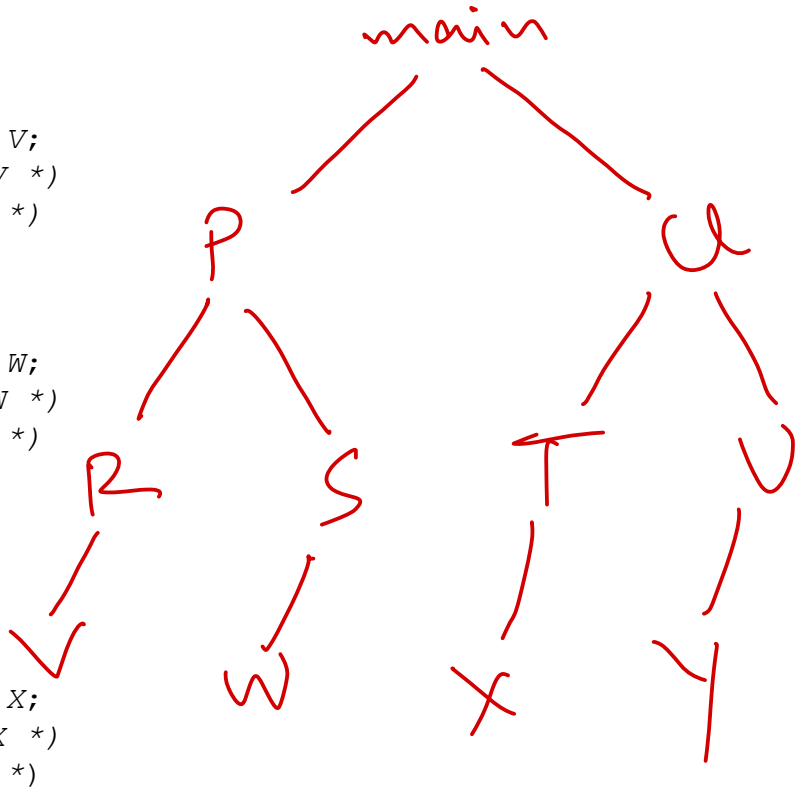
Nested Procedures

Languages such as Algol60, Algol68 and Pascal support nested procedures. (OCaml supports nested function definition). This is a natural consequence of the principles of qualification and abstraction, even without first-class procedures. After all, if we can define functions, why can't we qualify a command or a block with function definitions (apart from constant definitions, type definitions, and variable definitions). Nested function/procedures provide a pleasant degree of modularity, abstraction and a clear naming discipline — one can locally define helper procedures within a procedure, without the outer procedure having to be burdened by the information about all such nested procedure definitions.

Consider the following outline of a Pascal program (without the commands, but only the structure), where we have nested procedure definitions. (We shall go to 3 levels in our lesson).

```
program main
      procedure P;
            procedure R;
                  procedure V;
                  begin (* V *)
                  end; (* V *)
            begin (* R *)
            end; (* R *)
            procedure S;
                  procedure W;
                  begin (* W *)
                  end; (* W *)
            begin (* S *)
            end; (* S *)
      begin (* P *)
      end; (* P *)
      procedure Q;
            procedure T;
                  procedure X;
                  begin (* X *)
                  end; (* X *)
            begin (* T *)
            end; (* T *)
            procedure U;
                  procedure Y;
                  begin (* Y *)
                  end; (* Y *)
            begin (* U *)
            end; (* U *)
      begin (* Q *)
      end; (* Q *)
begin (* main *)
end; (* main *)
```



Now the idea is that the main program can in its body (between the begin and end of main) make procedure calls to the procedures defined immediately within it, namely `P` and `Q`. (Otherwise what is the point in defining these procedures?) Likewise, in the body of procedure `P`, the procedures defined immediately inside it, namely `R` and `S`, can be called. Similarly, in the body of procedure `Q`, the procedures defined immediately inside it, namely `T` and `U`, can be called.

But the *main* program cannot directly call the procedures defined inside `P`, namely `R` and `S`, and within `Q`, namely `T` and `U`. Of course each of these can be called indirectly, by calling the procedure in which they have been defined, and within those procedures, these can be called.

The next reasonable requirement is for procedure `P` to be able to call itself. Recursion is a fundamental and natural idea in programming.

But can procedure `Q` call procedure `P` ? Since these definitions are made *in parallel*, this seems perfectly reasonable — in fact, it is quite natural to have two mutually recursive procedures/functions being defined as helper procedures/ functions within the main program or a procedure. (Note: If, as in Pascal, `Q` could call procedure `P` but not *vice versa* — unless a special mechanism of forward declarations was used, it was not for any good theoretical reason, but only to simplify writing the compiler. In C, this unnecessary — and silly — restriction was dispensed with.).

So so far, the rules about procedure calls say: (1) A procedure can call procedures defined immediately within it (let us call these "children" — if we were to draw a tree to represent the nesting, we would see these as children) (2) A procedure can call a procedure — including itself — that is defined at the same level as itself, i.e., within their common "parent").

More interestingly, in the body of procedures `R` and `S`, one should be able to call procedure `P`, since its name appears naturally in the scoping rules that apply to procedures `R` and `S`. So we can add yet another rule: (3) A procedure can call its parent.

Now if procedures `R` and `S` can call `P`, is there any reason that they shouldn't also be able to directly call `Q` — which is defined in parallel with `P` ? So (4), a procedure should be able to call a sibling of its parent.

Taking this further, we can equally argue that since the name `P` appears in the scoping rules that apply to procedures `V` and `W`, they should be able to directly call `P` (their "grandparent") in their bodies. Generalising from rules (2) and (4), a procedure should be able to call any *direct ancestor* (including itself — counted as its level 0 ancestor; its parent — level 1 ancestor; and so on), but maybe not the root (i.e., the main program).

Now, we can ask whether the name `Q` appears in the scoping rules that apply to procedures `V` and `W`, and indeed one can reasonably argue that indeed it does (if one agrees that `P` and `Q` are defined in parallel, and so any procedure that can "see"

one can see the other too. So (5) a procedure should be able to call any sibling of any of its *direct ancestors*.
Generalising rules (1), (3) and (5) — as well as (2) and (4) as special cases, one can formulate a general rule: "a procedure should be able to call any *child* of any of its *direct ancestors*". (This also nicely avoids allowing the main program to be callable.)

Complementarily, we specify (0) "a procedure *cannot* directly call any other procedure". In particular, a procedure cannot call a procedure defined within a child procedure. Or any descendant other than an immediate child. Or any child of a sibling procedure, etc.

So in the nesting tree, if from procedure A one can go $n \geq 0$ levels *up* and then 1 level down to reach the definition of procedure B, then A can call B directly since it is "visible" according to the scoping rules.

Exercise: Can procedure Y call procedure W directly? Can procedure W call procedure Y directly?

Exercise For the example program layout, make a matrix to indicate which procedures can directly call (or not call) which others. What are the entries on the main diagonal?

"Global" Variables

In Algol-like languages, "global variables" are only relatively global… these are those variables which are declared in an outer (enclosing) scope. In fact, the notion of visibility is that anything defined in an outer scope should be visible within a nested scope — unless there is an intervening definition for that definition that redefines it (eclipsing the outer definition). The foregoing discussion we had about whether a procedure can call another one directly is actually also an instance of this same visibility rule.

Let us now revisit the (a lopped version of the) example procedural graph, and now endow the procedures with both formal parameters and local variables. For the moment, we will not mention the types of the variables — let us assume that all variables are of integer type.

```
program main
     var  x, y;
     procedure P(a, b);
          var c, d;
          procedure R(y);
               var v, w, z;
               procedure V(x);
               var z, a;
               begin (* V *)
                    z := x;   a := y; w := b
               end; (* V *)
          begin (* R *)
               d := y; v := 9; w:= 2; V(v);
```

```
        end; (* R *)
    begin (* P *)
        d := y;  c:= 7; R(c);
    end; (* P *)
begin (* main *)
      x := 4; y := 5; P(x,3)
end; (* main *)
```

The important point to remember is that with respect to any procedure, the Principle of Correspondence gives equal status — and the same access mechanism — to both the formal parameters and local variables. Both will be treated the same by the allocation mechanism. The only difference is that formal parameters will be given values from the calling context (although the procedure can change these contents) whereas local variables are set within the procedure.

Exercise: Determine the binding occurrences of each (bound) variable that appears in the body of a procedure. Now trace the execution of this program.

The order in which definitions appear in Pascal

Pascal follows a particular order in which constant definitions, type definitions, variable definitions and procedural definitions can appear.
First, constants are defined. This makes sense, since constants can be pre-evaluated at compile time. Moreover, the main use of constants is to fix certain parameters in the program such as array lengths (one innovation in C over Pascal was to leave array sizes unspecified, and letting these be determined by the context.)
For example, a Pascal constant definition is of the form

```
const N = 100;
```

Now such constant definitions could be followed by a type definition which depended on the constants, such as

```
type ids = array[1..N] of char;
```

If one decided that identifiers should be 200 characters long instead, one only needed to redefine the constant `N` ... and recompile the program.

Now since Pascal required the types of variables to be specified when declaring them, variable declarations needed to follow type definitions.

```
var first, last: ids;
```

Finally since procedures (and so-called functions) could access and manipulate variables defined in the enclosing procedure, their definitions needed to follow variable defintions.