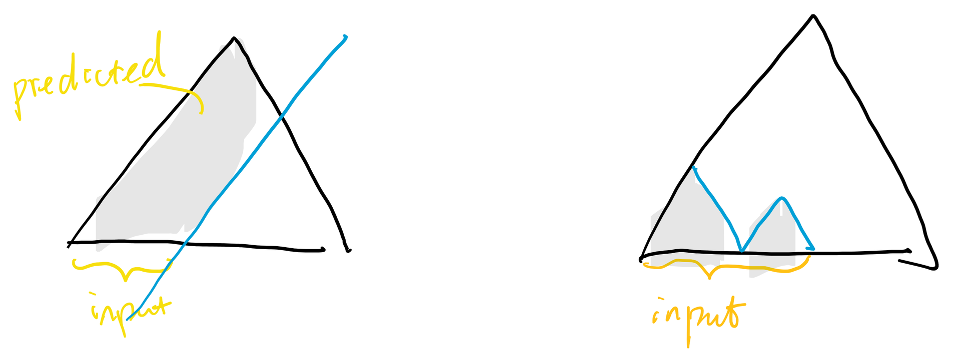
Bottom-Up Parsing

Last time: top-down parsers
    somewhat readable code, but there are important grammars it cannot handle

Next two lectures: bottom-up parsers
    today: a parser that can handle all grammars, as powerful as you can hope for
    next time: LR parsing (underlying technology of most parser generators)

Top-down parsing:
    must predict production before seeing input that goes into that production

Bottom-up parsing
    get to choose productions after seeing input



Earley parsing

bottom-up parser that chooses productions as late as it possible can

- handles all CFGs, including ambiguous ones
- On ambiguous grammars, gives "parse forest"

Straw man:
    For each possible production, fork off a new parser thread

$S \to S+E \mid E$

Idea:
    simulate threads in polynomial time
    by tracking all possible parses as sets of *items*

the input position where the item was created

Earley item $[A \to \beta \bullet \gamma,\ i]$

current input position

Earley Parser
- process input left-to-right, one token at a time
- for each input position j, compute a set of items I_j
- start from $I_0 = \{ [\ S' \to \bullet\, S,\ 0\ ] \}$, where S is the start symbol
- grammar accepts input $a_0 a_1 \ldots a_n$ if I_n contains $[\ S' \to S\, \bullet,\ 0\ ]$

How to compute set I_j?

    For each input position j, do until no changes to I_j are possible:

Predict: $[A \to \beta \bullet C\delta,\ k] \in I_j \Rightarrow$ add $[C \to \bullet \gamma,\ j]$ to I_j where $C \to \gamma \in G$

Complete: $[C \to \gamma \bullet,\ k] \in I_j$ and $[A \to \beta \bullet C\delta,\ m] \in I_k$
$\Rightarrow$ add $[A \to \beta C \bullet \delta,\ m]$ to I_j

Scan: $[A \to \beta \bullet c\delta,\ k] \in I_j$ and next token is $a_j = c$
$\Rightarrow$ add $[A \to \beta c \bullet \delta,\ k]$ to I_{j+1}

Example

$S \to S + E \mid E$ // left-recursive
$E \to n \mid (\ S\ )$

input = (1) + 2

I_0:
- $[S' \to \bullet S, 0]$
- $[S \to \bullet S{+}E, 0]$
- $[S \to \bullet E, 0]$
- $[E \to \bullet n, 0]$
- $[E \to \bullet (S), 0]$

I_1:
- $[E \to (\bullet S), 0]$
- $[S \to \bullet S{+}E, 1]$
- $[S \to \bullet E, 1]$
- $[E \to \bullet n, 1]$
- $[E \to \bullet (S), 1]$

I_2:
- $[E \to n \bullet, 1]$
- $[S \to E \bullet, 1]$
- $[E \to (S \bullet), 0]$
- $[S \to S \bullet {+}E, 1]$

I_3:
- $[E \to (S) \bullet, 0]$
- $[S \to E \bullet, 0]$
- $[S' \to S \bullet, 0]$
- $[S \to S \bullet {+}E, 0]$

I_4:
- $[S \to S{+} \bullet E, 0]$
- $[E \to \bullet n, 4]$
- $[E \to \bullet (S), 4]$

I_5:
- $[E \to n \bullet, 4]$
- $[S \to S{+}E \bullet, 0]$
- $[S' \to S \bullet, 0]$
- $[S \to S \bullet {+}E, 0]$

$O(n^3)$ for arbitrary grammar
$O(n^2)$ for nonamb grammar
$O(n)$ for LR(k) grammar (w. some optimizations)

$S' \to S \to S{+}E \to S+2 \to E+2 \to \cdots$

Predict: $[A \to \beta \bullet C\delta,\ k] \in I_j \Rightarrow$ add $[C \to \bullet \gamma,\ j]$ to I_j where $C \to \gamma \in G$

Complete: $[C \to \gamma \bullet,\ k] \in I_j$ and $[A \to \beta \bullet C\delta,\ m] \in I_k$ and $a_j \in Follow(C)$
$\Rightarrow$ add $[A \to \beta C \bullet \delta,\ m]$ to I_j

Scan: $[A \to \beta \bullet c\delta,\ k] \in I_j$ and next token is $a_j = c$
$\Rightarrow$ add $[A \to \beta c \bullet \delta,\ k]$ to I_{j+1}

rightmost derivation:
$S' \to S \to S + E \to S + 2 \to E + 2 \to (S) + 2 \to (E) + 2 \to (1) + 2$

LR parsers

Can be viewed as **specialization** of Earley parser:
- Still a bottom-up parser
- Precompute "all predictions"
  I.e., Precompute a parsing table (a DFA, really) that the parser can look up to decide if it should scan or complete

LR(k) vs. Earley:

Earley: explores all choices; handles all CFGs
LR(k): must decide scan/complete; handles a subset of CFGs

LR(k) a.k.a. "shift-reduce parsers"

shift = scan + take prediction closure
reduce = complete + take prediction --

Example

$S \to S + E \mid E$ // left-recursive; not an LL(k) grammar
$E \to n \mid (\ S\ )$

input = (1) + 2

action	stack	unconsumed input
		(1) + 2
shift	(	1) + 2
shift	(1	) + 2
reduce $E \to n$	(E	) + 2
reduce $S \to E$	(S	) + 2
shift	(S)	+ 2
reduce $E \to (S)$	E	+ 2
reduce $S \to E$	S	+ 2
shift	S +	2
shift	S + 2	ε
reduce $E \to n$	S + E	ε
reduce $S \to S{+}E$	S	ε

How to decide whether to shift or reduce?
    use a precomputed parsing table

What does this parsing table do?
    Given current stack, lookahead c, tells whether to shift or reduce
    If reducing, tells which production to reduce

    (stack is known as the parser state σ: stores part of derivation to "left of •")

constructed parsing table may be ambiguous:
- shift–reduce conflict
- reduce–reduce conflict

Recap

Earley:
- interprets the grammar
- derives all parses ⇒ works for all grammars, even ambiguous ones

LR parsers:
- Unlike Earley which basically interprets a CFG, they precompute a parsing table for efficient parsing.
- Trade-off