Name Identification for Languages with Explicit Scope Control

MASATO TAKEICHI*

This paper describes basic algorithms for analyzing the scope of identifiers of block-structured languages. The scope rules treated here include closed scopes in addition to the classical open scopes. Name identification process establishes the relation between defining and applied occurrences according to the scope rules. Analysis of the scope rules related to data types is left to the semantic analysis and is not treated here. The algorithms are defined on the (abstract) parse tree which has been generated by the syntax analyzer and will be transformed in the course of later semantic processing. As the algorithms do not require any additional tables to pass information to other semantic routines, they would be most desirable for construction of the compiler front-end together with various semantic routines.

In Section 1, objective of our research is presented. Section 2 introduces data structures of the parse tree and some notational conventions which will be used in describing our algorithms. In Section 3, a simple algorithm for the traditional open scope rules is described in order to clarify our idea. In Section 4, the closed scope rules in modern programming languages are treated. Section 5 is devoted to presenting more sophisticated algorithms which detect conflicts in declarations. These algorithms can be used in actual name identification routines. And the concluding remarks are given in Section 6.

1. Introduction

The compilation process is usually partitioned into a series of logically cohesive phases such as lexical analysis, syntax analysis, semantic analysis, and code generation. Identifiers appearing in the source program are transformed into some internal representation by the lexical analyzer at the first stage of compilation. Different instances of an identifier are made to share the same internal value regardless of the scope rules of the language. Each instance of identifiers is then examined again in subsequent phases with reference to the scope rules. Since the traditional scope rules of Algol-like languages are specified solely by the nested structure of the syntactic units, e.g., blocks, name identification could be done in parallel with the syntax analysis. Certain programming languages, however, provide extended facilities for the programmer to control the scope of identifiers. In processing these languages, the name identification requires more information than the syntactic structure of the source program and it will be done as a part of semantic analysis which follows the syntax analysis. As the semantic analysis, in general, performs many complex tasks, it is desirable to take the name identification process out of general semantic processing, when possible.

This paper gives simple name identification algorithms for extended scope rules which can be applied to construction of the compiler front-end. The front-end translates the source program into some internal form of the intermediate language and produces machine-independent description of the program which will be convenient for the code generator [1]. It is usually composed of the lexical analyzer, the syntax analyzer, and the semantic analyzer. The first stages of compilation, i.e., the lexical and syntax analyses, have been automated by program generators such as LEX and YACC of the UNIX operating system [2]. The name identification process as a part of the semantic analysis could be implemented in common for different languages with similar scope rules.

2. Data Structure

As our algorithms for name identification will be defined on the *parse tree* produced by the syntax analyzer, we will sketch here the first phases of compilation and introduce some notations for representing tree data structures. The lexical analyzer returns a token when called by the syntax analyzer. If the token returned is an identifier, additional value is returned so that the spelling of the identifier can be referenced by subsequent phases of the compiler. The lexical analyzer uses the name table to get the same internal value for different instances of an identifier. Although our implementation puts no assumption on the structure of the name table, the internal value for the spelling is assumed to be a pointer to the record in the name table. For example, the hash method with a fixed table for primary probing works well for our purpose. The syntax analyzer parses source programs and translates them into parse trees, which will be scanned and transformed by the name identification

*Department of Computer Science, The University of Electro-Communications.

procedure and further semantic routines. As we will discuss mainly the name identification problems, detailed structure of the parse tree is irrelevant and is not specified here.

We will refer to the node in the parse tree as *record*. Every record for the identifier is assumed to have an *occurrence tag* and a pointer to the record with spelling. The occurrence tag is used to specify the kind of occurrences of identifiers. We will use the term *defining occurrence* or *definition* for an instance of an identifier introduced in a declaration, and the term *applied occurrence* or *application* for other instance. The definition and application records have occurrence tags D and A, respectively.

Figure 1 shows an example of the parse tree for a program segment of Pascal. It should be noted that the record with spelling is effectively shared by as many records for the occurrences of that identifier.

The purpose of name identification is to establish the relation between each applied occurrence and the defining occurrence according to the scope rules. In Fig. 1, dotted lines show such relation under the classical scope rules of Pascal. Note that the hash table shown in Fig. 1 becomes unnecessary when the syntax analysis completes, and the hash link field located in the record with spelling becomes available for identification processing.

We will use the term *block* for a syntactic unit in which identifiers are declared and their visibility ranges over. In general, blocks may be nested and the nested structure is naturally reflected in the parse tree. We assume that each tree for a block has several subtrees for internal blocks as its descendants. A syntactic unit X such as a block, a declaration, or an applied occurrence of an identifier is called an *immediate constituent*, or a *constituent* in short, of some block B if B is the smallest block enclosing X.

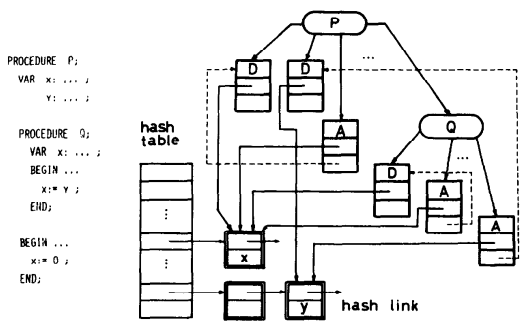

Fig. 1 Example of parse tree.

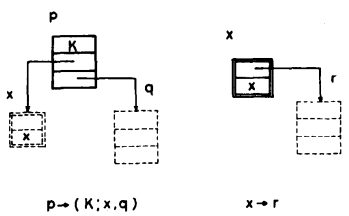

Fig. 2 Notations for data structure.

In order to describe algorithms concisely, we will use the following notations:

a) $p\rightarrow(K; x, q)$ means that the record referenced by p has an occurrence tag K, a pointer to the record with spelling "x", and a pointer q to an other record.

b) $x\rightarrow r$ means that the record with spelling referenced by x has a pointer r to an other record.

Abbreviated notations and corresponding forms of the record are shown in Fig. 2.

The parse tree produced by the syntax analyzer is expressed using the above notations:

a) for each occurrence of identifier x,
$p\rightarrow(K; x, \textbf{nil})$

b) for each identifier x,
$x\rightarrow\textbf{nil}$

where **nil** points to no record at all.
Note that the record with spelling "x" is unique and $x\rightarrow\textbf{nil}$.

After completion of the name identification process, each applied occurrence of identifier x should be
$p\rightarrow(A; x, q)$,
where q points to the definition of x with respect to the scope rules.

3. Name Identification for Open Scope

In the traditional scope rules of Algol-like languages, an identifier is declared in a block and is automatically inherited in all constituent blocks unless the identifier is redeclared in inner blocks. The definition of the identifier is not visible outside the block in which the declaration occurs. Such automatic inheritance with restricted visibility of identifiers is called an *open scope* rule [3].

A basic identification algorithm for the open scope is as follows:

block(b):
1: **for each** constituent definition $p\rightarrow(D; x, \textbf{nil})$, $x\rightarrow q$ of b **do**
$p\rightarrow(D; x, q)$, $x\rightarrow p$
2: **for each** constituent block b' of b **do**
block(b')
3: **for each** constituent application $p\rightarrow(A; x, \textbf{nil})$, $x\rightarrow q$ of b **do**
$p\rightarrow(A; x, q)$
4: **for each** constituent definition $p\rightarrow(D; x, q)$, $x\rightarrow p$ of b **do**
$x\rightarrow q$

The first statement should be read as follows:
for each defining occurrence of identifiers in block b, which is represented by a record
$p\rightarrow(D; x, \textbf{nil})$
with the record for x being
$x\rightarrow q$,
alter the contents of the records referenced by p and x so that relations

$p \rightarrow (D; x, q)$ and $x \rightarrow p$

hold.

Other statements should be read similarly. The procedure "*block*" is called recursively for inner blocks at step 2, which follows the data structure of the parse tree.

The algorithm does not detect any possible conflicts of definitions such as more than one defining occurrences of an identifier in a block. When we apply this algorithm to implementation of the name analysis procedure, it is necessary to make additional tests on definitions. An improved algorithm will be described in Section 5.

In many implementations of Algol-like languages, the symbol table is organized as a stack for processing the open scope rule [4]. The above algorithm uses the third field q of the definition record

$$p \rightarrow (D; x, q)$$

as a link to the definition of x appeared in some outer block. Such linked structure effectively behaves as a stack. If the symbol table is organized as a stack with contiguous storage locations, removal of local definitions at scope exit is performed by simply moving the stack pointer. Since the algorithm here uses the linked structure for the stack, another scan of definitions is necessary at step 4. The time required at scope exit is, however, considerably less than that of other operations for applied occurrences, which normally appear much more than definitions.

4. Name Identification for Closed Scope

Modern programming languages such as Modula-2 [5], Euclid [6], Mesa [7], and Ada [8] provide new facilities to control the visibility of identifiers in addition to the classical open scope. In these languages, the programmer can specify explicitly which identifiers are or are not inherited by inner blocks and which ones are visible outside the block where they are declared. This kind of a scope is called a *closed scope* in contrast to the open scope [3]. In Modula-2, the open scope is associated with a procedure declaration and the closed scope is with a module declaration. We will follow the terminology used in the definition of Modula-2 and refer to explicit specification of inheritance as importation and explicit widening of inner declaration as exportation.

Identifiers which are imported (exported) are assumed to be specified in an *import* (*export*) *list* of a block. Figure 3 shows an example of blocks with explicit scope control. An *importation* (*exportation*), i.e., an occurrence of an identifier in an import (export) list, is assumed to have an occurrence tag $I(E)$.

There may be conflicts in definitions, importations, and exportations. It is reasonable to assume that the "legal" block contains no two occurrences of the same identifier in definitions and importations, and no definition or importation shares any exported identifiers from inner blocks. This is the case with Modula-2. We assume for now that no conflict appears among definitions, importations, and exportations. We will describe an algorithm with conflict detection in the next section.

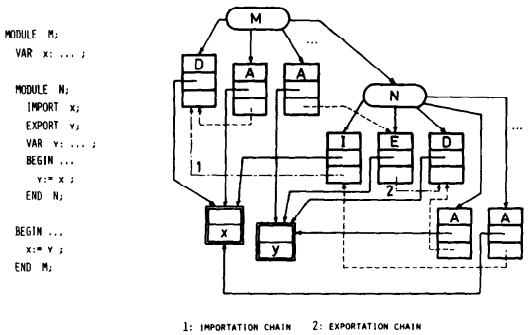

Fig. 3 Example of closed scope.

Each applied occurrence of an imported or exported identifier will be assigned a pointer to corresponding importation or exportation, which in turn points to another importation, exportation, or (finally) definition. Thus an application

$$p \rightarrow (A; x, \textbf{nil})$$

of an imported identifier x will become

$$p \rightarrow (A; x, q_0)$$

and an *importation chain*

$$q_0 \rightarrow (I; x, q_1)$$
$$q_1 \rightarrow (I; x, q_2)$$
$$\cdots$$
$$q_n \rightarrow (D; x, r)$$

will be established. The importation chain will be used in later phases to examine attributes given by the declara- of the identifier. An application of an exported identifier is processed in the same way and an *exportation chain* is also created. We will denote an importation or exportation chain for $q = q_0$, $n \geqq 0$ as

$$q \overset{*}{\rightarrow} (D; x, r)$$

A basic algorithm for the closed scope is described as follows:

block(b):

1: **for each** constituent block b' of b **do**
  for each constituent exportation $p \rightarrow (E; x, \textbf{nil})$, $x \rightarrow q$ of b' **do**
    $p \rightarrow (E; x, q)$, $x \rightarrow p$
2: **for each** constituent definition $p \rightarrow (D; x, \textbf{nil})$, $x \rightarrow q$ of b **do**
    $p \rightarrow (D; x, q)$, $x \rightarrow p$
3: **for each** constituent importation $p \rightarrow (I; x, \textbf{nil})$, $x \rightarrow q$ of b **do**
    $p \rightarrow (I; x, q)$, $x \rightarrow p$
4: **for each** constituent block b' of b **do**
    block(b')
5: **for each** constituent application $p \rightarrow (A; x, \textbf{nil})$, $x \rightarrow q$ of b **do**
    $p \rightarrow (A; x, q)$
6: **for each** constituent importation $p \rightarrow (I; x, q)$, $x \rightarrow p$ of b **do**
    $x \rightarrow q$

7: **for each** constituent definition $p\rightarrow(D; x, q)$,
$x\rightarrow p$ of b **do**
$x\rightarrow q$
if $q\rightarrow(E; x, r)$ is a constituent of b **then**
$q\rightarrow(E; x, p)$, $p\rightarrow(D; x, r)$

8: **for each** constituent block b' of b **do**
for each constituent exportation $p\rightarrow(E; x, q)$,
$x\rightarrow p$, $q\overset{*}{\rightarrow}(D; x, r)$ of b' **do**
$x\rightarrow r$
if $r\rightarrow(E; x, s)$ is a constituent of b **then**
$r\rightarrow(E; x, p)$, $q\overset{*}{\rightarrow}(D; x, s)$

The importation chain is established in step 3, which remains unchanged thereafter. On the other hand, creation of the exportation chain is somewhat intricate. In steps 7 and 8, the pointer in the third field of the exportation record is moved to its definition record and a pointer to an exportation or definition record is given to that field in place (Fig. 4). This makes extra scan of the exportation chain necessary at each scope exit. However, it would be reasonable to assume that the depth of block nesting is limited to 2 or 3 in actual programs. Observation of about 30 modules in the Modula-2 system distributed by ETH [9] reveals that only one module contains a module declaration in itself. Our algorithm would be acceptable if one recognizes that it uses no extra space to hold the exportation chain.

Another question may be posed. That is, the proposition

"... is a constituent of b"

in steps 7 and 8 is not expressed in terms of the data structure described so far. If every occurrence of the identifier had the nesting level of the block, it is easy to determine whether it is or is not a constituent of a specified block. Such extended structure can also be applied to the algorithms with conflict detection in the next section.

5. Conflict Detection

The algorithms described in previous sections work for programs which contain no conflict of names in definitions, importations, and exportations. The practical name identification routine should, however, detect such conflicts and perform identification properly even if

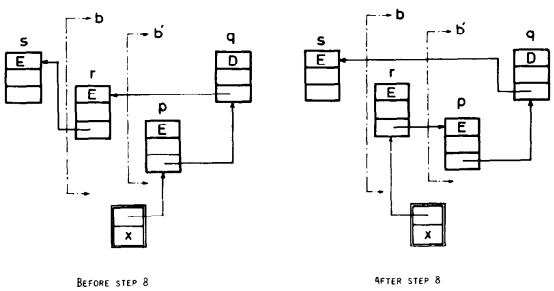

Fig. 4 Creation of exportation chain.

erroneous programs are given. In this section, we will incorporate conflict detection mechanism into the previous algorithms. As mentioned in the last section, the nesting level of the block will take an important role in deciding where the identifier occurs. Detection of errors such as applied occurrences of undefined identifiers is left to further semantic analysis. It is relatively easy if one recognizes that **nil** in the third field of the application record indicates the case.

We now extend the data structure for the occurrence of identifiers so that it can hold the nesting level l of the block, which is of the form

$$p\rightarrow(K; x, q, l).$$

We assume that the level l has been given to every record for identifiers in the phase of syntax analysis.

Using the extended structure, an improved algorithm for the open scope becomes:

block(b, l):

1. **for each** constituent definition $p\rightarrow(D; x,$ **nil**, $l)$,
$x\rightarrow q$ of b **do**
if $q\rightarrow(D; x, r, l)$ **then** *error* ("def/def conflict')
else
$p\rightarrow(D; x, q, l)$, $x\rightarrow p$
2. **for each** constituent block b' of b **do**
block(b', $l+1$)
3. **for each** constituent application $p\rightarrow(A; x,$ **nil**),
$x\rightarrow q$ of q **do**
$p\rightarrow(A; x, q, l)$
4. **for each** constituent definition $p\rightarrow(D; x, q, l)$,
$x\rightarrow r$ **do**
if $p=r$ **then**
$x\rightarrow q$

Note that the last statement in step 4 restores only the pointers which have been successfully stacked at step 1.

Similar improvement on the algorithm for the closed scope yields:

block(b, l):

1. **for each** constituent block b' of b **do**
for each constituent exportation $p\rightarrow(E; x,$ **nil**, $l+1)$,
$x\rightarrow q$ of b' **do**
if $q\rightarrow(E; x, r, l+1)$ **then** *error* ("exp/exp conflict")
else
$p\rightarrow(E; x, q, l+1)$, $x\rightarrow p$
2. **for each** constituent definition $p\rightarrow(D; x,$ **nil**, $l)$,
$x\rightarrow q$ of b **do**
if $q\rightarrow(D; x, r, l)$ **then** *error* ("def/def conflict")
else if $q\rightarrow(E; x, r, l+1)$ **then**
error ("def/exp conflict")
else
$p\rightarrow(D; x, q, l)$, $x\rightarrow p$
3. **for each** constituent importation $p\rightarrow(I; x,$ **nil**, $l)$,
$x\rightarrow q$ of b **do**
if $q\rightarrow(D; x, r, l)$ **then** *error* ("imp/def conflict")
else if $q\rightarrow(E; x, r, l+1)$ **then**
error ("imp/exp conflict")
else if $q\rightarrow(I; x, r, l)$ **then** *error*("imp/imp conflict")
else
$p\rightarrow(I; x, q, l)$, $x\rightarrow p$

4. **for each** constituent block b' of b **do**
   $block(b', l+1)$
5. **for each** constituent application $p \rightarrow (A; x, \textbf{nil}, l)$, $x \rightarrow q$ of b **do**
   $p \rightarrow (A; x, q, l)$
6. **for each** constituent importation $p \rightarrow (I; x, q, l)$, $x \rightarrow r$ of b **do**
   if $p = r$ **then**
   $x \rightarrow q$
7. **for each** constituent definition $p \rightarrow (D; x, q, l)$, $x \rightarrow s$ of b **do**
   if $p = s$ **then**
   $x \rightarrow q$
   if $q \rightarrow (E; x, r, l)$ **then**
   $q \rightarrow (E; x, p, l)$, $p \rightarrow (D; x, r, l)$
8. **for each** constituent block b' of b **do**
   for each constituent exportation $p \rightarrow (E; x, q, l+1)$, $x \rightarrow t$ of b' **do**
   if $p = t$, $q \overset{*}{\rightarrow} (D; x, r, l')$ **then**
   $x \rightarrow r$
   if $r \rightarrow (E; x, s, l)$ **then**
   $r \rightarrow (E; x, p, l)$, $q \overset{*}{\rightarrow} (D; x, s, l')$

In this algorithm, the tests for constituency is expressed by means of the level of the block as:

if $q \rightarrow (E; x, r, l)$ **then** $\cdots$

which should be read as:

if the record referenced by q has occurrence tag E, and level l (x and r do not matter here), then . . .

The algorithms described in this section would be applicable to practical name identification routines.

6. Conclusion

Practical languages with explicit scope control usually provide the block with the classical scope rules as well. In Modula-2, for example, the module (closed scope) may contain both modules and procedures (open scope) in itself. Our algorithms for open and closed scopes described in this paper are easily combined together to deal with such intermixed scopes.

Another kind of scope is found in languages with record types. The feature of selective widening of scopes by qualification has a similar property, which is found in Modula-2 modules and Simula classes. A record field is usually named by a field identifier, which has a scope different from that of identifiers defined in the block. The record field is designated by a *qualified name* composed of a qualifier indicating the variable identifier of that record type and the field identifier itself. This kind of names can be analyzed provided that the identification of the variable has been completed and the type has been processed. As our algorithms take no account of the semantic processing of declarations, analysis of such scope is not treated. However, algorithms for identification of qualified names could be defined using our data structure as well by extending our algorithms to handle the association of identifiers and types.

The symbol table described in [3] is especially designed for processing importations and exportations. It is based on the classical one using stack mechanism. The algorithm for handling that table seems complicated and has less generality. Forward references of identifiers are not treated there. And the operations at scope exit are emphasized and a method to reduce the time required at scope exit is proposed. Although assumptions of our algorithm differ from those of [3], our approach would give more general and useful frameworks in construction of name identification routines for many languages.

It is believed that our algorithms are so simple that they can be incorporated with other semantic processing in the compiler front-end. An implementation of Modula-2 based on our identification algorithms is in progress.

Acknowledgement

Katsuhiko Kakehi and Eiichiro Sumita participated in many discussions during this work. The author would like to thank them. He also acknowledges the referees for detailed comments and suggestions for improvements of an earlier draft.

References

1. Goos, G. and Winterstein, G. Towards a Compiler Front-End for Ada. Proc. ACM-SIGPLAN Symp. The Ada Programming Language, *SIGPLAN Notices* 15 (November 1980), 36–46.
2. Special Issue on the UNIX TIME-SHARING SYSTEM, *The Bell System Technical Journal* 57, Part 2 (June 1978).
3. Graham, S. L., Joy, W. N. and Roubine, O. Hashed symbol tables for languages with explicit scope control. Proc. ACM Symp. Compiler Construction, *SIGPLAN Notices* 14 (August 1979), 50–57.
4. Aho, A. V. and Ullman, J. D. Principles of Compiler Design. Addison-Wesley (1977).
5. Wirth, N. MODULA-2. Berichte des Institut für Informatik Nr. 36, Eidgenössische Technische Hochschule Zürich (1980).
6. Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G. and Popek, G. L. Report on the programming Language Euclid. *SIGPLAN Notices* 12 (February 1977).
7. Michell, J. G., Maybury, W. and Sweet, R. Mesa Language Manual. *Xerox PARC*, CSL-78-1 (1978).
8. Reference Manual for the Ada Programming Language. United States Department of Defense (1980).
9. Modula-2 compiler distribution tape. ETH Zürich (1980).

(Received August 27, 1981: revised October 21, 1981)