# COMP36512 – MODEL ANSWERS (RIS)

## QUESTION 1

a) a) Clearly, 1-c, 2-e, 3-h/i, 4-f/g, 5-h/i, 6-a, 7-b, 8-d, 9-f/g.
0.5 marks for each correct match rounded up.

**(5 marks)**

b) Lexical analysis error: a pattern in the input sequence cannot be detected by the rules of the grammar. For example, a symbol that is not part of the alphabet. Syntax analysis error: the tokens detected by lexical analysis do not form valid sentences. For example, the assignment statement a+b=3. Semantic analysis error: an example may be a function called using a different number of parameters wrt the function's declaration, incompatible types mixed in the same statement, etc…

**(3 marks)**

c) i) this would require a modification of the lexical analyser.
ii) this would require the modification of the lexical analyser, the parser, the semantic analysis component (and possibly the back-end if the target processor provides special instructions to deal with operations between complex numbers)
iii) in principle, nothing needs to change but the code wouldn't necessarily take advantage of the quad-core: to take advantage, one would expect that code generation has to be enhanced.
iv) this would require modifications of the lexical analyser, the parser and the semantic analyser (it is assumed that the construct would be mapped onto a standard loop structure in the intermediate representation).
Two marks each, any sensible explanations and answers will get full marks.

**(8 marks)**

d) Clearly, the whole of the back-end (to generate machine code) would not be necessary. One would expect a new component to generate the program slice and possibly additional intermediate representations to enable the generation of the program slice.

**(4 marks)**

**QUESTION 2**

a) This is an NFA because there are two transitions on the same symbol from S0 (symbol a pointing to S2 and S3).

**(2 marks)**

b) Checking all paths shows that all combinations of a b in strings of length 3 are acceptable: aaa, aab, aba, baa, abb, bab, bba, bbb.

**(3 marks)**

c) The subset construction algorithm repeatedly computes ε-closure(move(*,symbol)) for all symbols until no new states are found:

| step | | ε-closure(move(*,a)) | ε-closure(move(*,b)) |
|---|---|---|---|
| 1 | S0, S1 (A) | S0, S1, S2, S3 | S0, S1, S3 |
| 2 | S0, S1, S2, S3 (B) | S0, S1, S2, S3 | S0, S1, S3 |
| 3 | S0, S1, S3 (C) | S0, S1, S2, S3 | S0, S1, S3 |

Drawing the DFA essentially requires to draw the three new states and what transitions between them have been found – see the table above. The students should clearly draw the DFA.

**(5 marks)**

d) Hopcroft's algorithm for minimisation would start with two sets of states: final and non-final ones. It would then examine if all transitions on every symbol have a common pattern (that is, they lead to the same set, either within or outside). If they don't then the offending node(s) in the set would have to be split and form a new set. The algorithm is then repeated until no splits are possible: In this case, it is clear that all the final states are B and C and all transitions from A go to B and/or C and all transitions from B and C stay within B and C. Hence B and C can be grouped together. Final DFA not drawn here but students need to show it.

**(4 marks)**

e) This can be derived by taking all the paths of the final (minimised) DFA. The regular expression is (a|b) (a|b)*. This corresponds to all strings that can be formed with a and b and have a length of at least 1.

**(4 marks)**

f) Nothing would change as S2 would only mean that the new set of states B (the only that includes S2) is a final state but this is already the case for B (because of S3).

**(2 marks)**

**QUESTION 3**

a) (i) In order to show that the grammar is ambiguous, it is sufficient to show that there are two different rightmost (or leftmost) derivations for the same string. If we assume the string id or id and id, it is easy to build two different leftmost derivations:
E – E or E – id or E – id or E and E – id or id and E – id or id and id.
E – E and E – E or E and E – id or E and E – id or id and E – id or id and id

**(4 marks)**

(ii) One way of rewriting the grammar is:
E → E or T | T
T → T and F | F
F → not N | N
N → (E) | true | false | id

**(8 marks)**

b) The relevant steps are shown below:

| stack | input | action |
|---|---|---|
| $ 0 | (x+y)*z | Shift 2 |
| $ 0 ( 2 | x+y)*z | Shift 3 |
| $ 0 ( 2 x 3 | +y)*z | Reduce 4 |
| $ 0 ( 2 E 6 | +y)*z | Shift 4 |
| $ 0 ( 2 E 6 + 4 | y)*z | Shift 3 |
| $ 0 ( 2 E 6 + 4 y 3 | )*z | Reduce 4 |
| $ 0 ( 2 E 6 + 4 E 7 | )*z | Reduce 1 |
| $ 0 ( 2 E 6 | )*z | Shift 9 |
| $ 0 ( 2 E 6 ) 9 | *z | Reduce 3 |
| $ 0 E 1 | *z | Shift 5 |
| $ 0 E 1 * 5 | z | Shift 3 |
| $ 0 E 1 * 5 z 3 | EOF | Reduce 4 |
| $ 0 E 1 * 5 E 8 | EOF | Reduce 2 |
| $ 0 E 1 | EOF | Accept! |

The derivation found is:
E → E*E → E*id → (E)*id → (E+E)*id → (E+id)*id → (id+id)*id

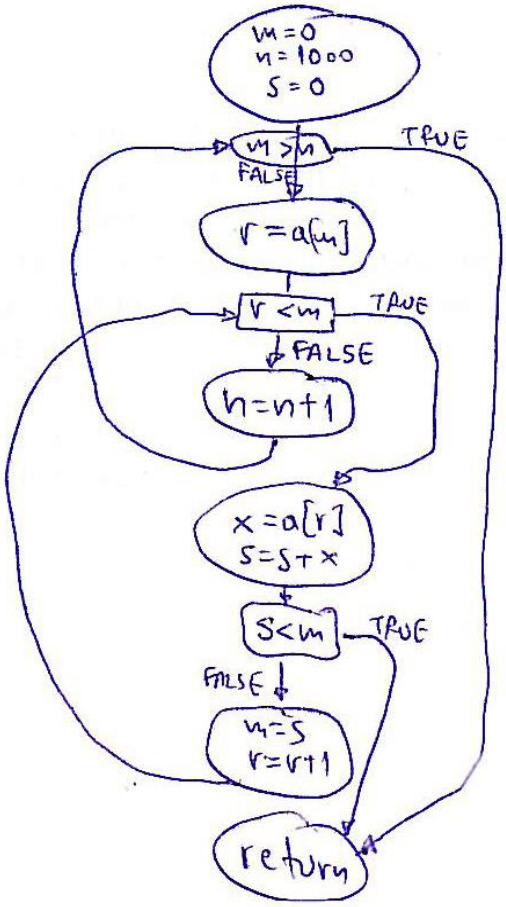
Parse Tree not drawn here but students must do it.

5 marks for showing the relevant steps; 3 marks for the parse tree.

**(8 marks)**

**QUESTION 4:**

a) Control Flow Graph:




**(4 marks)**

b) i) A generic approach could be:

1. Evaluate expr1; if expr2 is satisfied proceed else goto step 4.
2. loop body
3. Evaluate expr3; if expr2 is satisfied goto 2 else goto 4
4. code after loop

**(4 marks)**

ii) this is indeed happening with versions of the gcc compiler. The reason is that 2147483647 is MAXINT and line 3 above cannot produce an integer greater than MAXINT. Special care is needed when generating code to address the case where the upper bound of a loop is MAXINT.

**(4 marks)**

c) Applying copy propagation, constant propagation and constant folding the loop body becomes:
t=3; if (i==1) x=t+10; y=x+1000; a[i]=x+n;
If we split iteration 1 from the rest, we get: t=3; x=13; y=1013; a[1]=1013; followed by
for (i=2; i<=1000; i++) { a[i]=1013; } All the rest can be removed assuming that is not used later in the code. Students must show how the code looks like after every transformation.

**(8 marks)**

**QUESTION 5:**

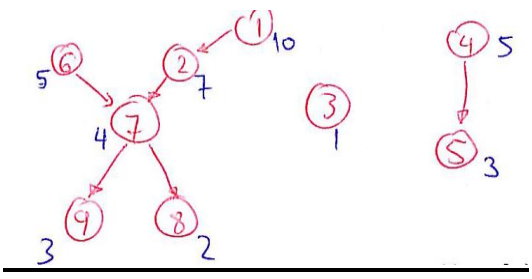
a) i) if we compute and check the live ranges for each value, we realize that no more than 4 values need to be live at any instruction, which proves the point:
r1 [1,4]
r2 [2,4]
r3 [3,6]
r4 [4,7]
r5 [5,8]
r6 [6,7]
r7 [7,8]
r8 [8,8] **(4 marks)**

ii) Best's algorithm can be used to ensure that for each operation the operands are already in a physical register (if they aren't, a physical register is allocated) – if they are not needed after the operation they are freed. As for the left hand side of the operation, a free register or the one used the farthest in the future is used (in which case, the current value will have to be spilled to the memory). One possible outcome of the algorithm is:
1. load r1, @x
2. load r2, @y
3. add r3, r1, r2
4. mult r4, r1, r2
5. add r1, r3, 1
6. add r2, r4, r3
7. sub r3, r2, r4
8. mult r2, r1, r3 **(4 marks)**

b) i) There are several ideas that relate to how instructions in the dependence graph are assigned weights as well as how they are chosen to be assigned to machines. If we use as a weight the latency of the fastest unit, we can identify the critical path. Then, when scheduling, the choice of the functional units can be made using the time that the instruction will be completed and when the result is needed. If the result is needed later in the execution, the instruction can be allocated to the slowest functional unit. Many solutions are acceptable. **(6 marks)**

ii) First we need to draw the precedence graph (see below), based on the observations:
9 depends on 7; 8 depends on 7 and 2; 7 depends on 2 and 6; 5 depends on 4; 2 depends on 1
Weights: 1: 10; 2: 7; 3: 1; 4: 5; 5: 3; 6: 5; 7: 4; 8: 3; 9: 1 (following minimum weight, not necessarily the best)

| Schedule: | | |
|---|---|---|
| 1st cycle: | 1 | 6 |
| 2nd cycle: | 4 | 3 |
| 3rd cycle: | nop | nop |
| 4th cycle: | 2 | 5 |
| 5th cycle: | 7 | nop |
| 6th cycle: | 9 | 8 |
| 7th cycle: | nop | nop |
| 8th cycle: | nop | nop |




**(6 marks)**