COMP36512 – May/June 2015

MODEL ANSWERS to exam questions
(6 pages)

QUESTION 1

a) Clearly (warm-up): 1-b; 2-h; 3-f or g; 4-f or g; 5-c; 6-a; 7-i; 8-d; 9-e.
0.5 marks for each correct mark rounded up.
(5 marks)

b) 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 (as there is backwards compatibility) but the code wouldn't necessarily take advantage of the dual 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) ; (v) this would require some modifications of the low-level code optimiser.
(**10 marks**, 2 each)

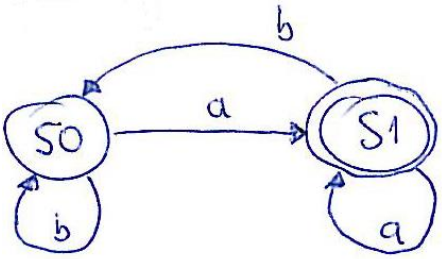
c) This transformation is a combination of loop skewing (add i to the innermost loop of the original code) and loop interchange (new loop bounds have to be calculated). This is a typical wavefront computation. In the original version, none of the loops is parallelizable because of the loop-carried dependences. In the transformed version, different iterations of the i loop can be executed in parallel, as the optimization groups the dependences for different iterations of the j loop. Students have seen a range of transformations and/or optimisations and their impact was discussed in the lectures (including the impact on code parallelization), but the specific example (parallelization of a wavefront computation) was not shown to them.
(5 marks)

QUESTION 2

a) Clearly, all such integers should end with one of 00, 25, 50, 75. Then, the regular expression needs to generate all 5-digit integers between 40000 (the lowest multiple of 25 which is greater than 39980) and 99999 that obey this rule. Hence:
RE → (4|5|6|7|8|9) (digit) (digit) (00 | 25 | 50 | 75)
(3 marks)

b) This language generates strings containing any number of a or b in any order, with the final symbol being always a. For example: a, aa, ba, aaa, aba, baa, bba, etc…

The DFA is shown below (it shouldn't need a detailed construction, it can be drawn by studying carefully the regular expression)



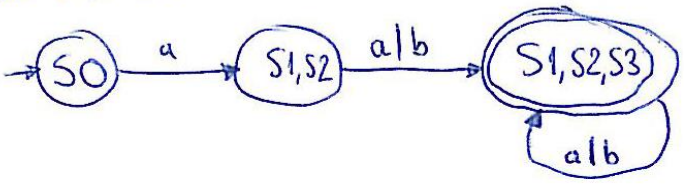
(**5 marks**: 2 marks for the description and 3 marks for the DFA)

c)

Starting from ε-closure(S0) and applying ε-closure and move repeatedly according to the subset construction algorithm, we get:

	a	b
S0	S1, S2	—
S1, S2	S1, S2, S3	S1, S2, S3
(final) S1, S2, S3	S1, S2, S3	S1, S2, S3

So the DFA is:



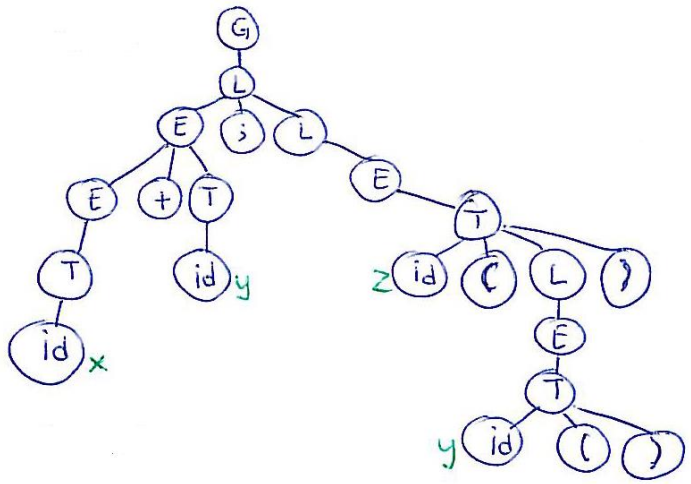
(6 marks)

d) comm → (/ x (other | \n | " | x | / | " x / ")* x /) | (/ / (other | / | " | x)* \n)
(the regular expression requires some thought about how to handle the double quotes)
(6 marks)

QUESTION 3

a) i) Leftmost derivation:
G → L → E ; L → E+T ; L → T + T ; L → id + T ; L → id + id ; L → id + id; E → id + id; T → id + id; id (L) → id + id; id (E) → id+id; id(T) → id+id; id (id( ))

Parse tree:



(5 marks)

ii) to transform this grammar so that it can be used to construct a top-down predictive parser with one symbol of lookahead we first need to eliminate left recursion as a result of the rules E → E + T | T, which become: E → T E' and E' → + T E' | ε
Then, we have to eliminate common prefixes to make sure that the LL(1) property holds. We do this by factoring. Then, rules: L → E ; L, L → E become: L→ E L' and L' → ;L | ε. Same for rules T → id, T → id ( ), T → id (L), which become: T → id T' and T' → (T'' | ε and T'' →) | L).
(6 marks)

b) i) The relevant steps taken are:

Stack	Input	Action taken
$ 0	(( )) ( )	Shift 3
$ 0 (3	( )) ( )	Shift 6
$ 0 (3 (6	)) ( )	Shift 10
$ 0 (3 (6) 10	) ( )	Reduce 5
$ 0 (3 P 5	) ( )	Shift 8
$ 0 (3 P 5) 8	( )	Reduce 4
$ 0 P 2	( )	Reduce 3
$ 0 L 1	( )	Shift 3
$ 0 L 1 (3	)	Shift 7
$ 0 L 1 (3) 7	eof	Reduce 5
$ 0 L 1 P 4	eof	Reduce 2
$ 0 L 1	eof	accept

(5 marks)

(ii) In the case of the Goto table, there are as many columns as non-terminal symbols, so no changes are expected. In the case of the Action table, when we have 2 lookahead symbols, we need to consider all possible combinations: ( ), ((,)),) (, (eof,) eof, eof, for a total of 7 columns. In the case of 3 lookahead symbols, we can easily calculate that we need a total of 15 combinations/columns. The key is to notice that some options are not possible, e.g., eof), etc.
(4 marks)

QUESTION 4

a) A possible solution (using only synthesized attributes) is the following:

G → E	G.val = E.val
E → E1 + T	E.val=E1.val + T.val
E → T	E.val=T.val
T → T1 * F	T.val=T1.val × F.val
T → F	T.val=F.val
F → (E)	F.val=E.val
F → digit	F.val=digit.lexval

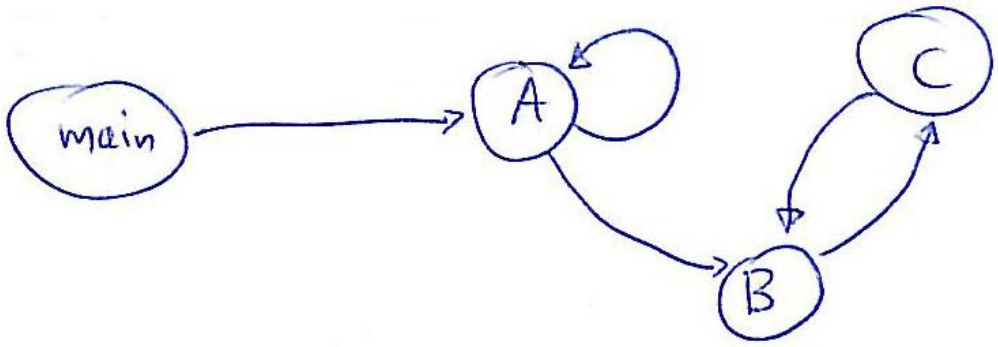
The only assumptions are related to the lexical value of digit and the use of E1 and T1 (2nd and 4th rules) to specify the order of calculation of the attributes.

(5 marks)

b) As there are only 18278 different names, a possible approach is to compute a different value for each name and then perform a modulo 1024 operation. This would map about 18 different names to each symbol table entry, and assuming a good distribution for common names (e.g., single letter names) chances of collision would be minimized. Thus, a good hash function could be: [(ascii(LETTER1)-(ascii('a')+1) + (ascii(LETTER2)-(ascii('a')+1)*26 + (ascii(LETTER3)-(ascii('a')+1)*26*26] modulo 1024 (assuming a three letter name, LETTER1 LETTER2 LETTER3 – for shorter names, the corresponding components are removed). Any sensible answers that guarantee a good/balanced distribution of names on the table will get full marks.

(5 marks)

c) For the call graph, see below:



When the execution reaches the printf statement for the first time, it has called the following functions: A(1) (from main), A(0) (from A), B(0) (from A), C(0) (from B). So, including main( ), there will be five activation records in the stack, each activation record corresponding to a function call.

(5 marks)

d) Applying constant propagation first: the 2nd line becomes if (3>7)…, the 3rd line becomes …my_function(z*0), the 4th line becomes for(i=10; i<=5; … Applying dead-code elimination, the 2nd and 4th line disappear, applying constant folding the 3rd line becomes my_function(0). This results in:

```
b=4; c=1; d=3; n=5; scanf("%d",&z);
q=my_function(0);
printf("final %d \n",q);
```

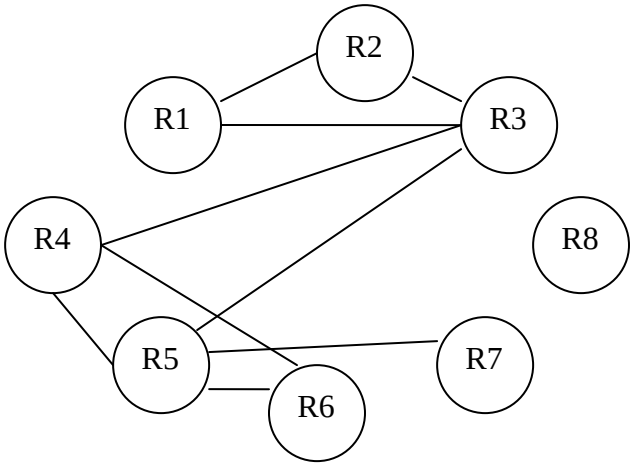
Assuming that the variables b, c, d, n, z are not needed anywhere else in the code, the first line can be eliminated too.

(5 marks)

QUESTION 5

a) i) 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]
(3 marks)

ii) interference graph



Top-down colouring starts with the nodes that have the largest number of edges. Then:
R3 – green; R5 – red; R1 – red; R2 – blue; R4 – blue; and R6,R7,R8:green
(5 marks)

b) We start from the leaves, hence:
Basic Block including line 7: LIVEOUT={}, LIVEIN={B,C,D}
Basic Block including lines 5,6: LIVEOUT={B, C, D}, LIVEIN={A, B}
Basic Block including lines 3,4: LIVEOUT = {B, C, D}, LIVEIN={A, D}
Basic Block including lines 1,2: LIVEOUT = {A,B,D}, LIVEIN={B}
(4 marks)

c) First build the precedence graph: (precedence graph not drawn here)
8 depends on 3 and 7; 7 depends on 5 and 2; 6 depends on 1 and 4; 5 depends on 4 and 1

Then assign weights; as latency is always 1 this is the longest path to exit:
1,4 have a weight of 4; 2,5 have a weight of 3; 3,7 have a weight of 2; 6,8 have a weight of 1

We schedule an instruction that is available, starting with those with highest weight:

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

Or, in the case of 2 functional units:

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

Comparing the two schedules: (i) the length is the same; (ii) in the first case, we use cheaper functional units but utilisation is low.
(8 marks)