1 Problem 1

My μPascal code to find $\sqrt{2 * 10^8}$:

```
begin
    v := 200000000;
    left := 0;
    right := 46340;
    (* i.e. largest # whose square fits on 32 bits *)

    (* Invariant: left <= right and left <= floor(sqrt(v)) < right *)
    while right - left > 1 do
        mid := (left + right) div 2;
        if mid * mid > v then
            right := mid
        else
            left := mid
        end
    end;

    print left ; newline
end.
```

The corresponding Keiko code:

`MODULE Main 0 0`	Declares the module Main
`IMPORT Lib 0`	Imports the library Lib 0
`ENDHDR`	Marks the end of the Keiko header
`PROC MAIN 0 0 0`	Declares the beginning of procedure MAIN
`! v := 200000000;`	
`CONST 200000000`	Pushes `2e8`
`STGW _v`	Assigns `2e8` to `v`
`! left := 0;`	
`CONST 0`	Pushes `0`
`STGW _left`	Assigns `0` to `left`
`! right := 46340;`	
`CONST 46340`	Pushes `46340`
`STGW _right`	Assigns `46340` to `right`
`JUMP L2`	Marks the beginning of a `while`-loop body
`LABEL L1`	Runs the contents of the `while`-loop again
`! mid := (left + right) div 2;`	
`LDGW _left`	Pushes `left`
`LDGW _right`	Pushes `right`
`PLUS`	Calculates `left + right`
`CONST 2`	Pushes `2`
`DIV`	Calculates `(left + right) / 2`
`STGW _mid`	And stores it in `mid`
`! if mid * mid > v then`	
`LDGW _mid`	Pushes `mid`
`LDGW _mid`	Pushes `mid`
`TIMES`	We now calculate `mid * mid`
`LDGW _v`	Pushes `v`

`JGT L4`	Start of an `if`, that branches on `mid * mid > v`
`JUMP L5`	Otherwise we go to `L4`
`LABEL L4`	The `then` clause of the `if` statement
`!  right := mid`	
`LDGW _mid`	Pushes `mid`
`STGW _right`	Saves the top of the stack to `right`
`JUMP L6`	Jumps over the `else` part of the `if` statement
`LABEL L5`	The `else` clause of the `if` statement
`!  left := mid`	
`LDGW _mid`	Pushes `mid`
`STGW _left`	Saves the top of the stack to `left`
`LABEL L6`	Marks the end of the `if` statement
`LABEL L2`	Marks the end of a `while`-loop
`!  while right - left > 1 do`	We now try to check if we contine the loop
`LDGW _right`	Pushes `right`
`LDGW _left`	Pushes `left`
`MINUS`	Calculates `right - left`
`CONST 1`	Pushes `1`
`JGT L1`	If `right - left > 1` we continue the `while`
`JUMP L3`	This jumps to after the while if we stop executing it
`LABEL L3`	And this is where we jump to
`!  print left ; newline`	
`LDGW _left`	Pushes `left`
`CONST 0`	Pushes `0`; means that the function is statically linked
`GLOBAL lib.print`	Pushes the global address lib.print to the stack
`PCALL 1`	Calls `Lib.Print` with 1 argument, to print `left`
`CONST 0`	Pushes `0`
`GLOBAL lib.newline`	Pushes the global address lib.newline to the stack
`PCALL 0`	Prints a newline
`RETURN`	Exits the procedure `MAIN`
`END`	Terminates the program
`GLOVAR _mid 4`	Declarations for global variables
`GLOVAR _v 4`	
`GLOVAR _left 4`	
`GLOVAR _right 4`	

2 Problem 2

2.1 a

To do this, simply use the following sequence of instructions:

Instruction	Stack after instruction
`CONST 1`	1
`LDGW _x`	1, x
`PLUS`	1+x
`CONST 1`	1+x, 1
`SWAP`	1, 1+x
`DIV`	1/(1+x)

2.2 b

Claim: Supposing that:

$$cost(x) = \text{optimal depth for some expression } x$$

Then:

$$cost(Binop(w, e_1, e_2)) = min\{max\{cost(e1), cost(e2) + 1\}, max\{cost(e1) + 1, cost(e2)\}\}$$

Proof: Note that in order to evaluate $Binop(w, e_1, e_2)$, we must first evaluate e_1 and e_2 in some order (we can do either first because of the `SWAP` instructions). Note that if we evaluate, say, e_1 first, then evaluating e_2 will lead to a stack height one larger for e_2 than it would otherwise (due to the result of evaluating e_1). So, evaluating e_1 first leads to a stack depth of $max\{cost(e1), cost(e2) + 1\}$. Symmetrically, evaluating e_2 first leads to a stack of depth $max\{cost(e1) + 1, cost(e2)\}$. Taking the smaller of these two leads us to the claim. Note that I ignore the possibility of using the information in one expression to calculate the other. ■

Also, this is an `Ocaml` definition of the function above:

```
let rec cost = function
      Variable str      -> 1
    | Constant v        -> 1
    | Monop (w, y)      -> cost y
    | Binop (w, e1, e2) -> min
        (max (cost e1) (1 + cost e2))
        (max (1 + cost e1) (cost e2))
```

Claim: $operands(e) < 2^N \Rightarrow cost(e) \leq N$, where $operands(e) =$ the number of operands in e

Proof: I prove this claim by induction on $|e|$, for all N, where $|e| =$ the number of operands and operators in e:

BC: Assume $|e| = 1$. Assume also that $2^N > operands(e)$. Then $2^N > operands(e) \geq |e| = 1$, so $N > 0$. Also, since $|e| = 1$, e is either `Variable` or a `Constant`, so $cost(e) = 1 = 2^0 < 2^N$.

IS: Supposing that the claim is true for all expressions f with $|f| < n$ (where $n > 1$), then, for some expression e with $|e| = n$, we have several cases:

- if e is a `Variable` or `Constant`, $operands(e) < 2^N \Rightarrow cost(e) \leq N$ is vacuously true, since no `Variable` or `Constant` has more than 1 operands or operators, yet $|e| = n > 1$.

- if $e =$ `Monop`(w, f), then by definition, $operands(e) = operands(f)$ and $cost(e) = cost(f)$. Since, by the inductive hypothesis, $operands(f) < 2^N \Rightarrow cost(f) \leq N$, then using the previous identities we get that: $operands(e) < 2^N \Rightarrow cost(e) \leq N$.

- if $e =$ `Binop`(w, f, g). Assume $operands(e) < 2^N$ for some N. Let M and K be the unique integers that satisfy $2^{M-1} \leq operands(f) < 2^M$ and $2^{K-1} \leq operands(g) < 2^K$. W.l.o.g. let $M \leq K$. Assume for contradiction that $N \leq M$. Then

$$\begin{aligned}
operands(e) &= 1 + operands(f) + operands(g) && \text{(by definition of } operands\text{)}\\
&\geq 1 + 2^{M-1} + 2^{K-1} && \text{(definition } M, K\text{)}\\
&\geq 1 + 2^{N-1} + 2^{N-1} && \text{(assumption and transitivity)}\\
&> 2^N
\end{aligned}$$

  a contradiction. So $N > M$. Assume for contradiction that $N < K$. Then

$$\begin{aligned}
operands(e) &= 1 + operands(f) + operands(g) && \text{(as before)}\\
&\geq 1 + 2^{M-1} + 2^{K-1} && \text{(as before)}\\
&\geq 1 + 0 + 2^{(N+1)-1} && (N < K \Rightarrow K \geq N + 1))\\
&> 2^N
\end{aligned}$$

  a contradiction. So $N \geq K$. Use the inductive hypothesis on f, M and g, K to get that $cost(f) \leq M$ and $cost(g) \leq K$. Now

$$\begin{aligned}
cost(e) &= min\{max\{cost(f), 1 + cost(g)\}, max\{1 + cost(f), 1 + cost(g)\}\}\\
&\leq min\{max\{M, 1 + K\}, max\{1 + M, K\}\}\\
&\leq min\{max\{N - 1, N + 1\}, \{1 + N - 1, N\}\}\\
&\leq N
\end{aligned}$$

So, as the base case *BC* holds, and the inductive step *IS* works in all cases, the claim is true ■

2.3 c

```
let rec gen_expr =
    function
      Constant x ->
        CONST x
    | Variable x ->
        SEQ [LINE x.x_line; LDGW x.x_lab]
    | Monop (w, e1) ->
        SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) ->
        if cost e1 >= cost e2 then
            SEQ [gen_expr e1; gen_expr e2; BINOP w]
        else
            SEQ [gen_expr e2; gen_expr e1; SWAP; BINOP w]
```

3 Problem 4

3.1 a

Some code with nested `if`'s.

```
begin
    if i then
        if i then
            i := 1
        else
            i := 2
        end
    else
        if i then
            i := 3
        else
            i := 4
        end
    end
end.
```

With it's associated `Keiko` code.

`MODULE Main 0 0`	
`IMPORT Lib 0`	
`ENDHDR`	
`PROC MAIN 0 0 0`	
`! if i then`	
`LDGW _i`	
`CONST 0`	
`JNEQ L1`	This row and the 2 below are very inefficient
`JUMP L2`	
`LABEL L1`	
`! if i then`	
`LDGW _i`	
`CONST 0`	
`JNEQ L7`	This row and the 2 below are very inefficient
`JUMP L8`	
`LABEL L7`	
`! i := 1`	
`CONST 1`	
`STGW _i`	
`JUMP L9`	
`LABEL L8`	
`! i := 2`	
`CONST 2`	
`STGW _i`	
`LABEL L9`	
`JUMP L3`	
`LABEL L2`	
`! if i then`	
`LDGW _i`	
`CONST 0`	

`JNEQ L4`	This row and the 2 below very inefficient
`JUMP L5`	
`LABEL L4`	
`! i := 3`	
`CONST 3`	
`STGW _i`	
`JUMP L6`	
`LABEL L5`	
`! i := 4`	
`CONST 4`	
`STGW _i`	
`LABEL L6`	These two labels could be combined
`LABEL L3`	
`RETURN`	
`END`	

3.2 b

One peephole optimiser rule that would fix the first three inefficiencies would be used would be to transform the sequence `JNEQ a; JUMP b; LABEL a` into `JEQ b`, with similar rules for all conditional jumps.
One rules that would fix the next last inneficiency would be to transform `LABEL a; LABEL b` into `LABEL a`, substituting `b` with `a` throughout the code.

3.3 c

I notice that the first issue comes from the way that we evaluate conditionals – more precisely, rather than always inserting jumps for "true" and "false" values, it would be more efficient to make the compiler either emit a jump or simulate "fall through" in certain cases.
The second issue comes from the gen_expr not having enough context to know if there is label to be place immediately after the statement we generate, and therefore sometimes places useless `LABEL`'s. Passing such labels to gen_expr fixes this.
The code generated by these approaches is:

`MODULE Main 0 0`
`IMPORT Lib 0`
`ENDHDR`
`PROC MAIN 0 0 0`
`! if i then`
`LDGW _i`
`CONST 0`
`JEQ L1`
`! if i then`
`LDGW _i`
`CONST 0`
`JEQ L4`
`! i := 1`
`CONST 1`
`STGW _i`
`JUMP L2`
`LABEL L4`
`! i := 2`

```
CONST 2
STGW _i
JUMP L2
LABEL L1
!  if i then
LDGW _i
CONST 0
JEQ L3
!  i := 3
CONST 3
STGW _i
JUMP L2
LABEL L3
!  i := 4
CONST 4
STGW _i
LABEL L2
RETURN
END
GLOVAR _i 4
```

Which is acceptable.

The code that generates this is:

```
open Tree
open Keiko

let optflag = ref false

(* |gen_expr| -- generate code for an expression *)
let rec gen_expr =
    function
        Constant x ->
            CONST x
    | Variable x ->
            SEQ [LINE x.x_line; LDGW x.x_lab]
    | Monop (w, e1) ->
            SEQ [gen_expr e1; MONOP w]
    | Binop (w, e1, e2) ->
            SEQ [gen_expr e1; gen_expr e2; BINOP w]

let logical_opposite = function
    Eq -> Neq
    | Neq -> Eq
    | Lt -> Geq
    | Geq -> Lt
    | Gt -> Leq
    | Leq -> Gt

(* gen_cond e tlab flab will generate a conditional on expression e
```

```
 * If e is true and tlab is of the form Some t, we will jump to t.
 * If e is false and flab is of the form Some f, we jump to f.
 * If e is true and tlab is of the form None, we fall through.
 * If e is false and flab is of the form None, we fall through. *)
let rec gen_cond e tlab flab =
    match (e, tlab, flab) with
        (_, None, None) -> NOP
      | (Constant x, Some t, None) -> if x <> 0 then JUMP t else NOP
      | (Constant x, None, Some f) -> if x <> 0 then NOP else JUMP f
      | (Constant x, Some t, Some f) -> if x <> 0 then JUMP t else  JUMP f
      | (Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2), Some t, None) ->
            SEQ [gen_expr e1; gen_expr e2; JUMPC (w, t) ]
      | (Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2), None, Some f) ->
            SEQ [gen_expr e1; gen_expr e2; JUMPC (logical_opposite w, f) ]
      | (Binop ((Eq|Neq|Lt|Gt|Leq|Geq) as w, e1, e2), Some t, Some f) ->
            SEQ [gen_expr e1; gen_expr e2; JUMPC (w, t); JUMP f ]
      | (Monop (Not, e1), _, _)->
            gen_cond e1 flab tlab
      | (Binop (And, e1, e2), Some t, None) ->
            let lab1 = label () in
            SEQ [gen_cond e1 None (Some lab1);
                gen_cond e2 (Some t) None; LABEL lab1]
      | (Binop (And, e1, e2), None, Some f) ->
            SEQ [gen_cond e1 None (Some f); gen_cond e2 None (Some f)]
      | (Binop (And, e1, e2), Some t, Some f)->
            let lab1 = label () in
            SEQ [gen_cond e1 (Some lab1) (Some f);
                LABEL lab1; gen_cond e2 (Some t) (Some f)]
      | (Binop (Or, e1, e2), Some t, None) ->
            let lab1 = label () in
            SEQ [gen_cond e1 (Some lab1) None;
                gen_cond e2 None (Some t); LABEL lab1]
      | (Binop (Or, e1, e2), None, Some f) ->
            SEQ [gen_cond e1 (Some f) None; gen_cond e2 (Some f) None]
      | (Binop (Or, e1, e2), Some t, Some f) ->
            let lab1 = label () in
            SEQ [gen_cond e1 (Some t) (Some lab1);
                LABEL lab1; gen_cond e2 (Some t) (Some f)]
      | (_, Some t, None) -> SEQ [gen_expr e; CONST 0; JUMPC (Neq, t); ]
      | (_, None, Some f) -> SEQ [gen_expr e; CONST 0; JUMPC (Eq, f); ]
      | (_, Some t, Some f) ->
            SEQ [gen_expr e; CONST 0; JUMPC (Neq, t); JUMP f]


(* gen_stmt final_lab s will generate code for a statement s
 * If final_lab is of the form Some lab, we will assume that
 * immediately following the code generated by s there exists
 * some label lab.
 * Otherwise, we have no such assumption *)
let rec gen_stmt final_lab s =
    match (s, final_lab)with
```

```
    (Skip, _) -> NOP
  | (Seq stmts, _) -> SEQ ((List.map (gen_stmt None)
      (List.rev (List.tl (List.rev stmts))))
      @ [gen_stmt final_lab (List.hd (List.rev stmts))])
  | (Assign (v, e), _)->
        SEQ [LINE v.x_line; gen_expr e; STGW v.x_lab]
  | (Print e, _)->
        SEQ [gen_expr e; CONST 0; GLOBAL "lib.print"; PCALL 1]
  | (Newline, _)->
        SEQ [CONST 0; GLOBAL "lib.newline"; PCALL 0]
  | (IfStmt (test, thenpt, elsept), Some lab)->
        let lab1 = label () in
        SEQ [gen_cond test None (Some lab1);
            gen_stmt (Some lab) thenpt ; JUMP lab;
            LABEL lab1; gen_stmt (Some lab) elsept ]
  | (IfStmt (test, thenpt, elsept), None) ->
        let lab1 = label () and lab2 = label () in
        SEQ [gen_cond test (None) (Some lab1);
            gen_stmt (Some lab2) thenpt ; JUMP lab2;
            LABEL lab1; gen_stmt (Some lab2) elsept ; LABEL lab2]
  | (WhileStmt (test, body), _)->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [JUMP lab2; LABEL lab1; gen_stmt (Some lab2) (body) ;
      LABEL lab2; gen_cond test (Some lab1) (Some lab3); LABEL lab3]

(* |translate| -- generate code for the whole program *)
let translate (Program ss) =
  let code = gen_stmt None ss in
  Keiko.output (if !optflag then Peepopt.optimise code else code)
```

4 Problem 7

4.1 a

An abstract syntax tree type that would work for the first construct is:

```
type expr = (* a type for expressions *)

type stmt =
    WhileStmt of (expr, stmt) list
    | (* all other constructs in the language, including a Seq statement *)
```

One for the second construct is:

```
type stmt =
    LoopStmt of stmt
    | ExitStmt
    | (* all other constructs in the language, including a Seq statement*)
```

Production rules for the first are:

```
expr: /* recognizes a expression */

stmts:
    stmt { Seq [$1] }
    | stmts SEMICOLON stmt { Seq [$1, $3] }

elseif_list:
    END { [] }
    | ELSEIF expr DO stmts elseif_list { ($2, $4) :: $5 }

stmt:
    WHILE expr DO stmts elseif_list { WhileStmt (($2, $4) :: $5) }
    | /* ... other control structures and statements ... */
```

and for the second are:

```
expr: /* recognizes a epxression */

stmts:
    stmt { Seq [$1] }
    | stmts SEMICOLON stmt { Seq [$1, $3] }

stmt:
    LOOP stmts END { LoopStmt($2) }
    | EXIT { ExitStmt }
    | /* ... Many other control structures ... */
```

4.2 b

I assume that we have syntax simmilar to Keiko for our machine, and a `Ocaml` type that represents this code. Then, this generates the while:

```
gen_cond c tlab flab = (* generates code that, if c evaluates to true,
                        * jumps to tlab, and otherwise jumps to flab *)
```

```
gen_stmt = function
    WhileStmt ls ->
        let first_lab = label ()
        and make_branch (expr, ast)
            = let lab1 = label () and lab2 = label () in
                SEQ [ gen_cond lab1 lab2 ; LABEL lab1
                    ; get_stmt ast ; JUMP first_lab
                    ; LABEL lab2 ]
        and branches = SEQ (List.map make_branch ls)
        in  SEQ [ LABEL first_lab ; branches ]
    | (* all other language constructs *)
```

And this generates the loop:

```
gen_stmt where
        (* where is an option that might contain
         * a label placed after our current loop

         * and contains nothing otherwise and contains nothing otherwise *)
    = function
     ExitStmt -> (match where with
        None -> failwith "Exit not within a loop"
        | Some lab -> JUMP lab)
    | LoopStmt a ->
        let lab1 = label () and lab2 = label () in
        SEQ [ LABEL lab1 ; gen_stmt (Some lab2) a
            ; JUMP lab1 ; LABEL lab2 ]
    | (* all other language constructs *)
```

4.3 c

Code for the first variant:

LABEL L1	
LDLW -4	
LDLW -8	
JGT L2	if x > y go to first branc
JUMP L3	jump to second branch
LABEL L2	first branch
LDLW -4	
LDLW -8	
MINUS	
STLW -4	
JUMP L1	end of first branch
LABEL L3	beginning of the second branch condition
LDLW -4	
LDLW -8	
JLT L4	if x < y jump to second branch
JUMP L5	jump to end
LABEL L4	second branch
LDLW -8	

`LDLW -4`	
`MINUS`	
`STLW -8`	
`JUMP L1`	jump back to beginning
`LABEL L5`	

`LABEL L1`	
`LDLW -4`	
`LDLW -8`	
`JGT L3`	if x > y go to then part
`JUMP L5`	go to elseif test
`LABEL L3`	then part
`LDLW -4`	
`LDLW -8`	
`MINUS`	
`STLW -4`	
`JUMP L4`	jump to end of if-elseif-else
`LABEL L5`	elseif test
`LDLW -4`	
`LDLW -8`	
`JLT L6`	if x < y jump to elseif part
`JUMP L7`	Jump to else part
`LABEL L6`	elseif part
`LDLW -8`	
`LDLW -4`	
`MINUS`	
`STLW -8`	
`JUMP L8`	Jump to end of elseif-else
`LABEL L7`	else part
`JUMP L2`	Exit statement
`JUMP L8`	Jump to end of elseif-else
`LABEL L8`	end of elseif-else
`LABEL L4`	end of if-elseif-else
`JUMP L1`	
`LABEL L2`	

4.4 d

Some rules that would help are:
`JUMP a; JUMP b` → `JUMP a`
`JGT a; JUMP b ; LABEL a` → `JLT b` *with similar rules for other conditional jumps, provided a appears nowhere else*
`LABEL a; LABEL b` → `LABEL a` *substituting a for b everywhere else*
`JUMP a; LABEL a` → `LABEL a`
`LABEL a` → `nothing` *supposing that a appears nowhere else*

5 Problem 8

An abstract syntax that would

```
type expr = IfExpr of expr * expr * expr
```

To add this to an Ocamlyacc grammar, take the grammar from lab 1 and replace the rules for *expr* with:

```
expr :
    IF expr THEN expr ELSE expr { IfExpr ( $2, $4, $6 ) }
    | ifless_expr               { $1 }

ifless_expr :
    simple                      { $1 }
    | ifless_expr RELOP simple { Binop ($2, $1, $3) }
```

This way of doing things makes this `if then else` construct have the highest possible precedence (i.e. `if e then e else e + if e then e else e` is interpreted as `if e then e else (e + if e then e else e)`).

5.1 b

To enhance gen_expr to deal with this, simply add the following case:

```
let rec gen_expr = (* all the previous cases *)
    | IfExpr (c, e1, e2) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
        SEQ [gen_cond c lab1 lab2 ;
            LABEL lab1 ; gen_expr e1 ; JUMP lab3 ;
            LABEL lab2 ; gen_expr e2 ; LABEL lab3 ]
```

Also, make it so that gen_expr and gen_cond can mutually recurse, as follows:

```
let rec gen_expr = (* ... *)
    and gen_cond = (* ... *)
```

5.2 c

The code generated by this is:

```
LDGW _i
CONST 0
JGEQ L4
JUMP L5
LABEL L4
LDGW _a
LDGW _i
OFFSET
LOADW
LDGW _x
JGT L1
JUMP L6
LABEL L5
```

```
CONST 0
LABEL L6
CONST 0
JNEQ L1
JUMP L2
LABEL L1
LDGW _i
CONST 1
PLUS
STGW _i
JUMP L3
LABEL L2
LABEL L3
```

Some rules that would partially fix this code are:
`JGT a; JUMP b ; LABEL a` → `JLT b` *with similar rules for other conditional jumps, assuming that a is not used elsewhere*
`LABEL a; LABEL b` → `LABEL a` *substituting a for b everywhere else*
`JUMP a; LABEL a` → `LABEL a`

Applying these leads to:

```
LDGW _i
CONST 0
JLT L5
LDGW _a
LDGW _i
OFFSET
LOADW
LDGW _x
JGT L1
JUMP L6
LABEL L5
CONST 0
LABEL L6
CONST 0
JNEQ L1
JUMP L3
LABEL L1
LDGW _i
CONST 1
PLUS
STGW _i
LABEL L3
```

This is still not quite as good as the code generated by the native and, as this code still has an annoying comparison to 0 (as a conditional on an if expression first evaluates the expression then compares the result to 0). I am not sure how to fix this.

6 Problem 3.1

```
LDGW s
LDGW q
LOADW
PLUS
STGW s
LDGW q
OFFSET 4
LOADW
STGW q
```

7 Problem 3.2

This design change has several parts that need to be treated:

- First, we need to add new tokens `LOCAL` (matching only `local`), and `IN` (matching only `in`).
- Second, we need to add statements of type `LocalDecl of decls list * stmt` to the abstract syntax. No change is needed to type `program`.
- Third, we need to add a new rule to the grammar: `stmt: LOCAL decls IN stmt END` {`LocalDecl(2,4)`}.
- Fourth, we add a new scoping check (i.e. we check that all variables are only used in blocks where they are declared).
- Now, we make an intermediate translation, with the eventual goal of transforming local statements into equivalent global declarations. First, dealing with name conflicts (i.e. redeclaring a variable in a more deeply nested block). I think the easiest and cleanest way to deal with scope is with name mangling – i.e. we have a translation step that transforms:

  ```
  local var y : t;
  in
      stmt[y]
  end
  ```

  into

  ```
  local var mangledy : t;
  in
      stmt[mangledy]
  end
  ```

  where stmt[t] is notation for a statement that contains some variable name t, stmt[u] means that same statement just with all unbound appearences of t replaced by u, and mangledy is some string that is distinct from all other variable names in the program (one such mangling scheme would be to append the number of `local` blocks already processed when the variable is declared, or to append a long random string). By unbound I mean that appearences within another `local` block of the same variable, where the variable was re-declared in that local block, do not count. Note that this transformation does not change the semantics of the program.
- After applying the previous translation, note that since all the variables now have distinct names, we could just have well as declared all of them globally. This suggests what we must now do; more precisely, move all declarations from `local` blocks into the global declaration.
- Now, as we know that variables are used only in the correct scopes, and also as the syntax tree has the same form as a syntax tree without local blocks, we can simply use the old checking, annotating and code generating functions.

8 Problem 3.4

It is impossible to be able to do this with perfect accuracy at compile-time, because the variables that are initialised at any time might depend on information known only at run-time (such as user-input). I would err on the side of being more restrictive rather than less restrictive – forcing variables to always have some though-out assigned value before use makes programmers think about these edge cases, which eliminates a whole class of bugs. Thus I would enforce a rule that a variable must be assigned before use no matter what the execution path through the code. The most difficult control structure to deal with is `exit`, as only these break normal control flow. A function that checks this property follows:

```
let intersect2 xs ys = filter (fun x -> mem x xs) ys
let intersect (xs :: xss) = fold_right intersect2 xss xs

let has_exit = function
    Seq (x :: xs) -> has_exit x || has_exit (Seq xs)
    | Exit -> true
    | IfStmt(e, ifpt, elsept) -> has_exit ifpt || has_exit elsept
    | WhileStmt(test, body) -> has_exit body
    | RepeatStmt(body, test) -> has_exit body
    | LoopStmt body -> false
    | CaseStmt (switch, cases, default) ->
        has_exit default || exists has_exit (map snd cases)
    | _ -> false

let will_exit = function
    Seq (x :: xs) -> will_exit x || will_exit (Seq xs)
    | Exit -> true
    | IfStmt(e, ifpt, elsept) -> will_exit ifpt && will_exit elsept
    | WhileStmt _ -> false
    | RepeatStmt (body, test) -> will_exit body
    | LoopStmt body -> false
    | CaseStmt (switch, cases, default) ->
        will_exit default && for_all will_exit (map snd cases)
    | _ -> false

(* f takes an expression and returns the variables it uses
 * g takes a statement and returns a tuple: the variables it
 * needs to be initialised prior, and the variables it
 * certainly initialises *)
let test_stmt =
    let rec f = function
        Constant _ -> []
        | Variable x -> [x.x_lab]
        | Monop (_, e) -> f e
        | Binop (_, e1, e2) -> f e1 @ f e2 in
    let rec g = function
        Skip -> ([], [])
        | Seq (x :: xs) ->
            (* A sequence needs everything that the first part needs,
             * together with everything the second part needs that is
```

```
     * not defined in the first part.
     * A sequence defines everything that the first or second
     * part defines
     * SPECIAL CASE: if the first thing will exit, then ignore
     * the rest. If the first thing might exit, then we
     * must take into account the entire Seq for needs,
     * and only the first part for things it defines *)
    let (usedl, defsl) = g x
    and (usedr, defsr) = g (Seq xs)
    and used = usedl @ (filter (fun x -> not (mem x defsl)) usedr)
    and defs = defsl @ defsr
    in if will_exit x then (usedl, defsl)
    else if has_exit x then (used, defsl)
    else (used, defs)
| Seq [] -> ([], [])
| Assign (v, e) -> (f e, [v.x_lab])
| Print e -> (f e, [])
| Newline -> ([], [])
| IfStmt (test, thenpt, elsept) ->
    (* An if statement needs everything any of its clauses
     * needs, and defines only what both of its clauses
     * defines *)
    let used1 = f test
    and (used2, defs2) = g thenpt
    and (used3, defs3) = g elsept
    in (used1 @ used2 @ used3
       , intersect2 defs2 def3)
| WhileStmt (test, body) ->
    (* a while statement needs everything its test or
     * body might need, and defines nothing (as the body
     * might not even be run *)
    let used1 = f test
    and (used2, _) = g body
    in (used1 @ used2, [])
| RepeatStmt (body, test) ->
    (* a repeat statement is like a while, just that
     * it defines everything its body defines, since
     * the body is guaranteed to be run *)
    let used1 = f test
    and (used2, defs) = g body
    in (used1 @ used2, defs)
| ExitStmt -> ([], [])
| LoopStmt body ->
    (* A loop statement needs everything its body needs,
     * and defines what it's body defines *)
     g body
| CaseStmt (switch, cases, default) ->
    (* A case statement needs everything any of its
     * body or switch needs, and defines whatever
     * all of the cases defines for sure *)
    let used1 = f switch
```

```
        and used2 = concat (map fst (map g (map snd cases)))
        and defs2 = map snd (map g (map snd cases))
        and (used3, defs3) = g default
        in (used1 @ used2 @ used3, intersect (defs3 :: defs2))
in function x -> length (fst (g x)) == 0
```