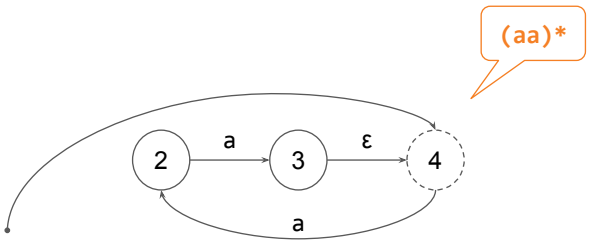


A?

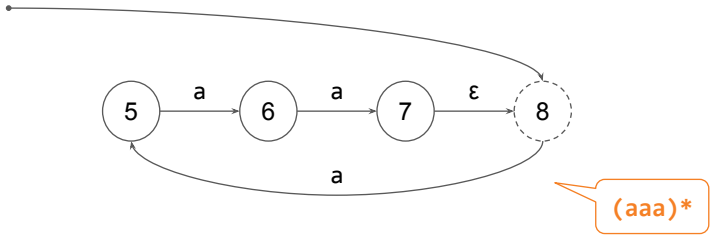


"abc"

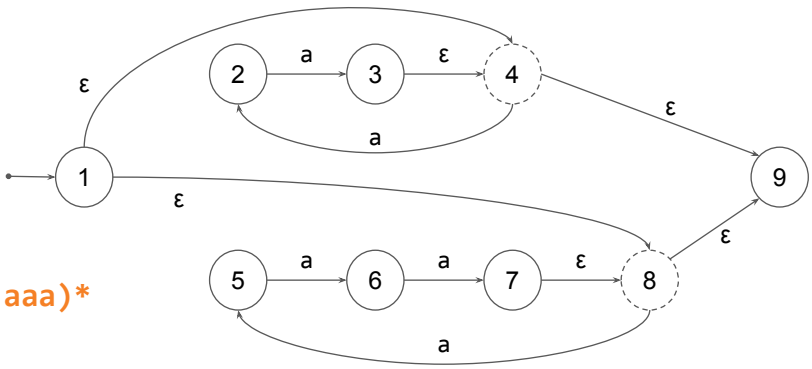
Making an RE into an NFA Example …



Making an RE into an NFA Example …



Making an RE into an NFA Example …



(aa)*|(aaa)*

Using an NFA for Lexical Scanning

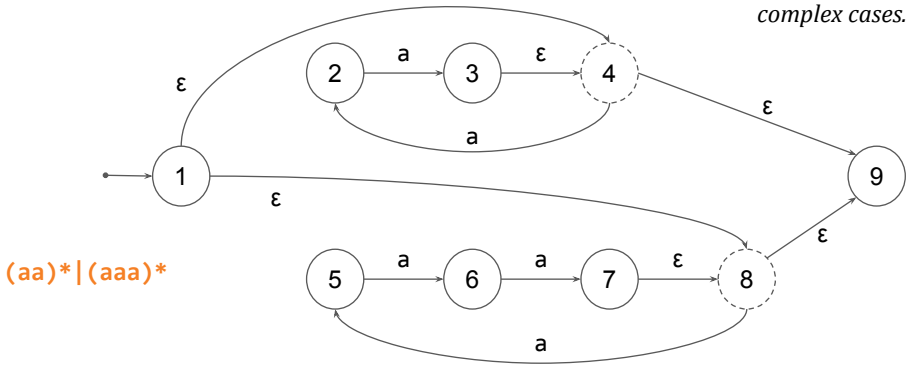
- It was trivial to convert that RE into an NFA.
- What about using the NFA for lexical scanning?
- This is not so trivial.
- As symbols are consumed, we might have to *guess* which edge to traverse.
  - If a state has multiple outgoing edges with the same symbol.
- We also might have to *guess* whether to use an edge or not.
  - If a state has outgoing edge(s) with ε.
- FYI: Few computers have good guessing hardware.

Converting an NFA into a DFA

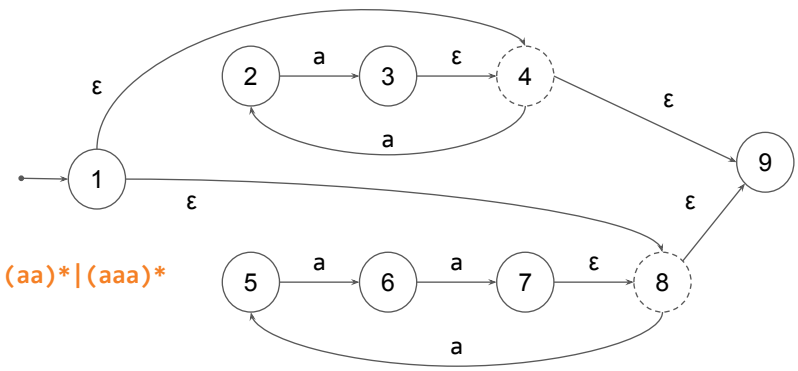
- We have to keep track of whichever state the NFA *might* be in at any point.
- We do this by taking the current state and computing its ε-*closure*.
  - That is, all states that can be reached from it solely by ε transitions (perhaps several in a row …).
- This results in a (possibly) *combined state*.
- For that combined state, we compute the ε-closure of the destination combined states for each symbol on outgoing edges.
- And then repeat until we formed the total set of possible ε-closed combined states …

NFA to DFA Example …

This is a simple example with only one symbol, a. *The process is the same for more complex cases.*

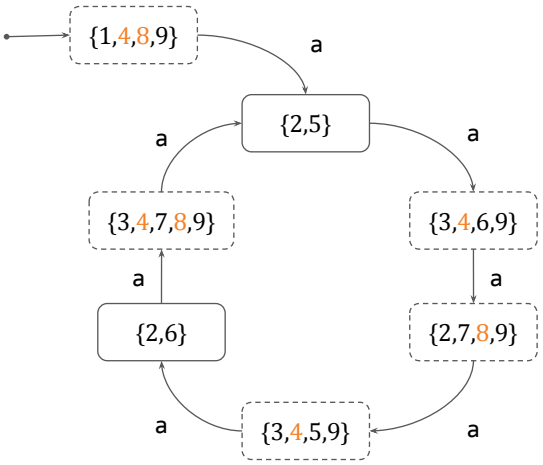


NFA to DFA Example …



State	a⟶	ε⟶
Start {1}	—	{1,4,8,9}
{1,4,8,9}	{2,5}	{2,5}
{2,5}	{3,6}	{3,4,6,9}
{3,4,6,9}	{2,7}	{2,7,8,9}
{2,7,8,9}	{3,5}	{3,4,5,9}
{3,4,5,9}	{2,6}	{2,6}
{2,6}	{3,7}	{3,4,7,8,9}
{3,4,7,8,9}	{2,5}	{2,5}

NFA to DFA Example …



State	a⟶	ε⟶
Start {1}	—	{1,4,8,9}
{1,4,8,9}	{2,5}	{2,5}
{2,5}	{3,6}	{3,4,6,9}
{3,4,6,9}	{2,7}	{2,7,8,9}
{2,7,8,9}	{3,5}	{3,4,5,9}
{3,4,5,9}	{2,6}	{2,6}
{2,6}	{3,7}	{3,4,7,8,9}
{3,4,7,8,9}	{2,5}	{2,5}

(aa)*|(aaa)*

Converting an NFA into a DFA

- Once the complete set of combined states is generated, the set of *final states* can be identified.
- A combined state that includes *any* final state from the original NFA is a final state in the DFA.
  - These are marked with a special color in the previous slides.
  - The final combined states are *dashed* boxes.

Converting a DFA to a Program

- As we stated previously, deciding if a DFA *accepts* a string is O(n), where n is the length of the string.
  - Every transition of the DFA is deterministic and therefore consumes a symbol from the input string.
- It's not hard to write a program that mechanically converts a set of *Regular Expressions* (and their corresponding action routines) into a *program*.
  - Such a program is a *Lexical Analyzer Generator*.

Lexical-Analyzer Generators

- Remember that we said an advantage of REs as a formalism is that there's lots of theory already developed to help us.
- `lex` was developed in 1975 to take sets of regular expressions and actions associated with each and automatically generate a lexical scanner.
  - Worked with C under UNIX.
- Zillions of subsequent tools were created to do the same kind of thing but with different notations, different target languages, different environments, etc.

`flex`

- We will be using `flex` (***f***ast ***lex***ical analyzer generator) …
  - A derivative of the original `lex` tool.

```
pi@icywits-01:~ $ apt-cache policy flex
flex:
  Installed: 2.6.4-6.2
  Candidate: 2.6.4-6.2
  Version table:
 *** 2.6.4-6.2 500
        500 http://raspbian.raspberrypi.org/raspbian buster/main armhf Packages
        100 /var/lib/dpkg/status
pi@icywits-01:~ $ flex --version
flex 2.6.4
pi@icywits-01:~ $
```

Lexical Analysis

- Remember the hand-coded tokenizer routine?
- We thought it was pretty cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
void tokenize( char *inStr ) {
  int ptr = 0;

  while ( inStr[ ptr ] ) {
    if ( isspace( inStr[ ptr ] ) ) {
      ptr += 1;

    } else if ( isdigit( inStr[ ptr ] ) ) {
      int n = 0;

      while ( isdigit( inStr[ ptr ] ) ) {
        n = n*10 + inStr[ ptr ]-'0';
        ptr += 1;
      }

      printf( "    INTEGER_LITERAL %d\n", n );

    } else if ( isalpha( inStr[ ptr ] ) || inStr[ ptr ] == '_' ) {
      int ptrBegin = ptr;

      while ( isalpha( inStr[ ptr ] ) ||
              isdigit( inStr[ ptr ] ) ||
              inStr[ ptr ] == '_' ) {
        ptr += 1;
      }

      printf( "    ID '%.*s'\n", ptr-ptrBegin, &inStr[ptrBegin] );

    } else {
      printf( "Illegal character '%c'\n", inStr[ ptr ] );
      ptr += 1;
    }
  }
```

`flex` Version

- Each token type has its own processing routine.
- Token formats expressed as regular expressions.
- Cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
[0-9]+  {
  printf( "    INTEGER_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
  printf( "    ID '%s'\n", yytext );
}

[ \t\r\n] { /* Ignore whitespace */ }

.       { printf( "Illegal character '%s'\n", yytext ); }
```

`flex` Version (2)

- Now with `REAL_LITERAL`.
- Not hard to add.
- *Really* cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    REAL_LITERAL 1234.345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

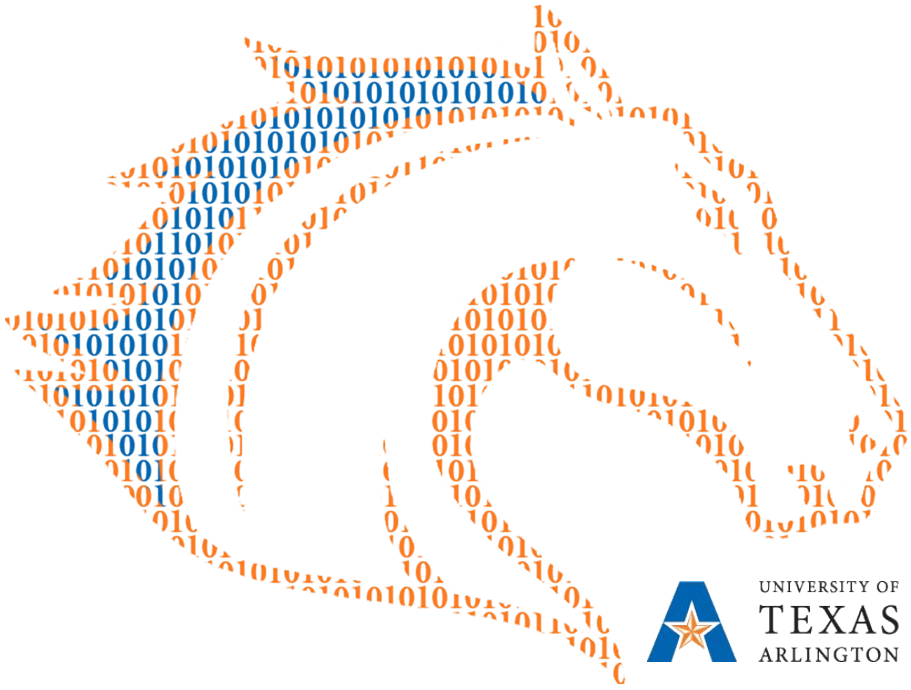
```
[0-9]+  {
  printf( "    INTEGER_LITERAL %s\n", yytext );
}

[0-9]+\.[0-9]+ {
  printf( "    REAL_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
  printf( "    ID '%s'\n", yytext );
}

[ \t\r\n] { /* Ignore whitespace */ }

.       { printf( "Illegal character '%s'\n", yytext ); }
```

`flex` Version (3)

- Now with `STRING_LITERAL`.
- Again, not hard to add.
- *Really, really* cool, huh?

```
[0-9]+      {
    printf( "    INTEGER_LITERAL %s\n", yytext );
}

[0-9]+\.[0-9]+  {
    printf( "    REAL_LITERAL %s\n", yytext );
}

["][^"\n]*["]   {
    printf( "    STRING_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
    printf( "    ID '%s'\n", yytext );
}

[ \t\r\n]   { /* Ignore whitespace */ }

.           { printf( "Illegal character '%s'\n", yytext ); }
```

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    REAL_LITERAL 1234.345
    STRING_LITERAL "bob"
    ID 'Maddog87'
```



UNIVERSITY OF TEXAS ARLINGTON | DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING