Exercises on Grammars

1. Consider the following grammar:

S -> (L) | a

L -> L , S | S

- Is this grammar ambiguous ?
- Is this grammar LL(1) ?
- Compute the First and Follow sets for the new grammar.
- Construct the parsing table for the LL(1) parser

Finding an LL(1) grammar

- No procedural way ! Practice …
- But there are some recommended practices that generally help in finding one.
- Eg. try to eliminate left recursion.
  - There is a procedure for this but you don't have to faithfully follow the entire approach.
  - Just think of what left recursion brings and what can be done to eliminate them

Removing Left Recursion

S -> (L) | a

L -> L , S | S

- How does a derivation starting from 'L' look ?
- L => L , S

  => L , S , S

  =>* L , S , … , S

  => S , … , S
- L -> L , S | S is equivalent to L -> S , L | S

  S -> (L) | a

  L -> S , L | S

Removing Left Recursion

- In general, L -> L α | β_1 | … | β_n
- L -> β_1 Z | … | β_n Z | β_1 | … | β_n
- Z -> α Z | ϵ
- This will remove immediate recursion but only when there are no epsilon productions in the grammar
- Otherwise, we need to remove epsilon productions which will discussed along with CYK parsing
- Removing indirect recursion

  S -> L a

  L -> S a | b

Removing Left Recursion

- Order nonterminals Eg. (1) S , (2) L
- Enforce that if A -> B then A should precede B in the ordering
- S -> L a and L -> b satisfy the constraint but L -> S a doesn't
- Inline the production of S in L -> S a
- We get, L -> L a a | b , Remove left recursion.
  - Result: L -> b Z | b Z -> a a Z | ϵ
- If inlining does not result in left recursive production or doesn't satisfy the constraints, inline again.

Example 1 [Cont.]

S -> (L) | a

L -> L , S | S

- After eliminating left recursion

S -> (L) | a

L -> S , L | S

- Is this LL(1) now ?

Example 1 [Cont.]

S -> (L) | a

L -> L , S | S

- After eliminating left recursion

S -> (L) | a

L -> S , L | S

- Is this LL(1) now ?

Left factorization

S -> (L) | a

L -> S , L | S

- Identify a common prefix and push the suffixes to a new nonterminal.

S -> (L) | a

L -> S Z

Z -> , L | ϵ

- Is this LL(1) now ? Yes

Exercise 2

Consider a grammar for expressions where the multiplication sign is optional.

ex ::= ex + ex | ex * ex | ex ex |ID

- Find a LL(1) grammar recognizing the same language
- Create the LL(1) parsing table.

Exercise 2 – Solution

- First let's make the grammar unambiguous by associating precedence with operators
- In the process we also made sure that the grammar does not have left recursion
- ex ::= S + ex | S
- S ::= ID * S | ID S | ID
- Left factorization:
- ex ::= S Z
- Z ::= + ex | ϵ
- S ::= ID Z2
- Z2 ::= * S | S | ϵ

Exercise 3

Balanced Parentheses over { (, [}

S ::= (S)| [S] | S S | ϵ

- Find a LL(1) grammar recognizing the language

Exercise 3 - Solution

- S ::= (S)| [S] | S S | ϵ
- 'S' produces epsilon. Hence, we need to first eliminate epsilon (discussed later) and then remove left recursion from S ::= S S
- Instead, let's apply the same logic as removing left recursion but without performing all the steps.
- The role of the production S ::= S S is to produce a sequence of S that begin with either (S) or [S]. i.e,
  - (S) S S …. S
  - [S] S S……. S

Exercise 3 - Solution

- Each of the successive S 'es can rewrite to either (S) or [S]. That is, in essence S ::= S S produces sequences given by the regular expression ((S) | [S]) *
  - E.g (S) (S) [S] (S) … is one such sequence
- The same effect can be achieved by the right recursive rules
  - S ::= (S) S | [S] S | ϵ
- The above grammar is LL(1)

Exercise 4

Prove that every LL(1) grammar is unambiguous.

Solution to Exercise 4

The answer is pretty simple. Every production of an LL(1) generates a set of strings that is completely disjoint from the other production simply because they start with different terminals

Formally, for any string w there is a unique left most derivation. We can prove this by induction over the length of w.

Solution to Exercise 4 [Cont.]

- Claim : for all k, |w| <= k => there is a unique left most derivation for 'w' from every nonterminal N such that N =>* w
- Base case, w = ϵ: claim holds as only one alternative can derive epsilon for every nonterminal in an LL(1) grammar
- Inductive case, |w| = k: Let w = ax where |x| < k and 'a' is a terminal of the grammar.
- For every nonterminal N, s.t N =>* w, the first rule that N uses is the alternative of N whose First set has 'a'
- Hence, N => $a\ \alpha$, $\alpha \Rightarrow^* x$
- From hypothesis it is easy to show that there is unique derivation for $\alpha \Rightarrow^* x$
- Hence, the claim holds

Exercise 5

Say that a grammar has a cycle if there is a *reachable, productive* non-terminal A such that $A \Rightarrow^* A$, i.e. it is possible to derive the nonterminal A from A by a nonempty sequence of production rules.

Show that if a grammar has a cycle, then it is not LL(1).

(the solution is provided in a separate pdf file in the lara wiki)

Exercise 6

Show that the regular languages can be recognized with LL(1) parsers. Describe a process that, given a regular expression, constructs an LL(1) parser for it.

Solution for Exercise 6

- Let the DFA for the regular language be A : $(\Sigma, Q, q_0, \delta, F)$
- Define a grammar G: (N, T, P, S) where,
- N = $\{ S_i \mid 1 \leq i \leq |Q| \}$
- $\mathrm{T} = \Sigma$
- S = S_0
- $\delta(q_i, a) = q_j \Rightarrow S_i \rightarrow a\, S_j \in P$
- $q_i \in F \Rightarrow S_i \rightarrow \epsilon \in P$

L(A) = L(G)

Exercise 7

Show that the language defined by the grammar

S -> a S | P

P -> a P b | a

cannot have a LL(1) grammar ?

Exercise 7 Solution

Intuitively, you cannot determine just by looking at the current input character whether it will have a matching 'b' or it is an excess 'a' that will have not matching 'b'.

Formally, we can prove this by showing that "For any grammar for the language, there exists a nonterminal which will have two productions whose first sets will intersect"

(the solution is provided in a separate pdf file in the lara wiki)