Topic 9 – Finite Automata and Regular Expressions

Key Ideas

- the relationship between Finite Automata (FA's) and Regular Expressions (RE's)
- equivalence of Regular Expressions (RE), DFA's, NFA's and ε-NFA's
- extensions to regular expressions
- maximal munch
- scanning pseudocode

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Recall: Finite Automata and Regular Expressions

Examples

Create a DFA and a Regular Expression for each language.

Σ ={a, b, c, r}, $\mathcal{L}_1$ = {cab, car, carb}

Σ ={a}, $\mathcal{L}_2$ = {w: w contains an even # of a’s}

Σ ={a, b}, $\mathcal{L}_3$ = {w: w contains an even # of a’s}

Recall: Finite Automata and Regular Expressions

Examples

Create a DFA and a Regular Expression for each language

Σ ={a, b}, $\mathcal{L}_1$ = {w: w contains either aa or bb}

Σ ={a, b}, $\mathcal{L}_2$ = {w: w contains no occurence of aa or bb}

Regular Expressions

Recursive Definition

The elements (base cases) of a regular expression are

- $\varnothing$ i.e. $\mathcal{L} = \{ \}$
- ε i.e. $\mathcal{L} = \{ \varepsilon \}$
- a where $a \in \Sigma$ i.e. $\mathcal{L} = \{ a \}$

The expressions are built up via

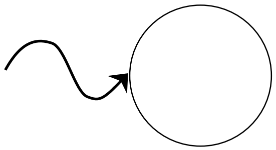
- *concatenation:* E_1E_2 where E_1 and E_2 are regular expressions
- *union:* $E_1|E_2$ where E_1 and E_2 are regular expressions
- *repetition:* E^* where E is a regular expression

Regular Expressions (RE) to ε-NFAs

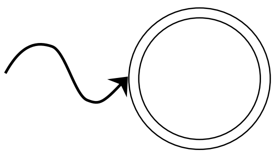
Convert an RE to an ε-NFA

Basic Idea: build up the ε-NFA recursively from the elements of a RE.

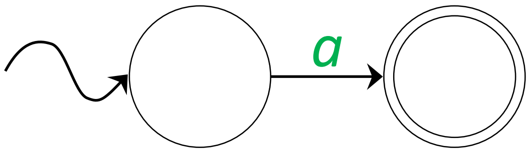
- If the RE is ∅ then the ε-NFA is:
  - no accepting state



- If the RE is ε then the ε-NFA is:
  - it accepts the empty string and nothing else



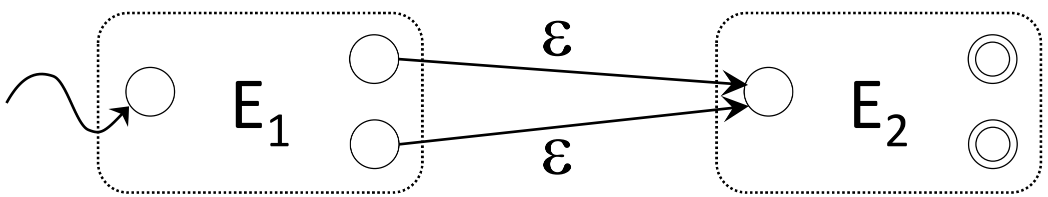
- If the RE is *a* then the ε-NFA is:



Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form E_1E_2 (i.e. *concatenation*) then convert the states of the ε-NFA that recognizes E_1 into non-accepting states and link them to the start state of the ε-NFS that recognizes E_2 via ε-transitions.

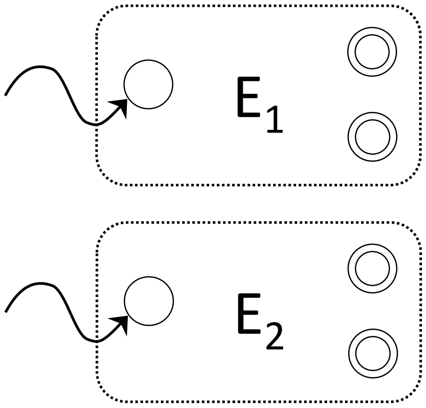


- Note: expressions and automata occur *in sequence*

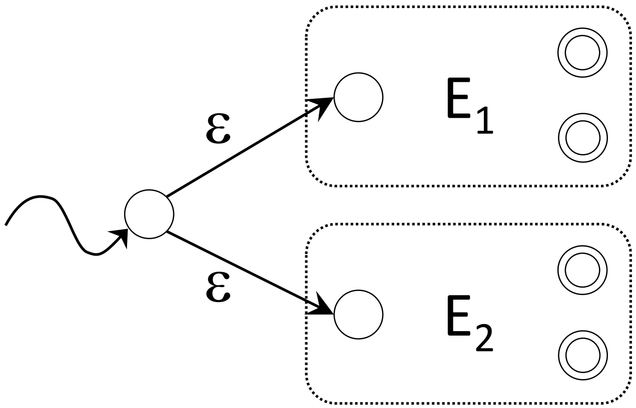
Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form $E_1|E_2$ (i.e. *union*): create a new start state and link it, via ε-transitions to the start states of the ε-NFAs that recognizes E_1 and E_2.



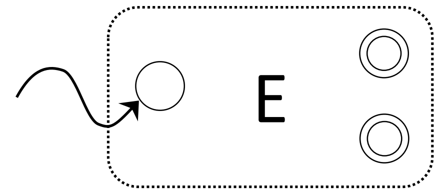
- Note: expressions and automata occur *in parallel*



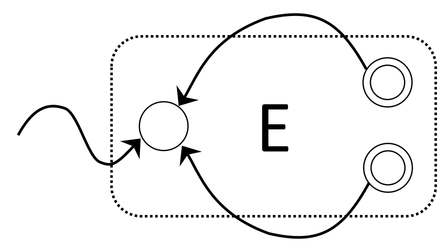
Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form E* (i.e. *repetition*): link all the accept states of the ε-NFA that recognizes E (via ε-transitions) to the start state.



- Note: expressions and automata occur *in a cycle.*

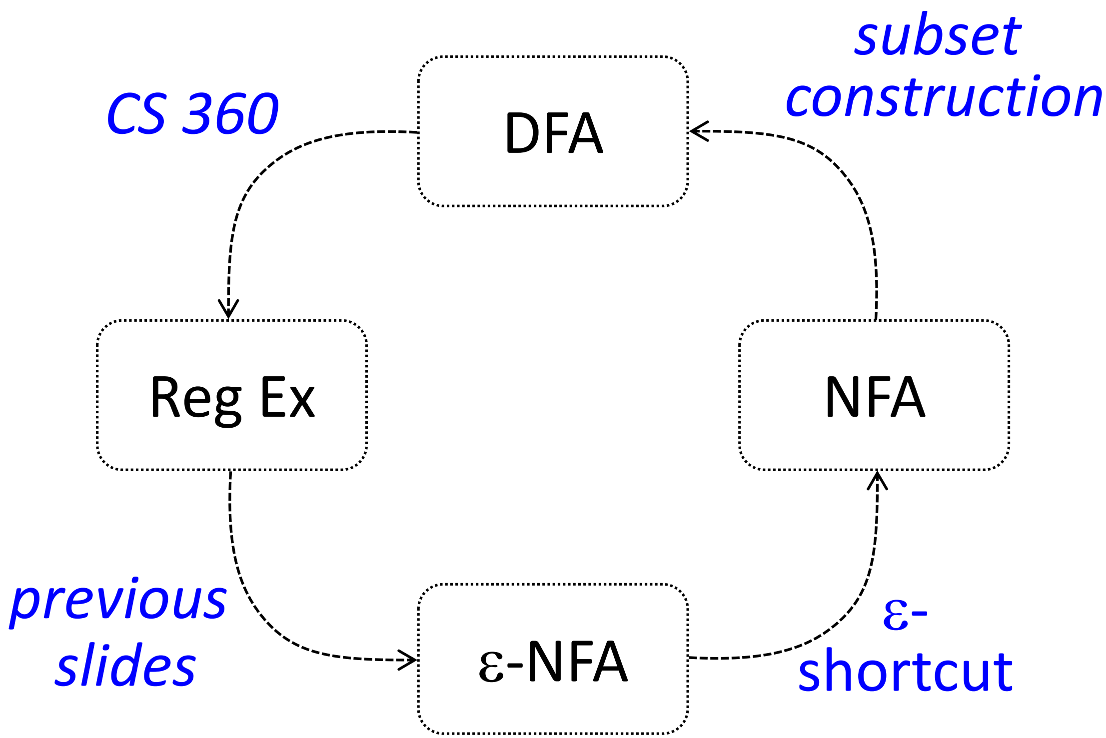


Regular Languages

Definition

A *regular language* is a language that can be

- specified by a regular expression
- recognized by an ε-NFA
- recognized by an NFA
- recognized by a DFA



Regular Expressions

Shorthands and Extensions

- will often see the use of the following to help simplify our expressions, especially in Linux
- *square brackets* (with ranges)
  - [a-z] means $a|b|c|\ldots|z$
  - match one of the given letters
  - [a-z] is all lowercase letters of the English alphabet
  - [A-Za-z] is all uppercase and lower case letters of the English alphabet

Regular Expressions

Shorthands and Extensions

- *plus sign*: like star but excluding ε
  - [0-9]+ means [0-9][0-9]*
  - matches non-negative integers (possibly with leading 0's).
- *dot* matches any single letter
  - .at matches hat, cat, fat, mat, bat, 7at, Aat, etc.
- *escape character* matches actual brackets, dots, etc.
  - [0-9]+\.[0-9]+ matches fractional numbers
  - e.g. 2.3425 (possibly with leading 0's).

Scanning

Quick Review

- Recall what we are trying to do: translate from a high level language to assembly language
- introduced regular expression and finite automata as a way to *specify* and *identify* words in the language
- Question: how does that work in practice?

Scanning

Scanner

- *Input:* some string *w* and a language *L*
  - in assembly language: “add $1, $2, $3”
  - in C++ “i += 1;”
- *Output:* a sequence of tokens
  - <Add> <Reg> <Comma> <Reg> <Comma> …
  - <Name> <Op> <Int> <Semicolon>
- *Challenge:* may be more than one possible answer:

  0x1234abcd vs 0 x 1234 abcd

  <HexInt> vs <Int> <Label> <Int> <Label>

  Answer: take the longest possible correct run of chars

Maximal Munch Scanning

Input

$c_0 c_1 c_2 \ldots c_{k-1}$ *// a series of characters*

Basic Idea: *keep going until you reach the error state*

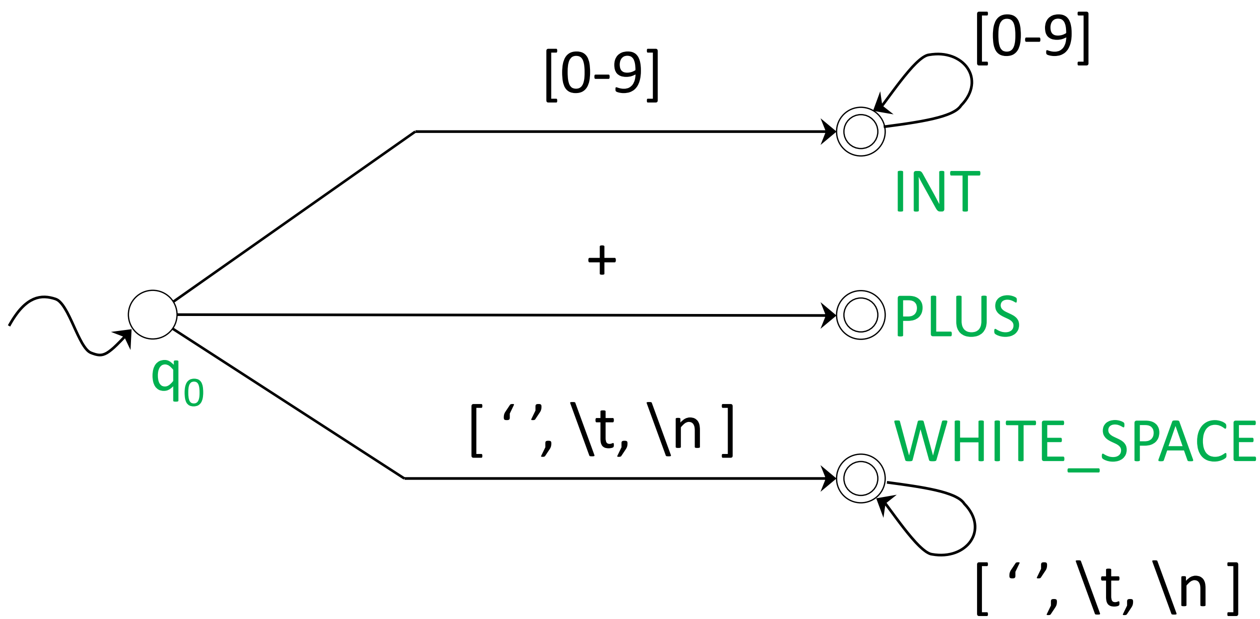
- *Step 1:* check next state based on character c_i (i.e. process one character at a time)
- *Step 2:* if you reach an error you've gone too far
  - look back at previous state
  - *Step 2a:* if it was not a final state, report error
  - *Step 2b:* if it was whitespace, ignore
  - *Step 2c:* if it was an accepting state, output the corresponding token
  - *Step 2d:* go back to start state (i.e. begin looking for the next token)

Maximal Munch Scanning

```
i = 0                                      // start at first char
state = q0                                 // start state of DFA
loop:
  next_state = ERROR                       // assume worst case
  if ( i < k ):                            // if not at end of input
    next_state = δ(state, ci)              //  1: go to next state
  if (next_state == ERROR):
    if (state is not an accepting_state):
      report error and exit                //  2a: report error
    if (state is not WHITE_SPACE):         //  2b: ignore white space
      output appropriate token             //  2c: output token
    state = q0                             //  2d: return to start state
    if (i == k):                           //  exit if no more input
      exit
  else:                                    //  process next char
    state = next_state
    i = i + 1
```

Scanners and NFAs

An DFA that Recognizes a Subset of WLP4 tokens



Maximal Munch Scanning

Maximal Munch Example 1

Input: $c_0c_1c_2c_3c_4c_5$ is 12+34;

- *Goal:* want to output a single token pair (INT, 12), not two token pairs, (INT, 1), (INT, 2).
- *Approach*: go until something other than *int* is seen
- when i = 2, c_2 = ‘+’, state will be <INT>
- next_state will be <ERROR>
- output token (INT, 12)
- go to q_0 (start state) again
- do not increment i, that is, skip over i = i+1
- now try processing c_i = ‘+’ in q_0 rather than in <INT>

Maximal Munch Scanning

Maximal Munch Example 2a

Input: $c_0c_1c_2c_3c_4c_5c_6c_7$ is 12 + 34;   // spaces have been added

- when i = 2, c_2 = ‘ ’, state will be <INT>
- next_state will be <ERROR>
- output token (INT, 12)
- go to q_0 (start state) again, process c_2 = ‘ ’ as white space

- when i = 3, c_3 = ‘+’, state will be <WHITE_SPACE>
- next_state will be <ERROR>
- do not output token
- go to q_0 (start state)
- do not increment i, that is, skip over i = i+1

Maximal Munch Scanning

Maximal Munch Example 2b

Input: $c_0c_1c_2c_3c_4c_5c_6c_7$ is 12 + 34;   // spaces have been added

- i = 3, c_3 = ‘+’, state will be q_0
- next_state will be <PLUS>
- increment i
- state becomes <PLUS>

- i = 4, c_4 = ‘ ’, state is <PLUS>
- next_state will be <ERROR>
- output token (PLUS, +)
- go to q_0 (start state) again, process c_4 = ‘ ’ as white space

Scanners and NFAs

Differences between a Scanner and a NFA

- A scanner *splits the input up into tokens*.
- An NFA *checks if the input is an element of a language* (set of strings).

Using an NFA to Implement a Scanner.

- describe each of the set of tokens by a regular expression (we'll do a small subset).
  - *keywords*: if int
  - *ID*: [a-zA-Z][a-zA-Z0-9]*
  - *operators*: [+ - * //]
  - *delimiters*: [( ) { } , :]

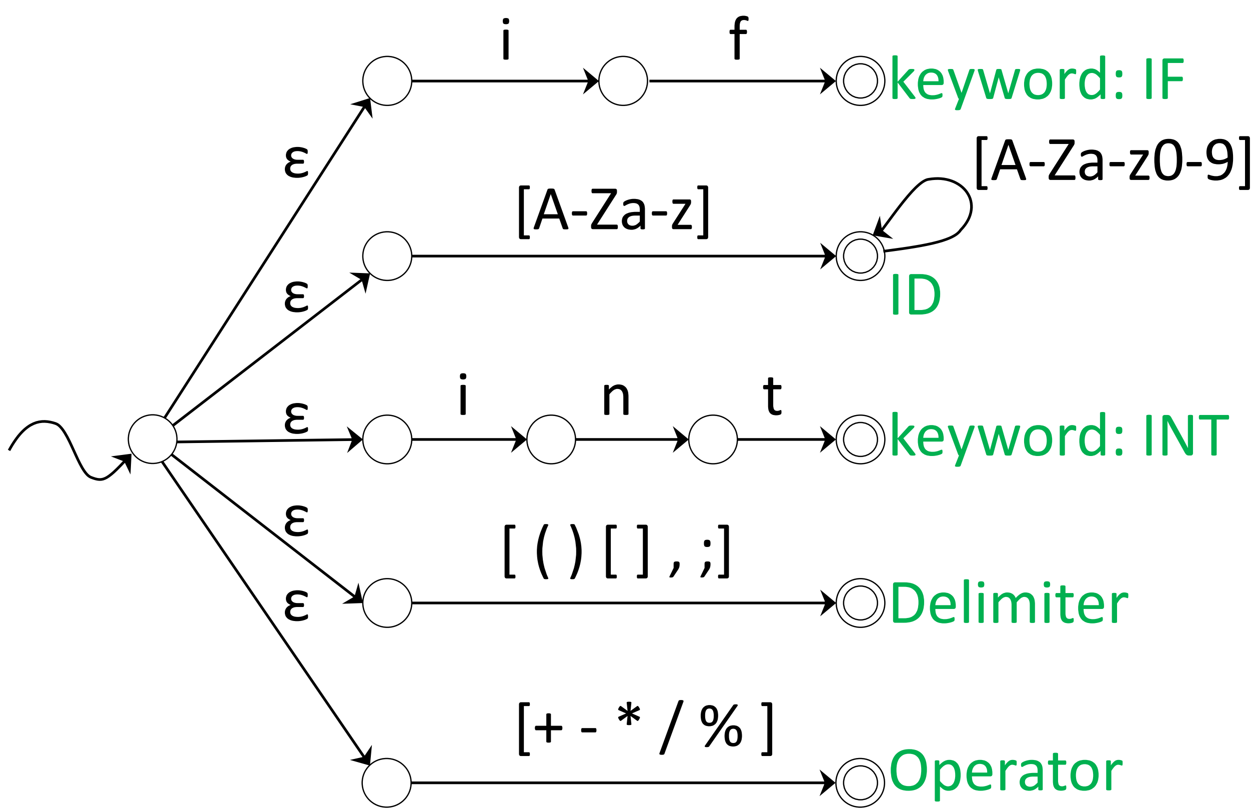
Scanners and NFAs

Using an NFA to make a Scanner

- create an NFA for each regular expression
- mark the final states by the type of token they accept
- combine all the individual NFAs into one large one (using ε transitions)
  - sometimes called λ (lambda) transitions
- convert NFA into DFA (we'll skip details here)
- *To keep the diagram simple*:
  - I'm using a subset of WLP4
  - I'm combining all the operators into Operator and all the delimiters into Delimiter while they should each get their own token.

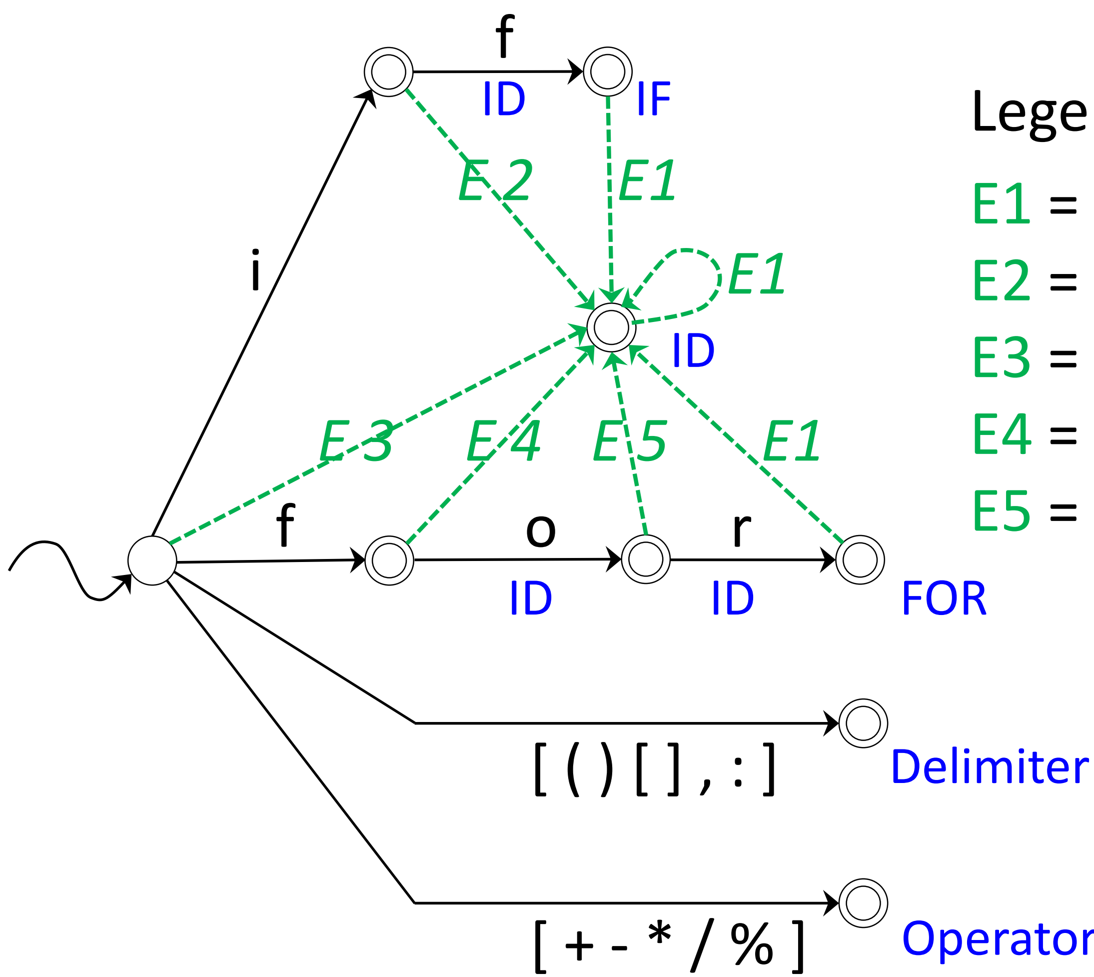
Scanners and NFAs

An NFA that Recognizes a Subset of WLP4 tokens



Scanners and DFAs

The Corresponding DFA that Recognizes our Tokens



Legend

E1 = [A-Za-z0-9]

E2 = [A-Za-eg-z0-9]

E3 = [A-Za-eghj-z]

E4 = [A-Za-np-z0-9]

E5 = [A-Za-qs-z0-9]