Lexical Analysis

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

• Example:

```c
double f = sqrt(-1);
```

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Symbol</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 “sqrt”
  - T_INTCONSTANT 1
- 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
Implementing Lexers: Loop and switch scanners

- Ad hoc scanners
- Big nested switch/case statements
- Lots of getc()/ungetc() calls
  - Buffering
- Can be error-prone, use only if
  - Your language’s lexical structure is simple
  - Tools don’t do what you want
- Changing or adding a keyword is problematic
- Key idea: separate the defn from the implementation
Formal Languages: Recap

- Symbols: a, b, c
- Alphabet := finite set of symbols $\Sigma = \{a, b\}$
- String: sequence of symbols bab
- Empty string: $\varepsilon$
- Set of all strings: $\Sigma^*$
Regular Expressions: Definition

• Every symbol of \( \Sigma \cup \{ \varepsilon \} \) is a regular expression
• If \( r_1 \) and \( r_2 \) are regular expressions, so are
  – Concatenation: \( r_1 r_2 \)
  – Alternation: \( r_1 | r_2 \)
  – Repetition: \( r_1^* \)
• Nothing else is.
  – Grouping re’s: e.g. aab|bc vs. ((aa)lb)c
Regular Expressions: Examples

• Alphabet \{ 0, 1 \}
• All strings that represent binary numbers divisible by 4 (but accept 0) \((0|1)*00|0\)
• All strings that do not contain “01” as a substring \(1*0*\)
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
Finite Automata: Recap

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

- What regular expression does this automaton accept?

Answer: \((0|1)^*00\)
FA: Pascal Example

- letter → 1 → letter| digit
- digit → 2 → digit
- { → 3 → }
- * → 4 → Any but }
- +,- → 5
- : → 6 = 7
- < → 8 > → 9 = A
- > → B = C
- = D
- . → E
- ; → F
- ( → G
- ) → H
Building a Lexical Analyzer

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
- NFA $\Rightarrow$ DFA
- DFA $\Rightarrow$ Lexical Analyzer
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 \( \varepsilon \), meaning states can be reached without reading any input, that is,
    \[ \delta: S \times \Sigma \cup \{ \varepsilon \} \Rightarrow 2^S \]
Thompson’s construction

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

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

- 0

- 1

(this version does not report errors: no sinkholes)
0|1
$(0|1)^*$
\((0|1)^*00\)
\(((0|1)^*00)\mid 0\)
Simulating NFAs

• 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
• Simulating NFAs is a problem that is closely linked to converting a given NFA to a DFA
NFA to DFA Conversion

• 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 $T' \leftarrow T$

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

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

push all states in \(T\) onto \textit{stack}
initialize \(\varepsilon\)-\textit{closure}(\(T\)) to \(T\)
\textbf{while} \textit{stack} is not empty \textbf{do} begin
  \textbf{pop} \(t\) off \textit{stack}
  \textbf{for} each state \(u\) with \(u \in \text{move}(t, \varepsilon)\) \textbf{do}
  \textbf{if} \(u \notin \varepsilon\)-\textit{closure}(\(T\)) \textbf{do} begin
    \text{add} \(u\) to \(\varepsilon\)-\textit{closure}(\(T\))
    \text{push} \(u\) onto \textit{stack}
  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

- Start state: $q_0$
- Input: $c_1, \ldots, c_k$

\[
T \leftarrow \epsilon\text{-closure}([q_0])
\]

\[
\text{for } i \leftarrow 1 \text{ to } k
\]

\[
T \leftarrow \text{DFAedge}(T, c_i)
\]
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
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 $U \notin Dstates$ then
      add $U$ to $Dstates$ unmarked
      $Dtrans[T, c] := U$;
  end
end
Subset Construction

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

p = j = 0

while \(j \leq p\) do begin

   for each symbol \(c\) do begin

      e = \(\text{DFAedge}(\text{states}[j], c)\)

      if e = states[i] for some \(i \leq p\)
          then Dtrans[j, c] = i
          else p = p + 1

      states[p] = e
      Dtrans[j, c] = p

   end

   j = j + 1

end
Example: subset construction
ε-closure(q₀)
move($\varepsilon$-closure($q_0$), 0)
$\varepsilon$-closure(move($\varepsilon$-closure($q_0$), 0))
move(ε-closure(q₀), 1)
\( \varepsilon\text{-closure}(\text{move}(\varepsilon\text{-closure}(q_0), 1)) \)
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 (I)
Regexp to DFA: \((ab \mid ba)^*\#\)

\[
\begin{array}{c}
\{1,3\} \\
(2,4)
\end{array}
\quad *
\quad \begin{array}{c}
\varepsilon\text{-node} \\
\begin{array}{c}
\{1,3\} \\
(2,4)
\end{array}
\quad \begin{array}{c}
\{1,3,5\} \\
(5)
\end{array}
\end{array}
\quad \begin{array}{c}
\# \\
\begin{array}{c}
\{5\}
\end{array}
\end{array}
\begin{array}{c}
\{1\}
\end{array}
\begin{array}{c}
\begin{array}{c}
(1)
\end{array}
\begin{array}{c}
a
\end{array}
\end{array}
\begin{array}{c}
\{2\}
\end{array}
\begin{array}{c}
\begin{array}{c}
b
\end{array}
\begin{array}{c}
(2)
\end{array}
\end{array}
\begin{array}{c}
\{3\}
\end{array}
\begin{array}{c}
\begin{array}{c}
b
\end{array}
\begin{array}{c}
(3)
\end{array}
\end{array}
\begin{array}{c}
\{4\}
\end{array}
\begin{array}{c}
\begin{array}{c}
a
\end{array}
\begin{array}{c}
(4)
\end{array}
\end{array}
\end{array}
\]

\text{firstpos} = \{\}
\quad \text{lastpos} = ()
Regexp to DFA: \textit{followpos}

- \textit{followpos} tells us which positions can follow a position \( k \)

- There are two rules that use the \textit{firstpos} \{\} and \textit{lastpos} () information

\[
\begin{align*}
\text{followpos}(i) &= k, l \\
\text{followpos}(j) &= k, l
\end{align*}
\]

\[
\begin{align*}
\text{followpos}(i) &= k, l \\
\text{followpos}(j) &= k, l
\end{align*}
\]

\[
\begin{align*}
\text{followpos}(i) &= k, l \\
\text{followpos}(j) &= k, l
\end{align*}
\]

\[
\begin{align*}
\text{followpos}(i) &= k, l \\
\text{followpos}(j) &= k, l
\end{align*}
\]

\[
\begin{align*}
\text{followpos}(i) &= k, l \\
\text{followpos}(j) &= k, l
\end{align*}
\]

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

- **fp(1)** += 2
- **fp(2)** += 1, 3
- **fp(3)** += 4
- **fp(4)** += 1, 3, 5

**root** = \{1, 3, 5\}
- **fp(1)** = 2
- **fp(3)** = 4
- **fp(2)** = 1, 3, 5
- **fp(4)** = 1, 3, 5
Regexp to DFA: \((ab \mid ba)^*\#

root = \{1, 3, 5\}
fp(1) = 2
fp(3) = 4
fp(2) = 1, 3, 5
fp(4) = 1, 3, 5

1:a
2:b
3:b
4:a
5:#
Regexp to DFA: (ab | ba) * #

$root = \{1, 3, 5\}$

$fp(1) = 2$

$fp(3) = 4$

$fp(2) = 1, 3, 5$

$fp(4) = 1, 3, 5$

$1: a$

$2: b$

$3: b$

$4: a$

$5: #$

$A: fp(5), \# \{\}, \# E, \#$

$B: fp(2), b \{1, 3, 5\}, b A, b$

$C: fp(4), a \{1, 3, 5\}, a A, a$

$A: fp(1), a \{2\}, a B, a$

$A: fp(3), b \{4\}, b C, b$

$A: fp(5), \# \{\}, \# E, \#$

$B: fp(2), b \{1, 3, 5\}, b A, b$

$C: fp(4), a \{1, 3, 5\}, a A, a$

$A: fp(1), a \{2\}, a B, a$

$A: fp(3), b \{4\}, b C, b$

$A: fp(5), \# \{\}, \# E, \#$
Equivalence of Regexps

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

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

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

- $(RS)^*R == R(SR)^*$
- $RS == (RS)$
- $R^* == RR^*|\varepsilon$
- $R == R|R$
- $R^* == RR^*|\varepsilon$
# NFA vs. DFA in the wild

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

• Each token is defined using a regexp $r_i$
• Merge all regexps into one big regexp
  – $R = (r_1 \ | \ r_2 \ | \ ... \ | \ r_n)$
• 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 match* for a token
  – e.g. `<print>` and not `<pr>`, `<int>`

• 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 $r_3$
Lexical Analyzer using DFAs

• Alternative method:
  – Organize all the DFAs for each token in an ordered list
  – For input \( i_1, i_2, \ldots, i_n \) run all DFAs until some reach a
    final state (pick the longest match for each DFA)
  – Pick the token for which some DFA could read 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
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

nextstate(s, x) :
L := base[s] + x
return next[L] if check[L] 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)
Summary

- Token $\Rightarrow$ Pattern
- Pattern $\Rightarrow$ Regular Expression
- Regular Expression $\Rightarrow$ NFA
  - Thompson’s Rules
- NFA $\Rightarrow$ DFA
  - Subset constructions
- DFA $\Rightarrow$ minimal DFA
  - Minimization

$\Rightarrow$ Lexical Analyzer (multiple patterns)