Note: I've done all of the text in latex, but all diagrams by hand in an annexe.

1

The rules mentioned (on lines 1, 4, 6, 9, 17, 22, 36–40, 42–44 and 49), are:

1. $reg \to \langle Const\ k \rangle$ (when fits move k)
2. $reg \to \langle Local\ n \rangle$ (when fits add n)
3. $reg \to \langle Global\ x \rangle$
4. $reg \to \langle Loadw, addr \rangle$
5. $reg \to \langle Binop\ PlusA, reg1, rand \rangle$
6. $reg \to \langle Binop\ Lsl, reg1, rand \rangle$
7. $rand \to \langle Const\ k \rangle$ (when fits immed k)
8. $rand \to \langle Binop\ Lsl, reg, \langle Const\ n \rangle \rangle$ (when $n < 32$)
9. $rand \to \langle Binop\ Lsl, reg1, reg2 \rangle$
10. $rand \to reg$
11. $addr \to \langle Local\ n \rangle$ (when fits offset n)
12. $addr \to \langle Binop\ PlusA, reg1, reg2 \rangle$
13. $addr \to \langle Binop\ PlusA, reg1, \langle Binop\ Lsl, reg2, \langle Const\ n \rangle \rangle \rangle$ (when $n < 32$)
14. $addr \to reg$
15. $stmt \to \langle Storew, reg, addr \rangle$

To systematically calculate the number of ways in which we can tile the tree with these rules, we use dynamic programming:

- For each node i, and each symbol $S \in \{stmt, addr, rand, reg\}$ we calculate d_i^S, the number of ways in which we can tile the subtree whose root is at i, such that the subtree can be generated from the symbol S.
- To find d_i^S, assuming that $d_j^{S'}$ has been found for all nodes $j \neq i$ in the subtree of i and all symbols S', we use the following formula:
  $d_i^S = \sum_{p \text{ is a pattern for } S \text{ that matches subtree at } i} \left(\prod_{X \text{is non-terminal in p}} d^X_{\text{node where } X \text{ falls}} \right)$

The answer is in d_{root}^{stmt}.
Numbering the nodes as in the diagram 1, the following table synthesizes the information found by the dynamic programming algorithm:

Node i	Symbol S	Rules	d_i^S
1	reg		0
1	rand	10	0
1	addr	14	0
1	stmt	15	<u>28</u>
2	reg	4	14
2	rand	10	14
2	addr	14	14
2	stmt		0
3	reg	2	1
3	rand	10	1
3	addr	11, 14	2
3	stmt		0
4	reg	5	8
4	rand	10	8
4	addr	12, 13, 14	14
4	stmt		0
5	reg	3	1
5	rand	10	1
5	addr	14	1
5	stmt		0
6	reg	6	4
6	rand	8, 9, 10	8
6	addr	14	4
6	stmt		0
7	reg	4	2
7	rand	10	2
7	addr	14	2
7	stmt		0
8	reg	1	1
8	rand	7, 10	2
8	addr	14	1
8	stmt		0
9	reg	2	1
9	rand	10	1
9	addr	11, 14	2
9	stmt		0

So the answer is that there are 28 different tilings.

By inspecting these, I've found that the shortest code is:

```
ldr r0, =_a
ldr r1, [fp, #40]
ldr r2, [r0, r1, LSL #2]
str r2, [fp, #-8]
```

And that the worst code is the one that uses the least specific rule at each possible choice (i.e. splits each node into its own tile), and that stores all temporary values in registers:

```
ldr r0, =_a
```

```
add r1, fp, #40
ldr r2, [r1]
mov r3, #2
lsl r4, r2, r3
add r5, r0, r4
ldr r6, [r5]
add r7, fp, #-8
str r6, [r7]
```

2

The code that implements this section of code generation is listed below:

```
| <BINOP Times, t1, t2> ->
    (* The mul instruction needs both operands in registers *)
    let v1 = e_reg t1 anyreg in
    let v2 = e_reg t2 anyreg in
    gen_reg "mul" [r; v1; v2]
```

The condition is enforced by recursively calling e_reg on t1 and t2, as these calls generate code that places these operands in registers.

3

Assume that both pointers and integers are 4 bytes long and are aligned to 4 bytes.
The record layout will be thus:

- 4 bytes for `data`
- 4 bytes for `next`

and the record will be 4 byte aligned.

One possible layout for the stack frame of *sum* is:

- at fp + 16, the parameter p
- at fp + 12, the static link
- at fp + 8, the code adress where *sum* begins.
- at fp + 4, the return address
- at fp, the dynamic link
- at fp - 4, the pointer q
- at fp - 8, the integer s

Where fp is the frame pointer.

4

The following is a Keiko instruction sequence that implements the given statements:

`Instruction`	Stack after instruction

LOCAL -8	&s
LOADW	s
LOCAL -4	s; &q
LOADW	s; q
LOADW	s; q↑.data
PLUS	s + q↑.data
LOCAL -8	s + q↑.data; &s
STOREW	
LOCAL -4	&q
LOADW	q
CONST 4	q; 4
OFFSET	q + 4
LOADW	q↑.next
LOCAL -4	q↑.next; &q
STOREW	

The associated trees are found in diagram 2.

5

The tiling shown in diagram 3 creates the following object code:

```
ldr r0, [fp, #-8]
ldr r1, [fp, #-4]
ldr r1, [r1]
add r0, r0, r1
str r0, [fp, #-8]
ldr r0, [fp, #-4]
ldr r0, [r0, #4]
str r0, [fp, #-4]
```

6

We notice that there is a common subexpression between the trees; we can express the optimisation that results from only calculating this once by transforming the trees as shown in diagram 4. This optimisation creates the following code:

```
ldr r0, [fp, #-4]
ldr r1, [fp, #-8]
ldr r2, [r0]
add r1, r1, r2
str r1, [fp, #-8]
ldr r1, [r0, #4]
str r1, [fp, #-4]
```

This is better because it no longer stores [fp, #-4] in a register twice before changing its value.

7

An optimisation that can be applied if we consider the entire loop is making q and s live in registers, say r0 and r1, for the duration of the loop. This saves us from having to continually store and load their values. The code inside the loop would then be reduced to just:

```
ldr r2, [r0]
add r1, r1, r2
ldr r0, [r0, #4]
```

We would also need to add some code before and after the loop, to load q and s into r0 and r1 initially, and then to store them back afterwards, i.e.:

```
ldr r0, [fp, #-4]
ldr r1, [fp, #-8]
```

and:

```
str r0, [fp, #-4]
str r1, [fp, #-8]
```

respectively. Note that this is an improvement in the number of instructions run for the loop, provided that the loop runs at least twice, and if it runs many times, it is a massive improvement, reducing the number of instructions to about a half of the number of instructions we used previously.

8

8.1

Suppose x is at offset -4 from the frame pointer and y is at offset -8 from the frame pointer.
Code without movm:

```
ldr r0, [fp, #-8]
str r0, [fp, #-4]
```

Code with movm:

```
sub r0, fp, #8
sub r1, fp, #4
movm [r1], [r0]
```

So under the assumptions given, the code with movm is 50% worse than the code without it.

8.2

Suppose x and y are local pointers to integers, again at offsets -4 and -8, and we want to compile `*x := *y`.
Code without movm:

```
ldr r0, [fp, #-8]
ldr r1, [fp, #-4]
ldr r0, [r0]
str r0, [r1]
```

Code with movm:

```
ldr  r0 ,  [fp ,  #-8]
ldr  r1 ,  [fp ,  #-4]
movm  [r1] ,  [r0]
```

8.3

Figure 14.2 seems to have been moved, becoming figure 8.5.
To add movm into the grammar we need to add the following rule:

stmt → ⟨ STOREW, ⟨ LOADW, reg1 ⟩, reg2 ⟩ {`movm [reg1] [reg2]`}

(i.e. if we know how to calculate two addresses in registers, then we can assign one address' value to the other using a `movm` instruction)

8.4

Suppose we consider the following sequence of Keiko instructions:

```
LOCAL 16
LOADW
GLOBAL _a
LOCAL 20
LOADW
OFFSET
STOREW
```

Such code would be useful for storing a procedure's first parameter in a word of a global array, whose index corresponds to the second parameter of the procedure. The greedy instruction selection algorithm would produce the following suboptimal code (which corresponds to tiling 1 of diagram 5):

```
add  r0 ,  fp ,  #16
ldr  r1 ,  =_a
ldr  r2 ,  [fp ,  #20]
add  r1 ,  r1 ,  r2
movm  [r1] ,  [r0]
```

Whereas the following sequence of instructions (which corresponds to tiling 2 of diagram 5) is better:

```
ldr  r0 ,  [fp ,  #16]
ldr  r1 ,  =_a
ldr  r2 ,  [fp ,  #20]
str  r0 ,  [r1 ,  r2]
```

So the greedy selection algorithm will not always generate correct code.

8.5

I was unable to find the definition of "cost vectors", but I assume that they are vectors v_i associated with each node i that contain the minimal number of tiles needed to make the subtree rooted at node i match any particular symbol. On this assumption, I computed

these vectors for the trees of the indicated statements, in a manner simmilar to the dynamic programming algorithm in 1. I wrote these vectors in diagrams 6 and 7. Using this information, at each node, we can calculate the total cost if we use any particular rule on the root, and thus we can select the rule that will certainly minimise the number of instructions used overall. Applying this recursively, we get the optimal tilings.

9

I assume that a is an array of 1 byte characters. Suppose i is at offset -4 from the frame pointer. Then the following is Keiko code that runs the desired statement:

```
GLOBAL _a
LOCAL -4
LOADW
OFFSET
LOADW
LOCAL -4
LOADW
PLUS
GLOBAL _a
GLOBAL _a
LOCAL -4
LOADW
OFFSET
LOADW
OFFSET
STOREW
```

The unoptimised tree is depicted in diagram 8; diagram 9 holds the tree optimised by common subexpression elimination. Both have tiles that correspond to the machine code below. The machine code for the unoptimised tree follows:

```
ldr r0, =_a
ldr r1, [fp, #-4]
ldr r0, [r0, r1]
ldr r1, [fp, #-4]
add r0, r0, r1
ldr r1, =_a
ldr r2, =_a
ldr r3, [fp, #-4]
ldr r2, [r2, r3]
str r0, [r1, r2]
```

And for the optimised tree:

```
ldr r0, [fp, #-4]
ldr r1, =_a
ldr r2, [r1, r0]
add r3, r2, r0
str r3, [r1, r2]
```

If the machine had such an adressing mode, the storing _a in a register would become pointless, since we only ever use r1 as a base in memory anyway; all other common subexpressions should still be shared. This policy would result in the following code:

```
ldr r0, [fp, #-4]
ldr r2, [=_a, r0]
add r3, r2, r0
str r3, [=_a, r2]
```

9.1

I assume that x, y and z are not aliases of each other.
Suppose that x, y and z are held at offsets -4, -8 and -12 w.r.t. the frame pointer. Diagram 10 holds the initial trees for this statement, and diagram 11 the trees that result after common subexpression elimination.
This optimisation occurs as follows: while the common subexpression eliminator does a depth first search through the trees, it maintains a hash table that associates each subexpression with the number of times it occurs in the tree, as well as a directed acyclic graph, with one node for each expression, that represents the way expressions depend on each other. Expressions that are referenced multiple times may be stored in temporary variables. When we process a `STOREW` instruction, in order to make sure that we don't share an expression which used the old value stored at that position, we remove all expressions that reference that memory from the hash table, without removing them from the directed acyclic graph. In order to optimise more, we also add a corresponding `LOADW` node to the table and the graph, that will produce the value that we have just written. This way we can automatically use the fact that we already have this value in a register, thus avoiding recalculating it.
This is the machine code associated with these trees:

```
ldr r0, [fp, #-8]
ldr r1, [fp, #-4]
sub r1, r1, r0
str r1, [fp, #-4]
sub r2, r1, r0
str r2, [fp, #-8]
sub r3, r1, r2
str r3, [fp, #-12]
```

10

10.1

The trees are found in diagram 12.

10.2

The trees are found in diagram 13-14. Diagram 14 also tiles the trees.

10.3

The main assumption is that the assignments a[i] := a[j], a[j] := t do not change the values of i and j. This is justified, as integer parameters are copied onto the stack, and thus should not alias any position in an array, or the local variable t. A conservative alias analyser might miss this observation.

10.4

```
!Put _a into r0
ldr r0, =_a

!Put i = [fp, #40] into r1
ldr r1, [fp, #40]

!Put _a+4i into r1
add r1, r0, r1, LSL 2

!Put j = [fp, #44] into r2
ldr r2, [fp, #44]

!Put _a+4j into r2
add r2, r0, r2, LSL 2

!Put a[i] into t=r4
ldr r4, [r1]

!Put a[j] into r3
ldr r3, [r2]

!Store a[j] in a[i]
str r3, [r1]

!Store r4=t in a[j]
str r4, [r2]
```

10.5

I believe this code to be optimal, as no common subexpressions are not shared, and since all trees in our sequence have been converted into instructions in an optimal way.