Patterns, Automata, and Regular Expressions

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 ε R = R ε = R
- Associativity and commutativity of union R | S = S | R, (A | (B | C)) = ((A | B) | C)
- Associativity of concatenation (RS) T = R( ST )
- Non-commutativity of concatenation
- Left distributivity of concatenation over union (R | (S | T)) = (R | S) | (R | T)
- Right distributivity of concatenation over union ((S | T) R) = (S R) | (T R)
- Idempotence of union (S | S) = S
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

Automated RE->NFA

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

NFA->DFA

- Subset construction algorithm
  - Each state in DFA corresponds to a set of states in NFA
  - q0 ← ε-closure(∅)
  - initialize Q with {q0}
  - while (Q is still changing)
    - for each qi ∈ Q
      - for each character α ∈ Σ
        - t ← ε-closure(move(qi, α))
        - T[qi, α] ← t
        - if t ∉ Q then
          - add t to Q

Example

DFA Minimization

- P ← P (S_r, (S - S_r)) while (P is still changing)
  - T ← 0
  - for each set p ∈ P
    - for each α ∈ 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 \( N \)
\[
R_{ij}^0 = (a|δ(a, a) = s_i)
\]
if \( (i = j) \)
\[
R_{ij}^0 = R_{ij}^0 U ε
\]
for \( k = 1 \) to \( N \)
for \( i = 1 \) to \( N \)
for \( j = 1 \) to \( N \)
\[
R_{ij}^k = R_{ik}^{k-1} R_{kk}^{k-1} R_{kj}^{k-1} U R_{ij}^{k-1}
\]
\[
L = U R_{ij}^N
\]

Regular Expression and DFA

Identifier

\( \text{letter} \to (a|b|c| \ldots |z|A|B|C| \ldots |Z) \)
\( \text{digit} \to (0|1|2|3|4|5|6|7|8|9) \)
\( \text{id} \to \text{letter} (\text{letter}|\text{digit})^* \)

Implementing Scanners (Recognizer)

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

Table-Driven Recognizer

Code for Semi-Mechanical Pure DFA

```
class = "ID"  /* code for ID */
class = "LET"  /* code for LET */
class = "DIG"  /* code for DIG */
class = "OTH"  /* code for OTH */
while \(!class\) {
    code = next_char();
    switch (class) {
    case "ID":
        token_type = "IDENTIFIER";
        token_value = token_value + char_value;
        next_char();
        break;
    case "LET":
        token_type = "LETTER";
        token_value = token_value + char_value;
        next_char();
        break;
    case "DIG":
        token_type = "DIGIT";
        token_value = token_value + char_value;
        next_char();
        break;
    case "OTH":
        token_type = "OTHER";
        break;
    default:
        next_char();
        break;
    }
return (token_type);
```
Table-Driven Recognizer/Scanner

```c
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: /* accepts 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