Positive

- List of always "dead" or "live" statements
- A lot of examples/illustrations to help clarify your text
- Clear, short and precise report

Constructive

- Internal representation:
  - Consider alternative internal representation than the syntax tree when you do analysis. Restructure it to simplify or highlight e.g. program flow.
  - Have you considered how to handle "long statements". e.g.
    a = x op y op z op 4.
    Analysis could become non-trivial to code if this is translated to a tree like:
    op->{x,op->{y, op->{z, 4}}}
- Minor details regarding program slicing:
  - Your definition of program slicing as removing the parts of the program that does not affect the behaviour of the program at the point of interest seems at odds with the definition given in the assignment: removing those parts that does not affect the values computed. This difference is significant, since statements like “while-true-skip” is removed in your algorithm. Such statements affect behaviour, but does not affect which values may be computed.
  - Your algorithm seems to correctly include those statements that affect values computed, but not the statements affecting behaviour.
  - Discussing issues like non-termination may be a good idea.
- Constant folding calculated via RD (which is a "MAY" analysis), instead of as a "MUST" analysis.
  - On all paths to l, the variable x "MUST" have the value c, otherwise we are not allowed to substitute it.
  - Alt. use "Use-Define"/"Define-Use" chains for this transformation, we believe they will work if used properly. (if this was your intention all along, we feel it has not been made clear).

Notes

- Program slice tables/examples lacks proof reading.
  E.g.: y := pi * r * r; becomes y := 2 * r * r;
- "Table 1" (in 2.1) was found to be rather confusing. Your text suggests that only the "conditions" plus the statements affecting "interesting" values are part of the program slice. We would recommend that you either made your definitions more clear or you corrected your examples (e.g. x > 0 -> skip, here only the x>0 is interesting and not the skip part as far as we understand your definition.)
- "read x" always has a side-effect even if the variable is not used.
  - If it does not receive a valid integer input it will abort.
    (possible abort != skip)
- Avoidance of handling of guarded commands: Your approach is safe, but may risk an analysis that is less precise than what it could be.
  - Program analysis: it may be possible to avoid unreferenced expressions in a do.