1) 

\[ P[[\text{declare } v: \text{integer} \begin{array}{c} \text{begin} \\ v \end{array}] = K[[\text{declare } v: \text{integer} \begin{array}{c} \text{begin} \\ v \end{array}] \ (\lambda i. \text{Undef}) \ (\lambda v'. \text{cases } (v') \]

\[ isTr() \to \lambda s. \text{inString}("\text{Program must yield an integer}") \] [ ]

\[ isInt(z) \to \lambda s. \text{inInt}(z) \] [ ]

\[ isVoid() \to \lambda s. \text{inString}("\text{Program must yield an integer}") \] [ ]

\[ K[[\text{declare } v: \text{integer} \begin{array}{c} \text{begin} \\ v \end{array}] = \lambda e. \lambda k. E[[v]] (D[[v: \text{integer}]] e) k \]

\[ = \lambda e. \lambda k. \text{E}[[v]] [v_1 \to \text{newloc}] e k \]

\[ = \lambda e. \lambda k. \text{cases } ([v_1 \to l_1] e)(v) \text{ of} \]

\[ isInt(z) \to k(\text{inInt}(z)) \] [ ]

\[ isLoc(l) \to \lambda s. (k(\text{inInt}(s(l)))) s \] [ ]

\[ isUndef() \to \text{inString}("\text{Undefined identifier}") \] [ ]

Therefore,

\[ P[[\text{declare } v: \text{integer} \begin{array}{c} \text{begin} \\ v \end{array}] = \lambda e. \lambda k. E[[v]] (\text{D}[[v: \text{integer}]] e) k \text{ of} \]

\[ isTr() \to \lambda s. \text{inString}("\text{Program must yield an integer}") \] [ ]

\[ isInt(z) \to \lambda s. \text{inInt}(z) \] [ ]

\[ isVoid() \to \lambda s. \text{inString}("\text{Program must yield an integer}") \] [ ]

\[ = (\lambda s. (\text{cases } (\text{inInt}(s(l_{1})))) \text{ of} \]

\[ isTr() \to \lambda s'. \text{inString}("\text{Program must yield an integer}") \] [ ]
The image contains a page of text that seems to be discussing programming concepts and denotational semantics. The text is partially visible and includes mathematical expressions and natural language explanations. Here is the reconstructed content:

```
isInt(z) → λs'. inInt(z)  []

isVoid() → λs'. inString("Program must yield an integer")  end) s)

= (λs. (λs'. inInt(s(l1)))) s

= (λs. inInt(s(l1)))
```

The text also contains sections labeled a, b, and c, discussing the structure and denotation of programs, simplification, and the role of compilation in understanding and translating programs. It mentions the use of definitions tables and phrase stacks or structure trees for evaluation.

The denotational definition has an environment that is propagated through the structure to establish the meanings of identifiers, and the compiler has a definition table that performs the same function in the same way. In the denotational definition, simplification leads to an expression containing only instances of the operations described by the semantic algebras that are characterized in Section 3.2 as being “provided by the target machine”, just as translation by the compiler leads to a program containing only target operations.

In order to simplify the expression further, we need the initial store. The compiler has produced a function from stores to answers, which is the object code. Further simplification will be carried out at execution time, when the object code is executed.

The assumption stated at the beginning of Section 2 of the definition is that the variable v will be given a value prior to execution. In other words, the initial store will contain a mapping l1 → n for some integer n.

2a) The definition table would be used to evaluate the cases expression of E[[I]]. The definition table contains the environment as a subset (it also specifies target information), and therefore specifies whether a particular identifier was declared by a variable declaration or an identity declaration, or was not declared in the current scope.

2b) The phrase stack or structure tree would be used to evaluate the cases expressions of E[[E1 IO E2]]. Information about the sub-domain of expressible values to which an expression belongs is independent of the particular values computed, and can therefore be determined by the compiler. This information is only of use during the processing of a single phrase, and should not be in the definition table. (If it were kept in the definition table then the size of the definition table would be proportional to the program length — an undesirable property.)

3) The voiding coercion is handled the same way in both of these definitions: A continuation is defined with a parameter that never appears in the body of the lambda expression. In the definition of E[[E1; E2]], the continuation for the whole expression is passed to the second expression, and the result is defined as the body of a
continuation that has \( v \) as a parameter. However, \( v \) does not appear anywhere in that body (since \( v \) never appears free in any of the definitions, it cannot be free in either \( E[E_2] \) or \( k \)). In \( E[if \ E_1 \ then \ E_2] \) there is no value for the continuation to be applied to, and therefore a void value must be manufactured. This manufacture does not constitute the coercion, however, since Section 1.4.2 defines voiding as “discard a computed value”. The actual discarding of the result of \( E_2 \) is described by passing it a continuation with a parameter \( v' \) that is not used in the body of that continuation.

4) The definition of \( E[E_1 \ or \ E_2] \) implements the short-circuit evaluation required by Section 1.4.3 by its use of its continuation \( k \). When the value of \( E_1 \) is \( true \), \( k \) is applied to \( true \) and \( E_2 \) does not appear in the evaluation at all. When the value of \( E_1 \) is \( false \), \( k \) is applied to the value resulting from the evaluation of \( E_2 \).