Programming Languages Recitation: Syntax – Regular Expressions and Context Free Grammars

Paul Fisher
New York University
pmf296@nyu.edu
Adapted from slides by Deepti Verma

Introduction

- Regular Expressions
- Context Free Grammar (CFG)
- Derivations and Parse trees

Phases of compiler

- Lexer
- Parser
- Semantic Analyzer
- Intermediate code generator
- Optimization(based on architecture of the system)
- Target code generation

Regular Expressions

- Tokens are the basic building blocks of a program. They are the shortest strings of characters with individual meaning.
- Examples include keywords, identifiers, symbols, constants and numbers.
- In order to specify tokens we use the notation of regular expressions
- Used in the Phase 1 -lexical analysis (Scanner) of the compiler.

Regular Expressions

- Given regular expressions R1 and R2 the following operations can be performed on them:
  - Concatenation: Two regular expressions next to each other. Eg. R1 R2
  - Alternation: Two regular expressions separated by a vertical bar, meaning any string generated by the first one or any string generated by second one. Eg. R1|R2 (OR operation)
  - Kleene Star (*)

Examples

- a - matches the character ‘a’
- ε - matches a null string
- a|b|c - matches ‘a’ or ‘b’ or ‘c’
- abc- matches ‘abc’ (a concatenated with b concatenated with c)
- [a-z] - matches any character between ‘a’ through ‘z’ (Shorthand)
- Alphabet (Uppercase and Lowercase) → [A-Za-z]
- digit → 0|1|2|3|4|5|6|7|8|9
- integer → digit digit*
- number → integer|real
- Identifier → Alphabet (digit|Alphabet)*

Regular Expressions

- Drawbacks:
  - Nesting cannot be expressed in regular expressions which is central to programming languages.
  - For example: Nested parenthesis, palindromes

Context Free Grammar

- More powerful than regular languages/expressions.
- By adding RECURSION, we can define many more sets of strings
- Recognized by parsers.
- Every regular grammar is context free but not every context free grammar is regular.
- Used in the Phase 2 - Syntactic analysis (Parser) of the compiler.

Context Free Grammar

- Consists of
  - Productions (Substitution Rules) : Rules in a CFG of the form

    $A \longrightarrow B$
  - Nonterminals: Symbols on the left side of the production (A).
  - Terminals: Symbols that make up the strings derived from grammar. They cannot appear on the left hand side of any production. They represent language's tokens. In the production shown above B is a string of terminals and nonterminals.
  - Start symbol: The nonterminal on the left side of the first production.
- The notation of CFG is sometimes called Backus-Naur form

Example 1:

- $A \longrightarrow 0A1 | \varepsilon$
  - Language of strings consisting of a number of 0's followed by an equal number of 1's.
  - Why can't this be represented by a regular expression?

Example 2:

- Give CFG for strings containing only 0's and 1's and contain equal number of 0's and 1's but in any order. Eq: 01, 0011, 0110,1010,1100, 11100010..
- Solution: $S \longrightarrow 0S1S | 1S0S | \varepsilon$

Example 3:

- Give a context free grammar for all the strings ending with character 'a' and containing the set of characters {a,b}. For example: aa, ba, aaba, abbaa..
- Solution:
  - S $\longrightarrow$ aS|bS|a

Example 4:

- Give context free grammar for all the strings containing the set of characters {a,b} and has even number of a's. For example: aa, aba, abba, abab, ababaa, bb
- Solution:
  - $S \longrightarrow aSaS \mid bS \mid \varepsilon$

Example 5:

- Give context free grammar for the following code:

  read X

  read Y

  prod=X*Y

  write prod

- Solution:

  Program → Stmt_List

  Stmt_List → Stmt Stmt_List |ε

  Stmt → id=Expr|read id| write id

  Expr → Expr OP Expr|-Expr|(Expr)|id|num

  OP → +|-|*|/

Conditional Statements

Let's expand our grammar to include statements like the following:

```
if a == b then
    statement1
    statement2
else
    statement3
    statement4
end
```

Example 5 ext.

Stmt → id=Expr|read id| write id | Cond

Cond → if BoolExpr then Stmt_List else Stmt_List end

BoolExpr → Expr relOp Expr | True | False

relOp → == | > | < | >= | <= | !=

Parse Tree

- A CFG shows how to generate syntactically valid string of terminals.
  - Begin with start symbol.
  - Choose a production with the symbol on the left hand side; replace the nonterminal with the right hand side of that production. Start with the production with start symbol on the left.
  - Repeat this process until no nonterminals remain.
- Derivation: A series of replacement operations that shows how to derive a string of terminals from the start symbol.
- Derivation can be represented graphically as a *parse tree.*
- The root of the parse tree is the start symbol of the grammar.
- Based on CFG the parser in compiler produces a parse tree out of the stream of tokens.

Example 1:

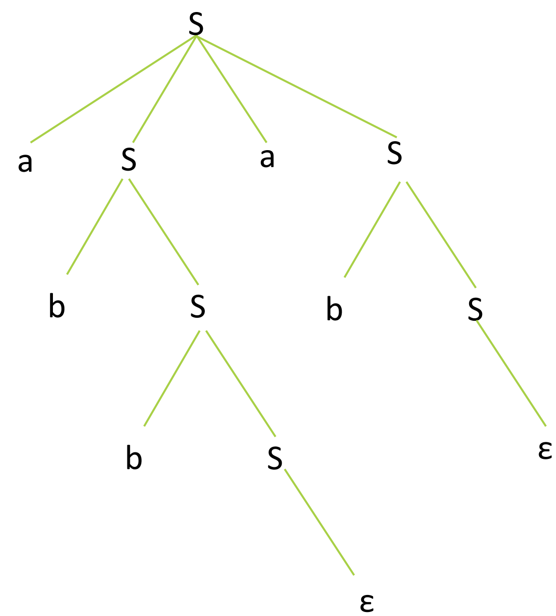
Given CFG for all the strings containing the set of characters {a,b} and having an even number of a's.

- $S \rightarrow aSaS \mid bS \mid \varepsilon$

Derivation:

$S \rightarrow aSaS \rightarrow abSaS \rightarrow abbSaS$
$\rightarrow abbaS \rightarrow abbabS \rightarrow abbab$

Parse Tree for string "abbab":



Example 2:

Assume that Expression E has CFG:

E → E+E|E*E|(E)|num|id

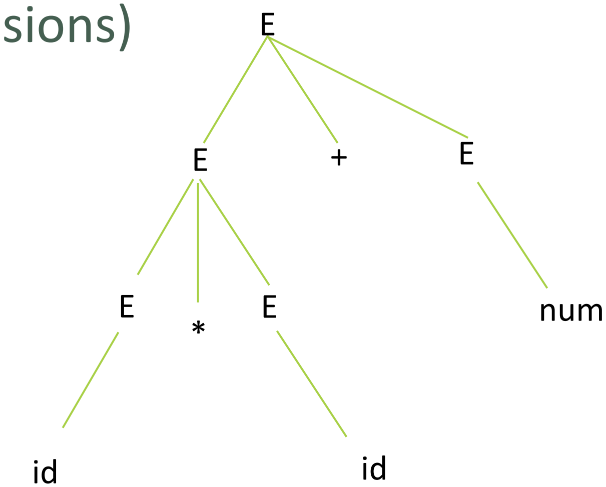
(num and id are defined by regular expressions)

Derivation:

E → E+E → E+num → E*E+num

→ id*E+num → id*id+num

One Parse Tree for string “id*id+num”:



Unambiguous Grammar for Arithmetic Expressions

A CFG is *ambiguous* if there are strings which that grammar can produce with more that one distinct derivation (more than one parse tree.)

E → Term | E+Term
Term → Factor | Term*Factor
Factor → (E) | num | id

Draw a Parse tree for the expression: 3 + (x + 2 * y)

Parse Tree for 3 + (x + 2 * y)