IPS: exam21 solution

Sebastian Giovanni Murgia Kristensen [xwj716]

August 22, 2022

Contents

1 Task 1

1.1 1.1

1 Regular languages and automata

Question 1.1 (6 pts) Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ be the alphabet under consideration.

a. Consider the language L consisting of words of unbounded length that contain no identical *adjacent* letters. Examples of words in L: ε, `a`, `acab`, `bab`; non-examples: `cc`, `abba`, `baaa`.

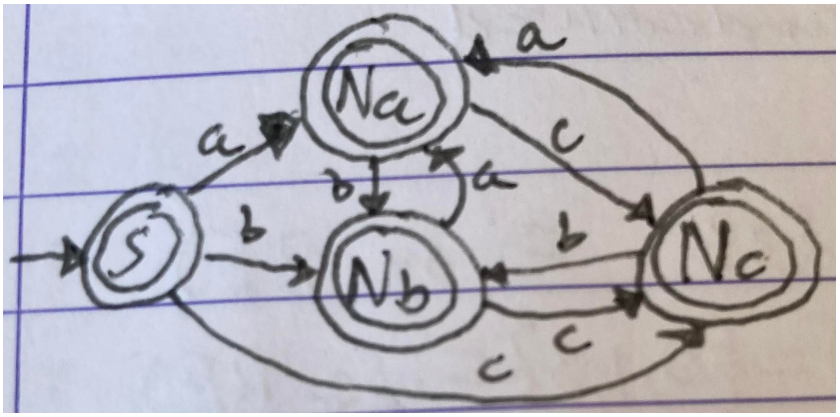
Is L regular? Explain why (for example, by showing that it can be described by a regular expression, recognized by a DFA, or similar), or why not (for example, by arguing that it cannot be recognized by any finite automaton).

b. Now, let the language L' be defined like L, but with the extra restriction that a word's first and last characters are *also* considered adjacent, in a wrap-around manner, and must hence be different. (In particular, words of length 1 do not satisfy the restriction.) Thus, the examples `a` and `bab` above would be excluded from L', but ε and `acab` are still allowed.

Is L' regular? Again, explain why or why not.

1.1.1 a)

L is regular, for example it is recognized by the following DFA:



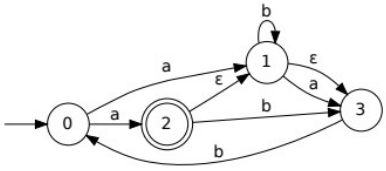
1.1.2 b)

L' is also regular. This can be shown by a somewhat larger DFA, with essentially three copies of the above one, one for each possible starting letter. Alternatively, we can note that L' = L/M, where M is

the language consisting of words that start and end with the same letter. This suffices, because regular language are closed under set difference (or intersection and complementatation), and M is evidently expressible with the following RE: $(a|b|c)|a(a|b|c)*a|b(a|b|c)*b|c(a|b|c)*c$

1.2 1.2

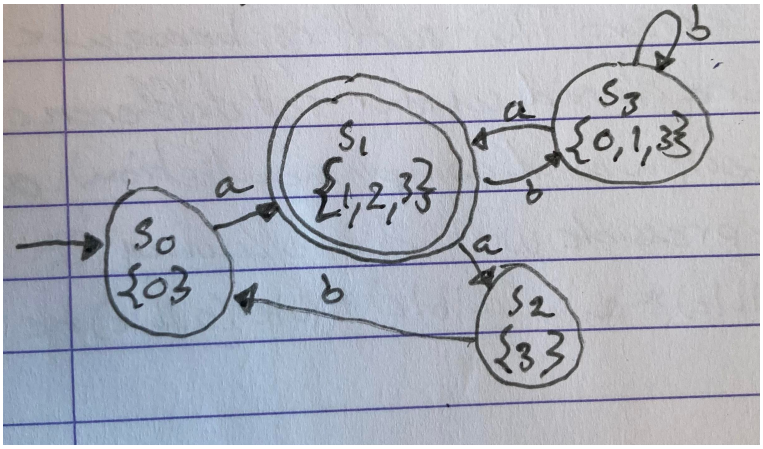
Question 1.2 (7 pts) Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Convert the following nondeterministic finite automaton (NFA) to a deterministic one (DFA), using the subset construction:



NFA for conversion

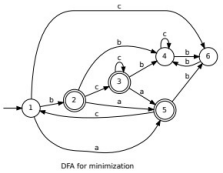
Show step-by-step how you determine the states and transition function of the DFA, and *draw* the final DFA. (You don't need to show in detail how you compute the ε-closures, but it should be clear in all instances what set you are taking the closure of, and what the result is.)

```
start = ec({0}) = {0} := s0 REJ

move(s0,a) = ec({1,2}) = {1,2,3} =: s1 ACC
move(s0,b) = ec({}) = undefined

move(s1,a) = ec({3}) = {3} =: s2 REJ
move(s1,b) = ec({1,3,0}) = {0,1,3} =: s3 REJ

move(s2,a) = ec({}) = undefined
move(s2,b) = ec({0}) = s0

move(s3,a) = ec({1,2,3}) = {1,2,3} = s1
move(s3,b) = ec({1,0}) = {0,1,3} = s3
```



1.3 1.3

Question 1.3 (10 pts) Let $\Sigma = \{a, b, c\}$. Minimize the following DFA, using the algorithm from the textbook. Show how the groups are checked and possibly split, and *draw* the final, minimized DFA.



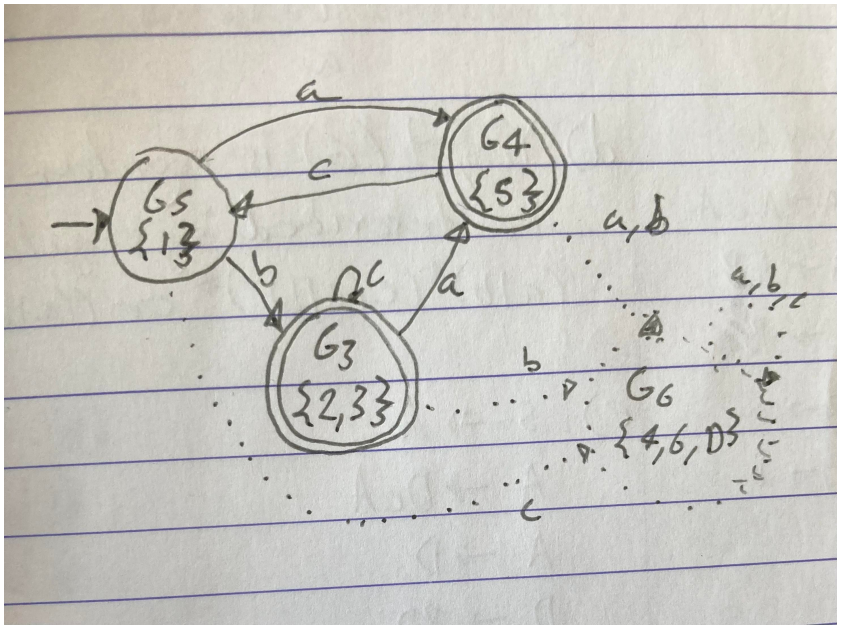
3

Be sure to deal properly with the limitations of the basic algorithm with respect to possible dead states: if you make *move* total, make it clear how you added the new dead state (but you don't need to actually *draw* the corresponding DFA), and how it is treated at the end. Conversely, if you eliminate any dead states before using the minimization algorithm, you must show how you *algorithmically* determined that those are precisely the dead states, not merely postulate it.

We add dead state D, replacing all originally undefined transitions with transitions to D.

```
G1 = {2,3,5} ACC
G2 = {1,4,6,D} REJ

G1  a   b   c
2   G1  G2  G1
```

```
3   G1  G2  G1
5   G2  G2  G2
Inconsistent

G3 = {2,3}
G4 = {5}

G2  a   b   c
1   G4  G3  G2
4   G2  G2  G2
6   G2  G2  G2
D   G2  G2  G2
Inconsistent

G5 = {1}
G6 = {4,6,D}

G3  a   b   c
2   G4  G6  G3
3   G4  G6  G3
Consistent

G6  a   b   c
4   G6  G6  G6
6   G6  G6  G6
D   G6  G6  G6

G4  a   b   c
5   G6  G6  G5

G5  a   b   c
1   G4  G3  G6

  | a   b   c
--------------
G3| G4  -   G3  ACC
G4| -   -   G5  ACC
G5| G4  G3  -   START, REJ
```



2 Task 2

2.1 2.1

2 Context-free grammars and syntax analysis

Question 2.1 (9 pts) Consider the following context-free grammar G:

$S \rightarrow A$
$A \rightarrow A \texttt{c} A$
$A \rightarrow A\ B$
$A \rightarrow B$
$B \rightarrow \texttt{a}$
$B \rightarrow \texttt{b}$

(The terminals are `a`, `b`, and `c`. The start symbol is S.)

a) Can string `caba` be derived from G? If it can, show a rightmost derivation $S \Rightarrow \ldots$.

b) Can string `baca` be derived from G? If it can, show a rightmost derivation $S \Rightarrow \ldots$.

c) Show that G is ambiguous. Find a string and show that it has two different syntax trees.

d) Is $L(G)$ regular? Justify your answer (max. 3 lines).

e) Make an unambiguous grammar for $L(G)$ that is right recursive.

2.1.1 a

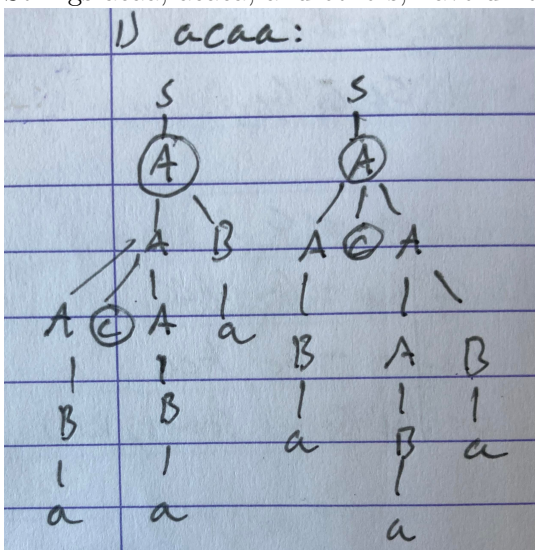
No, there is no derivation for caba

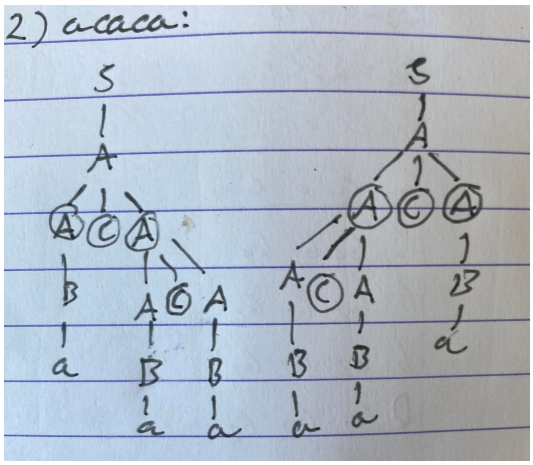
2.1.2 b

Yes, there is a derivation for baca. Rightmost derivation:
$S => A => AcA => AcB => Aca => ABca => Aaca => Baca => baca$

2.1.3 c

Strings acaa, acaca, and others, have different syntax trees. Thus, G is ambiguous:





2.1.4 d

Yes, L(G) is regular because it can be described by regular expression:
$(a|b)^+(c(a|b)^+)^*$ or $((a|b)^+c)^*(a|b)^+$

2.1.5 e

```
S -> A
A -> DcA
A -> D
D -> BD
D -> B
B -> a
```

```
B -> a
```

2.2 2.2

Question 2.2 (14 pts) Consider the following context-free grammar:

$S \rightarrow A\ \$$
$A \rightarrow [\ B\ ]$
$B \rightarrow 2\ C$
$B \rightarrow \epsilon$
$C \rightarrow 0\ B$
$C \rightarrow 1\ B$

(The terminals are `0`, `1`, `2`, `[`, and `]`, while `$` is the terminal representing the end of input. The start symbol is S.)

a) Show which of the above nonterminals are nullable. (Show the calculations.)

b) Compute the following first-sets:

4

$First(S) =$
$First(A) =$
$First(B) =$
$First(C) =$

c) Write down the constraints on follow-sets arising from the above grammar. (You may omit trivial or repeated constraints.)

$\{\$\} \subseteq Follow(A)$ (complete the rest)

d) Find the *least* solution of the constraints from (c). (Show the calculations.)

$Follow(A) =$
$Follow(B) =$
$Follow(C) =$

e) Compute the LL(1) lookahead-sets for all the productions of the grammar. Can one always uniquely choose a production?

$la(S \rightarrow A\ \$) =$
$la(A \rightarrow [\ B\ ]) =$
$la(B \rightarrow 2\ C) =$
$la(B \rightarrow \epsilon) =$
$la(C \rightarrow 0\ B) =$
$la(C \rightarrow 1\ B) =$

f) Construct the LL(1) table for the grammar. The table should work with the table-driven LL(1) parser program of the textbook.

g) Show the input and stack at each step during LL(1) parsing of the string `[21]$` using the table-driven LL(1) parser program of the textbook. You can assume that the program initially pushes start symbol S on the stack.

h) What is the derivation order performed by LL(1) parsers?

2.2.1 a

```
Null(S) = false // always because of $
Null(A) = null([B]) = false
```

```
Null(B) = null(2C) || null(eps) = false || true = true
Null(C) = null(0B) || null (1B) = false || false = false
```

2.2.2 b

```
First(S) = First(A$) = First(A) = {[}
First(A) = First([B]) = {[}
First(B) = First(2C) U First(eps) = {2} U {} = {2}
First(C) = First(0B) U First(1B) = {0} U {1} = {0,1}
```

2.2.3 c

```
Production          Constraints
S -> A$             {$}(- follow(A)
A -> [B]            {]}(- follow(B)
B -> 2C             Follow(B) (- Follow(C)
B -> eps
C -> 0B             Follow(C) (- Follow(B)
C -> 1B
```

2.2.4 d

```
Follow(A) = {$}
Follow(B) = {]}
Follow(C) = {]}
```

2.2.5 e

```
la(S->A$) = First(A) = {[}

la(A->[B]) = First([B]) = {[}

la(B->2C) = First(2C) = {2}
la(B->eps) = First(eps) U Follow(B) = {]}

la(C->0B) = First(0B) = {0}
la(C->1B) = First(1B) = {1}
```

2.2.6 f

	[	]	0	1	2	$
S	S→ A\$					
A	A→ $[B]$					
B		B→ ϵ			B→ $2C$	
C			C→ $0B$	C→ $1B$		

2.2.7 g

Input	Stack
[21]$	S
[21]$	A$
[21]$	[B]$
21]$	B]$
21]$	2C]$
1]$	C]$
1]$	C]$
]$	B$
]$	]$
$	$

2.2.8 h

LL(1) parsers perform leftmost derivations

3 Task 3

3.1 3.1

3 Interpretation and type checking

Question 3.1 (5 pts) This task refers to interpretation. Consider an extension of the functional source language from the textbook's Chapter 4 with the following production:

$$Exp \rightarrow \texttt{natcase } Exp \texttt{ of 0 -> Exp | id + 1 -> } Exp$$

(Note that the | on the RHS of the production is a terminal symbol, not an alternative-separator in the grammar.) The two numeric constants are only allowed to be precisely 0 and 1, as shown.

The intended semantics of an expression

$$\texttt{natcase } Exp_1 \texttt{ of 0 -> } Exp_2 \texttt{ | id + 1 -> } Exp_3$$

is that Exp_1 is evaluated first. If it evaluates to the integer 0, then Exp_2 is evaluated as the result of the `natcase`. On the other hand, if Exp_1 evaluates to an integer $n > 0$, then the result is obtained by evaluating Exp_3 with **id** bound to the integer $n - 1$. If Exp_1 evaluates to a negative integer, or to something that is not an integer at all, then an error is signaled.

For example, using the new construct, the usual factorial function could be written as follows:

```
int fac(int x) = natcase x of 0 -> 1 | y + 1 -> x * fac(y)
```

Extend the interpretation function $Eval_{Exp}$ with a case for the `natcase`-construct. That is, complete the following table entry:

$Eval_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
...	...
`natcase` Exp_1 `of` `0 ->` Exp_2 `| id + 1 ->` Exp_3	(Your code would go here)

Remember to check for error conditions!

```
v_1 = EvalExp(Exp1,vtab,ftab)
    if v_1 is not an integer then error()
    else if v_1 = 0 then EvalExp(Exp2,vtab,ftab)
    else if v_1 > 0 then
        vtab' = bind (vtab,getname(id),v_1-1)
        EvalExp(Exp3,vtab',ftab)
    else error()
```

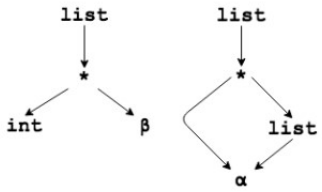
3.2 3.2

Question 3.2 (5 pts) This task refers to type-expression unification. Do the following types unify? (α and β are universally quantified type variables, and `list` and `*` are type constructors.)

`list(int * β)`

`list(α * list(α))`.

The graph representation of the two types is:



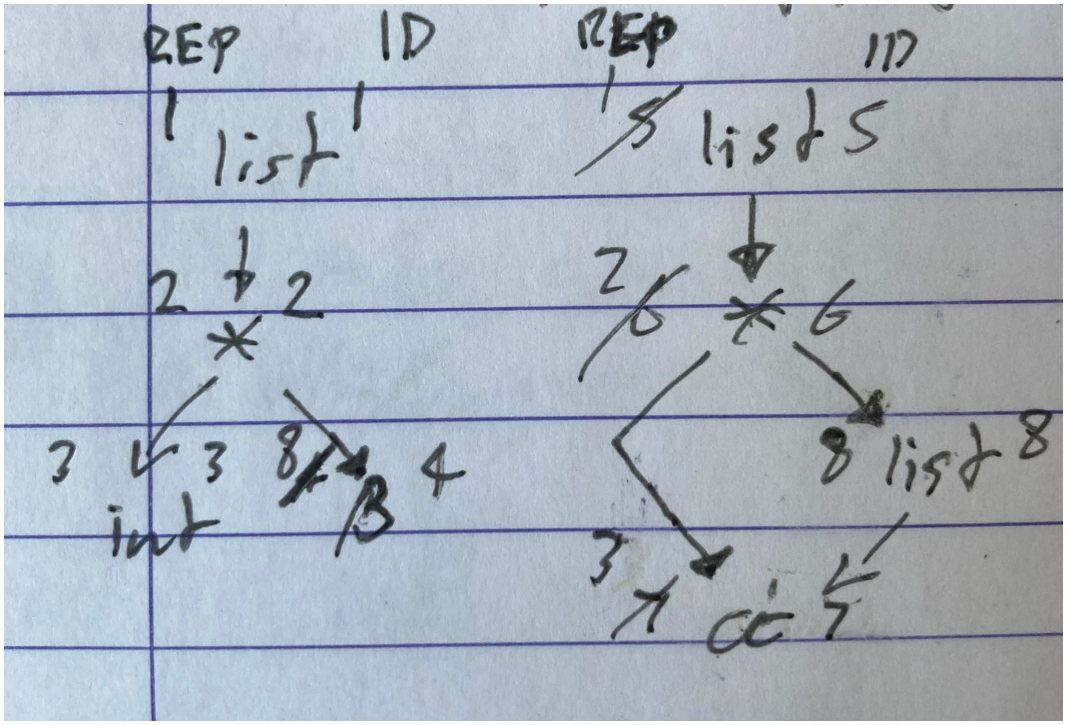
(a) Show the graph representation with the node identifiers (ID) and the final representatives (REP) after applying the most-general unification (MGU) algorithm[1].

(b) What are the type expressions of α and β after most-general unification?

(c) What is the unified type expression?

3.2.1 a

Graph applying MGU algorithm:



alpha = find(7) = 3 = int
beta = find(4) = 8 = list(alpha)

3.2.2 b

The type variables: alpha = int, beta = list(int).

3.2.3 c

The unified type expression : list(int*list(int))

4 Task 4

4.1 4.1

4 Code generation

Question 4.1 (7 pts) For the source and intermediate languages in the textbook's Chapter 6, generate intermediate code for the following source-code fragment:

```
x := 10;
while (!x) < 1 do
  x := x - a[x]
```

Assume that the symbol table maps `x` to r_x, and `a` to r_a, which contains the address of the first element of the array `a`. Remember that, like for C, no array-bounds checks are performed.

Use the translation schema for assignment statements from the book (*not* the slides). You don't *need* to reproduce the exact order in which the labels and temporary variables are generated, but we recommend that you still try to do so, to help ensure that you are otherwise following the translation specification exactly.

```
    t0 := 10
    rx := t0
Label L0
    t1 := 0
    t3 := rx
    if t3 := 0 then l4 else l3
Label L3
    t1 := 1
Label L4
    t2 := 1
    if t1 < t2 then l1 else l2
Label L1
    t5 := rx
    t7 := rx
    t7 := t7 * 4
    t7 := t7 + ra
    t6 := M[t7]
    t4 := t5 - t6
    rx := t4
    GOTO L0
Label L2
```

4.2 4.2

Question 4.2 (5 pts) This task refers to machine-code generation by the greedy technique that pattern matches the longest available intermediate-language (IL) pattern. Using the intermediate, three-address language (IL) from the textbook (slides) and the IL *patterns*:

t := r_s + k M[t^{last}] := r_t	sw r_t, k(r_s)
M[r_s] := r_t	sw r_t, 0(r_s)
r_d := r_s - r_t	sub r_d, r_s, r_t
r_d := r_s + k	addi r_d, r_s, k
IF r_s < r_t THEN lab_t ELSE lab_f LABEL lab_t	slt R1, r_s, r_t beq R1, R0, lab_f lab_t:
IF r_s < r_t THEN lab_t ELSE lab_f	slt R1, r_s, r_t bne R1, R0, lab_t j lab_f
LABEL lab	lab:

[1]Section"Advanced Type Checking" of type-checking slides; Sect. 6.5 of "Compilers: Principles, Techniques, and Tools", 2ed.

Translate the IL code below to MIPS code. (You may use directly, a, x, y, z as symbolic registers, or alternatively, use r_a, r_x, r_y, r_z, respectively.)

x := z^{last} - y
z := a^{last} + 5
M[z] := x
IF y < z THEN lab_1 ELSE lab_2
LABEL lab_3

```
sub x,z,y
addi z,a,5
sw x, 0(z)
slt R1,y,z
bne R1,R0,label1
j label2
label3:
```

4.3 4.3

Question 4.3 (8 pts) This question concerns code generation in the FASTO compiler. Consider an extension of FASTO with an additional expression form:

$$Exp \rightarrow \texttt{least ID >= } Exp \texttt{ satisfying } Exp$$

The intended semantics is that, in an expression

$$\texttt{least } name \texttt{ >= } Exp_1 \texttt{ satisfying } Exp_2\,,$$

Exp_1 should evaluate to some integer n. Then the `least`-construct repeatedly evaluates Exp_2 (which should return a boolean) with *name* locally bound to n, $n+1$, $n+2$, ..., until Exp_2 returns `true`, at which point the successful value for *name* is returned. (Any other uses of *name* outside of Exp_2 are not affected.) If Exp_2 never becomes true, the expression runs forever. Any errors arising during evaluation of Exp_1 or Exp_2 abort execution as usual.

For example, the FASTO expression

```
least x >= 0 satisfying x*x == 100
```

should evaluate to 10. Had we instead started from, e.g., `12`, not `0`, the program would run forever. On the other hand, had we started from, say, `~20`, the result would have been -10.

In the `AbSyn` module, we extend the type of abstract expressions correspondingly:

```
type Exp<'T> =
...
  | Least of string * Exp<'T> * Exp<'T> * Position
```

Assuming that parsing, type checking, etc. for `Least` have been taken care of, your task is to generate MIPS code for it, according to the above specification. That is, you should complete the following case in the code generator:

```
let rec compileExp (e: Exp<Type>)
                   (vtable: SymTab<Mips.reg>)
                   (place: Mips.reg) : Mips.Instruction list =
  match e with
  ...
  | Least (name, e1, e2, pos) ->
      (* your code would go here *)
```

Try to stick roughly to F# syntax, as used in the compiler, *not* the pseudocode language from the book.

```
let rn = newReg "least_n"
let rb = newReg "least_b"
let ltry = newLab "least_try"
let ldone = newLab "least_done"
let code1 = compileExp e1 vtable rn          // compile e1
let vtable' = SymTab.bind name rn vtable     // bind name to rn
let code2 = compileExp e2 vtable' rb         // compile e2 with new vtable

code1 @                                      // eval e1, result in rn
```

```
[Mips.LABEL ltry] @                    // loop start: try current value in rn
code2 @                                // eval e2, result in rb
[Mips.BNE (rb, RZ, ldone);             // if rb is true, exit loop
Mips.ADDI (rn, rn, 1);                 // rb was false, increment rn
Mips.J ltry;                           // and try again
Mips.LABEL ldone;                      // loop exit: solution found in rn
Mips.MOVE (place,rn)]                  // copy it to final destination
```

5 Task 5

5.1 5.1

5 Liveness analysis, register allocation, and optimization

Question 5.1 (5 pts) This task refers to liveness analysis. Suppose that, after a liveness analysis, instructions i and j have the following in/out sets:

$in[i] = \{\mathtt{a}, \mathtt{b}, \mathtt{d}\}$

i : ?

$out[i] = \{\mathtt{a}, \mathtt{b}, \mathtt{d}\}$

a) This means that instruction i can be:

(A) `a := b - d`

(B) `GOTO abc`

(C) `IF d < e THEN body ELSE end`

(D) `a := c + 5`

(E) `M[5] := b`

$in[j] = \{\mathtt{c}, \mathtt{d}\}$

j : ?

$out[j] = \{\mathtt{a}, \mathtt{d}\}$

b) This means that instruction j can be:

(A) `M[a] := c`

(B) `RETURN a`

(C) `a := c`

(D) `IF c = d THEN next ELSE end`

(E) `LABEL correct`

Justify why your choice for instruction i and j satisfies the corresponding dataflow equation (not more than 5 lines for each choice). You need *not* justify why other choices for i and j are incorrect. **Note:** Multiple choices may be correct in (a) and (b).

5.1.1 a

B (Kill = Ø, Gen = Ø) and E (Kill = Ø, Gen = b)

5.1.2 b

C (Kill = a, Gen = c)

5.2 5.2

Question 5.2 (12 pts) This task refers to liveness analysis and register allocation. Given the following program:

```
   F(x, y) {
 1:   a := x * y
 2:   b := x
 3:   LABEL loop
 4:   b := b + y
 5:   IF a < b THEN do ELSE end
 6:   LABEL do
 7:   y := b
 8:   b := y + y
 9:   GOTO loop
10:   LABEL end
11:   RETURN b
    }
```

1 Show **succ**, **gen** and **kill** sets for instructions 1–11.

2 Compute **in** and **out** sets for every instruction; stop after two iterations.

3 Show the interference table (Stmt | Kill | Interferes With).

4 Draw the interference graph for `a`, `b`, `x`, and `y`.

5 Color the interference graph with 3 colors; show the stack, i.e., the three-column table: Node | Neighbors | Color.

6 Explain briefly the main actions that are taken when a register is marked 'spill' (max. 6 lines). *Hint:* This task is independent of tasks (1-5).

5.2.1 a

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2	x,y	a
2	3	x	b
3	4		
4	5	b,y	b
5	6,10	a,b	
6	7		
7	8	b	y
8	9	y	b
9	3		
10	11		
11	-	b	

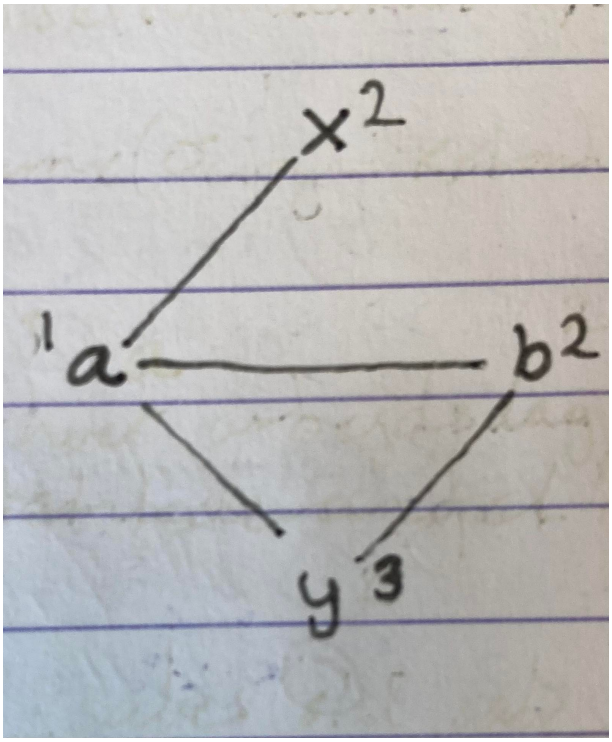
5.2.2 b

i	$out[i]_0$	$in[i]_0$	$out[i]_1$	$in[i]_1$	$out[i]_2$	$in[i]_2$
1			a,x,y	x,y	a,x,y	x,y
2			a,b,y	a,x,y	a,b,y	a,x,y
3			a,b,y	a,b,y	a,b,y	a,b,y
4			a,b	a,b,y	a,b	a,b,y
5			b	a,b	a,b	a,b
6			b	b	a,b	a,b
7			y	b	a,y	a,b
8				y	a,b,y	a,y
9					a,b,y	a,b,y
10			b	b	b	b
11				b		b

5.2.3 c

Instruction i	kill[i]	out[i]	Interferes with
1	a	a,x,y	x,y
2	b	a,b,y	a,y
4	b	a,b	a
7	y	a,y	a
8	b	a,b,y	a,y

5.2.4 d



5.2.5 e

Node	Neighbors	Colour
a	-	1
b	a	2
y	a,b	3
x	a	2

5.2.6 f

Three main actions are taken:
(1) Spill code is inserted in the program for each variable marked 'spill', so it can be kept in memory M. After this rewrite of the program, (2) liveness analysis and (3) register allocation are performed again. A subsequent register allocation may generate new spilled variables, and the steps are repeated.

5.3 5.3

Question 5.3 (7 pts) This question concerns dead-binding elimination (DBE) in FASTO, as seen in Lecture 6 and the project.

For each of the two FASTO expressions (a) and (b) below, with numbered subexpressions, work out the result of performing DBE on each subexpression. Show both the set of variables ultimately used in the optimized subexpression and the optimized subexpression itself. For subexpressions that are `let`s, also say explicitly whether that `let`-biding can be eliminated or not.

Preferably use the same format for your results as in Task 1 of Weekly 4. However, since none of the expressions evidently contain any I/O, you should just omit the "I/O?" column from the tables.

a. `(5)let y = (3)(let x = (1)y+1 in (2)y*y) in (4)2*x+z`

b. `(5)let x = (1)y+1 in (4)(let y = (2)y*y in (3)2*x+z)`

c. Note that the input expression in (b) is the result of performing the let-flattening transformation from copy/constant propagation on (a), even though no actual propagations are possible in this case. Comment on whether you get the same (or an equivalent) final result for the two expressions in (a) and (b), and briefly (3–5 lines) explain why/why not.

5.3.1 a

SubExp	Elim	UsedVars	OptSubExp
(1)	-	{y}	y+1
(2)	-	{y}	y*y
(3)	yes	{y}	y*y
(4)	-	{x,z}	2*x+z
(5)	yes	{x,z}	2*x+z

5.3.2 b

SubExp	Elim	UsedVars	OptSubExp
(1)	-	{y}	y+1
(2)	-	{y}	y*y
(3)	-	{x,z}	2*x+z
(4)	yes	{x,z}	2*x+z
(5)	no	{y,z}	let x = y+1 in 2*x+z

5.3.3 c

The results are different, ant not equivalent. This is not unexpected, because the transformation:
let y = (let x = E in F) in G $\rightarrow$ let x = E in let y = F in G
is only safe when x does not occur free in G; Otherwise, the binding of x to E on the RHS will shadow any previous binding for x. Evidently, this condition is violated in the example.