Question 1

For the purposes of our analysis we only consider assignment statements since only such statements may provide information for determining types of variables. For each assignment the dataflow analysis invokes the function to compute the type of the r.h.s. expression from the types of variables currently available (we assume such function is given to us, let's call it *exptype*(*exp*,*vartypes*)). Then it compares the expression's type t_{exp} against the current analysis info t_{var} for that variable and updates it roughly as follows:

- if t_{var} is "unknown," set it to t_{exp};
- if t_{var} is "inconsistent," leave it there;
- if t_{var} is some proper type and t_{exp} is a different type, set t_{var} to "inconsistent";
- if t_{var} is same as t_{exp}, happily leave it as is.

The lattice for a single variable consists of the following elements: the top, meaning "unknown: insufficient info to determine the variable's type," the bottom, meaning "inconsistent: no type is consistent with the variable's uses," and all legal variable types in the language, the set *Ts*. Variable types (as lattice elements) are incomparable with each other, so the lattice has height 2 and is similar to the integer constant propagation lattice. We assume that the function *exptype*() returns top or bottom as appropriate when it cannot determine a proper type for the expression.

To handle pointer variables, we assume that with each type $t \in Ts$ there is also type *pointer_to*(t)$\in Ts$.
To handle array variables, we assume that with each type $t \in Ts$ there is also type *array_of*(t)$\in Ts$.
These types are also incomparable with each other as lattice elements.*

Here is the formal definition of the analysis.

- per-variable lattice:
  - lattice elements: $Ts \cup \{\perp, \top\}$
  - ordering: $\perp \le t \le \top$ for all $t \in Ts$
  - meet function: $meet(t1, t2) = \perp$ where $t1 \neq t2$
    and is defined in a standard way otherwise: $meet(t, t) = t$; $meet(t, \top) = t$; $meet(t, \perp) = \perp$
- dataflow lattice: cross-product of per-variable lattices
  define $\mathtt{in}_S(x)$ to be the component corresponding to variable x in the dataflow lattice $\mathtt{in}_S$
- dataflow function is:

  $\mathtt{out}_S = \mathtt{in}_S$

  except that $\mathtt{out}_S(x)$ is modified as follows:†
  - for statement S: $x := exp$

    $\mathtt{out}_S(x) = meet(\mathtt{in}_S(x), exptype(exp, \mathtt{in}_S))$
  - for statement S: $*x := exp$

    $\mathtt{out}_S(x) = meet(\mathtt{in}_S(x), pointer\_to(exptype(exp, \mathtt{in}_S)))$
  - for statement S: $x[index] := exp$

    $\mathtt{out}_S(x) = meet(\mathtt{in}_S(x), array\_of(exptype(exp, \mathtt{in}_S)))$
- direction: forward
- iterative

* Handling pointers, arrays and records is not required for full credit.
† Note: I am using the lattice meet function here merely because it does what I want.

- initial value at the CFG's entry node: T for each variable
- operation at control-flow merges: the lattice meet

We could make this analysis non-iterative, but then we would miss some opportunities to infer types of variables that are used before being defined with an expression, such as formal arguments. In this example:

```
function test(x) {
    y := x;
    x := 1;
}
```

a non-iterative analysis would be unable to deduce the type of y because when its assignment is processed, the type of x is not known yet. After iterating once, the type for x and, therefore, y would be established. Note that "iteration" means "iterating over statements, in flow-insensitive manner,"

If the analysis is non-iterative, termination is trivial. For the iterative analysis, termination is guaranteed because the following two properties hold:

- the lattice is of finite height (2*number of variables), and
- the flow function is monotonic* (because that's the property of the meet operation which is used in the flow function).

Question 2

Consider a simple example:

```
loop
    ...
    i := i + c
    ...
end
```

where i is an induction variable (a "fundamental induction variable" in Muchnick's terms) and c is a constant. If we think about this expression in terms of a control-flow graph and program analysis, we have a cycle in the def-use graph for variable i. A def-use cycle can only arise in a loop, in which the value of one iteration is used to compute the value in the next iteration. We can also use the def-use graph to check that the value of i is not changed anywhere else in the loop (note that it can (and should) have other assignments, *outside* the loop) and to trace through more complex situations, such as:

```
loop
    a := 42;
    ...
    i := j + a;
    j := i;
    ...
end
```

Here is a sketch of the algorithm:

- Build a CFG and compute a use-def graph for all variables.

* Reminder: monotonicity of F means that for all $\mathtt{in}_1 \leq \mathtt{in}_2$, $F(\mathtt{in}_1) \leq F(\mathtt{in}_2)$.

- Find the cycles in this graph using any standard technique, such as strongly-connected components (SCC).
- For each SCC, examine all of the statements in it to determine that they are all of the form:

  x := y or
  x := y + I

  where I is a loop-invariant value. The value by which the variable changes can be computed by performing all of the arithmetic operations along a path through the cycle.
- Check that the value of a variable is changed by the same amount on all paths through the loop.* You can first check that the computation along all paths through the SCC produces the same result and then check that these computations execute on all paths through the loop body.

Note that this approach doesn't work directly with nested loops, because the backedge from the outer loop will bring statements into the SCC that aren't in the inner loop. It is easy to sidestep this problem by removing the surrounding backedges before computing the SCC.

Also note that with SSA, the algorithm sketched here works in pretty much the same way.

Uses of induction variable analysis:

- analysis of loop-carried dependencies, leading to loop/array optimizations and parallelization,
- variable range analysis (the values of the induction variable all lie between the loop bounds) and array bound check elimination,
- strength reduction.

* Not required for full credit for the problem.