Recitation - 01

Grammars, Parsers, Flex and Bison

Jahnavi - jp5867@nyu.edu

Phases of Compiler:

- Lexer – lexical analyzer – group symbols into tokens
- Parser - generates parse tree
- Semantic Analyzer - verifies whether the parse tree is correct or not
- Intermediate code generator
- Optimization
- Target code generation

Syntax - rules governing the organisation of symbols in a valid program

Semantics - meaning of the language

How do you define syntax?

1. Regular expressions - To specify the tokens during lexical analysis
2. Grammars - used during parsing

Regular Expressions:

- Tokens are the basic building blocks of a program.

- Examples include keywords, identifiers, symbols, constants and numbers.

- In order to specify tokens we use the notation of regular expressions

- Regular expressions can be formed by concatenating (.), alternating (|) two regular expressions or Kleene Star (*)

Example: Consider there are only three letters { a, b, c } in the grammar

-> The language that contains the string "aaa" at some point - (a | b | c)* aaa (a | b | c)*

-> The language that does not contain the string ca - (a | b | cc* b)* c*

Drawbacks : Nesting cannot be expressed in regular expressions

Context Free Grammar:

- It is a formal grammar which is used to generate all possible strings in a given formal language.

  G= (V, T, P, S)

  G describes the grammar

  T describes a finite set of terminal symbols.

  V describes a finite set of non-terminal symbols

  P describes a set of production rules

  S is the start symbol.

Example: Consider there are only three digits { 0,1 } in the grammar

CFG for equal number of 0's and 1's : S -> 0 S 1 S| 1 S 0 S | ε

CFG for 0^n1^n : S -> 0 S 1|ε

Note: Regular Grammar ⊂ CFG ⊂ CSG

Derivation:

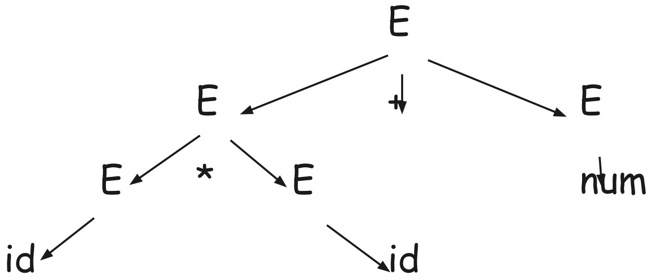
- Begin with the start non-terminal and repeatedly apply the production rules until we get the desired string

Parse tree: Graphical representation of the derivation

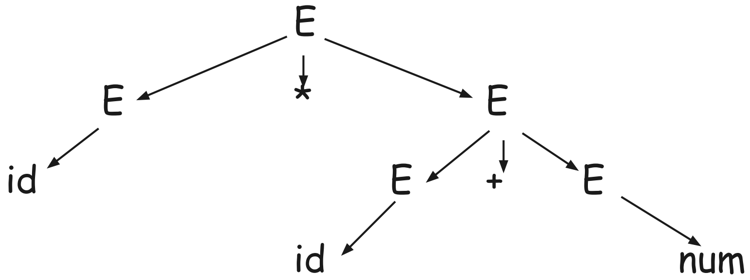
Consider a grammar :

E -> E+E | E*E | (E) | num | id

Parse Tree 1: x * x + 5



Parse Tree 2: x * x + 5



Since there are two parse trees, the grammar is ambiguous

How to remove ambiguity:

- If the ambiguity is due to the associativity, apply associative rules

  Example: E -> E + E / id

  The string int + int + int - has two parse trees. Hence to avoid this, we can add left associativity rules for the grammar

  E -> E + id / id

  In a similar fashion, ^ follows right associativity

- If the ambiguity is due to precedence - where some symbols have precedence over others

  Example: E -> E + E / E * E / id

  The string int + int * int has two parse trees. In this case, we have to introduce a new non-terminal. It's important to remember that the tree is evaluated bottom-up, so the priority in the grammar is inverted; in the first rule, there is the lowest priority.

  E -> E + T | T, T -> T * id / id

- **Dangling else problem:**

In a series of if terminating with an else, to which if the else is referring. The ambiguous grammar is:

stmt → if expr then stmt | if expr then stmt else stmt | other

How can we solve this ? - **Match each else with the closest unmatched then**

The list all the allowed productions:

-> if expr then matched_stmt

->if expr then open_stmt

->if expr then matched_stmt else matched_stmt

->if expr then matched_stmt else open_stmt

The productions not allowed are:

-> if expr then open_stmt elsematched_stmt

-> if expr then open_stmt else open_stmt

Updated rules of grammar:

stmt → open_stmt | matched_stmt

matched_stmt → if expr then matched_stmt else matched_stmt | other

open_stmt → if expr then stmt | if expr then matched_stmt else open_stmt

Parsers:

LL Parsers: Left-to-right Leftmost derivation

-> Leftmost derivation: always expand the left non-terminal first!

-> Uses top down recursive descent algorithm

-> LL(k) - uses lookahead upto k tokens

Problems:

1. Cannot parse grammars with left recursion
2. Rewrite the CFG so that it does not have common prefixes

   a. First/First conflict :

      Choices starting with the same k tokens => A ->ab | ac (k = 1)

   b. First/Follow conflict :

      Choice can start or (if it is nullable) be followed by the same k tokens

      S -> X Y , X -> ε | a , Y -> a | b

Parsers:

LR Parsers: Left-to-right Rightmost derivation

-> Leftmost derivation: always expand the left non-terminal first!

-> Uses bottom up algorithm - reconstructs a reverse rightmost derivation

-> Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree

Choices we have:

1. Perform a reduction 2. Look ahead further

So, this is called shift reduce parser.

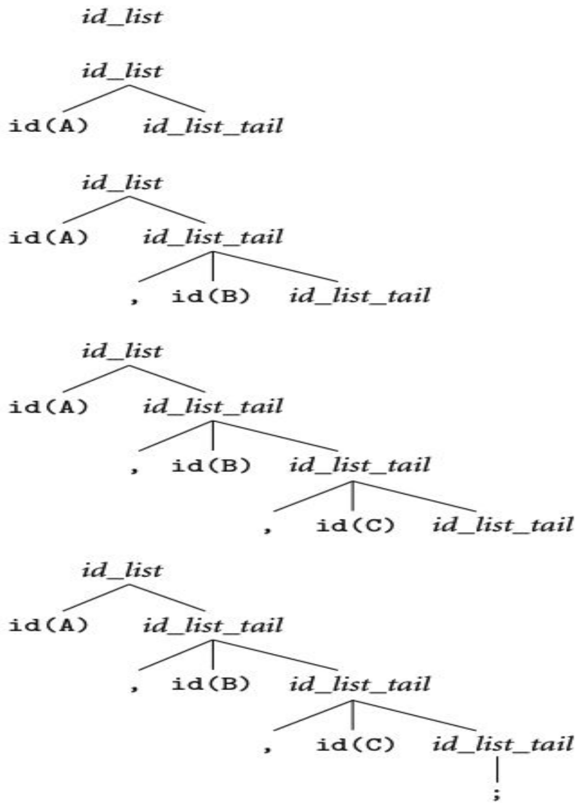
Problems:

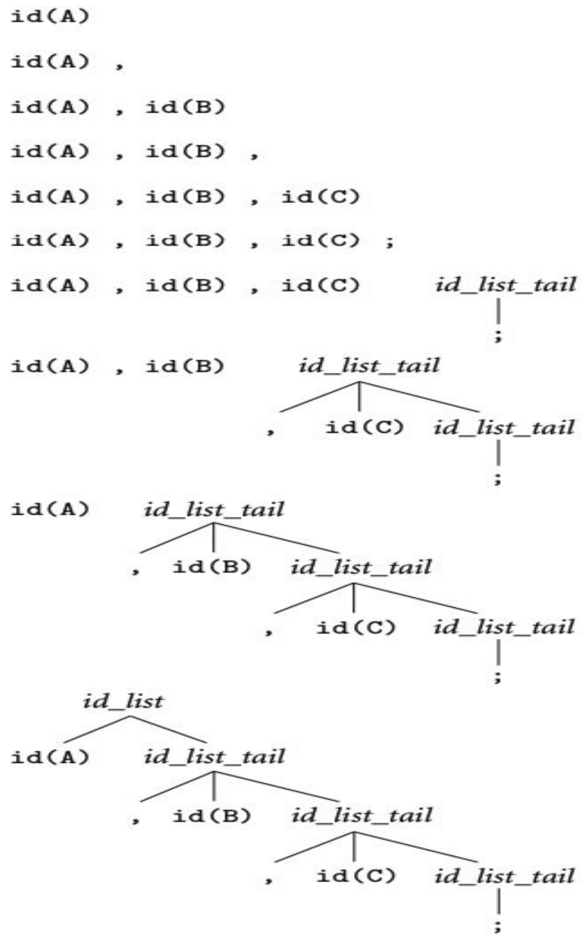
1. Shift-Reduce conflicts - both a shift and a reduce are possible at a given point in the parse

   Example: S -> if then S| if then S else S - Either fix the grammar or use longest matching rule

2. Reduce-Reduce conflicts - two different reductions are possible in a given state

   S -> A | B, A -> x, B -> x - Try to use lookahead information or fix the grammar





id_list ⟶ `id` *id_list_tail*
id_list_tail ⟶ `, id` *id_list_tail*
id_list_tail ⟶ `;`

Flex and Bison:

Flex - Scanning divides the input into meaningful chunks, called *tokens*

Bison - parsing figures out how the tokens relate to each other.

Flex and Bison programs :

DEFINITIONS

%%

RULES

%%

HELPER FUNCTIONS

yylex - whenever the program needs a token

yywrap - returns 1 if the input is exhausted, 0 - otherwise

yyparse - 0 - if the parsed input is correct and according to grammar provided, else 0

Convert EBNF to BNF:

- Convert every repetition { E } to a fresh non-terminal X and add
  X = ε | X E.
- Convert every option [E] to a fresh non-terminal X and add
  X = ε | E.
  (We can convert X = A [E] B. to X = A E B | A B.)
- Convert every group (E) to a fresh non-terminal X and add
  X = E.
- We can even do away with alternatives by having several productions with the same non-terminal.
  X = E | E'. becomes X = E. X = E'.

Thank you