1) If it is impossible to balance the types of the two expressions in Rule A_4_6_2, the result is *sInteger*. Therefore it fails to verify adherence to Section 1 of the MINILAX definition, which requires that the result be void in that case. Section 3 says nothing about balancing at all. One could therefore argue that the attribute grammar is *incorrect* in the light of Section 3.

2) Change the default result of *fBalance* from *sInteger* to *sVoid*.

3) *tDefTable* and *tPoss* together implement the domain *Env* as a list of the (argument,value) pairs. Such lists are often used in mathematics to define a function. One example from denotational semantics is the definition of the least fixed point. *tDef* implements the sum domain *Dval*, with *tConst*, *tVar* and *tUndef* implementing the component domains *Int*, *Loc* and *Undef* respectively. *tType* implements the sum domain *Eval*, with two extensions. One of these extensions reflects a difference between the definitions in Section 1 and Section 3, while the other is related to the different modes of operation between the compiler and the denotational definition.

Section 1 defines the result of an assignment as being the left-hand side reference (1.4.3), but Section 3 defines it as being the result of the right-hand side. Thus Section 3 never uses a location as an expressible value, but Section 1 does. It turns out that for MINILAX this distinction is irrelevant, and the denotational definition is slightly simpler as written. If we were dealing with LAX, however, the difference *would* be significant and the denotational definition would have to be changed.

The denotational definition is not concerned with the practical problem of detecting as many errors as possible in a single run, and consequently it just terminates when any type error is found. Termination is signaled by the appearance of a continuation that ignores any argument continuation and simply provides a string (the error message) as an answer. This behavior is appropriate for an interpretive system that is actually executing the program as it is checking it. A compiler, on the other hand, should continue with the static checking to find additional errors. *sBadType* is a marker that effectively carries the information that the value has already been found in error and any further errors involving it are superfluous.
4) The lifetime of \(x_{\text{Expr}.\text{type}}\) for the first operand of an operator extends over the processing of the second operand. Since the second operand may contain an arbitrary number of subexpressions, a stack is necessary. Therefore \(x_{\text{Expr}.\text{type}}\) cannot be implemented as a global variable. \(x_{\text{Block}.\text{type}}\) has a very short lifetime, however. Essentially all that it does is to transfer the type of its expression to its context. No other processing is done during that transfer, so the lifetimes of two instances of \(x_{\text{Block}.\text{type}}\) cannot possibly overlap. Therefore it can be implemented as a global variable.

The best implementation of \(x_{\text{Block}.\text{type}}\) is to simply remove it, using the stack that holds \(x_{\text{Expr}.\text{type}}\) to transfer the attribute from the lower expression to the upper. This analysis is rather involved, however, and GAG does not attempt it.

5) #289 implements the remote access to \(\text{area}\), #290 the remote access to \(\text{env}\), and #291 the remote access to the list of \(\text{area}_{\text{out}}\)’s.

6) It is quite difficult to make significant use of the activation record stack if there has been any transformation of the grammar. The reason is that the activation records no longer correspond precisely with the rules of the abstract grammar, and it is these rules that control the lifetimes of the attributes. Simple use of the activation record stack is possible in MINILAX. For example, the intrinsic attributes of terminals can be stored as local variables. In order to make use of local variables for synthesized attributes, the strategy is to declare the space for the attribute in the \textit{calling} routine and pass a \texttt{var} parameter to the routine that computes the attribute. Then when that routine exits, its parent can access the value. Inherited attributes can be passed as value parameters, without tying up local storage, if they are only used in one visit. Otherwise they too should be passed as \texttt{var} parameters to the children.