ALTERNATIVE ALGORITHMS: Equivalence

In computing, as with many areas, there are often many ways to do things. This leads to a choice to be made among the various ways. For example, consider the previous simple Gross Pay algorithm, which is repeated in figure 1. Another algorithm next to it, in figure 2, shows an alternative way to compute this pay. First the hours are input and pay is computed by multiplying these hours H by the regular rate of 10. In the example case of 50 hours, this yields 50*10= 500 dollars. Then the hours are compared to 40, and if they are greater than 40 then those hours over 40 are multiplied by 5 (half the rate of 10), and this total is added to the above pay. If the hours are not greater than 40 then nothing is added to the original "partial" pay. In this example case the (50 - 40) or 10 hours of overtime are multiplied by the 5 extra dollars per hour yielding 50 additional overtime dollars to add to the above 500 dollars (total of 550 dollars).

Equivalence of Algorithms

These two algorithms are different in structure, but they are equivalent in behavior. Which do you prefer? Why? In this example there is no serious reason to prefer one algorithm over the other. However in some cases one algorithm may be considerably smaller, or faster, or simpler, or more reliable than another. The significance is that one could be substituted for another, like a spare part or module. This "plug-in" replaceability of modules or building blocks can be quite useful.

More Equivalence

Let us consider a more complex algorithm, such as the extended GrossPay algorithm, repeated in figure 3. An algorithm equivalent to this extended pay is shown in figure 4. The equivalence may not be so clear in this case. Let us try this algorithm for an input of H = 100 hours. First P is computed from 10*H as $1,000. Then the first choice (with 100 > 40) adds to P the amount 5*(100 -4 0) or $300. The second choice (with 100 > 60) adds the amount 5*(100 - 60) or $200. Finally the output is the sum of these three amounts (1000 + 300 + 200) or $1500. This amount agrees with the value computed for the other algorithm at the left, which is:

```
P = 700 + 20 * (100 - 60) = 700 + 800 = 1500
```

It is extremely important to realize that comparing outputs for one single input value is not sufficient to determine equivalence for all values! One input value yielding different outputs for two algorithms would prove that the algorithms were not equivalent. More values must be tested to show equivalence.

Testing

In this particular example there are only a few ranges of values which could be used to test the algorithms. Figure 5 shows how the input values could be split into five distinct ranges. Input values less than zero and those greater than 168 indicate errors. Values from 0 to 40 (inclusive) indicate the regular range, values above 40 and up to 60 indicate time-and-a-half range, and values above 60 to 168 indicate a double-time range. So to compare these algorithms we could take a typical test value from each of these ranges (say 20, 50, and 100). These three values would trace through all possible paths on the flow charts. Other equally good test values are 30, 55, and 150, or 5, 45, and 65. It could also be useful to test some critical values, such as 0, 40, 60, and 168. Figure 6 is a table of test values which show identical outputs for all input values, so the algorithms are equivalent. It is not always easy to test algorithms in this way, for there may be thousands or billions of possible paths in some programs.

Now that the two algorithms are equivalent; which is best? There is not a great difference between them, but most people feel that the second is simpler and clearer. It consists of a "long and thin" series of smaller decisions, which usually appears simpler than a "nesting" of short-and-fat decisions.

The most important idea here is that there is a separation of WHAT is to be done from HOW it is to be done. There may be many different ways to do the same thing. It is important to separate the definition part of an algorithm (which shows Why and What) from the implementation part (which shows How and Where).