Identifiers

- We can now implement type information, but we need to determine which identifiers are valid (and their meaning) in our program.

  — Again, we use the AST:
  

```
Program test(output);
  var bar, testbar : integer;
  procedure foo;
    var bar : integer;
    begin (* foo *)
      bar := 10;
      testbar := 10;
    end; (* foo *)
    begin (* test *)
      foo;
    end.
begin (* test *)
  foo;
end.
```

The Identifier Table

- To access the information needed, we need an auxiliary data structure
  — We need to know what things are in scope

- Aside: Scope rules
  - The scope rules of the language define the referencing environment the module sees.
  
  - Pascal has a scope environment that chains back through the nested modules
    — In the example, the local identifier `bar` and the main program’s `testbar` identifiers are in scope inside procedure `foo`.
    — The identifier table also needs to implement the scope rules of the language.
  
  - Observation: number the scope regions, the scopes `nest`.
    — We have only one thing contributing to the referencing environment at any given level
    — Our scope regions are therefore a simple linear list of blocks in scope.

---

**Scope**

```
Program Scope(output);
  var bar, testbar : integer;
  procedure one;
    var bar : integer;
    procedure two;
      begin (* two *)
        bar := 1;
      end; (* two *)
      begin (* one *)
        bar := 10;
      end; two;
      testbar := 10;
      begin (* one *)
        bar := 10;
      end; one;
      end; (* three *)
    begin (* scope *)
      three;
    end.
  end.
begin (* scope *)
  three;
end.
```
Scope

- Note carefully how the local declaration of `bar` in procedure `one` hides the more global version in the main program.
  - But the other non-hidden entities are still visible, of course.
- Implementing this is not very hard!
- Look at the compilation process:
  - We see, in order, the following blocks:
    - `Scope, one, two, one, three, scope`
      - Including the bodies!
  - While compiling `one`, we need `Scope` and `one`.
    - While doing `two`, we need `Scope, one` and `two`.
    - While doing `three` we need `Scope`...
  - We can implement this by using an array structure where the $n^{th}$ element has the module at scope level $n$
    - We simply search this from end to beginning!

The Identifier Table

- As we traverse the AST, we need to resolve each identifier
  - build up the `idtable` array as we enter each new scope region.

Using the IDtable

- We discover an entity is *undeclared* by falling off the start of the `idtable` array
  - At this point, we need to enter the id into the `ast` with type set to `unknown`
    - Prevent spurious errors from happening on every future reference
- Once we have determined which identifier it is, the `lookup()` method should return a reference to the identifier.
- This reference should be written into the tree so that when we perform further work on the tree, we can directly locate what entity is being referenced.
  - You don’t strictly need to do this, of course.
    - We could just use the `lookup()` method again, but this is somewhat wasteful
- IMPORTANT – our AST is now a DAG