Name ______________________ Student No. ______________________

No aids allowed. Answer all questions on test paper. Use backs of sheets if necessary.

Total Marks: **60**

[20] 1. In the context of verification of software, explain the difference between *inspection*, *testing* and *formal verification*. What are the two main benefits of *formal verification*?

Solution: See the handout on formal verification. Main ideas are that inspection and testing are "existential" (they discover error) but formal verification is "universal"—it produces a demonstration of correctness. Formal verification, besides a direct proof of correctness, also makes explicit all the implicit assumptions that the programmer makes; the "theorems" that appear in the proof are assumptions about the environment where the program will run, and these "theorems" (or "assertions") may aid with portability.

[20] 2. Explain the semantics of the "While" rule for program verification:

$$\frac{\{\alpha \wedge \beta\}P\{\alpha\}}{\{\alpha\}\ \textbf{while}\ \beta\ \textbf{do}\ P\ \{\alpha \wedge \neg\beta\}}$$

Solution: This rule is saying the following: suppose it is the case that $\{\alpha \wedge \beta\}P\{\alpha\}$. This means that P is (partially) correct with respect to precondition $\alpha \wedge \beta$ and postcondition α. Then the program "**while** β **do** P" is (partially) correct with respect to precondition α and postcondition $\alpha \wedge \neg\beta$ because if α holds before it executes, then either β holds in which case the while-loop executes once again, with $\alpha \wedge \beta$ holding, and so α still holds after P executes, or β is false, in which case $\neg\beta$ is true and the loop terminates with $\alpha \wedge \neg\beta$.

[20] 3. Suppose that the design decision in a software project was to implement the dynamic programming solution to the "simple knapsack problem" where the array of partial solutions is given as follows:

$$R(i,j) = T \iff [R(i-1,j) = T \vee (j \geq w_i \wedge R(i-1,j-w_i) = T)].$$

(a) What is an implementation danger? (Hint: mention "lazy evaluation.")

(b) What would be the two natural stages of "prototyping"?

(c) Explain the "space-saving" technique in implementing the array.

Solution: An implementation danger is that for some i,j, $R(i-1,j) = F$, so the program moves on to check if $j \geq w_i$ and $R(i-1,j-w_i) = T$; if $j < w_i$, then by "lazy evaluate" the checking should end right here, rather than go on to $R(i-1,j-w_i)$ where an "out of bounds error" will arise.

The two natural stages of prototyping: first have a natural implementation where we keep a 2-dimensional array (with proper initializations for $R(0,*)$ and $R(*,0)$), and then to improve the implementation to a 1-dimensional array as in the following program; this is the space saving technique:

```
S(0) ⟵ T
for j : 1..C do
  S(j) ⟵ F
end for
for i : 1..d do
  for decreasing j : C..1 do
    if (j ≥ w_i and S(j − w_i) = T) then
      S(j) ⟵ T
    end if
  end for
end for
```

Note that the loop has to be executed in a "decreasing" order, to make sure that we get the proper values (from the "$i-1$-level"). Note that once an entry is T it will stay T until the end; hence there is no need to check $R(i-1,j) = T$.