Context Free Grammars

Sr. Nisha C D
Assistant Professor, Dept. of Computer Science
Little Flower College, Guruvayoor

Context Free Grammars

- One or more non terminal symbols
  - Lexically distinguished, e.g. upper case
- Terminal symbols are actual characters in the language
  - Or they can be tokens in practice
- One non-terminal is the distinguished start symbol.

Grammar Rules

- Non-terminal ::= sequence
  - Where sequence can be non-terminals or terminals
- At least some rules must have ONLY terminals on the right side

Example of Grammar

- S ::= (S)
- S ::= <S>
- S ::= (empty)
- This is the language D2, the language of two kinds of balanced parens
  - E.g. ((<<>>))
- Well not quite D2, since that should allow things like (())<>

Example, continued

- So add the rule
  - S ::= SS
- And that is indeed D2
- But this is ambiguous
  - ()<>() can be parsed two ways
  - ()<> is an S and () is an S
  - () is an S and <>() is an S
- Nothing wrong with ambiguous grammars

BNF (Backus Naur/Normal Form)

- Properly attributed to Sanskrit scholars
- An extension of CFG with
  - Optional constructs in []
  - Sequences {} = 0 or more
  - Alternation |
- All these are just short hands

BNF Shorthands

- IF ::= if EXPR then STM [else STM] fi
  - IF ::= if EXPR then STM fi
  - IF ::= if EXPR then STM else STM fi
- STM ::= IF | WHILE
  - STM ::= IF
  - STM ::= WHILE
- STMSEQ ::= STM {;STM}
  - STMSEQ ::= STM
  - STMSEQ ::= STM ; STMSEQ

Programming Language Syntax

- Expressed as a CFG where the grammar is closely related to the semantics
- For example
  - EXPR ::= PRIMARY {OP | PRIMARY}
  - OP ::= + | *
- Not good, better is
  - EXPR ::= TERM | EXPR + TERM
  - TERM ::= PRIMARY | TERM * PRIMARY
- This implies associativity and precedence

PL Syntax Continued

- No point in using BNF for tokens, since no semantics involved
  - ID ::= LETTER | LETTER ID
- Is actively confusing since the BC of ABC is not an identifier, and anyway there is no tree structure here
- Better to regard ID as a terminal symbol. In other words grammar is a grammar of tokens, not characters

Grammars and Trees

- A Grammar with a starting symbol naturally indicates a tree representation of the program
- Non terminal on left is root of tree node
- Right hand side are descendents
- Leaves read left to right are the terminals that give the tokens of the program

The Parsing Problem

- Given a grammar of tokens
- And a sequence of tokens
- Construct the corresponding parse tree
- Giving good error messages

General Parsing

- Not known to be easier than matrix multiplication
  - Cubic, or more properly n**2.71.. (whatever that unlikely constant is)
  - In practice almost always linear
  - In any case not a significant amount of time
  - Hardest part by far is to give good messages

Two Basic Approaches

- Table driven parsers
  - Given a grammar, run a program that generates a set of tables for an automaton
  - Use the standard automaton with these tables to generate the trees.
  - Grammar must be in appropriate form (not always so easy)
  - Error detection is tricky to automate

The Other Approach

- Hand Parser
  - Write a program that calls the scanner and assembles the tree
  - Most natural way of doing this is called recursive descent.
  - Which is a fancy way of saying scan out what you are looking for ☺

Recursive Descent in Action

- Each rule generates a procedure to scan out the procedure.
  - This procedure simply scans out its right hand side in sequence
- For example
  - IF ::= if EXPR then STM fi;
  - Scan “if”, call EXPR, scan “then”, call STM, scan “fi” done.

Recursive Descent in Action

- For an alternation we have to figure out which way to go (how to do that, more later, could backtrack, but that’s exponential)
- For optional stuff, figure out if item is present and scan if it is
- For a {repeated} construct program a loop which scans as long as item is present

Left Recursion ☹

- Left recursion is a problem
  - STMSEQ ::= STMSEQ STM | STM
- If you go down the left path, you are quickly stuck in an infinite recursive loop, so that will not do.
- Change to a loop
  - STMSEQ ::= STM {STM}

Ambiguous Alternation ☹

- If two alternatives
  - A ::= B | C
- Then which way to go
  - If set of initial tokens possible for B (called First(B)) is different from set of initial tokens of C, then we can tell
  - For example
    - STM ::= IFSTM | WHILESTM
    - If next token "if" then IFSTM, else if next token is "while then WHILESTM

Really Ambiguous Cases ☹

- Suppose FIRST sets are not disjoint
  - IFSTM ::= IF_SIMPLE | IF_ELSE
  - IF_SIMPLE ::= if EXPR then STM fi
  - IF_ELSE ::= if EXPR then STM else STM fi
- Factor left side
  - IFSTM ::= IFCOMMON IFTAIL
  - IFCOMMON ::= if EXPR then STM
  - IFTAIL ::= fi | else STM fi
- Last alternation is now distinguished

Recursive Descent, Errors

- If you don't find what you are looking for, you know exactly what you are looking for so you can usually give a useful message
- IFSTM ::= if EXPR then STM fi;
  - Parse if a > b then b := g ;
  - Missing FI!

Recursive Descent, Last Word

- Don’t need much formalism here
- You know what you are looking for
- So scan it in sequence
- Called recursive just because rules can be recursive, so naturally maps to recursive language
- Really not hard at all, and not something that requires a lot of special knowledge

Table Driven Techniques

- There are parser generators that can be used as black boxes, e.g. bison
- But you really need to know how they work
- And that we will look at next time