Lec 9 Parsing II: Why LR(1) & how parsers work

→ Try implement parser by hand

LR(1) balances:
- parse efficiency
- parser generation efficiency
- expressiveness

Parse efficiency

Ex. $\Sigma = \{a, b, c\}$

G (grammar) =

[1] $S \to aBSc$

[2] $S \to abc$

[3] $Ba \to aB$

[4] $Bb \to bb$

$L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}\}$

└ not context free! We need look around to decide if a rule can apply

▷ Chomsky hierarchy:

recursively enumerable	$\alpha \to \beta$	TM	undecidable	
context-sensitive	$\alpha \to \beta$ $\lvert\alpha\rvert \le \lvert\beta\rvert$	linear bounded TM	PSPACE complete	$a^n b^n c^n$
context-free	$A \to \gamma$	non-deterministic pushdown (automata with stack)	cubic	$a^n b^n$
regular	$A \to a$ $A \to ab$	NFA / DFA	linear	$a^* b^*$

▷ Grammar for parsing:
regular not expressive enough, context sensitive too slow
LR(1) between regular and context-free, and gives linear time
↓

▷ Deterministic context-free languages

Thm Language L can be defined by a deterministic pushdown automaton iff L can be defined by LR(k) grammar iff L can be defined by LR(1) grammar

LL(k) can also parse, but we don't have a LL(k) to LL(1) reduction

Parser Generation

Goal: find $S \to^* w_0$, keeping states $\gamma \| w_i$
starting $\varepsilon \| w_0$ ending $S \| \varepsilon$
transit by shift or reduction

Ex. $S \xrightarrow{emp} \varepsilon$ $S \xrightarrow{next} [S]S$

Situations to consider in parse table
1. $\gamma = \varepsilon$, try derive S
2. $\gamma = \gamma'[$, need to derive $S]S$
3. $\gamma = \gamma'[S$, need .. $]S$
4. $\gamma = \gamma'[S]$, need .. S
5. $\gamma = \gamma'[S]S$, should be done

Table filling:
- look at next token
- decide shift or reduce by rule or raise error

1. Compute prefix set
   └ $\text{prefix}(S) = \{a \mid \exists w \in \Sigma^*, S \to^* aw\} = \{[\}$
2. For each situation, consider each next token a and prefix set P of what we want to derive
   └ if $a \in P$, shift
   └ else if can reduce to exactly one other state in table, reduce
   └ else :(

γ	(want)	next tkn a: [	]	EOF
ε	S	shift	red(emp)	red(emp)
$\gamma'[$ ε	$S]S$	shift	red(emp)	red(emp)
$\gamma'[S$ ε	$]S$	err	shift	err
$\gamma'[S]$ ε	S	shift	red(emp)	red(emp)
$\gamma'[S]S$ ε	ε	(err)*	red(next)	red(next)
εS	ε	err	err	ACC

* can technically go to εS, so alg would split the line into cases