Name Analysis Reference Manual

Uwe Kastens

University of Paderborn
D-33098 Paderborn
Germany

W. M. Waite

University of Colorado
Boulder, CO 80309-0425
USA

Table of Contents

1 Fundamentals of Name Analysis 3

1.1 Scope graphs 4
1.2 The generic lookup 4
1.3 Interaction with type analysis 5

2 Tree Grammar Preconditions 9

2.1 Representation of identifiers 9
2.2 Representation of ranges 9

3 Establishing the Structure of a Scope Graph .. 13

3.1 The RootScope role 13
3.2 The RangeScope role 14
3.3 The AnyScope role 16
3.4 Path edge creation roles 16
3.5 The OutSideInDeps role 17

4 Defining Occurrences 19

5 Applied Occurrences 21

5.1 Worklist search 22
5.2 Graph-complete search 24
5.3 Report an unsuccessful search 25

6 Isomorphic Scope Graphs 27

7 Implementing Language-Specific Behavior ... 29

7.1 Information access 29
7.2 Obtain a range from a qualifier 30
7.3 Is the binding acceptable? 31
7.4 Deciding among possible bindings 32
7.5 Initialization and finalization 33
7.6 Useful graph operations 34

8 Pre-defined Identifiers 39

9 Name Analysis Testing 41

10 Application Program Interface 43
10.1 The state of an instantiation 43
10.2 Functions that create nodes, edges, and bindings 45
10.3 Functions that initiate worklist operations 48
10.4 Functions that search complete graphs 49
10.5 Functions that pre-define symbols 51

Index 53

This library contains a module and a set of roles that can be used to implement the name analysis task for both simple languages with nested scopes and complex languages that may involve inheritance, context-dependent access rules, and arbitrary developer-defined relations.

The ScopeGraphs specification module is instantiated in a `specs` file with or without an instance argument:

```
$/Name/ScopeGraphs.gnrc+instance=NLO_:inst
$/Name/ScopeGraphs.gnrc:inst
```

If the instantiation has an `instance` argument, the value of that argument is prefixed to every name exported by that instantiation. For example, one of the names exported by the first instantiation above would be `NLO_RootScope`. The second instantiation would export the name `RootScope` because that instantiation has no `instance` argument. In this document we prefix each exported names with an asterisk (e.g. `*RootScope`) to indicate that the value of the `instance` argument will be added as a prefix added to that name. For an explanation of why multiple instantiations might be necessary, see Section 1.1 [Scope graphs], page 4.

We begin by describing the model underlying the ScopeGraphs module (see Chapter 1 [Fundamentals of Name Analysis], page 3), and sketch properties of an abstract syntax tree that facilitate use of the module's computational roles (see Chapter 2 [Tree Grammar Preconditions], page 9). The module provides roles to support constructing a scope graph (see Chapter 3 [Establishing the Structure of a Scope Graph], page 13), creating bindings for defining occurrences (see Chapter 4 [Defining Occurrences], page 19), and searching for bindings given an applied occurrence (see Chapter 5 [Applied Occurrences], page 21). There is an efficient representation for a set of isomorphic scope graphs (see Chapter 6 [Isomorphic Scope Graphs], page 27), and a developer can provide analysis functions tailored to a specific language (see Chapter 7 [Implementing Language-Specific Behavior], page 29).

Name analysis computations are interdependent, and depend on other computations such as type analysis. In an attribute grammar specification, it is often necessary to make this dependence explicit to ensure that computations will be correctly ordered (see Section "Dependent Computations" in *LIDO – Computations in Trees*).

Two modules related to the ScopeGraphs module can be used to create predefined identifiers (see Chapter 8 [Pre-defined Identifiers], page 39), and generate a test system (see Chapter 9 [Name Analysis Testing], page 41).

Finally, we describe the functions that are used to support the computational roles and can be invoked directly by the developer to tailor the process to unusual languages (see Chapter 10 [Application Program Interface], page 43).

1 Fundamentals of Name Analysis

Identifiers are basic symbols of a language. Each occurrence of an identifier in a program corresponds to a leaf of the abstract syntax tree of that program. A programmer uses a freely-chosen identifier to represent some entity in the program's universe. The task of name analysis is to discover the entity represented by each identifier occurrence. Four concepts relating identifiers to the entities they represent are necessary to describe this task:

- A *binding* is a pair consisting of an identifier '`i`' and an entity '`k`'.
- A *scope* is a contiguous sequence of program text associated with a set of bindings. We say that a binding in the set *has* the scope.
- A *defining occurrence* is an occurrence of an identifier '`i`' that could legally be the only occurrence of that identifier in the program. A binding '`(i,k)`' is associated with each defining occurrence.

  Language rules specify the scope of a defining occurrence, and thus a scope of the binding associated with that defining occurrence. Other language rules may specify further scopes of a binding that are not in the context of the defining occurrence.
- An *applied occurrence* is an occurrence of an identifier '`i`' that is legal only if it *identifies* a binding '`(i,k)`' in some set associated with its context.

  Language rules specify the set(s) in which an applied occurrence may identify bindings.

Defining occurrences, applied occurrences, and scopes are concepts related to the program text; how are these concepts represented in the tree? Each identifier occurrence is a single basic symbol in the text, and is represented by a leaf of the tree. A scope, on the other hand, extends over some region of the text. The developer needs to map each scope to an abstract syntax subtree encompassing that scope. We call such a subtree a *range*; several scopes may map into the same range.

Because a defining occurrence may be the only occurrence of that identifier, the entity and range of the binding must be determined from the syntactic context of the corresponding abstract syntax tree leaf. The way in which this is done is language dependent, but involves no name analysis. Bindings for defining occurrences can therefore be regarded as constants for the purposes of name analysis.

Name analysis is a computation carried out on the tree that is the internal representation of the program being analyzed; its goal is to determine the appropriate binding for each applied occurrence, based on the concepts and language rules discussed above. That computation is specified in terms of the abstract syntax, and takes the form of assignments of values to attributes of tree nodes (see Section "Computations" in *LIDO - Reference Manual*).

The search for a binding for an applied occurrence begins in the range containing that applied occurrence. If the identifier is not bound in that range, then most languages require the search to continue in related ranges. Ranges may be related syntactically, but many languages also allow the programmer to specify arbitrary relationships as part of their program. Such relationships cannot reasonably be described in terms of the abstract syntax tree. Thus the name analyzer builds a separate data structure, a directed graph called a *scope graph*, to encode the relationships among ranges.

The ScopeGraphs specification module provides a generic lookup algorithm to implement the search for a binding in a scope graph. A language definition may specify context conditions to determine whether or not a binding found by the generic algorithm is appropriate. If the binding is not appropriate, language rules may require either that the search be continued (possibly with some restriction) or abandoned. Such rules can be enforced by developer-coded functions that are invoked by the generic algorithm whenever a binding for the applied occurrence is found (see Chapter 7 [Language], page 29).

The task of type analysis is to establish the type associated with every program construct (parameters, variables, expressions, etc.) that has a type (see *Type Analysis Reference Manual*). Name analysis of a program written in a simple language can often be completed before beginning the type analysis, but this is not possible in general. Rather than considering name and type analyzers as monolithic processes, we need to think of them as collections of computations to be applied in particular linguistic contexts. Each computation requires certain inputs and delivers certain outputs; it can be applied whenever the necessary inputs are available.

1.1 Scope graphs

Each node of a scope graph corresponds to a range, and each edge defines a relationship between ranges. We say that node n is the *tail* of edge (n, n') and node n' is its *tip*.

There are two kinds of edges, *parent* edges and *path* edges. A parent edge defines a textual containment relationship that is established by the syntactic structure of the program, while a path edge defines a relationship established by the semantics of some program construct.

We label node n with a partial function $Bind(n) : (identifier \mapsto entity)$ specifying the bindings associated with the range corresponding to node n, and we label each path edge of a scope graph with a positive integer further classifying that path edge. A node may have an arbitrary number of path edges incident upon it. A scope graph may be cyclic, provided each cycle contains at least one path edge.

Some languages allow a programmer to use the same identifier to represent entities of different kinds in the same region of the program. It may also be the case that the ranges associated with different kinds of identifiers are different. Because of these properties, the ScopeGraphs specification module requires the developer to use a distinct scope graph for each kind of identifier. When a particular kind of identifier has a unique set of ranges, as does the label identifier in Java, a unique instantiation of the ScopeGraphs specification module is required for that kind of identifier. Often, however, a number of different kinds of identifier share the same set of ranges. (For example, field identifiers and method identifiers in Java share the same set of ranges.) In this case, the scope graphs for the different kinds of identifier are isomorphic and a single instantiation of the ScopeGraphs specification module can be used for all (see Chapter 6 [Isomorphic Scope Graphs], page 27).

1.2 The generic lookup

The generic lookup algorithm implements the search for an applied occurrence's binding. An applied occurrence '`i`' can appear on its own as a simple name, or it can be a component of a qualified name like '`q.i`'. (Here '`q`' is the *qualifier*, and may itself be a qualified name. The leftmost applied occurrence in '`q`' is considered to be a simple name.) If a binding is

being sought for a simple name, the search begins at the scope graph node for the smallest range containing that applied occurrence; otherwise the search begins at the scope graph node for the entity named by the qualifier.

Suppose that the search begins at node n of a scope graph. The generic algorithm first checks $Bind(n)$. If $Bind(n)$ does not contain an appropriate binding, the algorithm explores all paths starting at node n and made up solely of edges labeled 1. At each node n_i on those paths, $Bind(n_i)$ is checked before continuing.

If the applied occurrence is qualified, and if no binding is found after exploring all paths with label 1, the algorithm reports that there is no binding for the applied occurrence. This behavior implements the normal semantics of a qualified name, if the path edges labeled 1 represent an inheritance relation: the binding must be in the entity named by the qualifier, or it must be in some entity from which the qualifier inherits.

The binding for a simple name might be inherited, but it might also be imported into the range for the scope graph node n or be found in an outer range n'. A developer would use integers larger than 1 to label path edges that express semantic relations like import, or that tune the generic lookup. In general, if no binding is found for a simple name after traversing all paths starting at node n and made up solely of edges labeled k, the process is repeated with paths starting at node n and made up solely of edges labeled $k + 1$. This search continues until either a binding is found or the value $k + 1$ is not an edge label. The developer must take that behavior into account when selecting edge labels to represent specific relationships.

If no binding is found starting at node n for a simple name, and if there is a parent edge $n \rightarrow n'$, then the algorithm re-starts at node n'. Note that the new search beginning at node n' is completely independent of the one that started at node n.

Whenever the generic lookup algorithm finds a node whose *Bind* function contains a binding for the applied occurrence, it invokes a specified function (see Chapter 7 [Language], page 29). The module contains default implementations of these functions, but the developer can easily override those defaults to provide language-specific behavior. (The default implementation simply accepts the binding found.)

If the largest path edge label, '`MaxLabel`', is greater than 1, then the developer must provide a file named `ScopeGraphs.h` as part of the specification. `ScopeGraphs.h` must contain the following declaration:

```
#define MaxKindsPathEdge 'MaxLabel'
```

The syntactic context of a defining occurrence must determine the scope graph in which it is bound, but the syntactic context of an applied occurrence might not. When a unique scope graph cannot be determined for an applied occurrence from its syntactic context, the generic algorithm conducts a search in each of the possible graphs. If there is more than one result from these searches, a language-dependent disambiguation process must be used (see Chapter 7 [Language], page 29).

1.3 Interaction with type analysis

Consider the construct '`v.f`', where '`v`' is a variable of a record type and '`f`' is a field of that record. Name analysis can find the key bound to '`v`', which will represent a variable. However, the field name '`f`' is bound in the range associated with the *type* of that variable,

not with the variable itself. In order to apply the generic lookup to '`f`' in '`v.f`', some computation of the type analyzer must establish the type associated with the variable '`v`'. Because the qualifier is a type, we call a construct like '`v.f`' a *type-qualified* name.

If an applied occurrence can be distinguished syntactically as a typed entity use, the type is guaranteed to be available (see Section "Accessing the type of an entity" in *Type Analysis Reference Manual*). This is not generally true for a type-qualified name, where the identifiers may denote either types (such as Java classes) or typed entities (such as Java fields). Here the value of a property of the key bound to the identifier must be accessed to obtain the type, and some dependence relation must guarantee that the property's value has been set. If Eli's Typing module is used, the appropriate relation can be established by:

```
CLASS SYMBOL TypedDefId INHERITS *SetTypeOfEntity END;
CLASS SYMBOL TypedUseId COMPUTE
  SYNT.TypeIsSet=INCLUDING OutSideInDeps.GotEntityTypes;
END;
```

Note that only one instantiation of the ScopeGraphs module can interact with Eli's Typing module.

In some languages, a type-qualified name may define the tip of a path edge. Examples of such situations are the Pascal `with` statement and the Java anonymous class. This situation is problematic because both name and type analysis computations are necessary to resolve the tip name, but the scope graph is not complete without the path edge.

A Pascal `with` statement or a Java anonymous class corresponds to a subgraph, W, of the scope graph G. The rules of the language are such that G will never contain any edge whose tip is in W and whose tail is in $G - W$. That means that name analysis in $G - W$ can never depend on any information from W.

Suppose that we partition the analysis of a program containing such constructs, completing both name and type analysis in $G - W$ before examining any applied occurrences in W. Since the type-qualified name defining the tip of the path edge in question lies in the text corresponding to W, but all of its component bindings are defined in $G - W$, all of the necessary information for creating the path edge will be available when analyzing W. Moreover, the missing path edge will be irrelevant for the analysis of $G - W$.

A different kind of interaction between name and type analysis occurs when the language permits a single function name to describe several distinct operations. In this case, a range may contain more than one defining occurrence for the function identifier, each of which denotes a different entity. We say that in such a situation the function identifier is *overloaded*.

Because $Bind(n)$ is a function, there is a single binding for an overloaded function identifier in a given range. The key to which the function identifier is bound represents an *indication* that is associated with the set of operations described by the defining occurrences (see Section "Selecting an operator at an expression node" in *Type Analysis Reference Manual*). The operations in the set are distinguished by the types of operands they require, information that must be derived from type analysis computations.

If none of the operations whose defining occurrences lie in a particular range satisfies the requirements of the actual operands, then the language may require that the search continue in the normal way (see Section 1.2 [The generic lookup], page 4). In this case, name analysis

computations advancing the lookup alternate with type analysis computations attempting to identify the indications found.

2 Tree Grammar Preconditions

The ScopeGraphs specification module provides computational roles to be inherited by nonterminal symbols of the abstract syntax tree. To make effective use of these roles, the abstract syntax tree must be designed with them in mind. Generally speaking, that means that the abstract syntax tree structure will differ somewhat from the structure of the parse tree. Eli offers tools that aid in mapping a parse tree to an appropriate abstract syntax tree (see Section "The Relationship Between Phrases and Tree Nodes" in *Syntactic Analysis*).

A developer can often use the parsing grammar provided by the language designer, and define simple mapping rules to obtain the desired abstract syntax tree (see Section "Syntax development hints" in *Syntactic Analysis*). Sometimes, however, a change in the parsing grammar is required. That can lead to difficulties, depending on the parsing algorithm (see Section "How to Resolve Parsing Conflicts" in *Syntactic Analysis*).

2.1 Representation of identifiers

An identifier is a basic symbol of the language being analyzed, and will correspond to a named terminal symbol of the concrete grammar describing the input text. (Typically the terminal symbol is named '`Identifier`' or '`Ident`'.) Named terminal symbols do not contribute to the trees specified by the tree grammar, but a value derived from the basic symbol may be used in computations associated with the rule of a production or with the symbol on the left-hand side of a production (see Section "Productions" in *LIDO – Reference Manual*). A unique integer value is typically derived from each identifier by the token processor `mkidn` (see Section "Available processors" in *Lexical Analysis*).

Identifiers usually play different roles in different syntactic contexts. We recommend that these distinctions *not* be made in the concrete syntax. The concrete syntax should use the same named terminal (e.g. `Ident`) to represent identifiers in every context. Syntactic contexts should be distinguished in the abstract syntax by LIDO `RULE`s.

Consider 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;
```

The new symbol names must, of course, be defined:

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

2.2 Representation of ranges

Recall that a range is a subtree of the abstract syntax tree that encompasses a scope. Most programming languages allow nested scopes, and allow the meaning of an identifier in an

inner scope to differ from the meaning in an outer scope. In that case, we say that the identifier's binding from the outer scope is not *visible* in the inner scope. The developer's goal for the abstract syntax tree should be that each range encompasses a scope containing all occurrences of identifiers defined in that scope but *not* containing any defining occurrence whose binding is visible outside of that scope.

For example, consider the following grammar fragment:

```
ProcedureDeclaration:
  'procedure' ProcedureHeading ProcedureBody /
  Type 'procedure' ProcedureHeading ProcedureBody .
ProcedureHeading:
  ProcedureIdentifier FormalParameterPart ';'
    ValuePart SpecificationPart .
ProcedureIdentifier: Identifier .
ProcedureBody: Statement .
```

A `ProcedureDeclaration` is nested inside a block (not shown) and the semantics of the language make the `ProcedureIdentifier` visible in the range containing that block. `FormalParameterPart`, `ValuePart`, `SpecificationPart`, and `Statement` all belong to a single range whose bindings are visible within the `ProcedureDeclaration`. An abstract syntax tree corresponding to this grammar does not meet our goal: `ProcedureDeclaration` is the only region containing all identifier occurrences defined in the procedure, but it also contains the defining occurrence `ProcedureIdentifier` that is visible outside of the region.

The solution is to add a nonterminal `ProcedureRange` to the grammar:

```
ProcedureDeclaration:
  'procedure' ProcedureIdentifier ProcedureRange /
  Type 'procedure' ProcedureIdentifier ProcedureRange .
ProcedureIdentifier: Identifier .
ProcedureRange:
  ProcedureHeading ProcedureBody .
ProcedureHeading:
  FormalParameterPart ';' ValuePart SpecificationPart .
ProcedureBody: Statement .
```

Here the `ProcedureRange` subtree contains all occurrences of identifiers defined in the procedure, and does *not* contain the defining occurrence `ProcedureIdentifier`.

The `ProcedureHeading` can be considered to be the *interface* to the procedure, and the `ProcedureBody` to be the *implementation* of that interface. Suppose that the language allows the interface and its implementation to be widely separated, instead of packaged into a single declaration:

```
ProcedureInterface:
  'procedure' ProcedureIdentifier ProcedureHeading /
  Type 'procedure' ProcedureIdentifier ProcedureHeading .
ProcedureImplementation:
  'implementation' ProcedureIdentifier ProcedureBody .
```

The `ProcedureInterface` and `ProcedureImplementation` are both nested within the same block (not shown) and the semantics of the language make the `ProcedureIdentifier` visible in the range containing that block.

The `ProcedureHeading` and `ProcedureBody` are now completely disjoint subtrees; there is no `ProcedureRange` to contain them. Together, they contain all occurrences of identifiers defined in the procedure and therefore together play the same role as the `ProcedureRange` subtree in the previous example.

We have defined a scope as a contiguous sequence of program text, and said that the developer needs to map each scope to an abstract syntax subtree encompassing that scope (see Chapter 1 [Fundamentals of Name Analysis], page 3). In order to deal with separate interface and implementation, it is convenient to generalize these concepts by allowing a scope to be a set of disjoint contiguous sequences of program text, and requiring the developer to map these regions into an abstract syntax tree subforest that encompasses those sequences. The subforest then has the desired properties of a range: it contains all of the occurrences in the scope, and does not contain any defining occurrence whose binding is visible outside.

3 Establishing the Structure of a Scope Graph

Each instantiation of the ScopeGraphs specification module implements a set of isomorphic scope graphs (see Section 1.1 [Scope graphs], page 4). Because the graphs are isomorphic, they can all be described by a single set of ranges and the relationships among those ranges. A range is represented by a a value of type `NodeTuplePtr`. `NodeTuplePtr` has a distinguished value, `NoNodeTuple`, that represents no range. By default, an instantiation of the module implements a singleton scope graph set; for the implementation of larger sets, see Chapter 6 [Isomorphic Scope Graphs], page 27. If different kinds of entities are bound in differently-structured ranges, then the binding and lookup of those kinds of identifiers must appear in a specific instantiation of the ScopeGraphs for each range structure.

The module provides three computational roles to establish ranges and relate them to the abstract syntax tree (see Section 2.2 [Representation of ranges], page 9). These roles also define any parent edges. A separate role is used to define path edges that do not depend on applied occurrences. (For path edges whose tips are defined by applied occurrences, see Section 5.1 [Worklist search], page 22.)

A role is also provided to partition a scope graph in order to deal with path edges whose tips are defined by type-qualified names (see Section 1.3 [Interaction with type analysis], page 5).

3.1 The RootScope role

The `*RootScope` role is automatically inherited by the root symbol of the grammar. It provides the following attributes:

`*Env` is a `NodeTuplePtr`-valued attribute representing a range with the bindings defined at the root of the grammar. This attribute is set by a module computation that must never be overridden by the developer.

`*ScopeKey`
: is a `DefTableKey`-valued attribute set by a module computation to the value `NoKey`. This computation must never be overridden by the developer.

`*GotEnv` is a `VOID` attribute representing the computation state in which:

- all scope graph nodes have been created
- all defining occurrences have been bound
- all computation states represented by `*IdDef.*GotDefKeyProp` attributes have been reached (see Chapter 4 [Defining Occurrences], page 19).

This attribute is set by a module computation that must never be overridden by the developer.

Whether or not identifiers are actually bound in `*RootScope.*Env` depends on the characteristics of the language and how the developer chooses to model program structure. For example, some languages provide built-in identifiers that are said to be "defined in a fictitious outer block". Such a situation is best handled by binding those identifiers in `*RootScope.*Env`.

See Section 10.1 [The state of an instantiation], page 43, for a discussion of computation state.

3.2 The RangeScope role

The `*RangeScope` role should be inherited by any symbol (other than the root of the grammar) that is the root of a subtree containing defining occurrences (see Section 2.2 [Representation of ranges], page 9). It provides the following attributes:

`*Env` is a `NodeTuplePtr`-valued attribute representing the range with the bindings defined in this range. This attribute is set by a module computation.

`*Parent` is a `NodeTuplePtr`-valued attribute representing the range of the scope graph node that should be the tip of this node's parent edge (see Chapter 1 [Fundamentals of Name Analysis], page 3). This inherited attribute is set by a module computation to the value of `INCLUDING *AnyScope.*Env`.

`*ScopeKey`
is a `DefTableKey`-valued attribute allowing the developer to associate a program entity with this range. This inherited attribute is set by a module computation to the value `NoKey`, indicating that the range has no associated entity.

If an overriding user computation sets the `*ScopeKey` attribute to '`key`', then we say that the program entity represented by '`key`' *owns* the range. A module computation then sets the following property of '`key`':

`OwnedNodeTuple`
is a `NodeTuplePtr`-valued property representing the value of the `*RangeScope.*Env` attribute.

If '`e`' is the value of a `*RangeScope.*Env` attribute, then `OwnerKeyOfNodeTuple('e')` yields the value '`key`'.

The default computation of `Parent` reflects the normal containment relation among ranges: nested ranges correspond to nested subtrees of the abstract syntax tree (see Section 2.2 [Representation of ranges], page 9). If no such relationship is defined by the language, the developer should override the default computation:

```
CLASS SYMBOL RangeScope COMPUTE INH.Parent=NoNodeTuple; END;
```

This example overrides the computation in *every* `RangeNode` context. The computation can also be overridden in a single context:

```
TREE SYMBOL record_type INHERITS RangeScope COMPUTE
  INH.Parent=NoNodeTuple;
END;
```

or

```
RULE: new_type ::= record_type COMPUTE
  record_type.Parent=NoNodeTuple;
END;
```

Ranges are often owned by program entities such as types or classes. Those entities are represented by definition table keys, and the ownership relation between the range and the entity is established by assigning the entity's definition table key to the range's `*ScopeKey` attribute. The default computation of `*ScopeKey` must be overridden in this case.

For example, consider a record type whose characteristics are established by the `TypeDenotation` role of the `Typing` module (see Section "Type denotations" in *Type*

Analysis Reference Manual). That type denotation owns the scope graph node in which field names are defined, and the ownership relation must be established:

```
TREE SYMBOL record_type INHERITS TypeDenotation, RangeScope COMPUTE
  INH.ScopeKey=THIS.Type;
END;
```

There is no need for the entity's definition table key to come from the same syntactic context as the range; the only requirement is that it be assigned as the value of the `*ScopeKey` attribute.

The `*ScopeKey` attribute is also the mechanism by which we tie together disjoint subtrees that form a single range: all of the subtrees in the range must have their `*ScopeKey` attributes set to the same value. Consider the example at the end of Section 2.2 [Representation of ranges], page 9. The appropriate computations are:

```
SYMBOL ProcedureIdentifier INHERITS IdDefScope END;
SYMBOL ProcedureHeading    INHERITS RangeScope END;
SYMBOL ProcedureBody       INHERITS RangeScope END;

RULE ProcedureInterface ::=
  'procedure' ProcedureIdentifier ProcedureHeading
COMPUTE
  ProcedureHeading.ScopeKey=ProcedureIdentifier.Key;
END;

RULE ProcedureInterface ::=
  Type 'procedure' ProcedureIdentifier ProcedureHeading
COMPUTE
  ProcedureHeading.ScopeKey=ProcedureIdentifier.Key;
END;

RULE ProcedureImplementation ::=
  'implementation' ProcedureIdentifier ProcedureBody
COMPUTE
  ProcedureBody.ScopeKey=ProcedureIdentifier.Key;
END;
```

Notice that the `ProcedureIdentifier` inherits the `IdDefScope` role (see Chapter 4 [Defining Occurrences], page 19). For a given procedure, the `Key` attribute will be the same for each `ProcedureIdentifier` occurrence, because the two occurrences lie in the same range (see Section 2.2 [Representation of ranges], page 9). Therefore the `ScopeKey` attributes of the interface and implementation will be identical, placing these two disjoint subtrees in the same range.

This example illustrates an important point: the value of the `*ScopeKey` attribute must be obtained from either a defining occurrence or from an invocation of `NewKey`. Never attempt to obtain a key from an applied occurrence; this would prevent the system from finding an attribute evaluation order.

3.3 The AnyScope role

The `*AnyScope` role is automatically inherited by any symbol inheriting either `*RootScope` or `*RangeScope`.

There are many situations in which a computation needs information about the range in which it is embedded. Any attribute of the enclosing range can be accessed via an `INCLUDING` construct (see Section "INCLUDING" in *LIDO – Reference Manual*). `AnyScope` can be used in `INCLUDING` constructs if `RootScope` and `RangeScope` need not be distinguished.

It is possible to define attributes and computations for `AnyScope`, but there is a small problem: `AnyScope` is inherited by the root of the grammar, which cannot have any inherited attributes. Therefore no attribute can be explicitly declared `INH` in `AnyScope`, and no `INH` assignment can be defined in `AnyScope` (see Section "Attributes" in *LIDO – Reference Manual*).

`Env` is an example of an attribute that is defined for `AnyScope`. It has class `SYNT` at the root of the grammar, and class `INH` at all grammar nodes inheriting `RangeScope`. The declaration of this attribute in `AnyScope` does not explicitly specify a class; the computation of its value in `RootScope` specifies `SYNT`, and the computation of its value in `RangeScope` specifies `INH`.

3.4 Path edge creation roles

When the tip of a path edge does not depend on an applied occurrence, the `*BoundEdge` role is appropriate. (Use `*WLCreateEdge` when the tip depends on an applied occurrence: see Chapter 5 [Worklist search], page 21.)

`*BoundEdge` should be inherited by some convenient symbol in a context where information about the tip and tail of the path edge can be obtained. It provides the following attributes:

`*tailEnv` is a `NodeTuplePtr`-valued attribute representing the range that is the tail of the edge. This attribute must be set by a developer computation.

`*toEnv` is a `NodeTuplePtr`-valued attribute representing the range that is the tip of the edge. This attribute must be set by a developer computation.

`*label` is an integer-valued attribute that contains the edge label (see Chapter 1 [Fundamentals of Name Analysis], page 3). This synthesized attribute is set by a module computation to 1.

`*EdgeKey` is a `DefTableKey`-valued attribute representing the key of the created edge. This attribute is set by a module computation.

A module computation sets the following properties of the value of the `EdgeKey` attribute:

`FromNodeTuple`
: is a `NodeTuplePtr`-valued property holding the value of the `*tailEnv` attribute.

`ToNodeTuple`
: is a `NodeTuplePtr`-valued property holding the value of the `*toEnv` attribute.

`EdgeLabel`
: is an integer-valued property holding the value of the `*label` attribute.

3.5 The OutSideInDeps role

The `OutSideInDeps` role is used to partition the abstract syntax tree into subtrees, in order to bind type-qualified names of path edge tips incrementally. It is automatically inherited by the root symbol, and can also be inherited by any symbol that inherits the `RangeScope` role. `OutSideInDeps` defines a subgraph W of the scope graph G. No edge of G may have its tail in $G - W$ and its tip in W (see Section 1.3 [Interaction with type analysis], page 5).

The `OutsideEdge` role can be used to create edges that have their tails at the `OutSideInDeps` node and their tips in $G - W$. It would be appropriate if several edges are to be created for one `OutSideInDeps` node.

`OutsideEdge` provides the following attributes:

`*tailEnv` is a `NodeTuplePtr`-valued attribute representing the range that is the tail of the edge. This attribute must be set by a developer computation.

`*toEnv` is a `NodeTuplePtr`-valued attribute representing the range that is the tip of the edge. This attribute must be set by a developer computation.

`*label` is an integer-valued attribute that contains the edge label (see Chapter 1 [Fundamentals of Name Analysis], page 3). This synthesized attribute is set by a module computation to 1.

`*EdgeKey` is a `DefTableKey`-valued attribute representing the key of the created edge. This attribute is set by a module computation.

A module computation sets the following properties of the value of the `EdgeKey` attribute:

`FromNodeTuple`
: is a `NodeTuplePtr`-valued property holding the value of the `*tailEnv` attribute.

`ToNodeTuple`
: is a `NodeTuplePtr`-valued property holding the value of the `*toEnv` attribute.

`EdgeLabel`
: is an integer-valued property holding the value of the `*label` attribute.

Both the Pascal `with` statement and the Java anonymous class illustrate the common case in which there is a single path edge whose tail is at the `OutSideInDeps` node and whose tip is in $G - W$. The `*OutSideInEdge` role can be used for this special case.

The `*OutSideInEdge` role must be inherited by a symbol that also inherits the `*OutSideInDeps` role. It uses the `*Env` attribute of the `*OutSideInDeps` role and the following two attributes to define the created edge:

`*tipEnv` is a `NodeTuplePtr`-valued attribute representing the tip of the created edge. This attribute must be set by a developer computation.

`*label` is an integer-valued attribute that contains the edge label. This synthesized attribute is set by a module computation to 1.

`*OutSideInEdge.*Env` specifies the tail of the new edge.

4 Defining Occurrences

Defining occurrences in a range are used to populate the $Bind(n)$ functions for the scope graph node(s) n corresponding to that range. In the default case there is only one such node; to deal with multiple nodes, see Chapter 6 [Isomorphic Scope Graphs], page 27. Only one computational role, `*IdDefScope`, is provided by the module for defining occurrences. (For a situation in which an identifier occurrence is both applied and defining, see Section 5.1 [Worklist search], page 22.)

The `*IdDefScope` role should be inherited by any defining occurrence (see Chapter 1 [Fundamentals of Name Analysis], page 3). It provides four attributes:

`Sym` is an integer-valued attribute specifying the identifier. This synthesized attribute is set by a module computation to the value derived from the identifier (see Chapter 2 [Tree Grammar Preconditions], page 9).

`*Env` is a `NodeTuplePtr`-valued attribute representing the range in which the identifier should be bound. This inherited attribute is set by a module computation to the value of `INCLUDING *AnyScope.*Env`.

`*Key` is a `DefTableKey`-valued attribute representing the entity to which the identifier has been bound. This attribute is set by a module computation.

A module computation sets the following properties of the value of the `*Key` attribute:

`Sym` is an integer-valued property holding the value of the `Sym` attribute.

`Coord` is a `CoordPtr`-valued property holding the source text coordinate of the definition (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*).

`Env` is a `NodeTuplePtr`-valued property holding the value of the `*Env` attribute.

`GraphIndex`
is an integer-valued property holding the kind derived from the context of the defining occurrence (see Chapter 6 [Isomorphic Scope Graphs], page 27).

Other `*Key` properties can be set by developer computations.

`*GotDefKeyProp`
is a void attribute representing the state that all properties of the `*Key` that are needed during a search have been set.

`*GotDefKeyProp` should be stated as the postcondition of any developer computations that set properties of `*Key` needed by the search process (see Section 7.1 [Information access], page 29). Accumulating computations must be used for this purpose (see Section "Accumulating Computations" in *LIDO – Reference Manual*).

5 Applied Occurrences

At every applied occurrence, name analysis computations must search for the defining occurrence that denotes the same entity. Recall that the module provides a generic search algorithm based on the structure of a scope graph (see Section 1.2 [The generic lookup], page 4). All of the computational roles implementing searches use the following four attributes to provide the information on which that search is based and to record the result:

`Sym` is an integer-valued attribute specifying the identifier. This synthesized attribute is set by a module computation to the value derived from the identifier (see Chapter 2 [Tree Grammar Preconditions], page 9).

`*UseKey` is a `DefTableKey`-valued attribute characterizing the applied occurrence. This synthesized attribute is set by a module computation.

A module computation sets the following properties of the value of the `*UseKey` attribute:

`Sym` is an integer-valued property holding the value of the `Sym` attribute.

`Coord` is a `CoordPtr`-valued property holding the source code location (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*).

Other `*UseKey` properties can be set by developer computations.

`*GotUseKeyProp`
is a void attribute representing the state that all properties of the `*UseKey` needed during a search have been set.

`*GotUseKeyProp` should be stated as the postcondition of any developer computations that set properties of `*UseKey` needed by the search process (see Section 7.1 [Information access], page 29). Accumulating computations must be used for this purpose (see Section "Accumulating Computations" in *LIDO – Reference Manual*).

`*Key` is a `DefTableKey`-valued attribute representing the key of the corresponding defining occurrence. This attribute is set by a module computation. `*Key` is a postcondition of the search.

`*Key` is set to the value `NoKey` if the computation has been unable to find a suitable defining occurrence. `*ChkIdUse` can be inherited by any applied occurrence to provide an error report when the value of `*Key` is `NoKey` (see Section 5.3 [Report an unsuccessful search], page 25).

For the properties of the value of the `*Key` attribute, see Chapter 4 [Defining Occurrences], page 19.

Suppose that the language allows a programmer to specify relationships between ranges by using applied occurrences:

```
1   interface A {
2     class D { ... }
3   }
4   class B extends C.D { ... }
```

```
5   class C implements A { ... }
```

The inheritance relationship specified on line 4 would be represented in the scope graph by a path edge with label 1 whose tail is the scope node owned by the class being declared and whose tip is the scope node found by looking up the applied occurrence `D`. That path edge can be added to the scope graph only after the search for `D` has been completed. But in order to complete the search for `D`, the path edge resulting from the declaration on line 5 must have been added to the scope graph before the search for `D` began. Thus a search for an edge tip name could fail in two ways: there might be no appropriate defining occurrence, or some necessary edge might not have been added to the scope graph before the search began. If the search failed because of missing edges, it must be repeated after edges have been added. Such repeated searches are costly.

Most of the applied occurrences in a program probably don't describe the tips of path edges. Although searches for those symbols *could* use the same approach as is used for edge tip names, there is no point in doing so. For performance reasons, it is best to defer searches that don't affect the set of edges until the scope graph is complete. Then if a search fails, we know that there is no appropriate defining occurrence; there is no need to repeat the search.

There are thus two distinct kinds of searches necessary to find the appropriate defining occurrence, depending upon the context of the applied occurrence:

Worklist search
: If a scope graph is incomplete at the time the computation is initiated, and an appropriate binding is not found, the lookup will be repeated when that scope graph has been augmented.

Graph-complete search
: If an appropriate binding is not found, the search fails.

5.1 Worklist search

`*WL` computational roles incorporate a worklist algorithm that iterates operations until all are complete. This algorithm is expensive, so if a language does not allow a programmer to specify edge tip names or to copy named bindings then the developer should not use `*WL` roles at all.

The items on the worklist are tasks, not values. Each of the `*WL` roles provides an attribute that identifies the corresponding task:

`*FPItem` is an `FPItemPtr`-valued attribute that represents a worklist computation. This attribute is set by a module computation.

There are two computational roles that embody worklist actions to search for bindings:

`*WLSimpleName`
: should be inherited by a simple identifier whose binding may be sought before the scope graph is complete.

`*WLQualName`
: should be inherited by a qualified identifier whose binding may be sought before the scope graph is complete. Searches for bindings of qualified identifiers do not explore parent edges of a scope graph.

The `*WLQualName` role provides an additional attribute:

`*DependsOn`
: is an `FPItemPtr`-valued attribute that represents the computation for the qualifier. This attribute must be set by a developer computation to the value of a `*WLSimpleName.*FPItem` attribute or a `*WLQualName.*FPItem` attribute.

There is one computational role that embodies worklist actions to add a path edge to a scope graph:

`*WLCreateEdge`
: should be inherited by some convenient symbol in a context where information about the tip and tail of a named path edge can be determined. It provides four additional attributes:

  `*tailEnv` is a `NodeTuplePtr`-valued attribute representing the range that is the tail of the created edge. This attribute must be set by a developer computation.

  `*tipFPItem`
  : is an `FPItemPtr`-valued attribute that represents the computation for the tip of the created edge. This attribute must be set by a developer computation to the value of a `*WLSimpleName.*FPItem` attribute or a `*WLQualName.*FPItem` attribute.

  `*label` is an integer-valued attribute that contains the edge label. This synthesized attribute is set by a module computation to 1.

  `*EdgeKey` is a `DefTableKey`-valued attribute representing the key of the created edge. This attribute is set by a module computation. `EdgeKey` is a postcondition for the worklist computation.

  A module computation sets the following properties of the value of the `EdgeKey` attribute:

  `FromNodeTuple`
  : is a `NodeTuplePtr`-valued property holding the value of the `*tailEnv` attribute.

  `ToNodeTuple`
  : is a `NodeTuplePtr`-valued property holding the value of the `*toEnv` attribute.

  `EdgeLabel`
  : is an integer-valued property holding the value of the `*label` attribute.

Some languages allow bindings from one range to be added to the *Bind* function of another range.

`*WLInsertDef`
: should be inherited by a defining occurrence that depends on the result of a search for the binding of an applied occurrence.

The *WLInsertDef role uses the following attributes to provide information and to record the result:

*DependsOn
: is an FPItemPtr-valued attribute that represents the computation yielding the existing binding. This attribute must be set by a developer computation to the value of a *WLSimpleName.*FPItem attribute or a *WLQualName.*FPItem attribute.

*Scope
: is a NodeTuplePtr-valued attribute representing the range into which the existing binding is to be inserted. This inherited attribute is set by a module computation to INCLUDING *AnyScope.*Env.

Here is an example of WLInsertDef usage. The goal is to import the binding of a type name defined elsewhere into the range of the enclosing compilation unit. There is a single identifier occurrence; its role as an applied occurrence is embodied in the symbol ITName, and its role as a defining occurrence is embodied in the symbol ImportedTypeName. The two symbols are connected by a chain rule:

```
TREE SYMBOL ITName           INHERITS WLSimpleName END;
TREE SYMBOL ImportedTypeName INHERITS WLInsertDef  END;

RULE: ITName ::= Identifier END;

RULE: ImportedTypeName ::= ITName COMPUTE
  ImportedTypeName.Sym=ITName.Sym;
  ImportedTypeName.DependsOn=ITName.FPItem;
  ImportedTypeName.Scope=INCLUDING CompilationUnit.TypeEnv;
END;
```

There are two ways that *WLInsertDef can fail: there may be no existing binding, or the range represented by the *Scope attribute may contain a local binding for the given identifier. In our example, the first case would result in ITName.Key having the value NoKey (see Chapter 5 [Applied Occurrences], page 21). In the second case, the value of ImportedTypeName.Key will differ from the the value of ITName.Key.

5.2 Graph-complete search

*GC computational roles assume that all work list tasks have been completed (see Section 5.1 [Worklist search], page 22).

*GCLocalName
: should be inherited by an identifier whose binding can be sought only after the scope graph is complete. Searches for bindings of local names do not explore any edges of a scope graph.

  The *GCLocalName role provides an additional attribute:

  *Scope
  : is a NodeTuplePtr-valued attribute representing the range in which the search for the identifier's binding should take place. This inherited attribute is set by a module computation to the value of INCLUDING *AnyScope.*Env.

`*GCSimpleName`
: should be inherited by a simple identifier whose binding can be sought only after the scope graph is complete.

  The `*GCSimpleName` role provides an additional attribute:

  `*Scope` is a `NodeTuplePtr`-valued attribute representing the range in which the search for the identifier's binding should begin. This inherited attribute is set by a module computation to the value of `INCLUDING *AnyScope.*Env`.

`*GCQualName`
: should be inherited by a qualified identifier whose binding can be sought only after the scope graph is complete. Searches for bindings of qualified identifiers do not explore parent edges of a scope graph.

  The `*GCQualName` role provides two additional attributes:

  `*ScopeKey`
  : is a `DefTableKey`-valued attribute representing the qualifier. This inherited attribute is set by a module computation to the value `NoKey`.

  `*Scope` is a `NodeTuplePtr`-valued attribute representing the qualifier. This inherited attribute is set by a module computation to the value `GetOwnedNodeTuple(THIS.ScopeKey,NoNodeTuple)` (see Section 3.2 [The RangeScope role], page 14).

`*GCQualName.*Scope` must contain the `NodeTuplePtr` value of the qualifier. There are two possibilities:

1. A developer computation sets `*GCQualName.*ScopeKey` to a `DefTableKey` value from which the desired value can be obtained.
2. A developer computation sets `*GCQualName.*Scope` to the desired value and `*GCQualName.*ScopeKey` is ignored.

The developer can change the default computation of `*GCQualName.*Scope` from `*GCQualName.*ScopeKey` by providing an extension function (see Section 7.2 [Obtain a range from a qualifier], page 30).

5.3 Report an unsuccessful search

If a search for a defining occurrence is unsuccessful, the `*Key` attribute of the applied occurrence is set to the value `NoKey`. The `*ChkIdUse` computational role is provided by the module to issue a report in that case.

The `*ChkIdUse` role may be inherited by any applied occurrence '`app`'. `*ChkIdUse` provides two additional attributes:

`*SymErr` is an integer-valued attribute. This synthesized attribute is set by the module computation `EQ(THIS.*Key, NoKey)`.

`*SymMsg` is a `VOID`-valued attribute. This synthesized attribute is set by the module computation

```
IF (THIS.*SymErr,
```

```
message(
  ERROR,
  CatStrInd("Identifier is not defined: ", THIS.Sym),
  0,
  COORDREF));
```

The behavior can be changed for all applied occurrences by symbol computations overriding the computation of `*SymError.*SymErr` and/or `*SymError.*SymMsg`. It can be changed for individual contexts by rule computations overriding the computation of '`app`'`.*SymErr` and/or '`app`'`.*SymMsg`.

6 Isomorphic Scope Graphs

Recall that when a language allows a programmer to use the same identifier to represent entities of different kinds in the same region of the program, the developer is required to use a distinct scope graph for each kind of identifier (see Section 1.1 [Scope graphs], page 4). Often a number of these graphs have the same ranges and the same relationships among those ranges. In that case the distinct scope graphs are isomorphic, and they can all be implemented by a single instantiation of the ScopeGraphs specification module.

A ScopeGraphs module instantiation *always* represents a set of isomorphic scope graphs; by default, that set has a single member. Ranges are represented by values of type `NodeTuplePtr` (see Chapter 3 [Establishing the Structure of a Scope Graph], page 13). These values are pointers to `NodeTuple` values, which are arrays of pointers to the actual scope nodes. A set of edges that are equivalent under the isomorphism is specified by giving `NodeTuplePtr` values for tail and tip; the module then creates the elements of the set. There is no additional mechanism needed for a larger set, one need only provide some additional information about particular identifier occurrences to specify the kind of identifier involved.

Let's consider a concrete example. A Java programmer may use the identifier `ambig` to represent a package, type, variable, and method in a single range. This requires four distinct scope graphs, one for each entity that `ambig` can represent. However, the language rules are such that these four scope graphs are isomorphic. Thus this set of scope graphs can be implemented by a single instantiation of the ScopeGraphs module. The `NodeTuple` corresponding to each range would be a four-element array in that implementation.

The number of elements in the set for a given instantiation of the ScopeGraphs module is given by the integer-valued `*RootScope.*NumberOfIsoGraphs` attribute. This synthesized attribute is set by a module computation to the value 1. A developer must override that computation when specifying several isomorphic scope graphs.

The `*RootScope` role is automatically inherited by the symbol at the root of the abstract syntax. The computation of the `*NumberOfIsoGraphs` attribute must be overridden by introducing a computation at that symbol. For example, if `Java` were the symbol at the root of the abstract syntax, the computation would be:

```
TREE SYMBOL Java COMPUTE SYNT.*NumberOfIsoGraphs=4; END;
```

The elements of the set of scope graphs implemented by a module instantiation are indexed by values i, $0 \leq i < *$`NumberOfIsoGraphs`. A specification is much more understandable if the possible graph indexes are named. Here is an appropriate set of graph index names for a Java specification:

```
enum { packageName, typeName, expressionName, methodName,
       packageOrTypeName, expressionOrTypeName, ambiguousName
     };
```

The first line names the kinds of entity that have scope graphs (an `expressionName` is a field name or a variable name).

Syntactic context must determine the specific kind of identifier at its defining occurrence. An applied occurrence at which the specific kind of identifier can be determined is called a *strong context*. In a *weak context*, the kind of identifier cannot be determined precisely although the set of possibilities could be narrowed. For example, in certain Java contexts an applied occurrence might refer to either a package or a type, but not to a method. Integer

values j, $j \geq$ `RootScope.NumberOfIsoGraphs` are used to specify the kind of identifier in a weak context. Meanings for these larger values are chosen by the developer, as shown in the second line of the enumeration.

An identifier occurrence's kind is specified to a computational role by the integer-valued `GraphIndex` attribute. This inherited attribute is set by a module computation to the value 0. The following computational roles provide the `GraphIndex` attribute:

```
*IdDefScope
*WLSimpleName
*WLQualName
*WLInsertDef
*GCSimpleName
*GCQualName
```

When there is more than one graph in the set, the default `GraphIndex` computations must be overridden. Here is a specific example:

```
TREE SYMBOL MethodIdDef INHERITS IdDefScope COMPUTE
  INH.GraphIndex = methodName;
END;
```

A module computation stores the value of the `*IdDefScope.*GraphIndex` attribute as the value of the `GraphIndex` property of the definition table key that is the value of `*IdDefScope.*Key`. Similarly, module computations establish values for the `GraphIndex` properties of the keys that are the values of the `*UseKey` attributes of the other roles listed above.

When the `GraphIndex` property of an applied occurrence indicates that the context is weak, searches are carried out in all graphs of the instantiation's set. If more than one binding is found, then `NoKey` is returned. The developer can change this behavior by providing a language-specific disambiguation function to determine the correct result (see Chapter 7 [Language], page 29).

Let '`MaxIndex`' be the maximum value of `*RootScope.*NumberOfIsoGraphs` over all instantiations of the ScopeGraphs module in the specification. If '`MaxIndex`' is greater than 1, then the developer must provide a file named `ScopeGraphs.h` as part of the specification. `ScopeGraphs.h` must contain the following declaration:

```
#define MaxIsoGraphs 'MaxIndex'
```

7 Implementing Language-Specific Behavior

The ScopeGraphs module is based upon a lookup algorithm that implements concepts developed over the years to handle visibility of names (see Section 1.2 [The generic lookup], page 4). While these concepts are present in most languages, they are often embellished with additional rules and restrictions. It is impossible to build a module that will handle any but the simplest language "out of the box"; a developer must almost always specialize it for their application. Our intent is to provide a powerful substrate for the developer, and to make the critical controls easily available.

We have chosen to define a developer interface consisting of functions called at specific points in the lookup process. A developer may provide implementations of some, all, or none of these functions; default versions of those not provided will be used automatically. The arguments of these functions are definition table keys representing the applied occurrence being sought and the candidate binding(s) found by the generic lookup. Some of the properties of those keys are set by module computations. The developer is free to set arbitrary properties of keys, so that the functions can have access to whatever information the developer deems necessary.

Some of the interface functions are called when the generic lookup finds a candidate binding. Those functions may apply semantic information to accept or reject the candidate binding. If the binding is to be rejected, then the function can specify whether and in what manner the search is to continue. Other functions are called to resolve ambiguities when several bindings have satisfied the search criteria. Finally, initialization and finalization routines allow for the use of state information that may vary from one lookup to another.

Information about the applied occurrence and the candidate binding is not always sufficient to make a decision. Sometimes the path through a scope graph between applied and defining occurrences is important, and therefore the module provides functions that the developer can call to explore the graph.

The name of the file containing the default version of each of the developer-coded functions is given after the interface description. If the default behavior of one of these routines is inadequate, it is nevertheless a good starting point for implementing a replacement. To obtain a copy of (say) the source code for `LookupBegin` as file `LookupBegin.c` in your current directory, give the Eli request:

```
-> $elipkg/Name/LookupBegin.c > LookupBegin.c
```

After modifying `LookupBegin.c`, simply add its name to your type-`specs` file to make it available.

7.1 Information access

Definition table keys are used as arguments to the interface functions because an arbitrary set of information can be associated with a definition table key without affecting the interface specification. This gives maximum flexibility without imposing either a physical or conceptual burden.

An argument representing the applied occurrence being sought is the value of the `*UseKey` attribute of that applied occurrence (see Chapter 5 [Applied Occurrences], page 21). Module computations set the following properties of the `*UseKey`:

`Sym` is an integer-valued property holding the value of the applied occurrence's `Sym` attribute.

`Coord` is a `CoordPtr`-valued property holding the source code location of the applied occurrence.

`*GraphIndex`
is an integer-valued property holding the value of the applied occurrence's `GraphIndex` attribute (see Chapter 6 [Isomorphic Scope Graphs], page 27).

Other `*UseKey` properties can be set by developer computations. To ensure that those property values are actually set before the search begins, an additional attribute, `*GotUseKeyProp`, is provided for all applied occurrences (see Chapter 5 [Applied Occurrences], page 21). `*GotUseKeyProp` should be stated as the postcondition of any computations the developer adds to set properties of `*UseKey` needed by language-specific functions. Accumulating computations must be used for this purpose (see Section "Accumulating Computations" in *LIDO – Reference Manual*). Here is an example that sets a property called `NodeTuple` to hold a representation of the range containing the applied occurrence:

```
TREE SYMBOL MethodIdUse INHERITS GCSimpleName COMPUTE
  SYNT.GotUseKeyProp +=
    ResetNodeTuple(THIS.UseKey,INCLUDING AnyScope.Env);
END;
```

An argument representing a binding candidate is the value of the `*Key` attribute of the candidate's defining occurrence. See Chapter 4 [Defining Occurrences], page 19, for the properties of that value.

7.2 Obtain a range from a qualifier

The construct '`q.i`' consists of a qualifier '`q`' and a qualified identifier '`i`'. Name analysis of '`q.i`' begins by looking up the qualifier '`q`'. Suppose that this lookup results in the definition table key '`k`'. In order to look up the qualified identifier '`i`', an appropriate `NodeTuplePtr` value representing a range will be obtained from '`k`' and the `*UseKey` of '`i`' by invoking `AccessNodesFromQualifier`. If the developer does not provide that routine, the following will be used automatically:

```
NodeTuplePtr
AccessNodesFromQualifier(DefTableKey qualifier, DefTableKey app)
/* On entry-
 *   qualifier specifies an entity from which a range can be obtained
 *   app specifies the qualified identifier sought in that range
 * On exit-
 *   AccessNodesFromQualifier represents the range
 ***/
{ return GetOwnedNodeTuple(qualifier, NoNodeTuple); }
```

Source code: `$elipkg/Name/AccessNodesFromQualifier.c`

In the case of the computational role `*GCQualName`, the developer has additional options for obtaining a suitable range (see Section 5.2 [Graph-complete search], page 24). Those options are not available for the computational role `*WLQualName`, where

`AccessNodesFromQualifier` is invoked by the module as soon as the qualifier's key is available (see Section 5.1 [Worklist search], page 22).

7.3 Is the binding acceptable?

If the generic algorithm finds a binding for an applied occurrence in $Bind(n)$, it immediately invokes one of three functions. The result of each function is one of three integer-valued flags with the names:

`AcceptBinding`
: The generic algorithm should terminate, returning the definition key of the binding.

`IgnoreContinue`
: The generic algorithm should continue as though no binding had been found.

`IgnoreSkipPath`
: The generic algorithm should continue, but it should not explore paths starting at node n (see Section 1.2 [The generic lookup], page 4).

The function called depends on the context of the search:

`isAcceptableSimple`
: is invoked in a search for a simple identifier when n is the initial node or a node reached by following a parent edge.

  If the developer does not provide `isAcceptableSimple`, the following will be used automatically:

```
int
isAcceptableSimple (DefTableKey def, DefTableKey  app)
/* On entry-
 *   def is the key of the binding found
 *   app is the UseKey of the applied occurrence sought
 * On exit-
 *   isAcceptableSimple is the desired continuation
 ***/
{ return AcceptBinding; }
```

  Source code: `$elipkg/Name/isAcceptableSimple.c`

`isAcceptableQualified`
: is invoked in a search for a qualified identifier when n is the initial node.

  If the developer does not provide `isAcceptableQualified`, the following will be used automatically:

```
int
isAcceptableQualified (DefTableKey def, DefTableKey  app)
/* On entry-
 *   def is the key of the binding found
 *   app is the UseKey of the applied occurrence sought
 * On exit-
 *   isAcceptableQualified is the desired continuation
```

```
 ***/
{ return AcceptBinding; }
```

Source code: `$elipkg/Name/isAcceptableQualified.c`

`isAcceptablePath`

is invoked in a search for a simple identifier or a qualified identifier when n is the tip of a path edge.

If the developer does not provide `isAcceptablePath`, the following will be used automatically:

```
int
isAcceptablePath (DefTableKey def, DefTableKey  app, int lab)
/* On entry-
 *   def is the key of the binding found
 *   app is the UseKey of the applied occurrence sought
 *   lab is the label of the path edge
 * On exit-
 *   isAcceptablePath is the desired continuation
 ***/
{ return AcceptBinding; }
```

Source code: `$elipkg/Name/isAcceptablePath.c`

7.4 Deciding among possible bindings

Recall that the generic algorithm traverses all paths starting at node n and made up solely of edges labeled k (see Section 1.2 [The generic lookup], page 4). When it finds an acceptable binding, it stops traversing the current path, backs up to node n, and traverses the next path made up solely of edges labeled k that starts at node n. This means that if there are several such paths, the search starting at node n may yield several bindings. The result is of type `DefTableKeyList`, implemented using the Eli `PtrList` module (see Section "Linear Lists of Any Type" in *Abstract data types to be used in specifications*).

When the complete search yields no bindings, the result is `NoKey`. If exactly one binding is found, the result is the key for that binding. If there is more than one binding, the generic algorithm invokes the function `DisambiguatePaths` and returns the result of that invocation.

If the developer does not provide `DisambiguatePaths`, the following will be used automatically:

```
DefTableKey
DisambiguatePaths (DefTableKeyList kl, DefTableKey app, int lab)
/* On entry-
 *   kl is the list of keys from the acceptable bindings
 *   app is the UseKey of the applied occurrence sought
 *   lab is the label of the edges making up the paths searched
 * On exit-
 *   DisambiguatePaths is the selected key
 ***/
{ return NoKey; }
```

Source code: `$elipkg/Name/DisambiguatePaths.c`

By returning the value `NoKey`, this implementation of `DisambiguatePaths` requires that all searches yield unambiguous results.

Another possible ambiguity occurs when the applied occurrence is in a weak context (see Chapter 6 [Isomorphic Scope Graphs], page 27). In that case, searches must be carried out in more than one scope graph and each search might yield an acceptable binding. When all searches are complete, the generic algorithm invokes `DisambiguateGraphs` and returns the result of that invocation.

If the developer does not provide `DisambiguateGraphs`, the following will be used automatically:

```
DefTableKey
DisambiguateGraphs (DefTableKey G[], int N, DefTableKey app)
/* On entry-
 *   G[] is an array with N elements giving the result for each search
 *   app is the UseKey of the applied occurrence sought
 * On exit-
 *   DisambiguateGraphs is the selected key
 ***/
{ int i;
  DefTableKey result = NoKey;

  for (i = 0; i < N; i++) {
    if (G[i] != NoKey) {
      if (result != NoKey) return NoKey;
      result = G[i];
    }
  }
  return result;
}
```

Source code: `$elipkg/Name/DisambiguateGraphs.c`

This implementation requires that exactly one of the graph searches yields an acceptable binding.

7.5 Initialization and finalization

The developer may sometimes need to initialize and/or finalize data structures that their functions use during a single lookup. Thus the generic algorithm invokes `LookupBegin` upon entry, and invokes `LookupComplete` as it exits.

If the developer does not provide `LookupBegin`, the following will be used automatically:

```
void
LookupBegin (DefTableKey app)
/* On entry-
 *   app is the UseKey of the applied occurrence sought
 ***/
{ return; }
```

Source code: `$elipkg/Name/LookupBegin.c`

If the developer does not provide `LookupComplete`, the following will be used automatically:

```
DefTableKey
LookupComplete(DefTableKey def, DefTableKey app)
/* On entry-
 *   def is the key of the binding found
 *   app is the UseKey of the applied occurrence sought
 * On exit-
 *   LookupComplete is the final result of the lookup
 ***/
{ return def; }
```

Source code: `$elipkg/Name/LookupComplete.c`

7.6 Useful graph operations

The functions described in this section are provided by the `ScopeGraphs` module. They may be invoked by developer-coded routines that need to traverse the scope graph in order to check language-specific name analysis rules. Their interfaces are defined in `LangSpecFct.h`, which must be included by any code using them.

The `DefTableKey` value of the `*ScopeKey` attribute of a symbol inheriting the `*RangeScope` role is used to represent a node in these functions (see Section 3.2 [The RangeScope role], page 14). That key has a property, `NodeTuple`, giving the value of the symbol's `*Env` attribute. Representing the node by a definition table key allows the developer to associate arbitrary language-specific information with a particular range.

Several of these functions require the developer to provide a function that will be called for each scope graph node visited. That function can determine whether or not the scope graph node satisfies some language-specific condition. The type of this function is `CallBackDTKFct`, defined by:

```
typedef int (*CallBackDTKFct)(DefTableKey)
```

The argument of a function of type `CallBackDTKFct` specifies the graph node to be examined. If the condition is *not* satisfied, the function returns 0.

`CheckPaths`
: follows paths in the scope graph that are made up of path edges with a given label. A developer-coded function is invoked at each node.

```
int
CheckPaths(DefTableKey f, DefTableKey t, int i,
             CallBackDTKFct fnc)
/* On entry-
 *   f specifies the initial node of the path
 *   t specifies the final node of the path
 *   i specifies the label of the edges making up the path
 *   fnc is a developer-code function invoked at each node
 * If no such path exists then on exit-
 *   CheckPaths=0
```

```
 * Else if any invocation of fnc yields 0 then
 *   CheckPaths exits immediately
 *   CheckPaths=0
 * Else on exit-
 *   CheckPaths=1
 ***/
```

`isAcceptablePath` might invoke `CheckPaths` to check whether a certain property holds for each node on the path between the context of the use of an identifier and its definition. For example, in Java an inherited name must be accessible in every class along the inheritance path. In this case, `fnc` would be a developer-coded function that verified the accessibility of the identifier, using information from the node and information stored about the identifier by `isAcceptablePath`.

`CheckPathsNsp`

follows paths in the scope graph that are made up of path edges with a given label. A developer-coded function is invoked at each node.

`CheckPathsNsp` is identical to `CheckPaths` except that `f` and `t` are `NodeTuple` values instead of definition table keys.

```
int
CheckPathsNsp (NodeTuplePtr f, NodeTuplePtr t, int i,
                  CallBackDTKFct fnc)
/* On entry-
 *   f specifies the initial node of the path
 *   t specifies the final node of the path
 *   i specifies the label of the edges making up the path
 *   fnc is a developer-code function invoked at each node
 * If no such path exists then on exit-
 *   CheckPathsNsp=0
 * Else if any invocation of fnc yields 0 then
 *   CheckPathsNsp exits immediately
 *   CheckPathsNsp=0
 * Else on exit-
 *   CheckPathsNsp=1
 ***/
```

`DFSCompleteFrom`

does a depth-first scan of the scope graph using path edges with a given label. A developer-coded function is invoked at each node.

```
void
DFSCompleteFrom (DefTableKey f, int i, CallBackDTKFct fnc)
/* On entry-
 *   f specifies the starting node for the search
 *   i specifies the label of the path edges to be used
 *   fnc is a developer-code function invoked at each node
 *     the results returned by fnc calls are ignored
 ***/
```

`HidesOnPaths`
: accesses bindings along a specified path that are hidden by a specified binding according to the generic lookup (see Section 1.2 [The generic lookup], page 4).

```
DefTableKey
HidesOnPaths (DefTableKey def, DefTableKey app, int i)
/* On entry-
 *   def specifies a defining occurrence that may hide other
 *         bindings
 *   app specifies an applied occurrence
 *   i specifies the label of the path edges to be searched
 * On exit-
 *   HidesOnPaths=binding that would be returned
 *     by the generic lookup when isAcceptablePath(def, app)
 *     returned IgnoreContinue
 ***/
```

`HidesNestAndPaths`
: accesses bindings that are hidden by a specified binding according to the generic lookup (see Section 1.2 [The generic lookup], page 4).

```
DefTableKey
HidesNestAndPaths (DefTableKey def, DefTableKey app)
/* On entry-
 *   def specifies a defining occurrence that may hide other
 *         bindings
 * On exit-
 *   HidesNestAndPaths=binding that would be returned
 *     by the generic lookup when isAcceptibleSimple(def, app)
 *     returned IgnoreContinue
 ***/
```

`reachableNode`
: checks for the existence of a path in a (possibly incomplete) scope graph. The path contains only path edges with a given label, and may or may not also contain parent edges.

```
int
reachableNode (DefTableKey f, DefTableKey t, int i, int par)
/* On entry-
 *   f specifies the initial node of the path
 *   t specifies the final node of the path
 *   i specifies the label of the edges making up the path
 *   if par==0 then the path cannot contain parent edges
 * If no such path exists then on exit-
 *   reachableNode=0
 * Else on exit-
 *   reachableNode=1
 ***/
```

Note that because the scope graph may not be complete when `reachableNode` is called, a node that is reported to be unreachable may become reachable later in the analysis.

`GCPathReachable`
: tests for the existence of a path in a complete scope graph. The path contains only path edges with a given label.

```
int
GCPathReachable (NodeTuplePtr f, NodeTuplePtr t, int i)
/* On entry-
 *   f specifies the initial node of the path
 *   t specifies the final node of the path
 *   i specifies the label of the edges making up the path
 * If f != t and such path exists then on exit-
 *   GCPathReachable=1
 * Else on exit-
 *   GCPathReachable=0
 ***/
```

The scope graphs are assumed to be complete. The function does not walk the graph, but it uses bitsets to decide reachability.

8 Pre-defined Identifiers

Most programming languages use natural-language representations for some basic entities. For example, the two Boolean values are often represented as `true` and `false`. There are two mechanisms for introducing such representations into a programming language:

Keyword
: A unique basic symbol of the language. Keywords have neither defining nor applied occurrences, and their meanings never change.

Pre-defined identifier
: A normal identifier, subject to the normal scope rules for identifiers, with a pre-definition in a particular range. Pre-defined identifiers have applied occurrences, but there is no defining occurrence for the pre-definition. They can be re-defined by defining occurrences in other ranges.

Pre-definitions are established by the `SGPreDefId` module, instantiated in a 'specs' file with or without an instance argument:

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

The `instance` parameter must have the same value as that of the ScopeGraphs module for which the pre-definition is intended. The `referto` parameter gives the name of the file containing the descriptions of the pre-definitions.

If the `SGPreDefId` module is instantiated, then the `MakeName` module must be instantiated to encode identifiers (see Section "Generating Optional Identifiers" in *Solutions of Common Problems*). For example, if `Ident` is the basic symbol denoting an identifier, then the `MakeName` module instantiation would be:

```
$/Tech/MakeName.gnrc +instance=Ident:inst.
```

A set of pre-definitions is enumerated in the file whose name is the `referto` argument of the `SGPreDefId` instantiation. Each pre-definition is specified by a macro call. The macro call may have some or all of the following arguments:

`_str`
: is the literal string being pre-defined.

`_sym`
: is the name of an integer-valued variable that will be set to the unique integer value derived from the `_str` argument by the `MakeName` module. If this argument is omitted, no variable is set.

`_key`
: is the name of the `DefTableKey`-valued known key representing the entity bound to the `_str` argument by this pre-definition (see Section "How to specify the initial state" in *Definition Table*). If this argument is omitted, no entity is associated with the identifier.

`_env`
: is a `NodeTuplePtr` value specifying the range in which the pre-definition occurs. If this argument is omitted, the `*Env` attribute of the root symbol of the grammar is assumed.

`_ndx`
: is an integer value specifying the graph index of the pre-definition (see Chapter 6 [Isomorphic Scope Graphs], page 27). If this argument is omitted, 0 is assumed.

The string argument need not obey the rules specified for the notation of identifier symbols. (Thus artifacts can be defined that cannot be referred to by a name in a program.) There is no need to separately declare variables or known keys. The appearance of the variable name or known key name as a macro argument ensures that it is declared.

Variable names and known key names are available for use in LIDO specifications; known key names may also be used in any other specification that has access to the definition table (see *Definition Table*).

The pre-definition file must contain a sequence of macro calls, with each macro call establishing a single pre-definition. The sequence should not contain any other code or text. Different pre-definitions may require different sets of arguments, and unfortunately the macro facility used requires each macro name to be associated with a specific set of arguments. Thus we have nine macros, all of which do the same thing. Choose the macro that fits the set of arguments you need to describe the specific pre-definition:

```
PreDefSymKeyEnvNdx(_str,_sym,_key,_env,_ndx)
PreDefSymKeyEnv(_str,_sym,_key,_env)
PreDefSymKeyNdx(_str,_sym,_key,_ndx)
PreDefSymKey(_str,_sym,_key)
PreDefSym(_str,_sym)
PreDefKeyEnvNdx(_str,_key,_env,_ndx)
PreDefKeyEnv(_str,_key,_env)
PreDefKeyNdx(_str,_key,_ndx)
PreDefKey(_str,_key)
```

In some cases, it may be necessary to pre-define ranges (for example, the range owned by the pre-defined `Object` class in Java) Therefore the module also provides macro calls for this purpose. Each call may have some or all of the following arguments:

`_key` is the name of a `DefTableKey`-valued known key allowing the developer to associate a pre-defined entity with this range (see Section 3.2 [The RangeScope role], page 14). It may have the value `NoKey`, indicating no association.

`_range` is the name of a `NodeTuplePtr`-valued variable that is set by this pre-definition and will be used as the `_env` argument for subsequent macro calls.

`_env` is a `NodeTuplePtr` value specifying the parent of the pre-defined range (see Section 3.2 [The RangeScope role], page 14). If this argument is omitted, the root symbol of the grammar is assumed.

Again we have the issue of different argument patterns needing different macro names:

```
PreDefNodeEnv(_key,_range,_env)
PreDefNode(_key,_range)
```

Pre-definitions are established before name analysis begins.

9 Name Analysis Testing

The name analysis testing module is instantiated by

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

The instance parameter must have the same value as that of the instantiation of the ScopeGraphs module under test, and the referto parameter must give the name of the grammar symbol for identifiers. Several instantiations may be used together in order to simultaneously test the bindings in a collection of instantiations of the ScopeGraphs module.

Without further specification, this module augments the ScopeGraphs module so that the generated processor produces messages of the form:

```
"file1", line 2:22 NOTE: i ("file2", line 3:7)
```

One message is written to the standard error file for each applied occurrence `i` (see Chapter 5 [Applied occurrences], page 21). The first coordinate specifies the location of the applied occurrence, the second specifies the location of the corresponding defining occurrence. No message is output for unbound identifier occurrences.

The SGProof module provides one computational role, `*ProofAppliedOcc`, that is automatically inherited by any symbol inheriting any of the applied occurrence roles (see Chapter 5 [Applied Occurrences], page 21). It provides five attributes:

`Sym` is an integer-valued attribute specifying the identifier. This attribute is usually set by a computation associated with the symbol that inherits `*ProofAppliedOcc`.

`*Key` is a `DefTableKey`-valued attribute representing the key of the corresponding defining occurrence. This attribute is usually set by a computation associated with the symbol that inherits `*ProofAppliedOcc`.

`*NoteKey` is a `DefTableKey`-valued attribute representing the defining occurrence whose coordinates are to be printed in the note. This synthesized attribute is set to the value of the `*Key` attribute by a module computation.

`*msg` is a `VOID` attribute that is the post-condition for the computation printing the note.

`*doProof` is an integer-valued attribute that controls the output of the note. This attribute is set to 1 (print the note) by a module computation.

If the `*Key` attribute of the symbol inheriting `*ProofAppliedOcc` does not represent the defining occurrence corresponding to this applied occurrence, use a developer computation to set the `*NoteKey` attribute to the correct key value.

The computation whose postcondition is the `*msg` attribute is carried out if the value of the `*doProof` attribute is 1, and is ignored if the value of the `*doProof` attribute is 0.

Override the computation of `*msg` to modify the output format or change the precondition for printing the note.

Any symbol can inherit the `ProofAppliedOcc` role, provided that the developer ensures that the `Sym` and `*Key` attributes have appropriate values. If the values are irrelevant for the particular symbol, set the `Sym` attribute to `NoIdn` and the `*Key` attribute to `NoKey`.

10 Application Program Interface

Computational roles are implemented by calls of functions that operate on the ScopeGraphs module's state. For certain language-specific features of name analysis it may be necessary or convenient for a developer to call those functions directly in a context that cannot play the corresponding role. For example, the package structure in Java requires the developer to create nodes that don't correspond to subtrees of the abstract syntax tree. The solution is to formulate a computation in a context where the information is available, even though that context does not fit the requirements of any appropriate role.

The functions described in this chapter can be invoked within any `.lido` file without further specification. Any other file invoking them must make the include file `ModelExport.h` available in the context of the invocation.

Invocations of the functions described here usually depend on values yielded by other computations, and/or on the effects that other computations have on the state of a ScopeGraphs module instantiation. Value dependence is explicit in the data flow: if a value is passed as an argument to a function call, then that value must be computed before the function is called. State dependence must be made explicit through the use of void attributes (see Section "State Dependencies" in *LIDO – Computations in Trees*). Eli uses explicit dependence information to statically schedule all computations. That schedule is independent of any particular program.

The ScopeGraphs module's roles make all of the necessary dependence relations explicit. If additional computations are necessary, then the developer must establish explicit dependence relations to define the effect of those additional computations on the state of the instantiation. Thus the developer must have an understanding of the internal state of the module instantiation in order to select the appropriate pre- and post-conditions for function invocations.

10.1 The state of an instantiation

All of the elements of the ScopeGraphs module's data structure are represented by pointers. In the documentation we will refer to pointers to elements as though they were the elements themselves.

`GraphsDescrPtr`
: represents an instantiation of the ScopeGraphs module. Each instantiation of the ScopeGraphs module results in a single value of type `GraphsDescrPtr`, which is available as the value of the attribute `*RootScope.*GraphsDescr`.

`NodeTuplePtr`
: represents a range. `NodeTuplePtr` has a distinguished value, `NoNodeTuple`, that represents no range. A `NodeTuple` is an array of scope graph nodes that are equivalent under the isomorphism defined by the ScopeGraphs module instantiation (see Section 1.1 [Scope graphs], page 4).

`ScopeNodePtr`
: represents a scope graph node. `ScopeNodePtr` has a distinguished value, `NoScopeNode`, that represents no scope graph node.

`FPItemPtr`
: represents a worklist task (see Section 5.1 [Worklist search], page 22).

As an example, consider the ScopeGraphs module instantiated by:

```
$/Name/ScopeGraphs.gnrc +instance=A_ :inst
```

Assume that `MaxIsoGraph` is set to 4, and that the attribute `A_RootScope.A_NumberOfIsoGraphs` has the value 2 (see Chapter 6 [Isomorphic Scope Graphs], page 27).

In this case, `NodeTuplePtr` values point to arrays of four `ScopeNodePtr` values. Instance `A_` uses only the array elements with graph index 0 or 1, however, since `A_RootScope.A_NumberOfIsoGraphs`=2. The `GraphsDescrPtr` value that represents the instantiation can be found at `A_RootScope.A_GraphsDescr`.

We have seen that the scope graph nodes, the parent edges of the graph, and some of the path edges can be established based on information that exists in the abstract syntax tree prior to name analysis (see Chapter 3 [Establishing the Structure of a Scope Graph], page 13). Similarly, the properties of almost all defining occurrences can be determined without reference to name analysis results (see Chapter 4 [Defining Occurrences], page 19).

The tip of a path edge that is identified by a name in the program cannot be located without a name analysis computation: the name must be resolved in order to create the edge. That resolution may depend on the creation of other path edges (see Chapter 5 [Applied Occurrences], page 21). The ordering of edge tip resolution operations is program-dependent, and therefore those operations cannot be scheduled statically.

Eli requires static scheduling of attribute computations. Therefore all of the `*WL` roles' computations create appropriate tasks and add them to a worklist rather than actually carrying them out (see Section 5.1 [Worklist search], page 22). Dependences among the tasks are represented by links between the worklist entries. Since none of the tasks is being executed, the program-specific dependences mentioned above do not affect the order of execution of the computations that create worklist tasks.

Once all tasks have been added to the worklist, they can be carried out by an indivisible operation called `FPSolveItems`. `FPSolveItems` scans through the worklist, completing each task that depends only on completed tasks, and iterates until all tasks have been completed. Although `FPSolveItems` could require n scans through a worklist of length n, in practice it seldom requires more than a single scan. The reasons seem to be that programmers don't use inherited names for edge tips, and that the ScopeGraphs module is careful to avoid out-of-order dependence by putting the individual tasks required to resolve a qualified edge tip name onto the worklist in execution order.

In some languages, names identifying path edge tips may be type-qualified (see Section 1.3 [Interaction with type analysis], page 5). Type-qualified tip names require that the abstract syntax tree be partitioned into subtrees, with the root of each subtree inheriting the `OutSideInDeps` role (see Section 3.5 [The OutSideInDeps role], page 17). `OutSideInDeps` is automatically inherited by the root symbol, so every tree node has an `OutSideInDeps` ancestor even if the language has no type-qualified tip names.

Let N be the set of tree nodes whose symbols inherit `OutSideInDeps`, and let T_n be the subtree rooted at node $n \in N$. Finally, let t_n be $T_n - \{T_k | k \in N$ is a descendent of $n\}$. Tasks must be added to the worklist, and `FPSolveItems` executed, independently for each t_n. The subtrees must be processed in outside-in order.

`FPSolveItems` cannot be executed in t_n until the instantiation's state satisfies three conditions:

- All of the ranges in t_n have been established (see Section 3.2 [The RangeScope role], page 14).
- All of the defining occurrences in t_n have been processed (see Chapter 4 [Defining Occurrences], page 19).
- All of the bound edges in t_n have been established (see Section 3.4 [The BoundEdge role], page 16).
- All of the tasks in t_n have been added to the worklist (see Section 5.1 [Worklist search], page 22).

This dependence is made explicit through the use of the abstract role `PreWork`, provided by the `ScopeGraphs` module.

`PreWork` is inherited by every role that establishes a range, processes a defining occurrence, establishes a bound edge, or creates a worklist task. The accumulating attribute `*PreWork.*PreWorkDone` represents the post-condition of the computation in each case (see Section "Accumulating Computations" in *LIDO - Reference Manual*). If a developer creates a computation that processes a defining occurrence, establishes a bound edge, or creates a worklist task, then the developer must arrange for that computation to have an attribute `*PreWork.*PreWorkDone` as a post-condition. Then the pre-condition for `FPSolveItems` in t_n is `*OutSideInDeps.*PreWorkDone`:

```
CLASS SYMBOL *OutSideInDeps COMPUTE
  SYNT.*PreWorkDone =
    CONSTITUENTS *PreWork.*PreWorkDone SHIELD *OutSideInDeps;
END;
```

`*OutSideInDeps.*FPSolved` is the void attribute that characterizes the state in which `FPSolveItems` has completed execution. In order to ensure that the subtrees are processed in outside-in order, any operation that adds a task to the worklist must have the following as a precondition:

```
INCLUDING *OutSideInDeps.*FPSolved
```

When `FPSolveItems` has completed execution, the scope graph is complete for the current t_n. Computations associated with the `*GC` roles can then be executed (see Section 5.2 [Graph-complete search], page 24). It may be necessary to intersperse type analysis computations to bind type names in declarations (see Section 1.3 [Interaction with type analysis], page 5).

10.2 Functions that create nodes, edges, and bindings

The functions described in this section do not depend on the state of the ScopeGraphs module. They can be called in any context where their arguments are available.

```
NodeTuplePtr
newNodeTuple (GraphsDescrPtr d, NodeTuplePtr p, int l)
/* Create a representation of a range in a scope graph
 *   On entry-
 *     d specifies the module instantiation
```

```
 *     p represents a range
 *     l is the current line number in the input text
 *   On exit-
 *     newNodeTuple points to a new range in d
 *     if p != NoNodeTuple then
 *       d has a parent edge with tail newNodeTuple and tip p
 *     else
 *       d has no parent edge with tail newNodeTuple
 ***/
```

The third argument is generally set to 0 when `newNodeTuple` is called by developer code, since that code is generally not associated with a particular line in the source text.

Any developer invocation of `newNodeTuple` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

```
NodeTuplePtr
ParentOfNodeTuple (NodeTuplePtr r)
/* On entry-
 *   r represents a range
 * If r is the tail of a parent edge then on exit-
 *   ParentOfNodeTuple represents the tip of r's parent edge
 * Else on exit-
 *   ParentOfNodeTuple=NoNodeTuple
 ***/

ScopeNodePtr
SelectNode (NodeTuplePtr r, int index)
/* On entry-
 *   r represents a range
 *   index is a valid graph index for r
 * On exit-
 *   SelectNode points to the scope node of r that is indexed by index

void
NodeTupleOwnerKey (NodeTuplePtr r, DefTableKey owner)
/* On entry-
 *   r represents a range
 *   owner specifies an entity
 * On exit-
 *   owner owns the range r
 ***/
```

Any developer invocation of `NodeTupleOwnerKey` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

```
DefTableKey
OwnerKeyOfNodeTuple (NodeTuplePtr r)
/* On entry-
 *   r represents a range
 * If an entity owns r then on exit-
 *   OwnerKeyOfNodeTuple specifies the entity that owns r
```

```
 * Else on exit-
 *   OwnerKeyOfNodeTuple=NoKey
 ***/
```

The developer must ensure that the owner key is set before being accessed.

```
DefTableKey
OwnerKeyOfOneNode (ScopeNodePtr n)
/* On entry-
 *   n represents a scope graph node
 * If n has an owner then on exit-
 *   OwnerKeyOfOneNode is the key of the owner
 * Else on exit-
 *   OwnerKeyOfOneNode=NoKey
 ***/
```

The developer must ensure that the owner key is set before being accessed.

```
DefTableKey
NABindIdn (NodeTuplePtr r, int i, int id, DefTableKey k, CoordPtr c)
/* Bind an identifier in a scope
 *   On entry-
 *     r[i] defines the desired scope graph node
 *     id defines the identifier
 *     k represents an existing binding or is NoKey
 *     c specifies the source coordinates or is NoPosition
 *   if Bind(r[i])(id) is undefined then
 *     if k==NoKey then Bind(r[i])(id)=NewKey()
 *     else             Bind(r[i])(id)=k
 *   else if Bind(r[i])(id)==NoKey and k!=NoKey then-
 *                      Bind(r[i])(id)=k
 *   On exit-
 *     NABindIdn returns Bind(r[i])(id)
 ***/
```

Any developer invocation of `NABindIdn` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

```
DefTableKey
addEdge (NodeTuplePtr tail, NodeTuplePtr tip, int l)
/* Add path edges
 *   On entry-
 *     tail represents a range
 *     tip represents a range
 *     label is a valid path edge label
 *   On exit-
 *     k=NewKey()
 *     A path edge has been added from tail[i] to node tip[i]
 *       for each valid scope graph index i
 *     Each added path edge has label l and key k
 *     addEdge returns k
```

```
 ***/
```

Any developer invocation of `addEdge` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

10.3 Functions that initiate worklist operations

The functions described in this section initiate worklist operations. No function in this section may be invoked in a particular context unless that invocation depends on the following attribute in that context:

```
INCLUDING *OutSideInDeps.*FPSolved
```

Any invocation of a function described in this section must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

Note: Do not use worklist operations for lookup of names if the lookup could be delayed until the graph is complete.

```
FPItemPtr
FPLookupPlain (NodeTuplePtr r, int i, DefTableKey u, GraphsDescrPtr d)
/* Establish a task to look up a simple identifier
 *   On entry-
 *     r represents a range
 *     i is a valid graph index
 *     u is the user key of the applied occurrence
 *     d specifies the module instantiation
 *   On exit-
 *     A task to seek applied occurrence u, starting at node r[i],
 *       has been added to the worklist
 *     FPLookupPlain returns a representation of the task
 ***/
FPItemPtr
FPLookupQual (FPItemPtr q, int i, DefTableKey u, GraphsDescrPtr d)
/* Establish a task to look up a qualified identifier
 *   On entry-
 *     q represents a task to find the qualifier
 *     i is a valid graph index
 *     u is the user key of the applied occurrence
 *     d specifies the module instantiation
 *   On exit-
 *     A task to seek applied occurrence u, starting at node q[i],
 *       has been added to the worklist
 *     FPLookupQual returns a representation of the task
 ***/
FPItemPtr
FPInsertDef (NodeTuplePtr r, int i, DefTableKey u, FPItemPtr def, GraphsDescrPtr d)
/* Establish a task to import a binding
 *   On entry-
 *     r represents a range
```

```
 *     i is a valid graph index
 *     u is the user key of an applied occurrence
 *     def describes a task yielding the binding to be imported
 *     d specifies the module instantiation
 *     if Bind(r[i])(u) is undefined then Bind(r[i])(u)=NoKey
 *   On exit-
 *     A task to set Bind(r[i])(u) to the result of def
 *       has been added to the worklist
 *     FPInsertDef returns a representation of the task
 ***/
```

If the applied occurrence is not already bound in the importing scope, `FPInsertDef` creates a temporary binding there that may be replaced later (see Section 10.2 [Creating Nodes], page 45).

```
FPItemPtr
FPAddEdge (NodeTuplePtr f, FPItemPtr t, int l, GraphsDescrPtr d, CoordPtr p)
/* Establish a set of equivalent edges
 *   On entry-
 *     f represents a range
 *     t represents a range
 *     l is a valid path edge label
 *     d specifies the module instantiation
 *     p specifies the source coordinates or is NoPosition
 *   On exit-
 *     A task to create a path edge from f[i] to t[i],
 *       for each valid scope graph index i,
 *       has been added to the worklist
 *     FPAddEdge returns a representation of the task
 ***/
```

10.4 Functions that search complete graphs

The functions described in this section assume that the worklist operations are complete. No function in this section may be invoked in a particular context unless that invocation depends on the following attribute in that context:

```
INCLUDING *OutSideInDeps.*FPSolved
int
getFPItemDone (FPItemPtr t)
/* On entry-
 *   t represents a task
 * If t has been completed then on exit-
 *   getFPItemDone returns 1
 * Else on exit-
 *   getFPItemDone returns 0
 ***/
DefTableKey
getFPItemKey (FPItemPtr t)
```

```
/* On entry-
 *   t represents a completed lookup task
 * On exit-
 *   getFPItemKey returns the key found by the task
 ***/
DefTableKey
getFPEdgeKey (FPItemPtr t, int l)
/* On entry-
 *   t represents a completed FPAddEdge task
 * On exit-
 *   getFPEdgeKey returns the edge key found by the task
 ***/
```

In the following functions, the argument `u` is a unique key whose properties characterize an applied occurrence. At least the properties `Sym`, `GraphIndex`, and `Coord` must be set to suitable values prior to invoking the function. Further properties may be set to support functions implementing the developer interface (see Chapter 7 [Implementing Language-Specific Behavior], page 29).

```
DefTableKey
GCLookupPlainId (NodeTuplePtr r, int i, DefTableKey u)
/* Look up a simple identifier
 *   On entry-
 *     r represents a range
 *     i is a valid graph index
 *     u is the user key of the applied occurrence
 *   On exit-
 *     GCLookupPlainId returns the key found for u starting at r[i]
 ***/
DefTableKey
GCLookupQualId (NodeTuplePtr r, int i, DefTableKey u)
/* Look up a qualified identifier
 *   On entry-
 *     r represents a range
 *     i is a valid graph index
 *     u is the user key of the applied occurrence
 *   On exit-
 *     GCLookupQualId returns the key found for u starting at r[i]
 ***/
```

Here the range `r` is obtained from the qualifier, and the search does not follow parent edges (see Section 1.2 [The generic lookup], page 4).

```
DefTableKey
GCLookupLocalId (NodeTuplePtr r, int i, DefTableKey u)
/* Look up an identifier
 *   On entry-
 *     r represents a range
 *     i is a valid graph index
```

```
 *     u is the user key of the applied occurrence
 *   On exit-
 *     GCLookupLocalId returns the key found for u starting at r[i]
 ***/
```

This search seeks *only* a binding $Bind(r[i])(id)$, where id is the identifier of the applied occurrence.

10.5 Functions that pre-define symbols

The `SGPreDefId` module provides the following functions, used to implement the pre-definition macros (see Chapter 8 [Pre-defined Identifiers], page 39). They may be called in any `.lido` file, and in any other context that includes the header file `SGPreDefMod.h`.

```
void
SGPreDefineSym(const char *name, int *sym)
/* Pre-code an identifier
 *   On entry-
 *     name is the external representation of the identifier
 *     sym addresses an integer-valued variable
 *   On exit-
 *     The variable sym contains the internal representation of name
 ***/
void
SGPreDefine (const char *name, int *sym,
             NodeTuplePtr env, int index, DefTableKey k)
/* Pre-define an identifier
 *   On entry-
 *     name is the external representation of the identifier
 *     sym addresses an integer-valued variable
 *     env specifies the range in which to define the identifier
 *     index specifies the graph in which to define the identifier
 *     k is the known key bound to the identifier
 *   On exit-
 *     The variable sym contains the internal representation of name
 *     The binding has been created
 ***/
```

Any developer invocation of `SGPreDefine` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

```
void
SGPreDefineNode (GraphsDescrPtr descr, const char *name, int *sym,
                 DefTableKey key, NodeTuplePtr *node, NodeTuplePtr parent)
/* Pre-define a range
 *   On entry-
 *     descr is the metadata for the ScopeGraphs module instantiation
 *       being initialized
 *     name is the external representation of the pre-defined range's owner
 *     key is the known key for the pre-defined range's owner
```

```
*     node addresses a NodeTuplePtr-valued variable
*     parent is tip of the pre-defined range's parent edge
*   On exit-
*     The variable sym contains the internal representation of name
*     the variable node contains a pointer to the pre-defined range
***/
```

Any developer invocation of `SGPreDefineNode` must guarantee that the `*PreWorkDone` attribute of some node inheriting `*PreWork` is assigned after the function returns.

Index

A

abstract syntax tree 3, 9, 10, 44
acceptable binding, check for 5
`AcceptBinding` 31
accumulating attribute 45
`addEdge` 47
`AnyScope` 14, 16, 19, 24, 30
applied occurrence . . 3, 6, 21, 23, 27, 29, 33, 39, 41, 50
attribute dependence 6, 43
attribute, accumulating 45

B

basic symbol 3, 9
`Bind` 4, 5, 6, 19, 23, 31, 47, 49, 51
binding 3, 29
binding, check for acceptable 5
bound edge 45
`BoundEdge` 16

C

`CallBackDTKFct` 34
check for acceptable binding 5
`CheckPaths` 34
`CheckPathsNsp` 35
`ChkIdUse` 21, 25
complete scope graph 22, 49
computational role 13, 22, 44
concrete grammar 9
concrete syntax 9
context, syntactic 5
`Coord` 19, 21, 30, 50
`CoordPtr` 19, 30, 47, 49

D

defining occurrence . . 3, 19, 21, 23, 27, 39, 41, 44, 45
definition table 40
definition table key 5, 14, 28, 29, 30, 31, 32, 34, 35, 41
`DefTableKey` . . 13, 14, 16, 17, 19, 21, 25, 30, 31, 33, 34, 39, 41, 46, 48, 49, 51
`DefTableKeyList` 32
dependence among attributes 6, 43
dependence among tasks 44
`DependsOn` 23
developer-coded functions 4
`DFSCompleteFrom` 35
`DisambiguateGraphs` 33
`DisambiguatePaths` 32
disambiguation 5, 28
`doProof` 41

E

edge 4
edge tail 4, 6, 16, 17, 22, 23, 27, 46
edge tip 4, 6, 14, 16, 17, 22, 27, 32, 44, 46, 52
edge, parent 5
`EdgeKey` 16, 17, 23
`EdgeLabel` 16, 17, 23
Eli's Typing module 6
entity 3, 14
`Env` 13, 14, 16, 17, 19, 24, 34, 39
extension function 25

F

fictitious outer block 13
finalization 29, 33
`FPAddEdge` 49, 50
`FPInsertDef` 48
`FPItem` 22
`FPItemPtr` 22, 43, 48, 49
`FPLookupPlain` 48
`FPLookupQual` 48
`FromNodeTuple` 16, 17, 23
functions, developer-coded 4

G

`GCLocalName` 24
`GCLookupLocalId` 50
`GCLookupPlainId` 50
`GCLookupQualId` 50
`GCPathReachable` 37
`GCQualName` 25, 28, 30
`GCSimpleName` 25, 28, 30
`getFPEdgeKey` 50
`getFPItemDone` 49
`getFPItemKey` 49
`GotDefKeyProp` 19
`GotEnv` 13
`GotUseKeyProp` 21, 30
graph, scope 3
Graph-complete search 22
`GraphIndex` 28, 30, 50
`GraphsDescrPtr` 43, 45, 48, 51

H

`HidesNestAndPaths` 36
`HidesOnPaths` 35

I

IdDefScope 19, 28
identifier 3, 9
identifier binding 29, 30, 31, 32, 36
identifier, kind of 4, 27
identifier, overloaded 6
identifier, qualified 5
IgnoreContinue 31, 36
IgnoreSkipPath 31
indication 6
indivisible operation 44
inheritance 22
initialization 29, 33
input text 9
isAcceptablePath 32, 35, 36
isAcceptableQualified 31
isAcceptableSimple 31
isomorphic scope graphs 4, 13, 27, 43

J

Java anonymous class 6, 17

K

Key 19, 21, 28, 41
keyword 39
kind of identifier 4, 27
known key 39

L

label 16, 17, 23
label, path edge 4, 5
LangSpecFct.h 34
local name 24
LookupBegin 33
LookupComplete 33

M

MakeName module 39
MaxIsoGraphs 28
MaxKindsPathEdge 5
missing edges 22
ModelExport.h 43
module instantiation 4, 39, 41, 44
module MakeName 39
module ScopeGraphs 1
module SGPreDefId 39
module SGProof 41
msg 41

N

NABindIdn 47
name, type-qualified 6, 13, 17, 44
nested scopes 9
newNodeTuple 45
NodeTupleOwnerKey 46
NodeTuplePtr 13, 14, 16, 17, 19, 23, 24, 27, 30, 35, 39, 43, 45, 48, 50, 51
nonterminal symbol 10
NoteKey 41
NumberOfIsoGraphs 27

O

OutSideInDeps 6, 17, 44, 48, 49
OutSideInDeps.GotEntityTypes 6
overloaded identifier 6
OwnedNodeTuple 14
OwnerKeyOfNodeTuple 14, 46
OwnerKeyOfOneNode 47

P

Parent 14
parent edge .. 4, 5, 13, 14, 22, 25, 31, 36, 44, 46, 50, 52
ParentOfNodeTuple 46
parse tree 9
Pascal with statement 6, 17
path edge .. 4, 6, 13, 16, 17, 22, 23, 32, 34, 44, 47, 49
path edge label 4, 5
post-condition 41, 43, 45
pre-condition 43, 45
pre-defined identifier 39
PreDefKey 40
PreDefKeyEnv 40
PreDefKeyEnvNdx 40
PreDefKeyNdx 40
PreDefSym 40
PreDefSymKey 40
PreDefSymKeyEnv 40
PreDefSymKeyEnvNdx 40
PreDefSymKeyNdx 40

Q

qualified identifier 5, 22, 25, 31
qualifier 50

R

range .. 3, 4, 9, 13, 14, 16, 19, 21, 24, 27, 30, 34, 39, 43, 45, 48, 50, 51
RangeScope 14, 16, 17, 34
reachableNode 36
RootScope 13, 16, 27, 43

S

scope ... 3
Scope ... 24
scope graph ... 3, 4, 34, 43
scope graph index ... 27, 30, 39
scope graphs, isomorphic ... 4
ScopeGraphs module ... 1
ScopeGraphs module ... 9
ScopeGraphs module instantiation ... 4, 13, 27
ScopeGraphs.h ... 28
ScopeKey ... 13, 14, 25, 34
ScopeNodePtr ... 43, 46
SelectNode ... 46
semantic information ... 29
SetTypeOfEntity ... 6
SGPreDefId module ... 39
SGPreDefId module ... 51
SGPreDefine ... 51
SGPreDefineNode ... 51
SGPreDefineSym ... 51
SGPreDefMod.h ... 51
SGProof module ... 41
simple identifier ... 22, 25, 31
strong context ... 27
Sym ... 19, 21, 24, 30, 41, 50
symbol, basic ... 3
SymErr ... 25
SymMsg ... 25
syntactic context ... 5, 27

T

tailEnv ... 16, 17, 23
task dependence ... 44
terminal symbol ... 9
tipEnv ... 17
tipFPItem ... 23
toEnv ... 16, 17, 23
ToNodeTuple ... 16, 17, 23
tree, abstract syntax ... 3
type analysis ... 4, 45
type analyzer ... 6
type-qualified name ... 6, 13, 17
type-qualified names ... 44
TypedDefId ... 6
TypedUseId ... 6
TypeIsSet ... 6
Typing module ... 6

U

universe ... 3
unsuccessful search ... 25
UseKey ... 21

V

visibility ... 10, 29
VOID ... 13

W

weak context ... 27, 33
WLCreateEdge ... 23
WLInsertDef ... 23, 28
WLQualName ... 22, 28, 30
WLSimpleName ... 22, 28
Worklist search ... 22
worklist task ... 44