CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Administrivia

- No lecture last week, and no playlist recorded – I apologize, I was really sick ☹
- Syllabus updated:
  - Lecture 3 today
  - Lecture 4 next week (regular Wednesday lecture time)
  - Lecture 5 will be recorded on Friday, 10/6 from 9-12am PT (attendance optional, video available afterwards – please watch before Lecture 6 on 10/13)
  - HW1 and HW2 deadlines moved by a week (HW1 released)
  - All quiz deadlines moved by a week
  - Once again, I sincerely apologize for the inconvenience.

Agenda

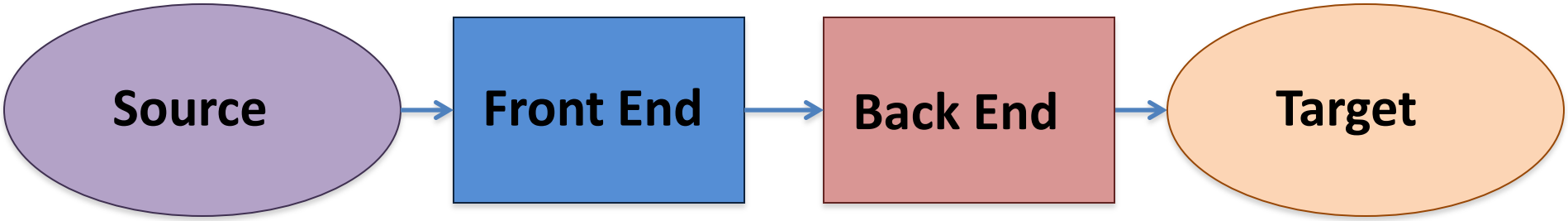
- Quick review:
  - Formal languages and grammars
  - Regular expressions
- Finite automata
  - Deterministic Finite Automata
  - Non-deterministic Finite Automata
- Scanners and Tokens

Credits For Course Material

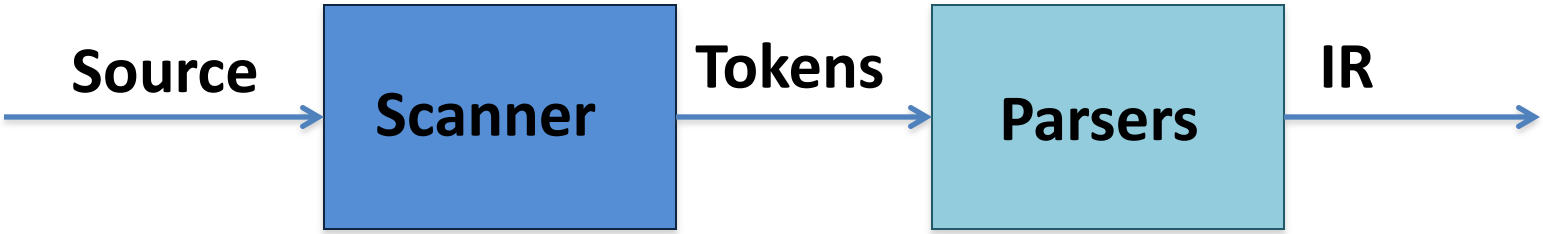
- **Big thank you to UW CSE faculty member, Hal Perkins**
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Review: a Structure of a Compiler

- At a high level, a compiler has two pieces:
  - **Front end – analysis**
    - Read source program
    - Discover its structure and meaning
  - **Back end – synthesis**
    - Generate equivalent target language program



Review: Compiler - Front End



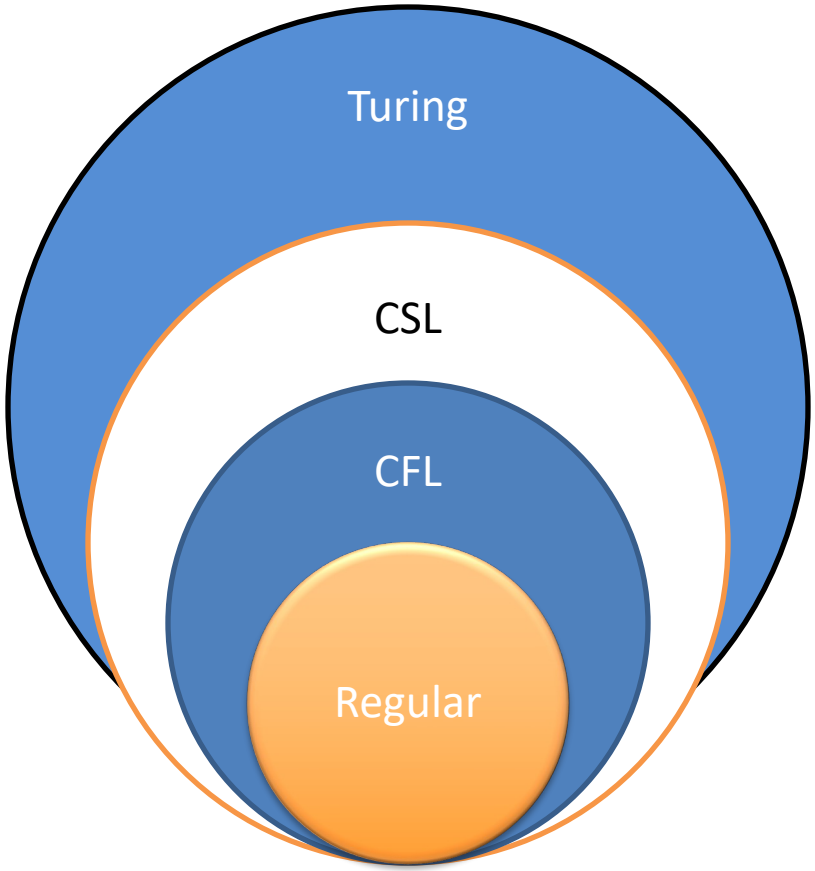
- **Front end is usually split into two parts:**
  1. **Scanner** – converts character stream into token stream: keywords, operators, variables, constants
     - Also: strips out white space, comments
  2. **Parser** - reads token stream; generates IR
     - Either here or shortly after, perform semantics analysis (checks for things like type errors)

Formal Languages & Automata Theory (One slide review)

- Alphabet: a finite set of symbols and characters
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set of strings (possibly empty or infinite)
- Finite specifications of (possibly infinite) languages
  - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  - Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Chomsky's Language Hierarchy: Quick Reminder

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
  - Specs and implementation of scanners
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
  - Specs and implementation of parsers
- Context-sensitive (Type-1) languages … aren't too important (at least for us)
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



Backus-Naur Form (BNF)

- Backus-Naur Form (BNF): a syntax for describing language grammars in terms of transformation rules, of the form:

  <symbol> ::= <expression> | <expression> ... | <expression>

  - Terminal: a fundamental symbol of the language
  - Non-terminal: a high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.

- Developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

Regular Expressions

Regular Expressions and FAs

- Lexical grammar (structure) of most programming languages can be specified with regular expressions
  - *(Sometimes a little cheating is needed)*
- Tokens can be recognized by a deterministic finite automaton
  - Can be either table-driven or built by hand, based on lexical grammar

Regular Expressions

- Defined over some alphabet Σ
  - For programming languages, alphabet is usually ASCII or Unicode
- If *re* is a regular expression, *L*(*re*) is the language (set of strings) generated by *re*

- In other words, regular expression *re* describes the set of strings, called a language *L*(*re*), over the elements of some alphabet *Σ*

Fundamental REs

re	*L*(*re*)	Notes
a	{ a }	Singleton set, for each a in Σ
ε	{ ε }	Empty string
∅	{ }	Empty language

Operations on REs

re	*L*(*re*)	Notes
r s	L(r)L(s)	Concatenation
r|s	L(r) ∪ L(s)	Combination (union)
r*	L(r)*	0 or more occurrences (Kleene closure)
r^+	$L(r)^+$	1 or more occurrences (Positive closure)

- Precedence: * (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed

Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex, JFlex do this automatically, given a set of REs

Finite State Automata

Finite State Automaton

- A finite state automaton is a formal mathematical object, defined as a five-tuple (S, Σ, δ, s_0, S_A):
  - S – the finite set of states of an automation (it may include an error state s_e)
  - Σ – the finite set of transition states (in the case of a scanner, the set of transition states is the alphabet)
  - $\delta(s, c)$ – the transition function that, for each state s, and each character (symbol) from the set $\{\Sigma \cup \varepsilon\}$ gives a set of new states
  - s_0 – the initial (start) state
  - S_A – the set of accepting (final) states

Finite State Automaton

- A finite set of states
  - One marked as initial state
  - One or more marked as final states
  - States sometimes labeled or numbered
- A set of transitions from state to state
  - Each labeled with symbol from Σ, or ε
  - Common to allow multiple labels (symbols) on one edge to simplify diagrams
- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - ε-transition can be taken at any time
- Accept when final state reached & no more input
  - Slightly different in a scanner where the FSA is a subroutine that accepts the longest input string matching a token regular expression, starting at the current location in the input
- Reject if no transition possible, or no more input and not in final state (DFA)
  - Some versions require an explicit “error” state and transitions to it on all “no legal transition possible” input

Finite State Automaton

- Based on their purpose, finite state automata (finite state machines) can be classified into three major groups:
  - Acceptors (recognizers) - automata that compute Boolean functions. They do so by either accepting or rejecting the inputs given to them
  - Classifiers – automata that has more than two final states and it gives a single output when it terminates
  - Transducers – automata that produces outputs based on current input and/or previous state is called a transducer. Transducers can be of two types:
    - Mealy machine
    - Moore machine

Finite State Automaton

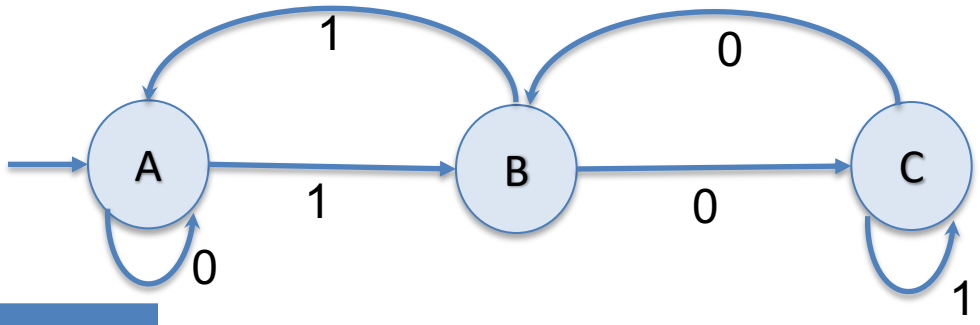
- Based on their transitions, finite state automata (finite state machines) can be classified into two major groups:
  - Deterministic Finite State Automaton (DFA)
  - Non-deterministic Finite State Automaton (NDFA/NFA)

DFA vs NFA

- Deterministic Finite Automata (DFA)
  - No choice of which transition to take under any condition
  - No empty (ε) transitions (arcs)
- Non-deterministic Finite Automata (NFA)
  - Choice of transition in at least one case
  - Accept if some way to reach a final state on given input
  - Reject if no possible way to final state
  - i.e., may need to guess right path or backtrack

DFA - Example

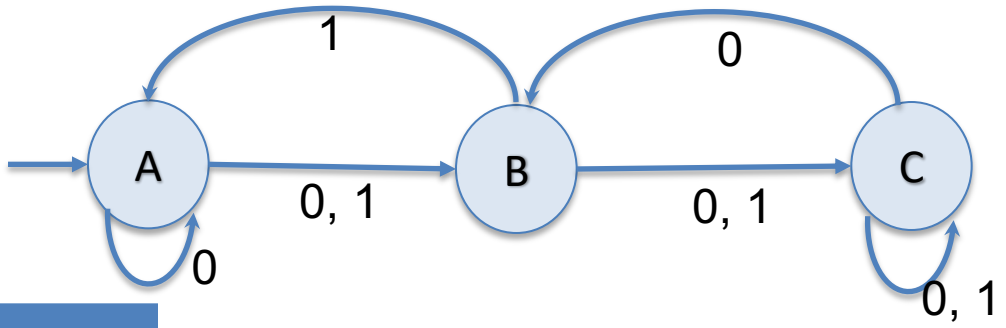
- Let's consider a deterministic automaton with:
  - States, Q = {A, B, C}
  - Inputs, Σ = {0, 1}
  - Initial state, q_0 = {A}
  - Final state, f = {C}
  - Transition table given as:



Current state	Next state with q =0	Next state with q = 1
A	A	B
B	C	A
C	B	C

NFA - Example

- Let's consider a non-deterministic automaton with:
  - States, Q = {A, B, C}
  - Inputs, Σ = {0, 1}
  - Initial state, q_0 = {A}
  - Final state, f = {C}
  - Transition table given as:



Current state	Next state with q =0	Next state with q = 1
A	A, B	B
B	C	A, C
C	B, C	C

DFA vs NFA

DFA	NFA
Deterministic – for every input symbol, a transition is from a current state to a particular next state	**Non-deterministic** – for at least one input symbol, a transition is from a current state to multiple possible next states
Empty ε transitions do not exist	Empty transitions are permitted
Backtracking is allowed.	Backtracking is not always possible.
Typically, requires more space	Typically, requires less space
Some sequence of inputs is accepted, if the automaton transitions to a final state	Some sequence of inputs is accepted, if at least one of the possible transitions ends in a final state

DFA – Example 2

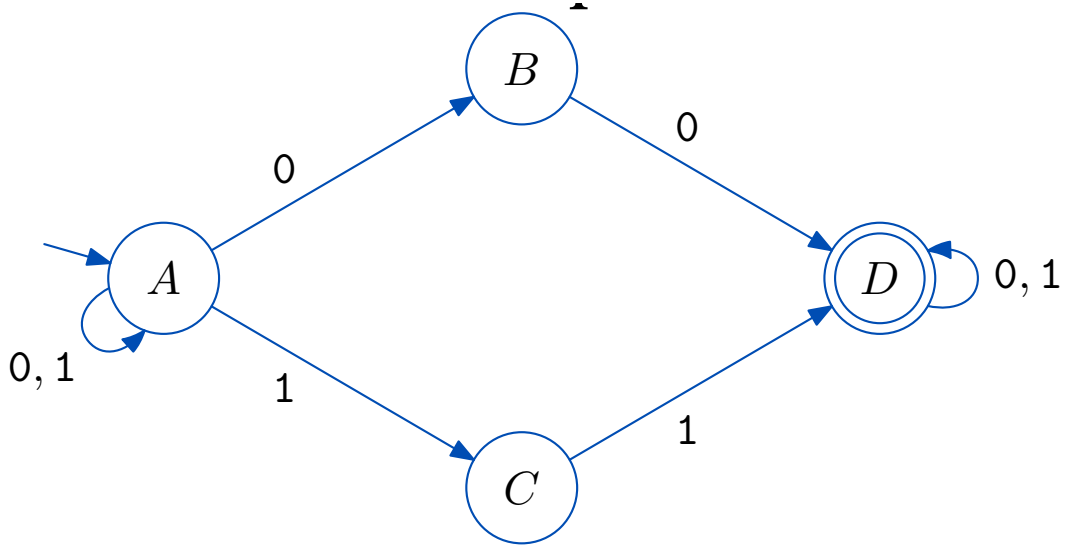
- Build a deterministic state automaton (DFA) that recognized word “pass123”.

DFA – Example 3

- Given an alphabet {4, 5, 6, 7}, build an automaton that accepts the words that contain {5, 6} as a subword.

NFA – Example 2

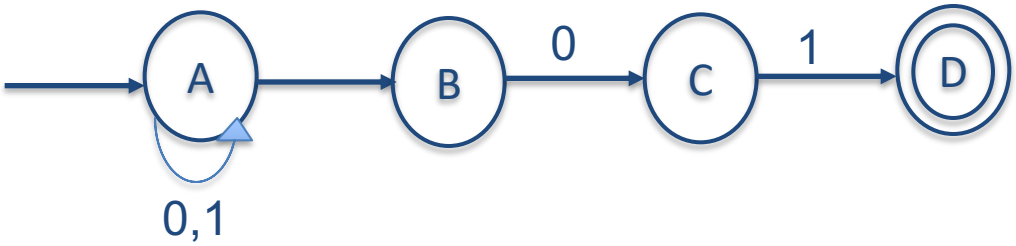
- What does this NFA accept?



[Exmple credit: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/3a.pdf

NFA – Example 3

- What does this NFA accept?



[Exmple credit: https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/3a.pdf]

Acceptability by DFA and NFA

- Some rules:
  - A string is accepted by a DFA/NFA if and only if the DFA/NFA starting at the initial state ends in an accepting state (any of the final states, for the NFA) after reading the whole string.

  In other words, some string $s \in S$ is accepted by a DFA/NFA $(Q, \Sigma, \delta, q0, F)$, if and only if $\delta^*(q0, S) \in F$

- A language L accepted by DFA/NFA if $\{S | S \in \Sigma^* \text{ and } \delta^*(q0,S) \in F\}$

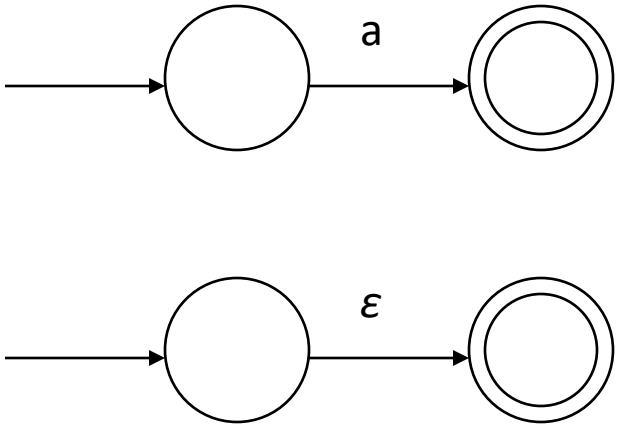
FAs in Scanners

- We want DFA for speed (no backtracking)
- But conversion from regular expressions to NFA is easy
- Fortunately, there are a well-defined procedure for converting a NFA to an equivalent DFA:
  - Subset construction
  - Brzozowski's algorithm

From RE to NFA – Thompson's Construction

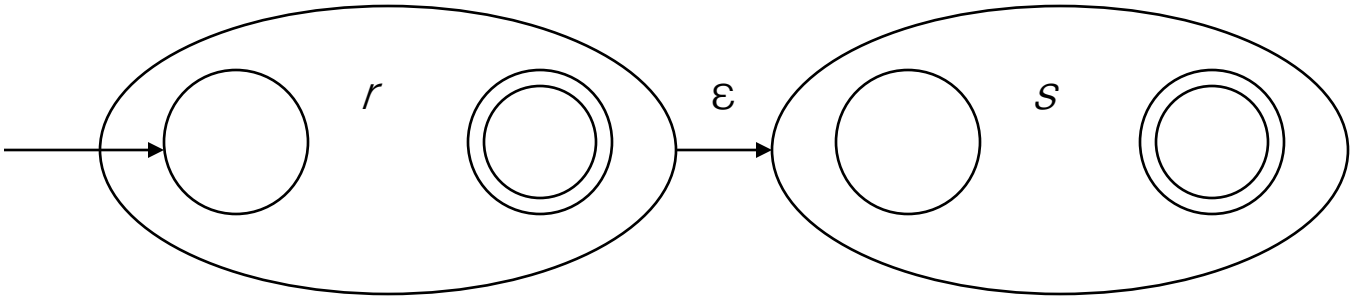
From RE to NFA – Thompson's Construction Base cases

- The construction begins by building trivial NFAs for each character in the alphabet (including ε-transition)
- Each NFA has one start and one accepting state



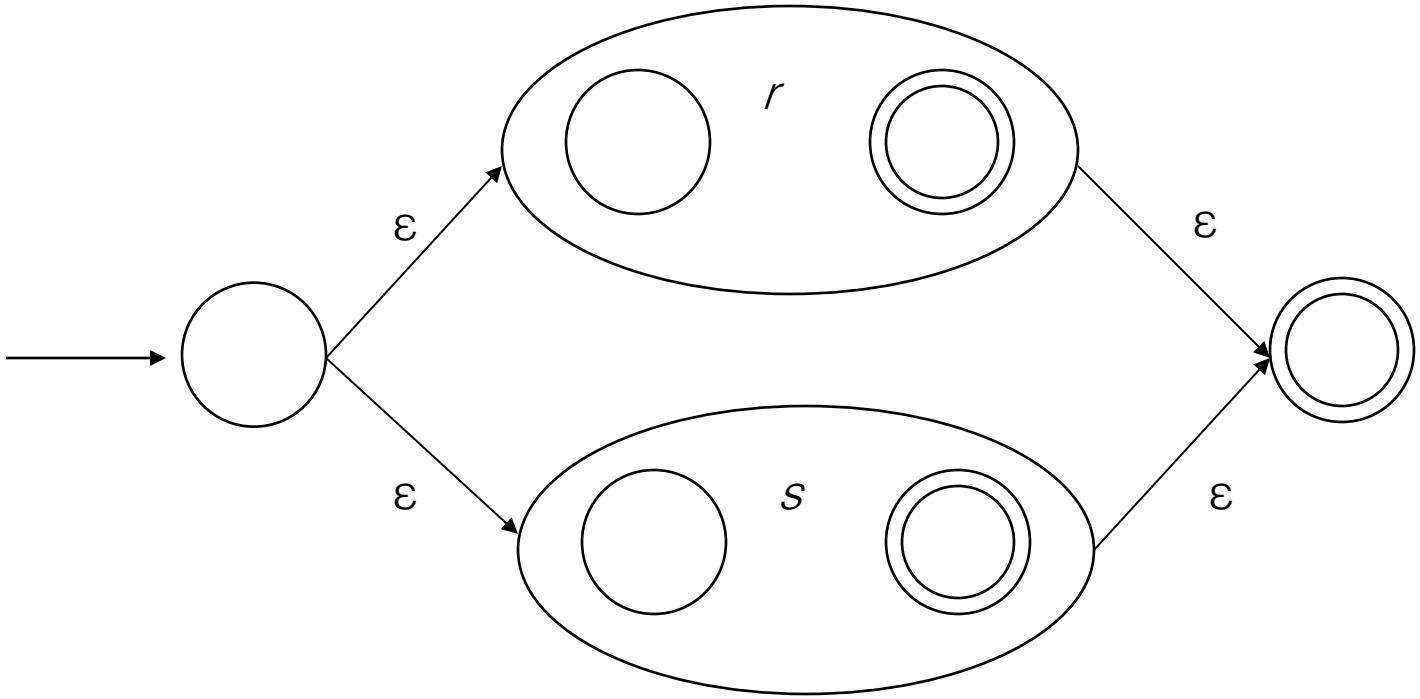
r s

- The construction begins by building trivial NFAs for each character in the alphabet (including *ε*-transition)
- Each NFA has one start and one accepting state
- **An *ε*-transition always connects two states that were, earlier in the process, the start and the accepting states of NFAs for some component REs**



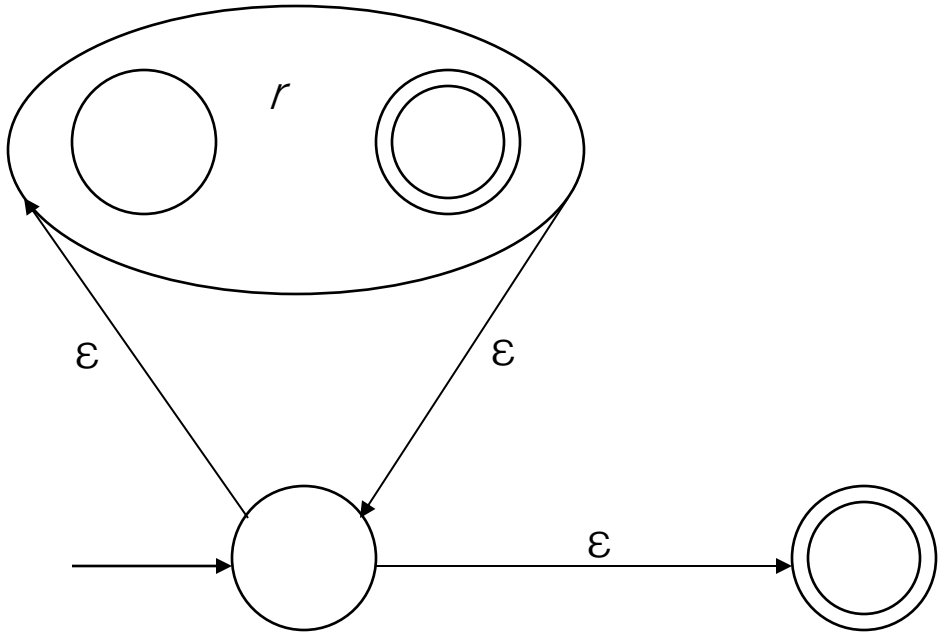
$r \mid s$

- **An ε-transition always connects two states that were, earlier in the process, the start and the accepting states of NFAs fro some component REs**



r^*

- **An ε-transition always connects two states that were, earlier in the process, the start and the accepting states of NFAs fro some component REs**

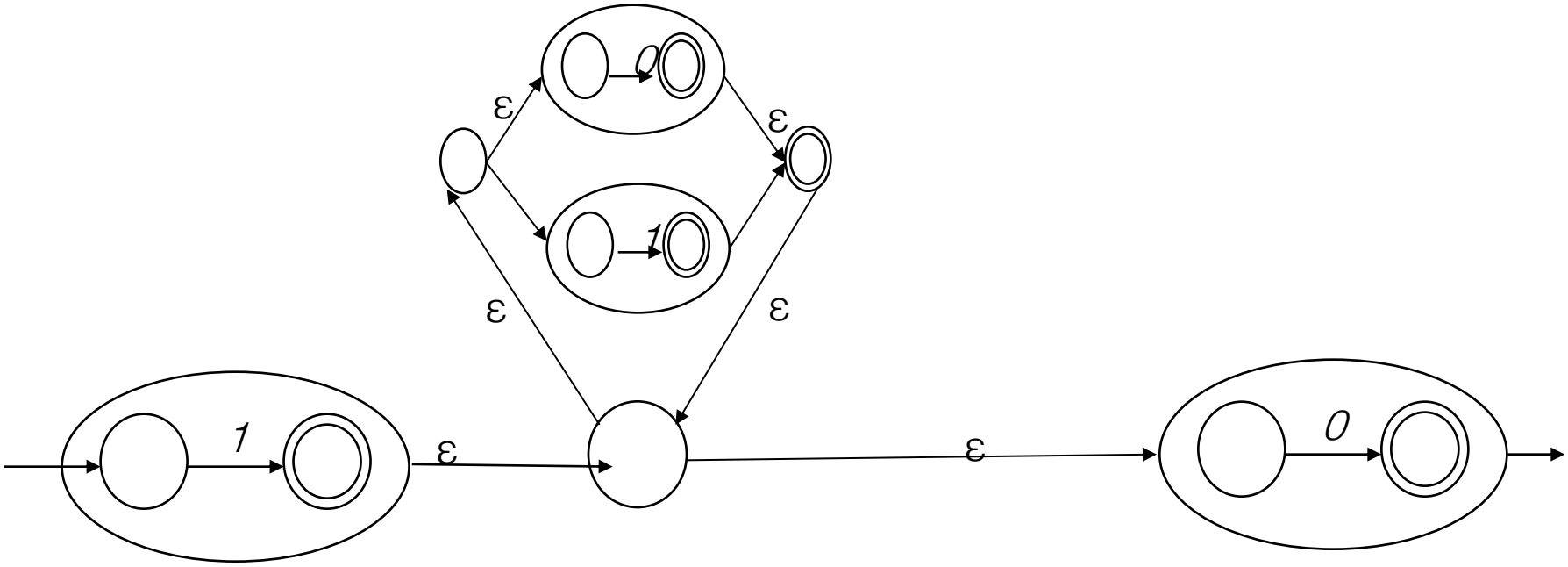


Example

- Convert the following RE into its equivalent DFA: 1(0|1)*0
- Approach one – Thompson construction
  - Concatenate expressions for 1, (0|1)* and 0

Example

- Convert the following RE into its equivalent DFA: 1(0|1)*0
- Approach one – Thompson construction
  - Concatenate expressions for 1, (0|1)* and 0

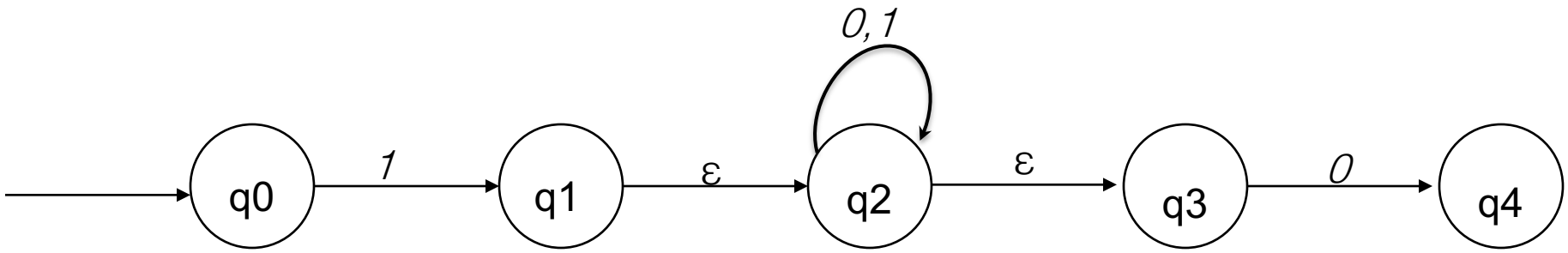


Example

- Convert the following RE into its equivalent DFA: 1(0|1)*0
- Approach one – Thompson construction
  - Concatenate expressions for 1, (0|1)* and 0
  - Simplify

Example

- Convert the following RE into its equivalent DFA: 1(0|1)*0
- Approach one – Thompson construction
  - Concatenate expressions for 1, (0|1)* and 0
  - Simplify



Example

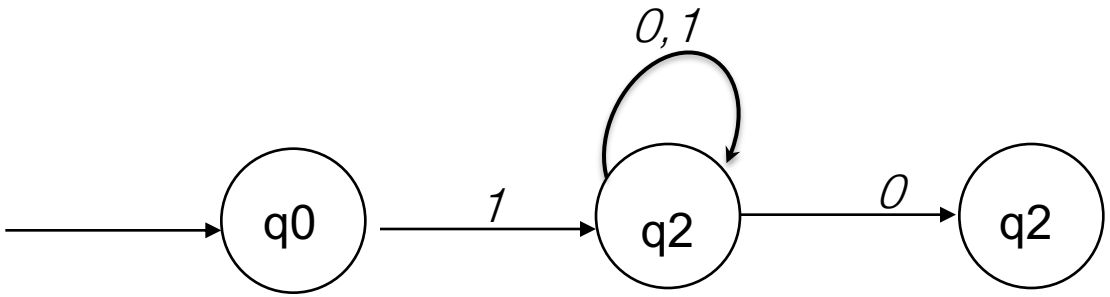
- Convert the following RE into its equivalent DFA: 1(0|1)*0

- Approach two – directly (no empty transitions)

Example

- Convert the following RE into its equivalent DFA: 1(0|1)*0

- Approach two – directly (no empty transitions)

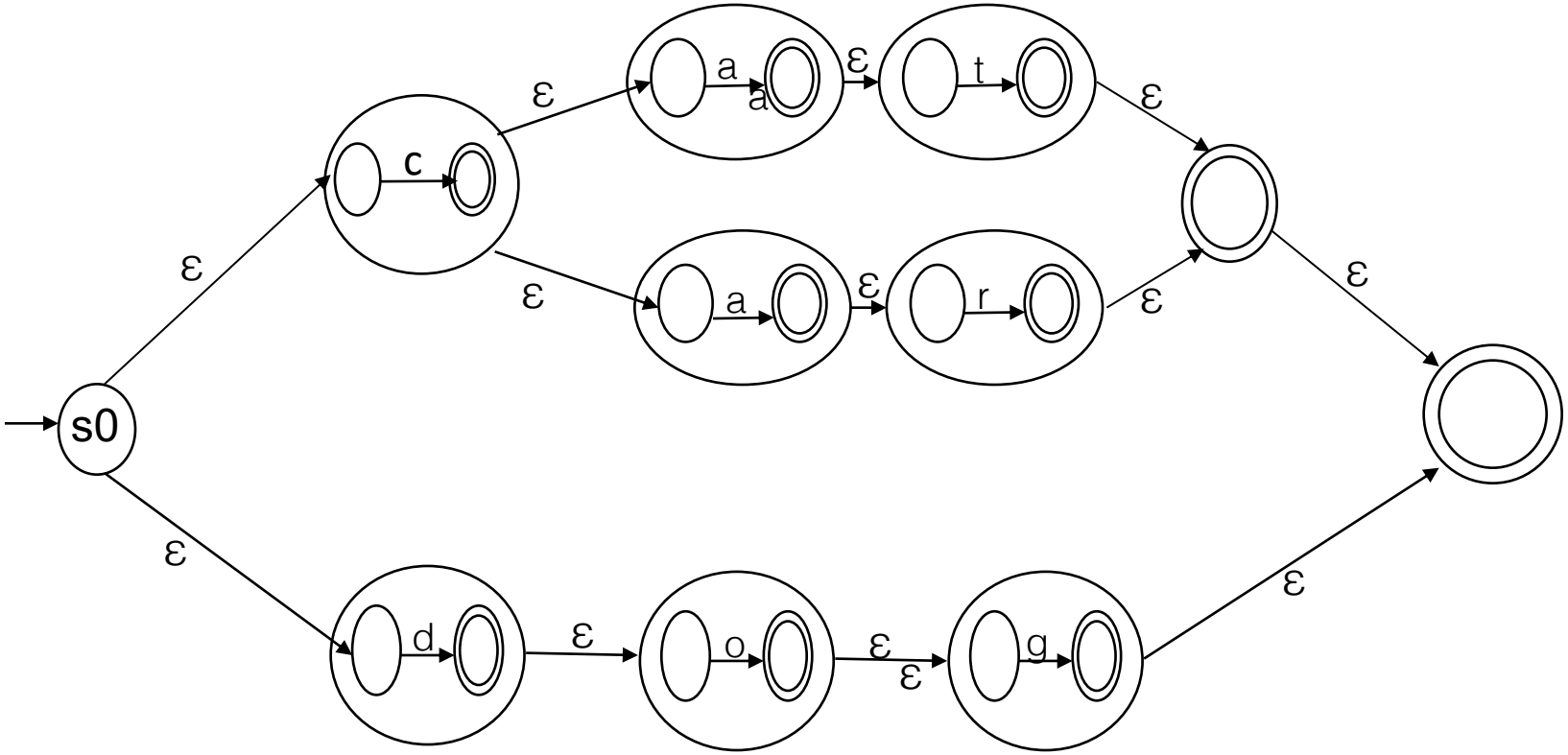


Exercise

- Draw the NFA for: c(at|ar) | dog

Exercise

- Draw the NFA for: c(at|ar) | dog



Removing Empty Transitions from Finite Automata

In there exists an empty transition between some nodes X and Y in an NFA, we can remove it as follows:

1. Find all the outgoing arcs from Y
2. Copy all these arcs, starting from X, without changing the arcs' labels
3. If X is an initial state, make Y also an initial state
4. If Y is a final state, make X also a final state

From NFA to DFA

From NFA to DFA

- Problem Statement:
  - Let $X = (Q_x, \Sigma, \delta_x, q_0, F_x)$ be an NFA which accepts some language L(X)
  - Design an equivalent DFA $Y = (Q_y, \Sigma, \delta_y, q_0, F_y)$ such that L(Y) = L(X), with respect to acceptance

From NFA to DFA

- Subset construction
  - Construct a DFA from the NFA, where each DFA state represents a set of NFA states
- Key idea
  - State of the DFA after reading some input is the set of all NFA states that could have reached after reading the same input
- Complex part of the algorithm: construction of the set of DFA states, *D*, from the NFA states, *N*, and the derivation of δ_{DFA}

From NFA to DFA

- Subset construction
  - Construct a DFA from the NFA, where each DFA state represents a set of NFA states
- Key idea
  - State of the DFA after reading some input is the set of all NFA states that could have reached after reading the same input
- Algorithm: example of a fixed-point computation
- If NFA has *n* states, DFA has at most 2^n states

  → DFA is finite, can construct in finite # steps
- Resulting DFA may have more states than needed
  - Solution: Hopcroft's Algorithm (DFA to Minimal DFA)

Fixed-Point Computation

- Fixed-point computation - computation characterized by an iterated application of a monotone function to some set of sets drawn from a known domain

- Some function f is monotone if:

$$\forall\, x, y \in D(f), x \leq y \rightarrow f(x) \leq f(y)$$

- Computation terminate when is reaches a state where further iteration produces the same answer—a "fixed point" in the space of successive iterates

From NFA to DFA

- Subset construction pseudocode

```
q0 → ε-closure({n0});
Q → q0;
   WorkList → {q0 };
while (WorkList ≠  empty) do
    remove q from WorkList;
for each character c in Σ do
    t → ε-closure(Delta(q, c));
   T[q, c] → t;
   if t ≠ Q then
       add t to Q and to WorkList;
   end;
end;
```

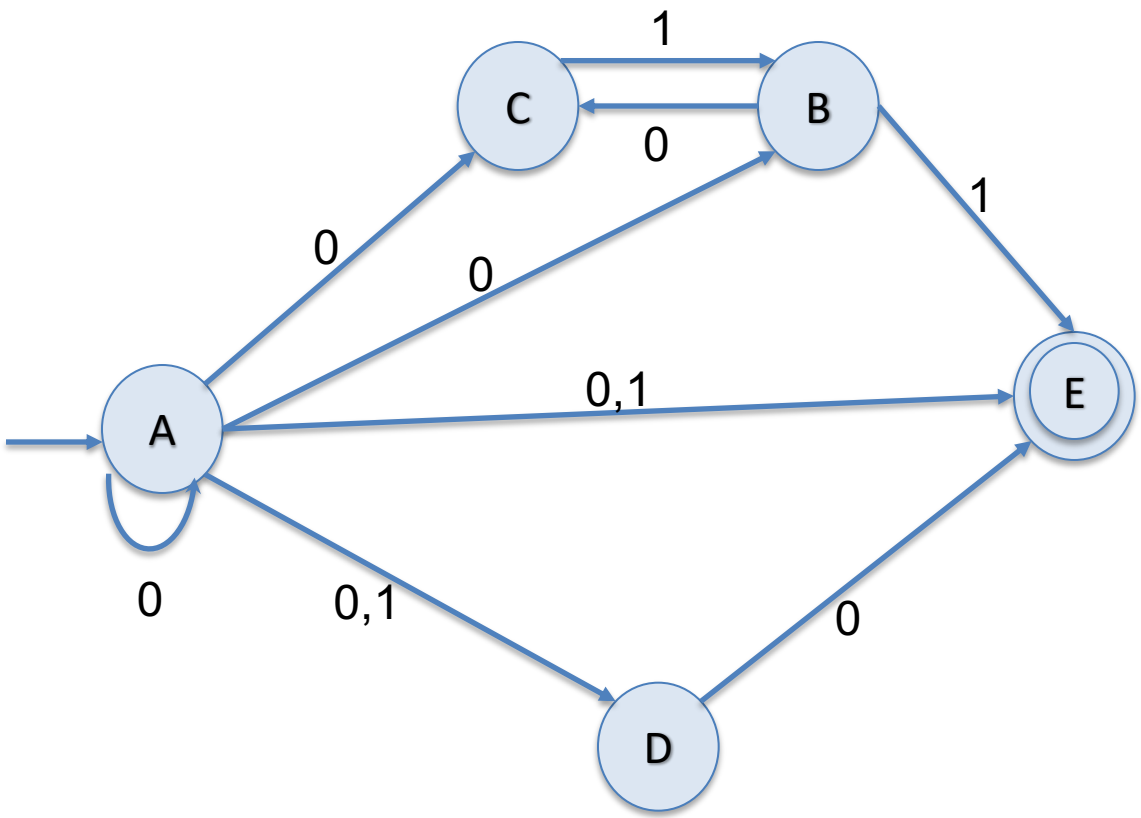
From NFA to DFA

- Subset construction algorithm:
  - Input: an NFA
  - Output: an equivalent DFA

1. Create state table from the given NFA
2. Create a blank state table under possible input alphabets for the equivalent DFA.
3. Mark the start state of the DFA the same as NFA, q_0
4. For every possible input, find the combination of states that can be reached from the current state. Those states form a new DFA state, {Q0, Q1,... , Qn}
5. Every time a new DFA state is generated under the input alphabet columns, repeate step 4 again, otherwise go to step 6
6. The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

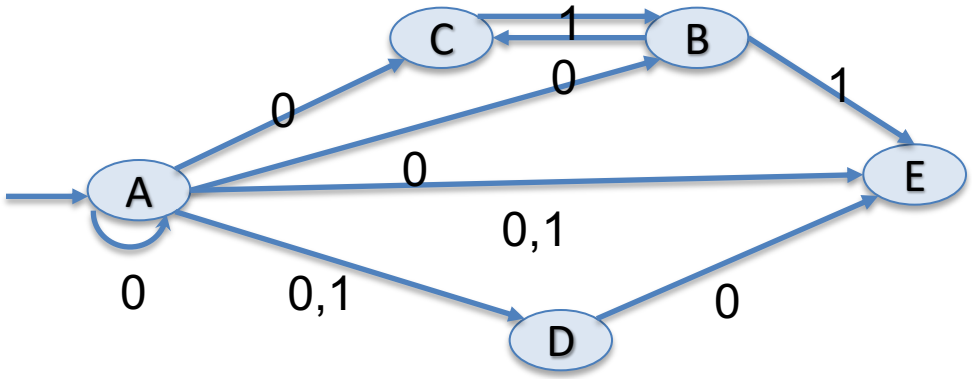
From NFA to DFA - Example

- Transform the given NFA into a DFA:
-



q	$\delta(q, 0)$	$\delta(q, 1)$
A	{A, B, C, D, E}	{D, E}
B	{C}	{E}
C	∅	{B}
D	{E}	∅
E	∅	∅

From NFA to DFA - Example

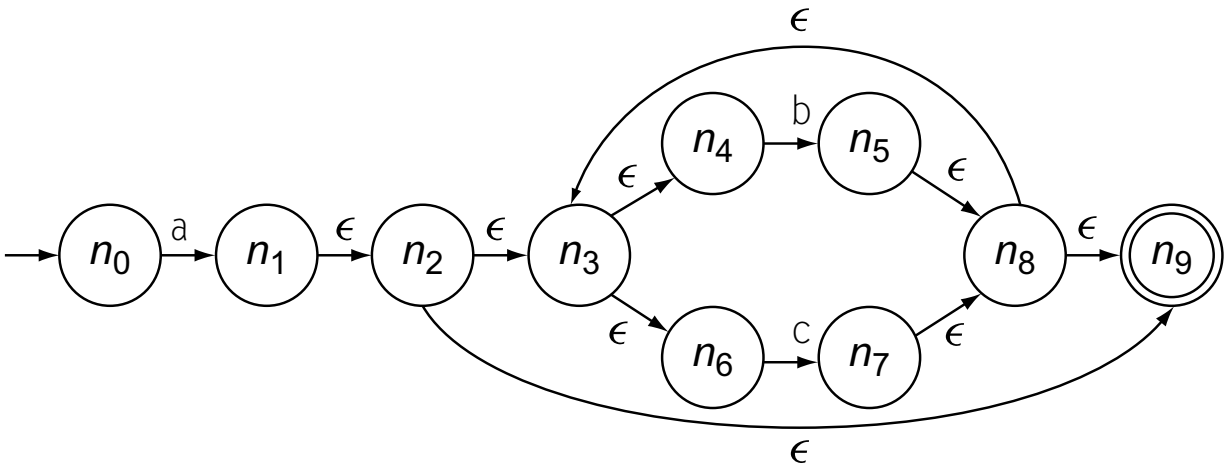


q	$\delta(q, 0)$	$\delta(q, 1)$
A	{A, B, C, D, E}	{D, E}
B	{C}	{E}
C	Ø	{B}
D	{E}	Ø
E	Ø	Ø

q	$\delta(q, 0)$	$\delta(q, 1)$
[A]	[A, B, C, D, E]	[D, E]
[A, B, C, D, E]	[A, B, C, D, E]	[B, D, E]
[D, E]	[E]	Ø
[B, D, E]	[C, E]	[E]
[E]	Ø	Ø
[C, E]	Ø	[B]
[B]	[C]	[E]
[C]	Ø	[B]

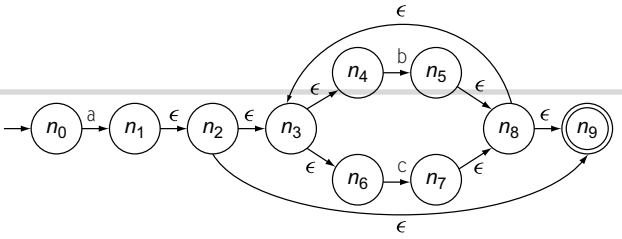
Exercise

- Build DFA for a(b|c)*, given the NFA



(a) NFA for "$a(b \mid c)^*$" (With States Renumbered)

Exercise



(a) NFA for "$a(b \mid c)^*$" (With States Renumbered)

- Build DFA for a(b|c)*, given the NFA

Set Name	DFA States	NFA States	ϵ-closure(Delta(**q**,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

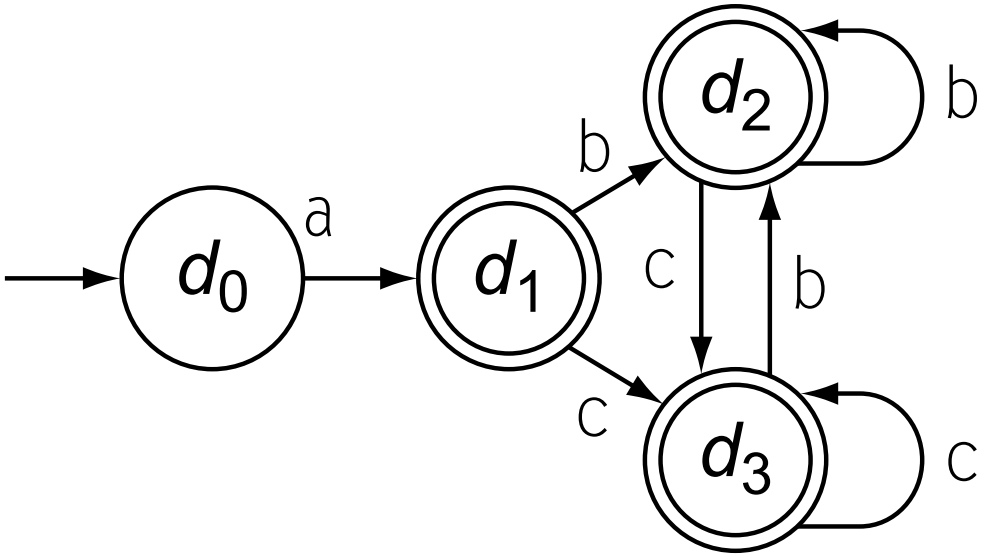
Exercise

- Build DFA for a(b|c)*, given the NFA

Set Name	DFA States	NFA States	ϵ-closure(Delta(**q**,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

Exercise

- Build DFA for a(b|c)*, given the NFA



(a) Resulting DFA

To Tokens

- A scanner is a DFA that finds the next token each time it is called
- Every “final” state of a DFA emits (returns) a token
- Tokens are the internal compiler names for the lexemes

  == becomes EQUAL

  (becomes LPAREN

  while becomes WHILE

  xyzzy becomes ID(xyzzy)
- You choose the names
- Also, there may be additional data … \r\n might count lines; token data structure might include source line numbers

DFA => Code

- Option 1: Implement by hand using procedures
  - one procedure for each token
  - each procedure reads one character
  - choices implemented using if and switch statements
- Pros
  - straightforward to write
  - fast
- Cons
  - a lot of tedious work
  - may have subtle differences from the language specification

DFA => Code [continued]

- Option 1a: Like option 1, but structured as a single procedure with multiple return points
  - choices implemented using if and switch statements
- Pros
  - straightforward to write
  - faster
- Cons
  - a lot of tedious work
  - may have subtle differences from the language specification

DFA => code [continued]

- Option 2: use tool to generate table driven scanner
  - Rows: states of DFA
  - Columns: input characters
  - Entries: action
    - Go to next state
    - Accept token, go to start state
    - Error
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”

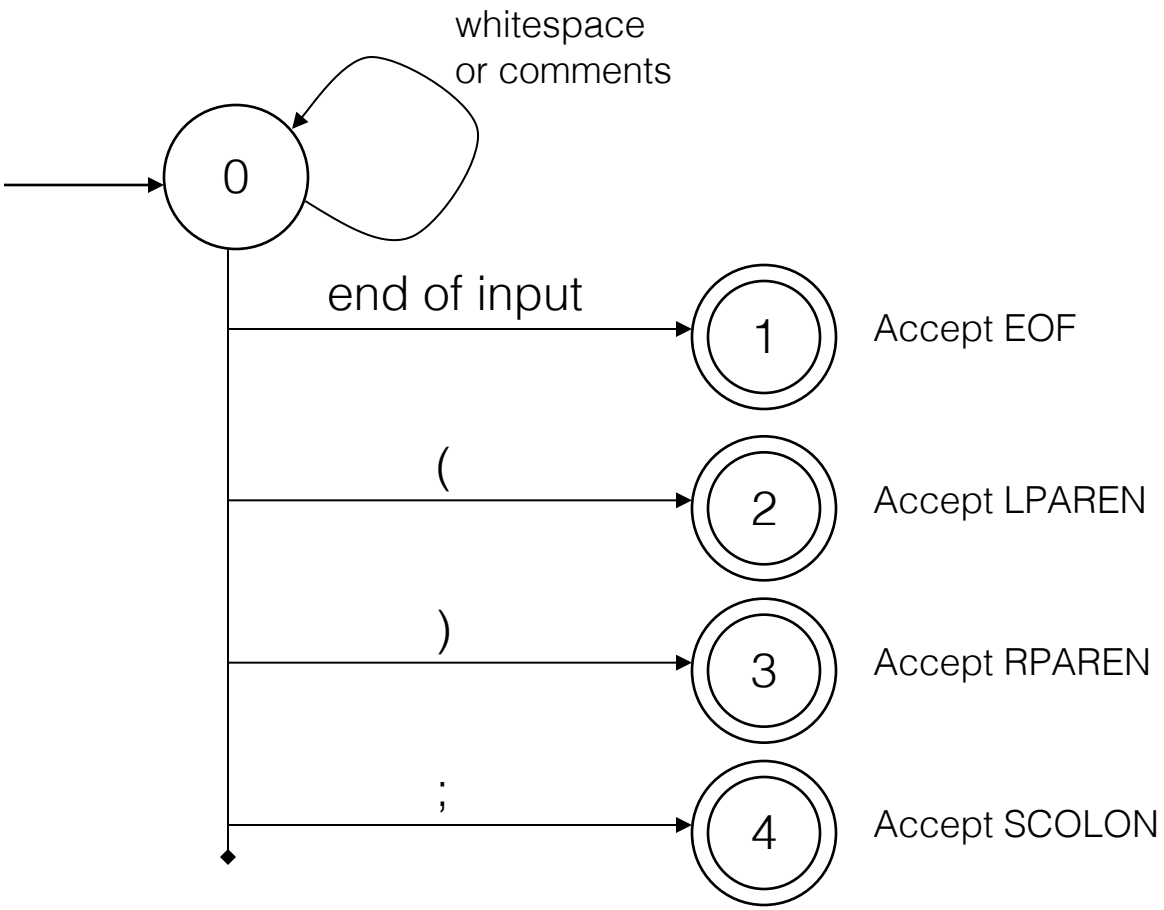
DFA => code [continued]

- Option 2a: use tool to generate scanner
  - Transitions embedded in the code
  - Choices use conditional statements, loops
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - "Magic"
  - Lots of code – big but potentially quite fast
    - Would never write something like this by hand, but can generate it easily enough

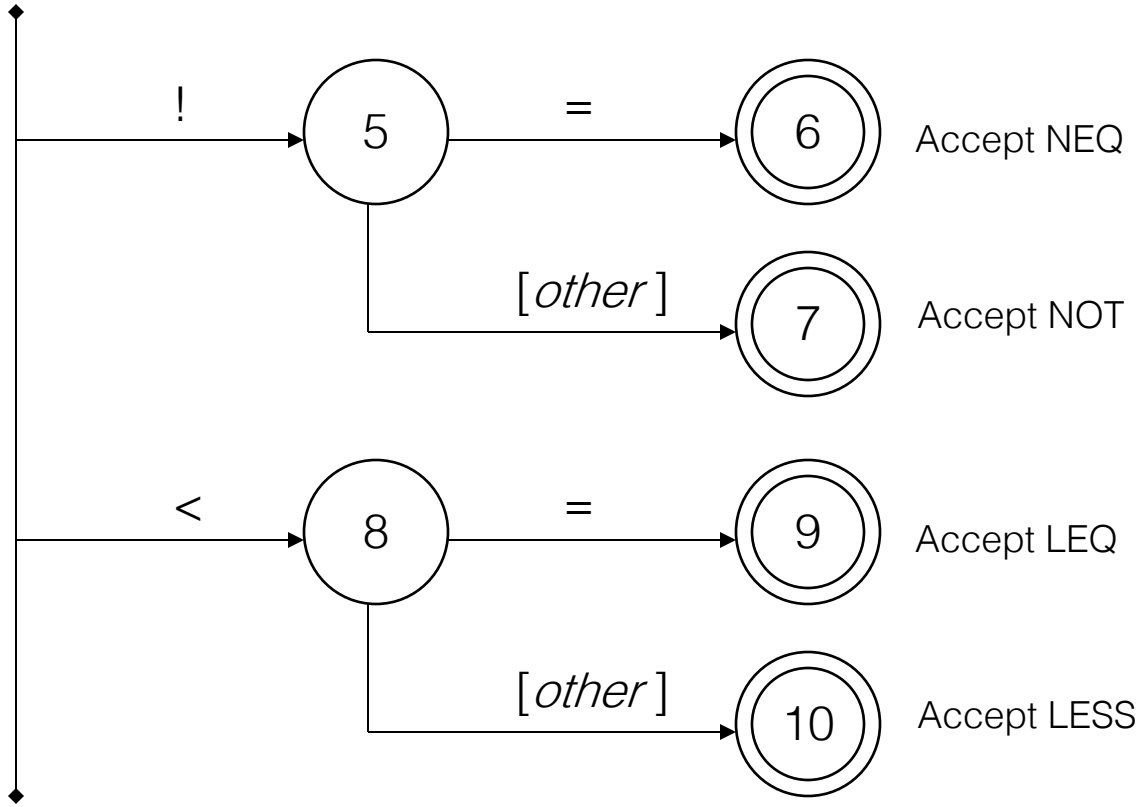
Example: DFA for hand-written scanner

- Idea: show a hand-written DFA for some typical programming language constructs
  - Then use to construct hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
  - Scanner stores current position in input
  - From there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token; save updated position for next time
- Disclaimer: Example for illustration only – you'll use tools for the course project
  - & we're abusing the DFA notation a little – not all arrows in the diagram correspond to consuming an input character, but meaning should be pretty obvious

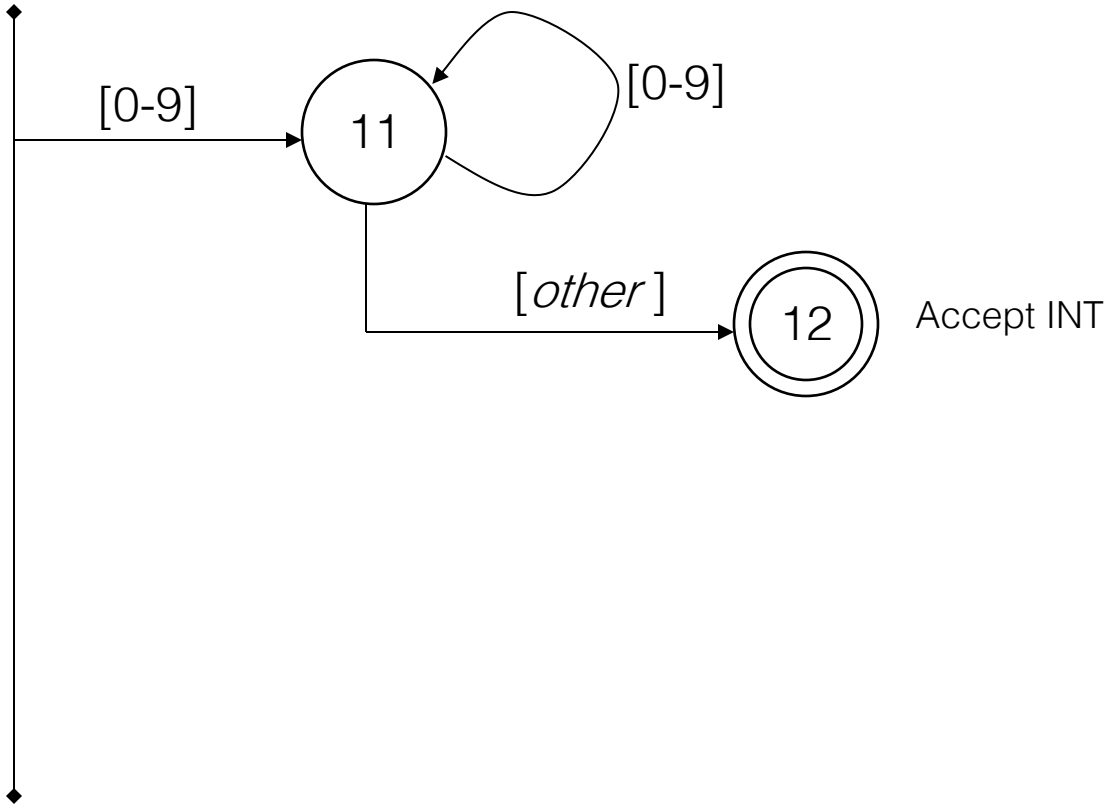
Scanner DFA Example (1)



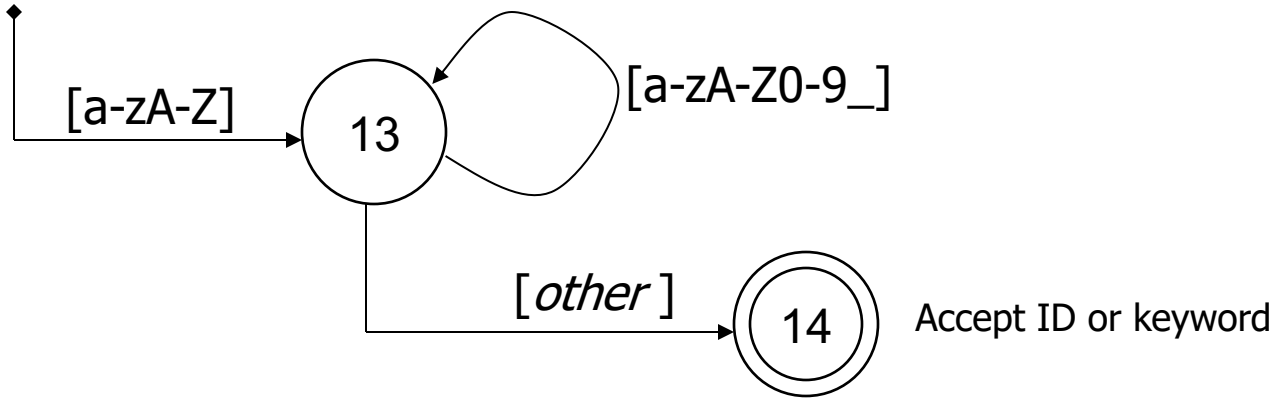
Scanner DFA Example (2)



Scanner DFA Example (3)



Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords
  - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - Machine-generated scanner: generate DFA will appropriate transitions to recognize keywords
    - Lots 'o states, but efficient (no extra lookup step)

Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure

```
public class Token {
  public int kind;                // token's lexical
  class
  public int intVal;  // integer value if class = INT
  public String id;       // actual identifier if class
  = ID
  // lexical classes
  public static final int EOF = 0;  // "end of file"
  token
  public static final int ID   = 1; // identifier, not
  keyword
  public static final int INT = 2;  // integer
  public static final int LPAREN = 4;
  public static final int SCOLN   = 5;
  public static final int WHILE   = 6;
  // etc. etc. etc. …
```

Simple Scanner Example

```
// global state and methods

static char nextch;  // next unprocessed input
   character

// advance to next input char
void getch() { … }

// skip whitespace and comments
void skipWhitespace() { … }
```

Scanner getToken() method

```
// return next input token
public Token getToken() {
  Token result;

  skipWhiteSpace();

  if (no more input) {
   result = new Token(Token.EOF); return result;
  }

  switch(nextch) {
   case '(': result = new Token(Token.LPAREN); getch();
   return result;
   case `)': result = new Token(Token.RPAREN); getch();
   return result;
   case `;': result = new Token(Token.SCOLON); getch();
   return result;

   // etc. …
```

getToken() (2)

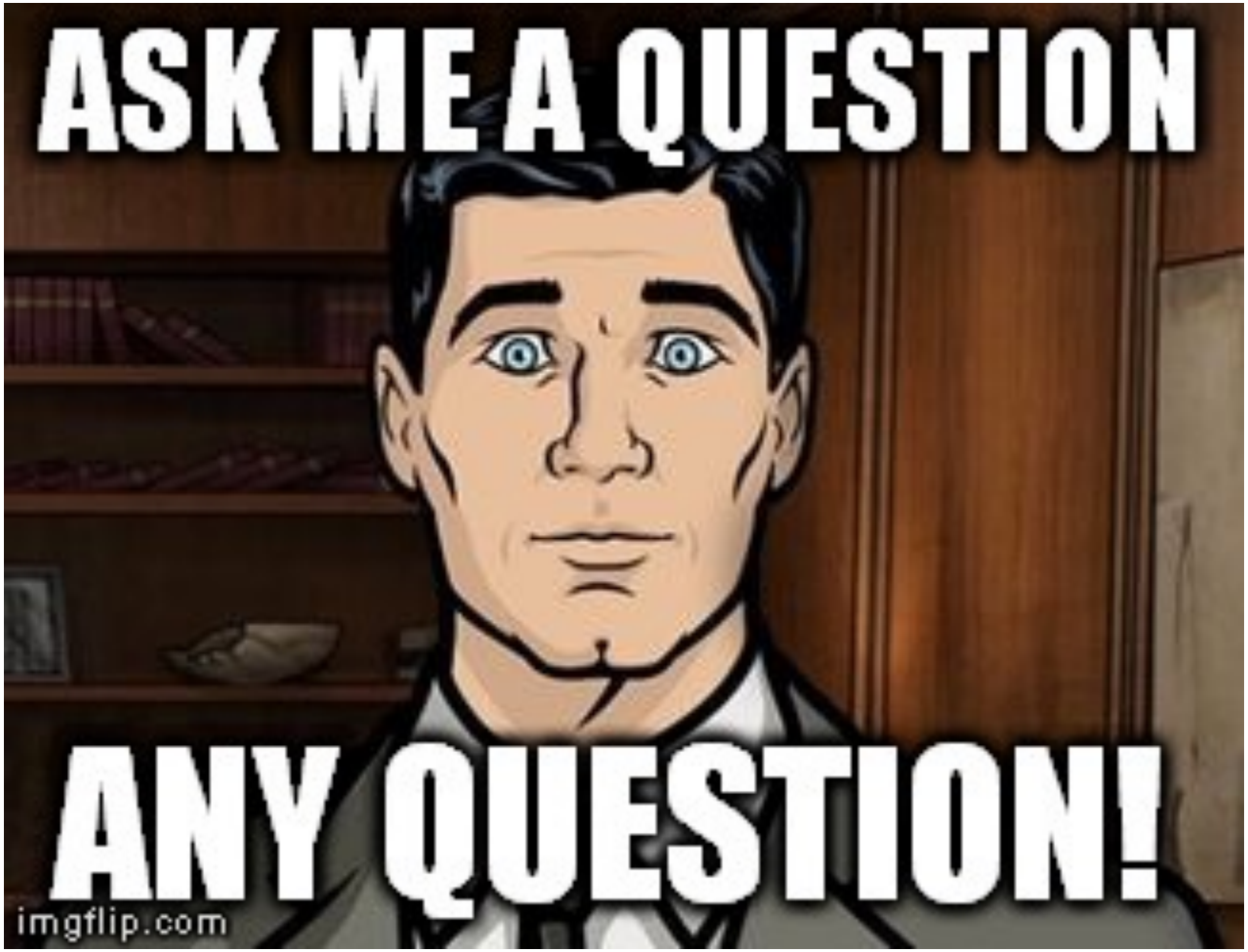
```
case '!': // ! or !=
    getch();
    if (nextch == '=') {
      result = new Token(Token.NEQ); getch(); return
result;
    } else {
      result = new Token(Token.NOT); return result;
    }

case '<': // < or <=
    getch();
    if (nextch == '=') {
      result = new Token(Token.LEQ); getch(); return
result;
    } else {
      result = new Token(Token.LESS); return result;
    }
// etc. …
```

getToken() (3)

```
 case '0': case '1': case '2': case '3': case '4':
 case '5': case '6': case '7': case '8': case '9':
    // integer constant
    String num = nextch;
    getch();
    while (nextch is a digit) {
       num = num + nextch; getch();
    }
    result = new Token(Token.INT,
Integer(num).intValue());
    return result;
…
```

getToken() (4)

```
case 'a': … case 'z':
case 'A': … case 'Z':  // id or keyword
 string s = nextch; getch();
 while (nextch is a letter, digit, or underscore)
{
    s = s + nextch; getch();
 }
 if (s is a keyword) {
    result = new Token(keywordTable.getKind(s));
 } else {
    result = new Token(Token.ID, s);
 }
 return result;
```



[Meme credit: imgflip.com]

Typical Tokens in Programming Languages

- Operators & Punctuation
  - + - * / ( ) { } [ ] ; : :: < <= == = != ! …
  - Each of these is a distinct lexical class
- Keywords
  - if while for goto return switch void …
  - Each of these is also a distinct lexical class (*not* a string)
- Identifiers
  - A single ID lexical class, but parameterized by actual id
- Integer constants
  - A single INT lexical class, but parameterized by int value
- Other constants, etc.

Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example:

  return maybe != iffy;

  should be recognized as 5 tokens

RETURN	ID(maybe)	NEQ	ID(iffy)	SCOLON

  i.e., != is one token, not two; “iffy” is an ID, not IF followed by ID(fy)