The above declarations of the standard environment are not syntactically valid in Mini-Triangle, and so cannot be introduced by processing a normal input file. In fact, these declarations are entered into the identification table using a method called `establishStandardEnvironment`, which the contextual analyzer calls before checking the source program.

Once the standard environment is entered in the identification table, the source program can be checked for any type errors. At every applied occurrence of an identifier, the identification table will be searched in exactly the same way (regardless of whether the identifier turns out to be in the standard environment or the source program), and its corresponding attribute used to determine its type.

□

5.2 Type checking

The second task of the contextual analyzer is to ensure that the source program contains no type errors. The key property of a statically-typed language is that the compiler can detect any type errors without actually running the program. In particular, for every expression E in the language, the compiler can infer *either* that E has some type T *or* that E is ill-typed. If E does have type T, then evaluating E will always yield a value of that type T. If E occurs in a context where a value of type T' is expected, then the compiler can check that T is equivalent to T', without actually evaluating E. This is the task that we call ***type checking***.

Here we shall focus on the type checking of expressions. Bear in mind, however, that some phrases other than expressions have types, and therefore also must be type-checked. For example, a variable-name on the left-hand side of an assignment command has a type. Even an operator has a type. We write a unary operator's type in the form $T_1 \rightarrow T_2$, meaning that the operator must be applied to an operand of type T_1, and will yield a result of type T_2. We write a binary operator's type in the form $T_1 \times T_2 \rightarrow T_3$, meaning that the operator must be applied to a left operand of type T_1 and a right operand of type T_2, and will yield a result of type T_3.

For most statically-typed programming languages, type checking is straightforward. The type checker infers the type of each expression bottom-up (i.e., starting with literals and identifiers, and working up through larger and larger subexpressions):

- *Literal:* The type of a literal is immediately known.
- *Identifier:* The type of an applied occurrence of identifier I is obtained from the corresponding declaration of I.
- *Unary operator application:* Consider the expression 'O E', where O is a unary operator of type $T_1 \rightarrow T_2$. The type checker ensures that E's type is equivalent to T_1, and thus infers that the type of 'O E' is T_2. Otherwise there is a type error.