Well-formed Formulas for Program Proving

To discuss program proving, we must first establish a language in which programs are written. We will pursue this in the context of what is usually referred to as *imperative programming* — the traditional paradigm whose primary focus is on the familiar assignment operation. In fact, the basic proving techniques can be applied in a variety of imperative languages (e.g., Pascal, C, C++), and we explore the ideas for a generic version of the facilities common to imperative languages rather than for one specific language.

The "formulas" we write will make claims about the properties of programs (program fragments actually). Therefore, they must include both the claim and the program to which it pertains. Our **program assertions** (wffs) have the form

$\{\Box\}\ P\ \{\Box\}$

whose three components consist of:

- P is a program fragment — a (possibly compound) "statement", and
- $\Box$ and $\Box$ are predicate logic formulas constructed using the variable and function/operation names of the program under consideration; the symbols {, } are meta-symbols used to denote the beginning and end of predicate logic formulas, and should not be interpreted as symbols in the programming language. The logic formula $\Box$ is called the **pre-condition**, and $\Box$ is called the **post-condition**.

As with traditional logic systems, program assertions have dual interpretations — a truth-theory interpretation, and a proof-theory interpretation. While the truth-theory interpretation is what we seek to establish, it is intractable to do this. However, we find we can establish much of what we seek using proof-theory methods.

Interpretation of Program Assertions
The truth-oriented interpretation of a program assertion {□} P {□} is:
if the pre-condition □ is true for the values of the program variables when the execution of program fragment P is initiated, and P eventually halts, then the post-condition □ will be true for the values of the program variables when P halts. This is referred to as the **partial correctness** of P (it is vacuously true when P does not halt) since it offers no conclusion one way or the other about P halting.

The essence of a program assertion {□} P {□} is a "contract" that expresses the results of the computation performed by the fragment P. If P is partially correct with respect to pre-condition □ and post-condition □, then we are assured that whenever P in initiated with values that satisfy □, upon termination the values will satisfy □. Pre/post-conditions provide a specification that is independent of program structure and algorithm.

For instance, if P is partially correct with respect to pre-condition $X \geq 0$, and post-condition $Y^2 = X$, then P is assured to be a program that correctly computes $Y = \sqrt{X}$ (provided it halts and doesn't change X).

The caveat requiring program termination as a premise is undesirable, but unavoidable. A program fragment that is partially correct and is assured to halt when started in any state that satisfies the pre-condition is said to be **totally correct**. We shall see a little later that entirely different techniques are required to establish termination than those that permit proving partial correctness.

Since program variables can generally take on an unlimited collection of values, partial correctness satisfaction cannot be exhaustively determined. This makes it impossible to establish partial correctness by truth analysis. Our objective will be to develop proof techniques that are based on "pattern matching" manipulation of program assertions, that can be mechanically performed and allow us to conclude the truth of partial correctness assertions.