PLT Practical 4 Handin

J L Gouws
19G4436

March 21, 2022

2. Grammar 1:
   - No – This grammar does not generate palindromes. The grammar cannot derive a string made of terminal characters. Use of the productions results in an infinite recursion.
   - No – The grammar is not LL(1) – it is not even a legitimate grammar

   Grammar 2:
   - Yes – This grammar looks like it will work.
   - No – This grammar, however, is not LL(1).
     FIRST("a" Palindrome "a:) $\bigcap$ FIRST("a"): = {"a"} $\neq \emptyset$. This disobeys rule 1 of an LL(1) grammar.

   Grammar 3:
   - No – This will is not able to describe the palindrome "a" or "b". It only checks palindromes with even length.
   - No – This grammar is not LL(1). Look at the construct [Palindrome], it is nullable.
     FIRST([Palindrome]) = FIRST(Palindrome) = {"a", "b"}
     And:
     FOLLOW([Palindrome]) = FOLLOW(Palindrome) = {"a", "b"}
     The intersection of these two sets is not empty, thus the grammar is not LL(1), it breaks "rule 2".

   Grammar 4:
   - Yes – This looks like it will work, but it does allow for the empty string to be parsed. This is different in behaviour to 2). Arguably the empty string is a palindrome, so it still describes palindromes. The decision really comes down to the language designer.
   - No – This grammar is not LL(1)
     FIRST("a" Palindrome "a:) $\bigcap$ FIRST("a"): = {"a"} $\neq \emptyset$. This disobeys rule 1 of an LL(1) grammar.

   I do not think that it is possible to find an LL(1) grammar to generate palindromes. Take, for example, the palindrome "aaa"–it cannot be parsed by an LL(1) parser. The parser can only see one "a" at a time, and so will not be able to find the midpoint of the string from which it can match the other "a"s.

3. a) This is true. If the grammar is LL(1), we can determine the exact parse tree by always looking at the next terminal. Since our expansion of non-terminals is always determined by the current terminal, there is only one option to parse, and so there can be no other parse tree/ambiguity.

   b) This is not true. A grammar might be LL(2) or LL(3) and so on, and if it is LL(x), x> 1, then the grammar will be unambiguous.

   c) This is true, if the language is ambiguous then it might not have a unique left parse tree for all sentences. An LL(1) grammar requires that at every step, it can determine which left most derivative to expand. This being said an LL(1) grammar might be able to generate the same language as the ambiguous grammar, but it will not be able to associate multiple meanings with a given sentence, that an ambiguous grammar can do. It is impossible for an LL(1) grammar to describe all the semantics of an ambiguous grammar.

   d) Take the palindrome grammar:
   Palindrome = "a" Palindrome "a" | "b" Palindrome "b" | "a" | "b" .
   There is nothing ambiguous about this grammar. This grammar cannot, however, be replaced by an LL(1) grammar.

4. This is not an LL(1) grammar.
   Take the production "StartI" for example:

   StartI = "I" [["I"] ["I"] | "V" | "X"] .

   This can be written in a clearer form:

   StartI = "I" Rest
   Rest = "I" | "I" "I" |"V" | "X" .

   We can see: FIRST($Rest_{"I"}$) = {"I"} and FIRST($Rest_{"I" "I"}$) = {"I"}
   These sets are not disjoint, and so the grammar is not LL(1), it breaks rule 1.
   "StartI" also breaks rule 2, well actually its first subproduction: ["I"] , breaks rule 2–its follow set contains "I". StartV also breaks rule 1 of checking if the grammar is LL(1).

   The grammar is also ambiguous. Take the string: "V I ", for example. This is a perfectly valid string of the grammar.
   The string can be derived from the grammar in the following ways:

   - V I ε ε
   - V ε Iε
   - V$\varepsilon\varepsilon$ I

   These have different parse trees, and so have potentially different meanings. The grammar is thus ambiguous.