DOES BLOCK = SCOPE IN PASCAL

T. P. Baker[*]
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
and
A. C. Fleck
Department of Computer Science
and
Weeg Computing Center
University of Iowa
Iowa City, Iowa 52242

INTRODUCTION

There seems to have developed some controversy over whether the scopes of identifiers are (or should be) synonymous with blocks in PASCAL. In this note we call attention to the formal statement of the "rules" dealing with this situation, point out several other items in the literature that address the question of the title, and present our own personal conclusions. We relate our comments first with respect to "Standard" PASCAL, and then to the new BSI/ISO Working Draft Standard PASCAL.

WIRTH'S STANDARD PASCAL

There are several levels of documentation to consider in this case, in decreasing order of abstraction: the Report [2], the User Manual [2], and the several E.T.H. compilers. Arthur Sale in [3] argues strongly the position that scope = block. But we would like to suggest that there are loopholes. The Report is unfortunately vague. In section 10 we are told that scope = procedure (or function) declaration and that identifiers are not known outside their scope. But it gives no details of how they arre known inside their scope. The crucial issue is for nested scopes which are mentioned in section 2 but for which no rules are given. Section 4 of the Report tells us that the association of an identifier must be unique within its scope. This is essentially the extent of the specification in the Report. In this light we consider the following example:

```
PROGRAM P1(OUTPUT);
  PROCEDURE Q; BEGIN WRITELN(1) END;
  PROCEDURE R;
    PROCEDURE S; BEGIN Q END;
    PROCEDURE Q; BEGIN WRITELN(2) END;
  BEGIN S END;
BEGIN R END.
```

Now there are two definitions provided for identifier 'Q' within nested scopes. The one within R must not be known outside R. There is only one invoking instance of the identifier 'Q' (hence its association must be unique) and its occurrence is validly within both scopes and the Report's rules give us no reason for preference.

[*] Present address: Mathematics Department, Florida State Univ., Tallahassee, FL 32306

Next we consider the User Manual. Here in Chapter 1 (pp. 6-7) we find it again stated that scope = procedure declaration. Also it is stated "the scope or range of an identifier x is the entire block in which x is defined, including those blocks defined in the same block as x." Applied to program P1 above, this would seem to imply that the correct output of P1 is 1. However the above quote has a parenthetical comment that all identifiers must be distinct for this to apply and refers to Section 3.E for the case where identifiers are not necessarily distinct (this is the case with P1). Reading Section 3.E, we find that the definition of a variable definition in an inner block is valid throughout that block. This might suggest that the correct output of P1 is 2. Actually this rule has nothing to do with program P1 as it deals exclusively with variable identifiers, the topic of Section 3.E. Unfortunately the other sections on type identifiers, procedure identifiers and constant identifiers give no rules at all.

The last, most specific and least satisfactory source for a resolution of scope rules (other than for variable identifiers) is the E.T.H. compilers. Because of Wirth's close association here, their performance must be considered significant. The output of both the Version 2 and Version 3 compiler for P1 is 1. This performance is supported by the rule in Chapter 1 (p. 8, item 16) of the User Manual that "All objects must be declared before they are referenced" (two exceptions noted are pointer types and forward procedures). In the absence of other rules about scope it is not unnatural to apply this one, hence accepting the outer definition throughout its scope until another occurs (the Version 2 and 3 compilers do violate the unique association rule which does not come up in P1). This presumably is the reason for Watt's [4] assumption that Sale [3] criticizes.

THE BSI/ISO STANDARD

We now turn our attention to the new Draft Standard [1]. While there are problems with the existing language specification, it is this new definition that causes us the most serious concern. The Draft Standard eliminates the previously existing omissions on the specification of scope rules. There is an explicit enumeration of the nested scope rules for all varieties of identifiers (see Section 6.2.1). Unfortunately, as we shall see, these rules imply that scope ≠ block for all cases except variable and type identifiers.

Each identifier has a defining occurrence and each defining occurrence has a scope which encloses all "corresponding occurrences" (a term not defined). Here the Draft Standard leaves some ambiguity as it does not state precisely where such scope begins and ends. Since the scope must enclose all "corresponding occurrences" we shall simply assume that the scope ends with the end of the block in whose heading the defining occurrence appears. The choice for the beginning of the scope is another question. Since each defining occurrence is prescribed as having a scope associated with it (i.e., scopes are associated with defining occurrences not blocks), one seems naturally forced to assume that such a scope begins with the defining occurrence. This assumption seems reinforced by the rule (in Section 6.4) that the scope of the defining occurrence of a type identifier does not include its own definition, except for pointer types. There is one exception to this assumption explicitly stated in rule (5) of Section 6.2.1. This rule states that the defining occurrence of any identifier or label must precede all its "corresponding occurrences" except for a pointer-type identifier which may have its defining occurrence anywhere in the type-definition part. Hence we assume that the scope of the pointer-type identifier begins with the beginning of the type-definition part rather than with its defining occurrence.

Now consider the previously given program example P1. There is no longer any doubt over what its correct output must be. This program has two defining occurrences of the identifier 'Q' (the specification of a defining occurrence for a procedure identifier is given in Section 6.6.1), in lines 2 and 5. The scope of the first extends to the end of P1 (i.e., lines 2-7) and the nested scope of the second extends to the end of procedure R (i.e., lines 5-6). Clearly then the call in line 4 is a "corresponding occurrence" for the definition in line 2, an association clearly violating ALGOL60-style scope rules.

The same situation prevails for constant identifiers. As an example consider

```
1 PROGRAM P2(OUTPUT);
2 CONST TWO = 2;
3   PROCEDURE Q;
4   CONST ONE = TWO;
5            TWO = 1;
6   BEGIN WRITELN(ONE) END;
7 BEGIN Q END.
```

We do not include the scope analysis as it is similar to that for program P1. The upshot is the same as for procedure identifiers, namely scope ≠ block for constant identifiers.

On the other hand since type-identifiers cannot occur in a heading prior to the type-definition part, rule (5) of Section 6.2.1 implies that scope = block for type identifiers. For instance, in contrast to the previous examples, the program

```
1 PROGRAM P3(OUTPUT);
2 TYPE A = RECORD L : ^A; C : REAL END;
3   PROCEDURE Q;
4   TYPE B = ^A;
5          A = RECORD L : B; C : INTEGER END;
6   VAR X : B;
7   BEGIN NEW(X); X^.C := 0.5 END;
8 BEGIN Q END.
```

is illegal because of the type conflict in the assignment in line 7 (however the Version 3 E.T.H. compiler finds it legal).

Also since variable identifiers cannot be used in the heading at all, these rules imply that scope = block for variable identifiers as well. Hence for the Draft Standard we get two answers to the question of the title; 'yes' for variable and type identifiers and 'no' for constant, procedure and enumeration-type identifiers.

CONCLUSIONS

The lack of specification rules for nested scopes in the original PASCAL definition has resulted in different interpretations being taken by different implementations. This point has already been made in [5]. The fact that so basic an issue must be settled has been recognized in the development of a draft standard.

We feel that while the Draft Standard does resolve the ambiguities of scopes, the solution that is proposed is very poorly conceived. The answer to the question "does scope = block?" should be uniform for all varieties of identifiers and furthermore we agree with Sale [3], that answer should be yes.

Programs P1 and P2 show how present scope rules provide for the binding of corresponding occurrences of identifiers to defining occurrences outside the block of the corresponding occurrence even though this block itself contains a defining occurrence. A convention that provides for the binding of one identifier to two definitions within the same block seems entirely contrary to the evolution of PASCAL.

The scope rules should state that the scope of a defining occurrence extends from the beginning of the block in whose heading it occurs to the end of this block. This would replace rules (1) and (2) of Section 6.2.1 of [1]. The other rules would be retained as stated; however we would rephrase rule (5) slightly to say that the completion of the definition for a defining occurrence must precede all corresponding occurrences–then the scope rule in Section 6.4 is dropped. This would make programs P1 and P2 illegal as they then violate rule (5)–the defining occurrence in the nested block does not precede the first use. It has already been suggested [5] how this interpretation can be handled in a one-pass compiler. The only complication to this comes in the exception to rule (5) for pointer-types which must force the binding of <u>all</u> such identifiers (even those with definitions in enclosing scopes) to be deferred until the end of the type-definition part.

We feel the approach we suggest provides a conceptually cleaner solution to the scoping questions. The treatment of all varieties of identifiers is internally consistent and consistent with the conventions of other block structure languages as well. Moreover it conforms with the principle of locality. With the rules given in the present Draft Standard, a block can contain identifiers with both a local and a nonlocal binding–a very confusing situation.

REFERENCES

1. A.M. Addyman et al., "A draft description of PASCAL", *Software–Pract. & Exper.* 9,5(1979), 381-424; also PASCAL News 14(1979), 7-54.

2. K. Jensen & N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, Second Edition, 1975.

3. A. Sale, "Scope and PASCAL", *SIGPLAN Notices* 14,9(Sept. 1979), 6-63.

4. D.A. Watt, "An extended attribute grammar for PASCAL", *SIGPLAN Notices* 14,2(Feb. 1979), 60-74.

5. J. Welch, W.J. Sneeringer & C.A.R. Hoare. "Ambiguities and insecurities in PASCAL", *Software–Pract. & Exper.* 7(1977), 685-696.