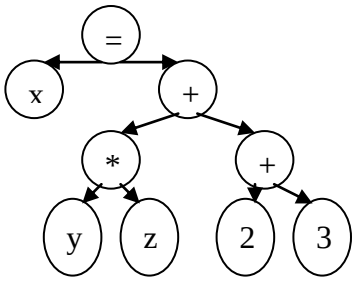
COMP36512 – QUESTION 1:

a) 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)

b) line 6 – syntax analyser (parser)
line 6 (declaration of y) – semantic analyser
line 7 – semantic analyser
line 7 – syntax analyser (parser)
line 8 – lexical analyser
line 9 – semantic analyser
line 10 – lexical analyser
It can be argued that, in some cases, the error message may be generated by another phase. Everything properly justified will be accepted as correct.
7 marks – 1 mark for each error message)

c) 1. Lexical Analysis generates tokens:
<id,x> <=,> <id,y> <*,> <id,z> <+,> <int,2> <+,><int,3>
2. Parsing will generate an abstract syntax tree (no need to produce a detailed parse tree):



3. Semantic analysis will annotate the syntax tree with context sensitive information. Since x,y,z are real numbers the representation of 2 will be 2.0 (or equivalent) and 3 will be 3.0 (or equivalent).
4. The intermediate code could be an abstract syntax tree (as above with annotations) or could be another form, for instance, 3-address code:

T1=y*z
T2=2+3
T3=T1+T2
x=T3

5. Intermediate code optimisation would apply constant folding (3+2 would become 5).
6. Initial code generation would produce symbolic machine code (using unlimited registers, without taking into account possible instruction scheduling constraints, etc):
7. Subsequently, register allocation and instruction scheduling would take into account resource constraints and availability of the target processor.
(**6 marks**)

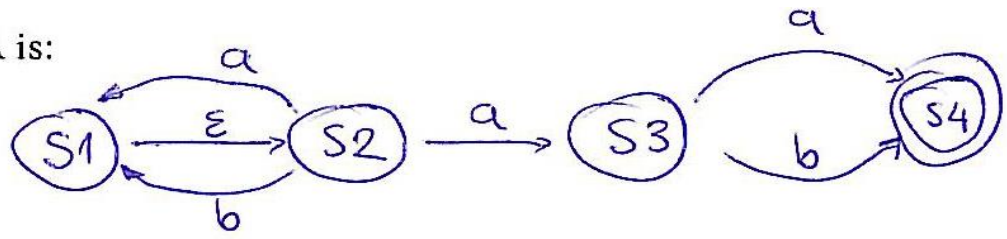
d) Back in the mid-1970s, when C was developed, it was considered necessary to let the programmer control which program variables reside in registers. At the time, target processors had very few physical registers and there wasn't much work on register allocation. Nowadays, with the plethora of registers (and the complexity that comes with different register types) and with the development of effective and sophisticated register allocation techniques, programmers cannot be considered as good judges of how register allocation should be performed and imposing some choices on the code may hurt performance (see Aho2, page 18)
(**4 marks**)

COMP36512 – QUESTION 2

a) Requires some thought 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)

b) 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 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) get up to half the marks.
(5 marks, 1 mark each correct answer)

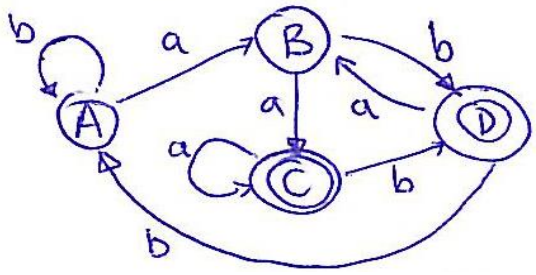
c) The NFA is:



Using the subset construction algorithm, we find sets of states:

	a	b
A: 1,2	1,2,3	1,2
B: 1,2,3	1,2,3,4	1,2,4
C: 1,2,3,4	1,2,3,4	1,2,4
D: 1,2,4	1,2,3	1,2

The DFA (which cannot be minimized further) is:

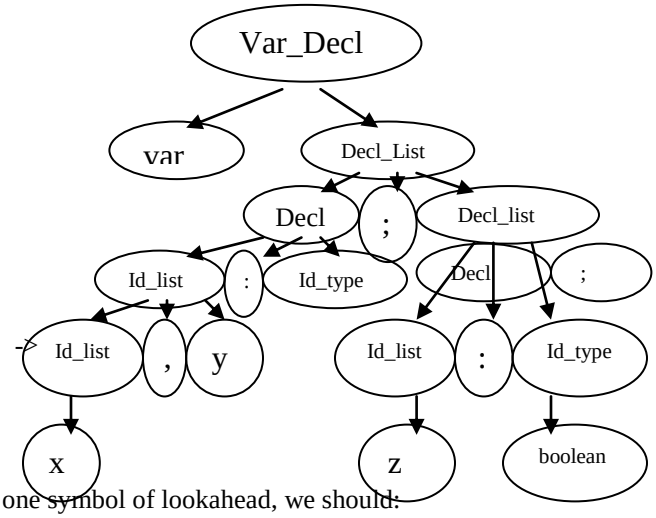


(7 marks)

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
(5 marks)

COMP36512 – QUESTION 3

a) 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 for each of derivation and parse tree)



b) To use the grammar to construct 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**)

c) In the most detailed form, the algorithm for a bottom-up shift-reduce parser would look like:

push $ onto the stack
token = next_token()
repeat
  if the top of the stack is a handle *A→b*
    then /* reduce *b* to *A* */ pop the symbols of *b* off the stack and push *A* onto the stack
    elseif (token != EOF)
      then /* shift */ push token; token=next_token()
      else /* error */ call error_handling()
until (top_of_stack == *Goal* && token==EOF)

A shift/reduce conflict arises when the parser can either shift or reduce.
The steps that a shift-reduce parser would follow to parse the string var x,y:integer; z:boolean; are:

Stack	Input	Action
$	var x,y:integer;	shift
var	x,y:integer;	shift
var x	,y:integer;	reduce Id_List → identifier_name
var Id_List	,y:integer;	shift
var Id_List,	y:integer;	shift
var Id_List, y	: integer;	reduce Id_List→Id_List,identifier_name
var Id_List	: integer;	shift
var Id_List :	integer;	shift
var Id_List : integer	;	reduce Id_Type→ integer
var Id_List:Id_Type	;	reduce Decl → Id_List : Id_Type
var Decl	;	shift
var Decl;	EOF	reduce Decl_List → Decl ;
var Decl_List	EOF	reduce Var_Decl → var Decl_List
Var_Decl	EOF	end

(**8 marks**)

COMP36512 – QUESTION 4

a) One possible way is:
push x
push 0
test
branch if false
push y
push 1
add
store x
(**3 marks**)

b) In the first approach code is generate from every subtree bottom-up. In the second approach, loads from memory are performed first. The first approach uses fewer registers but may cause more CPU stalls (waiting for values to arrive from the memory). In the second approach more registers are used but performance may increase (because loads are issued a few cycles before they are needed – prefetching – and more opportunities for executing instructions in parallel exist).
(**3 marks**)

c) One (simple and not necessarily the best) way to do this is to find the binary representation of the integer and then add all those powers of 2 that correspond to ones in the binary representation. For example:

```
Get binary representation of n and store it in an array of Boolean, size m, in ascending order of powers of 2 (true corresponds to 1 and false to 0):
i=0; first_power=true;
while (i<=m)
    if array[i] {
       if (first_power) {
             first_power=false; emit code(shl r1, 1, i; add r2, r1, 0;)
          } else {
             emit_code(shl r1, r1, i-last_power; add r2,r2,r1;)
          }
          last_power=i;
    }
```

(note that the usefulness of implementing multiplication with shift and add has been mentioned in the lectures, but no algorithm was presented)
(**5 marks**)

d) Java consists of many small procedures (i.e., methods). To minimize the overhead of calling these, method inlining, which is the replacement of a method call by the body of the method, is particularly useful. Also to minimize the overhead of garbage collection a compiler optimization can be used to allocate objects that are not accessible beyond a procedure on the stack instead of the heap.
(**4 marks**)

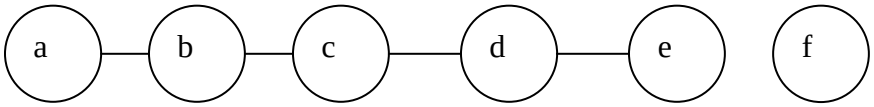
e) This is indeed an optimisation because it changes the access pattern of array a, allowing row-wise access of the array (which is expected to perform better). If the assignment statement was a[j][i]+=a[j-1][i+1] then the semantics of the code would change (because there is a dependence between elements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

COMP36512 – QUESTION 5

a) Register allocation relates to the problem of deciding what to keep in registers and what to store in the memory. Register assignment relates to the problem of assigning specific registers for specific values tha go to registers. It assumes that register allocation has been performed.
(2 marks)

b) Spilling is the process of storing a value into the memory (aiming to load it from memory when it is needed) because there are not enough registers to keep it in a register. Different approaches can be used to decide what to spill and have been presented in the lectures when describing different register allocation algorithms. For example, we can spill the values that are used less often in a basic block, we can spill the values that are used the furthest (from a given instruction), or, when using register allocation with graph colouring, we can spill live ranges that create more interferences (nodes with a high number of edges in the interference graph).
(4 marks)

c) 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:

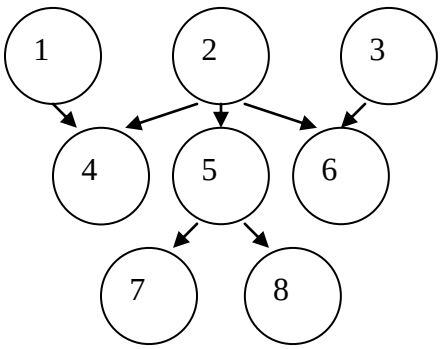


It is easy to realize that two colours are sufficient to colour the interference graph. Then, the code can be rewritten as follows:

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

(6 marks)

d) 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 tiebrakers 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)

e) Two schedules for the example in d) were given above. The first is based on a mechanism for setting priorities using the longest path to exit and resolving 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) and it can be demonstrated 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.
(3 marks)