RE and FA APPLICATION: SCANNERS

Groups chars into tokens, removes comments and "white space", saves text where needed (e.g. identifiers, strings, numbers), adds line or col numbers for better err. msgs.

Ad Hoc Code, often used in production, fast, efficient.

FA or RE $\rightarrow$ FA: easy to write code from automatically, in fact can automate much of the process.

Code: handwritten: generate `switch (case, goto)` statements, as seen rather earlier.

OR: automatically generated: use *tables* and a driver (as in `lex/flex`.) Or maybe produce tables for handwritten driver.

NESTED CASE SCANNING

Outer cases covers FA states. Inner cases cover transitions out of each state: most go to new state, some return with current token.

Two additions to pure FA: keywords and "peek-aheads" (e.g. for 3.14 vs 3..14 in Pascal).

Key (reserved) words look like identifiers. Telling them apart with scanner takes lots of states. So easiest to depart from formalism, ignore distinction until token is generated, then look it up (hash or trie) to see if it's a keyword.

We've seen the FORTRAN peek-ahead problem `DO100I=1.10`. If not too common, can treat these problems as special case hacks. But some languages just need more lookahead: scanner assumes longer token possible but remembers shorter ones along the way, buffers chars. Good syntax design can minimize these complications!

TABLE-DRIVEN SCANNING

Represent FA by it's two-dimensional (state and character) transition table. Table entries say whether to transition to new state and if so to which one, whether to return a token or error.

A second table indicates for each state whether we might be at the end of a token (and if so which): this table – when pass state that might have been end of a token can back up if hit error later on.

Other issues: Lexical error handling (often toss invalid token, skip to possible start of next, and keep scanning, thus punting to the parser for bad syntax. Pragmas are *significant comments* that control or hint to compiler (turn profiling on, put variable in register, routine not recursive...).

SMALL TABLE-DRIVEN E.G.

$letter \rightarrow (a \mid b \mid \ldots \mid z \mid A \mid \ldots \mid Z)$
$digit \rightarrow (0 \mid 1 \mid \ldots 8 \mid 9)$
$id \rightarrow letter(letter \mid digit)*$

TABLES

char-class	a-z	A-Z	0-9	other
VALUE	letter	letter	digit	other

next_state	CLASS	S0	S1	S2	S3
	letter	S1	S1	--	--
	digit	S3	S1	--	--
	other	S3	S2	--	--