## Programming Language Translation

## Practical 4 Handin – Group J

### Task 2 Palindromes

**Grammar 1:**

**Does it describe palindromes? If not, why not?**

Grammar 1 does not describe palindromes since it is non-terminating. Every production it creates has at least 1 non-terminal and so creates a never ending cycle continuously calling itself again and again.

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

Yes. At any point in the grammar the parser must be able to decide on the basis of a single look-ahead symbol which of several alternatives have to be selected. This grammar is therefore LL(1) since either of the options it can follow begins with distinct tokens (“a” or “b”).

**Grammar 2:**

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

No, since the single “a” or “b” options in the production, while allowing termination of the grammar, inhibits the ability to produce ALL palindromes. This grammar accepts palindromes of an odd length (such as ababa), but would fail a palindrome of an even length. This is due to the only terminating replacement for the ‘Palindrome’ production being a single ‘a’ or ‘b’, so while it accounts for checking a palindrome from the outside inwards, the centre HAS to end with a single ‘a’ or ‘b’, which only accounts for palindromes of even length.

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

No since the parser will not be able to decide which alternative to take. This can be seen by the fact that the token ‘a’ is the start of at least 2 different productions, the same with the token ‘b’.

**Grammar 3:**

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

No, though the use of the square brackets now allows for only even length palindromes, it doesn’t account for odd length palindromes that need to terminate with a single character at the centre.

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

No since there are LL(1) conflicts caused by “a” and “b” being at the start and successor of a deletable structure.

**Grammar 4:**

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

Yes.

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

No since the parser will not be able to decide which alternative to take. This can be seen by the fact that the token ‘a’ is the start of at least 2 different productions, the same with the token ‘b’.

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

Yes. The grammar below avoids any LL(1) errors.

Palindrome = [ “a” [Palindrome “a”] | “b” [Palindrome “b”] ] .

### Task 3 Thinking about ambiguity

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

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

True. At any point in the grammar the parser must be able to decide on the basis of a single look-ahead symbol which of several alternatives have to be selected. An ambiguous grammar can produce different parse trees for the same sentence and so would therefore have multiple alternatives for the parser to select. Thus an LL(1) grammar cannot be ambiguous.

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

False. A non-LL(1) grammar refers to an LL(k) grammar, meaning it looks ahead k symbols in the production to decide which alternative to take. While this then creates an environment in which ambiguous grammars could be created, it does not necessarily mean that all LL(k) grammars MUST be ambiguous. LL(k) grammars are able to describe more grammars than LL(1) grammars, including LL(1) grammars. Since LL(1) grammars cannot be ambiguous, therefore there is an LL(k) grammar that is not ambiguous.

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

True, not without the help of resolvers. This allows the parser to determine which alternative to take in instances of LL(1) conflicts.??????????????????????

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

False, although an LL(1) grammar cannot be ambiguous, this does not imply that the inverse is true. A language could be non-ambiguous, however it takes more than a single look-ahead to determine that and so will require a non-LL(1) grammar. Therefore certain non-ambiguous languages cannot be described by an LL(1) grammar.

### Task 4 Meet the family

**Is it LL(1) compliant? If not, which productions break the LL(1) rules, and why?**

No. Parents is the only production that breaks the LL(1) rules since ‘Mom’ is at the start of several alternatives. At any point in the grammar the parser must be able to decide, on the basis of a single look-ahead symbol, which of several alternatives have to be selected. This results in LL(1) conflicts as the parser is unable to determine which alternative to take with only a single look-ahead symbol.

**Can you find an equivalent grammar that does obey the LL(1) constraints? If so, give it. If not, explain why you think it cannot be done.**

Yes:

PRODUCTIONS

Home = Family Pets Vehicle "house" .

Vehicle = BikeOrScoot "fourbyfour" | .

Pets = "dog" Pets | "cat" Pets | .

BikeOrScoot = "scooter" | "bicycle" .

Family = Parents Children .

Children = Child | Children Child | .

Parents = Mother Father .

Mother = “Mom” | .

Father = “Dad” | .

Child = "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"

| "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .

### Task 5 RPN

**Are the given grammars equivalent?**

Yes, the languages are equivalent because they generate the same set of sentences.

**Is either (or both) ambiguous?**

RPN2 is ambiguous since two different parse trees can be derived for the same sentence. This is due to the recursive call for ‘REST’ appearing in 2 places in a single production result. RPN1 is not ambiguous since only one parse tree can be derived for the same sentences.

**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.**

Both fail to conform to LL(1) conditions. The first production in RPN1 has three alternatives that can all start with number. In RPN2 the “-” symbol is the start and successor of a deletable structure. This is because RPN2 can end in a “-” and unaryOP can begin with “-”. The parser therefore doesn’t know when the RPN2 derivation ends and where the unaryOP begins.

RPN = [number number RPN binOp | number RPN unaryOp] .

binOp = “+” | “-” | “\*” | “/” .

unaryOp = “-” | “sqrt” .