8 Basic Blocks and Traces

ca-non-i-cal: reduced to the simplest or clearest schema possible

Webster's Dictionary

The trees generated by the semantic analysis phase must be translated into assembly or machine language. The operators of the `Tree` language are chosen carefully to match the capabilities of most machines. However, there are certain aspects of the tree language that do not correspond exactly with machine languages, and some aspects of the `Tree` language interfere with compile-time optimization analyses.

For example, it's useful to be able to evaluate the subexpressions of an expression in any order. But the subexpressions of `Tree.exp` can contain side effects – ESEQ and CALL nodes that contain assignment statements and perform input/output. If tree expressions did not contain ESEQ and CALL nodes, then the order of evaluation would not matter.

Some of the mismatches between `Trees` and machine-language programs are:

- The CJUMP instruction can jump to either of two labels, but real machines' conditional jump instructions fall through to the next instruction if the condition is false.
- ESEQ nodes within expressions are inconvenient, because they make different orders of evaluating subtrees yield different results.
- CALL nodes within expressions cause the same problem.
- CALL nodes within the argument-expressions of other CALL nodes will cause problems when trying to put arguments into a fixed set of formal-parameter registers.

Why does the `Tree` language allow ESEQ and two-way CJUMP, if they are so troublesome? Because they make it much more convenient for the `Translate` (translation to intermediate code) phase of the compiler.

We can take any tree and rewrite it into an equivalent tree without any of the cases listed above. Without these cases, the only possible parent of a SEQ node is another SEQ; all the SEQ nodes will be clustered at the top of the tree. This makes the SEQs entirely uninteresting; we might as well get rid of them and make a linear list of `Tree.stm`s.

The transformation is done in three stages: First, a tree is rewritten into a list of *canonical trees* without SEQ or ESEQ nodes; then this list is grouped into a set of *basic blocks*, which contain no internal jumps or labels; then the basic blocks are ordered into a set of *traces* in which every CJUMP is immediately followed by its `false` label.

Thus the module `Canon` has these tree-rearrangement functions:

```
signature CANON =
sig
  val linearize : Tree.stm -> Tree.stm list

  val basicBlocks : Tree.stm list ->
                         (Tree.stm list list * Temp.label)

  val traceSchedule : Tree.stm list list * Temp.label ->
                         Tree.stm list
end

structure Canon : Canon
```

`Linearize` removes the ESEQs and moves the CALLs to top level. Then `BasicBlocks` groups statements into sequences of straight-line code. Finally `traceSchedule` orders the blocks so that every CJUMP is followed by its `false` label.

8.1 CANONICAL TREES

Let us define *canonical trees* as having these properties:

1. No SEQ or ESEQ.
2. The parent of each CALL is either EXP(...) or MOVE(TEMP t, ...).

TRANSFORMATIONS ON ESEQ

How can the ESEQ nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become SEQ nodes.

Figure 8.1 gives some useful identities on trees.

Identity (1) is obvious. So is identity (2): Statement s is to be evaluated; then e_1; then e_2; then the sum of the expressions is returned. If s has side effects that affect e_1 or e_2, then either the left-hand side or the right-hand side of the first equation will execute those side effects before the expressions are evaluated.

Identity (3) is more complicated, because of the need not to interchange the evaluations of s and e_1. For example, if s is MOVE(MEM(x), y) and e_1 is BINOP(PLUS, MEM(x), z), then the program will compute a different result if s is evaluated before e_1 instead of after. Our goal is simply to pull s out of the BINOP expression; but now (to preserve the order of evaluation) we must pull e_1 out of the BINOP with it. To do so, we assign e_1 into a new temporary t, and put t inside the BINOP.

It may happen that s causes no side effects that can alter the result produced by e_1. This will happen if the temporaries and memory locations assigned by s are not referenced by e_1 (and s and e_1 don't both perform external I/O). In this case, identity (4) can be used.

We cannot always tell if two expressions commute. For example, whether MOVE(MEM(x), y) commutes with MEM(z) depends on whether $x = z$, which we cannot always determine at compile time. So we *conservatively approximate* whether statements commute, saying either "they definitely do commute" or "perhaps they don't commute." For example, we know that any statement "definitely commutes" with the expression CONST(n), so we can use identity (4) to justify special cases like

$$\text{BINOP}(op, \text{CONST}(n), \text{ESEQ}(s, e)) = \text{ESEQ}(s, \text{BINOP}(op, \text{CONST}(n), e)).$$

The `commute` function estimates (very naively) whether two expressions commute:

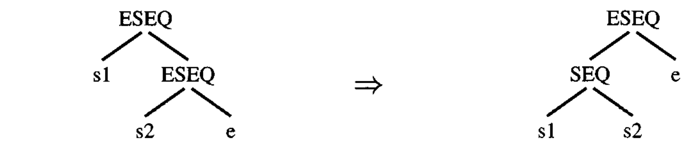
```
fun commute(T.EXP(T.CONST _ ), _ ) = true
  | commute(_, T.NAME _ ) = true
  | commute(_, T.CONST _ ) = true
  | commute _ = false
```

A constant commutes with any statement, and the empty statement commutes with any expression. Anything else is assumed not to commute.

GENERAL REWRITING RULES

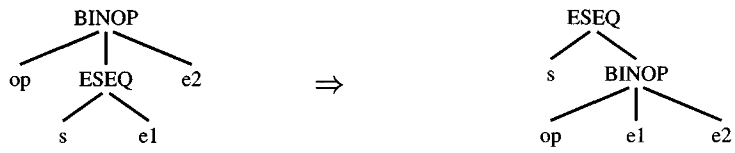
In general, for each kind of `Tree` statement or expression we can identify the subexpressions. Then we can make rewriting rules, similar to the ones in

(1)



$$\text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) \quad = \quad \text{ESEQ}(\text{SEQ}(s_1, s_2), e)$$

(2)



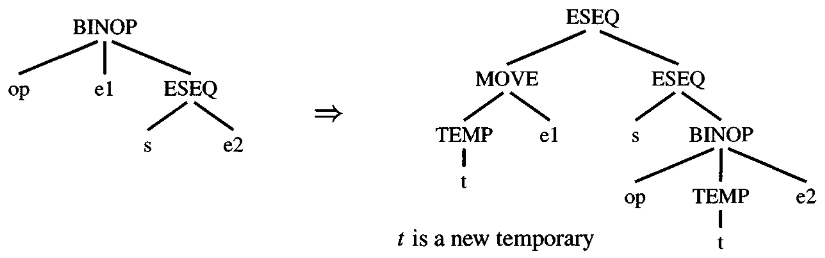
$$\text{BINOP}(op, \text{ESEQ}(s, e_1), e_2) \quad = \quad \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$$

$$\text{MEM}(\text{ESEQ}(s, e_1)) \quad = \quad \text{ESEQ}(s, \text{MEM}(e_1))$$

$$\text{JUMP}(\text{ESEQ}(s, e_1)) \quad = \quad \text{SEQ}(s, \text{JUMP}(e_1))$$

$$\text{CJUMP}(op, \text{ESEQ}(s, e_1), e_2, l_1, l_2) \quad = \quad \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$$

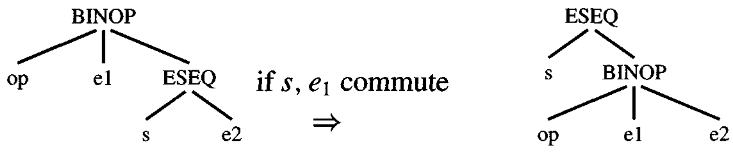
(3)



$$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) \quad = \quad \text{ESEQ}(\text{MOVE}(\text{TEMP } t, e_1), \text{ESEQ}(s, \text{BINOP}(op, \text{TEMP } t, e_2)))$$

$$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) \quad = \quad \text{SEQ}(\text{MOVE}(\text{TEMP } t, e_1), \text{SEQ}(s, \text{CJUMP}(op, \text{TEMP } t, e_2, l_1, l_2)))$$

(4)



if s, e_1 commute

$$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2)) \quad = \quad \text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$$

$$\text{CJUMP}(op, e_1, \text{ESEQ}(s, e_2), l_1, l_2) \quad = \quad \text{SEQ}(s, \text{CJUMP}(op, e_1, e_2, l_1, l_2))$$

FIGURE 8.1. Identities on trees (see also Exercise 8.1).

Figure 8.1, to pull the ESEQs out of the statement or expression.

For example, in $[e_1, e_2, \text{ESEQ}(s, e_3)]$, the statement s must be pulled leftward past e_2 and e_1. If they commute, we have $(s;\ [e_1, e_2, e_3])$. But suppose e_2 does not commute with s; then we must have

$$(\text{SEQ}(\text{MOVE}(t_1, e_1), \text{SEQ}(\text{MOVE}(t_2, e_2), s));\quad [\text{TEMP}(t_1), \text{TEMP}(t_2), e_3])$$

Or if e_2 commutes with s but e_1 does not, we have

$$(\text{SEQ}(\text{MOVE}(t_1, e_1), s);\quad [\text{TEMP}(t_1),\ e_2,\ e_3])$$

The `reorder` function takes a list of expressions and returns a pair of (statement, expression-list). The statement contains all the things that must be executed before the expression-list. As shown in these examples, this includes all the statement-parts of the ESEQs, as well as any expressions to their left with which they did not commute. When there are no ESEQs at all we will use EXP(CONST 0), which does nothing, as the statement.

Algorithm. Step one is to make a "subexpression-extraction" method for each kind. Step two is to make a "subexpression-insertion" method: given an ESEQ-clean version of each subexpression, this builds a new version of the expression or statement.

ML makes this easy to express using little "`fn`" functions:

```
val reorder_stm: Tree.exp list * (Tree.exp list -> Tree.stm)
                                    -> Tree.stm

val reorder_exp: Tree.exp list * (Tree.exp list -> Tree.exp)
                                    -> Tree.stm * Tree.exp

fun do_stm(T.JUMP(e,labs)) =
            reorder_stm([e],fn [e] => T.JUMP(e,labs))
  | do_stm(T.CJUMP(p,a,b,t,f)) =
            reorder_stm([a,b], fn[a,b]=> T.CJUMP(p,a,b,t,f))
    ⋮ and so on

and do_exp(T.BINOP(p,a,b)) =
            reorder_exp([a,b], fn[a,b]=>T.BINOP(p,a,b))
  | do_exp(T.MEM(a)) =
            reorder_exp([a], fn[a]=>T.MEM(a))
    ⋮ and so on
```

`Reorder_stm` takes two arguments – a list l of subexpressions and a *build* function. It pulls all the ESEQs out of the l, yielding a statement s_1 that contains all the statements from the ESEQs and a list l' of cleaned-up expressions. Then it makes SEQ$(s_1, \mathit{build}(l'))$.

`Reorder_exp`(l, build) is similar, except that it returns a pair (s, e) where s is a statement containing all the side effects pulled out of l, and e is $\mathit{build}(l')$.

To pull *all* the ESEQs from l, the functions `reorder_stm` and `reorder_exp` must call `do_exp` recursively. See Exercise 8.3.

The left-hand operand of the MOVE statement is not considered a subexpression, because it is the *destination* of the statement – its value is not used by the statement. However, if the destination is a memory location, then the *address* acts like a source. Thus we have,

```
  | do_stm(T.MOVE(T.TEMP t,b)) =
             reorder_stm([b],fn[b]=>T.MOVE(T.TEMP t,b))
  | do_stm(T.MOVE(T.MEM e,b)) =
             reorder_stm([e,b],fn[e,b]=>T.MOVE(T.MEM e,b))
```

Now, given a list of extracted subexpressions, we pull the ESEQs out, from right to left.

MOVING CALLS TO TOP LEVEL

The `Tree` language permits CALL nodes to be used as subexpressions. However, the actual implementation of CALL will be that each function returns its result in the same dedicated return-value register TEMP(RV). Thus, if we have

BINOP(PLUS, CALL(...), CALL(...))

the second call will overwrite the RV register before the PLUS can be executed.

We can solve this problem with a rewriting rule. The idea is to assign each return value immediately into a fresh temporary register, that is

$$\text{CALL}(\mathit{fun}, \mathit{args}) \quad \rightarrow \quad \text{ESEQ}(\text{MOVE}(\text{TEMP}\ t, \text{CALL}(\mathit{fun}, \mathit{args})), \text{TEMP}\ t)$$

Now the ESEQ-eliminator will percolate the MOVE up outside of its containing BINOP (etc.) expressions.

This technique will generate a few extra MOVE instructions, which the register allocator (Chapter 11) can clean up.

The rewriting rule is implemented as follows: `reorder` replaces any occurrence of CALL(f, args) by

$$\text{ESEQ}(\text{MOVE}(\text{TEMP}\ t_{\text{new}}, \text{CALL}(f, \mathit{args})),\ \text{TEMP}\ t_{\text{new}})$$

and calls itself again on the ESEQ. But `do_stm` recognizes the pattern

$$\text{MOVE}(\text{TEMP}\ t_{\text{new}}, \text{CALL}(f, \mathit{args})),$$

and does not call `reorder` on the CALL node in that case, but treats the f and *args* as the children of the MOVE node. Thus, `reorder` never "sees" any CALL that is already the immediate child of a MOVE. Occurrences of the pattern EXP(CALL(f, *args*)) are treated similarly.

A LINEAR LIST OF STATEMENTS

Once an entire function body s_0 is processed with `do_stm`, the result is a tree s_0' where all the SEQ nodes are near the top (never underneath any other kind of node). The `linearize` function repeatedly applies the rule

$$\text{SEQ}(\text{SEQ}(a, b), c) \;=\; \text{SEQ}(a, \mathit{seq}(b, c))$$

The result is that s_0' is linearized into an expression of the form

$$\text{SEQ}(s_1, \text{SEQ}(s_2, \ldots, \text{SEQ}(s_{n-1}, s_n)\ldots))$$

Here the SEQ nodes provide no structuring information at all, and we can just consider this to be a simple list of statements,

$$s_1, s_2, \ldots, s_{n-1}, s_n$$

where none of the s_i contain SEQ or ESEQ nodes.

These rewrite rules are implemented by `linearize`, with an auxiliary function `linear`:

```
fun linearize(stm0: T.stm) : T.stm list =
let
      ⋮ definitions of reorder_exp, reorder_stm, do_exp, do_stm, etc.

   fun linear(T.SEQ(a,b),l) = linear(a,linear(b,l))
     | linear(s,l) = s::l
 in
   linear(do_stm stm0, nil)
end
```

8.2 TAMING CONDITIONAL BRANCHES

Another aspect of the `Tree` language that has no direct equivalent in most machine instruction sets is the two-way branch of the CJUMP instruction. The

`Tree` language CJUMP is designed with two target labels for convenience in translating into trees and analyzing trees. On a real machine, the conditional jump either transfers control (on a `true` condition) or "falls through" to the next instruction.

To make the trees easy to translate into machine instructions, we will rearrange them so that every CJUMP($cond, l_t, l_f$) is immediately followed by LABEL(l_f), its "false branch." Each such CJUMP can be directly implemented on a real machine as a conditional branch to label l_t.

We will make this transformation in two stages: first, we take the list of canonical trees and form them into *basic blocks*; then we order the basic blocks into a *trace*. The next sections will define these terms.

BASIC BLOCKS

In determining where the jumps go in a program, we are analyzing the program's *control flow*. Control flow is the sequencing of instructions in a program, ignoring the data values in registers and memory, and ignoring the arithmetic calculations. Of course, not knowing the data values means we cannot know whether the conditional jumps will go to their true or false labels; so we simply say that such jumps can go either way.

In analyzing the control flow of a program, any instruction that is not a jump has an entirely uninteresting behavior. We can lump together any sequence of non-branch instructions into a basic block and analyze the control flow between basic blocks.

A *basic block* is a sequence of statements that is always entered at the beginning and exited at the end, that is:

- The first statement is a LABEL.
- The last statement is a JUMP or CJUMP.
- There are no other LABELs, JUMPs, or CJUMPs.

The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The sequence is scanned from beginning to end; whenever a LABEL is found, a new block is started (and the previous block is ended); whenever a JUMP or CJUMP is found, a block is ended (and the next block is started). If this leaves any block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block. If any block has been left without a LABEL at the beginning, a new label is invented and stuck there.

We will apply this algorithm to each function-body in turn. The procedure "epilogue" (which pops the stack and returns to the caller) will not be part of this body, but is intended to follow the last statement. When the flow of program execution reaches the end of the last block, the epilogue should follow. But it is inconvenient to have a "special" block that must come last and that has no JUMP at the end. Thus, we will invent a new label `done` – intended to mean the beginning of the epilogue – and put a JUMP(NAME `done`) at the end of the last block.

In the Tiger compiler, the function `Canon.basicBlocks` implements this simple algorithm.

TRACES

Now the basic blocks can be arranged in any order, and the result of executing the program will be the same – every block ends with a jump to the appropriate place. We can take advantage of this to choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.

At the same time, we can also arrange that many of the unconditional JUMPs are immediately followed by their target label. This will allow the deletion of these jumps, which will make the compiled program run a bit faster.

A *trace* is a sequence of statements that could be consecutively executed during the execution of the program. It can include conditional branches. A program has many different, overlapping traces. For our purposes in arranging CJUMPs and false-labels, we want to make a set of traces that exactly covers the program: each block must be in exactly one trace. To minimize the number of JUMPs from one trace to another, we would like to have as few traces as possible in our covering set.

A very simple algorithm will suffice to find a covering set of traces. The idea is to start with some block – the beginning of a trace – and follow a possible execution path – the rest of the trace. Suppose block b_1 ends with a JUMP to b_4, and b_4 has a JUMP to b_6. Then we can make the trace b_1, b_4, b_6.

But suppose b_6 ends with a conditional jump CJUMP$(cond, b_7, b_3)$. We cannot know at compile time whether b_7 or b_3 will be next. But we can assume that some execution will follow b_3, so let us imagine it is that execution that we are simulating. Thus, we append b_3 to our trace and continue with the rest of the trace after b_3. The block b_7 will be in some other trace.

Algorithm 8.2 (which is similar to `Canon.traceSchedule`) orders the blocks into traces as follows: It starts with some block and follows a chain of jumps, marking each block and appending it to the current trace. Eventually

```
Put all the blocks of the program into a list $Q$.
**while** $Q$ is not empty
    Start a new (empty) trace, call it $T$.
    Remove the head element $b$ from $Q$.
    **while** $b$ is not marked
        Mark $b$; append $b$ to the end of the current trace $T$.
        Examine the successors of $b$ (the blocks to which $b$ branches);
        **if** there is any unmarked successor $c$
          $b \leftarrow c$
    End the current trace $T$.
```

ALGORITHM 8.2. Generation of traces.

it comes to a block whose successors are all marked, so it ends the trace and picks an unmarked block to start the next trace.

FINISHING UP

An efficient compiler will keep the statements grouped into basic blocks, because many kinds of analysis and optimization algorithms run faster on (relatively few) basic blocks than on (relatively many) individual statements. For the Tiger compiler, however, we seek simplicity in the implementation of later phases. So we will flatten the ordered list of traces back into one long list of statements.

At this point, most (but not all) CJUMPs will be followed by their true or false label. We perform some minor adjustments:

- Any CJUMP immediately followed by its false label we let alone (there will be many of these).
- For any CJUMP followed by its true label, we switch the true and false labels and negate the condition.
- For any CJUMP$(cond, a, b, l_t, l_f)$ followed by neither label, we invent a new false label l'_f and rewrite the single CJUMP statement as three statements, just to achieve the condition that the CJUMP is followed by its false label:

  CJUMP$(cond, a, b, l_t, l'_f)$
  LABEL l'_f
  JUMP(NAME l_f)

The trace-generating algorithm will tend to order the blocks so that many

prologue statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($>$, i, N, *done*, *body*)
LABEL(*body*)
loop body statements
JUMP(NAME *test*)

* * *

LABEL(*done*)
epilogue statements

(a)

prologue statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($\leq$, i, N, *body*, *done*)
LABEL(*done*)
epilogue statements

* * *

LABEL(*body*)
loop body statements
JUMP(NAME *test*)

(b)

prologue statements
JUMP(NAME *test*)

* * *

LABEL(*body*)
loop body statements
~~JUMP(NAME *test*)~~
LABEL(*test*)
CJUMP($>$, i, N, *done*, *body*)

* * *

LABEL(*done*)
epilogue statements

(c)

FIGURE 8.3. Different trace coverings for the same program.

of the unconditional JUMPs are immediately followed by their target labels. We can remove such jumps.

OPTIMAL TRACES

For some applications of traces, it is important that any frequently executed sequence of instructions (such as the body of a loop) should occupy its own trace. This helps not only to minimize the number of unconditional jumps, but also may help with other kinds of optimization such as register allocation and instruction scheduling.

Figure 8.3 shows the same program organized into traces in different ways. Figure 8.3a has a CJUMP and a JUMP in every iteration of the **while**-loop; Figure 8.3b uses a different trace covering, also with CJUMP and a JUMP in every iteration. But 8.3c shows a better trace covering, with no JUMP in each iteration.

The Tiger compiler's `Canon` module doesn't attempt to optimize traces around loops, but it is sufficient for the purpose of cleaning up the `Tree`-statement lists for generating assembly code.

FURTHER READING

The rewrite rules of Figure 8.1 are an example of a *term rewriting system*; such systems have been much studied [Dershowitz and Jouannaud 1990].

Fisher [1981] shows how to cover a program with traces so that frequently executing paths tend to stay within the same trace. Such traces are useful for program optimization and scheduling.

EXERCISES

***8.1** The rewriting rules in Figure 8.1 are a subset of the rules necessary to eliminate all ESEQs from expressions. Show the right-hand side for each of the following incomplete rules:

a. $\mathrm{MOVE}(\mathrm{TEMP}\ t,\ \mathrm{ESEQ}(s, e)) \Rightarrow$

b. $\mathrm{MOVE}(\mathrm{MEM}(\mathrm{ESEQ}(s, e_1)), e_2) \Rightarrow$

c. $\mathrm{MOVE}(\mathrm{MEM}(e_1), \mathrm{ESEQ}(s, e_2)) \Rightarrow$

d. $\mathrm{EXP}(\mathrm{ESEQ}(s, e)) \Rightarrow$

e. $\mathrm{EXP}(\mathrm{CALL}(\mathrm{ESEQ}(s, e), args)) \Rightarrow$

f. $\mathrm{MOVE}(\mathrm{TEMP}\ t,\ \mathrm{CALL}(\mathrm{ESEQ}(s, e), args)) \Rightarrow$

g. $\mathrm{EXP}(\mathrm{CALL}(e_1, [e_2, \mathrm{ESEQ}(s, e_3), e_4])) \Rightarrow$

In some cases, you may need two different right-hand sides depending on whether something commutes (just as parts (3) and (4) of Figure 8.1 have different right-hand sides for the same left-hand sides).

8.2 Draw each of the following expressions as a tree diagram, and then apply the rewriting rules of Figure 8.1 and Exercise 8.1, as well as the CALL rule on page 178.

a. $\mathrm{MOVE}(\mathrm{MEM}(\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{CJUMP}(\mathrm{LT}, \mathrm{TEMP}_i, \mathrm{CONST}_0, L_{out}, L_{ok}), \mathrm{LABEL}_{ok}),$
$\mathrm{TEMP}_i)), \mathrm{CONST}_1)$

b. $\mathrm{MOVE}(\mathrm{MEM}(\mathrm{MEM}(\mathrm{NAME}_a)), \mathrm{MEM}(\mathrm{CALL}(\mathrm{TEMP}_f, [])))$

c. $\mathrm{BINOP}(\mathrm{PLUS}, \mathrm{CALL}(\mathrm{NAME}_f, [\mathrm{TEMP}_x]),$
$\mathrm{CALL}(\mathrm{NAME}_g, [\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}_x, \mathrm{CONST}_0), \mathrm{TEMP}_x)]))$

***8.3** The directory `$TIGER/chap8` contains an implementation of every algorithm described in this chapter. Read and understand it.

8.4 A primitive form of the `commute` test is shown on page 175. This function is conservative: if interchanging the order of evaluation of the expressions will change the result of executing the program, this function will definitely return false; but if an interchange is harmless, `commute` might return true or false.

Write a more powerful version of `commute` that returns true in more cases, but is still conservative. Document your program by drawing pictures of (pairs of) expression trees on which it will return true.

***8.5** The left-hand side of a MOVE node really represents a destination, not an expression. Consequently, the following rewrite rule is *not* a good idea:

$$\mathrm{MOVE}(e_1, \mathrm{ESEQ}(s, e_2)) \rightarrow \mathrm{SEQ}(s,\ \mathrm{MOVE}(e_1, e_2)) \qquad \text{if } s, e_1 \text{ commute}$$

Write an statement matching the left side of this rewrite rule that produces a different result when rewritten.

Hint: It is very reasonable to say that the statement MOVE(TEMP$_a$, TEMP$_b$) commutes with expression TEMP$_b$ (if a and b are not the same), since TEMP$_b$ yields the same value whether executed before or after the MOVE.

Conclusion: The only subexpression of MOVE(TEMP$_a$, e) is e, and the subexpressions of MOVE(MEM(e_1), e_2) are [e_1, e_2]; we should not say that a is a subexpression of MOVE(a, b).

8.6 Break this program into basic blocks.

```
1   $m \leftarrow 0$
2   $v \leftarrow 0$
3   if $v \geq n$ goto 15
4   $r \leftarrow v$
5   $s \leftarrow 0$
6   if $r < n$ goto 9
7   $v \leftarrow v + 1$
8   goto 3
9   $x \leftarrow M[r]$
10  $s \leftarrow s + x$
11  if $s \leq m$ goto 13
12  $m \leftarrow s$
13  $r \leftarrow r + 1$
14  goto 6
15  return $m$
```

8.7 Express the basic blocks of Exercise 8.6 as statements in the `Tree` intermediate form, and use Algorithm 8.2 to generate a set of traces.