Name analysis according to scope rules

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.20 $

Table of Contents

Languages usually use names to identify objects. An object is created by an explicit or implicit definition and bound to a name. In a certain range of the text occurrences of that name refer to that object. The scope rules of the language determine where that binding holds. For language implementation a unique name (key) is created for each distinct object and associated to identifiers within the scope of that binding. Name analysis is completed by certain checks of relationships between identifier occurrences as required by the language, e.g. existence of a definition for each identifier use, multiple definitions (See Section 'Common Aspects of Property Modules' in Association of properties to definitions, see Section 'Check for Unique Object Occurrences' in Association of properties to definitions), or identifier use before its definition (see Section 'Set a Property at the First Object Occurrence' in Association of properties to definitions).

This library contains a set of modules which can be used to implement the name analysis task according to a large variety of language rules. The results of these modules are used to check required relationships between identifier occurrences and to solve further subtasks of language implementation, such as type analysis or transformation. Solutions of these tasks are supported by modules of other libraries: Section 'Property Library' in Association of properties to definitions, Section 'Type Analysis' in Type analysis tasks, Section 'Generating Output' in Tasks related to generating output.

The module support for name analysis is decomposed into subtasks of increasing complexity. They are described in subsections each. There you find three solution variants for Algol-like, C-like scope rules, and C-like scope rules computed bottom-up while the input is read:

The chapter is structured as follows:

`Preconditions`
: Requirements for the user's tree grammar

`Basic Scope Rules`
: Modules for basic scope rules

`Predefined Identifiers`
: Modules that introduce predefined entities

`Joined Ranges`
: Several subtrees form one conceptual range

`Scope Properties`
: Scopes being propagated as object properties

`Inheritance of Scopes`
: Scopes inherited by other scopes

`Name Analysis Test`

Test output for name analysis

`Environment Module`

Implementation of the Contour-Model

The use of these modules is demonstrated and explained in three complete executable specifications called

```
AlgLike.fw,
CLike.fw, and
BuCLike.fw
```

You can obtain a copy of these specifications by calling Eli and requesting

```
$elipkg/Name/Examples > .
```

This request creates a subdirectory `Examples` in your current working directory containing the example specifications, test input files, and an Odinfile for automatic regression testing.

1 Tree Grammar Preconditions

Names are usually represented by identifier terminals. Their notation is determined by a scanner specification. The grammar has one (or several) terminals representing identifiers, e.g. `Ident`, as in the running example. The encoding of a particular identifier as computed by a scanner processor is available in contexts where an identifier terminal occurs.

Identifiers occur in different contexts: Defining and applied occurrences, or different kinds of identifiers (variables, labels, etc.) may be distinguished. Usually the concrete syntax is designed first, and the different computational roles of identifiers are incrementally developed during the design of the `.lido` specification. Hence, it is recommended NOT to make the distinction in the concrete syntax. It should have the terminal `Ident` in any context. It is rather recommended to distinguish them by LIDO `RULE`s.

Our running example has the concrete productions

```
ObjDecl:        TypeDenoter Ident.
TypeDenoter:    Ident.
Variable:       Ident.
```

We distinguish the different roles of identifiers by introducing new symbol names in the corresponding LIDO `RULE`s:

```
RULE: ObjDecl     ::= TypeDenoter DefIdent END;
RULE: TypeDenoter ::= TypeUseIdent END;
RULE: Variable    ::= UseIdent END;
```

Furthermore, we have to add the necessary chain `RULE`s:

```
RULE: DefIdent     ::= Ident END;
RULE: UseIdent     ::= Ident END;
RULE: TypeUseIdent ::= Ident END;
```

The name analysis modules require that identifier occurrences are represented by nonterminals, like `DefIdent`, `UseIdent`, `TypeUseIdent` as in the example. Each of these symbols has to have an attribute named `Sym` of type `int` representing the identifier encoding. A specification using these modules has to contain suitable computations of the `Sym` attributes. For our example they may be specified like:

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

SYMBOL DefIdent     INHERITS IdentOcc END;
SYMBOL UseIdent     INHERITS IdentOcc END;
SYMBOL TypeUseIdent INHERITS IdentOcc END;
```

If your language does not syntactically distinguish between defining and applied identifier occurrences, i.e. objects are introduced by using their name, the above distinction is not necessary. You just introduce `DefIdent` symbols for all occurrences.

Your grammar should have a symbol representing a phrase that contains all (defining and applied) occurrences of a name space. It is usually the root of the whole grammar, e.g. in the running example the symbol `Program`.

If your language has hierarchically nested ranges defining boundaries for the scope of definitions, the abstract syntax should have one or several symbols, e.g. `Range`, `Block`, `Routine`, each representing a range of a name space. If the language does not have nested ranges for definitions you don't need such symbols.

2 Basic Scope Rules

The consistent renaming task associates to each identifier occurrence a key that uniquely identifies the object named by the identifier. The following modules solve the basic problems of that task.

`AlgScope` Algol-like scope rules

`CScope` C-like scope rules

`BuScope` C-like scope rules analyzed while processing input

Each of the three modules implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey`.

The `AlgScope` module applies Algol-like scope rules. They are characterized by the following description:

A binding is valid within the whole smallest range containing the definition, except in inner ranges where a binding for the same identifier holds. That means a definition of an `a` in an inner range hides definitions of `a` in outer ranges. An identifier may be used before its definition.

Usually, the scope rules of a real language are further elaborated. We call them Algol-like, if the above description is their underlying principle. For example Pascal's scope rules are Algol-like. They additionally require that an identifier is not used before its definition. That restriction can be checked using an instance of the `SetFirst` module (see Section 'Set a Property at the First Object Occurrence' in Association of properties to definition).

The `CScope` module applies C-like scope rules. They are characterized by the following description:

A binding is valid from the definition upto the end of the smallest range containing the definition, except in inner ranges from a definition of the same identifier to the end of that range. That means definitions of `a` in outer range are hidden by a definition of an `a` in an inner range from the point of the definition upto the end of the range. It implies that an identifier is not used before its definition.

Usually, the scope rules of a real language are further elaborated. We call them C-like, if the above description is their underlying principle. For example the scope rules of the language C are defined for variable names C-like. But for labels names of jumps they are defined Algol-like.

The `BuScope` module applies C-like scope rules. Its computations can be executed while the input is read (i.e. while the tree is constructed bottom-up). An application may need this technique if results of the name analysis task influence further reading of input, or results are to be presented to the user while typing the input.

Both Algol-like and C-like scope rules are described by six basic concepts. The modules provide `.lido` specifications with symbol computations for each concept:

`IdDefScope` is a symbol representing a defining identifier occurrence that is bound in the scope of the smallest enclosing range.

`IdUseEnv` is a symbol representing an applied identifier occurrence that is bound in the enclosing environment.

`IdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range.

`ChkIdUse` is a role that may be inherited by an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`RootScope` is the root symbol containing all identifier occurrences and all `RangeScope`. It is automatically inherited by the root of the grammar.

`RangeScope` is a symbol representing a range for the binding of defining identifier occurrences `IdDefScope`. It may be nested in `RootScope` or other ranges.

These computational roles are associated to symbols of the user's grammar to solve the basic consistent renaming task. Make sure that your tree grammar is constructed according to the advices given in Chapter 1 [Preconditions], page 3. More details about symbol computations and attributes provided by the three modules are given in the description of each module.

Complete executable specifications of our running example for each of the three scope rule variants are available in

```
$/Name/Examples/AlgLike.fw
$/Name/Examples/CLike.fw
$/Name/Examples/BuCLike.fw
```

In our running example the roles are used as follows:

```
SYMBOL Program      INHERITS RootScope END;
SYMBOL Block        INHERITS RangeScope END;
SYMBOL DefIdent     INHERITS IdDefScope END;
SYMBOL UseIdent     INHERITS IdUseEnv END;
SYMBOL TypeUseIdent INHERITS IdUseEnv END;
```

Depending on which of the three modules is instantiated Algol-like or C-like name analysis is performed for this example.

The main result of the task is the computation of the attributes `IdDefScope.Key`, `IdUseEnv.Key`, i.e. `DefIdent.Key`, `UseIdent.Key`, and `TypeUseIdent.Key` in our example. They identify the object each identifer is bound to. It may be used in further computations to associate properties to it.

If no binding is found for an applied identifier occurrence the `Key` attribute has the value `NoKey`. If that is a violation of language rules an error message can be issued using the role `ChkIdUse`:

```
SYMBOL UseIdent INHERITS ChkIdUse END;
```

Along with each `Key` attribute there is an attribute `Bind` of type `Binding`, e.g. `UseIdent.Bind`. Its value characterizes a binding of an identifer `idn` in the innermost scope of an environment `env` to a key `k`. The three values `idn`, `env`, and `k` can be obtained from a `Binding` using functions defined in Chapter 8 [Environment Module], page 44. If no binding is found for an applied identifier occurrence the `Bind` attribute has the value `NoBinding`.

Although both Algol-like and C-like scope rules are defined for nested ranges, the modules may be used for languages that do not have nested ranges, i.e. there is only one single flat range in which definitions are valid. In such a case `RootScope` is used for that range, and `RangeScope` is not used.

Another variant of these scope rules arises if a language does not distinguish between defining and applied identifier occurrences: identifiers are defined implicitly by their occurrences. In that case `IdDefScope` is used for that kind of occurrences, and `IdUseEnv` is not used. Of course, this concept does not make sense in languages that have ranges: One could not refer to an outer identifier definition from within an inner range.

We extend our running example to show implicit definitions within a flat range. We add a new kind of variables, say control variables to the language. They are implicitly defined by their use in special statements or operands, given by the following concrete productions:

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

These control variable identifiers are bound in a new name space separate from that of the other entities. Hence, we use a second instance of one of the modules. That instance is identified by the generic `instance` parameter `CtrlVar`:

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

There is only one kind of occurrences for these variables, `CtrlVarUse`, which has the role `CtrlVarIdDefScope`. The `Program` symbol has the role `CtrlVarRootScope`:

```
RULE:  Statement  ::= 'set' CtrlVarUse 'to' Expression ';' END;
RULE:  Expression ::= 'use' CtrlVarUse END;
SYMBOL CtrlVarUse INHERITS CtrlVarIdDefScope, IdentOcc END;
SYMBOL Program    INHERITS CtrlVarRootScope END;
```

2.1 Algol-like Basic Scope Rules

This module implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to Algol-like scope rules:

A binding is valid within the whole smallest range containing the definition, except in inner ranges where a binding for the same identifier holds.

Make sure that you have considered the advices given in Chapter 2 [Basic Scope Rules], page 5.

The module is instantiated by

```
$/Name/AlgScope.gnrc+instance=NAME +referto=KEY :inst
```

Both generic parameters can be omitted in most of the usual applications. The `instance` parameter is used to distinguish several instances of this module. The scope rules of a language may require that identifiers are bound in different name spaces that do not affect each other. Then for each name space an instance of this or of the other basic scope modules is used. The `referto` parameter modifies the names of `Key` attributes and of `Bind` attributes. It is only used if there is an identifier occurrence in the language that is bound in more than one name space. These bindings are then described by one pair of `Key` and `Bind` attribute each.

The module provides computational roles for the symbols `NAMERootScope`, `NAMERangeScope`, `NAMEAnyScope`, `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, and `NAMEChkIdUse` to be used in `.lido` specifications. The computations of the module use functions of the library's environment module.

`NAMEIdDefScope` is a symbol representing a defining identifier occurrence.

`NAMEIdUseEnv` is a symbol representing an applied identifier occurrence.

`NAMEIdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range. The outer environment of this range is not considered.

`NAMEChkIdUse` is a role that may be inherited by an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`NAMERootScope` is the root symbol containing all identifier occurrences and all `NAMERangeScope`. It is automatically inherited by the root of the grammar.

`NAMERangeScope` is a symbol representing a range for the binding of defining identifier occurrences `NAMEIdDefScope`. It may be nested in `NAMERootScope` or in other ranges.

`NAMEAnyScope` comprises the roles of `NAMERootScope` and `NAMERangeScope`. It may be used in constructs like

```
INCLUDING NAMEAnyScope.NAMEGotKeys
```

The main results of using this module are the bindings of identifier occurrences represented by the attibutes `NAMEIdDefScope.KEYKey` and `NAMEIdUseEnv.KEYKey`. Along with each `Key` attribute there is an attribute `KEYBind` of type `Binding`, e.g. `UseIdent.Bind`. Its value characterizes a binding of an identifer `idn` in the innermost scope of an environment `env` to a key `k`. The three values `idn`, `env`, and `k` can be obtained from a `Binding` using functions defined in Chapter 8

[Environment Module], page 44. If no binding is found for an applied identifier occurrence the `Bind` attribute has the value `NoBinding`.

Usually both `NAMEIdDefScope` and `NAMEIdUseEnv` are used. In specific cases of language rules any combination of `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope` may be used.

The attributes `NAMEIdDefScope.Sym`, `NAMEIdUseEnv.Sym`, `NAMEIdUseScope.Sym` must represent the identifier encoding.

`NAMERootScope.NAMEEnv` is a root environment where all environments of this name space are embedded in. It has the value of a global variable `NAMERootEnv` that is assigned in the initialization phase of the processor. It allows to introduce predefinitions by initialization code, which then must include the file `NAMEAlgScope.h`. (see Chapter 3 [Predefined Identifiers], page 16)

`NAMERangeScope.NAMEEnv` is an inherited attribute for the environment of bindings of this range.

`NAMEAnyScope.NAMEGotKeys` indicates that all keys defined in this and in all enclosing ranges are defined in `NAMEAnyScope.NAMEEnv`. `NAMEAnyScope.NAMEGotKeys` is a precondition for finding a binding using `IdUseEnv`.

`NAMEAnyScope.NAMEGotLocKeys` indicates that all keys are defined in this range are in `NAMEAnyScope.NAMEEnv`.

2.2 C-like Basic Scope Rules

This module implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to C-like scope rules:

A binding is valid from the definition upto the end of the smallest range containing the definition, except in inner ranges from a definition of the same identifier to the end of that range.

Note: The scope rules of the programming language C are defined C-like in most but not in all respects: For example the scopes of names of variables and functions are defined C-like, but those of jump labels are defined Algol-like.

Make sure that you have considered the advices given in Chapter 2 [Basic Scope Rules], page 5.

The module is instantiated by

```
$/Name/CScope.gnrc+instance=NAME +referto=KEY :inst
```

Both generic parameters can be omitted in most of the usual applications. The `instance` parameter is used to distinguish several instances of this module. The scope rules of a language may require that identifiers are bound in different name spaces that do not affect each other. Then for each name space an instance of this or of the other basic scope modules is used. The `referto` parameter modifies the names of `Key` attributes. It is only used if there is an identifier occurrence in the language that is bound in more than one name space. These bindings are then described by one `Key` attribute each.

The module provides computational roles for the symbols `NAMERootScope`, `NAMERangeScope`, `NAMEAnyScope`, `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, `NAMEChkIdUse`, `NAMEIdDefUse`, `NAMEDeclaratorWithId`, and `NAMEIdInDeclarator` to be used in `.lido` specifications. The computations of the module use functions of the library's environment module.

`NAMEIdDefScope` is a symbol representing a defining identifier occurrence.

`NAMEIdUseEnv` is a symbol representing an applied identifier occurrence.

`NAMEChkIdUse` is a role that may be inherited by an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`NAMEIdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range. The outer environment of this range is not considered.

`NAMEIdDefUse` represents a defining identifier occurrence like `NAMEIdDefScope` if `INH.NAMEDefCond` is non-zero, otherwise an applied occurrence like `NAMEIdUseEnv`. `NAMEDefCond` is to be computed by an upper computation. There is a default computation provided that sets `INH.NAMEDefCond` to 1 iff the identifier is not yet bound in the current environment.

The pair of roles `NAMEDeclaratorWithId` and `NAMEIdInDeclarator` are used to model the scope concept of declarators as defined in the programming language C: A defining occurrence of an identifer may be part of *Declarator*, that is a larger construct which determines the type of the defined identifer, for example the definition of the array `a` in

```
int a[a+1];
```

Here `a[a+1]` is the `Declarator` and the first `a` is its defining occurrence. The scope rules of C state that the scope of the defined identifier begins immediately after the end of the declarator, rather than at the position of the defining occurrence. Hence, the `a` within the brackets is *not* bound to the defined array. This rule is only relevant if declarators may contain applied identifier occurrences. To achieve this effect, the role `NAMEDeclaratorWithId` is to be inherited by a symbol which is the root of the declarator construct, and the role `NAMEIdInDeclarator` is inherited by the symbol that characterizes defining identifier occurrences within declarators. Make sure that the grammar guarantees a 1:1 relation the nodes of these symbol roles in any declarator tree. The attribute `NAMEIdInDeclarator.Sym` has to be provided as usual. The the symbol roles compute the the `Sym` attribute for `NAMEDeclaratorWithId` and the `KEYKey` attribute for both symbols.

`NAMERootScope` is the root symbol containing all identifier occurrences and all `NAMERangeScope`. It is automatically inherited by the root of the grammar.

`NAMERangeScope` is a symbol representing a range for the binding of defining identifier occurrences `NAMEIdDefScope`. It may be nested in `NAMERootScope` or other ranges.

`NAMEAnyScope` comprises the roles of `NAMERootScope` and `NAMERangeScope`. It may be used in constructs like

```
INCLUDING NAMEAnyScope.NAMEEnv
```

The main results of using this module are the bindings of identifier occurrences represented by the attibutes `NAMEIdDefScope.KEYKey` and `NAMEIdUseEnv.KEYKey`.

Along with each `Key` attribute there is an attribute `Bind` of type `Binding`, e.g. `UseIdent.Bind`. Its value characterizes a binding of an identifer `idn` in the innermost scope of an environment `env` to a key `k`. The three values `idn`, `env`, and `k` can be obtained from a `Binding` using macros defined in Chapter 8 [Environment Module], page 44. If no binding is found for an applied identifier occurrence the `Bind` attribute has the value `NoBinding`.

Usually both `NAMEIdDefScope` and `NAMEIdUseEnv` are used. In specific cases of language rules any combination of `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, `NAMEIdDefUse` may be used.

The attributes `NAMEIdDefScope.Sym`, `NAMEIdUseEnv.Sym`, `NAMEIdUseScope.Sym`, `NAMEIdDefUse.Sym` must represent the identifier encoding.

`NAMERootScope.NAMEEnv` is a root environment where all environments of this name space are embedded in.

It has the value of a global variable `NAMERootEnv` that is assigned in the initialization phase of the processor. It allows to introduce predefinitions by initialization code, which then must include the file `NAMECScope.h`. (see Chapter 3 [Predefined Identifiers], page 16)

`NAMERangeScope.NAMEEnv` is an inherited attribute for the environment of bindings of this range.

`NAMEAnyScope.NAMEGotKeys` indicates that all identifer ocurrences from the begin of the `NAMERootScope` upto the end of this range are bound to keys in `NAMEAnyScope.NAMEEnv`.

2.3 C-like Basic Scope Rules Computed Bottom-Up

This module implements consistent renaming of identifiers. The computations of this module are specified such that they are executed while the input program is read. Identifier occurrences are bound to object keys of type `DefTableKey` according to C-like scope rules.

Make sure that you have considered the advices given in Chapter 2 [Basic Scope Rules], page 5.

The module is instantiated by

```
$/Name/BuScope.gnrc+instance=NAME +referto=KEY :inst
```

The functionality provided by this modules is almost the same as that of the `CScope` module (see Section 2.2 [CScope], page 10). Only the differences are described here.

During the bottom-up computation phase values can not be propagated using inherited (`INH`) attributes; and computations that affect a whole subtree, like creation of the scope for a range, have to be associated to a symbol node that precedes that subtree. For that purpose this module provides additional computational roles that cooperate with the usual basic name analysis roles.

Usually it is necessary to introduce additional symbols into the concrete grammar preceding range subtrees. They derive to nothing, and are used to carry the specific bottom-up computations.

In our running example we would replace the productions

```
Source:         Block.
Statement:      Block.
```

by the productions

```
Source:         BuBlock Block.
Statement:      BuBlock Block.
```

and add

```
SYMBOL BuBlock INHERITS CreateNewScope, OpenNewScope END;
```

to the LIDO specification. All other module roles can be used as described for C-like scope rules.

Usually each symbol representing a NAMERangeScope has to be preceded by a symbol that inherits both roles NAMECreateNewScope and NAMEOpenNewScope.

If scopes are used as properties of objects it may be necessary to inherit the roles NAMECreateNewScope, NAMERecentNewScope, and NAMEOpenNewScope to different symbols which precede a symbol representing a NAMERangeScope. (see Section 5.4 [BuScopeProp], page 29, see Section 6.3 [BuInh], page 38)

`NAMECreateNewScope` creates a new scope that is embedded in the scope of the smallest enclosing range. That scope can be obtained from the attribute `SYNT.NAMENewScope`, or be accessed by a subsequent role `NAMERecentNewScope` (see below).

`NAMEOpenNewScope` makes the scope obtained from `SYNT.NAMENewScope` become the current scope. The attribute `SYNT.NAMEOpenPrecond` can be used to specify a precondition for this operation. If `NAMEOpenNewScope` is inherited by a symbol representing an identifier occurrence `SYNT.NAMEOpenPrecond = THIS.KEYKey` ensures that the identifier is bound before the new scope is opened.

`NAMERecentNewScope` accesses the most recently created new scope and provides it by the attribute `SYNT.NAMENewScope`. This role is used together with `NAMEOpenNewScope` if `NAMECreateNewScope` is inherited by a preceding symbol.

We demonstrate the use of these roles for our running example. The grammar introduced in See Section 'Running Example' in Introduction of specification modules, has to be modified in order to allow bottom-up computation. A new symbol `BuBlock` is introduced. It derives to nothing and precedes the symbol `Block` on right-hand sides of productions:

```
Source:        BuBlock Block.
Statement:     BuBlock Block.
```

The roles `CreateNewScope` and `OpenNewScope` introduce the scope for the subsequent `Block`:

```
SYMBOL Program  INHERITS RootScope END;
SYMBOL Block    INHERITS RangeScope END;
SYMBOL BuBlock  INHERITS CreateNewScope, OpenNewScope END;
```

The other roles for basic scope rules are used as described in See Section 2.2 [CScope], page 10.

3 Predefined Identifiers

In most languages some identifiers are predefined, e.g. names for basic types or for constants like `true` and `false`. Their definitions are valid in any program as if they were bound in the outermost environment. The two modules `PreDefine` and `PreDefId` described here allow to easily introduce such predefinitions. They require that one of the basic scope rule modules (see Chapter 2 [Basic Scope Rules], page 5) is used.

Both modules `PreDefine` and `PreDefId` are to be instantiated to introduce a set of predefined entities in a name space.

The implementation of the modules use two functions which introduce a source identifier into the identifier table and establish a binding for it in some environment. These functions can be used directly for example in cases where predefinitions are to be established for other environments than the outermost one. Those functions are described below.

The `PreDefine` module is instantiated by

```
$/Name/PreDefine.gnrc +instance=NAME +referto=IDENT :inst
```

The optional `instance` parameter characterizes the name space in which identifiers are to be predefined. The `instance` parameter has to be the same as that of the basic scope rule module instance used for that name space. Several instances of this module may address different name spaces.

The `referto` parameter specifies the symbol name used for identifier terminals in the grammar. The `referto` parameter must not be omitted.

If a grammar has several identifier terminal symbols predefinitions can be made using several instances of this module, if they belong to different name spaces.

The module provides two functions `NAMEPreDefine` and `NAMEPreDefineSym` which are called by the instance of the `PreDefId` module. `NAMEPreDefineSym` inserts a string into the identifier module to be used as an `IDENT` symbol. `NAMEPreDefine` additionally binds that symbol to a key in the root environment given by the global variable `NAMERootEnv`.

The predefined identifiers are to be described in a file as explained below. The name of that file has to be given as `referto` parameter of the instantiation of the `PreDefId` module:

```
$/Name/PreDefId.gnrc +instance=NAME +referto=(FILENAME) :inst
```

The `instance` parameter has to be the same as that of the `PreDefine` instance. If this instantiation is contained in a `.specs` file and if the description file, say `Predef.d` is contained in the same directory, it may read

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

This can also be used if the `.specs` file and `Predef.d` are contained in a `.fw` specification.

The description file contains a sequence of macro calls, one for each predefined identifier, e.g.

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
PreDefSym ("external", externSym)
PreDefSymKey ("fail", failSym, failKey)
PreDefSymKeyBind ("write", writeSym, writeKey, writeBind)
```

The sequence should not contain anything else, because it is expanded in several contexts where different definitions of those macros are valid.

Each call of one of the macros establishes a predefinition for one identifier, and makes the result accessable via the supplied variable names. Usually not all of those variables are needed. Hence, the available macros differ in the combinations of those variables. We first explain the most general macro. The meanings of the other macros are deduced from it.

`PreDefSymKeyBind ("xxx", sym, key, bind)` encodes the character string `xxx` as an identifer, stores it in the identifier table, and stores the encoding in the `int` variable `sym`.

Note: The string need not obey the rules specified for the notation of `IDENT` symbols. That facility can be used if artifacts are predefined, which can not be referred to by a name in a program.

`key` is introduced as a PDL known key.

`key` is bound to `sym` in the environment `NAMERootEnv`. That binding is assigned to the `Binding` variable `bind`. The key, the identifier code, and the environment can be accessed from the `Binding` value (`KeyOf`, `IdnOf`, `EnvOf`).

The variables `sym` and `bind` and the known key `key` are defined, exported, and made accessable via a `.HEAD.phi` specification. The binding is established and the assignments are made in the initialization phase of the processor. Hence, the results can be used only after that phase, i.e. during all computations in the tree.

According to the above description the following macro call

```
PreDefSymKeyBind ("write", writeSym, writeKey, writeBind)
```

creates the following variables to be defined and initialized as described:

```
int writeSym;
DefTableKey writeKey;
Binding writeBind;
```

The other macros that are provided cause a subset of the effects described for `PreDefSymKeyBind`:

`PreDefSymKey ("xxx", sym, key)` As described above, except: The binding is established but not assigned to a variable.

`PreDefKeyBind ("xxx", key, bind)` As described above, except: The symbol is encoded and stored in the identifier table, but the encoding is not assigned to a variable.

`PreDefKey ("xxx", key)` As described above, except: Neither the symbol encoding nor the binding are stored in a variable.

`PreDefBind ("xxx", bind)` As described above, except: The symbol encoding is not stored in a variable. The key is created dynamically rather than as a known key. Both, symbol encoding and the key can be accessed via the stored `Binding` value.

`PreDefSym ("xxx", sym)` encodes the character string `xxx` as an identifer, stores it in the identifier table, and stores the encoding in the `int` variable `sym`. No binding is established.

The thus introduced variables and known keys may be used in `.lido` specifications; the known keys may be additionally used in any specification where PDL defined entities are available.

The described modules are based on a C module which provides the following two functions. They may be used directly to establish bindings in other environments than the outermost one, for example. In that case it is sufficient to use the module `PreDefMod.specs`. Then the modules `PreDefine` and `PreDefId` need not be instantiated, if the macros explained above are not used.

The two functions are:

`void PreDefineSym (char *name, int code, int *sym)`
: The string `name` is encoded with the given syntax `code`. That is usually the code of the symbol used for identifier terminals in the grammar (cf. the `referto` parameter in the instantiation of the module `PreDefine` explained above). `*sym` is set to the symbol index.

`void PreDefine (char *name, int code, int *sym, Environment env, DefTableKey key,Binding *bind)`
: The string `name` is encoded with the given syntax `code` which is bound to `key` in the given environment `env`. `*sym` is set to the symbol index. `*bind` is set to the created binding, if successful, otherwise to `NoBinding`.

In our running example we introduce predefined names for some basic types and for boolean constants by the module instantiations

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

The file `Predef.d` contains

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

Then key names like `intKey` can be used e.g. in computations for type checking (see Section 'Type Analysis' in Type analysis tasks). In that case it is necessary to state that `true` and `false` are of type bool in a `.pdl` specification:

```
trueKey -> TypeOf = {boolType};
falseKey -> TypeOf = {boolType};
```

It associates the `TypeOf` property to the predefined objects.

4 Joined Ranges

In some situations it is not possible to specify the tree grammar such that each range in the sense of scope rules is rooted by one single grammar symbol as required for using the role `RangeScope` of the basic scope module. The following three modules extend the basic scope rule modules (see Chapter 2 [Basic Scope Rules], page 5) by facilities that support such cases:

`AlgRangeSeq`
: Joined Ranges Algol-like

`CRangeSeq`
: Joined Ranges C-like

`BuRangeSeq`
: Joined Ranges C-like Bottom-up

Using one of these modules requires that the corresponding basic scope rule module is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

This module implements the following concept: Several ranges in the program form one single range in the sense of scope rules, i.e. the definition in these ranges contribute to a single scope in which the applied identifier occurrences of these ranges are bound. There is a symbol which is the subtree root for all these ranges. But it is not a range in the sense of scope rules because it may also contain identifier occurrences that are bound in the enclosing range.

The modules provide `.lido` specifications for the following computational roles:

`NAMERangeSequence` is to be inherited by a symbol that is the root of a subtree which contains all to be joined ranges. It provides all attributes provided by `NAMERangeScope`, but it is not a range in the sense of scope rules. Other range roles may not be inherited by `NAMERangeSequence`.

`NAMERangeElement` is the role to be inherited by symbols that represent ranges to be joined. It is a specialized `NAMERangeScope`. Other range roles may not be inherited by such a symbol. It is a specialized `NAMERangeScope` that must be contained in a `NAMERangeSequence` subtree without having a `NAMERangeScope` in between.

We demonstrate the use of these facilities by extending the language of our running example by introducing an artificial language construct: It consists of a defining identifier occurrence, that is to be bound in the enclosing range, and two compound statements which form one single range in the sense of scope rules:

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

Hence the `Join` symbol has the role `RangeSequence`, and the `JoinedBlock` symbol has the role `RangeElement`:

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

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

This example is applicable with either the `AlgRangeSeq` module or the `CRangeSeq`. In case of Algol-like scope rules an applied identifier occurrence in either of the two ranges may be bound to a definition in either of the two ranges. In case of C-like scope rules an applied identifier occurrence in the second of the two ranges may be bound to a definition in either of the two ranges.

In case of bottom-up computations using the `BuRangeSeq` module some modifications have to be applied as described for that module.

4.1 Joined Ranges Algol-like

This module implements joined ranges for Algol-like scope rules as described in (see Chapter 4 [Joined Ranges], page 20).

The module is instantiated by

```
$/Name/AlgRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `AlgScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence` and `NAMERangeElement` as described in See Chapter 4 [Joined Ranges], page 20.

4.2 Joined Ranges C-like

This module implements joined ranges for C-like scope rules as described in (see Chapter 4 [Joined Ranges], page 20).

The module is instantiated by

```
$/Name/CRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `CScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence` and `NAMERangeElement` as described in See Chapter 4 [Joined Ranges], page 20.

4.3 Joined Ranges C-like Bottom-up

This module implements joined ranges for C-like scope rules as described in (see Chapter 4 [Joined Ranges], page 20). Its computations are executed `BOTTOMUP` while the input is read.

The module is instantiated by

```
$/Name/BuRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `BuScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence`, `NAMEOpenSeqScope` and `NAMEOpenElemScope`.

`NAMERangeSequence` is to be inherited by a symbol that is the root of a subtree which contains all to be joined ranges. It provides all attributes provided by `NAMERangeScope`, but it is not a range in the sense of scope rules. Other range roles may not be inherited by `NAMERangeSequence`.

`NAMEOpenSeqScope` is to be inherited by a symbol that is in the subtree of `NAMERangeSequence` and precedes all to be joined ranges.

`NAMEOpenElemScope` is to be inherited by a symbol that precedes each to be joined range.

A nested `NAMERangeSequence` may not occur between `NAMEOpenSeqScope` and the first `NAMEOpenElemScope`.

The example given in See Chapter 4 [Joined Ranges], page 20 is here modified for the bottom-up case.

We demonstrate the use of these facilities by extending the language of our running example by introducing an artificial language construct: It consists of a defining identifier occurrence, that is to be bound in the enclosing range, and two compound statements which form one single range in the sense of scope rules.

We introduce two new symbols `BuJoin` and `BuJoinEl` that derive to empty.

```
Statement:      Join.
Join:           'join' BuJoin DefIdent
                       BuJoinEl Block BuJoinEl Block
                'joined' ';'.
BuJoin:         .
BuJoinEl:       .
```

The module roles are inherited as described above:

```
RULE: Join ::= 'join' BuJoin DefIdent
                      BuJoinEl Block BuJoinEl Block
               'joined' ';'
END;

SYMBOL Join INHERITS RangeSequence END;
SYMBOL BuJoin INHERITS OpenSeqScope END;
SYMBOL BuJoinEl INHERITS OpenElemScope END;
SYMBOL Block INHERITS RangeScope END;
```

5 Scopes Being Properties of Objects

Language constructs like modules, classes, or record types have a body that is a range. The set of bindings for the components defined in that range constitutes its scope. In an applied context of a module, class, or record identifier its components may be selected, e.g. in `m.k`, where `m` is a module identifier and `k` is one of its components. These constructs are also called *qualified names* in some language descriptions. In order to bind such applied occurrences of component identifiers in contexts outside their defining range, the scope of the range is associated as a property to the key of the module, class, or record identifier.

This specific task of consistent renaming for component identifiers is often closely related to type analysis. If `v` in `v.k` is a variable that has a record type, then that type key has the scope of the record range associated as a property (see Section 'Type Analysis' in Type analysis tasks).

The following four modules extend the basic scope rule modules (see Chapter 2 [Basic Scope Rules], page 5) by facilities that support scope properties. How to select one of the modules is explained below.

`ScopeProp`
: Scope Properties without Ordering Restrictions

`AlgScopeProp`
: Scope Properties Algol-like (outdated)

`CScopeProp`
: Scope Properties C-like (recommended only with CInh)

`BuScopeProp`
: Scope Properties C-like analyzed while processing input

The design of scope rules and their description needs careful consideration if the concept of scopes being properties is involved. We have to answer some questions on the described language before we can decide which of the library modules is to be used:

It is easily decided that we need the facility of scope properties: Assume the language has named program objects, say modules, which consists of a range with definitions of components or members. Those members are accressible outside their defining range wherever the name of the module is accessible:

```
module m
```

```
  { int i;
    float f (); {...}
  }
m:f();
```

In this example the module body is a range where the members `i` and `f` are defined. The scope of the range contains bindings for `i` and `f`. It is a property of the module `m` which is set in the module definition. The construct `m:f` is a qualified name: A binding for `f` is to be found in the scope property of the qualifying module name `m`. The definitions valid in the context of the qualified name are irrelevant for the binding of `f`.

The same application pattern occurs for example with types that have components, like record types, structure types, and union types. There component selections are usually qualified with variables having such a type rather than with the type identifier itself.

It is recommended to use the `ScopeProp` module for the specification of such scope patterns. It fits to any of the basic scope rule modules, Alg-like, C-like, or bottom-up. It does *not* impose any ordering restriction that would require the definition of a member to occur before its qualified use. For example in a language with C-like basic scope rules the following sequence would be acceptable:

```
module m;
m:f();
module m
  { int i;
    float f (); {...}
  }
```

Even if it should be considered erroneous to use the qualified name `f` before its definition, it is recommended to specify the binding in the described way, and to enforce that restriction by a check of the related positions. The same holds for bottom-up basic scope rules. One only has to be aware that the binding of qualified names is determined *after* the bottom-up computations.

There are a few specific reasons where the modules `CScopeProp` or `BuScopeProp`, the C-like variants of `ScopeProp` are to be used instead:

If the basic scope rules are specified C-like using `BuScope` and the binding of qualified names has to be done by bottom-up computations, then `BuScopeProp` is to be used.

If the basic scope rules are specified C-like using `CScope` and the `CInh` module is used to implement the concept of inheritance, then `CScopeProp` is to be used. That is always necessary when bindings of scope properties are needed to solve the binding of non-qualified names in ranges where C-like scope rules apply. As a consequence it is enforced that the definitions of such members precede their uses.

The general description of this set of module is given in the section see Section 5.1 [ScopeProp], page 26, the deviations of its variants are described in see Section 5.2 [AlgScopeProp], page 28, see Section 5.3 [CScopeProp], page 28, and see Section 5.4 [BuScopeProp], page 29.

5.1 Scope Properties without Ordering Restrictions

This module `ScopeProp` implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that all scope properties are associated before any is accessed. This strategy fits to Algol-like and to C-like scope rules.

The module is instantiated by

```
$/Name/ScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

It is required that a basic scope rule module is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

Each of the modules introduces a `PDL` property named `NAMEScope` where `NAME` is the value of the `instance` parameter.

The module provide `.lido` specifications for the following computational roles:

`NAMERangeScopeProp` is to be inherited by a symbol representing a range, for example a module body. Its scope is associated as the `NAMEScope` property to the key `THIS.KEYScopeKey`. An upper or lower computation of `THIS.KEYScopeKey` has to be provided, for example the key representing the module. All local definitions are bound in this scope.

`NAMEIdUseScopeProp` is inherited by an applied ocurrence of a qualified identifier. It is bound in a scope that is obtained as a `NAMEScope` property from the attribute `INH.NAMEScopeKey`. A computation of `INH.NAMEScopeKey` has to be provided in the upper context. The obtained scope is available in the attribute `INH.NAMEScope`, e.g. to support a check whether the qualification is correct. For compatibility with older versions user computations may access the scope property and set the attribute `INH.NAMEScope`. In that case a suitable dependency has to be established (see below).

The attribute `NAMERootScope.NAMEGotScopeProp` represents the condition that the scope properties are associated thoughout the program. It may be used as a precondition for computations that access the `NAMEScope` property.

We demonstrate the use of these facilities by extending the language of our running example by module declarations and access of module components. The notation is specified by the following two concrete productions:

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

The symbols inherit the roles provided by the scope property module as described above:

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

SYMBOL ModBlock INHERITS RangeScopeProp END;
```

In the context of the module declaration it is specified that the scope of the module body is associated to the key of the module identifer.

In the context of the selection the scope is specified in which the selected component is to be bound. It is accessed from the key of the module identifer. The dependency ensures that all scope properties are associated before accessed here:

```
RULE: Expression  ::= ModUseIdent '::' QualIdent COMPUTE
      QualIdent.ScopeKey = ModUseIdent.Key;
END;

SYMBOL ModUseIdent INHERITS
       IdUseEnv, ChkIdUse, IdentOcc
END;

SYMBOL QualIdent   INHERITS
       IdUseScopeProp, ChkIdUse, IdentOcc
```

```
END;
```

In order to make sure that the it is really a module identifier to which the selection is applied we specify the following check

```
RULE: Expression  ::= ModUseIdent '::' QualIdent COMPUTE
  IF (AND (NE (ModUseIdent.Key, NoKey),
           EQ (QualIdent.Scope, NoEnv)),
  message (FATAL, CatStrInd ("module identifier required: ",
                             ModUseIdent.Sym),
           0, COORDREF));
END;
```

The message is only issued if the identifier is defined but does not have a scope property.

(The `Strings` module is used to compose the message text (see Section 'String Concatenation' in Solutions of common problems).)

5.2 Scope Properties Algol-like

This module `AlgScopeProp` is outdated; `ScopeProp` should be used instead. `AlgScopeProp` will be kept for a while to keep existing specifications valid.

5.3 Scope Properties C-like

This module implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that scope properties are associated and accessed in left-to-right depth-first order. It imposes the strong requirement that a qualified name, for example the `f` in `m.f`, may not preceed its definition.

It is recommended to use this module only if it is needed as a companion of the module `CInh`. Otherwise `ScopeProp` should be used (see Chapter 5 [Scope Properties], page 24).

The module is instantiated by

```
$/Name/CScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `CScope` is instantiated with the same values of the generic parameters.

The module provides a `PDL` property named `NAMEScope` and the computational roles `NAMERangeScopeProp` and `NAMEIdUseScopeProp`, as described in see Section 5.1 [ScopeProp], page 26.

In case of `NAMEIdUseScopeProp` a user computation has to provide the attribute `THIS.NAMEScope` (rather than `THIS.NAMEScopeKey` as described for see Section 5.1 [ScopeProp], page 26).

All computations of this module follow strictly C-like scope rules, i.e. binding of identifier occurrences, association of scope properties, and access of scope properties are done in left-to-right depth-first order.

Calls of `GetNAMEScope` in a user computation do not need a specific precondition if they depend on a key attribute of a context which is to the right of the context where the property is set. That is usually true for situations where the module role `NAMEIdUseScopeProp` is used. Only if a particular computation is to depend on the fact that all scope properties of the program are associated, it may depend on `INCLUDING NAMERootScope.NAMEGotScopeProp`.

5.4 Scope Properties C-like Bottom-Up

This module implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that scope properties are associated and accessed in left-to-right depth-first order. It imposes the strong requirement that a qualified name, for example the `f` in `m.f`, may not preceed its definition.

It is recommended to use this module only if qualified identifiers have to be bound in the bottom-up phase, or if the module is needed as a companion of the module `BuInh`. Otherwise `ScopeProp` should be used (see Chapter 5 [Scope Properties], page 24).

The computations provided by this module are executed while reading the input.

The module is instantiated by

```
$/Name/BuScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module BuScope is instantiated with the same values of the generic parameters.

The module provides a PDL property named NAMEScope and the computational roles NAMEIdSetScopeProp, NAMEIdGetScopeProp, and NAMEIdUseScopeProp. A role NAMERangeScopeProp is NOT provided; NAMERangeScope has to be used instead.

All computations of this module follow strictly C-like scope rules, i.e. binding of identifier occurrences, association of scope properties, and access of scope properties are done in left-to-right depth-first order.

As a consequence of bottom-up computation the value of a key can not be propagated by an upper computation to the range symbol. Hence, if the defining identifier occurrence precedes the range, the scope has to be created by the role NAMECreateNewScope (see Section 2.3 [BuScope], page 13) and associated to the key in the identifier context using the role NAMEIdSetScopeProp.

The role that opens the range scope (NAMEOpenNewScope, see Section 2.3 [BuScope], page 13) may also be associated to that identifier context, avoiding an additional symbol that derives to empty.

The range symbol has the role NAMERangeScope.

The module declaration of our example then reads:

```
RULE: Declaration ::= 'module' ModDefIdent Block ';' END;

SYMBOL ModDefIdent INHERITS
       CreateNewScope, OpenNewScope, IdSetScopeProp,
       IdDefScope, IdentOcc
COMPUTE
  SYNT.OpenPrecond = SYNT.Key;
END;
```

NAMEOpenPrecond is specified to depend on the key attribute to ensure that the identifier is bound in the enclosing environment before the environment of the module range is opened.

In component selections the scope property needs to be propagated from the context that provides it to the selector context. The module role NAMEGetScopeProp accesses the scope from the key specified by KEYScopeKey and assigns it to a variable. It is used at the selector context right of it by the role NAMEIdUseScopeProp.

Hence, in our running example the selection is specified as follows:

```
RULE: Expression  ::= ModUseIdent '::' QualIdent END;

SYMBOL ModUseIdent INHERITS
       GetScopeProp, IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = THIS.Key;
END;

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

If we had a typed record expression instead of the module identifier to select from, `ScopeKey` would be set to the type key instead of the module key.

6 Inheritance of Scopes

The basic scope rule concepts are described by hierarchically nested environments which reflect the structure of nested ranges in a program. Using scopes as properties of objects, as described in See Chapter 5 [Scope Properties], page 24 allows to bind single identifiers that occur outside of the range which forms the scope property, e.g. a component identifier that is qualified by a module name.

In this section we further extend that concept such that scope rules for language constructs like `with` statements of Pascal, `use` qualifications of Ada, or inheritance of classes as in object-oriented languages can be specified. All these constructs allow that non-qualified identifier occurrences may be bound to definitions contained in surrounding ranges or to definitions of scopes that are inherited by a surrounding range, for example

```
module m { int i; float f() {...} }
{ float g;
  with m
    { int i; g = f();}
}
```

The new concept is described by an inheritance relation between scopes that is used when applied identifier occurrences in a range are bound to definitions. In the above example the range of the `with`-statement inherits the scope of the module `m` and is embedded in the surrounding range.

Name analysis computations for such constructs rely on several different operations: scopes being created, bindings in a scope being established, scope properties being set, inheritance relations between scopes being established. The propagation of scope properties is not limited to strictly nested structures. Hence, the dependencies between the computations are rather sophisticated. That is why the combination of modules is restricted.

There are three modules that provide computations for the consistent renaming task based on inheritance. They rely on the use of the corresponding modules for basic scope rules and for scope properties:

`AlgInh` Inheritance with Algol-like Scope Rules

`CInh` Inheritance with C-like Scope Rules

`BuInh` Inheritance computed while processing input

Using one of these modules requires that the corresponding basic scope rule module and the scope property is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

Each of the three modules implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to the following inheritance rule:

An inheritance relation between scopes is introduced: A scope `c1` may inherit the bindings of a scope `c2`, i.e. a definition of `c2` is inherited by `c1` unless it is hidden by another definition of the same identifier in `c1`. A scope may inherit from several scopes (multiple inheritance). The inheritance relation is transitive and must be acyclic.

Together with the nesting of ranges the following general scope rule is applied:

An applied occurrence of an identifier `a` is bound to a definition of `a` which is contained in or inherited by the smallest enclosing range.

Definitions contained in a range hide definitions inherited (directly or indirectly) by that range.

Definitions inherited by a range hide definitions of enclosing ranges.

Using multiple inheritance a scope `c1` may inherit from a scope `c2` and from `c3`, where `c2` also inherits from `c3`. If both `c2` and `c3` define an identifier `a`, then the definition of `a` in `c3` is hidden by that of `c2`. This holds for `c1`, too, although there is an inheritance path from `c3` to `c1` that does not pass `c2`.

If several definitions of an identifier `a` are inherited via different unrelated inheritance paths, the applied occurrence is bound to an arbitrary one of them. This module provides a means to detect that situation, in order to issue an error message or to access all those definitions, depending on the requirements of the particular language.

The modules provide `.lido` specifications for the following computational roles:

`NAMERangeSnglInh` is a range that inherits a single scope. It is obtained from the NAMEScope property of the attribute `THIS.KEYInhKey`. An upper or lower computation for the attribute `THIS.KEYInhKey` has to be provided. The attribute `SYNT.NAMEInheritOk` is set to 1 iff the inheritance relation is legal, i.e. non cyclic and both scopes belong to the same environment hierarchy.

`NAMERangeMulInh` is a range that may inherit several scopes using the role `NAMEInheritScope` (described below) or the function `InheritClass` (provided by the environment module). An upper or lower computation for the `VOID` attribute `THIS.NAMEGotInh` has to be provided such that it is the postcondition stating that all those inheritances are done.

`NAMEInheritScope` requires that two attributes of type `Environment`, `THIS.NAMEOuterScope` and `THIS.NAMEInnerScope`. are computed by upper or lower computations. The inheritance relation from `THIS.NAMEOuterScope` to `THIS.NAMEInnerScope` is established, i.e. definitions of `THIS.NAMEOuterScope` are to be inherited by `THIS.NAMEInnerScope`.

The attribute `SYNT.NAMEInheritOk` is set to 1 iff the inheritance relation is legal, i.e. non cyclic and both scopes belong to the same environment hierarchy.

`NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` are roles to be associated to an applied identifier occurrence. They issue a message if several definitions of the identifer are inherited on different unrelated inheritance paths. `NAMEChkInhIdUse` may be used together with `NAMEIdUseEnv` or `NAMEIdUseScope`; `NAMEChkInhIdUseScopeProp` may be used together with `NAMEIdUseScopeProp`.

The modules provide also attributes `NAMEAnyScope.NAMEGotInhScopes` of type `VOID`. This is a precondition for any consistent renaming that may be affected by an inheritance. It has to be stated in any such user's computation not provided by this module.

We demonstrate the use of single inheritance by extending our running example by a `with` statement for modules (see Chapter 5 [Scope Properties], page 24).

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

The identifier should be bound to a module. The `WithBody` inherits the module's scope. I.e. the definitions of the module body are valid in the `WithBody`. They may be hidden by definitions in ranges contained in the `WithBody`. They may hide definitions in ranges enclosing the `with` statement. Hence, the `WithBody` plays the role of a `RangeSnglInh` and its scope is the target of the inheritance relation. The module's scope property is its source:

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

SYMBOL WithBody INHERITS RangeSnglInh END;
```

Note: In this example the `with`-expression can only be a simple identifier, `ModUseIdent`. In case of Algol-like scope rules special means have to be taken if the inherited scope property could be obtained from an arbitrary part of the tree using qualified names, see Section 6.1 [AlgInh], page 36.

Similarly we can extend the language of our running example by classes with multiple inheritance:

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

A declaration of a class resembles that of a module (see Chapter 5 [Scope Properties], page 24). Additionally other classes may be inherited by a class, i.e. their definitions are valid within the class body, if not hidden by innner definitions. The inherited definitions may hide definitions in ranges the class declaration is contained in. Hence, the scope of the class body is the target of all `Inheritances`, their sources are given by the scope property associated to the classes identified in the `Inheritances`.

```
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;

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

  IF (NOT (THIS.InheritOk),
  message (ERROR, CatStrInd ("wrong inheritance: ", THIS.Sym),
           0, COORDREF));
END;
```

Note: In this example the inherited classes are determined by a simple identifier each, `InheritIdent`. In case of Algol-like scope rules that can not be generalized, because the dependency setting of the computations in our modules prohibit that the inherited scope property is obtained from an arbitrary part of the tree using qualified names, see Section 6.1 [AlgInh], page 36.

Languages (like C++) allow that different definitions of an identifier may be inherited on different inheritance paths to a range. But in that case such an identifier may not be applied in that range. This restriction is checked by the roles `ChkInhIdUse` and `ChkInhIdUseScopeProp`. They have to be associated to applied identifier symbols which are bound in the enclosing environment or in the scope obtained from a property, respectively:

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

The above specification also fits to the specification for identifiers that are qualified by a module name given in (see Chapter 5 [Scope Properties], page 24). If in a construct `c::x` `c` is a class, then `x` is bound to a component defined in `c` or in a class inherited by `c`. This is the concept of the scope operator in C++.

These examples are applied in the same way for Algol-like and for C-like scope rules. The differences for bottom-up computation are explained in the description of the `BuInh` module.

6.1 Inheritance with Algol-like Scope Rules

This module implements consistent renaming of identifiers according to inheritance relations based on Algol-like scope rules as described in See Chapter 6 [Inheritance of Scopes], page 32.

The module is instantiated by

```
$/Name/AlgInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `AlgScope` and `AlgScopeProp` are instantiated with the same values of the generic parameters.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Chapter 6 [Inheritance of Scopes], page 32.

However, there is an additional restriction for `NAMERangeSnglInh` and `NAMERangeMulInh`: The computation of the scopes which they inherit may not depend on `NAMERootScope.NAMEGotInhScopes`, in particular not on computations using `NAMEIdUseScopeProp` for qualified identifiers. They may be determined by identification of non-qualified identifiers, as shown in the example in See Chapter 6 [Inheritance of Scopes], page 32.

An additional role `NAMERangeQualInhScope` is provided for cases where the computation of the scopes which are inherited may depend on `NAMERootScope.NAMEGotInhScopes`, in particular on computations using `NAMEIdUseScopeProp` for qualified identifiers. However, the scope of this range may not be used as a property.

The attribute `NAMEAnyScope.NAMEGotInhScopes` states that all inheritance relations are established for this range and for the direct decendant ranges of all its ancestors. The attribute is used as precondition for the consistent renaming computations, `NAMEIdUseEnv`, `NAMEIdUseScope`.

For binding of qualified identifiers using the role `NAMEIdUseScopeProp` the attribute `NAMERootScope.NAMEGotInhScopes` is stated as precondition, because named scopes of arbitrary parts of the tree may be involved.

The symbol roles `NAMEIdUseEnv` and `NAMEIdUseScope` may be used to bind names in constructs which establish inheritance relations for `NAMERangeSnglInh` or `NAMERangeMulInh`. Additionally `NAMEIdUseScopeProp` may be used to bind names in constructs which establish inheritance relations for `NAMERangeQualInhScope`.

The attribute `NAMEAnyScope.NAMEGotInhScopes` should be used in the same way as precondition of user computations, if they may depend on visible inheritances, and may contribute to determination of an inheritance relation of a scope that may be used as a scope property.

The attribute `NAMEAnyScope.NAMEGotInhNest` states that additionally to `NAMEGotInhScopes` the inheritances are established for all ranges that are immediate descendants of this range.

The attribute `NAMERootScope.NAMEGotAllInh` states that all inheritance relations are established in the whole tree. It can be used as a precondition for computations which may depend on arbitrary inheritances, but which do NOT contribute to determination of any inheritance relation.

6.2 Inheritance with C-like Scope Rules

This module implements consistent renaming of identifiers according to inheritance relations based on C-like scope rules as described in See Chapter 6 [Inheritance of Scopes], page 32.

The module is instantiated by

```
$/Name/CInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `CScope` and `CScopeProp` are instantiated with the same values of the generic parameters.

The use of this module enforces the requirement that for any kind of identifier occurrence strictly hold that the definition precedes its uses.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Chapter 6 [Inheritance of Scopes], page 32.

The attribute `NAMERootScope.NAMEGotInhScopes` states that all inheritance relations are established for the whole tree.

6.3 C-like Inheritance Bottom-Up

This module implements consistent renaming of identifiers according to inheritance relations based on C-like scope rules. The computations can be executed while input is read.

The module is instantiated by

```
$/Name/BuInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `BuScope` and `BuScopeProp` are instantiated with the same values of the generic parameters.

The use of this module enforces the requirement that for any kind of identifier occurrence strictly hold that the definition precedes its uses.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Chapter 6 [Inheritance of Scopes], page 32. No additional range role is provided by this module. The role `NAMERangeScope` of the basic scope rule module is to be used for ranges that are affected by inheritance, too.

The role `NAMEInheritScope` differs from the description in See Chapter 6 [Inheritance of Scopes], page 32:

The target scope for the inheritance relation is assumed to be computed by the role `NAMECreateNewScope` in this context or in a preceding context. It is passed via a variable. If `NAMECreateNewScope` and `NAMEInheritScope` are used in the same context, a computation `SYNT.NAMEInhPrecond = THIS.NAMENewScope;` has to be added, in order to guarantee proper use of the variable.

A lower computation of `SYNT.NAMEOuterScope` is required for this context.

The examples given in See Chapter 6 [Inheritance of Scopes], page 32 are modified here to allow for bottom-up computation using this module.

We demonstrate the use of single inheritance by extending our running example by a `with` statement for modules (see Chapter 5 [Scope Properties], page 24).

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

The identifier should be bound to a module. The `WithBody` inherits the module's scope. I.e. the definitions of the module body are valid in the `WithBody`. They may be hidden by definitions in ranges contained in the `WithBody`. They may hide definitions in ranges enclosing the `with` statement. `WithBody` plays the role of a `RangeScope`. In the preceding `WithUseIdent` context the scope is created and determined to be target of an inheritance relation. The scope property of the module key is stated to be the outer scope of the inheritance relation.

```
RULE: Statement ::= 'with' WithUseIdent 'do' WithBody END;

SYMBOL WithBody INHERITS RangeScope END;
SYMBOL WithUseIdent INHERITS
       GetScopeProp, CreateNewScope, InheritScope,
       OpenNewScope, IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = SYNT.Key;
  SYNT.OuterScope = SYNT.ScopeProp;
  SYNT.OpenPrecond = SYNT.Key;
```

```
END;
```

Similarly we can extend the language of our running example by classes with multiple inheritance:

The scope of the class body is created in the context `ClassDefIdent`, associated a property of the class identifier, and used as a target for the inheritance relations established in all `Inheritances`. The roles `RecentNewScope` and `OpenNewScope` in the newly introduced context `BuClass` access and open that scope.

```
RULE: Declaration ::= 'class' ClassDefIdent Inheritances
                              BuClass ClassBlock ';'
END;

SYMBOL ClassDefIdent INHERITS
       CreateNewScope, IdSetScopeProp, IdDefScope, IdentOcc
END;

SYMBOL BuClass INHERITS RecentNewScope, OpenNewScope END;
```

In the `InheritIdent` contexts the scope property of the identifer is accessed and determined to be the outer scope to be inherited to the prevoiusly created scope.

```
SYMBOL InheritIdent INHERITS
       GetScopeProp, InheritScope,
       IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = SYNT.Key;
  SYNT.OuterScope = SYNT.ScopeProp;

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

The above specification also fits to the specification for identifiers that are qualified by a module name given in (see Chapter 5 [Scope Properties], page 24). If in a construct `c::x` `c` is a class, then `x` is bound to a component defined in `c` or in a class inherited by `c`. This is the concept of the scope operator in C++.

7 Name Analysis Test

This module augments the specified processor such that it produces output that makes the results of name analysis visible. For each identifier occurrence that has one of the identifier roles of the name analysis modules a line of the form

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

is written to the standard output file. The first line number is that of the identifier occurrence, the second states where its binding was established by a defining occurrence, and the third where the scope of the binding has been created, i.e. usually the begin of the range. For unbound identifier occurrences a line of the form

```
m unbound in line 35
```

is written. The output is produced in left to right order of the identifier occurrence, independent of the order in which the bindings are found. The computations for producing that output are scheduled after the bindings are computed at all identifier occurrences, in order to avoid problems of evaluation order scheduling.

The output of the processors specified in `$/Name/Examples` is produced by using this module.

To achieve the effect of this module it is simply instantiated. No inheritance of any roles is necessary.

The module is instantiated by

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

The instance parameter must have the same value as that of the instantiation of the basic name analysis module, i.e. `AlgScope`, `CScope`, or `BuScope`. Several instances may be used for testing the bindings in different name spaces. Unfortunately, this module is NOT usable if the name analysis module is instantiated with a referto parameter that modifies the key attribute name.

The module makes use of the facility to associate a `DefTableKey` to scopes: for each `NAMERangeScope` a new key, and for each `NAMERangeScopeProp` its `ScopeKey`. `NAMERootScope.NAMEGotEnvKey` indicates that all those keys are associated. If that facility is also used independent of the `ShowBinding` module, the computations of `NAMERangeScope.NAMEGotEnvKey`

and
`NAMERangeScopeProp.NAMEGotPropEnvKey` have to be overridden to avoid interference with the intended computations.

The module associates a property named `NAMELine` to each identifier key, and to each key of a scope. Its value is the line number where the binding is established. An instance of the `NAMESetFirst` module is used for that purpose. Line number 0 is shown for definitions of identifiers and for scopes if they are not established by roles of name analysis modules. That holds in particular for predefined identifiers and for the root environment.

8 Environment Module

This module implements a standard contour model for name analysis. The data structure is a tree of *scopes*, each of which can contain an arbitrary number of definitions. A definition is a binding of an identifier to an object in the definition table (see PDL Reference Manual). For an identifier `idn` and a scope `sc` there is at most one binding in `sc`.

The environment module provides operations for building scope trees, adding definitions to specific scopes, and searching individual scopes or sequences of scopes for the binding of a particular identifier. Inheritance relations can be established between scopes to support object-oriented name analysis.

The module is capable of building multiple trees of scopes in order to model distinct name spaces, such that bindings in one tree do not effect the lookup in another tree.

The module places no constraints on the sequence of construction, definition and lookup operations; there is one exception: an inheritance relation may not be established for a scope that has already been involved in a lookup operation.

The module implements certain lookup operations such that linear search through several scopes is avoided in order to reduce the amortized asymptotic cost of name analysis. This effect on efficiency can be lost if the sequence of those lookup operations arbitrarily often switches the scopes they are applied to.

The modules described in the Name Analysis Library (see Section 'Name Analysis' in Specification Module Library: Name Analysis) provide solutions for common name analysis tasks based on this environment module. If they are used the interface of this module is available for use in `.lido` specifications; otherwise the interface is made available by adding

`$/Name/envmod.specs` to the processor specification. In C modules the interface of the environment module is introduced by `#include "envmod.h"`.

The environment module exports the following types, values, functions and macros. Exported types and their null values:

`Environment`

A value of type `Environment` is a pointer to a node in the tree of scopes. It is used either to refer to a single scope, or to refer to this scope and all the scopes that are visible from it, i.e. its ancestors in the tree and the scopes that are inherited by them.

`NoEnv` Is a value that represents no environment. Operations applied to `NoEnv` have no effect.

`Binding` A value of type `Binding` is a pointer to a triple `(int idn, Environment sc, DefTableKey key)` that represents a binding of of an identifier `idn` in a scope `sc` to an object `key`. The bindings of a scope form a list that can be traversed using macors described below.

`NoBinding`
: is a value that represents no binding.

`Scope, NoScope`
: are synonyms for `Binding` and `NoBinding`

`InheritPtr`
: is an opaque type that is used to traverse inheritance relations.

`NoInherit`
: is a value that indicates the end of inheritance traversal.

The following functions are provided for constructing the tree of scopes:

`Environment NewEnv ()`
: creates a new tree consisting of a single, empty scope and returns a reference to that empty scope.

`Environment NewScope (Environment env)`
: creates a new empty scope as a child of the scope referenced by its argument and returns a reference to that empty scope.

The following functions are provided to establish and to check inheritance relations between scopes:

`int InheritClass (Environment tocl, Environment fromcl)`
: establishes an inheritance relation from the scope `fromcl` to the scope `tocl` is established, if `tocl` and `fromcl` are different scopes in the same tree of scopes, and if the graph of inheritance relations remains acyclic when adding the relation, and if the scope `tocl` has not yet been involved in a lookup for a binding. `InheritClass` returns 1 if the inheritance relation could be established; otherwise it returns 0.

`int Inheritsfrom (Environment tocl, Environment fromcl)`
: returns 1 if `tocl` and `fromcl` are the same scopes, or if there is a direct or indirect inheritance relation from the scope `fromcl` to the scope `tocl`; otherwise it returns 0.

After a call of `Inheritsfrom` no further inheritance relation can be established for `tocl` or `fromcl`.

The following functions are provided to establish a binding within a scope. They differ in the following aspects:

The key for the new binding is given as an argument (`BindKey`, `AddIdn`, `BindKeyInScope`), or a new key is created for a new binding (`BindIdn`, `DefineIdn`, `BindInScope`).

Functions that should be preferred for efficiency reasons if several operations on one scope occur in a row (`BindKey`, `AddIdn`, `BindIdn`, `DefineIdn`), or if scopes are arbtrarily switched (`BindKeyInScope`, `BindInScope`).

`DefineIdn` and `AddIdn` are macros for calls of `BindIdn` and `BindKey` respectively. They are provided for compatibility with previous versions of the environment module.

`Binding BindKey (Environment env, int idn, DefTableKey key)`

: checks the scope referenced by its `env` argument for a binding of the identifier specified by its `idn` argument. If no such binding is found, a binding of the identifier `idn` to the definition table object specified by `key` is added to scope `env`. `BindKey` returns the value `NoBinding` if a binding already exists, and returns the new binding otherwise.

`int AddIdn (Environment env, int idn, DefTableKey key)`

: is a macro that calls `BindKey`. `AddIdn` returns the value `0` if a definition for `idn` already exists, and returns `1` otherwise.

`Binding BindKeyInScope (Environment env, int idn, DefTableKey key)`

: has the same effect as `BindKey`. `BindKeyInScope` should be used for efficiency reasons if bindings are established in several different scopes before lookups are performed in them.

`Binding BindIdn (Environment env, int idn)`

: behaves exactly like `BindKey`, except that if the referenced scope contains no binding of `idn` then `BindIdn` obtains a value from `NewKey()` and binds `idn` to that value in scope `env`. In addition, `BindIdn` always returns the the binding associated with `idn`.

`DefTableKey DefineIdn (Environment env, int idn)`

: is a macro that calls `BindIdn`. Instead of a binding it returns its key.

`Binding BindInScope (Environment env, int idn)`

: has the same effect as `BindIdn`. `BindInScope` should be used for efficiency reasons if bindings are established in several different scopes before lookups are performed in them.

The following functions are provided to lookup bindings for given identifiers. They differ in the following aspects:

Only the scope given as argument and those scopes it inherits from are considered for the lookup (`BindingInScope`, `KeyInScope`), or the scope given as argument, its ancestors in the tree of scopes and those scopes they inherit from are considered for the lookup (`BindingInEnv`, `KeyInEnv`).

`KeyInScope` and `KeyInEnv` are macros for calls of `BindingInScope` and `BindingInEnv` respectively. They are provided for compatibility with previous versions of the environment module.

For ease of understanding the effects of the functions are described here as if the bindings of scopes were traversed in a linear search. In fact the implementation avoids such linear search where possible.

`Binding BindingInScope (Environment env, int idn)`

: checks the scope referenced by its `env` argument, for a binding of the identifier specified by its `idn` argument. Thereafter the scopes that are directly or indirectly inherited by `env` are searched, such that a scope `tocl` is considered before a scope `fromcl`, if `tocl` inherits from `fromcl`. The binding that is found first is returned; If no binding is found `NoBinding` is returned.

`DefTableKey KeyInScope (Environment env, int idn)`

: is a macro for a call of `BindingInScope` such that the key of the found binding is returned, or `NoKey` if no binding is found.

`Binding BindingInEnv (Environment env, int idn)`

: has the same effect as `BindingInScope` except that if no binding for `idn` is found for scope `env` then the search continues as if `BindingInScope` was applied to successive ancestors of `env`. The binding that is found first is returned; If no binding is found `NoBinding` is returned.

`DefTableKey KeyInEnv (Environment env, int idn)`

: is a macro for a call of `BindingInEnv` such that the key of the found binding is returned, or `NoKey` if no binding is found.

The following functions allow to find further bindings that are related in some way to a given one:

`Binding OverridesBinding (Binding bind)`

: yields a hidden binding. Let `bind` be a binding of identifier `idn` in a scope `e`. Then the binding for `idn` is returned which the operation `BindingInEnv` would have found next, if any; otherwise `NoBinding` is returned.

`Binding NextInhBinding (Environment env, Binding bind)`

yields a binding that is also reachable and not hidden due to multiple inheritance relations. Let `bind` be a binding of identifier `idn` in a scope `e` that has been obtained by a call `BindingInScope (env, idn)` by `BindingInEnv (env, idn)`, or by `NextInhBinding (env, idn)`, and let `tocl` be `env` or its next ancestor that inherits from `e`. Then `NextInhBinding` returns a binding of identifier `idn`, if any, in a scope `ep` that is inherited by `tocl` but not by `e`; otherwise `NoBinding` is returned.

`DefTableKey NextInhKey (Environment env, int idn, DefTableKey key)`

has the same effect as `NextInhBinding`, except that the object keys of bindings are supplied and returned instead of bindings.

The following functions are provided to obtain information on environments:

`Binding DefinitionsOf(Environment env)`

returns the first binding of the scope `env`, if any; otherwise `NoBinding` is returned.

`int IdnOf(Binding b)`

returns the identifier of the binding triple.

`DefTableKey KeyOf(Binding b)`

returns the key of the binding triple.

`Environment EnvOf(Binding b)`

returns the environment of the binding triple.

`Binding NextDefinition(Binding b)`

returns the next binding of the scope `env`, if any; otherwise `NoBinding` is returned.

`Environment ParentOf(Environment env)`

returns the parent of `env` int the tree of scopes, if any; otherwise `NoEnv` is returned.

`DefTableKey SetKeyOfEnv(Environment env, DefTableKey k)`

associates a key `k` to the scope `env`, that may be used to relate properties to it. Initially `NoKey` is associated to `env`.

`DefTableKey KeyOfEnv(Environment env)`

returns the key associated to `env`.

`int IsClass(Environment env)`

returns 1 if the scope `env` has been argument of a call of `InheritClass`; otherwise 0 is returned.

`InheritPtr DirectInherits(Environment env)`

returns one direct inheritance relation to `env` established by a call of `InheritClass(env,fromcl)`, if any; otherwise `NoInherit` is returned.

`Environment EnvOfInherit(InheritPtr inh)`

returns the scope `fromcl` of the inheritance relation `inh`.

`InheritPtr NextInherit(InheritPtr inh)`

returns the next direct inheritance relation, if any; otherwise `NoInherit` is returned.

Index

F

G

I

J

K

L

M

N

O

P

R

S

T