## Programming Language Translation

## Practical 4 Handin: G16W4712

### Task 2 Palindromes [4 marks] – for tutor marking

Does grammar 1 describe palindromes? If not, why not?

No, this grammar would never terminate and thus cannot describe anything.

Is it an LL(1) grammar? If not, why not?

Yes, it seems to follow rule 1 as the terminals “a” and “b” are mutually exclusive to each other.

Does grammar 2 describe palindromes? If not, why not?

No, not perfectly, this grammar only allows for palindromes of odd length.

Is it an LL(1) grammar? If not, why not?

No, this grammar breaks rule 1 as the terminals “a” and “b” are not mutually exclusive, they allow for more than 1 unique starting point.

Does grammar 3 describe palindromes? If not, why not?

No, not perfectly, this grammar only allows for palindromes of even length.

Is it an LL(1) grammar? If not, why not?

No, this grammar has a nullable production where ‘[Palindrome]’ is used.

Does grammar 4 describe palindromes? If not, why not?

Yes.

Is it an LL(1) grammar? If not, why not?

No, the entire production is nullable and the terminals “a” and “b” are not mutually exclusive, they allow for more than 1 unique starting point.

Can you find a better grammar to describe palindromes? If so, give it, if not, explain why not.

I could not find an LL(1) grammar that better describes this non-LL(1) compliant grammar. An LL(1) grammar is not best suited to tackling this issue, by looking at 1 token an LL(1) parser should always know the next step, however, with palindromes of uncertain length a type of recursion is needed which won’t specifically know exactly what to do next by looking at only 1 token. Keeping this in mind an LL(1) parser would be able to parse a palindrome of any length as long as it could determine where the halfway point is, for this to happen it would probably need to be denoted by a unique character. This technique would unfortunately put an unnecessary restriction on what types of palindromes would be valid and thus an LL(1) grammar is not suited to describing palindromes.

### Task 3 Thinking about ambiguity [4 marks]

Which of the following statements are true? Justify your answers.

(a) An LL(1) grammar cannot be ambiguous.

True, for a grammar to be LL(1) compliant the parser when given a valid input must always be able to find 1 parse tree which is distinct and valid and when given an invalid input no parse tree should be yielded. This level of determinism is the definition of unambiguity and thus an LL(1) grammar cannot be ambiguous as it will only be able to yield a valid or invalid result and no other alternatives.

(b) A non-LL(1) grammar must be ambiguous.

False, a grammar that is non-LL(1) must only break one of the rules of an LL(1) grammar

(c) An ambiguous language cannot be described by an LL(1) grammar.

True, LL(1) grammars only describe deterministic parsing.

(d) It is possible to find an LL(1) grammar to describe any non-ambiguous language.

False, just because a grammar is not ambiguous does not mean that it is LL(1) compliant. In order to be LL(1) compliant a grammar must also not have any nullable productions, if a grammar breaks this rule (Rule 2) it can be non-ambiguous and also non-LL(1) compliant.

### Task 4 RPN [6 marks] - for tutor marking

Are the given grammars equivalent?

Yes.

Is either (or both) ambiguous?

They are both ambiguous.

Do either or both conform to the LL(1) conditions? If not, explain clearly where the rules are broken, and come up with an LL(1) grammar that describes RPN notation, or else explain why it might be necessary to modify the language itself to overcome any problems you have uncovered.

Neither conforms to LL(1), G1 violates Rule 1, it has ambiguous grammar with the shared terminal “-”, First sets are not disjoint. G2 also violates this same rule in the same way, additionally it violates Rule 2 as the REST production is nullable.

I could not think of an LL(1) grammar to describe RPN, however if modifications were to take place to the notation to disambiguate it such as using the keyword neg to symbolise the unaryOp for “-“ and then apply it similarly to how “sqrt” is implemented, this modification could make the grammar LL(1) compliant.