LL Parsing

Core concepts from last time:

- Derivation: sequence of rewrites from start symbol to token stream

  Each step: $\alpha X\beta \to \alpha\gamma\beta$, where $X \to \gamma$ is a production

- Leftmost derivation: expand leftmost nonterminal
- Rightmost derivation: expand rightmost nonterminal

- ambiguous grammar:
  Grammar G is ambiguous if some string in L(G) has >1 parse trees.

This lecture: LL parsing (aka recursive-descent parsing)

Ex/ recursive-descent parser for S-expressions

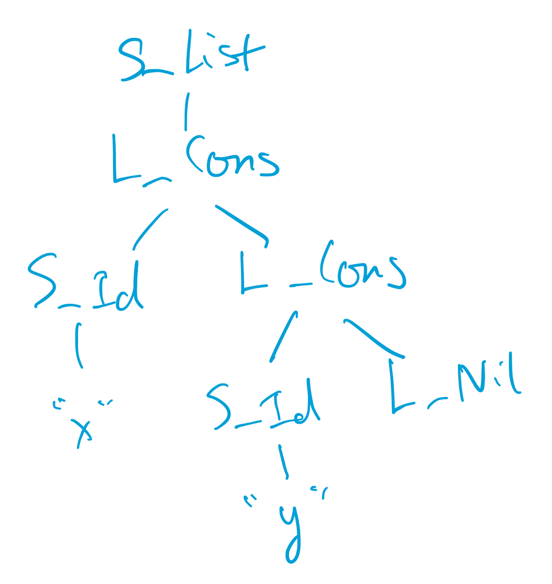
$$\boxed{\begin{array}{l} S \to (\, L\,) \mid x \\ L \to \varepsilon \mid S\, L \end{array}}$$

Code almost direct from the grammar:
- nonterminal -> mutually recursive functions
- production -> switch cases
- parse tree -> call tree

```
SExpr
void parseS() {
  switch (peek()) {
    case LPAREN:    // S → (L)
      consume(LPAREN);
LExpr l = parseL();
      consume(RPAREN);
      return; S_List(l)
    case ID:        // S → x
Token id = consume(ID);
      return; S_Id(id.attribute())
    default:
      throw new SyntaxError(peek());
  }
}
LExpr
void parseL() {
  switch (peek()) {
    case ID:     // L → S L
    case LPAREN: // L → (L)
SExpr s = parseS();
LExpr l = parseL();
      return; L_Cons(s, l)
    case RPAREN: // L → ε
      return; L_nil
    default:
      throw new SyntaxError(peek());
  }
}
```

data SExpr :=
| S_Id : String → SExpr
| S_List : LExpr → SExpr
with LExpr :=
| L_Nil : LExpr
| L_Cons : SExpr → LExpr → LExpr

" (x, y) "



Predicative parsing table (PPT)

General recipe for mechanically producing recursive-descent parsers

PPT tells parser what to do next in derivation based on
- expected nonterminal and
- lookahead

$S \to (L) \mid x$
$L \to \varepsilon \mid S L$

	(	x	)
S	S→(L)	S→x	error
L	L→SL	L→SL	L→ε

Req: each cell contains at most one production.

Def/ A grammar is LL(1) if 1-lookahead PPT can be constructed.

$\Sigma \qquad |\Sigma|^k$

Constructing PPT

Recall notational convention:
  uppercase (X): nonterminals
  lowercase (x): terminals
  Greek (αβγ): strings of symbols (symbol = nonterminal/terminal)

Nullable(X): whether X can derive ε   Nullable(γ)
First(X): set of tokens that can begin expansion of X   First(γ)
Follow(X): set of tokens that can follow X in a derivation

Example:
$S \to (\,L\,) \mid x$
$L \to \varepsilon \mid S\,L$

Nullable(S) = false   Nullable(L) = true
First(L) = { "(", x }
Follow(L) = { ")" }
First("(L)") = { "(" }

Cell(X, a) in PPT contains
- every production X→γ where a ∈ First(γ)
- every production X→γ where a ∈ Follow(X) and Nullable(γ)

Nullable
Proof rules and algorithms

$$\frac{X\to\gamma \in G \quad \text{Nullable}(\gamma)}{\text{Nullable}(X)} \qquad \frac{}{\text{Nullable}(\varepsilon)} \qquad \frac{\text{Nullable}(X) \quad \text{Nullable}(\beta)}{\text{Nullable}(X\beta)}$$

Equations

$$\text{Nullable}(X) = \bigvee_{X\to\gamma\in G} \text{Nullable}(\gamma)$$

$\text{Nullable}(\varepsilon) = \text{true} \qquad \text{Nullable}(X\beta) = \text{Nullable}(X) \wedge \text{Nullable}(\beta)$

$\text{Nullable}(a\beta) = \text{false}$

Iterative solving alg

1. Initialize nullable map   nullable = {X ↦ false, Y ↦ false, ... }
2. Repeat until no more change to nullable map is possible.
   for each nonterminal X ∈ G :
     update nullable [X] per X's equation.
3. return nullable.

Partial correctness. by induction.   termination. main loop ≤ N iterations (N = # non-term.)

Worklist alg

1. Initialize nullable map --
2. Initialize worklist $w = \{ X \mid X \to \varepsilon \in G \}$
   invariant. X's equation not satisfied ⇒ X ∈ w
3. while w not empty :
   remove some X from w.
   update Nullable [X] per its equation.
   if Nullable [X] changed :
     for each $Z \to \alpha X \beta \in G$
       add Z to w.

$O(N^2)$
$O(M)$
$O(MN^2)$

First
Proof rules and algorithms

$$\frac{X\to\gamma\in G \quad a\in\text{First}(\gamma)}{a\in\text{First}(X)} \qquad \text{First}(X) = \bigcup_{X\to\gamma\in G}\text{First}(\gamma)$$

$$\frac{}{a\in\text{First}(a\beta)} \qquad \frac{a\in\text{First}(X)}{a\in\text{First}(X\beta)} \qquad \frac{\text{Nullable}(X) \quad a\in\text{First}(\beta)}{a\in\text{First}(X\beta)}$$

First(ε) = ∅
First(aβ) = { a }
First(Xβ) = First(X) , if Nullable(X) = false
First(Xβ) = First(X) ∪ First(β)   otherwise

$S \to (L) \mid x$
$L \to \varepsilon \mid S L$

# iter	First [S]	First [L]
0	∅	∅
1	"(", x	∅
2	"(", x	"(", x
3	...	...

First [S] = First("(" L ")") ∪ First(x) = { "(", x }
First [L] = First [S]

Follow
Proof rules and algorithms

$$\frac{Y\to\alpha X\beta\in G \quad a\in\text{First}(\beta)}{a\in\text{Follow}(X)} \qquad \frac{Y\to\alpha X\beta\in G \quad \text{Nullable}(\beta) \quad a\in\text{Follow}(Y)}{a\in\text{Follow}(X)}$$

$S \to (L) \mid x$
$L \to \varepsilon \mid S L$

# iter	Follow [S]	Follow [L]
0	∅	∅
1	"(", x	")"
2	"(", x, ")"	")"
3	...	...

Follow [S] = First(L) ∪ Follow [L]
Follow [L] = { ")" } ∪ Follow [L]

Construct PPT

Cell(X, a) in PPT contains
- every production X→γ where a ∈ First(γ)
- every production X→γ where a ∈ Follow(X) and Nullable(γ)

Cell(X, a) empty?

raise syntax error

Cell(X, a) has more than one productions?

G is not LL(1)

Tricks

Shared First sets

L → S | S L $L \to S L'$
        $L' \to \varepsilon \mid L$

Left recursion

L → S | L + S   not LL(k) for any k

First(L+S) ⊇ First(L) ⊇ First(S)

$L \to S(+S)^*$

```
Expr parseL() {
  Expr left = parseS()
  while (peek() = PLUS) {
    consume(PLUS)
    Expr right = parseS()
    left = ExprPlus(left, right)
  }
  return left
}
```