1

a)

A → A or B | B

B → B and C | C

C → not C | D

D → ‘(‘ A ‘)‘ | true | false

Alternative:

Expr -> Conj {or Conj}

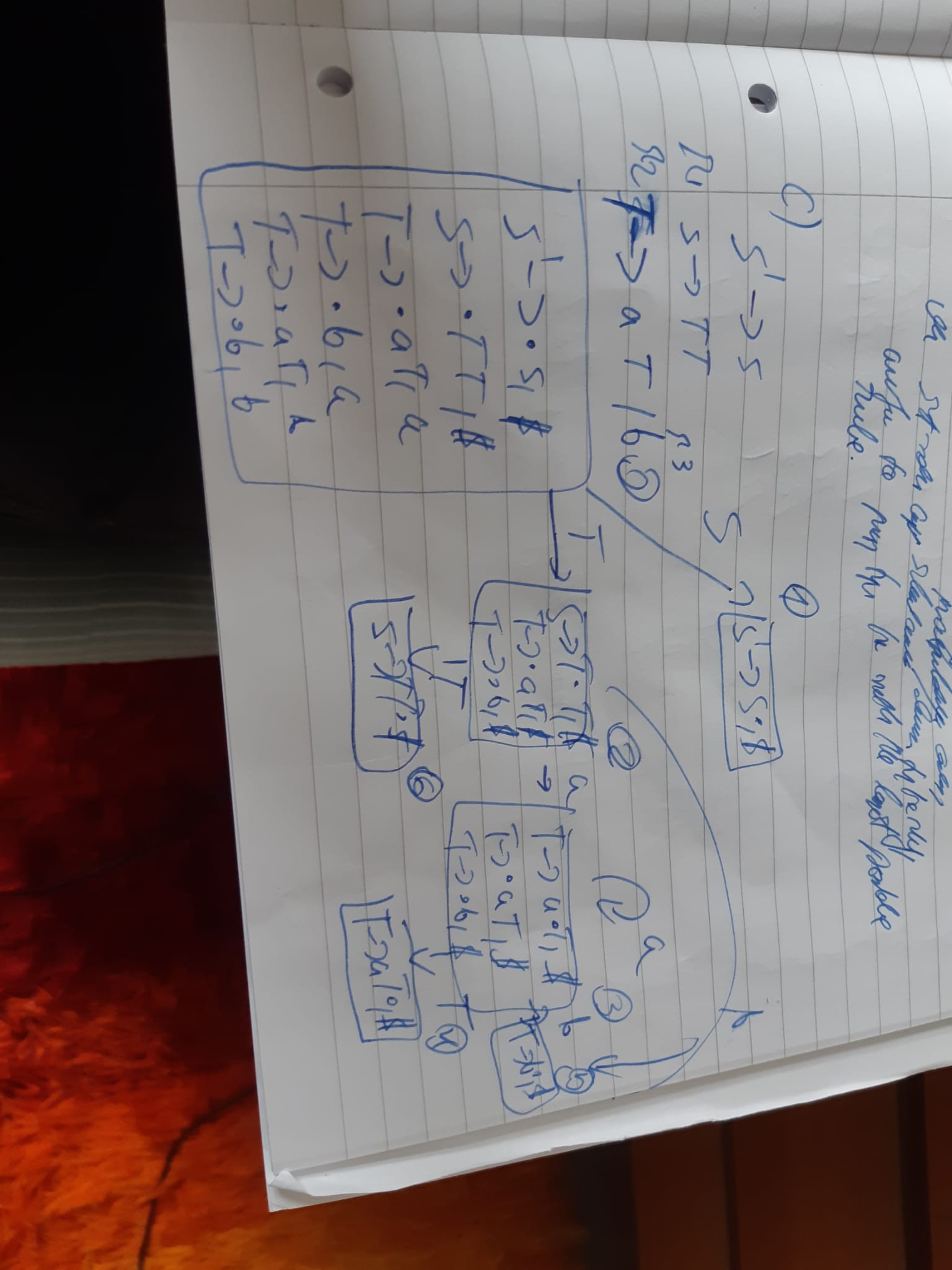
Conj -> Atom {and Atom}

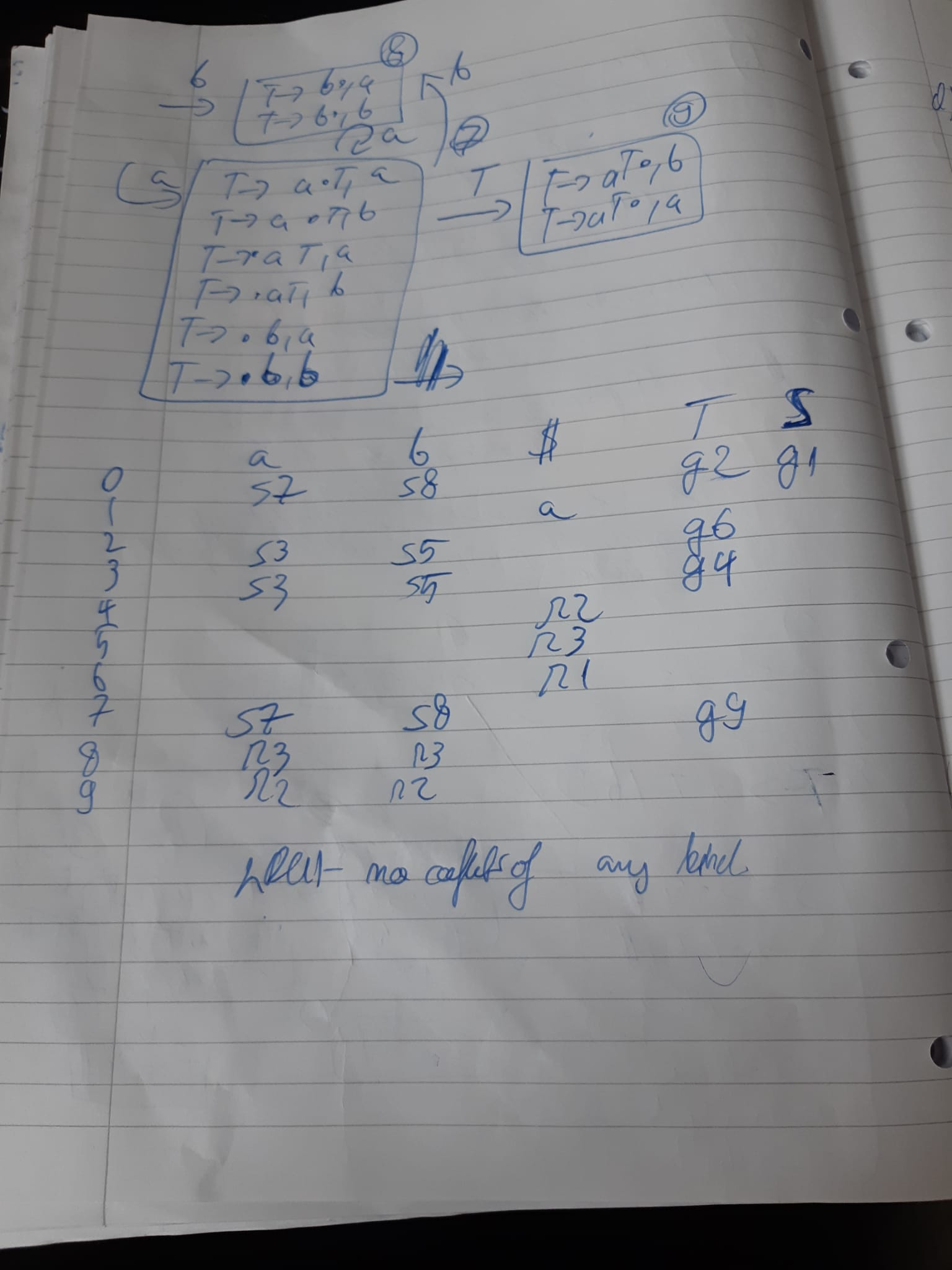
Atom -> not Atom | true | false | ( Expr )

Assumes and, or are left associative

b) Checkout his slides

c)





I have very ugly writing sorry

d)

From Wikipedia:

Advantages:

it is a non-trivial test of the language being compiled, and as such is a form of dogfooding.

compiler developers and bug reporting part of the community only need to know the language being compiled.

compiler development can be performed in the higher-level language being compiled.

improvements to the compiler's back-end improve not only general-purpose programs but also the compiler itself.

it is a comprehensive consistency check as it should be able to reproduce its own object code.

Disadvantages:

massive hassle

The languages may just not be well suited to compiler writing due to their fundamental nature

2 a)

i)

LiveIn(14) = uses(14) U (liveOut(14)- defs(14))

= {T4}

ii) 1 in kill(10), therefore 11, 12 are never reached by definition in line 1

iii) T4 is an induction variable, T3 equals T4 so is also an induction variable. T1 is not (as it is an intermediary value for the multiply instruction). T2 is not as constant. T0 is not as the relationship to T4 is non linear.

b)

i)

Trick is to define transBExp as taking a label which it jumps to.

transBExp bexp elseLabel regs

++ flatMap (flip transStat regs) ifs

++ [Bra finishLabel, Define elseLabel]

++ flatMap (flip transStat regs) elses

++ [Define finishLabel]

transBExp (lessThan e1 e2) label (r1:r2:rs) =

transAExp e1 r1:r2:rs

++ transAExp e2 r2:rs

++ [Cmp (Reg r2) (Reg r1) ] // If I remember correctly its reversed for r1 op r2.

++ [Bge label]

ii)

Define auxiliary function log which outputs a [Instructions]. Then put that log in at top of Assign, If, TransBExp. It needs to save all overwritten values and then note down jumps (so we can reverse the control flow graph).

I think the solution is a bit more intricate than above (^), because you don’t actually know the value of the condition in transBExp, or even at the top of the if-statement. We don’t have a way to ‘access’ the boolean. I’ve tried the following, although I don’t think it stores ‘enough’ information, but I’m not really sure how else to do it (e.g. how would you actually ‘note down jumps’ like the answer above suggests?)

transStat :: Stat -> [Register] -> [Instr]

transStat (Assign n e) dst:rs

= transAExp e dst:rs ++

[Mov (Abs n) (Ind (Reg A0))] ++ -- Store the value

[Add (ImmNum 4) (Reg A0)] ++ -- Move pointer along

[Mov (Reg dst) (Abs n)]

transStat (If c tss fss) rss@(dst:nxt:rs)

= transBExp c rss falseLabel ++

[Mov (ImmNum 1) (Ind (Reg A0))] ++ -- Store 1 for true

concatMap (\s -> transStat s rss) tss ++

[Bra endLabel] ++

[Define falseLabel] ++

[Mov (ImmNum 0) (Ind (Reg A0))] ++ -- Store 0 for false

concatMap (\s -> transStat s rss) fss ++

[Define endLabel] ++

[Add (ImmNum 4) (Reg A0)] ++ -- Move pointer along

where

endLabel = newlabel

falseLabel = newlabel

transBExp :: BExp -> [Register] -> String -> [Instr]

transBExp (LessThan e e’) rss@(dst:nxt:rs) label

= transExp e rss ++

transExp e’ nxt:rs ++

[Cmp (Reg dst) (Reg nxt)] ++

[Bge label]

Assumptions:

* Memory is byte addressable
* Integers stored using 4 bytes
* True condition stored as 1, false as 0