Context Free Grammar and Parsing

CFG and PDA

- Language and grammar are the same things if semantics are disregarded
- Programming languages are more than context-free
    - CFG is the one efficiently implemented with its algorithm
    - Remaining non-CFG parts of a language are done separately
        - Static semantics
- The CFG component is processed by *parser* (PDA=PushDown Automaton)
- Disregarding the non-CFG components, we can speak of any programming language as CFG and thus we can speak of parser as PDA

Grammar and CFG

- General grammar is quadruple (T,N,S,R)
    - T is finite set of terminal symbols (tokens in PL)
    - N is a finite set of nonterminal symbols used for specification only
    - S is a unique nonterminal
    - R is a set of productions $\{\alpha \rightarrow \beta\}$
      where both sides are arbitrary sequences of terminal and nonterminals
        - when $\alpha \in N$ then the grammar is context-free (and thus possibly regular)
        - when also β = TN | T (or can be wriiten as), grammar is indeed regular

Application of CFG for Sentence Generation

1. start with S
2. at any step, replace one nonterminal by its production's rhs
3. stop when no more nonterminals

Example 1. Assume the following CFG, generate the shortest program, then some more programs. Upper case letters and elements starting with upper case are nonterminals.

A	-> begin Vars Stats end
Vars	-> id Vars | ε
Stats	-> Stat | Stat Stats
Stat	-> id = #tk ;

Using Sentence Generator for Parsing

- The sequence generator can be used to answer the question whether a given program is in the CFG (whether it is free of syntax errors or not)
    - This approach is highly inefficient though
    - Guided generation, also called recognition instead of generation, is used in practice

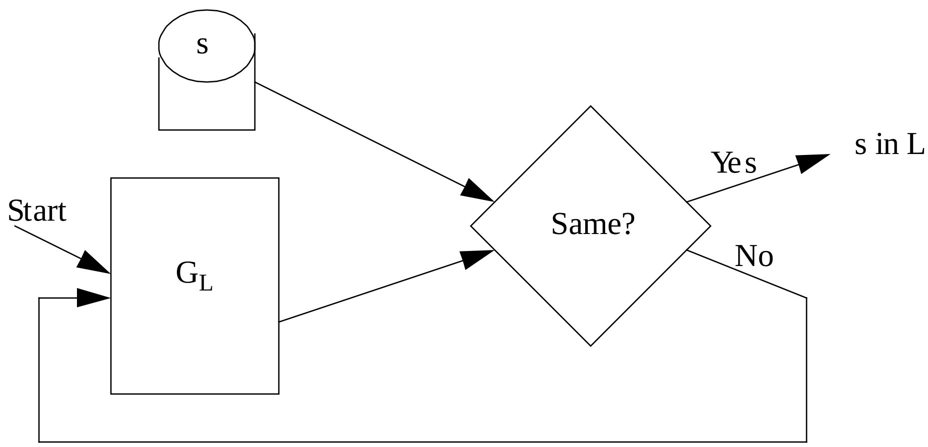

Figure 1. Parser based on CFG sentence generator.

- G_L is sentence generator for some CFG
- The above may answer the question whether user program s is free of errors
    - If G_L randomly generates sentences, the process may never end
    - If G_L is modified to start with shortest outputs (programs, sentences), the process will terminate if we compare generated program sizes against size of s
    - In practice, even this is not efficient and we use language recognizers RL
        - Generation is guided by source s to produce an equivalent program if possible

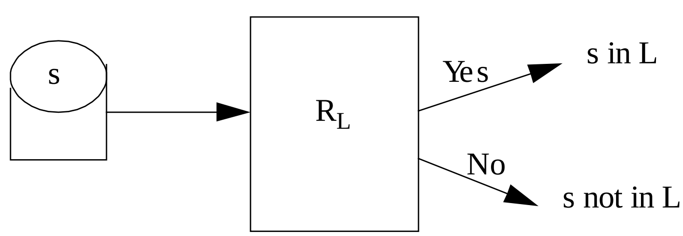

Figure 2.Parser based on language recognizer.

Some CFG Facts and Nondeterminism

- <> or upper cases for nonterminals
- Lower cases for terminals
- Avoid extended notation other than "|" or be careful not to mix with tokens when used
- PushDown Automaton PDA recognizes CFG and utilizes unlimited memory in form of a stack
    - Every CFG has a corresponding automaton PDA

- Unfortunately nondeterministic NDCFG are more general than deterministic DCFG
  - it is only computationally feasible to implement deterministic grammars
  - grammar must thus be deterministic to be practical for a programming language
- Determinism can be accomplished by lookahead on tokens
  - for efficient implementation, only one lookahead token is used
  - a programming language must have unambiguous grammar parsable with 1 lookahead
  - we will study methods for providing determinism with lookahead

Derivation

- Derivation is a sequence of top-down applications of the CFG productions
  - Starting with S and ending in T*
  - => denotes one step derivation
    - replace one of the current nonterminals with one of its rhs
  - =>* denotes a sequence of derivations
- Thus
  S =>* T*
  is a complete derivation because T* means there are no more nonterminals to work with.
  - T* means tokens only and thus T* means a program
    - A program is a sequence of tokens (after scanner)
    - The program can be valid (syntax valid) or invalid (syntax errors)
- During derivation, if more than 1 nonterminals is found, the process can
  - Replace random nonterminal
  - Always replace leftmost
    - *leftmost* derivation
    - use on *top-down* parsers
  - Always replace rightmost
    - *rightmost* derivation
    - used in *bottom-up* parsers
- Leftmost derivation

Example 2. Use the CFG of Example 1 to show sample short derivations: leftmost and rightmost.

- Derivation is helpful when the question is syntax error or not but not sufficient in compilation
  - In compilation, the structure of the input program must also be recovered to properly generate subsequent target
    - *Parse trees* are used for this propose

Parse Tree

- Parse tree (*syntax tree*) uses derivation while building and operating on a tree
  - Successful generation of a parse tree for some CFG upon exhausted input s
    - means s is a valid program according to the CFG
- Parse tree represents the meaning of the program in its structure and detail
  - left-to-right traversal represents the input program
  - parse trees can be *isomorphically* equivalent (abstract tree)
    - usually so called syntactic tokens are not entered into three: `{}, (), begin/end, while`, *etc*.
      They are needed in program source (linear) but not in the structured tree
    - *semantic* tokens are stored in the tree: identifiers, operators, types, *etc*.
  - tree can be eventually decorated with semantics information for static semantic processing
- Parse tree is the intermediate program representation in compilers

Example 3. Repeat Example 2 using a parse tree.

Ambiguity of Expressions

Example 4. First grammar: E-> E + E | E – E |E * E | E / E |(E) | id

Example 5. Using Example 4 grammar parse, producing parse tree, the input program: (x+y)/(x-y).
Draw parse tree.
Show isomorphic trees and minimal tree.

- Grammar is ambiguous if a sentence can be parsed with at least two different parse trees (not just isomorphic)

Example 6. Use CFG from Example 5 to parse program: x + y * z
Draw parse tree
Draw another structurally different parse tree
Show different results assuming x=1, y=2, z=3.
The problem is lack of precedence on operators.

- Grammar ambiguity on precedence can be fixed by arranging productions in an order
  - Weaker precedence goes higher in grammar
    - Higher in tree
  - Stronger precedence goes lower in grammar
    - Lower in tree and thus binding stronger

Example 7. CFG of Example 4 with precedence set so that + and – are weaker
E -> E + E | E – E | T
T -> T * T | T / T | F
F -> (E) | id

Example 8. Use CFG from Example 7 to parse x + y * z. Then try x – y + z.
Show two different trees on the latter.

- Grammar ambiguity on associativity can be fixed by properly using recursion
  - Left associative operators must have left recursion only
  - Right associative operators must have right recursion only

Example 9. CFG from Example 8 fixed to have left associativity on all operators except * which is right associative.
E -> E + T | E – T | T
T -> F * T | T / F | F
F -> (E) | id

CFG Rewriting Rules

- Grammar can be rewritten in equivalent alternative form
  - by substituting a nonterminal symbol with ALL its productions
  - by introducing a new nonterminal, and then an appropriate production, in place of any sequence of terminals and nonterminals
  - nonterminal names are irrelevant

Example 10. Show examples of grammar rewrite.