Tutorial on Name Analysis

$Revision: 4.16 $

Compiler and Programming Language Group
University of Paderborn, FB 17
33098 Paderborn, FRG

Table of Contents

Overview

This tutorial introduces to the solution of the name analysis task in language implementation. It demonstrates many aspects of that task which occur in programming languages or special purpose languages. They are developed from basic ones, such as nested scopes, up to more complex ones, such as classes with multiple inheritance.

This tutorial may be used as a practical introduction to the specification techniques for name analysis, or as a source of examples that show how to solve certain problems, or as a source of hints for language design.

This file is an executable specification. An analyzer for a small artificial language can be generated from it. The language is kept small by restricting it to those constructs necessary to demonstrate the name analysis task, not regarding its usability for programming. Its notation is inspired by Pascal, although the name analysis rules are not taken from Pascal. The generated analyzer produces output that reports the result of name analysis, i. e. the binding of identifier occurrences.

The explanations in this tutorial assume that the reader is familiar with the use of the Eli system, with the use of its module library, and knows the general description of the name analysis task in the documentation of the module library, see Section 'Name Analysis' in Specification Module Library: Name-Analysis.

This tutorial is available in three variants that differ in the basic underlying scope rules, C-like or Algol-like. The third variant is based on C-like scope rules; its computations are executed while the input is read. The specifications differ only in which library modules being used, and, in a few places, where they are applied. This variant of the tutorial shows the Algol-like scope rules.

1 Kernel Grammar

The kernel grammar of the example language specifies the structure for `Program`, `Block` consisting of `Declarations` and `Statements`, and trivial `Expressions`. Different forms of `Declarations`, `Statements`, and `Expressions` are added to the grammar as the name analysis task is further refined.

Core.con[1]==

```
Program:        Source.

Source:         Block.
Block:          Compound.
Compound:       'begin' Declaration* Statement* 'end'.


Statement:      Expression ';'.
Expression:     Operand.
Operand:        IntNumber.
```

This macro is attached to a product file.

`Expressions` and `Operands` are represented by `Expression` contexts in the tree grammar.

Core.sym[2]==

```
Expression ::= Operand.
```

This macro is attached to a product file.

The notation of identifiers and numbers is chosen as in Pascal.

Core.gla[3]==

```
Ident:          PASCAL_IDENTIFIER
IntNumber:      PASCAL_INTEGER
                PASCAL_COMMENT
```

This macro is attached to a product file.

In the course of refining the name analysis task we will introduce several different contexts for identifier occurrences. Each occurrence has to have the attribute `Sym` representing the identifier encoding. Hence we specify a computational role `IdentOcc` that provides that attribute, and will be inherited by any identifier occurrence.

Core.lido[4]==

```
TERM Ident: int;
ATTR Sym: int;
CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;
```

This macro is attached to a product file.

2 Basic Scope Rules

In many cases of name analysis the language distinguishes between defining occurrences of identifiers in declaration and applied occurrences in expressions (an example with only one kind of identifier occurrence is shown below). Hence, we extend our kernel grammar by `DefIdent` for defining occurrences of identifiers and `UseIdent` for applied occurrences, and by `Declarations` that introduce names for values. An `Expression` can be an applied identifier occurrence.

Furthermore we introduce nested `Blocks` by deriving them from `Expressions`, and hence also from `Statements`.

CoreScope.con[5]==

```
Declaration:    'val' ValDecls ';'.
ValDecls:       ValDecl // ','.
ValDecl:        DefIdent '=' Expression.
Expression:     Block.
Operand:        UseIdent.

DefIdent:       Ident.
UseIdent:       Ident.
```

This macro is attached to a product file.

The basic task of name analysis is consistent renaming. For each identifier occurrence a `Key` attribute is computed such that identifier occurrences that refer to the same object in the program have the same `Key` attribute value. Hence `Keys` identify program objects uniquely. `Keys` are used to associate properties with program objects and to retrieve those properties in different contexts.

The scope rules of a language determine how identifier occurrences are bound to program objects.

The basic Algol-like scope rule reads:

> A definition of an identifier `a` is valid in the whole smallest range that encloses that definition, except inner ranges that contain another definition of `a`.

Hence, a definition in an outer range is hidden by a definition of the same identifier in an inner range for the whole inner range. Identifiers may be applied before they are defined.

We instantiate a library module that provides computations according to this scope rule:

CoreScope.specs[6]==

```
$/Name/AlgScope.gnrc:inst
```

This macro is attached to a product file.

The computational roles `RangeScope`, `IdDefScope`, and `IdUseEnv` are associated with the corresponding symbols of our grammar.

CoreScope.lido[7]==

```
SYMBOL Block    INHERITS RangeScope END;
SYMBOL DefIdent INHERITS IdDefScope, IdentOcc END;
SYMBOL UseIdent INHERITS IdUseEnv, IdentOcc END;
```

This macro is attached to a product file.

As a result attributes `DefIdent.Key` and `UseIdent.Key` are computed according to the scope rules.

3 Produce Output of Identifier binding

We want to test the name analysis of our compiler. For that purpose we instantiate a library module that provides computations which report the results of name analysis. It is instantiated in the same way as the basic name analysis module:

NameTest.specs[8]==

```
$/Name/ShowBinding.gnrc:inst
```

This macro is attached to a product file.

As a result the generated compiler will produce a line for every identifier occurrence like

```
m in line 23 bound in line 4 of scope in line 3
```

4 Messages on Scope Rule Violations

Erroneous programs may violate the scope rules in one of two cases:

- A particular applied identifier occurrence has no valid defining identifier occurrence.
- An identifier in a range may have more one defining occurrences.

Such situations shall be indicated by error messages. Furthermore, we want every defining occurrence of a multiply defined identifier to be marked by a message.

For that purpose we use the following two library modules:

CoreChk.specs[9]==

```
$/Tech/Strings.specs
$/Prop/OccCnt.gnrc:inst
```

This macro is attached to a product file.

The `Strings` module provides a function that concatenates an error message string and an identifier for error messages related to identifiers, see Section 'String Concatenation' in Specification Module Library: Common Problems.

The `OccCnt` module provides computations that count how often an object identified by a `Key` attribute occurs in a certain context, in our case in a defining context. See Section 'Count Occurrences of Objects' in Specification Module Library: Properties of Definitions, for more information.

The check for existence of a definition is directly obtained from the module role `ChkIdUse`. For the second check we specify a computational role `ChkUnique` in order to reuse it for several grammar symbols. If an object occurs more than once in the `ChkUnique` context it is multiply defined.

CoreChk.lido[10]==

```
SYMBOL ChkUnique INHERITS Count, TotalCnt COMPUTE
  IF (GT (THIS.TotalCnt, 1),
  message (ERROR, CatStrInd ("identifier is multiply defined: ",
                             THIS.Sym),
           0, COORDREF));
```

```
END;

SYMBOL UseIdent INHERITS ChkIdUse END;
SYMBOL DefIdent INHERITS ChkUnique END;
```

This macro is attached to a product file.

5 Predefined Identifiers

Allmost all languages have certain identifiers that are predefined for any program. This facility is demonstrated by introducing typed variable declarations to our language:

Predef.con[11]==

```
Declaration:    'var' VarDecls ';'.
VarDecls:       VarDecl // ','.
VarDecl:        TypeUseIdent DefIdent.
TypeUseIdent:   Ident.
```

This macro is attached to a product file.

The type of a declared variable is given by a `TypeUseIdent`. A `TypeUseIdent` has the same computational roles which we associated with applied identifier occurrences, except that it is not checked for being defined since we assume that only predefined type identifiers are used:

Predef.lido[12]==

```
SYMBOL TypeUseIdent INHERITS IdUseEnv, IdentOcc END;
```

This macro is attached to a product file.

The technique for predefining identifiers is provided by the library modules `PreDefine` and `PreDefId`, see See Section 'Predefined Identifiers' in Specification Module Library: Name Analysis. `PreDefine` is instantiated with the `Ident` symbol name as generic parameter. `PreDefId` is instantiated with the file name `Predef.d`. That file contains a sequence of macro calls, like `PreDefKey ("int", intKey)`. Each introduces a string and the name of a key. The identifier is then bound to that key in the outermost environment of each program. Those `Keys` could be used in computations, e.g. for type checking (We do not make use of this facility here.)

Predef.specs[13]==

```
$/Name/PreDefine.gnrc+referto=Ident:inst
$/Name/PreDefId.gnrc+referto=(Predef.d):inst
```

This macro is attached to a product file.

Predef.d[14]==

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
PreDefKey ("bool", boolKey)
PreDefKey ("true", trueKey)
PreDefKey ("false", falseKey)
```

This macro is attached to a product file.

6 Joined Ranges

In our language subtrees rooted by a `Block` symbol exactly cover a range of the program in the sense of the scope rules. Hence, we could simply associate the role `RangeScope` with `Block` above.

However, there are situations where a range in the sense of the scope rules extends over several subtrees, but their common root can not be taken as the scope range. Such a situation occurs for example in Pascal, where the formal parameter list of a procedure and the procedure body form a single range.

The `join` statement below demonstrates such a situation. It consists of two blocks which together shall form one range in the sense of the scope rules. I. e. any definition in each of the blocks is valid in both of them. But the identifier after the `join` symbol is defined in the enclosing range.

Join.con[15]==

```
Statement:      Join.
Join:           'join' DefIdent JoinedBlock JoinedBlock 'joined' ';'.
JoinedBlock:    Compound.
```

This macro is attached to a product file.

We could modify the grammar in order to get a single symbol representing that range. But that may not be desirable due to parsing reasons. The problem is solved by the following technique:

A symbol, here `Join`, is identified such that it contains both ranges. It has the module role `RangeSequence`, which does not constitute a range in the sense of scope rules. Hence the `DefIdent` belongs to a range that encloses the statement.

The symbol `JoinedBlock` has the module role `RangeElement`. The two `JoinedBlock` below the `Join` symbol then form a range in the sense of scope rules.

The roles `RangeSequence` and `RangeElement` are obtained from the library module `AlgRangeSeq`, see See Section 'Joined Ranges' in Specification Module Library: Name Analysis.

Join.specs[16]==

```
$/Name/AlgRangeSeq.gnrc:inst
```

This macro is attached to a product file.

Join.lido[17]==

```
RULE: Join ::= 'join' DefIdent JoinedBlock JoinedBlock
               'joined' ';'
END;

SYMBOL Join INHERITS RangeSequence END;
SYMBOL JoinedBlock INHERITS RangeElement END;
```

This macro is attached to a product file.

7 A Second Name Space

The scope rules of some languages define several distinct name spaces, i.e. the identifier occurrences in one name space do not affect bindings in another name space. In C, for example, variable identifiers and label identifiers belong to different name spaces.

We demonstrate that aspect by introducing a special kind of variable to our language. Such variables are set by a special statement, and accessed by special operands. Hence, the identifier occurrences are syntactically distinguished from identifier occurrences of the name space used so far.

Flat.con[18]==

```
Statement:      'set' CtrlVarUse 'to' Expression ';'.
Operand:        'use' CtrlVarUse.
CtrlVarUse:     Ident.
```

This macro is attached to a product file.

We use another instantiation of the scope rule library module to specify the scope rules for the second name space, and require test output for it:

Flat.specs[19]==

```
$/Name/AlgScope.gnrc+instance=CtrlVar:inst
$/Name/ShowBinding.gnrc+instance=CtrlVar:inst
```

This macro is attached to a product file.

The `instance` parameter is set to `CtrlVar` to distinguish the module instance from the one for the first name space.

We demonstrate another variant of scope rules for our new `CtrlVar` name space: There is no nesting of ranges, i.e. the `Program` is the only range of a flat name space. Furthermore, identifiers are implicitly defined by using them. Hence, there is only one kind of identifier occurrences. It has the role of a defining occurrence (`CtrlVarIdDefScope`), i.e. a new object key is created, if the identifier is not yet bound.

Flat.lido[20]==

```
SYMBOL Source  INHERITS CtrlVarRangeScope END;
SYMBOL CtrlVarUse INHERITS CtrlVarIdDefScope, IdentOcc END;
```

This macro is attached to a product file.

Since we allow the use of `CtrlVar` before it is set, these scope rules can not be violated. Hence, we do not need any checks or messages.

8 Scopes being Properties of Objects

Certain language constructs require that a set of bindings, i.e. a scope is associated as a property of an object. We demonstrate this facility by introducing modules to our language:

ScopeProp.con[21]==

```
Declaration:    'module' DefIdent ModBlock ';'.
ModBlock:       Compound.
Operand:        ModUseIdent '::' QualIdent.
ModUseIdent:    Ident.
QualIdent:      Ident.
```

This macro is attached to a product file.

Any object `a` declared in the `ModBlock` of a module `m`, but not in deeper nested `Blocks`, can be accessed by `m::a` wherever `m` is bound to that module. We say the identifier occurrence of `a` is qualified by `m`.

A library module (see Section 'Scopes Being Properties of Objects' in Specification Module Library: Name Analysis) provides computational roles for scopes being associated with object keys:

ScopeProp.specs[22]==

```
$/Name/AlgScopeProp.gnrc:inst
```

This macro is attached to a product file.

The scope of the module body, with its local definitions, is associated as a property with the key representing the module. The role `RangeScopeProp` of the library module characterizes such an association. `ModBlock.ScopeKey` is used to specify the key with which the scope property is associated.

ScopePropDef.lido[23]==

```
SYMBOL ModBlock INHERITS RangeScopeProp END;

RULE: Declaration ::= 'module' DefIdent ModBlock ';' COMPUTE
  ModBlock.ScopeKey = DefIdent.Key;
END;
```

This macro is attached to a product file.

In binding a qualified identifier occurrence, `QualIdent`, the scope property associated with the `ModUseIdent.Key` is accessed, and the role `IdUseScopeProp` is used for `QualIdent`. This binding requires a specification of `QualIdent.Scope`.

We assume that `ModUseIdent` indeed has an associated scope . An error message is issued if that assumption is violated, e.g. in the case of a variable identifier.

In addition, the roles used for applied identifier occurrences are associated.

ScopePropUse.lido[24]==

```
RULE: Expression ::= ModUseIdent '::' QualIdent COMPUTE
  QualIdent.Scope = GetScope (ModUseIdent.Key, NoEnv)
        <- INCLUDING RootScope.GotScopeProp;
END;

SYMBOL ModUseIdent INHERITS
        ChkModUseIdent,
        IdUseEnv, ChkIdUse, IdentOcc
END;

SYMBOL QualIdent INHERITS
        IdUseScopeProp,
        ChkIdUse, IdentOcc
END;

CLASS SYMBOL ChkModUseIdent COMPUTE
  IF (AND (NE (THIS.Key, NoKey),
          EQ (GetScope (THIS.Key, NoEnv), NoEnv)),
  message (FATAL, CatStrInd ("module or class identifier required: ",
                             THIS.Sym), 0, COORDREF))
  <- INCLUDING RootScope.GotScopeProp;
END;
```

This macro is attached to a product file.

9 Insertion of Scopes into the Environment Hierarchy

We now demonstrate how scopes obtained from object properties are inserted into the environment hierarchy given by the syntactically nested ranges. For this purpose we extend our module example.

We introduce a `with` statement that allows the components of a module to be used without qualification in the `WithBody`. (This construct is similar to the `with` statement in Pascal, where record variables are used instead of the module identifiers discussed here. It directly corresponds to the `use` construct in Ada.)

ScopeInsert.con[25]==

```
Statement:      'with' ModUseIdent 'do' WithBody.
WithBody:       Statement.
```

This macro is attached to a product file.

The `WithBody` is a special kind of range: The scope rule for our `with` statement require that the scope of the module stated by the `ModUseIdent` is inserted in the environment hierarchy between the scope of the `WithBody` and the environment formed by the range nest that encloses the `with` statement. I.e. a definition in a range enclosing the `with` statement may be hidden by a definition of the module; those definitions may be hidden within the `WithBody`. (In the case of our language the `WithBody` may not directly contain declarations, although deeper nested `Blocks` may contain such declarations.)

We use another library module (see Section 'Inheritance of Scopes' in Specification Module Library: Name Analysis) to support such an embedding of environments:

ScopeInsert.specs[26]==

```
$/Name/AlgInh.gnrc:inst
```

This macro is attached to a product file.

The facility of inserting an environment obtained from a scope property of an object is provided by the module role `RangeSnglInh`. This role specializes `RangeScope` for single inheritance of a scope associated with `RangeSnglInh.InhKey`.

ScopeInsert.lido[27]==

```
SYMBOL WithBody INHERITS RangeSnglInh END;

RULE: Statement ::= 'with' ModUseIdent 'do' WithBody COMPUTE
  WithBody.InhKey = ModUseIdent.Key;
END;
```

This macro is attached to a product file.

10 Classes with Multiple Inheritance

Our previous examples of modules and `with` statements can be easily combined to demonstrate the scope rules for object oriented classes with multiple inheritance.

To avoid confusion with the so far specified scope rules we introduce a new language construct for declaration of classes:

Class.con[28]==

```
Declaration:    'class' DefIdent Inheritances ClassBlock ';'.
ClassBlock:     Compound.
Inheritances:   Inheritance*.
Inheritance:    ':' InheritIdent.
InheritIdent:   Ident.
```

This macro is attached to a product file.

Applied identifier occurrences within the body of a class are bound to definitions of that range, or to definitions that are visible due to inheritances from other classes (or modules), or to definitions in the ranges that enclose the class declaration.

Hence, the scopes obtained from inheritances are inserted into the environment hierarchy of the class body, as in the case of our `with` statements.

Since classes that are used for inheritance may inherit from other classes, the inheritance relation must form a partial order. It must not be cyclic. A class `c1` may inherit from a class `c2` via several paths through the inheritance relation.

That partial order governs hiding of definitions: A definition of an identifier `a` in the body of a class `c` hides definitions of `a` in any class directly or indirectly inherited by `c`.

Hence, the scope property of a class is the scope of the class body embedded in the environment of the inheritance relation for that class.

Wherever the class identifier is visible it can be used for qualified access, as introduced for modules: A qualified access `c::a` identifies an `a` defined in the body of class `c` or in a class inherited by `c` according to the inheritance relation.

These scope rules are specified using the techniques of the last two examples: A class has a scope property, as a module has; and a class body inherits other scopes as our `with` statement does. In this case there may be multiple scopes to be inherited. Hence, the `ClassBlock` combines the two roles `RangeScopeProp` and `RangeMulInh` of the library module.

Class.lido[29]==

```
SYMBOL ClassBlock INHERITS RangeScopeProp, RangeMulInh END;

RULE: Declaration ::= 'class' DefIdent Inheritances ClassBlock ';' COMPUTE
  ClassBlock.ScopeKey = DefIdent.Key;
  ClassBlock.GotInh = Inheritances CONSTITUENTS InheritIdent.InheritOk;
  Inheritances.ToEnv = ClassBlock.Env;
END;

SYMBOL Inheritances:    ToEnv: Environment;
```

This macro is attached to a product file.

The `Inheritances` affect the scope of the class body, like the `WithBody` in the example above. As there may be several `Inheritances` the attribute `Inheritances.ToEnv` is accessed from each. The `InheritIdent` has the role `InheritScope` provided by the library module for scope properties already used above. It adds each single inheritance to the inheritance relation of the class scope specified by `INCLUDING Inheritances.ToEnv`.

The role `InheritScope` yields an attribute `InheritOk`. It indicates whether the inheritance relation is not cyclic. It is used to issue an error message on violation.

ClassInherit.lido[30]==

```
SYMBOL InheritIdent INHERITS
        InheritScope, ChkModUseIdent,
        IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.InnerScope = INCLUDING Inheritances.ToEnv;
  SYNT.OuterScope = GetScope (THIS.Key, NoEnv)
        <- INCLUDING RootScope.GotScopeProp;
```

```
    IF (AND (NOT (THIS.InheritOk), NE (THIS.Key, NoKey)),
    message (FATAL, CatStrInd ("cyclic inheritance: ", THIS.Sym),
             0, COORDREF))
    ;
  END;
```

This macro is attached to a product file.

It has also to be checked that the `InheritIdent` is bound to an object (class or module) that has a scope property. The role `ChkModUseIdent` defined above is reused for that purpose.

The library module for scope properties ensures that all relevant inheritances are considered, all relevant scope properties are associated, and all relevant definitions are encountered, before applied identifier occurrences are bound in the class body or in qualified accesses.

We now introduce an additional uniqueness requirement for inheritance: If an applied identifier occurrence is bound to a definition in an inherited environment there must not be a not hidden binding of that identifier in another inherited environment. Such an alternative binding is checked by `ChkInhIdUse` for all `IdUseEnv` occurrences and by `ChkInhIdUseScopeProp` for all `IdUseScopeProp` occurrences:

UniqueInherit.lido[31]==

```
SYMBOL UseIdent INHERITS ChkInhIdUse END;
SYMBOL TypeUseIdent INHERITS ChkInhIdUse END;
SYMBOL QualIdent INHERITS ChkInhIdUseScopeProp END;
SYMBOL SelectIdent INHERITS ChkInhIdUseScopeProp END;
```

This macro is attached to a product file.

11 Objects Having a Scope Type

This example demonstrates a typical situation where the tasks of name analysis and type analysis are interleaved. Since type analysis is not the topic of this tutorial, we concentrate on one aspect where it affects name analysis.

We extend our language by class variables. Such a variable is declared by `v : c` where `c` is a class identifier. The variable `v` is a structure that has the components declared for `c` and for the classes inherited by `c`.

With this extension class declarations can be considered as declarations of type names which are used as type identifiers in variable declarations.

In order to access the components of a class variable, we introduce a selection construct that is similar to the qualified access construct:

ScopeType.con[32]==

```
Operand:        UseIdent '.' SelectIdent.
SelectIdent:    Ident.
```

This macro is attached to a product file.

We here specify a very simple version of type analysis: Types are represented by `DefTableKeys`. A property `TypeOf` associates a type with an object key:

ScopeType.pdl[33]==

```
TypeOf: DefTableKey;
```

This macro is attached to a product file.

The following computational roles specify how the `TypeOf` property is set and accessed in proper order:

TypeModule.lido[34]==

```
ATTR Type: DefTableKey;

CLASS SYMBOL RootType COMPUTE
  SYNT.GotType = CONSTITUENTS SetType.GotType;
END;

CLASS SYMBOL SetType COMPUTE
  SYNT.GotType = ResetTypeOf (THIS.Key, INH.Type);
END;

CLASS SYMBOL GetType COMPUTE
  SYNT.Type = GetTypeOf (THIS.Key, NoKey)
        <- INCLUDING Program.GotType;
END;
```

This macro is attached to a product file.

Usually the defining occurrences of identifiers, `DefIdent` in our language, are the contexts where the type of the object is specified. Hence they have the role of `SetType`.

As we here are only interested in types of variables, we specify a default unknown type represented by `NoKey`. In variable declarations the type of the declared identifier is specified to be the key of the type identifier.

In the context of applied identifier occurrences, `UseIdent`, their type may be used for further analysis. They have the role `GetType`.

ScopeType.lido[35]==

```
SYMBOL DefIdent INHERITS SetType COMPUTE
  INH.Type = NoKey;
END;

RULE: VarDecl ::= TypeUseIdent DefIdent COMPUTE
  DefIdent.Type = TypeUseIdent.Key;
END;

SYMBOL UseIdent INHERITS GetType END;

SYMBOL Program INHERITS RootType END;
```

This macro is attached to a product file.

The select construct combines the technique of using a scope property, as introduced for qualified access (`QualIdent` above), and type analysis: `SelectIdent` has the `IdUseScopeProp` role. The identifier is bound in the scope associated with the type of the variable identifier. `SelectIdent.Scope` is specified to be that scope.

SelectType.lido[36]==

```
SYMBOL SelectIdent INHERITS
        IdUseScopeProp,
        ChkIdUse, IdentOcc
END;

RULE: Expression ::= UseIdent '.' SelectIdent COMPUTE
  SelectIdent.Scope = GetScope (UseIdent.Type, NoEnv)
        <- INCLUDING RootScope.GotScopeProp;

  IF (EQ (SelectIdent.Scope, NoEnv),
  message (FATAL, "module variable required for selection",
           0, COORDREF))
  ;
END;
```

This macro is attached to a product file.

Similar to previous examples we have to check that the type of the variable really allows selection. The precondition of the check guarantees that all scope properties are set. It is established by the library module.