MODEL ANSWERS – COMP36512, May 2016

QUESTION 1:

a) Clearly: 1-g, 2-d, 3-h, 4-e, 5-i, 6-a, 7-b, 8-c, 9-f. 0.5 marks for each correct answer rounded up as no halves are used.

(5 marks)

b) i) It has been mentioned in the lectures that FORTRAN stores arrays in a column-major fashion. Therefore, the first 2 columns of I4 would map to I1, the second 2 columns would map to I2, the third 2 columns would map to I3. Thus, the problem boils down to finding how to map a (50,2) array to a (10,10) array. The answer is that (i,j) would map to (1+((i-1)mod 10), (j-1)*5+int((i-1)/10)+1). It will require some thought and it may be challenging, hence 6 marks.

(6 marks)

b) ii) It has been mentioned in the lectures that compilers typically collapse any high-dimension array to consecutive memory locations. Using this capability, the compiler can convert all high-dimension arrays of two COMMON statements to single-dimension, virtual consecutive memory locations, which can then be linked together. Hence, this mapping is done through mapping to consecutive memory locations.

(5 marks)

b) iii) Clearly, COMMON statements are also a way to reuse memory space and minimize the memory requirements. This was particularly important in times when memory was limited. It can also be seen as a simple substitute for dynamic memory allocation.

(4 marks)

QUESTION 2

a) Clearly any integer greater than 599 will have either at least 4 digits or 3 digits and the first digit will be 6, 7, 8 or 9. Hence, the regular expression is:

Integer599 → digit0 digit digit digit digit* | 6 digit digit | 7 digit digit | 8 digit digit | 9 digit digit

where digit is any digit (0 to 9) and digit0 is any digit except 0

(4 marks)

b) The regular expressions are: b (ab)* and (ba | ab) b*
The DFAs are:

[not drawn]

(4 marks)

c) i) Yes, e.g., digit* (1 | 3 | 5 | 7 | 9)
ii) No, regular expressions cannot cope with 'memory'.
iii) No, same as above, we cannot capture the notion of 'same number of digits'.
iv) Yes, 11*00*.

(4 marks)

d) The regular expression is:

Float → digit+ . digit+ ((E (+|-|ε) digit+) | ε)
Digit → 0 | 1 | 2 | … | 9

(4 marks)

e) The DFA is:

[not drawn]

(4 marks)

QUESTION 3

a) i) This is both a left-recursive (because of rule 9) and a right-recursive (because of rule 2) grammar. Top-down parsing methods cannot handle left-recursive grammars because they attempt to build a leftmost derivation and left-recursion may lead to infinite substitutions.

(3 marks)

a) ii)

G-> L -> SL ->
If E then S else S endif L ->
If id then S else S endif L ->
If id then if E then S endif else S endif L ->
If id then if id then S endif else S endif L ->
If id then if id then id=E endif else S endif L ->
If id then if id then id=constant endif else S endif L ->
If id then if id then id=constant endif else id=E endif L ->
If id then if id then id=constant endif else id=constant endif L ->
If id then if id then id=constant endif else id=constant endif S L ->
If id then if id then id=constant endif else id=constant endif id=E L ->
If id then if id then id=constant endif else id=constant endif id=id L ->
If id then if id then id=constant endif else id=constant endif id=id endprogram

[parse tree drawn here]

(6 marks)

a) iii) To construct a top-down predictive parser with one symbol of lookahead, we should:
1) eliminate left recursion (as a result of the rule E → E id). This rule (and all other rules that have E on the left hand side) can be replaced by:
E → id Id_Rest, E → constant Id_Rest and Id_Rest → + id Id_Rest | ε
(essentially, the left recursion is transformed to right recursion, but we don't care about the latter!)
2) make sure that the grammar has the LL(1) property (currently, S has two productions with common prefix that need to be factored). Hence, rewriting these two rules (S → if E then S endif and S → if E then S else S endif) we get:
S → if E then S Remaining_If and Remaining_If → else S endif | endif

(6 marks)

b)

$ s0	zxy	shift 5
$ s0 z s5	xy	reduce (5)
$ s0 B s4	xy	shift 7
$ s0 B s4 x s7	y	reduce (4)
$ s0 A s3	y	shift 6
$ s0 A s3 y s6	eof	reduce (3)
$ s0 S s1	eof	accept

(5 marks)

QUESTION 4

a) For the call graph see:

[not drawn]

When the execution reaches the printf statement for the first time, it has called the following functions: A(4) (from main), A(2) (from A), A(0) (from A), C(0) (from A), B(0) (from C). So, including main( ), there will be six activation records in the stack, each activation record corresponding to a function call.
(4 marks)

b) After constant propagation n is replaced by 64 throughout. After constant folding tmp=64*4 is replaced by tmp=256. After copy propagation tmp is replaced by 256 throughout. Dead-code elimination would remove tmp. Finally, after reduction in strength:

```
n=64
i=1
L1: if (i>=64) goto L2
A(i)=256
B(i*4)=1
i=i+1
if (i<64) goto L1
L2: …
```

Of course students have to show the code after each transformation is applied as requested.
(6 marks)

c) One simple solution is the following:

```
for (i=0; i<n-s+1; i+=s)
{ loop_body (repeated s times, from i to i+s-1) }
for (j=i; j<n; j++)
        { loop_body }
```

Loop unrolling has been mentioned in the lectures in terms of how it is applied, but not as in the above (i.e., how it is derived).
(6 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 optimisation can be used to allocate objects that are not accessible beyond a procedure on the stack, instead of the heap.
(4 marks)

QUESTION 5

a) i) All we need to do is find live ranges: r1 [1,4]; r2 [2,4]; r3 [3,6]; r4 [4,7]; r5 [5,8]; r6 [6,7]; r7 [7,8]; r8 [8,8]

R1	X	X	X	X				
R2		X	X	X				
R3			X	X	X	X		
R4				X	X	X	X	
R5					X	X	X	X
R6						X	X	
R7							X	X
R8								X

We can observe that no more than 4 registers need to be live at any given instruction.

(4 marks)

a) 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) First build the precedence graph:

9 depends on 3 and 1
8 depends on 3
7 depends on 5
6 depends on 5 and 3
5 depends on 4
3 depends on 2

Then assign weights based on latencies and maximal sum of latencies to exit:

1,2,4 have a weight of 4
3,5 have a weight of 3
9 has a weight of 2
6,7,8 have a weight of 1

We schedule an instruction that is available, starting with those with highest delay (in case of a tie we choose randomly - one could think of other ways of resolving ties too –see next):

Cycle1	1	2
Cycle2	4	3
Cycle3	5	9
Cycle 4	nop	nop
Cycle5	6	7
Cycle6	8	

However, if we resolve ties by considering, say, the number of descendants, then:

Cycle1	4	2
Cycle2	1	5
Cycle3	3	nop
Cycle4	9	6
Cycle5	8	7

(5 marks)

c) Multiple basic blocks can be scheduled by drawing the dependence graph of each basic block, assigning weights as usual and then picking nodes based on their weight and regardless of the fact that they belong to different graphs.

(4 marks)

d) The availability of parallelism in a basic block (hence no loops, etc) will depend on the number of instructions that can be executed in parallel. This information can be found from the dependence graph, looking for the largest number of independent tasks.

(3 marks)