The rules mentioned (on lines 1, 4, 6, 9, 17, 22, 36–40, 42–44 and 49), without semantic actions (which make them all distinct), are:

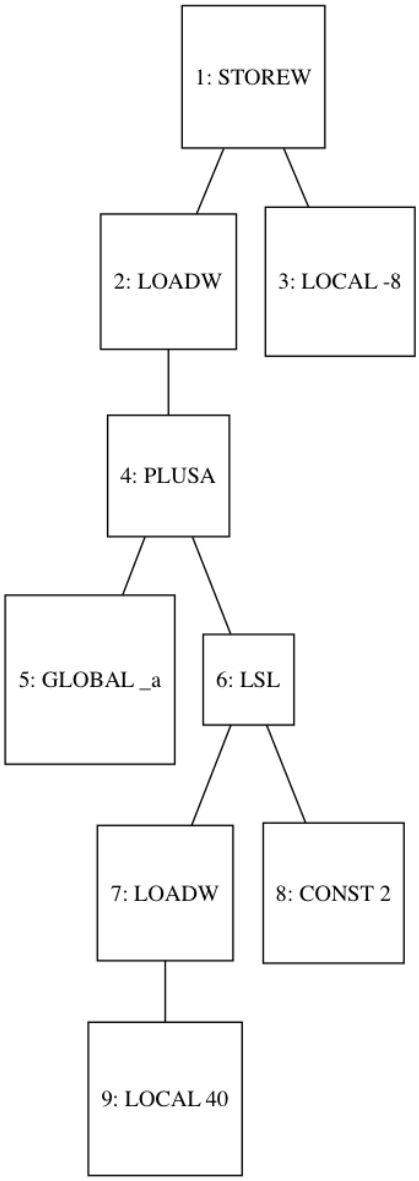
1. $reg \to \langle Const\ k \rangle$ (when fits move k)
2. $reg \to \langle Local\ n \rangle$ (when fits add n)
3. $reg \to \langle Global\ x \rangle$
4. $reg \to \langle Loadw, addr \rangle$
5. $reg \to \langle Binop\ PlusA, reg1, rand \rangle$
6. $reg \to \langle Binop\ Lsl, reg1, rand \rangle$
7. $rand \to \langle Const\ k \rangle$ (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. $stmt \to \langle Storew, reg, addr \rangle$

Now to systematically calculate in how many ways 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 with it's root at i, and such that the tree can be generated from the symbol S, from bottom to top.
- To find d_i^S, assuming that $d_{\bar{j}}$ has been found for all nodes $j \neq i$ in the subtree of i, use the following formula:
  $d_i^S = \sum_{\text{Pattern } p \text{ for } S \text{ matches subtree at } i} \left(\prod_{X \text{non-terminal in p}} d^X_{\text{node of } X} \right)$

The answer is in d_{root}^{stmt}.

Numbering the nodes as in the diagram, the table holds 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	28
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 all 28 trees, I've found that the shortest code is precisely the one in the notes:

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

This is almost the "bad" code from the notes, just that instead of directly computing `lsl r3, r1, #2`, we first move `#2` into a register and then compute this.

Assume pointers are 4 bytes and aligned to 4 bytes, and integers are 4 bytes and 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.

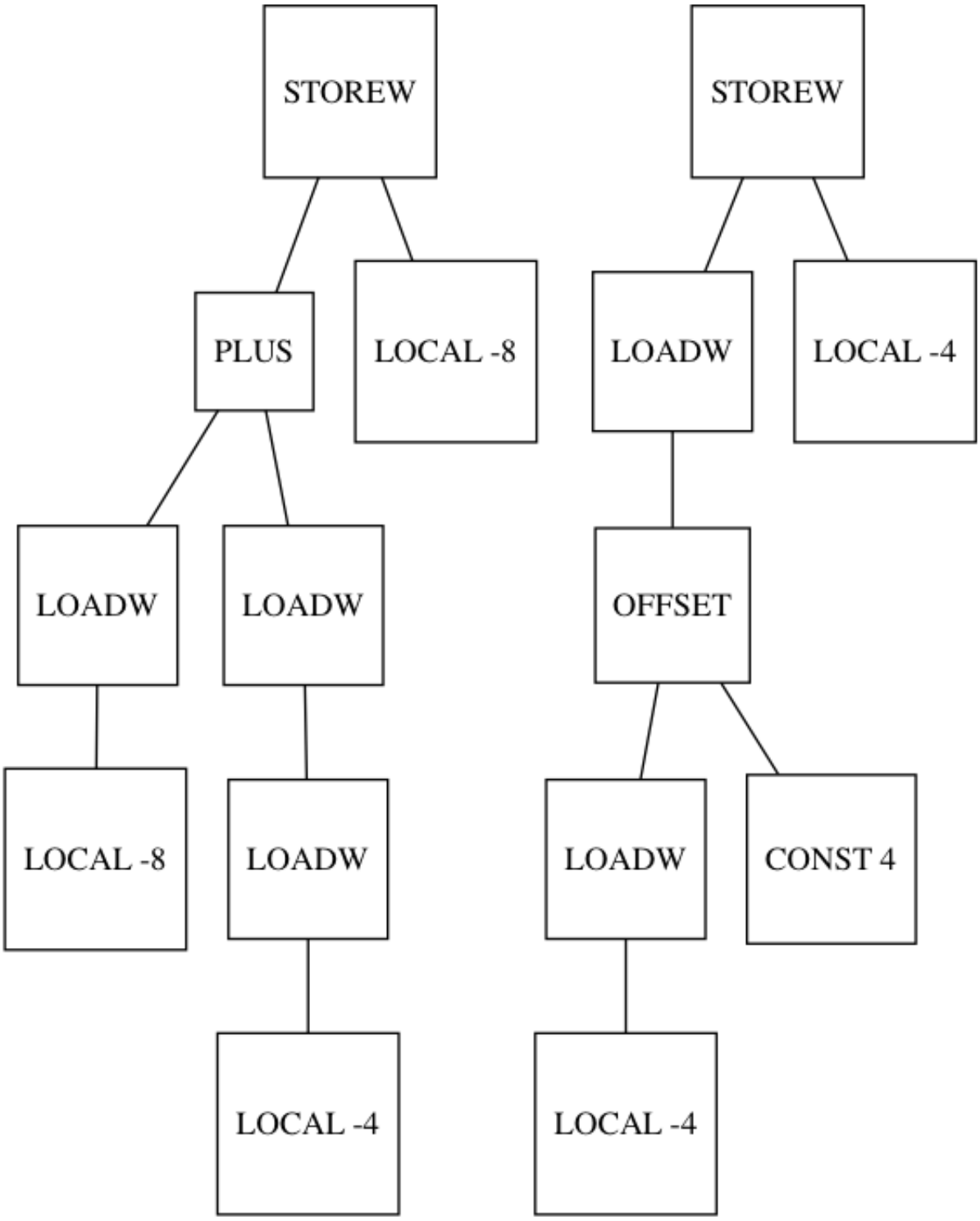
Suppose that the stack layout is thus:

- at fp - 8, the second local variable byte (supposijng it is 4 bytes long)
- at fp - 4, the first local variable (supposing it is 4 bytes long)
- at fp, the dynamic link
- at fp + 4, the return address
- at fp + 8, the beginning of the procedure currenty being run
- at fp + 12, the static link
- at fp + 16, the first argument
- at fp + 20, the second argument
- ...

(I assume that I'm meant to build these in Keiko syntax)

Instruction sequence, commented with the stack, where &s is the address of s:

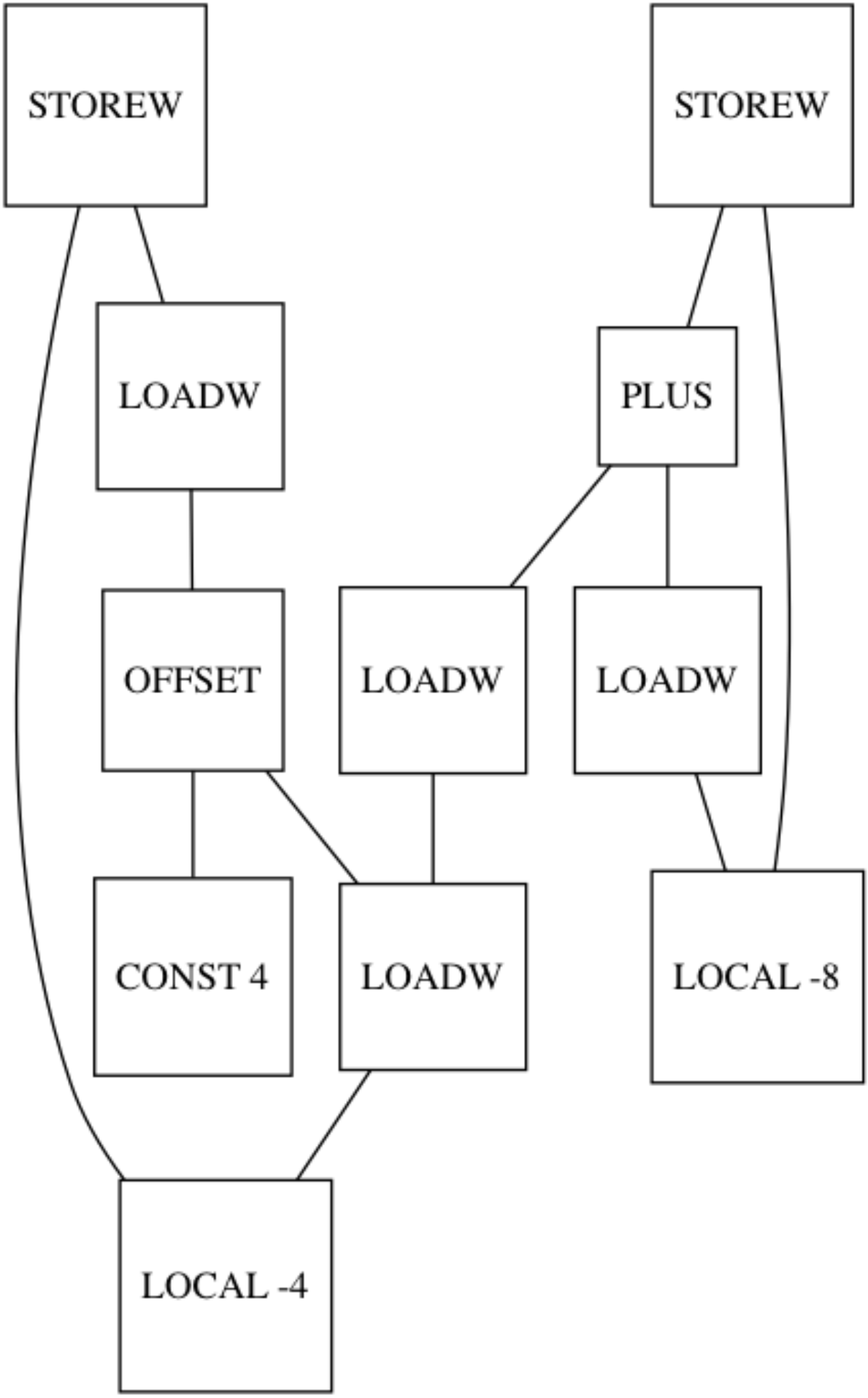
```
LOCAL -8    ! &s
LOADW       ! s
LOCAL -4    ! s; &q
LOADW       ! s; q
LOADW       ! s; q->data
PLUS        ! s + q->data
LOCAL -8    ! s + q->data; &s
STOREW      !
LOCAL -4    ! &q
LOADW       ! q
CONST 4     ! q; 4
OFFSET      ! q+4
LOADW       ! q->next
LOCAL -4    ! q->next, &q
STOREW
```


STOREW
STOREW
PLUS
LOCAL -8
LOADW
LOCAL -4
LOADW
LOADW
OFFSET
LOCAL -8
LOADW
LOADW
CONST 4
LOCAL -4
LOCAL -4

TODO: tiling
This tiling creates the following object code:

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

We notice that there is a common subexpression between the trees; we can express the optimisation that results from only calculating this once by turning our trees into a DAG, as follows:


STOREW
STOREW
LOADW
PLUS
OFFSET
LOADW
LOADW
CONST 4
LOADW
LOCAL -8
LOCAL -4

(note that although we eliminated the common `LOCAL` subtrees, this does not actually improve the object code generated, as those values are handled by ARM's adressing modes).

This optimisation creates the following code, that no longer recalculates [fp, #-4].

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

Considering the entire loop leads us to holding q and p in registers, say $r0$ and $r1$. This means that we no longer need to load the value of q and the value of s from memory, nor do we need to store their new values in memory, until the end of the loop. This leads to the following code inside the loop:

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

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], [r2]
```

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

2

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

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

Code with movm:

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

3