Context-Free Grammars

- Adds recursion/allows non-terminals to be expressed in terms of themselves
- Can be used to count/impart structure – e.g., nested parentheses
- Notation – grammar G(S,N,T,P)
  - S is the start symbol
  - N is a set of non-terminal symbols (LHS)
  - T is a set of terminal symbols (tokens)
  - P is a set of productions or rewrite rules
    \( P : N \Rightarrow N \cup T \)

Derivations

- A sequence of application of the rewrite rules is a derivation or a parse (e.g., deriving the string \( x + 2 - y \))
- The process of discovering a derivation is called parsing

Parse Trees

A parse tree for a grammar G is a tree where
- The root is the start symbol for G
- The interior nodes are non-terminals of G
- The leaf nodes are terminal symbols of G
- The children of a node T (from left to right) correspond to the symbols on the right hand side of some production for T in G

Every terminal string generated by a grammar has at least one corresponding parse tree; every valid parse tree represents a string generated by the grammar (yield of the parse tree)

Advantages of CFGs

- Precise syntactic specification of a programming language
- Easy to understand, avoids ad hoc definition
- Easier to maintain, add new language features
- Can automatically construct efficient parser
- Parser construction reveals ambiguity, other difficulties
- Imparts structure to language
- Supports syntax-directed translation

Calculator Example

- All variables are integers
- There are no declarations
- The only statements are assignments, input, and output
- Expressions use one of four arithmetic operators and parentheses
- Operators are left associative, with the usual precedence
- There are no unary operators
Regular Expressions

id → letter ( letter | digit)*
literal → digit digit*
read, write, "="", "+", "-", "+", "/", "("", ")"$$ */ end of input */

Grammar for Calculator

<pgm> → <stmtlist> $$
<stmtlist> → <stmtlist> <stmt> |
<stmt> → id := <expr> | read <id> | write <expr>
<expr> → <term> | <expr> <addop> <term>
<term> → <factor> | <term> <multop> <factor>
<factor> → ( <expr> ) | id | literal
<addop> → + | -
<multop> → * | /

Types of derivations

• Leftmost derivation
  – The leftmost non-terminal is replaced at each step
• Rightmost derivation
  – The rightmost non-terminal is replaced at each step
• Ambiguous grammar – one with multiple leftmost (or multiple rightmost) derivations for a single sentential form

Types of parsers

• Top-down (LL) parsers
  – Left to right, leftmost derivation
  – Starts at the root of the derivation tree and fills in
  – Predicts next state with n lookahead
• Bottom-up (LR) parsers
  – Left to right, rightmost derivation
  – Starts at the leaves and fills in
  – Start with input string, end with start symbol
  – Starts in a state valid for legal first tokens
  – Change state to encode possibilities as input is consumed
  – Use a stack to store both state and sentential form

Top-Down Parsing

• At a node labeled A, select a production with A on its LHS and for each symbol on its RHS, construct the appropriate child
• When a terminal is added that does not match the input, backtrack
• Find the next node to be expanded (must have a label in NT)

Eliminating Left Recursion

• A grammar is left recursive if there exists A in NT such that A → Aδ for some string δ
• Transform the grammar to remove left recursion
  <foo> → <foo> δ
  | μ
  →
  <foo> → μ <bar>
  <bar> → δ <bar> | ε
  where <bar> is a new non-terminal
Eliminating Common Prefixes

foo → bar δ
   → bar ( μ )

foo → bar footail
footail → δ | ( μ )

LL Grammar for Calculator

<pgm> → <stmtlist> $$
<stmtlist> → <stmt> <stmtlist> | ε
<stmt> → id := <expr> | read <id> | write <expr>
<expr> → <term> <termtail>
<termtail> → <addop> <term> <termtail> | ε
<term> → <factor> <factortail>
<factortail> → <multop> <factor> <factortail> | ε
<factor> → ( <expr> ) | id | literal
<addop> → + | -
<multop> → * | /

Parser Construction

• Recursive descent parsing
  – Top-down parsing algorithm
  – Built on procedure calls (may be recursive)
  – Write procedure for each non-terminal, turning each production into clause
  – Insert call to procedure A() for non-terminal A and to match(x) for terminal x
  – Start by invoking procedure for start symbol S

Predictive (Table-Driven) Parser

The FIRST Set

• FIRST(α) is the set of terminal symbols that begin strings derived from α
• To build FIRST(X):
  – If X is a terminal, FIRST(X) is {X}
  – If X=ε, then ε ∈ FIRST(X)
  – If X–Y1 Y2 … Yk then put FIRST(Y1) in FIRST(X)
  – If X is a non-terminal and X–ε Y1 Y2 … Yk, then ε ∈ FIRST(X) if a ∈ FIRST(Y1) and ε ∈ FIRST(Y1), for all 1≤i
The Follow Set

- For a non-terminal A, FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form
- To build FOLLOW(B) for all B:
  - Starting with goal, place eof in FOLLOW(<goal>)
  - If \( A \rightarrow \alpha \beta \), then put \( \{ \text{FIRST}(\beta) - \varepsilon \} \) in FOLLOW(B)
  - If \( A \rightarrow \alpha B \), then put FOLLOW(A) in FOLLOW(B)
  - If \( A \rightarrow \alpha \beta \) and \( \varepsilon \in \text{FIRST}(\beta) \), then put FOLLOW(A) in FOLLOW(B)

Using FIRST and FOLLOW

- For each production \( A \rightarrow \alpha \) and lookahead token
  - Expand A using the production if token \( \varepsilon \in \text{FIRST}(\alpha) \)
  - If \( \varepsilon \in \text{FIRST}(\alpha) \), expand A using the production if token \( \varepsilon \in \text{FOLLOW}(A) \)
  - All other tokens return error
- If there are multiple choices, the grammar is not LL(1) (predictive)

LL(1) Grammars

A Grammar G is LL(1) if and only if, for all non-terminals A, each distinct pair of productions \( A \rightarrow \alpha \) and \( A \rightarrow \beta \) satisfy the condition \( \text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \Phi \), i.e.,

For each set of productions \( A \rightarrow \alpha_1 \alpha_2 | \ldots | \alpha_n \)

- \( \text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \ldots, \text{FIRST}(\alpha_n) \) are all pairwise disjoint
- If \( \alpha_i \notin \varepsilon \) for any i, then \( \text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \Phi \), for all \( j \neq i \)

The Complexity of LL(1) Parsing

- Inside main loop – bounded by constant (function of symbols on RHS)
- How many times does the main loop execute?
  - Number of iterations is the number of nodes in the parse tree, which is \( N^*P \) (\( N \) is the number of tokens in the input, \( P \) is the number of productions)
  - \( P \) is a constant, therefore running time is \( O(N) \)

CFGs versus Regular Expressions

- CFGs strictly more powerful than REs
  - Any language that can be generated using an RE can be generated by a CFG (proof by induction)
  - There are languages that can be generated by a CFG that cannot be generated by an RE (proof by contradiction)

Example non-LL Grammar

Construct

\[
\text{stmt} \rightarrow \text{if condition thenClause elseClause} \\
| \text{otherStmt}
\]

\[
\text{thenClause} \rightarrow \text{thenstmt} \\
\text{elseClause} \rightarrow \text{elsestmt} \mid \varepsilon
\]

\( \Rightarrow \) Ambiguous – allows dangling else to be paired with either then in if A then if B then C else D
Fix – Bottom-up parsing (LR)

<table>
<thead>
<tr>
<th>stmt</th>
<th>balanced_stmt</th>
<th>unbalanced_stmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>balanced_stmt</td>
<td>if condition then balanced_stmt</td>
<td>else balanced_stmt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other_stmt</td>
</tr>
<tr>
<td>unbalanced_stmt</td>
<td>if condition then stmt</td>
<td>if condition then balanced_stmt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else unbalanced_stmt</td>
</tr>
</tbody>
</table>

OR

Use special disambiguating rules → use production that occurs first in case of conflict