Lexical Analysis: Wrap Up
DFA minimization (revisited)

Important sentence in the book on how to perform split.

Refers to a set $p$ in the partition $P$.

“...creating one consistent state and lumping the rest of $p$ into another state will suffice.” pg. 55, EaC
DFA Minimization (the algorithm)

\[
P \leftarrow \{ D_F, \{D-D_F\}\}
\]

while ( \(P\) is still changing)

\[
T \leftarrow \emptyset
\]

\[
\text{for each set } p \in P
\]

\[
T \leftarrow T \cup \text{Split}(p)
\]

\[
P \leftarrow T
\]
DFA Minimization (the algorithm)

// S is a particular group in P

Split(S)
for each $\alpha \in \Sigma$
    // two states transition to diff groups in P
    if $\alpha$ splits S
        $s_1$ = a set w/ states internally consistent on $\alpha$
        $s_2$ = S-$s_1$
        return \{ $s_1$, $s_2$ \}
    return S

Internally consistent on $\alpha$:
$q_i$ and $q_j \in s$ st $\delta(q_i, \alpha) = q_x$ and $\delta(q_j, \alpha) = q_y$
$q_x$ and $q_y$ are in the same set
Building Faster Scanners

Hashing keywords versus encoding them directly

• Some *(well-known)* compilers recognize keywords as identifiers and check them in a hash table

• Encoding keywords in the DFA is a better idea
  → \(O(1)\) cost per transition
  → Avoids hash lookup on each identifier

*It is hard to beat a well-implemented DFA scanner:* While scanner generators can produce reasonably fast scanners, many compiler writers still hand-code scanners.
Building Scanners

The point

• All this technology lets us automate scanner construction
• Implementer writes down the regular expressions
• Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
• This reliably produces fast, robust scanners

For most modern language features, this works

• You should think twice before introducing a feature that defeats a DFA-based scanner
• The ones we’ve seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting
What we expect of the Scanner

• **Report errors** for lexicographically malformed inputs
  → reject illegal characters, or meaningless character sequences
  → E.g., “lo#op” in COOL

• Return an **abstract representation** of the code
  → character sequences (e.g., “if” or “loop”) turned into **tokens**.

• Resulting sequence of tokens will be used by the parser

• Makes the design of the parser a lot easier.
How to specify a scanner

- A scanner specification (e.g., for JLex), is list of (typically short) regular expressions.
- Each regular expression has an action associated with it.
- Typically, an action is to return a token.
How to specify a scanner (cont'd)

• On a given input string, the scanner will:
  → find the longest prefix of the input string, that matches one of the regular expressions.
  → will execute the action associated with the matching regular expression highest in the list.

• Scanner repeats this procedure for the remaining input.

• If no match can be found at some point, an error is reported.
Example of a Specification

• Consider the following scanner specification.
  1. aaa { return T1 }
  2. a*b { return T2 }
  3. b { return S }

• Given the following input string into the scanner
  aaabbaaa
Example of a Specification

• Consider the following scanner specification.
  1. aaa  { return T1 }
  2. a*b  { return T2 }
  3. b    { return S }

• Given the following input string into the scanner
  aaab  b  aaa
  T2   T2   T1

• Note that the scanner will report an error for example on the string 'aa'.
What can be so hard?

Poor language design can complicate scanning

- **Reserved words are important**
  
  if then then then = else; else else = then  \((PL/I)\)

- **Insignificant blanks**  \((Fortran & Algol68)\)
  
  do 10 i = 1,25   (this is a loop)
  do 10 i = 1.25  (this is an assignment to variable “do10i“)

Note: This is handled by performing an initial pass to insert “significant” blanks.
What can be so hard? (cont’d)

- String constants w/ special (“escape”) characters (C, C++, Java, …)
  - newline, tab, quote, comment delimiters, …

- Finite closures (Fortran 66 & Basic)
  - Limited identifier length
  - Adds states to count length
Limits of Regular Languages

Advantages of Regular Expressions

• Simple & powerful notation for specifying patterns
• Automatic construction of fast recognizers
• Many kinds of syntax can be specified with REs

Example — an expression grammar

\[\text{Term} \rightarrow [a-zA-Z] ([a-zA-Z] | [0-9])^*\]
\[\text{Op} \rightarrow + | - | \ast | /\]
\[\text{Expr} \rightarrow (\text{Term Op})^* \text{Term}\]

Of course, this would generate a DFA ...

If REs are so useful ...

\textit{Why not use them for everything?}
Limits of Regular Languages

Not all languages are regular

\( RL's \subset CFL's \subset CSL's \)

You cannot construct DFA’s to recognize these languages

- \( L = \{ p^k q^k \} \) (parenthesis languages)
- \( L = \{ w^r \mid w \in \Sigma^* \} \) (finite closures)

Neither of these is a regular language (nor an RE)
Limits of Regular Languages

But, this is a little subtle. You can construct DFA’s for

- Strings with alternating 0’s and 1’s
  \[(\varepsilon | 1)(01)^*(\varepsilon | 0)\]

- Strings with an even number of 0’s and 1’s
  \[(00)^*(11)^*(00)^*\]
  0011, 1100, 1111, 0000, 110000, 001111, …