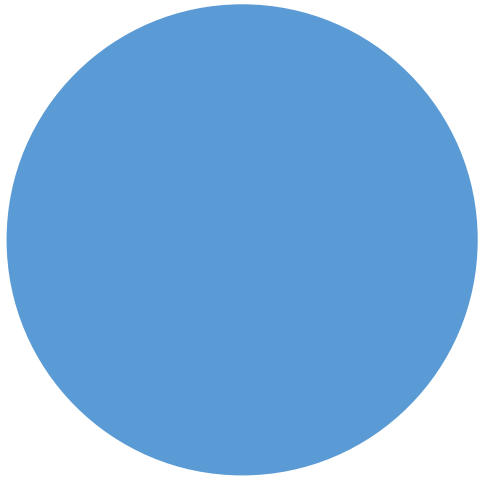
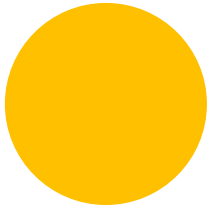



CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

LAST WEEK

- Regular Expressions
- Lexical specification using regular expressions

TODAY

- Towards an implementation
- Finite Automata
- Implementation of a DFA
- Regular Expression → DFA

Finite Automata

Recall, a finite automaton consists of:

1. input alphabet: Σ
2. set of states: *S*
3. a start state: *n*
4. set of final accepting states: $F \subseteq S$
5. set of transitions/edges: $state \rightarrow^{input} state$

The Idea

- transition $s_1 \rightarrow^a s_2$, in state s_1, on input a, go to state s_2
- at the end of the input: if in an accepting state $s \in F$, then ACCEPT
- at the end of the input: if not in an accepting state, $s \notin F$, then REJECT
- if we "get stuck" somewhere and cannot transition based on the next input, then REJECT

Drawing Out

Σ=0,1

Example: A finite automaton that accepts only "0”

Language of a finite automaton is the set of accepted strings.

Drawing Out

Σ=0,1

Example: A finite automaton that accepts any number of 0's followed by a single 1.

Example

Σ=0,1

Example: Given this finite automaton …

Which of the below languages is equivalent to the language denoted by the FA?

A. (0|1)*
B. (0|1)*00
C. (1*|0)(1|0)
D. 1*|(01)*|(001)*|(000*1)*

DFA: Deterministic Finite Automaton

DFA Rules

Rules

- No two edges leaving from the same state can have the same input.
- No ε moves.
  - (ε moves allowed in an NFA.)

Properties

- Takes only one path through the state graph per given input.
- Fast to execute.
- No choices to consider about which edge to follow from the current state.
- Just follow the input.

Stepping back…the big goal

1. Given a lexical specification...
2. …with regular expressions that represent the language for each token class...
3. ...create a deterministic finite automaton (DFA).

How? We'll see shortly.

Given Lexical Specification:

`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	REAL
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	*no token, just white space*
`.`	*error*

FIGURE 2.2. Regular expressions for some tokens.

Individual finite automaton for each lexical token:

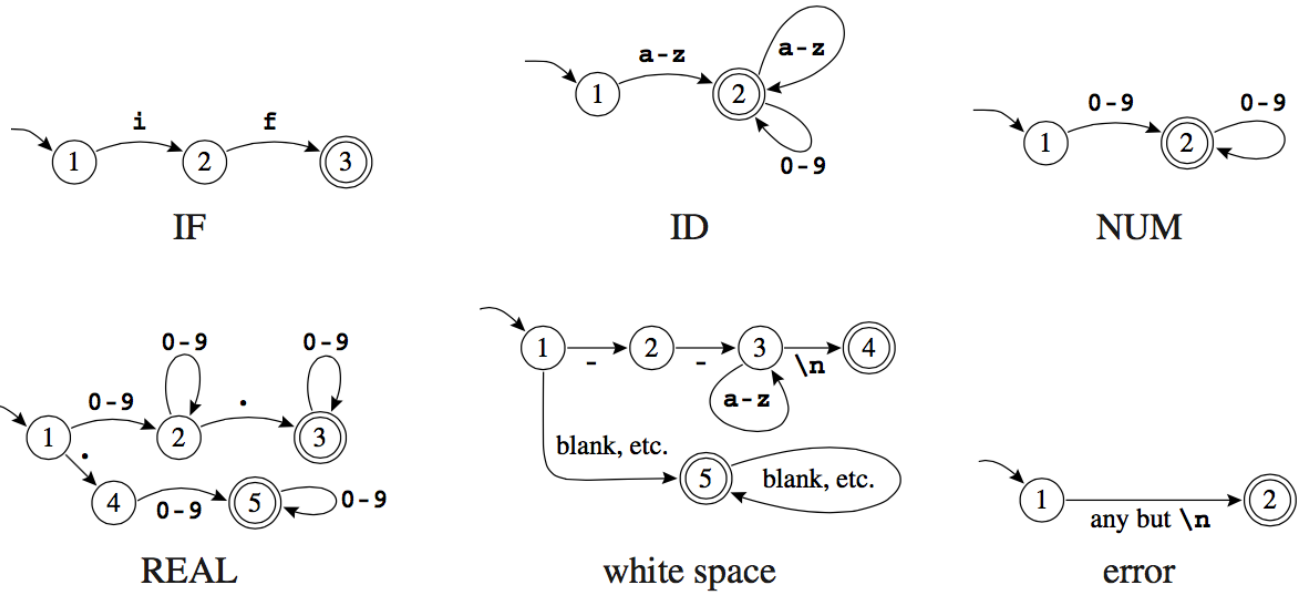


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

Each FA language is equivalent to the language of the regex.

We can cleverly combine the *separate* FAs into a *single* FA.

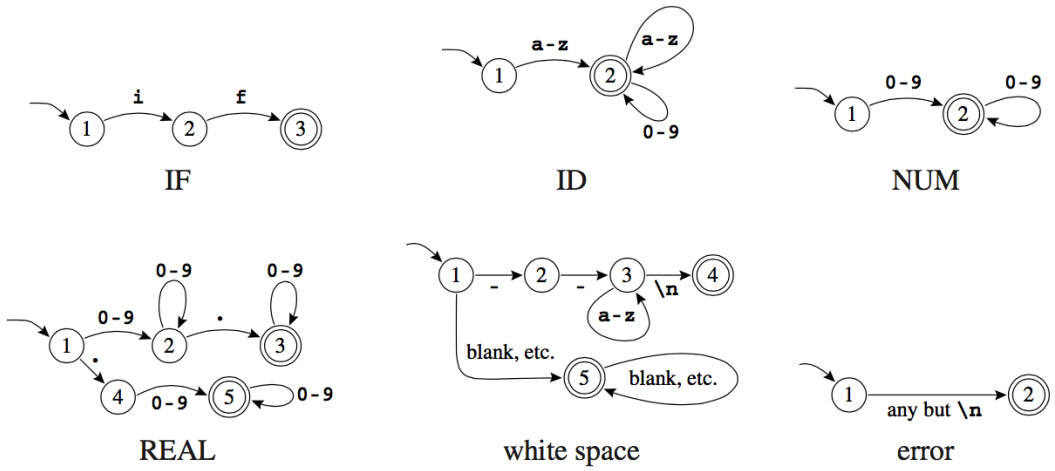


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

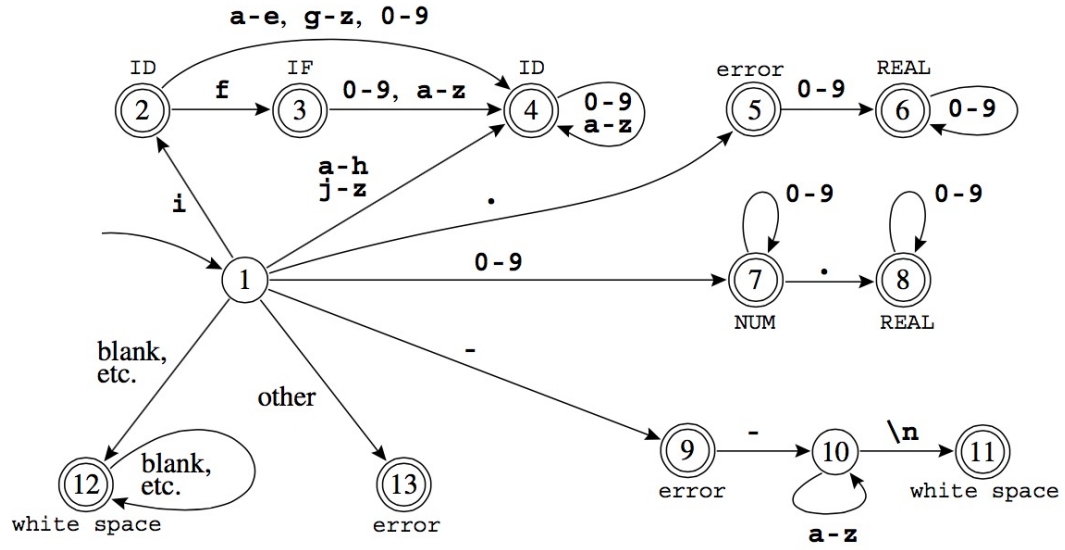


FIGURE 2.4. Combined finite automaton.

Implementation of DFAs

Implementing Finite Automata: Table-Driven Approach

Encode as a two dimensional transition table

- Rows: *states* - representing the current state
- Columns: *input* - representing the next input symbol
- For every transition: $s_j \rightarrow^a s_k$

```
int edges[][] = {  /* ···0 1 2···-···e f g h i j··· */
/* state 0 */     {0,0,···0,0,0···0···0,0,0,0,0,0···},
/* state 1 */     {0,0,···7,7,7···9···4,4,4,4,2,4···},
/* state 2 */     {0,0,···4,4,4···0···4,3,4,4,4,4···},
/* state 3 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 4 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 5 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 6 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 7 */     {0,0,···7,7,7···0···0,0,0,0,0,0···},
/* state 8 */     {0,0,···8,8,8···0···0,0,0,0,0,0···},
   et cetera
}
```

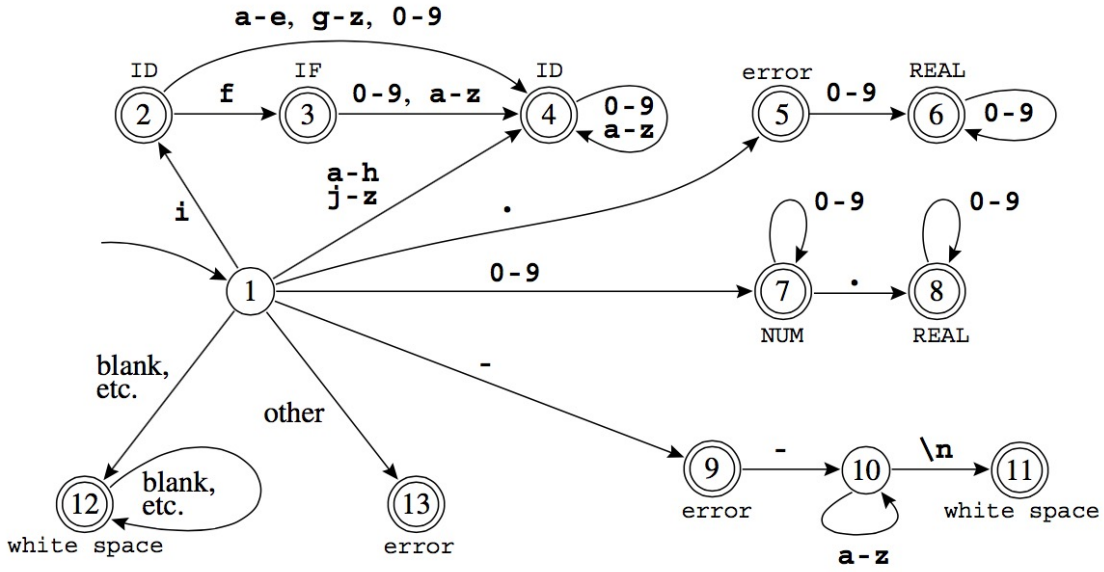


FIGURE 2.4. Combined finite automaton.

Pseudocode for Table-Driven Implementation of DFA

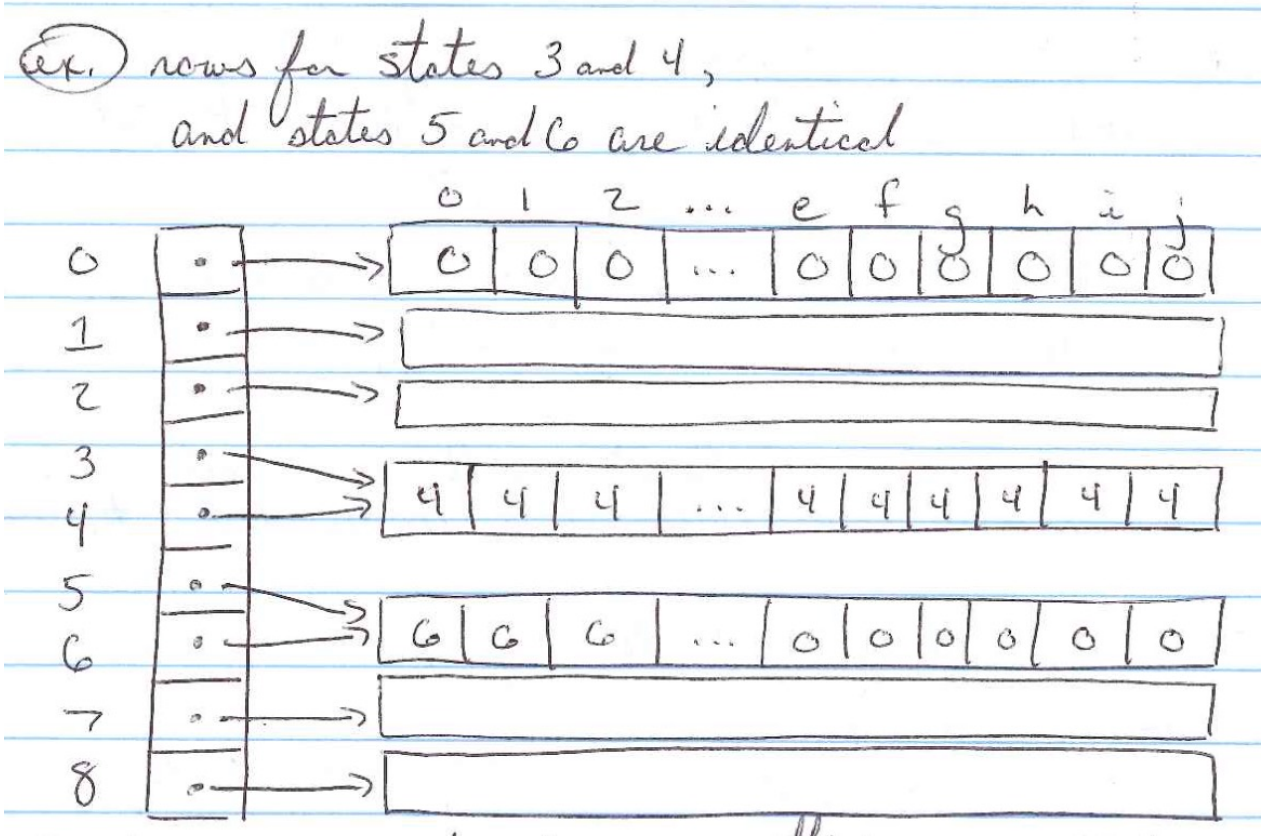
```
int i = 0;      // at position 0 in input string
int state = 1; // set state to the start state

while (input[i]) { // while there is more input
  // transition on the input,
  // ... set new state,
  // ... bump the input pointer position
  state = edges[state, input[i++]];
}
```

One approach: use a *finality array* if we successfully consumed all input.

```
String[] finalityAry = {"", "", "ID", "IF", ... };
```

A More Compact Implementation

Convert any rows that have identical values to pointers that point to a single representation of the "common-data" row.



How to recognize *all* tokens in a source program

- Recall the lexical specification rules.
  - Want to always fine the *longest match*

Input string: `if --not-a-comment`

What is the tokenization?

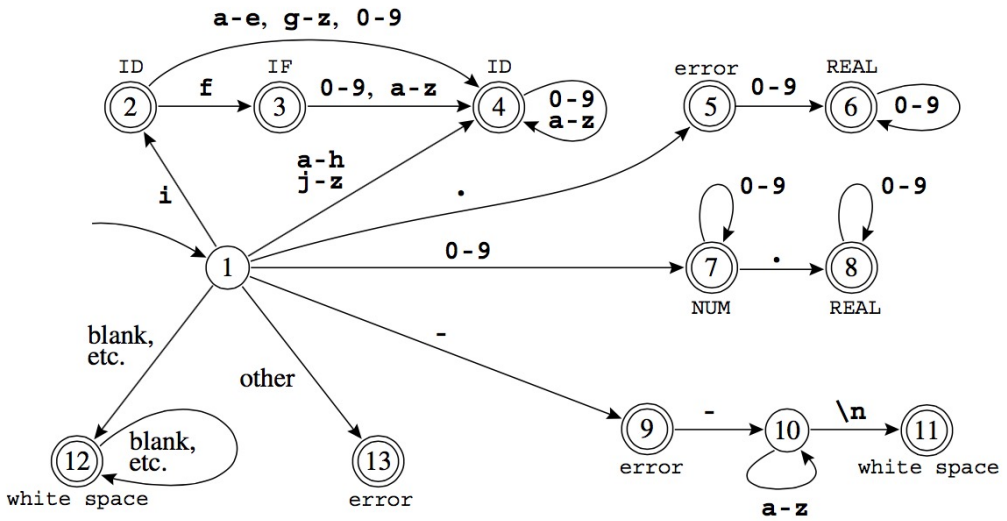


FIGURE 2.4. Combined finite automaton.

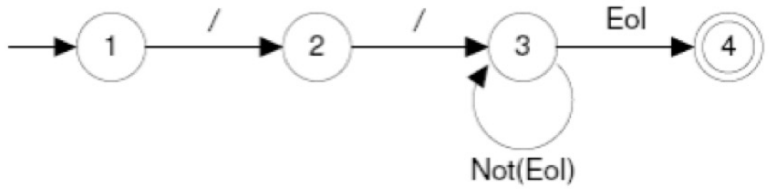
Last Final	Current State	Current Input	Accept Action
0	1	⊥if --not-a-com	
2	2	|i⊥f --not-a-com	
3	3	|if⊥ --not-a-com	
3	0	|if⊤ ⊥--not-a-com	*return* IF
0	1	if⊥ --not-a-com	
12	12	if| ⊥--not-a-com	
12	0	if| ⊤-⊥-not-a-com	*found white space; resume*
0	1	if ⊥--not-a-com	
9	9	if |-⊥-not-a-com	
9	10	if |-⊤-⊥not-a-com	
9	10	if |-⊤-n⊥ot-a-com	
9	10	if |-⊤-no⊥t-a-com	
9	10	if |-⊤-not⊥-a-com	
9	0	if |-⊤-not-⊥a-com	*error, illegal token '-'; resume*
0	1	if -⊥-not-a-com	
9	9	if -|-⊥not-a-com	
9	0	if -|-⊤n⊥ot-a-com	*error, illegal token '-'; resume*

FIGURE 2.5. The automaton of Figure 2.4 recognizes several tokens. The symbol | indicates the input position at each successive call to the lexical analyzer, the symbol ⊥ indicates the current position of the automaton, and ⊤ indicates the most recent position in which the recognizer was in a final state.

```
int currPos = beginTokPos = endTokPos = 0;
int currState, lastFinalState;

while (input[endTokPos]) {                      // while more input
    currentState = START_STATE;                 // start state = 1
    beginTokPos = currPos = endTokPos;
    while (currentState != 0) {
        currState = edges[currState, input[currPos++]];
        if (finalityAry[currState] != null) { // currState is a final state
            lastFinalState = currState;         // remember this state
                                                // and position where token ends,
                                                // for output later
            endTokPos = currPos - 1;
        }
    }
    // at this point, either have the longest match,
    // or a single char that we got stuck on
    if (lsatFinalState > 0)                     // longest match
        append( < finalityAry[lastFinalState], input.substring(beginTokPos, endTokPos + 1) > );
    else {                                      // single char
        endTokPos = currPos;
        append( < "ERROR", input.substring(beginTokPos, endTokPos + 1) > );
    }
}
return appended list
```

Another Implementation Approach: Explicit Control

(a)



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Transition Table

```
/⋆  Assume CurrentChar contains the first character to be scanned  ⋆/
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then  /⋆ Return or process the valid token ⋆/
else  /⋆ Signal a lexical error ⋆/
```

Figure 3.3: Scanner driver interpreting a transition table.

Explicit Control

```
/⋆  Assume CurrentChar contains the first character to be scanned  ⋆/
if CurrentChar = '/'
then
    CurrentChar ← READ( )
    if CurrentChar = '/'
    then
        repeat
            CurrentChar ← READ( )
        until CurrentChar ∈ { Eol, Eof }
    else  /⋆ Signal a lexical error ⋆/
else  /⋆ Signal a lexical error ⋆/
if CurrentChar = Eol
then  /⋆ Finished recognizing a comment ⋆/
else  /⋆ Signal a lexical error ⋆/
```

Figure 3.4: Explicit control scanner.

Table-Driven Approach

ADVANTAGES

- Size of code is reduced.
- Same code works for many different problems.
- Maintainability (code is easier to change).

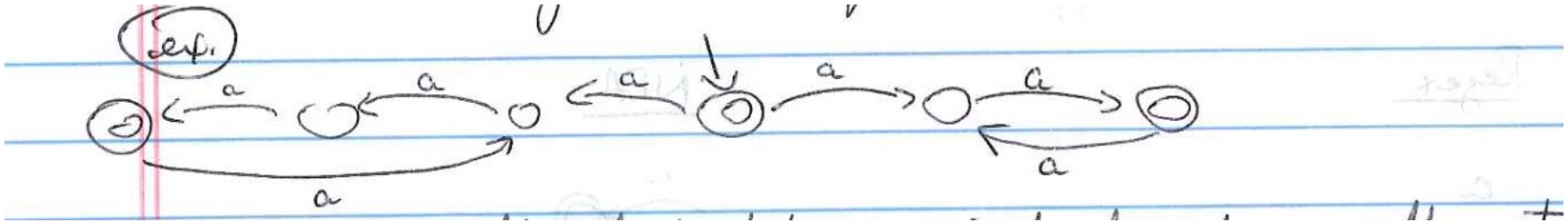
DISADVANTAGES

- May have very large tables, causing significant increase in the space used by the program.
- Thus, often rely on advanced table compression representations, such as sparse array representations.
- But there is a time penalty for compression.

NFA: Nondeterministic Finite Automata

Extends DFA's w/ potential *choice* of which edge to follow out of a state.

1. multiple edges have the same symbol
2. ε edges exist

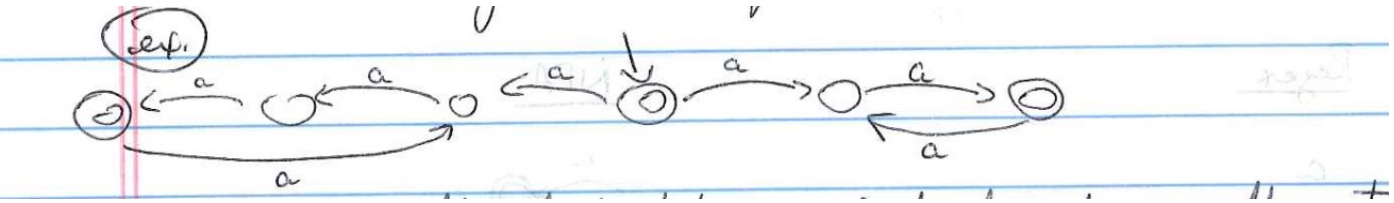


- In start state, on input character a, the automaton can move either right or left.
- If the *right* direction is chosen, <u>even length strings of a will be accepted</u>.
- What will be accepted if the *left* is chosen?

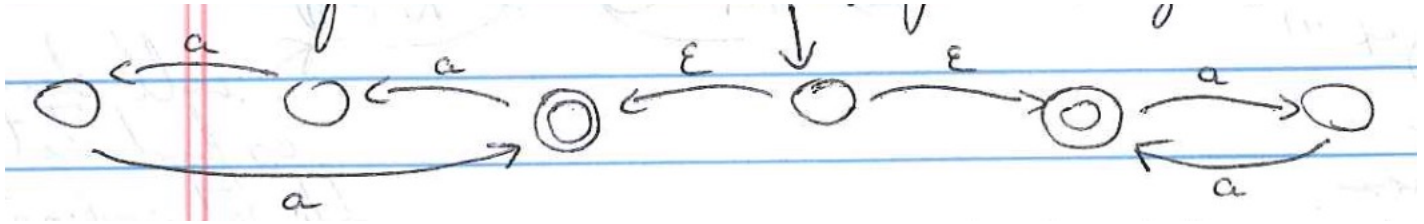
- In this example on the first transition, the machine first must "guess" which way to go.
  - ... thus, at first, NFAs may seem very practical.

Two equivalent NFAs

From previous slide



w/ epsilon edges



- Edges labeled with ε may be taken without using up a symbol from the input.
- Thus, more guessing is required.
- If *any* guess works, then some string is in the language of the machine.

Road Map

1. Convert regular expressions that define token classes into NFAs.
2. Then convert NFAs into DFAs.
3. We already discussed implementation of DFAs (previous slides).
4. *Motivation:* easier to convert regex's into NFAs than DFAs.

Converting a Regex into a NFA

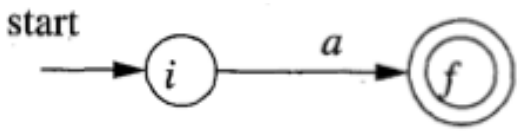
“Thompson’s Construction” (McNaughton-Yamada-Thompson Alg.)

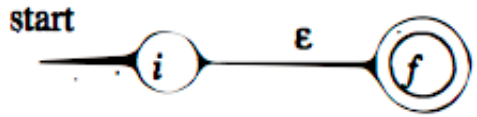
Turns each regex into an NFA with:

1. a *tail* (start edge)
2. a *head* (ending state)

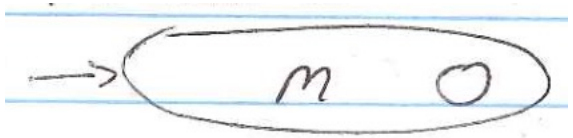
Will define a construction for each of the defined regular expressions (five of them).

The two *atomic* regular expressions are easy: (`a` and ε)



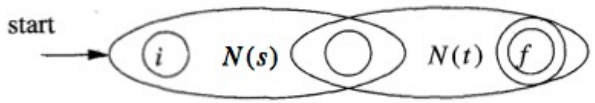


Generalizing, any regex `M` will have some NFA w/ a tail and a head.
It is sometimes drawn like this, with a tail and a head:

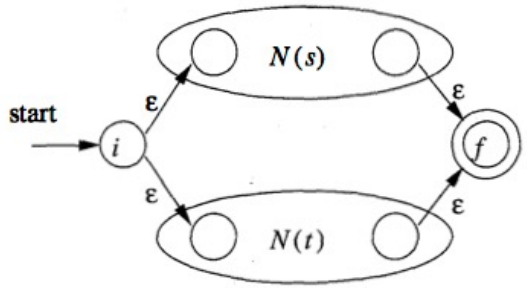


The three compounding operations `AB`, `A|B`, and `A*` can also be represented.

Concatenation:

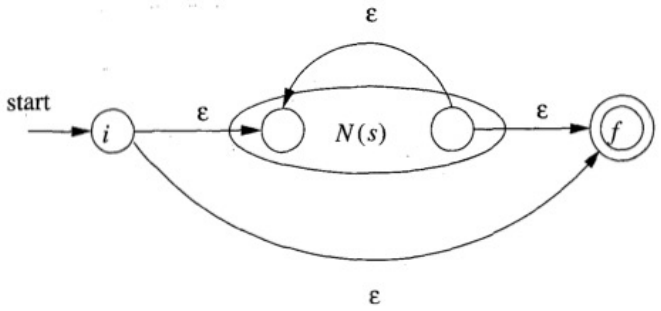


Alternation:



- Either the string will be in the language of `N(s)` or language of `N(t)`.
- ε-transitions are used to capture this.

Kleene Closure:



- Guarantees that "" (empty string) is in the language.
- Otherwise, make ε-transition to the start state of `N(s)`.
- From final state of `N(s)`, if we reach it, we can go back to the start state of `N(s)` (to handle the iteration).
- Can also transition from final state of `N(s)` to the final state of the machine.

Example

Given the regular expression: (0|1)*0

Build an equivalent NFA machine that recognizes the same language.

Converting a NFA into a DFA

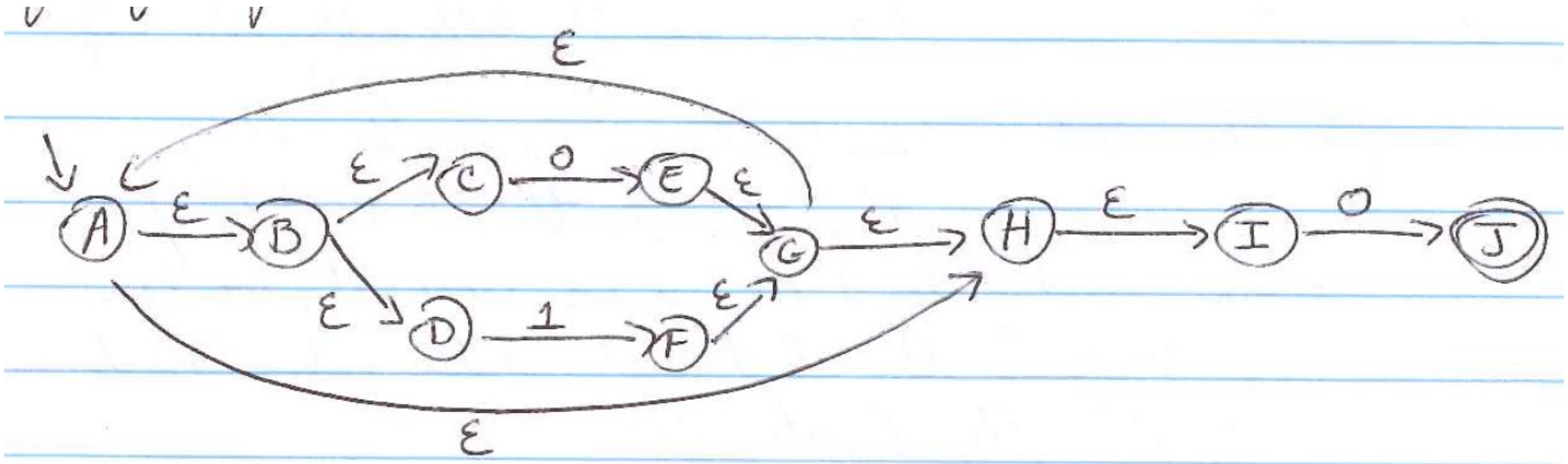
- Already looked at implementing DFAs.
- Implementing NFAs seem non-trivial because of the "guessing".
- *Idea:* Algorithmic approach to convert NFAs into DFAs.

Closure Function

ε-closure(X): set of all states that can be recursively reached by following any number of ε-transitions from the state X.

Example

ε-closure(X): set of all states that can be recursively reached by following any number of ε-transitions from the state X.



ε-closure(B) = {B, C, D}

ε-closure(G) = ???

An NFA can be in *many states* at any time.

- Seems potentially very bad.
- How many different states is the worse-case scenario?

BIG IDEA: Simulate the NFA by keeping track of the set of *all* states that the machine may be in.

- *Idea:* Each unique set becomes a state in the DFA.

Subset Construction Alg.

- *Idea:* D will be in state {x,y,z} after reading a given input string, if and only if, N could be in *any* of the states x, y, or z, depending on the transitions it chooses.
- D keeps track of all of the possible routes N might take and runs them simultaneously.

Need to define five things for each NFA:

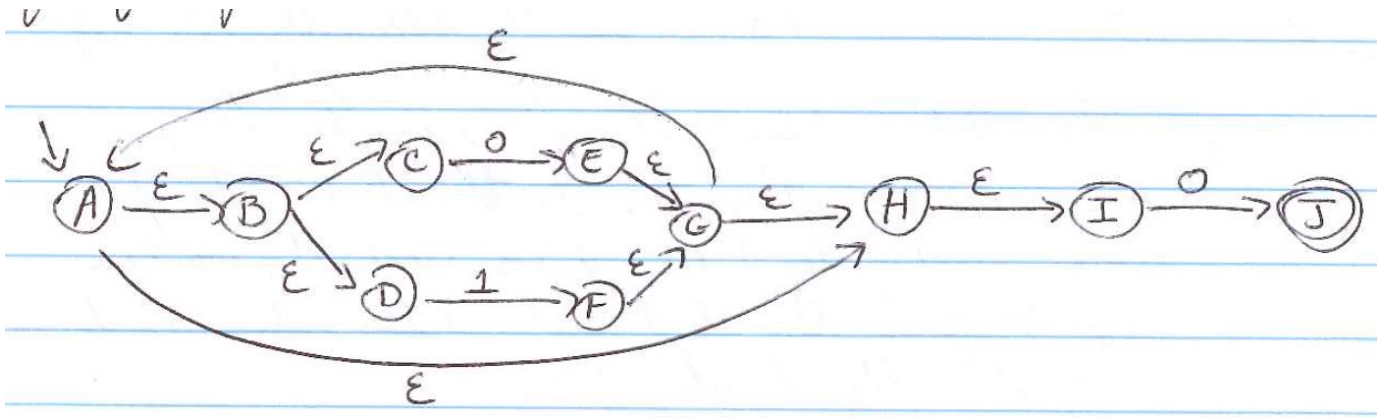
1. states: S
2. start state: $s \in S$
3. final states: $F \in S$
4. edge transition function: edge(s,c) = $\{t \mid s \rightarrow^{c} t\}$
   - set of all states reachable by following a single edge with label c from state s
5. ε-closure function

Need to define for DFA:

1. states: all possible combinations of S except the empty set
2. start state: ε-closure(s)
3. final states: {X | X ∩ F ≠ Ø }
4. edge transition function: Y = ε-closure (edge(X,c))

New machine is deterministic because we have finite states, start states, set of final states, transition fn (deterministic) and no more ε-transitions.

NFA → DFA Example



Idea: rather than enumerating all possible states, just going to enumerate the subsets that we actually need.

1. Begin w/ start state of NFA. Which additional states may the NFA initially be in without consuming any input?
2. What happens on each of the possible input values?
3. Compute ε closures along the way.
4. If any DFA set includes a final state from the NFA, that DFA state should be marked as final.
5. Keep repeating on possible input values as long as new subsets of NFA states are being computed.
6. Algorithm does not visit unreachable states of the DFA.

Next Week

Finish Lexer/Scanner!

Discussion of scanning tools used in practice.

HW #1 is OUT.