COMP 442/6421 Compiler Design

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TA: Zachary Lapointe zachary.lapointe@mail.Concordia.ca

LAB 2 – LEXICAL ANALYSIS

Lexer vs DFA

- Lexers and DFAs are not quite the same
  - A lexer combines many patterns into a single matching system, whereas a DFA operates on a single pattern
  - A lexer terminates/resets on a finished token, a DFA terminates on end of input
    - After a lexeme is found, the lexer has more input to process
    - Deciding *when* a token finishes is a not trivial problem
    - Examples:
      - `classy`
      - `public_x`
      - `01.23`
    - These are design issues, which are not easily solved by an algorithm!

Lexer vs DFA - design decisions

- **Prioritization**, which RegEx takes priority?
  - `classy`
    - `[class]:keyword  [y]:identifier`
      - Should keywords be prioritized?
      - Should this be allowed in the language?
    - `[classy]:identifier`
      - What if this were a typo?

Lexer vs DFA - design decisions

- **Permissiveness**, do you assume a user knows what they're doing?
  - `public_x`
    - `[public]:keyword  [_x]:invalid token`
      - Should keywords be prioritized?
      - What about context?
    - `[public_x]:identifier`
      - Allows more expressive identifiers
      - The possible number of lexical errors are diminished
      - Could it lead to confusion?

Lexer vs DFA - design decisions

- **Permissiveness**, do you assume a user knows what they're doing?
  - `01.23`
    - `[0]:integer [1.23]:float`
      - Is allowing this useful?
    - `[01.23]:invalid token`
      - Is this due to user error?

Lexer vs DFA - design decisions

- There are a lot of these decisions to make
  - Some of them are straightforward
  - Some involve trade-offs
    - The costs and benefits may only become clear in later stages of the compiler's design!
      - What if you realize you made a bad lexer choice while you're working on A3? . . .

Programming with Data - why?

- Motivations:
  - Separating *business logic* and *application logic*
    - When mixed, they exhibit *high coupling* and *low cohesion*
    - Often conceived by different people, or in different stages of application development
  - Applications which contain no business logic can be reused in any context
    - Libraries (ex: RegEx, STL containers, Java generics)
  - Changing business logic dynamically and easily
    - Bugs due to faulty business logic can be more easily found
    - A programs core functioning can be changed easily
      - A lexer that can tokenize multiple programming languages, by being supplied with different lexical specifications

Programming with Data - how?

- Literal separation:
  - Business logic is encoded differently
    - In non programming files (text, tables, i.e. any data format)
      - Simpler than programming
    - In another programming language
      - Interpreted languages (compilation delayed till runtime)
        - Plugins are frequently developed this way
      - Domain specific languages
    - In another application
      - Business rule management system (BRMS)
- Logical separation:
  - Specific architectures can encourage or enforce separation of business and application logic

Programming with Data - compilers

- Compilers have complex and changing business logic
  - Language updates
  - Complexity
    - Lexical specifications
    - Syntax specifications
    - Data types
    - Code generation
  - It may be productive to separate some of these . . .

A1 tools - handling RegEx and DFA

- https://regexr.com/
  - RegEx simulator web-tool
- http://ivanzuzak.info/noam/
  - JavaScript library for regular languages
- http://www.madebyevan.com/fsm/
  - simple and fast DFA drawing web-tool (small DFAs)
- https://cyberzhg.github.io/toolbox/
  - A set of web-tools for handling regular languages
    - In particular RegEx to DFA conversion

Making a simple lexer - Lexicon

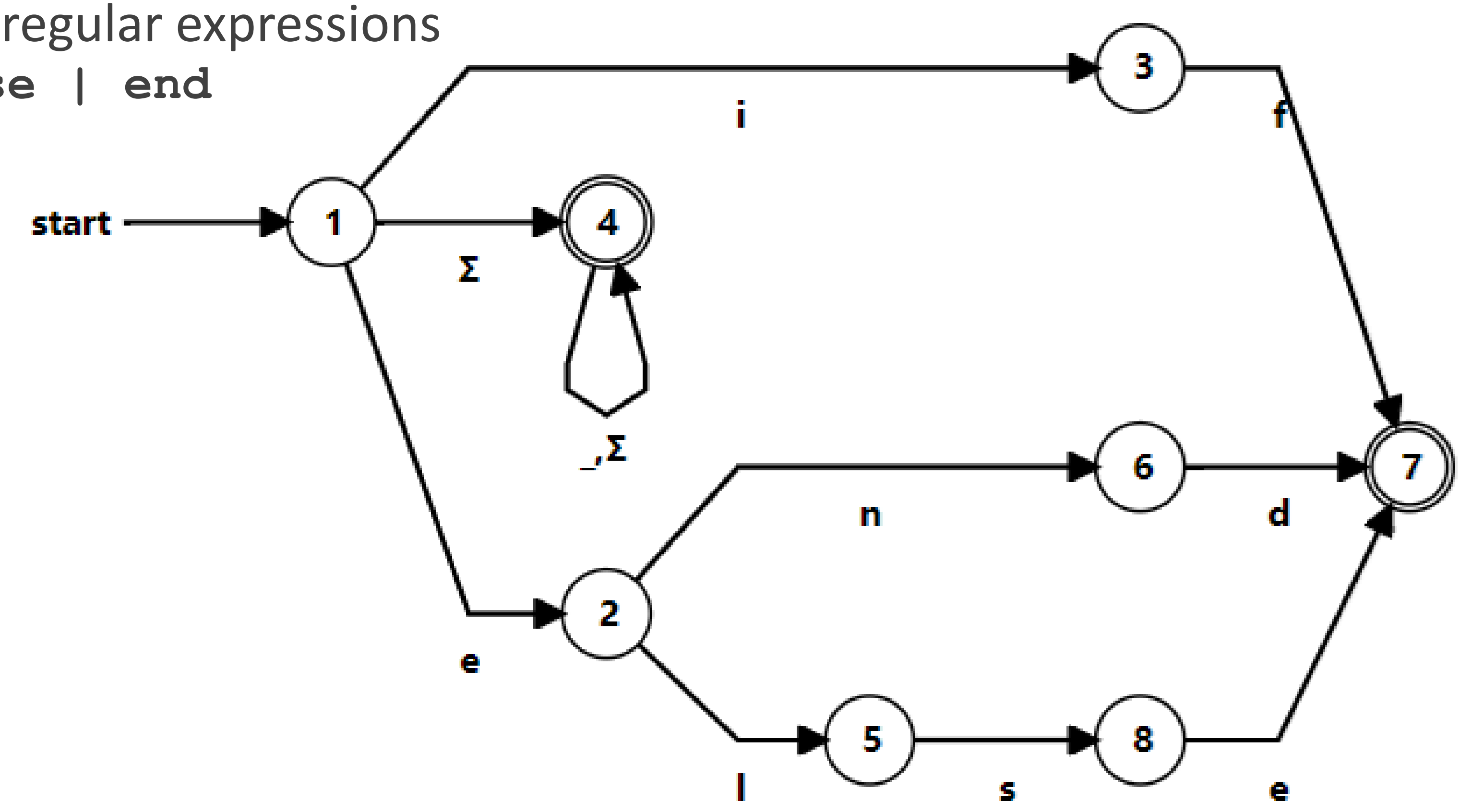
- We'll use a reduced version of the A1 specification

- ***id*** **::=** ***letter*** **[*letter* | _]***
- ***letter*** **::= a..z|A..Z**
- **Keywords:**
  - **if**
  - **then**
  - **else**
  - **while**

- **Valid characters: Σ = {a..z}U{A..Z}**

Making a simple lexer - RegEx

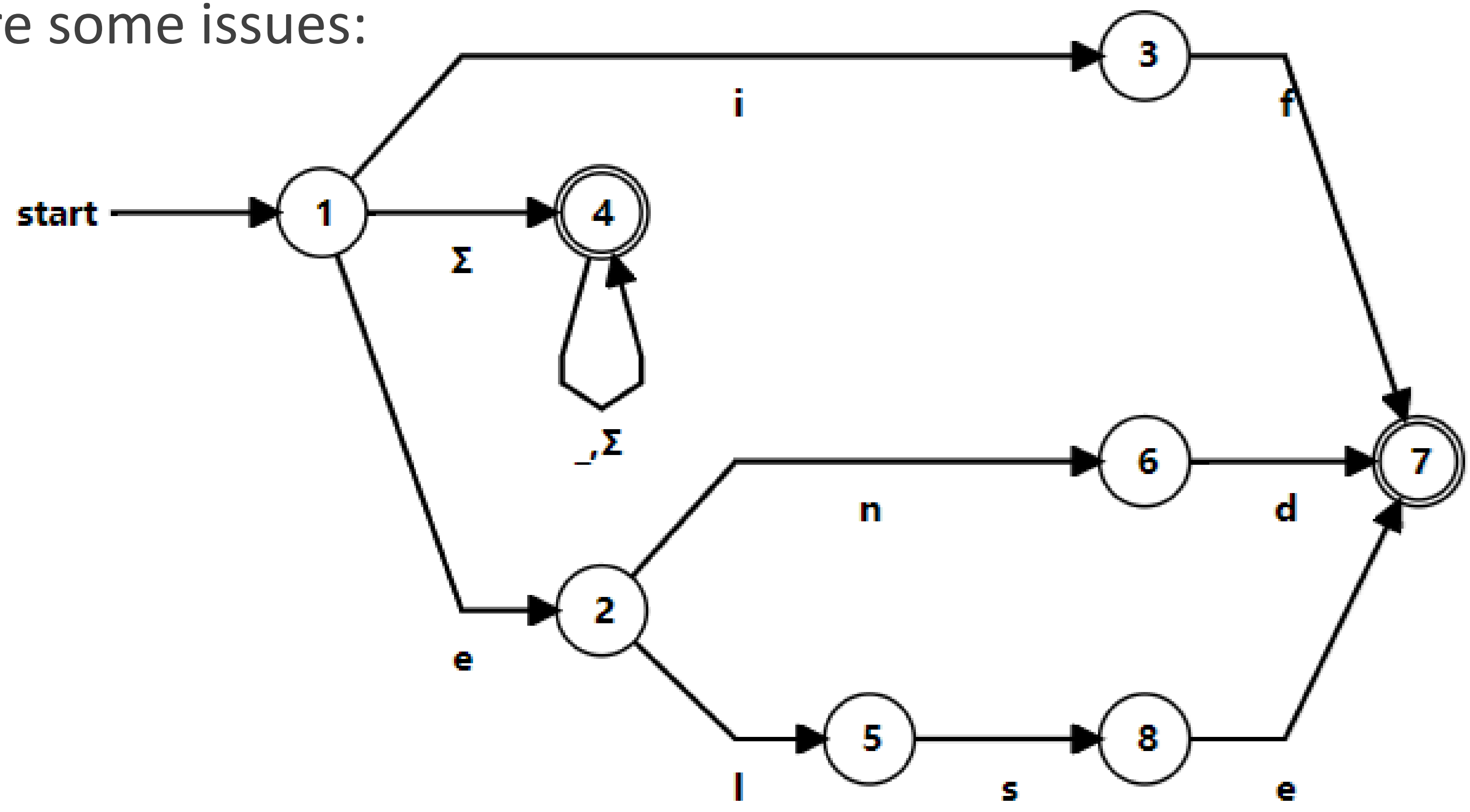
- Approach: combine all the regular expressions
  - `(Σ(Σ|_)*) | if | else | end`
- **Process into DFA**
  - Using *cyberzhg* tool



- Σ = {a..z}U{A..Z}

Making a simple lexer - Correcting DFA

- A good start, but there are some issues:
  - Ambiguities with letter input
  - Missing transitions
  - Only 2 final states
    - Which token is found?
  - What about backtracking?

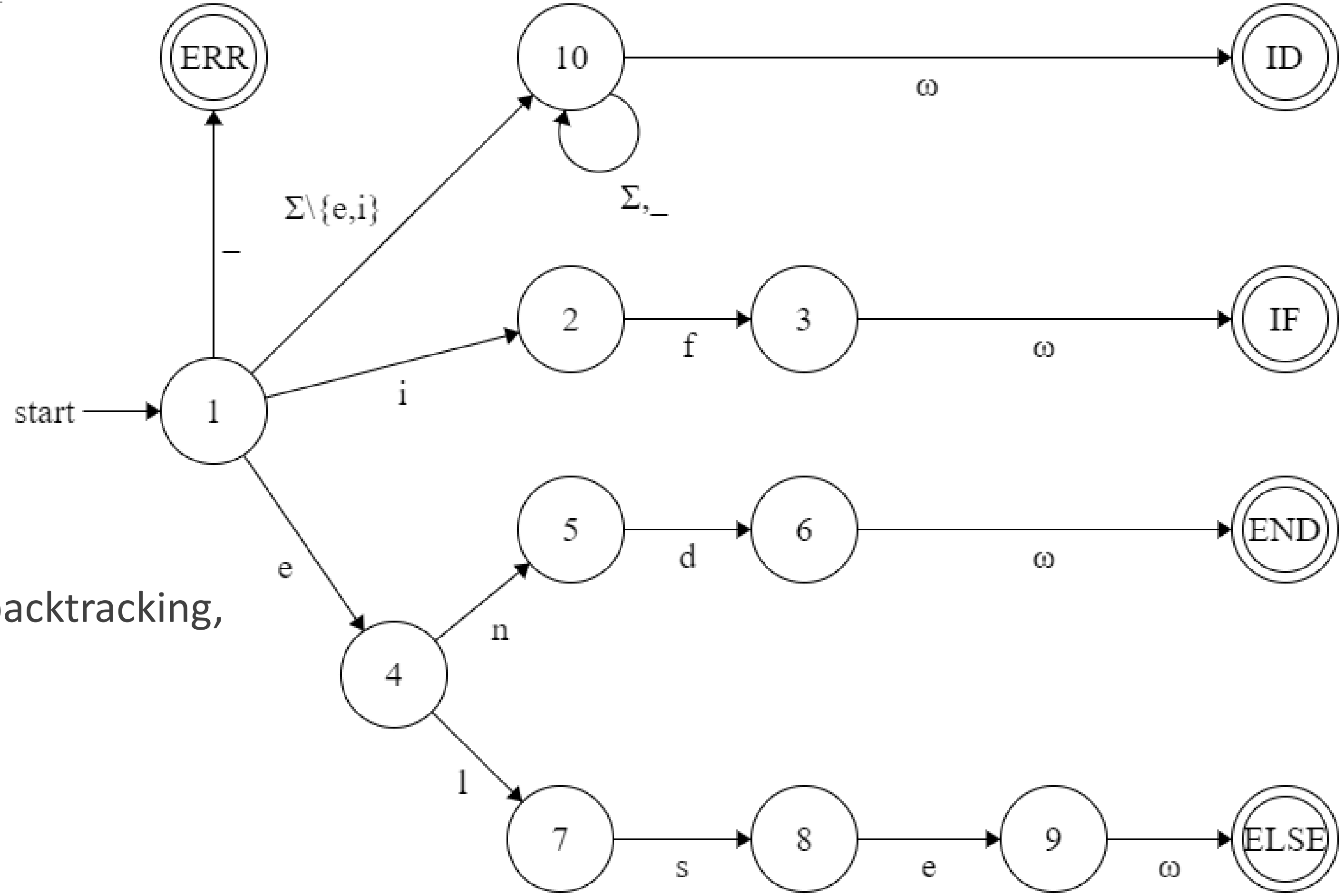


- Σ = {a..z}∪{A..Z}

Making a simple lexer - Correcting DFA

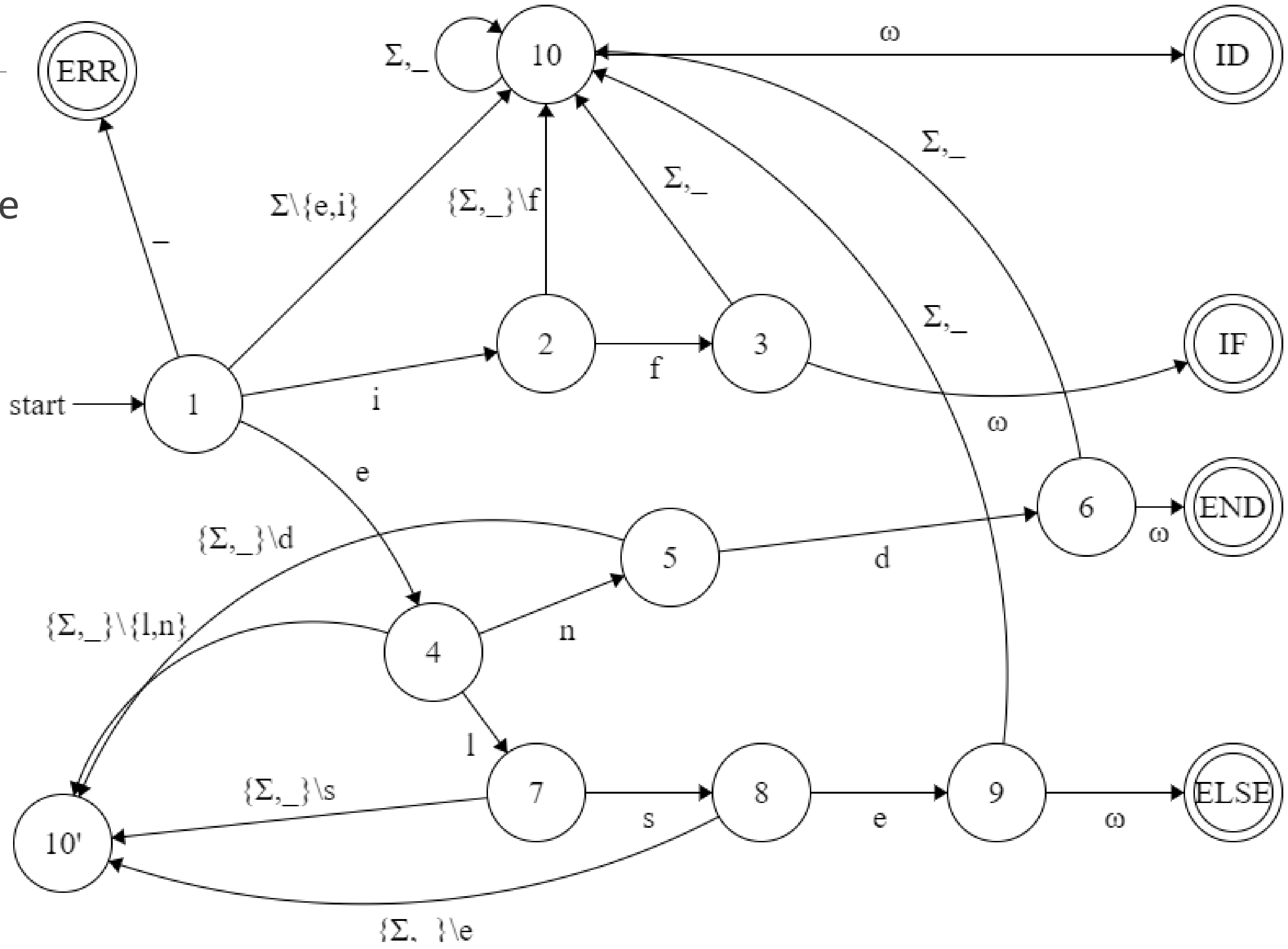
- Adding final states
  - These have no transitions, as they terminate the automata
  - Use special transition **ω**
    - Any character which guarantees the end of a lexeme
    - Here, whitespace fulfills the role
    - Maybe be different for different end states (<, +, :, etc.)
    - This character should be used for backtracking, if backtracking.

- `Σ = {a..z}U{A..Z}`



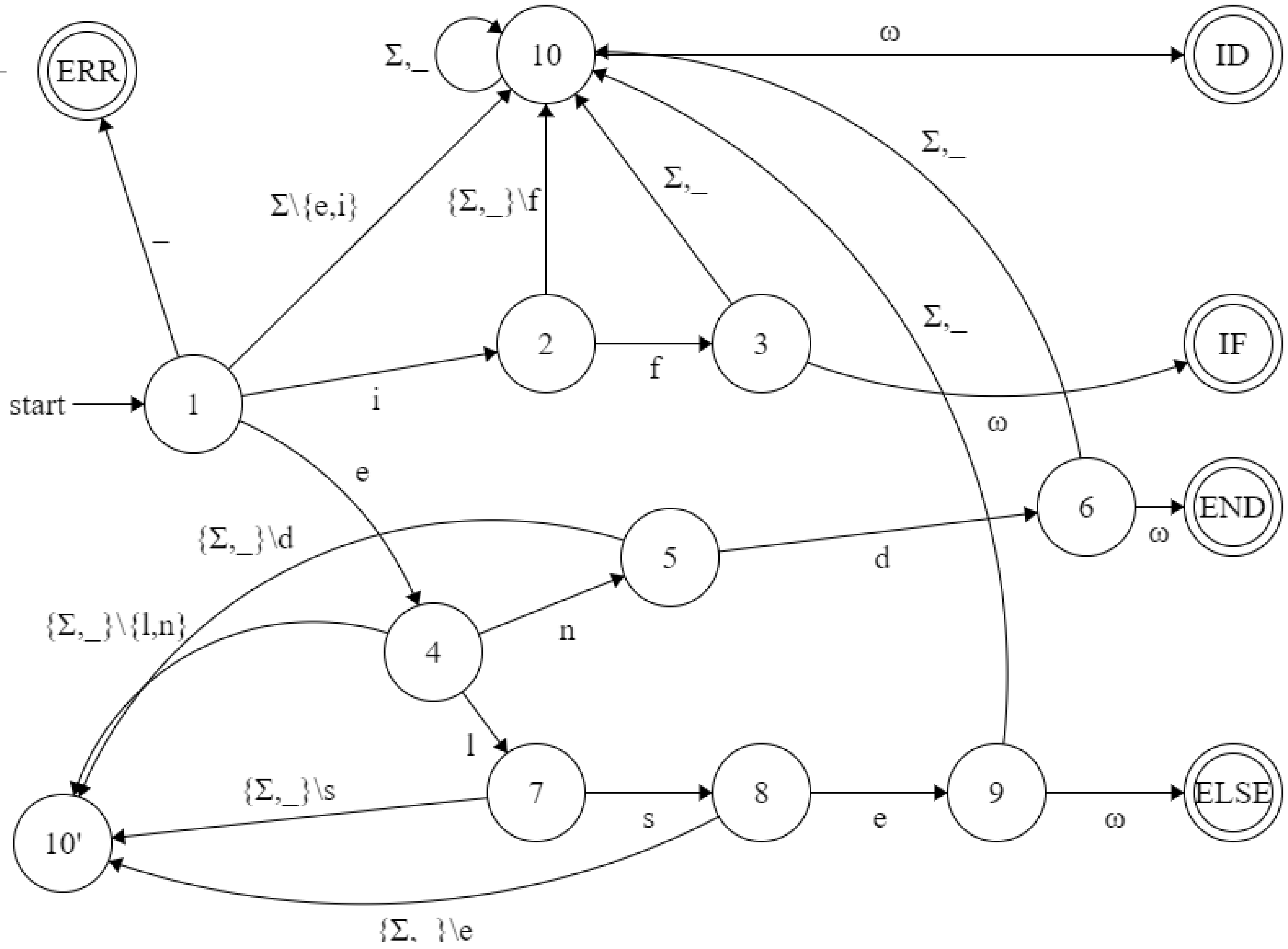
Making a simple lexer - Correcting DFA

- Adding missing transitions
  - DFA is deterministic, every state must have a unique transition for every possible input
  - Using set notation as a shorthand will aid with clarity and
    - Some sets might merit their own names:
      - Set of keyword starting letters
      - Set of letters not in keywords
- **`Σ = {a..z}U{A..Z}`**



Making a simple lexer – The result

- This DFA is complex!
  - It's only for a subset of the language!!
    - Staying organized is critical
  - Fortunately, it can be used as is, almost like a normal DFA
  - Table-based lexer
    - The table will be easier to manage than the DFA
  - Hard-coded lexer
    - Make sure your DFA is:
      - Easy to read
      - Easy to change, if necessary
      - You'll be referring to it a lot
- **Σ = {a..z}U{A..Z}**



Quick Review - Context-free Grammars

- A proper super-set of *regular languages*
  - *Context-free grammars*
  - *Deterministic context-free grammars*
    - A subset of context-free grammars
- Can be expressed using:
  - Production rule notation
  - Backus–Naur form
  - Push-down automata
- Two important subtypes, depending on uniqueness of derivations
  - Ambiguous
    - Non-deterministic
  - Unambiguous
    - Deterministic
- Some examples on the board . . .