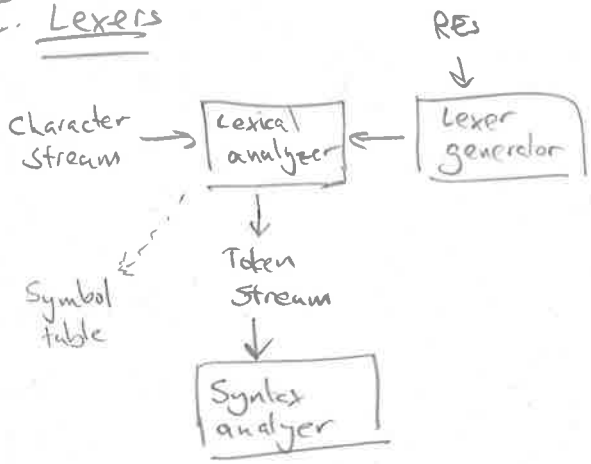
Lexical analyzer (pg. 1)

Lecture topics:

I. Lexers
II. Regular expressions
III. Finite automatons

I. Lexers



Task of lexer:
- find tokens
- find token attributes
- strip whitespace and comments

Why separate lexer from parser?
- more modular design
- may use more efficient algorithms
- may hide input format

Some tools are scannerless
- do not separate lexer from parser
- altho can be scannerless, but we won't use it that way

II. Regular expressions

(RE example: $(a|b)^* abb$
accepted strings: abb, aabb, abbabb ...)

	Regular Expressions	Context free grammars
Basic constructs	sequence (e.g. ab) alternation (e.g. a|b) empty string (ε) grouping ((...)) closure (...*) aka. Kleene star, repetition	sequence alternation empty string (ε) recursion
Extended constructs	character classes e.g. [_|t|n] [a-zA-Z0-9] helper rules, but no recursion +, ?, -, !, &	grouping closure +, ?

Can emulate every RE with a CFG, e.g.

RE: $(a|b)^* abb$

CFG:
$X \to Y\,abb$
$Y \to YZ \mid \varepsilon$
$Z \to a|b$

$\Rightarrow$ CFGs are <u>at least as</u> powerful as REs.

Cannot evaluate every CFG with an RE, e.g.:

CFG:
$X \to \{X\} \mid y$

accepted strings: $y, \{y\}, \{\{y\}\}, \{\{\{y\}\}\} \ldots \{^n y\}^n$

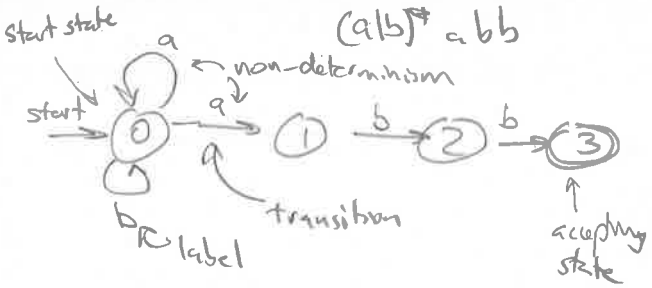
RE: cannot "count" parentheses

$\Rightarrow$ CFGs are <u>more</u> powerful than REs.

Lexical analyzer (pg 2)

III Finite automatons

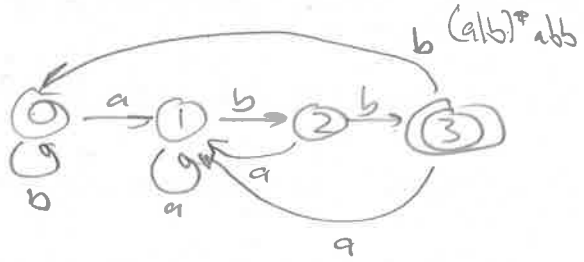
NFA = non-deterministic finite automaton



Transition table:

State	a	b	ε
0	{0,1}	{0}	{ }
1	{ }	{2}	{ }
2	{ }	{3}	{ }
3	{ }	{ }	{ }

DFA = deterministic finite automaton



Transition table

State	a	b
0	1	0
1	1	2
2	1	3
3	1	0

	DFA	NFA
labels from same state	unique complete	may be same may be incomplete
ε-labels	no	yes
transition table	states	state sets
expressiveness	same as REs	same as REs
tradeoffs	easy to simulate	easy to come up with