Extended regular expression syntax

If $R_1$, $R_2$ are legal regular expressions, so are:

- $a$ for any ordinary symbol $a$
- $R_1R_2$ (concatenation)
- $R_1|R_2$ (or)
- $R_1^*$ (Kleene star: 0 or more concats)
- $R_1?$ (0 or 1)
- $R_1+$ (1 or more)
- $(R_1)$ (no effect: grouping)
- $[abc...]$ (any of the listed)

Lexer generator

- Reads in list of regular expressions $R_1,...,R_n$, one per token, with attached actions
  - ?[1-9][0-9]* { return new Token(Tokens.IntConst, Integer.parseInt(yytext()) }  
- Generates scanning code that decides:
  1. whether the input is lexically well-formed
  2. what is the corresponding token sequence
- Observation: Problem 1 is equivalent to deciding whether the input is in the language of the regular expression $(R_1|...|R_n)^*$
- Goal: how can we efficiently test membership in $L(R)$ for arbitrary $R$?
Regular expression matching

- Sketch of an efficient implementation:
  - start in some initial state
  - look at each input character in sequence, update scanner state accordingly
  - if state at end of input is an accepting state, the input string matches the RE
- For tokenizing, only need a finite amount of state: (deterministic) finite automaton (DFA) or finite state machine
- State of automaton = single integer

Finite Automata

- Automaton (DFA) can be represented as
  - A transition table
    |   | Non-
    | --- | --- |
    | 0  | 1   | Error |
    | 1  | 2   | 1     |
    | 2  | Error | Error |
  - A graph

A regexp matcher

```java
boolean accept_state[NSTATES] = { ... };
int trans_table[NSTATES][NCHARS] = { ... };
int state = 0;

while (state != ERROR_STATE) {
    c = input.read();
    if (c < 0) break;
    state = table[state][c];
}
return accept_state[state];
```

RE → Finite automaton?

- Can we build a finite automaton for every regular expression?
- Strategy: consider every possible kind of regular expression (define by induction on size of regular expression)
**Definition: NFA**

- Non-deterministic finite automaton has:
  - set of states; start state; accepting state(s)
  - arrows connecting states labeled by input symbols, or \( \epsilon \) (which does not consume input)
  - two arrows leaving a state may have same label

**Example:**

```
regexp?
```

---

**DFA vs NFA**

- DFA: action of automaton on each input symbol is fully determined
  - obvious table-driven implementation
- NFA:
  - automaton may have choice on each step
  - automaton accepts a string if there is *any way* to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
  - not obvious how to implement efficiently!

---

**RE \( \Rightarrow \) NFA intuition**

-\?[0-9]+ \( \rightarrow | \epsilon \) [0-9][0-9]*

**NFA construction**

- NFA only needs one stop state (why?)
- Canonical NFA:
**Inductive Construction**

- **$R_1 R_2$**
- **$R_1 \mid R_2$**
- **$R^*$**

**Executing NFA**

- Problem: how to execute NFA efficiently? “strings accepted are those for which there is some corresponding path from start state to an accept state”
- Conclusion: search all paths in graph consistent with the string
- Idea: search paths in parallel
  - Keep track of subset of NFA states that search could be in after seeing string prefix
  - “Multiple fingers” pointing to graph

**Example**

- Input string: -23
- NFA states:
  - $\{0,1\}$
  - $\{1\}$
  - $\{2, 3\}$
  - $\{2, 3\}$

**NFA-DFA conversion**

- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: $\{0,1\}$, $\{1\}$, $\{2, 3\}$
**DFA minimization**

- DFA construction can produce large DFA with many states
- Lexer generators perform additional phase of *DFA minimization* to reduce to minimum possible size (see Dragon Book for details)

**Handling multiple token REs**

Longest-match rule: on “error” in DFA, output token (invoke action) from last reached accept state.

**Summary**

- Lexical analyzer converts text stream to tokens
- Regular expressions define tokens precisely
- Regular expressions (+priority order) converted to a fast table-driven scanner by converting them to NFAs, then to DFAs
- Result: shorter, easily maintained code
  - NFA-DFA conversion handles “overlapping” tokens that can be hard to code, maintain
  - usually as or more efficient than hand-written code
- Lexer generators available off-the-shelf
- Usable for all kinds of input parsing tasks
- Read chapters 1-2 from Appel