1) Expressions delivering Boolean results require different information from those delivering integer results. Clearly $E_1 \text{ or } E_2$, $E_1 \text{ and } E_2$, $E_1 = E_2$, $E_1 IO E_2$ (for $IO \in \{<,>,\}$) and not $E$ belong to the first class, while $I := E$, $E_1 IO E_2$ (for $IO \in \{+,\text{-},\times,\text{div},\text{mod}\}$), $-E$, $I$ and $N$ belong to the second. $E_1; E_2$ and if $E_1$ then $E_2$ else $E_3$ could belong to either class.

2) The information that will be required for Boolean expressions will be:
   - True and false jump targets
   - Flag to tell which target is the fall-through
   - Flag to tell whether the expression is in a value context
   - If in a value context, where the result will be left

   The information that will be required for integer expressions will be:
   - Flag to tell whether a quad result is requested

3) No information from the definition table will be relevant to the construction of these two nodes. The phrase stack will hold a pointer to the node representing each component. This is necessary in any tree building compiler; no further information is needed for node construction, since the type of phrase determines the kind of node in most cases and in the remaining ones (which could belong to either class) it is sufficient to check the node types of the components via the pointers in the phrase stack.

4) None of the information in (2) is made available by the answer to (3). The jump targets and fall-through flag will be obtained from the parent of the node, and will be either passed from its parent or computed on the basis of information in that node. For example, targets would be generated for a condition’s results in a conditional or iteration as shown in Figure 3.9 of the text. The fall-through of the first expression of an and would be set to true, and the fall-through of the second expression would be the fall-through of the and’s parent. The value context flag would be set to true at a “same” node, false at a condition or iteration, and passed by every other node. If a “same” node is in a value context then it can pass its result location to its children; otherwise it must reserve a register for the result at the beginning of the generated code, pass that register to its children, and release it on completion of the generated code. The quad required flag is set true for the left operand of mod, and false in every other situation.
5) Certain trees can be coded and discarded by a MINILAX compiler that produces the postulated translation, but it is a rather tricky business and not recommended unless space requirements prove excessive. The first cut would be to code and remove any tree that yielded a void result. Unfortunately, such trees can appear within expressions (an expression may contain a block as a subexpression, and all but the last statement of a block are voided). That means that code for one of the component statements of the block might precede a fetch of the containing expression on which it had a side effect. Since MINILAX uses collateral evaluation this would not be a problem in most cases, but suppose that the block were in the second operand of and and the code it preceded were in the first operand...

To avoid this problem, it would be necessary to code and discard (in appropriate order) all trees representing expressions in progress at the point that a block was entered. But the condition of a while is in progress at the point when a block making up the body of the while is encountered, and yet we want to delay that computation and place it textually after the body (see Figure 3.9d of the text). (It will be executed before the body, and hence the side effects will not interfere.) It seems clear that the problems of guaranteeing correctness are very difficult to solve, and hence not worth tackling until the need for smaller trees is proven.

A statement-oriented language like Pascal avoids the problems. Each condition and statement can be coded and discarded separately in such a language.