Solution Set 2

DATE: 22 SEPTEMBER, 1992

1. Problem 2.1.2: Show that the flow diagram of every while-program is a flowchart program. The flow diagram of any while-program is the connecting together of squares and diamonds. The labels in each box are allowable under the rules of flowchart programs and so are the ways in which the boxes are connected together. So every such flow diagram is a flowchart program.

   However, not every flowchart program is the flow diagram of a while-program. In particular, in a while-program the flow of one branch of any diamond box must always return to just above that box. On page 24, 2(b), we have an example of a flowchart program which breaks this rule, and therefore can never be the flow diagram of a while-program.

   I can fill in the boxes of the flowchart on page 24 so that an equivalent while-program is simply,

   ```
   begin
     y := succ(x) ;
     y := pred(y)
   end
   ```

   The two diamonds are labeled $x \neq y$; the first square is labeled $y := 0$; the second square is labeled $y := succ(y)$.

2. Problem 2.1.3: Show that flowchart programs and goto-programs are intertranslatable. What you must be careful about in this question is to show a goto-program whose flow diagram is *exactly the same* as any given flowchart program. To do this we first must connect to the goto statement an acceptable graphic. For this, just draw an arrow from the statement before the goto to the statement which is the target of the goto. Now it is evident that every goto-program is a correct flowchart program: it is built out of the correct components in the correct way.

   It is harder is to show that any flowchart program is realizable by the flow diagram of a goto-program. Given a flowchart program, label each box $1, 2, \ldots, n$ such that the entry point is labeled 1. Write a goto-program,

```
begin
1: S1 ;
2: S2 ;
...
n: Sn ;
n+1: Sn+1
end
```

where each Si is determined as follows. If box i is labeled with the assignment $X := 0$ and has an outgoing arrow to statement j, then Si is,

```
begin X := 0 ; goto j end
```

and similarly for the other assignment statements. If box i is a diamond labeled $X \neq Y$ and has the true outgoing arrow to statement t and the false outgoing arrow to statement f, then Si is,

```
begin while X <> Y do goto t ; goto f end
```

The statement Sn+1 simply serves as a common exit point for the multiple exit points of the flowchart program and can be,

```
begin end
```

3. Problem 2.2.1: In the following programs, `--` is monus, `<>` is $\neq$, etc.

   (a) $Z := X * Y$

   ```
   begin z := 0 ; while y <> 0 begin
     z := z * x ; y := pred(y) end
   end
   ```

   (b) $Z := X$ div Y,

   ```
   begin z := 0 ; while x >= y do begin
     x := x -- y ; z := succ(z) end
   end
   ```

(c) $Z := X \bmod Y$,

```
begin while x >= y do x := x -- y ;
  z := x end
```

(d) $Z := X ** Y$,

```
begin z := 1 ; while y <> 0 do begin
  z := z * x ; y := pred(y) end
end
```

(e) $Z := 2 ** X$,

```
begin y := x ; x := 2 ; z := x ** y end
```

(f) $Z := \log_2(X)$,

```
begin z := 0 ; x := x div 2 ;
  while x <> 0 do begin
    x := x div 2 ; z := succ(z) end
end
```

4. Problem 2.2.2: Write macro definitions for **if-then-else** and **repeat-until** constructions. A little care helps avoid conflicts between the evaluation of the test condition and that of the statements. Assume the test C returns 1 for true and 0 for false. Replace,

```
if C then S1 else S2
```

with,

```
begin test := C ;
  while test = 1 do begin
    S1 ; test := succ( test ) end ;
  while test = 0 do begin
    S2 ; test := succ( test ) end
end
```

Replace,

```
repeat S until C
```

with,

```
begin S ; while not C do S end
```

5. Problem 2.2.3: Show that the pred operator is not required for a fully-powerful while-program language. The following macro which computes pred using only succ, zero assignment and while $X \neq Y$ statments is due to Joe Tano. Replace each,

   ```
   z := pred( x )
   ```

   with,

   ```
   temp := 0 ; z := 0 ;
   while z <> x do
     begin temp := z ; z := succ(z) end
   ```

6. Problem 2.2.4: Show that we could get by with just while $x \neq 0$ in our while-program language. Replace each,

   ```
   while x <> y do S
   ```

   with,

   ```
   begin test := ( x -- y ) + ( y -- x ) ;
     while test <> 0 do begin
       S ; test := (x--y)+(y--x) end
   ```

   We must verify that test is non zero exactly when x does not equal y and that test can be calculated using only while x not equal 0 loops. In fact, the macros in the book for addition and monus test only for non-equality to zero, so they are acceptable in this new language.

7. Problem 2.2.5: Show that goto and while-programs are equivalent and that, as an interesting corollary, only depth-2 while programs are really necessary.

Covert the goto-program to a flowchart program. Number all the boxes in the flow diagram and build for each box i a statement S_i which executes the assignment or test contained in the box and sets a counter to the number of the next box to execute. For the test boxes (diamonds), two possible settings of the counter are possible. Build a while-program from these statements S_i by setting the counter initially to 1, and looping until the counter is $n + 1$, where n is the total number of boxes. This is the termination condition.

The body of the loop contains if statements **if** counter is i **then** S_i, this for each $i = 1, \ldots, n$. This completes the proof that goto-programs are no more powerful than while-programs. Conversely, each while-program is a goto-program, so that while-programs are no more powerful than goto-programs.

To show that only depth-2 while programs are necessary, we carefully improve our simulation to be only two deep. We have already used one while-loop to check if the counter is $n + 1$. Inside this we are allowed one more level of while-loops. The remainder of the proof is simply a verification that we can squeeze by with so small a ration.

First, we do not have if-then-else. We must use while $x \neq y$. So each if-then-else is proceeded by the calculation of a test variable t which is True (0) if and only if counter equals I, where I is any integer,

```
    t := ( counter -- I ) + ( I -- counter )
```

We must verify that test can be calculated using only one level of while-loops. In fact, the macros in the text for monus and addition use only one level of while-loops. We can then process assignment statements as,

```
    t := ( counter = I ) ;
    while t<>False do begin t := False ;
      assignment statement ;
      counter gets next box number
    end ;
```

To be really precise, True and False would be set up as variables,

```
True := 0 ; False := succ(True) ;
```

and setting t false is accomplished `t := succ(True);`.

BUT, we cannot do the obvious to simulate while-boxes. This would lead us into another level of while-looping. We merge the while-decision into the test variable t,

```
tTrue := ( counter = I ) AND ( x <> y ) ;
tFalse := ( counter = I ) AND ( x = y ) ;
while tTrue<>False ... etc.
```

Once again, we must verify that the macros in the book require just a single level of while-loops to accomplish logical AND as well as turning the condition $x \neq y$ into a variable setting.

```
t := (counter=I) ;
xtemp := x ;
tFalse := t ; {that is, true if counter=I}
tTrue := succ(0) ; {that is, false}
while xtemp<>y do begin
  tFalse := succ(0) ;
  tTrue := t ;
  xtemp := y ;
end ;
```