Patterns, Automata, and Regular Expressions

- Finite Automata – formal or abstract machine to recognize patterns
- Regular expressions – formal notation to describe/generate patterns

Finite Automata

- A finite collection of states
- An alphabet
- A set of transitions between those states labeled with symbols in the alphabet
- A start state, S0
- One or more final states

Deterministic – single-valued transition, no epsilon transitions

Non-deterministic – multi-valued transitions

Regular Expressions

- Defines a set of strings over the characters contained in some alphabet, defining a language
- Atomic operand can be
  - a character,
  - the symbol ε,
  - the symbol Φ, or
  - a variable whose value can be any pattern defined by a regular expression
- Three basic operations/operators
  - Union – e.g., a|b
  - Concatenation – e.g., ab
  - Closure – (Kleene closure) – e.g., a* - where a can be a set concatenated with itself zero or more times

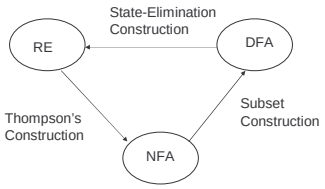
Precedence of Regular Expression Operators

- Closure (highest)
- Concatenation
- Union

Algebraic Laws for Regular Expressions

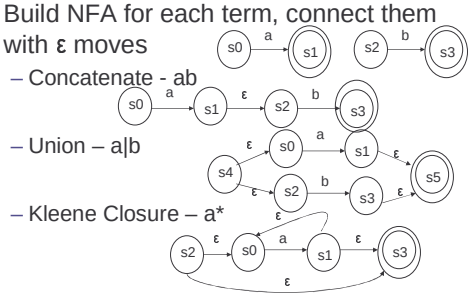
- Identity for Union Φ|R = R|Φ= R
- Identity for concatenation εR=Rε=R
- Associativity and commutativity of union R|S=S|R, ((R|S)|T) = (R|(S|T))
- Associativity of concatenation (RS)T = R(ST)
- Non-commutativity of concatenation
- Left distributivity of concatenation over union (R(S|T)) = (RS|RT)
- Right distributivity of concatenation over union ((S|T)R) = (SR|TR)
- Idempotence of union (R | R) = R

RE <-> DFA



Automated RE->NFA

- Build NFA for each term, connect them with ε moves
  - Concatenate - ab
  - Union – a|b
  - Kleene Closure – a*



Thompson's Construction

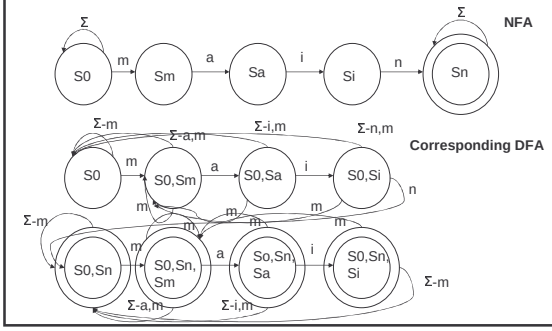
- Each NFA has a single start state and a single final state, with no transitions leaving the final state and only the initial transition entering the start state
- An ε-move always connects two states that were start or final states
- A state has at most 2 entering and 2 exiting ε-moves, and at most 1 entering and 1 exiting move on a symbol in the alphabet

NFA->DFA

- Subset construction algorithm
  - Each state in DFA corresponds to a set of states in NFA

```
q0 ← ε-closure(n0)
initialize Q with {q0}
while (Q is still changing)
   for each qi ε Q
      for each character α εΣ
            t← ε-closure(move(qi, α))
            T[qi, α] ← t
            if t ∉ Q then
                  add t to Q
```

Example



DFA Minimization

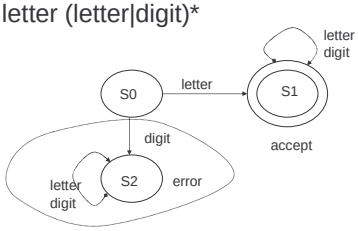
```
P ← P {S_F, {S - S_F}
while (P is still changing)
  T ← 0
  for each set p ε P
       for each  α ε Σ
            partition p by α
                 into p1, p2, p3, … pk
            T← T U p1, p2, p3, … pk
  if T ≠ P then
       P ← T
```

Automated DFA->RE

for i = 1 to N
  for j = 1 to N
    $R_{ij}^0 = \{a | \delta(s_i,a) = s_j\}$
    if (i == j)
      $R_{ij}^0 = R_{ij}^0 \cup \varepsilon$
for k = 1 to N
  for i = 1 to N
    for j = 1 to N
      $R_{ij}^k = R_{ik}^{k-1} R_{kk}^{k-1} R_{kj}^{k-1} \cup R_{ij}^{k-1}$
$L = \cup R_{1j}^N$
  ${}^s j \varepsilon S_F$

Regular Expression and DFA

Identifier
  letter → (a|b|c| … |z|A|B|C| … |Z)
  digit → (0|1|2|3|4|5|6|7|8|9)
  id → letter (letter|digit)*



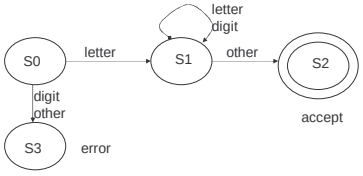
Implementing Scanners (Recognizer)

- Ad-hoc
- Semi-mechanical pure DFA
- Table-driven DFA

Code for Semi-Mechanical Pure DFA

```
state = S0; /* code for S0 */
done = false;
token_value = "" /* empty string */
token_type = error;
char = next_char();
while (not done) {
      class = char_class[char];
      switch(state) {
            case S0:
                  switch (class)
                        case 'letter': token_type = identifier; token_value = token_value+char;
                                       state = S1; char = next_char(); break;
                        case 'digit':  done = true; break;
                        case 'other':  done = true; break;
                  break;
            case S1:
                  switch(class)
                        case 'letter':
                        case 'digit': token_value = token_value+char;  char = next_char(); break;
                        case 'other': done = true; break;
                  break;
      }
}
return(token_type);
```

Table-Driven Recognizer



Tables Driving the Recognizer

char_class

	a-z	A-Z	0-9	other
value	letter	letter	digit	other

next_state

class	S0	S1	S2	S3
letter	S1	S1	--	--
digit	S3	S1	--	--
other	S3	S2	--	--

To change language, we can just change tables

Table-Driven Recognizer/Scanner

```
char = next_char();
state = S0; /* code for S0 */
token_value = "" /* empty string */
while (not done) {
      class = char_class[char];
      state = next_state[class,state];
      switch(state) {
            case S2: /* accept state */
                  token_type = identifier;
                  done = true; break;
            case S3: /* error */
                  token_type = error;
                  done = true; break;
            default: /* building an id */
                  token_value = token_value + char;
                  char = next_char; break;
      }
}
return(token_type);
```

Error Recovery

- E.g., illegal character
- What should the scanner do?
  - Report the error
  - Try to correct it?
- Error correction techniques
  - Minimum distance corrections
  - Hard token recovery
  - Skip until match

Scanner Summary

- Break up input into tokens
- Catch lexical errors
- Difficulty affected by language design
- Issues
  - Input buffering
  - Lookahead
  - Error recovery
- Scanner generators
  - Tokens specified by regular expressions
  - Regular expression -> DFA
  - Highly efficient in practice