Regular expressions \[\rightarrow\] NFA

Lexical Specification

DFA

Table-driven Implementation of DFA
In mathematics, computer science, and linguistics, a formal language consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules.

A formal language $L$ over an alphabet $\Sigma$ is a subset of $\Sigma^*$, that is, a set of words over that alphabet.
Regular expressions are used to define patterns of characters; they are used in UNIX tools such as awk, grep, vi and, of course, lex.

A regular expression is just a form of notation, used for describing sets of words. For any given set of characters in a set $\Sigma$, a regular expression over $\Sigma$ is defined by:

- The empty string, $\varepsilon$, which denotes a string of length zero, {""} and means take nothing from the input. It is most commonly used in conjunction with other regular expressions to denote optionality.

- Any character in $\Sigma$ may be used in a regular expression. For instance, if we write 'a' as a regular expression, this means take the letter a from the input; ie. it denotes the (singleton) set of words {“a”}
1/ The union operator, |, which denotes the union of two sets of words. Thus the regular expression a|b denotes the set {“a”, “b”}, and means take either the letter a or the letter b from the input.

2/ Writing two regular expressions side-by-side is known as concatenation; thus the regular expression ab denotes the set {“ab”} and means take the character a followed by the character b from the input.
3/ The Kleene closure of a regular expression, denoted by *, indicates zero or more occurrences of that expression. Thus a* is the (infinite) set \{\epsilon, “a”, “aa”, “aaa”, ...\} and means take zero or more ‘a’ from the input.

Brackets may be used in a regular expression to enforce precedence or increase clarity.

The above are the three compound expressions
Each *regular expression* represents a set of strings.

- **Symbol**: For each symbol $a$ in the language, the regular expression $a$ denotes the string $a$.
- **Alternation**: If $M$ and $N$ are 2 regular expressions, then $M|N$ denotes a string in $M$ or a string in $N$.
- **Concatenation**: If $M$ and $N$ are 2 regular expressions, then $MN$ denotes a string $\alpha\beta$ where $\alpha$ is in $M$ and $\beta$ is in $N$.
- **Epsilon**: The regular expression $\epsilon$ denotes the empty string.
- **Repetition**: The *Kleene closure* of $M$, denoted $M^*$ is the set of zero or more concatenations of $M$.

Kleene closure binds tighter than concatenation, and concatenation binds tighter than alternation.
### Regular Expression Examples

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`(0</td>
<td>1)`*</td>
</tr>
<tr>
<td>`(0</td>
<td>1)`*0</td>
</tr>
<tr>
<td>`(a</td>
<td>b)`*</td>
</tr>
<tr>
<td>`(a</td>
<td>b)<code>*</code>aa(a</td>
</tr>
<tr>
<td>$b^<em>(abb^</em>)^*(a</td>
<td>\epsilon)$</td>
</tr>
<tr>
<td>$ab^*(c</td>
<td>\epsilon)$</td>
</tr>
</tbody>
</table>
Regular Expression Shorthands

The following abbreviations are generally used:

- \([axby]\) means \((a|x|b|y)\)
- \([a–e]\) means \([abcde]\)
- \(M?\) means \((M|\epsilon)\)
- \(M^+\) means \((MM^*)\)
- \(\cdot\) means any single character except a newline character
- "a*" is a quotation and the string in quotes literally stands for itself.
Is do99 and identifier or a keyword (do) followed by a number (99)?

Most modern lexical-analyser generators follow 2 rules to disambiguate situations like above.

- Longest match: The longest initial substring that can match any regular expression is taken as the next token.

- Rule priority: In the case where the longest initial substring is matched by multiple regular expressions, the first regular expression that matches determines the token type.

So do99 is an identifier.
Finite Automata

Regular expressions are good for specifying lexical tokens. Finite automata are good for recognising regular expressions.

A finite automata consists of a set of nodes and edges. Edges go from one node to another node and are labelled with a symbol. Nodes represent states. One of the nodes represents the start node and some of the node are final states.

A deterministic finite automaton (DFA) is a finite automaton in which no pairs of edges leading away from a node are labelled with the same symbol.

A nondeterministic finite automaton (NFA) is a finite automaton in which two or more edges leading away from a node are labelled with the same symbol.
A finite automaton consists of
  – An input alphabet \( \Sigma \)
  – A finite set of states \( S \)
  – A start state \( n \)
  – A set of accepting states \( F \subseteq S \)
  – A set of transitions \( \text{state} \rightarrow^{\text{input state}} \text{state} \)

- Transition
  \[ s_1 \rightarrow^a s_2 \]

- Is read
  In state \( s_1 \) on input \( a \) go to state \( s_2 \)

- If end of input and in accepting state => accept

- Otherwise => reject
- A state
- The start state
- An accepting state
- A transition
Regular Expressions for Tokens

do
\[a-zA-Z][a-zA-Z0-9]^*\]
\[0-9]^+
\[(0-9)^+\.*[0-9]*\.(0-9)^+\]
\[//\*\[a-zA-Z0-9\]*\*\*/\n\["\"\"/\"\"\n\["\"\"/\n\["\"\"/\n\["\"\"/\]t\]^+\]

DO
ID
NUM
REAL
comment or white space

1 \rightarrow 2 \rightarrow 3
DO

1 \rightarrow 2
ID

1 \rightarrow 2
NUM

1 \rightarrow 2 \rightarrow 3
REAL

1 \rightarrow 2 \rightarrow 3 \rightarrow 4
blank

1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5
blank

1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5
white space

1 \rightarrow 2
ERROR

3071
Each final state must be labelled with the token-type that it accepts.
A finite automaton can be encoded by:

- **Transition matrix:** a 2-dimensional array, indexed by input character and state number, that contains the next state.
  
  ```c
  int edges[][] = {{/* ws,..., 0, 1, 2,... d, e, f,... o,... */
    /* state 0 */ { 0,..., 0, 0, 0,..., 0, 0, 0,..., 0, 0,... },
    /* state 1 */ { 0,..., 7, 7, 7,..., 2, 4, 4,..., 4, 4,... },
    /* state 2 */ { 0,..., 4, 4, 4,..., 4, 4, 4,..., 3, 4,... },
    /* state 3 */ { 0,..., 4, 4, 4,..., 4, 4, 4,..., 4, 4,... },
    /* state 4 */ { 0,..., 4, 4, 4,..., 4, 4, 4,..., 4, 4,... },
    /* state 5 */ { 0,..., 6, 6, 6,..., 0, 0, 0,..., 0, 0,... },
    ... }}
  ```

- **Action array:** an array, indexed by final state number, that contains the resulting action, e.g. if the final state is 2 then return ID, if the final state is 3 then return DO, etc.
• Another kind of transition: $\varepsilon$-moves move to another state without consuming input

• Deterministic Finite Automata (DFA)
  – One transition per input per state
  – No $\varepsilon$-moves

• Nondeterministic Finite Automata (NFA)
  – Can have multiple transitions for one input in a given state
  – Can have $\varepsilon$-moves
• NFAs and DFAs recognize the same set of languages
  — regular languages

• DFAs are faster to execute
  — There are no choices to consider

• NFAs are, in general, smaller
How an NFA operates

We begin in the start state (usually labelled 0) and read the first character on the input.

Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state. There may be more than one possible transition, in which case we choose one at random.

If at any stage there is an output transition labelled with the empty string, $\varepsilon$, we may take it without consuming any input. We keep going like this until we have no more input, or until we have reached one of the final states.
If we are in a final state, with no input left, then we have succeeded in recognising a pattern.

Otherwise we must backtrack to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

The key concept here is that: every NFA corresponds to a regular expression.

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm.
Regular Expressions to NFA

- Notation: NFA for rexp $M$

- For $\varepsilon$

- For input $a$
• For $AB$

• For $A + B$
• For $A^*$
• Consider the regular expression

$$(1+0)^*1$$
(1+0)
(1+0)∗
(1+0)*1
• Consider the regular expression

$$(1+0)^*1$$