Started on	Thursday, 10 August 2023, 14:05
State	Finished
Completed on	Thursday, 10 August 2023, 14:35
Time taken	30 mins
Grade	**12.5** out of 20.0 (**62.5**%)

Quiz navigation

1 2 3 4 5 6

Show one page at a time

Finish review

Question 1

Incorrect

Mark 0.0 out of 2.0

Flag question

Part of the grammar for the extended Parva language you met in Prac 4 might have been expressed as follows:

```
Statement      =  (  Block | WhileStatement | ReadStatement | Assignment  |  VarDeclarations ).
Block          =  "{"   { Statement }    "}" .
WhileStatement =  "while"   "("  Condition  ")"   Statement .
```

Assuming that the obvious productions exist for all the non-terminals that are not defined above (i.e. ReadStatement, Assignment, Condition, etc.), which of the following Parva statements can be derived from the Statement production?

Select one:

- while (a< 10) a = a+1;
- while (a<10) { } ✗
- while (a<10) { a=a+1; }
- All of the statements given can be derived from Statement in the given grammar

Your answer is incorrect.

The correct answer is: All of the statements given can be derived from Statement in the given grammar

Question 2

Complete

Mark 3.5 out of 4.0

Flag question

Define exactly the conditions that must be satisfied for a grammar to earn the title "LL(1) grammar". That is, give the rules for determining LL(1) compliance.

Rule 1: Given options within a Terminal. (A | B). The first value of option 1 cannot be equal to the first value of option 2. ie they must be pairwise disjoint, this applies to as many pairs (options) as there are (A | B | C | D etc, none can have the same first as each other).

Rule 2: Given there is a null terminal option. The following set cannot be equal to that of the first set.

Comment:

Question 3

Complete

Mark 2.0 out of 4.0

Flag question

Is the following grammar LL(1) compliant? Show all your workings.

```
A ->  B | C A
B ->  "x" | "y"
C ->  "z" | ε

A -> B | D
D -> C A
B -> "x" | "y"
C -> "z" | null
First(C) -> {"z"}   **** null
First(b1) -> {"x"}
First b2 -> {"y"}
First B - > {"x","y"}
First D -> First C -> {"z"}   **** null
First A1 -> First B -> {"x","y"}
First A2 -> First D -> {"z"}   **** null
First A -> {"x","y","z"}   *** null
```

Rule 1 is satisfied as there are no options where there are duplicate values. They are pairwise disjoint.
Rule 2 breaks as the follow set of D is A (first -> {"x","y","z"}) while the first set of C is {"z"}. This means that the intersect of the two has a {"z"} and thus breaks rule two.
This means the grammar is not LL(1) compliant.

Not compliant. Fails both Rules.

Rule 1: First(A) = {x,y} OR First(C) u Follow(C) == {z} u First{A}
Thus two options for A do not have disjoint First sets.

Rule 2: First(C) n Follow(C) must be disjoint
{z} n First(A) must be disjoint -- Failed

Comment:
You have confused yourself by adding in production D -- there is no need to do this. The follow set of D is the follow set of A -- not the first set of A. The follow set of C is the first set of A.

Question 4

Complete

Mark 4.0 out of 4.0

Flag question

Consider the following "silly" grammar:

```
Goal = { X } [ Y ] { X Z } Goal .
X    = a | b .
Y    = c | d .
Z    = [ f ] g ] .
```

a) Write an equivalent grammar that does not use any of the meta-brackets, e.g. [ ] or { }

b) For your equivalent grammar, indicate which are the nullable non-terminals.

a)
Goal = A B C Goal
A = X A | null
B = Y | null
C = X Z C | null
X = a | b
Y = c | d
Z = D | null
D = f | g

b)
The nullable non-terminals are: A, B, C, Z, Goal

```
Goal = X1 Y1 XZ1 Goal .
  X1   = X X1 |  ε  .
  X    = a | b .
  Y1   = Y |  ε  .
  Y    = c | d .
  XZ1  = A C AC1 |  ε  .
  Z    = f | g |  ε  .          (3 marks)
```

All except X and Y (1 mark)

Comment:

Question 5

Complete

Mark 3.0 out of 4.0

Flag question

Consider the following simple grammar that has a single non-terminal A and two terminal symbols "(" and ")", and which satisfies both Rule 1 and Rule 2 for LL(1) compliance :

```
A  =  "("  A  ")"  A  |  ε
```

a) Write down 3 different strings that can be derived from production A.

b) Is this grammar ambiguous? Give a reason.

a)
i. (())
ii. (()) ()
iii. (()) () ()

b)
Yesit is ambiguous, although it is LL(1) compliant it can still be ambiguous and we can test this by giving it a string to parse.
If we try parse (()(())) for example.
A route would be, A -> (A) A -> ((A)A) A -> ((())(A)A -> ((())((A)A)A -> ((())(())) ; Making the second A null and the outer A expand.
Or it could be, A -> (A) A -> (A) (A) A -> ((A)A) ((A)A) A -> ((())(())) ; Making the A's null here would also make ((())(()))

Cannot be ambiguous as we are told it is LL(1) compliant.

Strings: () OR () () () OR ((())) () etc. basically any matching parenthesis.

Comment:
You cannot mix left- and rightmost derivations to show ambiguity.

Question 6

Complete

Mark 0.0 out of 2.0

Flag question

What incorrect Parva syntax will be accepted by a Parva parser programmed with the following productions to support an if-else statement :

```
Parva           =  "void" "main" "(" ")" Block .
Block           =  "{" { Statement } "}".
Statement       =  (  Block | IfStatement | ElseStatement | Assignment  ).
IfStatement     =  "if" "(" Condition ")" Block  [  ElseStatement  ] .
ElseStatement =  "else" Block.
```

Note: only comment on incorrect syntax related to the if-else statement.

```
if ( x<10)
  if(x = 10)
    if(x=10-5)
else
   x
```

This is the dangling else problem, this if statement will be able to create ifs without else and it will parse but still be incorrect. Sadly using a LL(1) parser we are not able to create a LL(1) compliant solution to this problem but could cure it by latching a else to the nearest if statement.

Comment:
The syntax you have used is not correct -- and what you have produced is not an error in this language or most other programming languages. A dangling else is not an error -- it is an acceptable form of programming.

Finish review