1 Problems

1. Enumerate with examples, the different kinds of allocation in a block-structured language, with heap allocation.

   Solution: The three kinds of allocation in a block-structured language with heap allocation are: static (e.g., global variables), automatic (e.g., local variables) and dynamic (e.g., heap allocation through malloc-like calls).

2. Consider the following C fragment.

   ```
   int i;
   int a[10];
   
   for( i = 0; i < 10; i++)
      i[a] = i;
   ```

   Provide an explanation on whether or not the above code will compile and run correctly.

   Solution: A C compiler interprets the `a[b]` operation as: `(a + b)`. If neither `a` nor `b` are array pointers, then the operation will produce a compile time error, since `(a + b)` is not a valid L-value. Likewise, if `a` (or `b`) is an array pointer with `b (a)` being an integer offset, it could still be the case that the address referenced is improper and this results in a run-time error, although `(a + b)` is a valid L-value. In our case, `a` is an array pointer and `i` is a valid offset, so `(a + i)` points to a valid address and hence so does `(i + a)`. Consequently, the code fragment will compile and run correctly.

3. Provide an informal definition of the term type constructor. Enumerate (with one example each) 3 different types of type constructors that occur in a typical programming language.

   Solution: We define data types to be a set of values with an associated set of operations. Since a data type is a set, we can apply set operations to construct new types out of existing types. Such set operations are called type constructors. Typical type constructors include:

   (a) Cartesian Product. For example, `struct` in C.

   (b) Union. For example, `union` in C.

   (c) Subset. For example, the Subrange type in Pascal.

4. Informally describe what is meant by the term Unification in polymorphic type checking. Apply the rules of unification to deduce the types of all names in the expression `a[i] + i`, assuming that these types are not known.

   Solution: Unification is a pattern matching mechanism, used in programming language type checkers to deduce the types of unknown names.
Consider the expression \( a[i] + i \). Let \( a[i] \) have the type \( \alpha \) and \( i \) have the type \( \beta \), for some unknown \( \alpha \) and \( \beta \). Since \( i \) is used as an array subscript, it must be the case that \( \beta \) is \( \text{int} \). Likewise, since \( a[i] \) is an array-dereferencing operation, it follows that \( \alpha \) has type “array of \( \gamma \),” for some unknown \( \gamma \). If \( a[i] \) and \( i \) are correctly paired, as operands of the + operator, it must be the case that \( \gamma \) has type \( \text{int} \). □

5. Briefly describe the issues involved in the implementation of the \textbf{for} loop structure in a programming language, that make it different from the \textbf{while} loop structure.

\textbf{Solution:} In a typical implementation, the \textbf{for} loop has some of the following restrictions placed on its implementation, so as to increase efficiency:

1. The value of the loop-index cannot be changed within the body of the loop;
2. The value of the loop-index is undefined after the loop terminates;
3. The loop index, belongs to certain restricted types only, e.g., integer and subrange types.

These implementational quirks make the \textbf{for} loop structure different from the \textbf{while} loop structure. □