QUIZ 3

Date: 1st April 2003 Max. Marks 40

1. Given is a grammar for a switch statement.

   $switchstat \longrightarrow$ `switch` exp `begin` $sclauses$ `end`
   $sclauses \longrightarrow sclause\ sclauses \mid sclause$
   $sclause \longrightarrow const :\ statement$

   We would like to generate 3-address code for switch statements. The nature of the code that we would like to generate is illustrated below:

   ```
   switch E                    1. code to evaluate E in t
      begin                    2. if t <> V1 goto 5
        V1: S1                 3. code for S1
        V2: S2                 4. goto 9
      end                      5. if t <> V2 goto 8
                               6. code for S2
                               7. goto 9
                               8. skip
                               9. ...
   ```

   Give L-attributed definitions to generate 3-address code for switch statements. Assume that as code is being generated, it is being dumped into a global data structure. Use the familiar functions gen (or $emit$), $backpatch$, $nextstat$ etc. You can also use other functions, but explain what they do. Assume that code has already been generated for expressions and other statements, and that statements *are not required to* produce code containing unfilled gotos.

   Before answering this question, clearly describe all the attributes that you will be using.

   Soln:

   (a) $next$ is an attribute containing statement numbers of unfilled gotos. These gotos will be backpatched after all the clauses have been seen. They will be targeted to the statement following the switch.

   (b) $expname$ is an attribute containing the name of the variable which contains the value of the switch expression. This will be passed to each clause enabling it to generate the statement `if t <> V2 goto ...` ($expname$ being `t` in this case).

$$
\begin{array}{lll}
switchstat & \longrightarrow & \texttt{switch}\ exp \\
& & \quad \{sclauses.expname := exp.name\} \\
& & \texttt{begin}\ sclauses\ \texttt{end} \\
& & \quad \{backpatch(sclauses.next, nextquad)\} \\
sclauses_1 & \longrightarrow & \quad \{sclause.expname := sclauses_2.expname := \\
& & \quad\ sclauses_1.expname\} \\
& & sclause\ \ sclauses_2 \\
& & \quad \{sclause_1.next := sclause.next \cup sclause_2.next\} \\
sclauses & \longrightarrow & \quad \{sclause.expname := sclauses.expname\} \\
& & sclause \\
& & \quad \{sclauses.next := sclause.next; \\
& & \quad\ gen(skip)\} \\
sclause & \longrightarrow & \quad \{tmp := nextquad; \\
& & \quad\ gen(if\, sclause.expname \neq const.literal goto\_)\} \\
& & const:\ \ statement \\
& & \quad \{sclause.next := nextquad; \\
& & \quad\ gen(goto\_); \\
& & \quad\ backpatch(tmp, nextquad)\}
\end{array}
$$

2. Shown below is the gcd function written in Pascal augmented with C-style blocks. Assuming that the function is being called from the main program as `gcd(x,y)`, where x is 6 and y is 9, draw the *entire stack of activation records* at the point when the stack has grown to its fullest.

   The contents of the activation record that you have to show are (i) static link (ii) dynamic link (iii) return address (iv) return value (v) parameters (vi) locals, and (vii) block variables. The order of these in the activation record is not important. Try to keep the stack size to the smallest possible.

```
function gcd (var a, b: integer): integer;
  var e : integer;
  begin
    if a=0 then
      blockbegin
        var d: integer; /* d and e are useless except */
        begin           /* for the purpose of the quiz */
          d := b;
          gcd := b
      blockend
    else
      blockbegin
```

```
          var c: integer;
          begin
            c := b mod a;
            gcd := gcd(c,a)
          end
        blockend
    end
```

Soln:

3	d3	
	e3	
BPgcd2	dyn link	
@	ret addr	gcd3
BPmain	stat linK	
3	ret val	
3	b3	
&c2	a3	
0	c2	
	e2	
BPgcd1	dyn link	
@	ret addr	gcd2
BPmain	stat linK	
	ret val	
6	b2	
&c1	a2	
3	c1	
	e1	
BPmain	dyn link	
#	ret addr	gcd1
BPmain	stat linK	
	ret val	
9	b1	
&x	a1	
		main
9	y	
6	x	

Note that:

(a) `#` and `@` have been used to mark the return addresses in `main` and `gcd`.

(b) The topmost slot in the activation record has been used to store both `d` and `c`. They can share the same slot because they have disjoint scopes (and extents).