Context-Free Grammars

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 ser of productions or rewrite rules (P : N → N U 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> | ε
<stmt> → id := <expr> | read <id> | write <expr>
<expr> → <term> | <expr> <add op> <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)

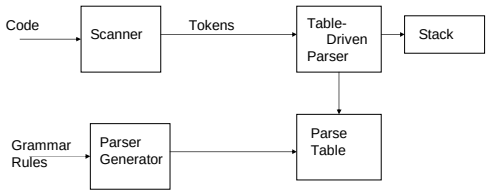
LL Grammar for Calculator

<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



Predictive (Table-Driven) Parsing

- Actions –
  - Match a terminal
  - Predict a production
  - Announce a syntax error
- Push as yet unseen portions of productions onto a stack
- Use –
  - FIRST (A)
  - FOLLOW(A)

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 ε FIRST(X)
  - If X←Y1 Y2 … Yk then put FIRST(Y1) in FIRST(X)
  - If X is a non-terminal and X←Y1 Y2 … Yk, then a ε FIRST(X) if a ε FIRST(Yi) and ε ε FIRST(Yj), for all 1≤j<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←αBβ, then put {FIRST(β)-ε} in FOLLOW(B)
  - If A←αB, then put FOLLOW(A) in FOLLOW(B)
  - If A←αBβ and ε ε FIRST(β), then put FOLLOW(A) in FOLLOW(B)

Using FIRST and FOLLOW

- For each production A←α and lookahead token
  - Expand A using the production if token ε FIRST(α)
  - If ε ε FIRST(α), expand A using the production if token ε 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 ← α and A←β satisfy the condition FIRST(α) ∩ FIRST(β) = Φ, i.e.,

For each set of productions A← α1|α2|…|αn

- FIRST(α1), FIRST(α2), …, FIRST(αn) are all pairwise disjoint
- If α_i ←ε for any i, then FIRST(α_j) ∩ FOLLOW(A) = Φ, for all j≠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

stmt → if condition then_clause else_clause
| other_stmt

then_clause → then stmt

else_clause → else stmt | ε

➔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)

stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if condition then balanced_stmt
else balanced_stmt
| other_stmt
unbalanced_stmt → if condition then stmt
| if condition then balanced_stmt
else unbalanced_stmt

OR

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