COMP36512 – QUESTION 1:

a) Easy:

1 – e or f
2 – g
3 – b
4 – i
5 – a
6 – c
7 – d
8 – h
9 – e or f

0.5 marks for each correct match rounded up.
(5 marks)

b) Warm up. A symbol table is a data structure that holds information about symbol names (variables, etc). Symbol table entries will be created by the front-end, depending on the implementation information may be captured by the lexer, the parser etc.
(3 marks)

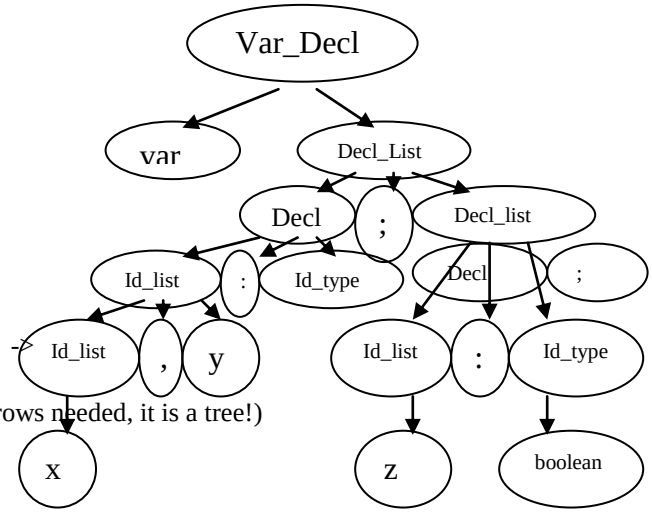
c) Students need to demonstrate an understanding of the main limitation of regular expressions: they lack 'memory'. So, the answer to (i), (iv), (v) is yes (and can be justified by showing the regular expression or making a reasonably good argument), the answer to (ii) and (iii) is not (and can be justified by just mentioning this regular expression limitation, which is a consequence of the strictness in describing rules). Poorly justified answers (or simple yes/no) will get at best half the marks.
(5 marks, 1 mark each correct answer)

d) First we write a regular expression for an integer from 0 to 255. Many ways to do this, one way is:
Integer255→ 0* digit | 0* digit digit | 0* 1 digit digit | 0* 2 (0 | 1 | 2 | 3 | 4) digit | 0* 2 5 (0 | 1 | 2 | 3 | 4 | 5)
Then, the regular expression for an IP address is:
IP_reg_expr → Integer255 . Integer255 . Integer255 . Integer255
(4 marks)

e) Requires some thought as the students may have not seen this, but the answer is straightforward. Add extra ε-transitions from all final states of the current NFA to a new state and make this new state the final one.
(3 marks)

COMP36512 – QUESTION 2

a) (i) A leftmost derivation:
Goal -> Var_Decl ->
var Decl_List ->
Decl ; Decl_List ->
Id_List : Id_Type ; Decl_List ->
Id_List, identifier_name : Id_Type ; Decl_List ->
identifier_name, identifier_name : Id_Type ; Decl_List ->
identifier_name, identifier_name : integer ; Decl_List ->
identifier_name, identifier_name : integer ; Id_List : Id_Type ->
identifier_name, identifier_name : integer ; identifier_name : Id_Type ->
identifier_name, identifier_name : integer ; identifier_name : boolean



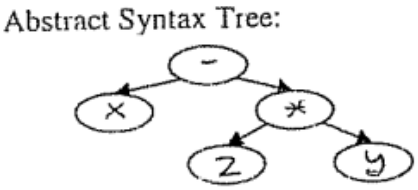
(**6 marks** – 3 marks for each of derivation and parse tree, NB: no arrows needed, it is a tree!)

ii) To use the grammar to build a top-down predictive parser with one symbol of lookahead, we should:
1) eliminate left recursion (as a result of the rule Id_List → Id_List , identifier_name). This rule can be replaced by:
Id_List → identifier_name Id_Rest and Id_Rest → , identifier_name Id_Rest | ε
2) make sure that the grammar has the LL(1) property (currently, Decl_List and Id_List have two productions with common prefix that need to be factored). Hence:
Id_List → identifier_name Id_Rest and Id_List → identifier_name become:
Id_List → identifier_name Id_List2 and Id_List2 → Id_Rest | ε
and
Decl_List → Decl ; Decl_List and Decl_List → Decl ; become:
Decl_List → Decl ; Decl_List2 and Decl_List2 → Decl_List | ε
(**6 marks**)

b) Easy application of the construct. The control flow graph is not drawn here, but students need to make sure that:
(i) maximal sequences of code without control flow interruptions are represented by a single node;
(ii) goto are just edges;
(iii) ifs clearly show the path for true and the path for false;
(iv) edges always show direction of control (the graph is a directed graph).

(4 marks)

c)



3-address code:

```
load  r1,y
loadi r2,2
mult r3,r2,r1
load r4,x
sub r5,r4,r3
```

2-address code:

```
load  r1,y
loadi r2,2
mult r2,r2,r1
load r3,x
sub r3,r3,r2
```

1-address code (stack code)

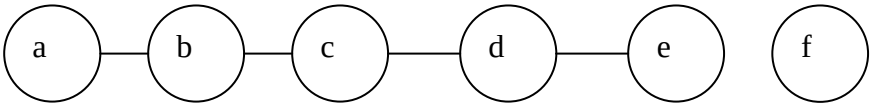
```
push  2
push  y
multiply
push  x
subtract
```

(4 marks)

COMP36512 – QUESTION 3

a) This is indeed an optimisation because it changes the access pattern of array a, allowing row-wise access of the array (which is expected to perform better).
If the assignment statement was a[j][i]+=a[j-1][i+1] then the semantics of the code would change (because there is a dependence between elements of the array and the transformation would violate this dependence, hence it would be illegal)
(**5 marks**)

b) First we find live ranges: a: [1, 3], b: [2, 4], c:[3, 5], d: [4, 6], e: [5, 6], f: [6, 7]
Then, we draw the interference graph, assuming that there is no interference in an instruction where a live range terminates and another one starts. Then, the interference graph becomes a simple linear graph:

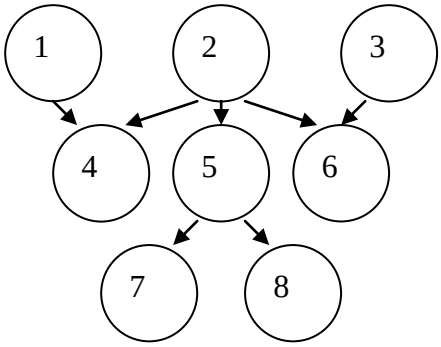


It is easy to realize that two colours are sufficient to colour the interference graph. Then, the code can be rewritten as follows:

```
a=1
b=a+3
a=a+b
b=b+5
a=a+7
b=a+b
return b
```

(6 marks)

c) Precedence graph:



To calculate priorities we take the maximum path to an exit node, hence:
1, 2, 3: have a priority of 3
5: has a priority of 2
4, 6, 7, 8: have a priority of 1

The schedule will depend on what tie-brakers we use (see next question), but one possibility is:
Cycle 1: 1, 3
Cycle 2: 2, nop
Cycle 3: 5, 4
Cycle 4: 6, 7
Cycle 5: 8, nop
Another possibility is:
Cycle 1: 1, 2
Cycle 2: 3, 5
Cycle 3: 4, 7
Cycle 4: 6, 8 **(5 marks)**

d) Two schedules for the example in c) were given above. They are both setting priorities using the longest path to exit. The first resolves ties randomly, the second one resolves ties by using the number of descendants. There are different possibilities to weigh instructions (e.g., number of descendants, longest path to exit, longest path + number of descendants, etc). The answer has to demonstrate that some approaches will produce schedules of length 4 (essentially if they recognize that instruction 2 is more important than 1 or 3), other approaches will produce schedules of length 5, as shown above.
(4 marks)