CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop
Lexical Analysis

• Also called *scanning*, take input program *string* and convert into tokens

• Example:

\[
\text{double } f = \text{sqrt}(-1);
\]

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_DOUBLE</td>
<td>&quot;double&quot;</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>T_OP</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>&quot;sqrt&quot;</td>
</tr>
<tr>
<td>T_LPAREN</td>
<td>&quot;(&quot;</td>
</tr>
<tr>
<td>T_OP</td>
<td>&quot;-&quot;</td>
</tr>
<tr>
<td>T_INTCONSTANT</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>T_RPAREN</td>
<td>&quot;)&quot;</td>
</tr>
<tr>
<td>T_SEP</td>
<td>&quot;;&quot;</td>
</tr>
</tbody>
</table>
Token Attributes

• Some tokens have attributes
  – T_IDENT
  – T_INTCONSTANT

• Other tokens do not
  – T_WHILE

• Token=T_IDENT, Lexeme=“sqrt”, Pattern

• Source code location for error reports
Lexical errors

• What if user omits the space in “doublef”?  
  – No lexical error, single token  
    T_IDENT(“doublef”) is produced instead of sequence T_DOUBLE, T_IDENT(“f”)!

• Typically few lexical error types  
  – E.g., illegal chars, opened string constants or comments that are not closed
Lexical errors

• Lexical analysis should not disambiguate tokens,
  – e.g. unary op + versus binary op +
  – Use the same token T_PLUS for both
  – It’s the job of the parser to disambiguate based on the context

• Language definition should not permit crazy long distance effects (e.g. Fortran)
  DO 5 I = 1,5  T_DO T_INT(5) T_ID(I)
  DO 5 I = 1.5  T_ID(DO5I) T_EQ
Ad-hoc Scanners
Implementing Lexers: Loop and switch scanners

- Ad hoc scanners
- Big nested switch/case statements
- Lots of getc()/ungetc() calls
  - Buffering; Sentinels for push-backs; streams
- Can be error-prone, use only if
  - Your language’s lexical structure is very simple
  - The tools do not provide what you need for your token definitions
- Changing or adding a keyword is problematic
- Have a look at an actual implementation of an ad-hoc scanner
Implementing Lexers: Loop and switch scanners

• Another problem: how to show that the implementation actually captures all tokens specified by the language definition?
• How can we show correctness
• Key idea: separate the definition of tokens from the implementation
• Problem: we need to reason about patterns and how they can be used to define tokens (recognize strings).
Specification of Patterns using Regular Expressions
Formal Languages: Recap

• Symbols: \(a, b, c\)
• Alphabet: finite set of symbols \(\Sigma = \{a, b\}\)
• String: sequence of symbols \(\text{bab}\)
• Empty string: \(\varepsilon\) Define: \(\Sigma^\varepsilon = \Sigma \cup \{\varepsilon\}\)
• Set of all strings: \(\Sigma^*\) cf. The Library of Babel, Jorge Luis Borges
• (Formal) Language: a set of strings \(\{a^n b^n : n > 0\}\)
Regular Languages

• The set of regular languages: each element is a regular language

• Each regular language is an example of a (formal) language, i.e. a set of strings
  e.g. \{ a^m b^n : m, n \text{ are } +ve \text{ integers} \}
Regular Languages

• Defining the set of all regular languages:
  – The empty set and \( \{a\} \) for all \( a \) in \( \Sigma^e \) are regular languages
  – If \( L_1 \) and \( L_2 \) and \( L \) are regular languages, then:
    
    \[ L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\} \] (concatenation)
    
    \[ L_1 \cup L_2 \] (union)
    
    \[ L^* = \bigcup_{i=0}^{\infty} L^i \] (Kleene closure)
    
    are also regular languages
  – There are no other regular languages
Formal Grammars

• A formal grammar is a concise description of a formal language
• A formal grammar uses a specialized syntax
• For example, a regular expression is a concise description of a regular language
  
  \((a|b)^*abb\) : is the set of all strings over the alphabet \{a, 
  
  b\} which end in \(abb\)

• We will use regular expressions (regexps) in order to define tokens in our compiler,
  
  – e.g. lexemes for string tokens are "(Σ")* \\"
Regular Expressions: Definition

- Every symbol of $\Sigma \cup \{ \varepsilon \}$ is a regular expression
  - E.g. if $\Sigma = \{a,b\}$ then ‘a’, ‘b’ are regexps
- If $r_1$ and $r_2$ are regular expressions, then the core operators to combine two regexps are
  - Concatenation: $r_1r_2$, e.g. ‘ab’ or ‘aba’
  - Alternation: $r_1|r_2$, e.g. ‘a|b’
  - Repetition: $r_1^*$, e.g. ‘a*’ or ‘b*’
- No other core operators are defined
  - But other operators can be defined using the basic operators (as in lex regular expressions) e.g. $a^+ = aa^*$
<table>
<thead>
<tr>
<th>Expression</th>
<th>Matches</th>
<th>Example</th>
<th>Using core operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>\c</td>
<td>character c literally</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>string s literally</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>any character but newline</td>
<td>a.*b</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>beginning of line</td>
<td>^abc</td>
<td>used for matching</td>
</tr>
<tr>
<td>$</td>
<td>end of line</td>
<td>abc$</td>
<td>used for matching</td>
</tr>
<tr>
<td>[s]</td>
<td>any one of characters in string s</td>
<td>[ abc ] (alb</td>
<td>lc)</td>
</tr>
<tr>
<td>[^s]</td>
<td>any one character not in string s</td>
<td>[ ^a ] (blc) where $\Sigma = {a,b,c}$</td>
<td></td>
</tr>
<tr>
<td>r*</td>
<td>zero or more strings matching r</td>
<td>a*</td>
<td></td>
</tr>
<tr>
<td>r+</td>
<td>one or more strings matching r</td>
<td>a+</td>
<td>aa*</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one r</td>
<td>a?</td>
<td>(alɛ)</td>
</tr>
<tr>
<td>r{m,n}</td>
<td>between m and n occurrences of r</td>
<td>a{2,3}</td>
<td>(aalaaa)</td>
</tr>
<tr>
<td>r₁r₂</td>
<td>an r₁ followed by an r₂</td>
<td>ab</td>
<td></td>
</tr>
<tr>
<td>r₁</td>
<td>r₂</td>
<td>an r₁ or an r₂</td>
<td>a</td>
</tr>
<tr>
<td>(r)</td>
<td>same as r</td>
<td>(a</td>
<td>b)</td>
</tr>
<tr>
<td>r₁</td>
<td>r₂</td>
<td>r₁ when followed by an r₂</td>
<td>abc/123</td>
</tr>
</tbody>
</table>
Regular Expressions are Trees
Regular Expressions: Definition

• Note that operators apply recursively and these applications can be ambiguous
  – E.g. is $aa|bc$ equal to $a(a|b)c$ or $((aa)|b)c$?

• Avoid such cases of ambiguity - provide explicit arguments for each regexp operator
  – For convenience, for examples on this page, let us use the symbol ‘⋅’ to denote the operator for concatenation

• Remove ambiguity with an explicit regexp tree
Regular Expressions: Definition

- Remove ambiguity with an explicit regexp tree
  - a(a|b)c is written as 
    \((\cdot (\cdot a (\mid ab))c)\)
  - or in postfix: aab\cdot c\cdot

- ((aa)|b)c is written as 
  \((\cdot (\mid (\cdot aa)b)c)\)
  - or in postfix: aa\cdot b\mid c\cdot

- Does the order of concatenation matter?
Equivalence of Regexps

- $(R|S)|T = R|(S|T) = R|S|T$
- $(RS)T = R(ST)$
- $(R|S) = (S|R)$
- $R^*R^* = (R^*)^* = R^*$
  $= RR^*|\varepsilon$
- $R^{**} = R^*$
- $(R|S)T = RT|ST$

- $R(S|T) = RS | RT$
- $(R|S)^* = (R^*S^*)^* = (R^*S)^*R^* = (R^*|S^*)^*$
- $RR^* = R^*R$
- $(RS)^*R = R(SR)^*$
- $R = R|R = R\varepsilon$
Equivalence of Regexps

- $0(10)^*1|(01)^*$
- $(01)(01)^*|(01)^*$
- $(01)(01)^*|(01)(01)^*|\varepsilon$
- $(01)(01)^*|\varepsilon$
- $(01)^*$

- $(RS)^*R == R(SR)^*$
- $RS == (RS)$
- $R^* == RR^*|\varepsilon$
- $R == R|R$
- $R^* == RR^*| \varepsilon$
Implementing Regular Expressions with Finite-state Automata
Regular Expressions

• To describe all lexemes that form a token as a pattern
  – \((0|1|2|3|4|5|6|7|8|9)^+\)

• Need decision procedure: to which token does a given sequence of characters belong (if any)?
  – Finite State Automata
  – Can be deterministic (DFA) or non-deterministic (NFA)
Deterministic Finite State Automata: DFA

- A set of states $S$
  - One start state $q_0$, zero or more final states $F$
- An alphabet $\sum$ of input symbols
- A transition function:
  - $\delta: S \times \sum \rightarrow S$
- Example: $\delta(1, a) = 2$
DFA: Example

- What regular expression does this automaton accept?

Answer: \((0|1)^*00\)

A: start state
C: final state
DFA simulation

- Start state: A
  1. \( \delta(A,0) = B \)
  2. \( \delta(B,0) = C \)
  3. \( \delta(C,1) = A \)
  4. \( \delta(A,0) = B \)
  5. \( \delta(B,0) = C \)
- no more input and C is final state: accept

Input string: 00100

DFA simulation takes at most \( n \) steps for input of length \( n \) to return accept or reject
Building a Lexical Analyzer

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
- NFA $\Rightarrow$ DFA
- DFAs or NFAs for all the tokens $\Rightarrow$ Lexical Analyzer
- Two basic rules to deal with multiple matching: greedy match + regexp ordering

Note that greedy means longest leftmost match
Lexical Analysis using Lex

{%
#include <stdio.h>
#define NUMBER  256
#define IDENTIFIER 257
%

/* regexp definitions */
num [0-9]+ %

{num} { return NUMBER; }
[a-zA-Z0-9]+ { return IDENTIFIER; }
%

int main () {
    int token;
    while ((token = yylex())) {
        switch (token) {
            case NUMBER: printf("NUMBER: %s, LENGTH:%d\n", yytext, yyleng); break;
            case IDENTIFIER: printf("IDENTIFIER: %s, LENGTH:%d\n", yytext, yyleng); break;
            default: printf("Error: %s not recognized\n", yytext);
        }
    }
}
%

simpletok.lex
Converting Regular Expressions into Non-deterministic Automata
NFAs

• NFA: like a DFA, except
  – A transition can lead to more than one state, that is, \( \delta: S \times \Sigma \Rightarrow 2^S \)
  – One state is chosen non-deterministically
  – Transitions can be labeled with \( \epsilon \), meaning states can be reached without reading any input, that is,

\[
\delta: S \times \Sigma \cup \{ \epsilon \} \Rightarrow 2^S
\]
Thompson’s construction
Converts regexps to NFA

Build NFA recursively from regexp tree

Build NFA with left-to-right parse of postfix string using a stack

Input = aab l c •
• read a, push n1 = nfa(a)
• read a, push n2 = nfa(a)
• read b, push n3 = nfa(b)
• read |, n3 = pop(); n2 = pop(); push
  n4 = nfa(or, n2, n3)
• read •, n4 = pop(); n1 = pop(); push
  n5 = nfa(cat, n1, n4)
• read c, push n6 = nfa(c)
• read •, n6 = pop(); n5 = pop(); push
  n7 = nfa(cat, n5, n6)
Thompson’s construction

• Converts regexps to NFA
• Six simple rules
  – Empty language
  – Symbols
  – Empty String
  – Alternation ($r_1$ or $r_2$)
  – Concatenation ($r_1$ followed by $r_2$)
  – Repetition ($r_1^*$)

Used by Ken Thompson for pattern-based search in text editor QED (1968)
To keep things simple our version is more verbose
Thompson Rule 0

- For the empty language $\phi$ (optionally include a sinkhole state)
Thompson Rule 1

- For each symbol $x$ of the alphabet, there is a NFA that accepts it (include a sinkhole state)
Thompson Rule 2

• There is an NFA that accepts only $\varepsilon$
Thompson Rule 3

- Given two NFAs for $r_1$, $r_2$, there is a NFA that accepts $r_1 | r_2$
Thompson Rule 3

• Given two NFAs for $r_1$, $r_2$, there is a NFA that accepts $r_1| r_2$
Thompson Rule 4

• Given two NFAs for $r_1$, $r_2$, there is a NFA that accepts $r_1r_2$
Thompson Rule 4

- Given two NFAs for $r_1$, $r_2$, there is a NFA that accepts $r_1r_2$
Thompson Rule 5

• Given a NFA for $r_1$, there is an NFA that accepts $r_1^*$
Thompson Rule 5

• Given a NFA for $r_1$, there is an NFA that accepts $r_1^*$
Example

• Set of all binary strings that are divisible by four (include 0 in this set)
• Defined by the regexp: \(((0|1)^*00)\ |\ 0\n• Apply Thompson’s Rules to create an NFA
Basic Blocks 0 and 1

- 0
- 1

(this version does not report errors: no sinkholes)
01
\((0|1)^*\)
\(((0|1)^*00)\vdash 0\)
Matching Patterns using Non-deterministic Automata
(conversion from NFA to DFA)
Simulating NFAs

• Simulation == Given a NFA and input string, does the string match the pattern?
• Similar to DFA simulation
• But have to deal with $\varepsilon$ transitions and multiple transitions on the same input
• Instead of one state, we have to consider sets of states
NFA to DFA Conversion

• Simulation implicitly converts NFA -> DFA
• Subset construction
• Idea: subsets of set of all NFA states are equivalent and become one DFA state
• Algorithm simulates movement through NFA
• Key problem: how to treat $\varepsilon$-transitions?
ε-Closure

- Start state: $q_0$
- $ε$-closure(S): S is a set of states

initialize: $S \leftarrow \{q_0\}$

$T \leftarrow S$

repeat $T' \leftarrow T$

$T \leftarrow T' \cup \bigcup_{s \in T'} \text{move}(s, ε)$

until $T = T'$
ε-Closure (T: set of states)

push all states in T onto stack
initialize ε-closure(T) to T
while stack is not empty do begin
  pop t off stack
  for each state u with u ∈ move(t, ε) do
    if u ∉ ε-closure(T) do begin
      add u to ε-closure(T)
      push u onto stack
    end
  end
end
NFA Simulation

• After computing the \( \varepsilon \)-closure move, we get a set of states
• On some input extend all these states to get a new set of states

\[
\text{DFAedge}(T, c) = \varepsilon\text{-closure} \left( \bigcup_{q \in T} \text{move}(q, c) \right)
\]
NFA Simulation

- $\text{DFAedge}(T, c) = \varepsilon\text{-closure} \left( \bigcup_{q \in T} \text{move}(q, c) \right)$
- Start state: $q_0$
- Input: $c_1, \ldots, c_k$

$$T \leftarrow \varepsilon\text{-closure}(\{q_0\})$$

for $i \leftarrow 1$ to $k$

$$T \leftarrow \text{DFAedge}(T, c_i)$$
Conversion from NFA to DFA

• Conversion method closely follows the NFA simulation algorithm
• Instead of simulating, we can collect those NFA states that behave identically on the same input
• Group this set of states to form one state in the DFA
Example: subset construction
$\varepsilon$-closure($q_0$)
move($\varepsilon$-closure($q_0$), 0)
\(\epsilon\)-closure(move(\(\epsilon\)-closure(q_0), 0))
move(ε-closure(q₀), 1)
\(\varepsilon\text{-closure}(\text{move}(\varepsilon\text{-closure}(q_0), 1))\)
Subset Construction

add \(\varepsilon\)-closure\((q_0)\) to \(Dstates\) unmarked

while \(\exists\) unmarked \(T \in Dstates\) do begin
  mark \(T\);
  for each symbol \(c\) do begin
    \(U := \varepsilon\)-closure\((\text{move}(T, c))\);
    if \(U \not\in Dstates\) then
      add \(U\) to \(Dstates\) unmarked
      \(Dtrans[d, c] := U;\)
  end
end
Subset Construction

states[0] = \text{\texttt{\textbf{\varepsilon-closure}}}({q_0})

p = j = 0

\textbf{while} j \leq p \textbf{ do begin}

\hspace{1em} \textbf{for each symbol } c \textbf{ do begin}

\hspace{2em} e = \texttt{DFAedge}(states[j], c)

\hspace{2em} \textbf{if } e = \text{states}[i] \text{ for some } i \leq p

\hspace{2em} \textbf{then} \quad \text{Dtrans}[j, c] = i

\hspace{2em} \textbf{else} \quad p = p+1

\hspace{3em} \text{states}[p] = e

\hspace{3em} \text{Dtrans}[j, c] = p

\hspace{2em} j = j + 1

\hspace{1em} \textbf{end}

\textbf{end}
DFA (partial)

[1, 2, 3, 4, 6, 9, 12] 0 [3, 4, 5, 6, 8, 9, 10, 13, 14]

1 [3, 4, 6, 7, 8, 9]
DFA for \(((0|1)^*00)\)|0
Minimization of DFAs
Minimization of DFAs

[3, 4, 6, 7, 8, 9]

[3, 4, 5, 6, 8, 9, 10, 11, 14]

[3, 4, 5, 6, 8, 9, 10]
NFA to DFA Complexity Analysis
NFA to DFA

- Subset construction converts NFA to DFA
- Complexity:
  - For FSAs, we measure complexity in terms of initial cost (creating the automaton) and per string cost
  - Let $r$ be the length of the regexp and $n$ be the length of the input string
  - NFA, Initial cost: $O(r)$; Per string: $O(rn)$
  - DFA, Initial cost: $O(r^2s)$; Per string: $O(n)$
  - DFA, common case, $s = r$, but worst case $s = 2^r$
NFA to DFA

• A regexp of size \( r \) can become a \( 2^r \) state DFA, an exponential increase in complexity
  – Try the subset construction on NFA built for the regexp \( A^*aA^nA \) where \( A \) is the regexp \( (a|b) \)

• Note that the NFA for regexp of size \( r \) will have \( r \) states

• Minimization can reduce the number of states

• But minimization requires determinization
NFA to DFA
NFA to DFA
NFA to DFA

\[2^5 = 32\] states
# NFA vs. DFA in the wild

<table>
<thead>
<tr>
<th>Engine Type</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA</td>
<td><code>awk</code> (most versions), <code>egrep</code> (most versions), <code>flex</code>, <code>lex</code>, MySQL, Procmail</td>
</tr>
<tr>
<td>Traditional NFA</td>
<td>GNU Emacs, Java, <code>grep</code> (most versions), <code>less</code>, <code>more</code>, .NET languages, PCRE library, Perl, PHP (pcre routines), Python, Ruby, <code>sed</code> (most versions), vi</td>
</tr>
<tr>
<td>POSIX NFA</td>
<td><code>mawk</code>, MKS utilities, GNU Emacs (when requested)</td>
</tr>
<tr>
<td>Hybrid NFA/DFA</td>
<td>GNU <code>awk</code>, GNU <code>grep/egrep</code>, Tcl</td>
</tr>
</tbody>
</table>
Extensions to Regular Expressions

• Most modern regexp implementations provide extensions:
  – matching groups; \1 refers to the string matched by the first grouping (), \2 to the second match, etc.,
    • e.g. ([a-z]+)\1 which matches abab where \1=ab
  – match and replace operations,
    • e.g. s/([a-z]+)/\1\1/g which changes ab into abab where \1=ab

• These extensions are no longer “regular”. In fact, extended regexp matching is NP-hard
  – Extended regular expressions (including POSIX and Perl) are called REGEX to distinguish from regexp (which are regular)

• In order to capture these difficult cases, the algorithms used even for simple regexp matching run in time exponential in the length of the input
Implementing a Lexical Analyzer
Lexical Analyzer using NFAs

- For each token convert its regexp into a DFA or NFA
- Create a new start state and create a transition on $\varepsilon$ to the start state of the automaton for each token
- For input $i_1, i_2, \ldots, i_n$ run NFA simulation which returns some final states (each final state indicates a token)
- If no final state is reached then raise an error
- Pick the final state (token) that has the longest match in the input,
  - e.g. prefer DFA #8 over all others because it read the input until $i_{30}$ and none of the other DFAs reached $i_{30}$
  - If two DFAs reach the same input character then pick the one that is listed first in the ordered list
Lexical Analysis using NFAs

1. a

2. b

3. a

4. b

5. b

6. c

7. b

8. a

9. b

TOKEN_A = a

TOKEN_B = abb

TOKEN_C = a*b+
Lexical Analysis using NFAs

TOKEN_A = a
TOKEN_B = abb
TOKEN_C = a*b+

Input: aaba

0a1a2b3a4

TOKEN_A matches 0,1
TOKEN_C matches 0,3

2013-09-24
Lexical Analysis using NFAs

Input: aaba

0, 1, 3, 7

Output:
TOKEN_C aab [0,3]
TOKEN_A a [3,4]
Lexical Analyzer using DFAs

- Each token is defined using a regexp $r_i$
- Merge all regexps into one big regexp
  - $R = (r_1 \mid r_2 \mid \cdots \mid r_n)$
- Convert $R$ to an NFA, then DFA, then minimize
  - remember orig NFA final states with each DFA state
Lexical Analyzer using DFAs

• The DFA recognizer has to find the *longest leftmost match* for a token
  – continue matching and report the last final state reached once DFA simulation cannot continue
  – e.g. longest match: `<print>` and not `<pr>`, `<int>`
  – e.g. leftmost match: for input string `aabaaaaaabb` the regexp `a+b` will match `aab` and not `aaaaaab`

• If two patterns match the same token, pick the one that was listed earlier in `R`
  – e.g. prefer final state (in the original NFA) of `r_2` over `r_3`
Lookahead operator

- Implementing $r_1/r_2$: match $r_1$ when followed by $r_2$
- e.g. $a^*b+/a^*c$ accepts a string $bac$ but not $abd$
- The lexical analyzer matches $r_1\varepsilon r_2$ up to position $q$ in the input
- But remembers the position $p$ in the input where $r_1$ matched but not $r_2$
- Reset to start state and start from position $p$
Summary

• Token $\Rightarrow$ Pattern
• Pattern $\Rightarrow$ Regular Expression
• Regular Expression $\Rightarrow$ NFA
  – Thompson’s Rules
• NFA $\Rightarrow$ DFA
  – Subset construction
• DFA $\Rightarrow$ minimal DFA
  – Minimization

$\Rightarrow$ Lexical Analyzer (multiple patterns)
Extra Slides
Efficient data-structures for DFAs
Implementing DFAs

• 2D array storing the transition table
• Adjacency list, more space efficient but slower
• Merge two ideas: array structures used for sparse tables like DFA transition tables
  – base & next arrays: Tarjan and Yao, 1979
  – Dragon book (default+base & next+check)
Implementing DFAs

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
Implementing DFAs

\[
\begin{array}{c|cccc}
& a & b & c & d \\
\hline
0 & - & 1 & - & 2 \\
1 & 1 & - & 1 & - \\
2 & 1 & 2 & 1 & - \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\hline
& - & 1 & - & 2 \\
\hline
1 & 2 & 1 & - & \\
1 & 2 & 1 & 1 & 1 & 2 & 1 & - \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 2 & 2 & 0 & 1 & 0 & 1 & - \\
\end{array}
\]

\[
\text{nextstate}(s, x) : \quad L := \text{base}[s] + x \\
\text{return next}[L] \quad \text{if check}[L] \text{ eq } s
\]
Implementing DFAs

nextstate(s, x) :
L := base[s] + x
return next[L] if check[L] eq s
else return nextstate(default[s], x)
Converting Regular Expressions directly into DFAs

This algorithm was first used by Al Aho in egrep, and used in awk, lex, flex
Regexp to DFA: \((ab) \mid (ba)\)^*#$
Regexp to DFA: followpos

• $\text{followpos}(p)$ tells us which positions can follow a position $p$

• There are two rules that use the $\text{firstpos}()$ and $\text{lastpos}()$ information

\[
\begin{align*}
\text{followpos}(i) & \mathbin{+}= \{k,l\} \\
\text{followpos}(j) & \mathbin{+}= \{k,l\}
\end{align*}
\]
Regexp to DFA: \(((ab) \mid (ba))^*\#$

\[
\begin{array}{c}
fp(2)+=1,3 \\
fp(4)+=1,3
\end{array}
\]

\[
\begin{array}{c}
fp(1)+=2 \\
fp(3)+=4
\end{array}
\]

\[
\begin{array}{c}
\{1\} a \{2\} b \{3\} b \{4\} a \\
(1) 1 (2) 2 (3) 3 (4) 4
\end{array}
\]

\[
\begin{array}{c}
root=\{1,3,5\} \\
fp(1)=2 \\
fp(3)=4 \\
fp(2)=\{1,3,5\} \\
fp(4)=\{1,3,5\}
\end{array}
\]
Regexp to DFA: \((ab) \mid (ba)\)^*\#
Minimization of DFAs
Minimization of DFAs

• Algorithm for minimizing the number of states in a DFA

• Step 1: partition states into 2 groups: accepting and non-accepting
Minimization of DFAs

• Step 2: in each group, find a sub-group of states having property P

• P: The states have transitions on each symbol (in the alphabet) to the same group

A, 0: blue  
A, 1: yellow  
E, 0: blue  
E, 1: yellow  
D, 0: yellow  
D, 1: yellow

B, 0: blue  
B, 1: yellow  
C, 0: blue  
C, 1: yellow
Minimization of DFAs

- Step 3: if a sub-group does not obey P split up the group into a separate group
- Go back to step 2. If no further sub-groups emerge then continue to step 4
Minimization of DFAs

• Step 4: each group becomes a state in the minimized DFA
• Transitions to individual states are mapped to a single state representing the group of states
Converting an NFA into a Regular Expression
NFA to RegExp

What is the regular expression for this NFA?
NFA to RegExp

- $A = aB$
- $B = bD \mid bC$
- $D = aB \mid \varepsilon$
- $C = aD$
NFA to RegExp

- Three steps in the algorithm (apply in any order):
  1. Substitution: for $B = X$ pick every $A = B \mid T$ and replace to get $A = X \mid T$
  2. Factoring: $(R \ S) \mid (R \ T) = R \ (S \mid T)$ and $(R \ T) \mid (S \ T) = (R \mid S) \ T$
  3. Arden's Rule: For any set of strings $S$ and $T$, the equation $X = (S \ X) \mid T$ has $X = (S^*) \ T$ as a solution.
NFA to RegExp

• A = a B
  B = b D | b C
  D = a B | ε
  C = a D
• Substitute:
  A = a B
  B = b D | b a D
  D = a B | ε

• Factor:
  A = a B
  B = ( b | b a ) D
  D = a B | ε

• Substitute:
  A = a ( b | b a ) D
  D = a ( b | b a ) D | ε
NFA to RegExp

\[ A = a (b | b a) D \]
\[ D = a (b | b a) D | \varepsilon \]

• Factor:
  \[ A = (a b | a b a) D \]
  \[ D = (a b | a b a) D | \varepsilon \]

• Arden:
  \[ A = (a b | a b a) D \]
  \[ D = (a b | a b a)^* \varepsilon \]

• Remove epsilon:
  \[ A = (a b | a b a) D \]
  \[ D = (a b | a b a)^* \]

• Substitute:
  \[ A = (a b | a b a) \]
  \[ (a b | a b a)^* \]

• Simplify:
  \[ A = (a b | a b a)^+ \]
NFA to Regexp using GNFAs

Generalized NFA: transition function takes state and regexp and returns a set of states

Algorithm:
1. Add new start & accept state
2. For each state $s$: rip state $s$ creating GNFA, consider each state $i$ and $j$ adjacent to $s$
3. Return regexp from start to accept state
NFA to Regexp using GNFAs

\[ a*b(a|b)^* \]
NFA to Regexp using GNFAs

Rip states 1, 2, 3 in that order, and we get: 

\[(a(aalb)^*ablb) ((bala)(aalb)^*ablb)^*((bala)(aalb)^*\varepsilon)la(aalb)^*\]