z 1)

a) i) C → a C a | b C b | epsilon | a | b

ii)

Method 1:

C → a b C | b a C | a C b | b C a | C b a | C a b | epsilon

6

Method 2:

C → aCbC

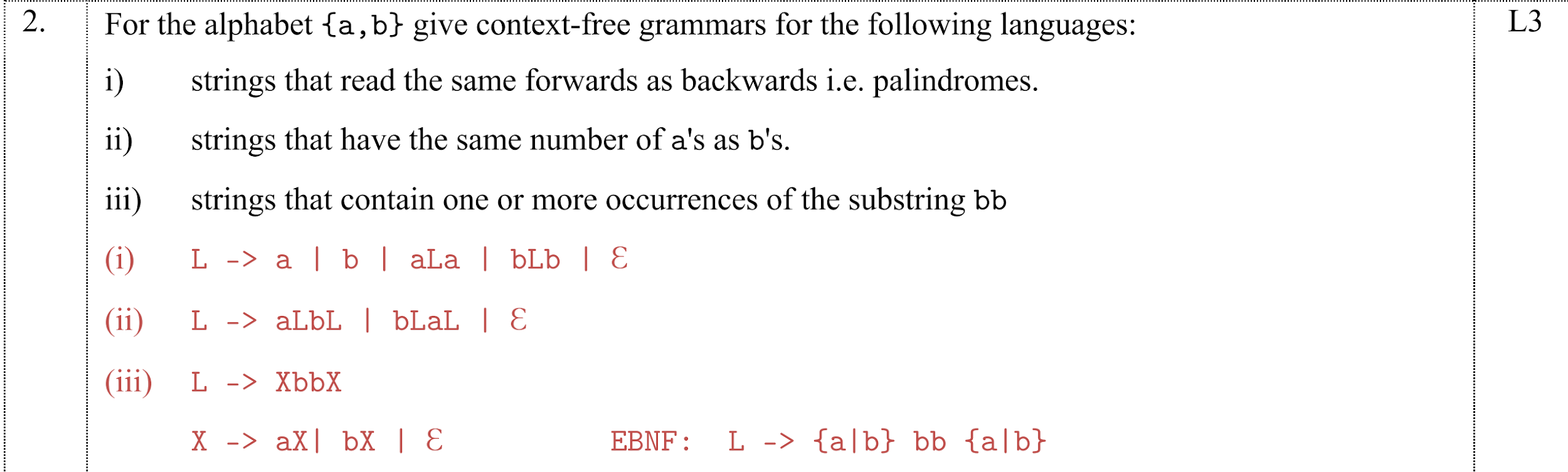
C → bCaC

C → epsilon

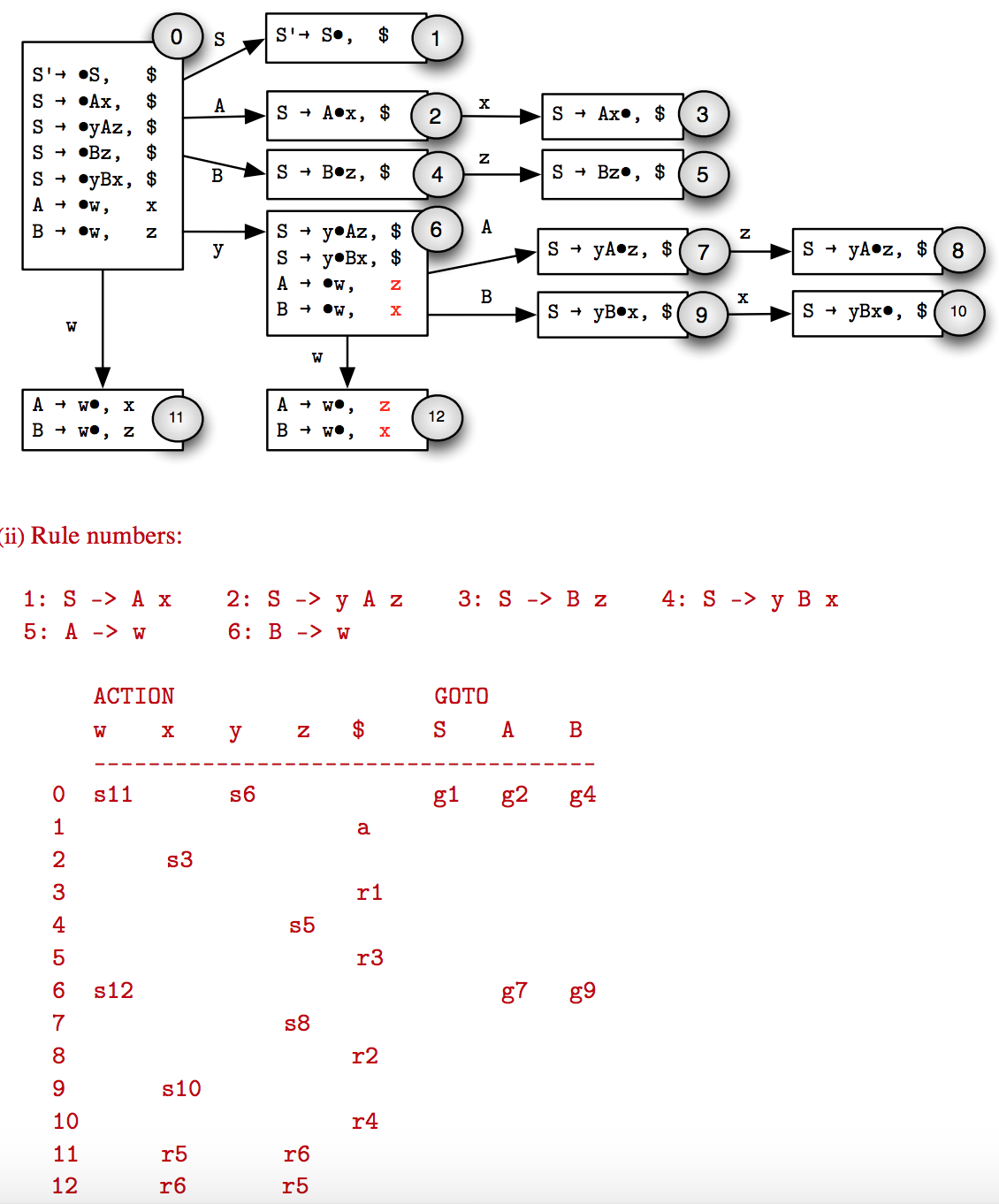
iii)C → D b b D

D → a D | b D | epsilon

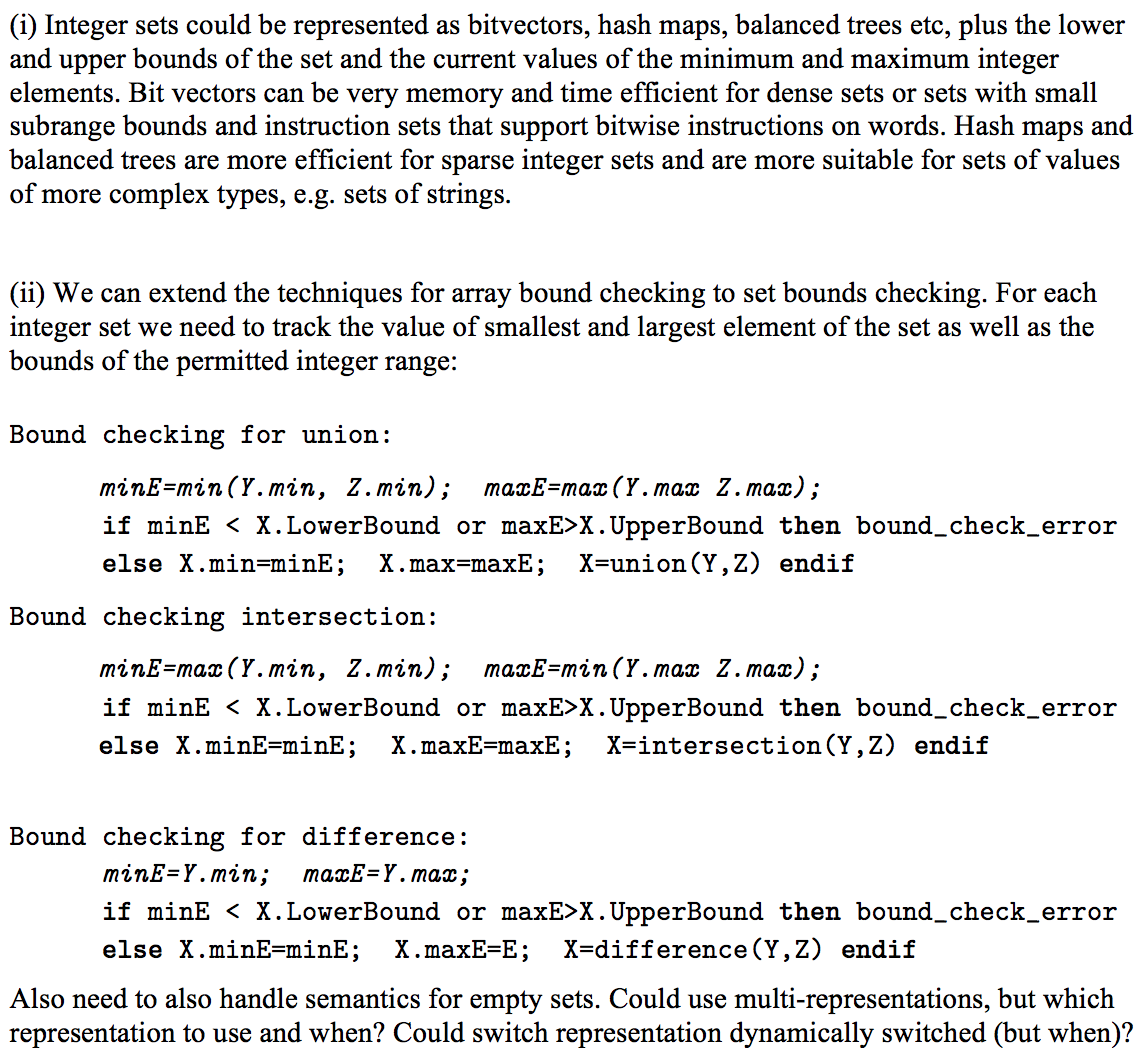
Official solutions:



b) i) ii)   
Note State 8 should be instead.



iii) There are no shift-reduce or reduce-conflicts, therefore the grammar is LR(1). For LALR, we can combine states 11 and 12. However, this new state would have a reduce-reduce conflict since the new state would result in an r5 and an r6 entry in both the x column and the z column.



2)

a)

An unconditional branch is not necessary. You can essentially have a “do {} while (condition)” structure where the last instruction will be a conditional branch. This will also need a conditional branch at the top to see if the code in the do block should ever be run.

CMP condition

BNE end ;condition never holds. Inner block never runs

Loop:bit bit

{} ;code block

CMP condition

BE loop ;condition held. Execute inner block again

end:

b) The callee save registers are general purpose registers that can be used anywhere. The compiler just has to make sure to save and restore them if they are used as part of a function call.

The compiler should choose to use callee-save registers if it does not know which registers the function called will need and definitely want the content of the callee-save registers to be preserved after the call.

The compiler should prioritise using caller-saved registers inside function as they already are saved by the calling function.

Optimising for code size (i.e. less instructions - not necessarily more efficient)

Also:

* Use callee-saves when caller is using a lot of registers and it would be expensive to save every single one of them in the caller, in case the callee happens to overwrite one
* Use callee-saves when callee does not use many registers, so it would be cheap to to save those few that it does actually use

**Paul’s Response to callee vs caller:**

If you choose a caller-saves register to hold some temporary t:

* You know it was saved by the caller(s) of f(). So it's free for you to use it
* But if f() contains a call to some other function g(), you'll have to generate code to save and restore the register (if t is used after the call).

If you choose a callee-saves register:

* You will have to generate code to save the old contents before you use it, and restore them before you return.
* But when you encounter a call to some other function g(), you don't have to save it. In fact you could have calls to lots of functions, interspersed with uses of t, and you still don't have to save it.

Basically:

* Choose a caller-saved register if you know this function does not many other calls in it (Each call = more assembly code to save and reload this register).
* Choose a callee-saved register if you know this function calls a number of other functions (You know those functions will save it at their start, so this function will not need to save/reload, it’s the job of those other functions to do so).

c)

Int y = 0

for (int i = 0; i < 3; i++) {

y++

}

for(int i = 0; i < 3; i++) {

if(i % 2 == 0) {

y++

} else {

y--

}

}

Could also do this (I think):

while (i < 10) {

while (j < 10) {

if (i = j) {

j += 1;

continue; // one back edge

}

print (i+j);

j += 1;

// another back edge

}

i += 1;

// and yet another back edge

}

I think this is clearer:

while (x < 10) {

while (y < 10) {

if (x < 5) {

...

y++; // 1st backedge

} else

{

...

y++; // 2nd backedge

}

}

x++; // 3rd backedge

}

d)

i)

LiveIn(n) : The set of temporaries live immediately before node n.

LiveOut(n) : The set of temporaries live immediately after node n.

ii)

They should be initialised to the empty set.

iii)

If you have t temporaries you can have up to t elements in the LiveIn and LiveOut set of each node.

iv)

– nodeDefs cfgnode = list of temporaries which this instruction updates

– nodeUses cfgnode = list of temporaries which this I nstruction reads

– nodeSuccs cfgnode = list of nodes which might be executed next

(below are probably implicitly already part of the node?)

* nodeId = unique CFG node identifier
* nodeInstruction = the instruction this node represents

v)

Backward data-flow problem because LiveOut(n

) depends on LiveIn(s) of its successors.

vi)

For each temp t

Set interferes(t) = {}

For each node n

If t in LiveOut(n)

Then interferes(t) = interferes(t) union LiveOut(n)

vii)

No, graph colouring is just too slow for a JIT compiler. A JIT compiler compiles bytecode at runtime so speed is essential. Graph colouring is more appropriate for a compiler that doesn’t run at run time. The difference in the quality of the code produced is not worth the performance cost of graph colouring. A better algorithm for a JIT compiler is Linear Scan as this is much faster than graph colouring but produces decent quality code too. This algorithm doesn’t produce a graph and just linearly scans the code to determine live ranges.