Tutorial for Name Analysis Using ScopeGraphs

Uwe Kastens

University of Paderborn
D-33098 Paderborn
Germany

W. M. Waite

University of Colorado
Boulder, CO 80309-0425
USA

Table of Contents

1 Kernel Language

This chapter considers the name analysis problem for a very simple subset of a language called NameLan. We begin with specifications of the phrase structure and basic symbols. A processor built from these specifications will accept a text in NameLan and build a tree. These specifications can then be augmented by others, enhancing that processor to make it solve the subset's name analysis problem. The sample text that we will analyze is:

machar.nl[1]==

```
int radix;
{ float a = 1.0;
  while (((a + 1.0) - a) - 1.0 == 0.0)
    a = a + a;

  float b = 1.0;
  while ((a + b) - a == 0.0)
    b = b + b;

  radix = (a + b) - a;
}
```

This macro is attached to a non-product file.

This program determines the radix of the floating point representation of the machine on which it runs, and stores that value in a global variable. NameLan does not require fully-parenthesized expressions, but parentheses are needed here to guarantee the order of evaluation within the expressions. (Optimizers do not generally violate the integrity of parenthesized expressions.)

1.1 Text structure

Text is considered to be made up of a sequence of *comments* and *basic symbols*. Comments are character sequences to be ignored, while basic symbols are character sequences that correspond to terminal symbols of the grammar describing the phrase structure of the text. Basic symbols can be grouped according to the information they carry:

Delimiter A marker that serves to establish the structure of the source text. (Examples: `while` and `(`.)

Denotation A representation of a source language entity. Different occurrences of the same denotation all represent the same entity. (Examples: `0.0` and `int`.)

Identifier A name for some entity. Different occurrences of the same identifier may name different entities. (Examples: `radix` and `a`.)

Name analysis is concerned with establishing the entity named by each identifier in the text.

Since basic symbols correspond to terminal symbols of the grammar describing a language's phrase structure, it is generally simplest to start by specifying that grammar (see Section "Context-Free Grammars and Parsing" in *Syntactic Analysis*). The terminal symbols are then precisely the symbols that do not occur on the left-hand side of any grammar rule.

The *concrete syntax* of a language is a specification of the phrase structure of a program written in that language. Here is a concrete syntax for the NameLan subset of interest here:

Phrase structure[2]==

```
Program:        Declaration* Block.

Declaration:    VarDecl.

Block:          '{' DeclStmt* '}'.
DeclStmt:       VarDecl / Statement.

VarDecl:        Type VarDefs ';'.
VarDefs:        VarDef // ',' .
VarDef:         Ident [ '=' Expr ].

Type:           'int' / 'float' .

Statement:      Name '=' Expr ';' /
                'if' Expr Statement $'else' /
                'if' Expr Statement  'else' Statement /
                'while' '(' Expr ')' Statement /
                Block.

Expr:           AExpr [ Relop AExpr ] .
AExpr:          [Addop] Term / AExpr Addop Term .
Term:           Factor / Term Mulop Factor .
Factor:         Name / Integer / Real / '(' Expr ')' .

Relop:          '<' / '<=' / '==' / '!=' / '>=' / '>'.
Addop:          '+' / '-'.
Mulop:          '*' / '/'.

Name:           Ident.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

(See Section "The effect of a $-modification" in *Syntactic Analysis*, for an explanation of the `if` statement definition.)

NameLan.con[3]==

Phrase structure[2]

This macro is attached to a product file.

Notice that many of the terminal symbols of this grammar are given literally (for example, `'='`, `';'`, and `'if'`). Clearly, these literal terminals need no further specification. We are left with only three terminals (and the comment) whose form the grammar leaves unspecified:

NameLan.gla[4]==

```
Ident:          C_IDENTIFIER
```

```
Integer:        C_INTEGER
Real:           C_FLOAT
                C_COMMENT
```

This macro is attached to a product file.

This specification indicates that the formats of these NameLan terminal symbols and comments are identical to those of their counterparts in C (see *Lexical Analysis*).

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Kernel > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Recall that we suggested that you begin by specifying the grammar that describes a language's phrase structure. File `NameLan.con` is such a grammar specification. Eli can't generate a processor from `NameLan.con` because that file does not specify the structure of the non-literal terminals. Nevertheless, ask Eli to do so and examine the resulting errors:

   ```
   -> NameLan.con :exe :error >
   ```

   Compare the error report with file `NameLan.gla`. How do they differ, and why?

2. Eli *is* able to generate a processor from the two files `NameLan.con` and `NameLan.gla` together. In order to generate a processor from a number of specification files, Eli needs a single file of type `specs` listing those specifications:

   Text.specs[5]==

   ```
   NameLan.con
   NameLan.gla
   ```

   This macro is attached to a product file.

   Ask Eli to generate a processor from `Text.specs` and save the executable for that processor in the current directory:

   ```
   -> Text.specs :exe > kern
   ```

   Apply `kern` to `machar.nl`:

   ```
   -> !./kern machar.nl
   ```

   (You can also exit Eli and run `kern` as you would any program.)

   a. Why is there no output?
   b. Delete the second line of `machar.nl` and again apply `kern` to it. Is the output what you expected?
   c. Briefly explain what your generated processor is actually doing.

1.2 Tree Structure

`NameLan.con` is the concrete syntax of our NameLan subset. It describes all of the phrases that can be used to construct a program like `machar.nl`. The processor generated from `NameLan.specs` in the last section verifies that a text file represents a syntactically-correct program in the NameLan subset.

An Eli-generated processor represents a program internally by a tree, and carries out tasks like name analysis by computing the values of *attributes* attached to the nodes of the tree (see *LIDO – Computation in Trees*). The *abstract syntax* of a language describes all of the possible trees that a generated processor could build to represent syntactically-correct programs in that language. Eli uses LIDO notation to define the abstract syntax of a language (see Section "top" in *LIDO – Reference Manual*).

Eli can deduce an abstract syntax from a concrete syntax, but the trees described by that abstract syntax might not be the best ones for subsequent analysis. The rules describing expressions and operators in `NameLan.con` provide an excellent example. They implement the operator precedence and association rules of NameLan, and guarantee that the phrase structure reflects the relationship between operators and operands (see Section "Using structure to convey meaning" in *Syntactic Analysis*). Those rules determine the structure of the tree to be built.

Consider the expression in the second `while` statement of `machar.nl`:

```
(a + b) - a == 0.0
```

According to the grammar, this is an `Expr` phrase with the following structure:

```
AExpr Relop AExpr
```

The interpretation of this structure is that the equality operator `==` compares the subexpression `(a + b) - a` with the number `0.0`. It reflects the fact that addition and subtraction operations should be carried out before comparison operations, i.e. that they should have higher precedence (see Section "Operator precedence" in *Syntactic Analysis*). `NameLan.con` lists groups of operators in order of increasing precedence. This program text would therefore be represented internally by a tree with three subtrees: one representing `(a + b) - a`, one representing `==`, and one representing `0.0`.

The parentheses in the expression to the left of `==` indicate that `a + b` is the left operand of the subtraction. Suppose that those parentheses were omitted. According to the grammar, `a + b - a` is an `AExpr` phrase with the structure:

```
AExpr Addop Term
```

Thus the structure is the same as that of the parenthesized form.

The `+` and `-` operators have the same precedence; `NameLan.con` implements the rule that these operators associate to the left (see Section "Operator associativity" in *Syntactic Analysis*).

There is no need to retain the distinction between different levels of expression (e.g. `AExpr` and `Term`), or between different precedences of operators (e.g. `Relop` and `Addop`), in a tree representing `machar.nl` because the information they provided is already built into the tree structure. We therefore map all expression symbols to `Expr` and all operator symbols to `Oper` (see Section "The Relationship Between Phrases and Tree Nodes" in *Syntactic Analysis*).

Mappings from concrete symbols to abstract symbols[6]==

```
MAPSYM
Expr ::= AExpr Term Factor.
Oper ::= Relop Addop Mulop.
```

This macro is defined in definitions 6.
This macro is invoked in definition 7.

NameLan.map[7]==

Mappings from concrete symbols to abstract symbols[6]

This macro is attached to a product file.

`NameLan.map` removes the distinction between symbols of the concrete syntax. We also need to be able to *introduce* distinctions where a single symbol of the concrete syntax plays different roles in different contexts. For example, identifiers are always represented by the concrete symbol `Ident`. But two identifiers can require very different computations depending on the context in which they occur. Because computations are associated with symbols in the tree, we need different abstract syntax symbols to represent different identifier occurrences.

We recommend that, wherever possible, abstract syntax symbols needed to distinguish identifier contexts *not* be added to the concrete syntax (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). The reason for this recommendation is that the parser often does not have sufficient contextual information to recognize the phrase, but that information is available when the tree is being built.

The abstract syntax derived from `Text.specs` contains the following rule:

```
RULE rule_024:
Name ::= Ident
END;
```

In order to distinguish the identifier context of a simple name, let's add the symbol `SimpleName` to the abstract syntax. This can be done by writing two explicit LIDO rules:

Abstract syntax of identifiers[8]==

```
RULE: Name       ::= SimpleName END;
RULE: SimpleName ::= Ident      END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

When the parser reports that it has recognized the phrase correponding to `rule_024`, the tree builder will automatically build two tree nodes described by these two new rules instead of building a node described by `rule_024`.

We use a similar strategy to distinguish the two contexts in which an identifier is declared:

Abstract syntax of identifiers[9]==

```
RULE: VarDef     ::= VarDefName          END;
RULE: VarDef     ::= VarDefName '=' Expr END;
RULE: VarDefName ::= Ident               END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

As we extend NameLan to demonstrate more complex name analysis issues, we will need to distinguish other identifier contexts.

Abstract syntax tree[10]==

Abstract syntax of identifiers[8]

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,
65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

NameLan.lido[11]==

Abstract syntax tree[10]

This macro is attached to a product file.

We can now specify a processor that will take account of these mappings when deducing the abstract syntax, and produce a simpler tree from `machar.nl`:

Specification files[12]==

```
NameLan.con
NameLan.gla
NameLan.map
NameLan.lido
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

NameLan.specs[13]==

Specification files[12]

This macro is attached to a product file.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Tree > .
```

None of these files will have write permission in your current directory.

1. Extract the description of the abstract syntax deduced directly from the concrete syntax for the kernel language and open it in your editor:

   ```
   -> Text.specs :absyntax <
   ```

   Using these abstract syntax rules, draw the tree representing the following expression from `machar.nl`:

   ```
   (a + b) - a == 0.0
   ```

   Label each node with the rule name, and indicate any additional information contained in a leaf. To get you started, the root is a `rule_011` node and its second child is a `rule_028` node. One of the leaves is a `rule_015` node whose additional information is the internal representation of the denotation `0.0`.

2. Extract the description of the mapped abstract syntax for the kernel language and open it in your editor:

   ```
   -> NameLan.specs :absyntax <
   ```

   a. Draw the tree for `(a + b) - a == 0.0` using these abstract syntax rules.

   b. Does this tree provide any less descriptive power than the tree drawn from the original abstract syntax? Explain briefly.

   c. Search for `Expr ::=` and `Oper ::=`. Verify that the nodes of the tree described by these rules can represent any possible expression in the language.

1.3 Basic name analysis

When you read `machar.nl`, your programming experience tells you that that it is manipulating three variables. Two of these variables are named `a` and `b`, and are used to do floating point computations that explore the properties of the number representation. The third variable, named `radix`, is set to an integer value that is the radix of the representation.

In order to formalize this intuition, the designer of NameLan makes use of four concepts that relate identifiers to the entities they name (see Section "Fundamentals of Name Analysis" in *Name Analysis Reference Manual*).

- 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.

As language designers, we have decided to specify the rules for the kernel language as follows:

- The scope of a defining occurrence `VarDefName` is the smallest enclosing `Block` or `Program`.
- An applied occurrence that is a `SimpleName` '`i`' identifies the binding '`(i,k)`' having the smallest scope encompassing '`i`'.

A binding '`(radix,k)`', where '`k`' is the integer variable entity named by `radix`, is associated with the defining occurrence of `radix` in the first line of `machar.nl`. According to the first rule, the scope of that defining occurrence (and hence the scope of '`(radix,k)`') is the `Program` phrase – the complete program text. Since the applied occurrence of `radix` in line 10 lies within that scope, it identifies '`(radix,k)`' by the second rule. We can therefore conclude that this applied occurrence names the integer variable named by the defining occurrence of `radix` in the first line of `machar.nl`. Similar statements can be made about the variables `a` and `b`, the scope of whose defining occurrences is the `Block` phrase.

Consider the following trivial program:

```
float x = 0;
{int x; x = 2;}
```

The scope of the defining occurrence of `x` on line 1 is the `Program` phrase, while that of the defining occurrence of `x` on line 2 is the `Block` phrase making up line 2. Suppose that the binding associated with the defining occurrence on line 1 is '`(x,kf)`' and the one associated

with the defining occurrence on line 2 is '`(x,ki)`'. Which of these two bindings will the applied occurrence of `x` on line 2 identify?

Both of the scopes encompass the applied occurrence, but '`(x,ki)`' has the smaller scope. Therefore the applied occurrence will identify '`(x,ki)`'. The term usually used for this effect is *hiding*. We say that the binding in the `Block` phrase *hides* any binding for the same identifier in a surrounding context.

Name analysis is a computation carried out on the tree that is the internal representation of the program being analyzed (see Section "Fundamentals of Name Analysis" in *Name Analysis Reference Manual*). 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 goal of the computation is to determine the entities named by the program's identifiers, based on the concepts and language rules discussed above.

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 such text region to an abstract syntax subtree encompassing that region. We call such a subtree a *range*; several scopes may map into the same range. A scope is represented by the root node of its range.

For example, consider the defining occurrence `radix` in the first line of `machar.nl`. According to the scope rules of the kernel language, the scope of that defining occurrence is the `Program` phrase. The only abstract syntax subtree encompassing that scope is the entire tree, rooted in the `Program` node. We therefore conclude that the scope of the defining occurrence `radix` is represented by the `Program` node.

Applying similar reasoning to the defining occurrence of `a` in `machar.nl`, we see that its scope is encompassed both by a `Block` subtree and by the `Program` tree. By our definition, either of the `Program` node or the `Block` node could be chosen to represent the scope of the defining occurrence `a`. The developer should choose the range that reflects the semantics of the language, and will simplify the name analysis computations. In the case of the kernel language, the scope of a defining occurrence `VarDefName` is the *smallest* enclosing `Block` or `Program` phrase. Therefore the developer should choose the `Block` node to represent the scope of `a`. (For a more complex situation, see Section 6.2 [Position control], page 73.)

Specific identifiers are character sequences in an input text, and are represented internally by integer values (see Section "Character String Storage" in *The Eli Library*). Identifiers with the same spelling are represented by the same integer (see Section "Available Processors" in *Lexical Analysis*).

Conventionally we use the attribute `Sym` to hold the internal representation of an identifier. To cut down on the number of attribute declarations, we use an attribute name specification to specify that all attributes named `Sym` have type `int` (see Section "Types and Classes of Attributes" in *LIDO Reference Manual*). That allows us to use the attribute name `Sym` without having to write an individual declaration for each symbol to which it is attached:

Attribute representing an identifier[14]==

```
ATTR Sym: int;
```

This macro is invoked in definition 18.

An entity is represented internally by a definition table key, of type `DefTableKey` (see Section "How to create and use definition table keys" in *Definition Table*). Definition table keys must be passed around the tree and be stored as attributes at many different nodes, and we conventionally use the attribute name `Key` for this purpose:

Attribute referencing an entity[15]==

```
ATTR Key: DefTableKey;
```

This macro is invoked in definition 18.

Name analysis determines a value of the `Key` attribute for each identifier occurrence in the program.

The value of the `Key` attribute of a defining occurrence is known in the context of that defining occurrence. For example, the fact that `radix` names a distinct program entity is known at the defining occurrence because the context is a declaration. A `DefTableKey` value representing that distinct entity can therefore be assigned to the `Key` attribute of the corresponding tree node, without regard to any other computations (see Section "Defining Occurrences" in *Name Analysis Reference Manual*). The *properties* of that definition table key (e.g. the fact that the entity is an integer variable) depend on other computations, but the key itself does not (see *Property Definition Language*).

The value of the `Key` attribute of an applied occurrence is determined by identifying a binding of that applied occurrence. When the applied occurrence is a simple name, it identifies the binding whose scope was mapped to the smallest range containing that applied occurrence.

Scope rules determining the relationships between identifiers and entities follow patterns that do not vary significantly among programming languages, and standard attribute computations have therefore been developed to implement name analysis. Eli has packaged those computations in a *specification module* (see *Introduction of specification modules*). The developer can obtain them by instantiating that module:

Specification files[16]==

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

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

An Eli specification module provides *computational roles*: constellations of attributes and operations that are relevant for particular contexts in the tree. For example, a tree node playing the role of a defining occurrence is given three attributes and a particular set of computations to be carried out at specific visits to that node (see Section "Defining Occurrences" in *Name Analysis Reference Manual*). Computational roles must be associated with appropriate tree nodes so that the computations they describe can be carried out as those nodes are visited during traversals of the tree. This is done by using LIDO *inheritance* (see Section "Inheritance of Computations" in *LIDO - Reference Manual*).

The `ScopeGraphs` module provides three roles that can be used to implement basic name analysis computations:

- `RangeScope` implements the range concept (see Section "The RangeScope role" in *Name Analysis Reference Manual*).

- `IdDefScope` implements the concept of a defining occurrence (see Section "Defining Occurrences" in *Name Analysis Reference Manual*).
- `GCSimpleName` implements the concept of an applied occurrence that is a simple name (see Section "Graph-complete search" in *Name Analysis Reference Manual*).

Our scope rule for the kernel language defines the scope of a `VarDefName` as the smallest enclosing `Block`, which we have mapped into a range rooted in a `Block` node. The appropriate specifications associating the name analysis roles with symbols of the kernel language abstract syntax are thus:

Tree nodes playing ScopeGraphs roles[17]==

```
SYMBOL Block      INHERITS RangeScope   END;
SYMBOL VarDefName INHERITS IdDefScope   END;
SYMBOL SimpleName INHERITS GCSimpleName END;
```

This macro is invoked in definition 18.

The `VarDefName` rule also describes `Program` as a possible scope. `Program` is the symbol at the root of the tree, which automatically inherits the `RootScope` role (see Section "The RootScope role" in *Name Analysis Reference Manual*). `RootScope` combines the behavior of `RangeScope` with attributes and operations that are specific to the root of the tree. Therefore `Program` should *not* inherit `RangeScope`. We also recommend that no symbol explicitly inherit `RootScope`.

The computations associated with the three roles use an internal data structure called a *scope graph* (see Section "Fundamentals of Name Analysis" in *Name Analysis Reference Manual*). Each scope graph node specifies a set of bindings; scope graph edges describe search paths used to find a binding for a given identifier.

Any tree node that plays the `RangeScope` or `RootScope` role is the root node of a range, and corresponds to a scope graph node. (There may also be scope graph nodes that do not correspond to tree nodes.) The set of bindings specified by a scope node corresponding to the root node of a range is exactly the set of bindings mapped to that range.

A basic understanding of the way in which the computations attached to the abstract syntax tree by computational roles use the scope graph will help you to understand why we do things in a certain way, and how to extend name analysis to more complex situations. We will therefore briefly look at the name analysis of `machar.nl`.

A computation inherited by the root node of the tree from the `RootScope` role creates a scope graph node for the `Program` range. Computations inherited from `RangeScope` by the `Block` node of the tree also create a scope graph node, and use the tree structure to determine that the newly-created node should be connected by a *parent edge* to the created ancestor node (see Section "Scope graphs" in *Name Analysis Reference Manual*). Parent edges represent the enclosure relation for ranges. The `Block` is enclosed by the `Program` in `machar.nl`, so the parent edge is directed from the scope node being created (the *tail* of the parent edge) to the created ancestor scope node (the *tip* of the parent edge).

Computations inherited from the `IdDefScope` role use the tree structure to find the enclosing range of each `VarDefName` and specify the binding `(VarDefName.Sym,VarDefName.Key)` at the scope node corresponding to that range.

The search for a binding for the applied occurrence of `radix` in `machar.nl` begins in the scope node corresponding to the smallest range containing that `SimpleName` (see Section

"The generic lookup" in *Name Analysis Reference Manual*). This scope node specifies two bindings, for `a` and `b`, but no binding for `radix`. Because this scope node is the tail of a parent edge, the search continues in the scope node that is the tip of that edge and is successful.

The complete specification of a processor that does name analysis on the kernel language requires the `ScopeGraphs` module instantiation, the concrete syntax and symbol mapping for NameLan, and the LIDO file connecting the abstract syntax to the module:

Abstract syntax tree[18]==

```
Attribute representing an identifier[14]
Attribute referencing an entity[15]
Tree nodes playing ScopeGraphs roles[17]
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

A processor generated from this specification will build a tree and perform name analysis, but it will not produce any output. For the purpose of this tutorial, however, we want our processor to make the results of the name analysis visible. We want to be able to see the relationships between applied and defining occurrences so that we can check our understanding. Eli provides another module that we can instantiate for this purpose (see Section "Name Analysis Testing" in *Name Analysis Reference Manual*).

Bindings.specs[19]==

```
NameLan.specs
$/Name/SGProof.gnrc +referto=Ident :inst
```

This macro is attached to a product file.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Basic > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Extract the description of the abstract syntax on which name analysis is based, and open it in your editor:

   ```
   -> NameLan.specs :absyntax <
   ```

   a. Verify that the tree for `machar.nl` will contain nodes that have the `GCSimpleName` role.

   b. How can you distinguish rules that were deduced from the concrete syntax from rules that were added explicitly?

2. Draw the scope graph that would be constructed for `machar.nl`. Use circles for the nodes, and write an identifier in the circle if that identifier has a binding specified by the node. Explain briefly how the generic lookup would use your scope graph to find an appropriate binding for the occurrence of `radix` in the last line of `machar.nl` (see Section "The generic lookup" in *Name Analysis Reference Manual*).

3. Use `Bindings.specs` to generate a processor named `bind` that will show bindings for applied occurrences, executable for that processor in the current directory:

   ```
   -> Bindings.specs :exe > bind
   ```

   Apply `bind` to `machar.nl`:

   ```
   -> !./bind machar.nl
   ```

   (You can also exit Eli and run `bind` as you would any program.)

   a. Does the output indicate that the name analysis is correct according to the language rules? Explain briefly.

   b. Delete the variable declaration from the first line of `machar.nl` and again apply `bind` to it. Is the output what you expected?

   c. Duplicate the variable declaration in the first line of `machar.nl` and again apply `bind` to it. Is the output what you expected?

1.4 Error reporting

If the first line were removed from `machar.nl`, the applied occurrence of `radix` in the last line would not lie in the scope of any defining occurrence of `radix`. The scope rules of the previous section do not define the result when an applied occurrence does not lie in the scope of a defining occurrence of its identifier.

Recall that the search for the meaning of the applied occurrence of `radix` begins in the scope node for the `Block`, and follows the parent edge to the scope node for the `Program`. There is now no binding for `radix` in that node, and the node has no parent edge to follow. At that point, the computation sets the attribute `SimpleName.Key` to the value `NoKey` (see Section "How to create and use definition table keys" in *Definition Table*).

A `NoKey` result always indicates that the name analysis computation has been unable to find a suitable binding for an applied occurrence. Therefore the ScopeGraphs module provides a role, `ChkIdUse`, which can be inherited by an applied occurrence to provide an error report when the entity is `NoKey`:

Report an undefined identifier[20]==

```
SYMBOL SimpleName INHERITS ChkIdUse END;
```

This macro is invoked in definition 22.

Now suppose that the first line of `machar.nl` were:

```
int radix; float radix;
```

Both defining occurrences of `radix` have the `Program` range. The scope rules of the previous section yield an ambiguous result when several defining occurrences of the same identifier have the same range: the assignment in the last line of `machar.nl` could be setting either the integer or the floating point variable.

The ScopeGraphs module creates a new definition table key to represent the entity named by the first defining occurrence the attribute computation encounters in a specific range. (That defining occurrence may not be textually first; attribute computations are not necessarily carried out in textual order.) Any subsequent defining occurrence of the same identifier in that range will be represented by the same key.

For NameLan, we should report multiple defining occurrences bound to the same definition table key as an error. The ScopeGraphs module does not provide an error-reporting

role for this condition because there are a number of situations in which it is legal. However, a simple computation based on the `Unique` role provided by the Eli module library allows us to produce an appropriate report (see Section "Check for Unique Object Occurrences" in *Property Library*). Since we will need to report multiply-defined identifiers in several contexts, we wrap this computation in a role named `MultDefChk` that `VarDefName` and other symbols can inherit:

Report a multiply-defined identifier[21]==

```
SYMBOL MultDefChk INHERITS Unique COMPUTE
  IF(NOT(THIS.Unique),
    message(
      ERROR,
      CatStrInd("Identifier is multiply defined: ", THIS.Sym),
      0,
      COORDREF));
END;

SYMBOL VarDefName INHERITS MultDefChk END;
```

This macro is invoked in definition 22.

(See Section "Source Text Coordinates and Error Reporting" in *The Eli Library*, for an explanation of the `message` routine.)

Combining these computations into a single specification that we add to those of the previous section, and instantiating the `Unique` module, gives us:

Abstract syntax tree[22]==

```
Report an undefined identifier[20]
Report a multiply-defined identifier[21]
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Specification files[23]==

```
$/Prop/Unique.gnrc :inst
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Report > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Ask Eli to generate a processor and save the executable for that processor in the current directory:

   ```
   -> NameLan.specs :exe > rept
   ```

Apply `rept` to `machar.nl`.

a. Why is there no output?

b. Delete the variable declaration from the first line of `machar.nl` and again apply `rept` to it. Is the output what you expected?

c. Duplicate the variable declaration in the first line of `machar.nl` and again apply `rept` to it. Is the output what you expected?

1.5 Procedures

Suppose that we want to be able to define and invoke procedures in our simple language. Here's an example of a program that defines a procedure to compute the greatest common divisor (gcd) of two integers. It also defines a global variable, and sets that variable to the gcd of two specific values:

gcd.nl[24]==

```
int gcd(int x, int y) {
  while (x != y) {
    if (x > y) x = x - y;
    else y = y - x;
  }
  return x;
}

int x;

{ x = gcd(9, 12); }
```

This macro is attached to a non-product file.

Our extension of the phrase structure reflects object-oriented terminology because we will eventually make NameLan an object-oriented language:

Phrase structure[25]==

```
Declaration:    MethodDecl.

MethodDecl:     Type Ident MethodBody.
MethodBody:     '(' Parameters ')' '{' DeclStmt* '}'.
Parameters:     [ Parameter // ',' ].
Parameter:      Type Ident.

Type:           'void'.

Statement:      Name '(' Arguments ')' ';' / 'return' [ Expr ] ';'.

Factor:         Name '(' Arguments ')'.
Arguments:      [ Expr // ',' ].
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

We need abstract syntax symbols to distinguish the two new identifier contexts found in `Method.con` (see Section "Representation of identifiers" in *Name Analysis Reference Manual*).

Abstract syntax of identifiers[26]==

```
RULE: MethodDecl        ::= Type MethodDefName MethodBody END;
RULE: MethodDefName     ::= Ident                          END;

RULE: Parameter         ::= Type ParamDefName              END;
RULE: ParamDefName      ::= Ident                          END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

Our extension of NameLan introduces new defining occurrences, applied occurrences, and ranges. These changes require us to modify the scope rule for `VarDefName` from the kernel language, and provide additional rules:

- The scope of a defining occurrence `VarDefName` is the smallest enclosing `Block`, `MethodBody`, or `Program`.
- The scope of a defining occurrence `MethodDefName` is the smallest enclosing `Program`.
- The scope of a defining occurrence `ParamDefName` is the smallest enclosing `MethodBody`.

This is the first section in which the exercises will ask you to create a specification file on your own. You will need to add the names of such specification files to those of the files supplied by the tutorial. We support that by adding the name of the file `Solutions.specs` to the tutorial's list of specification files:

Specification files[27]==

```
Solutions.specs
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

`Solutions.specs` is not supplied by the tutorial, but by you. In the introdution to this manual, we suggested that you create a directory containing an empty file `Solutions.specs` before beginning the exercises. (If you decided to use a workbook, `Solutions.specs` should contain the name of the workbook file.)

`Solutions.specs` is entirely under your control, as is your workbook if you use one. When you need to create a specification to be used in generating a processor, simply add its name to your `Solutions.specs` or define it in your workbook.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Procs > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Consider the program `gcd.nl`.
   a. Describe the four ranges. Where does each begin and end? What are the enclosure relations?

   b. Identify the four defining occurrences. To which range does each belong?
   c. Identify one applied occurrence that has a defining occurrence in the smallest range containing that applied occurrence.
   d. Identify one applied occurrence that does not have a defining occurrence in the smallest range containing that applied occurrence,

2. Write LIDO rules that use inheritance to connect the grammar in `NameLan.lido` to the scope rules for the language. Will any symbols need to inherit `MultDefChk`? Explain briefly.
3. The LIDO rules that connect the grammar in `NameLan.lido` to the scope rules for the language constitute the first specification that you have created on your own. You need to combine those rules with the specification developed in the tutorial to generate a processor that analyzes NameLan procedures.

   If you are using a workbook, define a FunnelWeb output file named `Method.lido` containing your LIDO rules in that workbook (see Section "Output Files" in *FunnelWeb*).

   If you are not using a workbook, create a file named `Method.lido` containing your LIDO rules and add the name `Method.lido` to your `Solutions.specs` file.

   a. Could you have named your specification file `Procs.lido`? Explain briefly.
   b. Could you have named your specification file `Method.con`? Explain briefly.
4. Use `Bindings.specs` to generate a processor named `p1` that will show bindings for applied occurrences, and apply it to `gcd.nl`. Do you get the output you expected? Explain briefly.
5. Draw the scope graph that was built by `p1` for `gcd.nl`.
6. Apply `p1` to the file `lcl.nl`, which declares a local variable `x` in the body of `gcd`:

   lcl.nl[28]==

   ```
   int gcd(int x, int y) {
     while (x != y) {
       if (x > y) x = x - y;
       else y = y - x;
     }
     return x;
     int x;
   }

   int x;

   { x = gcd(9, 12); }
   ```

   This macro is attached to a non-product file.

   What is the effect of the local variable declaration? Briefly explain the output.
7. Open the file `NameLan.con` in your editor and change the definition of a `MethodBody` to:

   ```
   MethodBody:     '(' Parameters ')' Block.
   ```

   Use `Bindings.specs` to generate another processor named `p2`.

   a. Apply `p2` to `gcd.nl`. Briefly explain the output.

b. Apply `p2` to `lcl.nl`. Briefly explain why `p1` and `p2` give different results.

c. The difference between `p1` and `p2` represents a language design choice. Briefly evaluate the two possibilities and state your reasons for preferring one over the other.

2 Classes

A class is an entity that can encapsulate both storage and behavior. File `random.nl` contains an example:

random.nl[29]==

```
class Random {
  int state = 100001;

  float ran() {
    state = state * 125;
    state = state - (state / 2796203) * 2796203;
    return state / 2796203.0;
  }
}

{ float p, q;
  p = Random.ran();
  q = Random.ran();
}
```

This macro is attached to a non-product file.

The behavior encapsulated by the class `Random` is the pseudo-random number generator `ran`, and the storage `state` holds the internal state of that generator.

The general form of a NameLan class declaration is:

Phrase structure[30]==

```
Declaration:    ClassDecl.

ClassDecl:      'class' Ident Inheritance ClassBody.
Inheritance:    Default.
Default:                .
ClassBody:      '{' Declaration* '}'.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

We need an abstract syntax symbol to distinguish the new identifier context found in the `ClassDecl` (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). That identifier is the class name, so `ClassDefName` is a reasonable symbol for the context:

Abstract syntax of identifiers[31]==

```
RULE: ClassDecl    ::= 'class' ClassDefName Inheritance ClassBody END;
RULE: ClassDefName ::= Ident                                      END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

Here are our new and modified scope rules for NameLan with classes:

- The scope of a defining occurrence `VarDefName` is the smallest enclosing `Block`, `MethodBody`, `ClassBody`, or `Program`.

- The scope of a defining occurrence `MethodDefName` is the smallest enclosing `ClassBody` or `Program`.
- The scope of a defining occurrence `ClassDefName` is the smallest enclosing `ClassBody` or `Program`.

The LIDO computations associated with these scope rules are similar to the ones associated with methods, and we leave them to the reader as an exercise.

Classes introduce two new facilities, both of which affect name analysis:

- *Qualified names* may be used to access the class members.
- Classes may *inherit* members from other classes.

We will discuss these aspects in the next two sections.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Class > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Consider the LIDO specifications necessary to implement the scope rules for classes.
   a. Briefly explain why these scope rules do not require any changes in earlier specifications.
   b. Create a specification that uses LIDO inheritance to implement name analysis for classes, and ensure that the file containing it is named in `Solutions.specs`.
   c. Use `Bindings.specs` to generate a processor named `cl` that will show bindings for applied occurrences. Apply `cl` to `machar.nl` and `gcd.nl`. Do you get the output you expected? Explain briefly.
2. Apply `cl` to `random.nl`. A syntax error is reported because `cl` does not understand qualified names, but the name analysis continues with the dot removed from the input text. The two applied occurrences `Random` and `ran` remain, and are considered individually.
   a. Is the treatment of each applied occurrence correct? Explain briefly.
   b. Can you conclude that your specification is correct from the tests that you have run? Explain briefly.

2.1 Qualified names

The defining occurrences `state` and `ran` in `random.nl` have the `ClassBody` of `Random` as their scope. We introduce qualified names in order to be able to access such entities from outside of the scope of their defining occurrences. Here is a specification of the phrase structure of a qualified name:

Phrase structure[32]==

```
Name: Name '.' Ident.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55,

67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

The `Name` preceding the dot in a qualified name is called the *qualifier*, and the `Ident` is an applied occurrence. We need an abstract syntax symbol to distinguish this new identifier context (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). `QualifiedId` is a reasonable choice:

Abstract syntax of identifiers[33]==

```
RULE: Name              ::= Name '.' QualifiedId          END;
RULE: QualifiedId       ::= Ident                         END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

In order to formalize the intuitive meaning of a qualified name, we need one concept in addition to the four discussed earlier (see Section "Basic Scope Rules" in *Name analysis according to scope rules*).

- An entity may *own* a set of bindings called its *members*.

  Language rules specify the set of members of an entity.

As language designers, we provide the following rules to define the members of a class and the meaning of a qualified name:

- The members of a `Class` are the bindings having the `ClassBody` as a scope.
- An applied occurrence that is a `QualifiedId` '`i`' in a `Name` '`q.i`' identifies the binding '`(i,k)`' that is a member of the qualifier '`q`'.

Here is an analysis of the qualified name `Random.ran` of `random.nl`:

1. The applied occurrence `Random` names a class entity '`e`'.
2. The members of '`e`' are the bindings having the `ClassBody` on lines 2-8 as a scope.
3. The binding '`(ran,k)`' is a member of '`e`'.
4. The applied occurrence `ran` in `Random.ran` identifies '`(ran,k)`', and hence the name `Random.ran` names the pseudo-random number generator.

The ownership relation between a class and its members is established by overriding the default computation for the `ClassBody.ScopeKey` attribute (see Section "The RangeScope role" in *Name Analysis Reference Manual*). Recall that `ClassDefName.Key` represents the class entity:

Establish the ownership relation[34]==

```
RULE: ClassDecl ::= 'class' ClassDefName Inheritance ClassBody COMPUTE
  ClassBody.ScopeKey = ClassDefName.Key;
END;
```

This macro is invoked in definition 38.

A qualified identifier plays the `GCQualName` role (see Section "Graph-complete search" in *Name Analysis Reference Manual*). That role provides three attributes used in name analysis:

`ScopeKey` A `DefTableKey`-valued attribute that should be set by user computation to the key of the entity named by the qualifier.

`Scope` A `NodeTuplePtr`-valued attribute that will be set by a module computation to the set of bindings owned by the entity given by the `ScopeKey` attribute (see Section "Obtain a range from a qualifier" in *Name Analysis Reference Manual*). If that entity does not own bindings, the computation will set the `Scope` attribute to the value `NoNodeTuple` (see Section "Establishing the Structure of a Scope Graph" in *Name Analysis Reference Manual*).

`Key` A `DefTableKey`-valued attribute that will be set by the following module computation: Let '`i`' be the qualified identifier. If the binding '`(i,k)`' is an element of the `Scope` set, then set `Key` to '`k`'. Otherwise set `Key` to `NoKey`.

In some contexts, the set of bindings owned by a qualifier may not be known until some other computation has taken place (for an example, see Section 4.2 [Type-qualified names], page 47). This means that we need to establish a precondition, `ContextIsReady`, for setting the `ScopeKey` attribute. That precondition is true by default:

Qualified names lookup in complete graphs[35]==

```
SYMBOL Name COMPUTE INH.ContextIsReady += "yes"; END;
```

This macro is defined in definitions 35 and 36.
This macro is invoked in definition 38.

By using an accumulating computation for the void attribute `ContextIsReady`, we can allow any of several different computations to provide the precondition (see Section "Accumulating Computations" in *LIDO – Reference Manual*).

If we use an attribute `Name.Key` to represent the entity named by a `Name`, then that attribute can be computed by a left-to-right (bottom-up) induction:

Qualified names lookup in complete graphs[36]==

```
SYMBOL QualifiedId INHERITS GCQualName, ChkIdUse END;

RULE: Name ::= Name '.' QualifiedId COMPUTE
  QualifiedId.ScopeKey = Name[2].Key <- Name[1].ContextIsReady;
  Name[1].Key = QualifiedId.Key;
END;

RULE: Name ::= SimpleName COMPUTE
  Name.Key = SimpleName.Key;
END;
```

This macro is defined in definitions 35 and 36.
This macro is invoked in definition 38.

At this point the abstract syntax contains four `Name` contexts in addition to the contexts in the qualified name rules:

```
Statement ::= Name '(' Arguments ')' ';'
Expr      ::= Name '(' Arguments ')'
Statement ::= Name '=' Expr ';'
Expr      ::= Name
```

In the first two contexts, `Name` is the name of a method, and in the last two it is the name of a variable. It is likely that method names and variable names will need different attributes, and that those attributes will differ from the attributes needed for qualified names. Since

attributes are associated with symbols of the abstract syntax, it would be useful to add two new symbols, `MethName` and `ExprName`, to the abstract syntax. The technique is identical to the one we have been using to provide abstract syntax symbols to distinguish identifier contexts (see Section "Basic Scope Rules" in *Name analysis according to scope rules*). We replace `Name` in each of the four rules with the desired symbol and add rules defining each of the new symbols as a `Name`:

Make contexts of complete names explicit[37]==

```
RULE: Statement ::= MethName '(' Arguments ')' ';'      END;
RULE: Expr      ::= MethName '(' Arguments ')'          END;
RULE: Statement ::= ExprName '=' Expr ';'               END;
RULE: Expr      ::= ExprName                            END;

RULE: MethName  ::= Name                                END;
RULE: ExprName  ::= Name                                END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

We need to attach all of these rules to the abstract syntax tree:

Abstract syntax tree[38]==

```
Establish the ownership relation[34]
Qualified names lookup in complete graphs[35]
Make contexts of complete names explicit[37]
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Qual > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Use `Bindings.specs` to generate a processor named `qual` that will show bindings for applied occurrences.
   a. Apply `qual` to `random.nl`, `machar.nl`, and `gcd.nl`. Do you get the output you expected? Explain briefly.
   b. Explain briefly why `QualifiedId` inherits the role `ChkIdUse`.
2. In `random.nl`, change the two assignments of the program body to:

   ```
   p = p.ran();
   q = Random.ra();
   ```

   a. Apply `qual` to the modified file. Did you get the result you expected?
   b. The two error reports produced by `qual` when it is applied to the modified `random.nl` are identical except for the undefined identifier. Do you think that the two errors should be reported in the same way? Explain briefly.

c. How could you use the value of `QualifiedId.Scope` to provide different reports for the two errors in the modified `random.nl`? (See Section "Error Reporting" in *The Eli Library*, for information on constructing error reports.)

2.2 Inheritance

The program `gambler.nl` implements a coin tossing class `Coin` and a class `Dice` that throws an arbitrary number of dice. It uses inheritance to provide each class with its own random number generator, without duplicating code:

gambler.nl[39]==

```
class Random {
  int state = 100001;

  float ran() {
    state = state * 125;
    state = state - (state / 2796203) * 2796203;
    return state / 2796203.0;
  }
}

class Coin extends Random {
  int state = 0;

  int toss() {
    state = 2 * ran();
    return state;
  }
}

class Dice extends Random {
  int state = 1;

  int throw(int n) {
    state = n;
    return (6 * n) * ran() + 1;
  }
}

{ int n; float p;
  n = Coin.toss();
  n = Dice.throw(5);
  n = Coin.state;
  p = Coin.ran();
  n = Dice.state;
  p = Dice.ran();
}
```

This macro is attached to a non-product file.

Both `Coin` and `Dice` use `extends` clauses to name `Random` as their *direct superclass* from which they inherit. We use the following scope rule to define the effect of inheritance:

- Let '`C`' be a class which inherits from its direct superclass '`S`'. Any binding having a scope that is the `ClassBody` of '`S`' also has a scope that is the `ClassBody` of '`C`', unless a defining occurrence of that binding's identifier has the `ClassBody` of '`C`' as its scope.

In `gambler.nl`, `Coin` has `Random` as its direct superclass. A binding for `ran` has the `ClassBody` of `Random` as its scope, and no defining occurrence of `ran` has the `ClassBody` of `Coin` as its scope. Therefore the binding for `ran` that has the `ClassBody` of `Random` as its scope also has the `ClassBody` of `Coin` as its scope. This means that the applied occurrence of `ran` in the second line of `toss` names the `ran` method of class `Random`.

In contrast, although a binding for `state` has the `ClassBody` of `Random` as its scope, there *is* a defining occurrence of `state` having the `ClassBody` of `Coin` as its scope. Therefore the applied occurrence of `state` in the second line of `toss` names the `state` variable of class `Coin`.

Similar reasoning applies to class `Dice`: `throw` invokes the `ran` method of class `Random` and assigns a value to the `state` variable of class `Dice`.

It is important to understand that both `Coin` and `Dice` actually have *two* integer variables. One of these variables is inherited from `Random`, and the other is defined locally. The local definition has merely hidden the inherited integer variable in the body of the inheriting class. Thus the programmer cannot access it directly within the inheriting class. The `ran` method, however, accesses the `state` variable inherited from `Random` because that method is defined in `Random`.

An inheritance relation is modeled in the scope graph by a *path edge* that is directed from the node for a class to the node for its direct superclass (see Section "Scope graphs" in *Name Analysis Reference Manual*). The tip of that path edge is given by a name, and lookup operations are required to determine the node bound to that name. The lookup of an edge tip name may depend on the existence of path edges in the scope graph, and it contributes a path edge to the scope graph. In `edges.nl`, for example, the lookup for the edge tip `C.D` contributes a path edge from node `B` to node `D` and depends on the existence of the edge from node `C` to node `A`:

edges.nl[40]==

```
class X {
  int k;

  class A {
    class D { int k; }
  }

  class B extends C.D {
    int m() {
      return k;
    }
  }

  class C extends A { }
```

```
}

{ }
```

This macro is attached to a non-product file.

Our previous analysis of applied occurrences was based on the roles `GCSimpleName` and `GCQualName`, which depend on a complete scope graph (see Section "Graph-complete search" in *Name Analysis Reference Manual*). Clearly those roles will not be satisfactory for analyzing the applied occurrences in a superclass name. The scope graph module provides two analogous roles, `WLSimpleName` and `WLQualName`, that solve potential ordering conflicts by using a worklist algorithm (see Section "Worklist search" in *Name Analysis Reference Manual*). In order to take advantage of these roles, we define the phrase structure using symbols that are different from those inheriting roles `GCSimpleName` and `GCQualName`:

Phrase structure[41]==

```
Inheritance:    'extends' WLName.

WLName:         Ident.
WLName:         WLName '.' Ident.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

We need abstract syntax symbols to distinguish the two new identifier contexts found in this rule (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). These contexts are analogous to `SimpleName` and `QualifiedId`, so we just add `WL` to distinguish them:

Abstract syntax of identifiers[42]==

```
RULE: WLName            ::= SimpleWLName              END;
RULE: SimpleWLName      ::= Ident                     END;

RULE: WLName            ::= WLName '.' QualifiedWLId  END;
RULE: QualifiedWLId     ::= Ident                     END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

The attribute computations for `WLName` are very similar to those for `Name`. They also implement a left-to-right induction, but use `FPItemPtr` values instead of `DefTableKey` values. An `FPItemPtr` value represents a worklist computation, whereas a `DefTableKey` value represents the result of such a computation. `FPItemPtr` values can be computed before the worklist computation takes place because they specify what that computation is to do, rather than the result it will obtain:

Specify worklist computations[43]==

```
ATTR FPItem: FPItemPtr;

SYMBOL SimpleWLName INHERITS WLSimpleName, ChkIdUse END;
SYMBOL QualifiedWLId INHERITS WLQualName, ChkIdUse END;

RULE: WLName ::= SimpleWLName COMPUTE
```

```
  WLName.FPItem = SimpleWLName.FPItem;
END;

RULE: WLName ::= WLName '.' QualifiedWLId COMPUTE
  QualifiedWLId.DependsOn = WLName[2].FPItem;
  WLName[1].FPItem = QualifiedWLId.FPItem;
END;
```

This macro is invoked in definition 47.

`WLName` appears in one abstract syntax rule beyond the two above:

```
RULE: Inheritance ::= 'extends' WLName END;
```

As with `Name` in the last section, it is likely that the attributes needed for `WLName` in this context will differ from those needed to handle qualified names. Therefore we provide a new abstract syntax symbol, `SuperClass`, to distinguish this context:

Make contexts of complete names explicit[44]==

```
RULE: Inheritance ::= 'extends' SuperClass      END;
RULE: SuperClass  ::= WLName                    END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

Each `WLName` represents a distinct source language entity, and therefore it must have a `Key` attribute:

Specify the Key attribute of a WLName[45]==

```
RULE: WLName ::= SimpleWLName COMPUTE
  WLName.Key = SimpleWLName.Key;
END;

RULE: WLName ::= WLName '.' QualifiedWLId COMPUTE
  WLName[1].Key = QualifiedWLId.Key;
END;
```

This macro is invoked in definition 47.

A path edge is established by the role `WLCreateEdge` (see Section "Worklist search" in *Name Analysis Reference Manual*). The `tailEnv` attribute of the node inheriting `WLCreateEdge` must be set to the `NodeTuplePtr` value representing the scope graph node that is the tail of the edge. The `tipFPItem` attribute of the node inheriting `WLCreateEdge` must be set to an `FPItem` value describing the computation of the tip of the edge. In our grammar, `SuperClass` provides an appropriate context for `WLCreateEdge`:

Establish a path edge to a superclass[46]==

```
SYMBOL SuperClass INHERITS WLCreateEdge END;

RULE: SuperClass ::= WLName COMPUTE
  SuperClass.tailEnv = INCLUDING Inheritance.SubClassEnv;
  SuperClass.tipFPItem = WLName.FPItem;
END;
```

This macro is invoked in definition 47.

This rule obtains the computation for the tip value from its child, but reaches up the tree for the tail value. We leave the computation of `Inheritance.SubClassEnv` to the reader as an exercise.

The NameLan inheritance scope rule says that an inherited binding hides bindings from enclosing ranges. That means that the generic search algorithm must deal with inheritance before it moves to an enclosing range (see Section "The generic lookup" in *Name Analysis Reference Manual*).

The two fragments discussed in this section form the basis for implementing NameLan inheritance:

Abstract syntax tree[47]==

Specify worklist computations[43]
Specify the Key attribute of a WLName[45]
Establish a path edge to a superclass[46]

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Path > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. We have used the attribute `Inheritance.SubClassEnv` in the super class computations, but not specified how that attribute is set. Complete our specification, and ensure that the file containing your material is named in `Solutions.specs`. It must define the attribute `Inheritance.SubClassEnv`, and set its value to the `NodeTuplePtr` value representing the range of the subclass (see Section "The RangeScope role" in *Name Analysis Reference Manual*).
2. Use the scope rule for inheritance to decide the meaning of the applied occurrence of `k` in the `return` statement of `edges.nl`.
3. Use `Bindings.specs` to generate a processor named `path` that will show bindings for applied occurrences. Apply `path` to `gambler.nl` and `edges.nl`. Do you get the output you expected? Explain briefly.
4. In `edges.nl`, change the symbol `A` in the `extends` clause for class `C` to `k`. Apply `path` to the modified file. Do you get the output you expected? Explain briefly.
5. Consider the program `wlblock.nl`:

   wlblock.nl[48]==

   ```
   class CC { }

   class C1 extends C2.BB {
     class AA extends CC { }
   }
   ```

```
class C2 extends C1.AA {
  class BB extends CC { }
}

{ }
```

This macro is attached to a non-product file.

a. Draw a scope graph showing the path edges for the superclass relations in `wlblock.nl`.

b. Simulate the worklist computation for `wlblock.nl`. Do you see any problems?

c. Apply `path` to `wlblock.nl`. Did you get the result you expected?

3 Libraries

It's rare that a program is developed by one person, without reference to the work of others. Usually a programmer relies on a number of *libraries* for things like common computations (e.g. `sqrt`) and input/output operations. Name analysis must have access to information from libraries in order to establish bindings, but library code resides in separate files that are not physically part of the user's program.

There are many strategies for linking with library files, but all must support the basic name analysis concepts. For the purposes of this tutorial, we will assume that the processor input is a concatenation of source files. That concatenation will begin with the program file, followed by any number of library files. Eli provides mechanisms for collecting source files into a single input, but these are beyond the scope of this tutorial (see Section "Insert a File into the Input Stream" in *Tasks related to input processing*).

Without use of additional facilities, a generated processor named `proc` could be invoked on a sequence of files as follows:

```
cat pgmfile libfile libfile | ./proc
```

In the remainder of this tutorial, any examples illustrating libraries will be single files that are the result of a concatenation.

We will extend NameLan by defining each library file as an entity called a *package*. File 'pkg.nl' illustrates this extension:

pkg.nl[49]==

```
{ float rate;
  rate = verbs.fly.distance / insects.fly.legs;
}

package insects;
class ant { }
class fly { int legs; }
class bee { }

package verbs;
class run { }
class fly { int distance; }
```

This macro is attached to a non-product file.

Our original grammar's root symbol was `Program`, which now describes only the first part of a larger construct. To describe this larger construct, we need to extend the NameLan concrete grammar above `Program`. In the process, we create a new root called `Collection`:

Phrase structure[50]==

```
Collection:     Pgmfile Libfile*.
Pgmfile:        Program.

Libfile:        'package' Ident ';' PackageBody.
PackageBody:    Declaration*.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.

This macro is invoked in definition 3.

We need a new abstract syntax symbol to distinguish the new context for identifiers (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). The obvious choice is `PackageDefName`:

Abstract syntax of identifiers[51]==

```
RULE: Libfile        ::= 'package' PackageDefName ';' PackageBody END;
RULE: PackageDefName ::= Ident                                    END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

Here are our new and modified scope rules for NameLan with packages:

- The scope of a defining occurrence `VarDefName` is the smallest enclosing `Block`, `MethodBody`, `ClassBody`, `Program`, or `PackageBody`.
- The scope of a defining occurrence `MethodDefName` is the smallest enclosing `ClassBody`, `Program`, or `PackageBody`.
- The scope of a defining occurrence `ClassDefName` is the smallest enclosing `ClassBody`, `Program`, or `PackageBody`.
- The scope of a defining occurrence `PackageDefName` is the enclosing `Collection`.
- The members of a `Package` are the bindings having the `PackageBody` as a scope.

Since `Program` is no longer the grammar root, it does not automatically inherit any role and must explicitly inherit `RangeScope`:

Abstract syntax tree[52]==

```
SYMBOL Program INHERITS RangeScope END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,
65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The LIDO implementation of the remaining roles is left as an exercise.

The ownership relation between a package and its members is established by overriding the default computation for the `PackageBody.ScopeKey` attribute (see Section "The RangeScope role" in *Name Analysis Reference Manual*). Recall that `PackageDefName.Key` represents the package entity:

Abstract syntax tree[53]==

```
RULE: Libfile ::= 'package' PackageDefName ';' PackageBody COMPUTE
  Libfile.Key = PackageDefName.Key;
END;

SYMBOL PackageBody COMPUTE INH.ScopeKey = INCLUDING Libfile.Key; END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,
65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Pack > .
```

None of these files will have write permission in your current directory.

1. Consider the LIDO specifications necessary to implement the scope rules for NameLan with packages. Briefly explain why these scope rules do not require any changes in earlier specifications.
2. Create a specification that uses LIDO inheritance to establish the roles implied by the scope rules for `PackageDefName` and `PackageBody`, and ensure that the file containing it is named in `Solutions.specs`.
3. Use `Bindings.specs` to generate a processor named `pk` that will show bindings for applied occurrences. Apply `pk` to `gambler.nl` and `pkg.nl`. Do you get the output you expected? Explain briefly.

3.1 Single import

A *single import* construct makes it possible to refer to a single entity defined in some package by a simple name rather than a qualified name. For example, consider file `single.nl`:

single.nl[54]==

```
import P.D.m;
import Q.C.a;
{ a = 42; m(); }

package Q;
import P.D;
class C extends D {
  int a;
  void m() {
    a = b;
  }
}

package P;
class D {
  int b;
  void m() {
    b = Q.C.a;
  }
}
```

This macro is attached to a non-product file.

Single imports are used here to allow the program to refer to variable `a` of class `Q.C` and method `m` of class `P.D` by simple names, and to allow package `Q` to refer to the class `P.D` by a simple name.

In order to implement single imports, we need to think about how a single import interacts with the entities defined in the program or package. The nature of these interactions is specified by the language designer, and there are several possibilities. We will describe the decision that we made for NameLan, and show how that decision leads to a phrase structure. Other decisions would lead to other structures.

As language designers, we consider imports to be a convenience for the programmer that should not have any effect beyond the obvious one illustrated by `single.nl`. We therefore do not want the scope of an imported binding to be the `Program` or a `PackageBody`. For example, if the scope of the imported binding for `D` were the `PackageBody` of `Q` then that binding would be available for another package to import from `Q`. That is an effect beyond the obvious one.

If the scope of an imported binding were a phrase that encompassed the program or package body, then that binding could be referred to by a simple name within the program or package body unless it was hidden by a local defining occurrence. Although the `PgmFile` (resp. `LibFile`) encompasses the program (resp. package body), those phrases also encompass the import declarations. If we chose those phrases as scopes for the imported bindings, import declarations could interfere with one another. That is another effect beyond the obvious one.

We can avoid these undesired effects if we establish a structure for a program containing import declarations that has an additional phrase encompassing the file body, but not the import declarations. The desired structure can be defined by adding six productions to the NameLan grammar:

Phrase structure[55]==

```
Pgmfile:        ImportDecls PgmFileBody.
PgmFileBody:    Program.

Libfile:        'package' Ident ';' ImportDecls LibFileBody.
LibFileBody:    PackageBody.

ImportDecls:    ImportDecl+.
ImportDecl:     'import' WLName ';'.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

The first and second productions introduce a new phrase `PgmFileBody` that encompasses the normal `Program` phrase, and ensure that `PgmFileBody` is derived only if the `PgmFile` contains import declarations. If a `PgmFile` contains no import declarations, that `PgmFile` is derived directly to `Program` by the production defined earlier. The third and fourth productions define the corresponding structure for a `Libfile`.

To see how this grammar addition has the desired effect, consider `single.nl`. `Pgmfile` contains import declarations, and therefore the grammar derives a `PgmFileBody` phrase encompassing the file's `Program` phrase. Similarly, the first `Libfile` contains an import declaration and a `LibFileBody` phrase encompassing its `PackageBody` phrase is derived. On the other hand, the second `Libfile` has no import declarations. This means that none of the added rules applies, and thus there is no additional phrase.

Notice that we have chosen to derive the name of the imported entity from `WLName`. Recall that this phrase is used when there is a potential ordering conflict in analyzing names (see Section 2.2 [Inheritance], page 24). In `single.nl`, the path edge between class `C` and class `D` modeling the inheritance relation depends on the operand `P.D` of the import declaration in package `Q`. Since path edges might depend on single imports, the imported name must be processed by the worklist algorithm. We will use the rather generic "`ImportName`" to characterize the imported entity because any declared entity might be imported:

Make contexts of complete names explicit[56]==

```
RULE: ImportDecl ::= 'import' ImportName ';'    END;
RULE: ImportName ::= WLName                     END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

There is also a new context for `PackageDefName`:

Make contexts of complete names explicit[57]==

```
RULE: Libfile    ::=
        'package' PackageDefName ';' ImportDecls LibFileBody
COMPUTE
  Libfile.Key = PackageDefName.Key;
END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

Here are the corresponding scope rules:

- Suppose that an `ImportDecl` in the `Pgmfile` encloses an `ImportName` identifying a binding '`(i,k)`'. The binding '`(i,k)`' also has the `PgmFileBody` as a scope.
- Suppose that an `ImportDecl` in a `Libfile` encloses an `ImportName` identifying a binding '`(i,k)`'. The binding '`(i,k)`' also has the `LibFileBody` as a scope.

These scope rules imply the following roles:

Inherit the appropriate roles[58]==

```
SYMBOL PgmFileBody INHERITS RangeScope END;
SYMBOL LibFileBody INHERITS RangeScope END;
```

This macro is defined in definitions 58 and 59.
This macro is invoked in definition 65.

An `ImportName` is an applied occurrence that establishes an additional scope for the binding it identifies. The `WLInsertDef` role supports this behavior (see Section "Worklist search" in *Name Analysis Reference Manual*).

Inherit the appropriate roles[59]==

```
SYMBOL ImportName INHERITS WLInsertDef END;
```

This macro is defined in definitions 58 and 59.
This macro is invoked in definition 65.

`WLInsertDef` provides the standard attributes of an applied occurrence:

`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 Section 1.2 [Tree Grammar Preconditions], page 3).

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

`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.

`WLInsertDef` adds the following attributes to those for the standard applied occurrence:

`DependsOn` is an `FPItemPtr`-valued attribute that represents the computation yielding the existing binding. This attribute must be set by a developer computation.

`Scope` is a `NodeTuplePtr`-valued attribute representing the additional scope for the identified binding. This inherited attribute is set by a module computation to `INCLUDING AnyScope.Env`.

We must write computations to set the values of four of these five `WLInsertDef` attributes in the context of an `ImportName`. The first three are synthesized attributes, the fourth is inherited:

Set WLInsertDef attributes[60]==

```
RULE: ImportName ::= WLName COMPUTE
  ImportName.Sym = WLName.Sym;
  ImportName.UseKey = WLName.UseKey;
  ImportName.DependsOn = WLName.FPItem;
END;

SYMBOL ImportName COMPUTE
  INH.Scope =
    INCLUDING (Pgmfile.ImportEnv, Libfile.ImportEnv);
END;
```

This macro is defined in definitions 60, 61, 62, and 63.
This macro is invoked in definition 65.

The `Sym` and `UseKey` values must be propagated to `WLName` from its component identifiers; this was done earlier for `WLName.FPItem` (see Section 2.2 [Inheritance], page 24).

Set WLInsertDef attributes[61]==

```
SYMBOL WLName: UseKey: DefTableKey;

RULE: WLName ::= SimpleWLName COMPUTE
  WLName.Sym = SimpleWLName.Sym;
  WLName.UseKey = SimpleWLName.UseKey;
END;

RULE: WLName ::= WLName '.' QualifiedWLId COMPUTE
  WLName[1].Sym = QualifiedWLId.Sym;
  WLName[1].UseKey = QualifiedWLId.UseKey;
```

```
END;
```

This macro is defined in definitions 60, 61, 62, and 63.
This macro is invoked in definition 65.

In order to obtain a value for `ImportName.Scope` in every context, we need to establish values for `Pgmfile.ImportEnv` and `Libfile.ImportEnv`. The simplest case is the one in which there are no import declarations. If there are no import declarations there will be no `ImportName` nodes, and therefore the value of `Pgmfile.ImportEnv` or `Libfile.ImportEnv` is not used and should represent no range:

Set WLInsertDef attributes[62]==

```
ATTR ImportEnv: NodeTuplePtr;

RULE: Pgmfile ::= Program COMPUTE
  Pgmfile.ImportEnv = NoNodeTuple;
END;

RULE: Libfile ::= 'package' PackageDefName ';' PackageBody
COMPUTE
  Libfile.ImportEnv = NoNodeTuple;
END;
```

This macro is defined in definitions 60, 61, 62, and 63.
This macro is invoked in definition 65.

If import declarations *are* present, the additional scope for those bindings is the associated `PgmFileBody` or `LibFileBody` phrase:

Set WLInsertDef attributes[63]==

```
RULE: Pgmfile ::= ImportDecls PgmFileBody COMPUTE
  Pgmfile.ImportEnv = PgmFileBody.Env;
END;

RULE: Libfile ::=
        'package' PackageDefName ';' ImportDecls LibFileBody
COMPUTE
  Libfile.ImportEnv = LibFileBody.Env;
END;
```

This macro is defined in definitions 60, 61, 62, and 63.
This macro is invoked in definition 65.

The intent of the statement `import 'q.i';` is to make it possible to refer to the entity named by '`q.i`' by the simple name '`i`'. Suppose that a user added the following line at the beginning of `single.nl`:

```
import Q.C.m;
```

The result is a contradiction, allowing two distinct entities to have the same simple name `m`, and an error must be reported (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*). When a single import collides with another one, the value of its `ImportName.Key` attribute differs from the value of the `WLName.Key` attribute (see Section "Worklist search" in *Name Analysis Reference Manual*). Again, `WLName.Key` is the `Key` attribute of the rightmost applied occurrence:

Report a collision error in a single import[64]==

```
RULE: ImportName ::= WLName COMPUTE
  IF(NE(ImportName.Key, WLName.Key),
    message(
      ERROR,
      CatStrInd("Ambiguous import: ", WLName.Sym),
      0,
      COORDREF));
END;
```

This macro is invoked in definition 65.

Attaching the LIDO specifications to the abstract syntax tree:

Abstract syntax tree[65]==

```
Inherit the appropriate roles[58]
Set WLInsertDef attributes[60]
Report a collision error in a single import[64]
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Single > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Draw the scope graph for `single.nl`, and label the path edges. Suppose that the name analysis were to be done strictly from left to right, without using a worklist algorithm. Briefly explain why this analysis would fail.
2. Use `Bindings.specs` to generate a processor named `pk` that will show bindings for applied occurrences.
   a. Apply `pk` to `single.nl` and verify that the result is correct.
   b. Change the body of the program in `single.nl` to:

      ```
      { float a; a = 42; m(); }
      ```

      Apply `pk` to the modified file. Explain why `float a` hides the import rather than colliding with it (for a hint, see Section "Basic Scope Rules" in *Name analysis according to scope rules*).
3. Modify `single.nl` by changing the name of the class `C` in the package `Q` to `E`. Apply `pk` to the modified file. Is the output what you expected? Do you think that the error report could be improved? Explain briefly.

3.2 Import on demand

An *import on demand* is used to make it possible to refer to every entity declared in a package (or a class) by a simple name rather a qualified name. Here is an example:

demand.nl[66]==

```
import insects.*;
{ float rate;
  rate = verbs.fly.distance / fly.legs;
}

package insects;
class ant { }
class fly { int legs; }
class bee { }

package verbs;
class run { }
class fly { int distance; }
```

This macro is attached to a non-product file.

The import declaration in `demand.nl` makes all of the members of the package `insects` (`ant`, `fly`, and `bee`) accessible via simple names in the program code. The user takes advantage of this effect in the expression's denominator to use a simple name for the `fly` class (compare `demand.nl` with `pkg.nl`).

Here is the addition to the NameLan concrete syntax that supports import on demand:

Phrase structure[67]==

```
ImportDecl: 'import' WLName '.' '*' ';'.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

We shall see that this declaration adds a path edge to the scope graph, and therefore its applied occurrence must be sought using the worklist algorithm.

A new abstract syntax symbol is needed to represent the occurrence of a `WLName` in the context of an `ImportDecl`. Since the operand of `import` might refer to either a package or a class, we'll use `PCName`:

Make contexts of complete names explicit[68]==

```
RULE: ImportDecl ::= 'import' PCName '.' '*' ';' END;
RULE: PCName     ::= WLName                      END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

Our scope rules governing NameLan's import on demand construct are:

- Suppose that an ImportDecl in the `Pgmfile` encloses a `PCName` '`p`'. Any binding that is a member of '`p`' also has the `PgmFileBody` as a scope, unless a single import of that binding's identifier has the `PgmFileBody` as a scope.
- Suppose that an ImportDecl in a `Libfile` encloses a `PCName` '`p`'. Any binding that is a member of '`p`' also has the `LibFileBody` as a scope, unless a single import of that binding's identifier has the `LibFileBody` as a scope.
- No binding having the `PgmFileBody` or a `LibFileBody` as a scope hides any other binding.

The intent of the statement `import 'q.*';` is to make it possible to refer to some of the members of '`q`' by simple names rather than qualified names. Suppose that the user added a statement `import verbs.*;` to `demand.nl`. In that case, both `insects.fly` and `verbs.fly` could be referred to by the simple name `fly` and that reference would be ambiguous. No error should be reported, unless that simple name is actually used (as it is in the third line of `demand.nl`).

Each import on demand can be modeled by a path edge whose tail is the scope graph node created for the `PgmFileBody` or `LibFileBody` in which the import on demand occurs, and whose tip is the scope graph node owned by the entity being imported (see Section "Scope graphs" in *Name Analysis Reference Manual*). The generic lookup algorithm will then implement the import rule by following that path edge (see Section "The generic lookup" in *Name Analysis Reference Manual*).

The semantics of a path edge modeling an import on demand differ from those of a path edge modeling inheritance. We therefore need to use a second edge label for import edges (see Section "Scope graphs" in *Name Analysis Reference Manual*). Eli allows us to use the default edge label 1 without comment, and we have taken advantage of this avoid specifying edge labels. The result is that Eli has silently used the label 1 to indicate an inheritance edge. We will use the label 2 to indicate an import edge.

Adding an import path edge to the scope graph is similar to adding an inheritance path edge (see Section 2.2 [Inheritance], page 24). The only real difference is that we must specify the `label` attribute because it is not the default value 1:

Abstract syntax tree[69]==

```
SYMBOL PCName INHERITS WLCreateEdge COMPUTE
  SYNT.tailEnv = INCLUDING AnyScope.Env;
  SYNT.label = 2;
END;

RULE: PCName ::= WLName COMPUTE
  PCName.tipFPItem = WLName.FPItem;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Import edges do not address the scope rule that prohibits hiding. Consider the following program:

hide.nl[70]==

```
import hide.*;
{ float rate;
  rate = verbs.fly.distance / insects.fly.legs;
}

package insects;
class ant { }
class fly { int legs; }
class bee { }
```

```
package verbs;
class run { }
class fly { int distance; }

package hide;
class insects { }
class verbs { }
```

This macro is attached to a non-product file.

When the applied occurrences `verbs` and `insects` in the assignment statement are analyzed, bindings are first sought in the scope graph node created for the enclosing `Block`, then in the node created for the enclosing `Program`, and finally in the node created for the enclosing `PgmFileBody`. The scope graph node created for that range is the tail of an import edge whose tip is the scope graph node owned by the `hide` package. Both `verbs` and `insects` are defined in the `hide` package, so the entities bound to those identifiers will be returned by the generic lookup.

If there had been no import on demand, bindings for `verbs` and `insects` would have ultimately been sought (and found) in the scope graph node for `Collection`. Thus the import on demand has hidden the defining occurrences in `Collection`, violating the prohibition on hiding.

We can solve the problem by creating an edge from each scope graph node created for a `Program` or `PackageBody` to the scope graph node created for the `Collection`. The semantics of this *bypass* edge differ from those of inheritance and import edges, and we'll assign it the index 3. Because path edges are followed before parent edges, the search will bypass the scope graph node created for `PgmFileBody` and find the package bindings for `verbs` and `insects`. In other words, the bypass edge will implement the prohibition on hiding.

Neither the tips nor the tails of these bypass edges are defined by names. Therefore the `BoundEdge` role is appropriate (see Section "Path edge creation roles" in *Name Analysis Reference Manual*).

Abstract syntax tree[71]==

```
SYMBOL Toplevel INHERITS BoundEdge COMPUTE
  SYNT.tailEnv = THIS.Env;
  SYNT.tipEnv = INCLUDING Collection.Env;
  SYNT.label = 3;
END;

SYMBOL Program     INHERITS Toplevel END;
SYMBOL PackageBody INHERITS Toplevel END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Because we now have more than one edge label, we must define `MaxKindsPathEdge` in a file named `ScopeGraphs.h` (see Section "Scope graphs" in *Name Analysis Reference Manual*).

ScopeGraphs.h content[72]==

```
#define MaxKindsPathEdge 3
```

This macro is defined in definitions 72, 107, and 109.
This macro is invoked in definition 73.

We must protect against `ScopeGraphs.h` being included more than once in some C compilation. Conventionally, Eli uses the name of the include file, in capital letters with dots replaced by underscores, as the controlling symbol:

ScopeGraphs.h[73]==

```
#ifndef SCOPEGRAPHS_H
#define SCOPEGRAPHS_H
ScopeGraphs.h content[72]
#endif
```

This macro is attached to a product file.

We must add `ScopeGraphs.h` to the set of specification files:

Specification files[74]==

```
ScopeGraphs.h
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Demand > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Draw the scope graph for `hide.nl`. Identify the path edges introduced by the computations in this section.
2. Use `Bindings.specs` to generate a processor named `pk` that will show bindings for applied occurrences. Apply `pk` to both `demand.nl` and `hide.nl`, and verify that the analysis is correct.
3. Add the following line as the first line of `demand.nl`, and apply `pk` to the modified file:

   ```
   import verbs.*;
   ```

   a. Is the output what you expected? See Section "Deciding among possible bindings" in *Name Analysis Reference Manual*, for an explanation.
   b. Change the denominator of the assignment to `insects.fly.legs` and apply `pk` to the changed file. Is the output what you expected?
   c. This example shows that introducing ambiguity through import on demand does not result in an error report unless the ambiguous name is actually used. Do you agree or disagree with this design decision? Explain briefly.
4. Add a class `noun` to package `hide` and create a reference to `noun` in the program. Apply `pk` to the modified `hide.nl`.
   a. Did you get the result you expected?
   b. Explain how the generic lookup routine found the binding for `noun`.

4 Interaction with Type Analysis

Classes provide the structure for *objects*. An object can be created by applying the `new` operator to a class. The state of an object, embodied in the values of its variables, can be changed by executing operations on it. A reference to an object can be assigned to a variable of the object's class type or any of its superclass types.

Here is an example whose classes implement a very simple NameLan expression tree:

expr.nl[75]==

```
class Expr {
  int eval() { }
}

class Dyadic extends Expr {
  Oper op; Expr left; Expr right;
  int eval() {
    return op.eval(left.eval(), right.eval());
  }
}

class IntDenot extends Expr {
  int v;        /* v is set when the object is created */
  int eval() { return v; }
}

class Oper {
  int eval(int l, int r) { }
}

class Plus extends Oper {
  int eval(int l, int r) { return l + r; }
}

class Minus extends Oper {
  int eval(int l, int r) { return l - r; }
}

{ IntDenot l = new IntDenot; l.v = 3;
  Expr e = l;
  int v = e.eval();
  /* v now holds the integer value 3 */
}
```

This macro is attached to a non-product file.

The block here creates an object representing an integer expression "`3`" and then evaluates that expression.

The only obvious extensions necessary to accommodate this example are to accept a `Name` as a `Type` and to recognize object creation:

Phrase structure[76]==

```
Type:   Name.
Expr:   'new' Name.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

In these contexts, the `Name` will always be a class name:

Make contexts of complete names explicit[77]==

```
RULE: Type      ::= ClassName       END;
RULE: Expr      ::= 'new' ClassName END;
RULE: ClassName ::= Name            END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

Although these syntactic extensions allow us to accept programs like `expr.nl`, they are not sufficient to support name analysis.

Consider the three occurrences of `eval` in the `return` statement in class `Dyadic`. In none of these cases is `eval`'s qualifier the name of a class having members. Instead, each is a variable containing a reference to an object of a class which has members. The type of that variable tells us the class from which the object was created. For this reason, we speak of a qualified name in which the qualifier is a variable as a *type-qualified* name.

Notice that a type-qualified `ClassName` makes no sense: variables do not have members that are types.

You can see the effect of type-qualified names by generating a processor that is missing computations to derive members from types, and applying that processor to `expr.nl` (see Section "Execute a command in a specified directory" in *Products and Parameters Reference Manual*).

```
-> $elipkg/Name/LearnSG%Type > .
-> . +cmd=(NameLan.specs :exe) (expr.nl) :run
```

4.1 Connect to the Typing module

In order to analyze a type-qualified name, we need to know the qualifier's type. The task of type analysis is to determine a type for every construct that has a type, and therefore name analysis of type-qualified names needs support from type analysis.

The basic type analysis computations are provided by the Eli `Typing` module (see Section "Typed Entities" in *Type Analysis Reference Manual*). We should instantiate this module in order to help us to analyze type-qualified names:

Specification files[78]==

```
$/Type/Typing.gnrc :inst
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

NameLan uses name equivalence for types, so `Typing` represents each type by a unique definition table key. There are three language-defined types, and each class is a user-defined type.

We need to initialize definition table keys for the three language-defined types (see Section "How to specify the initial state" in *The Property Definition Language*). Each of these definition table keys should have a property `IsType` that is set to 1. The type analysis module uses the `IsType` property to distinguish keys representing types from keys representing other entities:

Properties and property computations[79]==

```
intType   -> IsType={1};
floatType -> IsType={1};
voidType  -> IsType={1};
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

Property definitions and computations are specified in a file of type `.pdl`, which must be added to the set of specifications:

NameLan.pdl[80]==

Properties and property computations[79]

This macro is attached to a product file.

Specification files[81]==

```
NameLan.pdl
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

Definition table keys for user-defined types are created by the `TypeDenotation` role (see Section "User-Defined Types" in *Type Analysis Reference Manual*). If a symbol inherits this role, a module computation creates a new key, sets its `IsType` property to `1`, and stores it as the symbol's `Type` attribute:

Construct that represents a subtree denoting a type[82]==

```
SYMBOL ClassBody INHERITS TypeDenotation END;
```

This macro is invoked in definition 88.

This guarantees that each `ClassBody.Type` has a unique definition table key whose `IsType` property value is `1`.

Type keys are made available to a variable declaration via the `DefTableKey`-valued `Type` attribute of the `Type` symbol. The `Type` symbol for a language-defined type is a keyword, while the `Type` symbol for a user-defined type is an identifier:

Establish the Type attribute of Type[83]==

```
ATTR Type: DefTableKey;

RULE: Type ::= 'int'            COMPUTE Type.Type = intType;        END;
RULE: Type ::= 'float'          COMPUTE Type.Type = floatType;      END;
RULE: Type ::= 'void'           COMPUTE Type.Type = voidType;       END;
RULE: Type ::= ClassName        COMPUTE Type.Type = ClassName.Type; END;
```

This macro is invoked in definition 88.

Note that in order to establish the `Type` attribute of a `Type` construct representing a class type, name analysis must first determine the meaning of a (qualified) name. We will return to this point in the next section.

An identifier that is a `Type` symbol plays a different role in type analysis than an identifier that names a typed entity. That means we need to inherit relevant type analysis roles and set any attribute values that the `Typing` module requires:

Construct defining one or more entities of the same type[84]==

```
SYMBOL VarDecl INHERITS TypedDefinition END;

RULE: VarDecl ::= Type VarDefs ';' COMPUTE
  VarDecl.Type = Type.Type;
END;

SYMBOL Parameter INHERITS TypedDefinition END;

RULE: Parameter ::= Type ParamDefName COMPUTE
  Parameter.Type = Type.Type;
END;
```

This macro is invoked in definition 88.

Defining occurrence of an identifier for a typed entity[85]==

```
SYMBOL VarDefName       INHERITS TypedDefId END;
SYMBOL ParamDefName     INHERITS TypedDefId END;
```

This macro is invoked in definition 88.

Defining occurrence of a type identifier[86]==

```
SYMBOL ClassDefName INHERITS TypeDefDefId END;

RULE: ClassDecl ::= 'class' ClassDefName Inheritance ClassBody COMPUTE
  ClassDefName.Type = ClassBody.Type;
END;
```

This macro is invoked in definition 88.

Applied occurrence of a type identifier[87]==

```
SYMBOL ClassName INHERITS TypeDefUseId END;

RULE: ClassName ::= Name COMPUTE
  ClassName.Key = Name.Key;
END;
```

This macro is invoked in definition 88.

In summary, the computations to connect the `Typing` module to our existing name analysis framework are:

Abstract syntax tree[88]==

Establish the Type attribute of Type[83]
Construct defining one or more entities of the same type[84]
Defining occurrence of an identifier for a typed entity[85]
Construct that represents a subtree denoting a type[82]
Defining occurrence of a type identifier[86]
Applied occurrence of a type identifier[87]

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,

65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

4.2 Type-qualified entity names

Consider the declaration of `Dyadic` from `expr.nl`:

```
class Dyadic extends Expr {
  Oper op; Expr left; Expr right;
  int eval() {
    return op.eval(left.eval(), right.eval());
  }
}
```

The variable `op` is a qualifier in the `return` expression, so the processor must determine its type in order to find the member set containing the binding for `eval`. But before the type property of the variable `op` can be determined, name analysis must find the meaning of `Oper`. (We mentioned this constraint in the last section.)

We must make certain that the type of `op` is known before we can analyze the type-qualified name `op.eval`. The type module provides the necessary information by setting the attribute `RootType.TypeIsSet` after all typed identifiers have their type properties set (see Section "Dependence in Type Analysis" in *Type Analysis Reference Manual*). We can use this attribute to establish the precondition for analyzing qualified names in the contexts where those names might be type-qualified (see Section 2.1 [Qualified names], page 20). The precondition must be passed down to *every* qualifier in a qualified name:

Abstract syntax tree[89]==

```
RULE: MethName  ::= Name COMPUTE
  Name.ContextIsReady += INCLUDING RootType.TypeIsSet;
END;

RULE: ExprName  ::= Name COMPUTE
  Name.ContextIsReady += INCLUDING RootType.TypeIsSet;
END;

RULE: Name ::= Name '.' QualifiedId COMPUTE
  Name[2].ContextIsReady += Name[1].ContextIsReady;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,
65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

We cannot make `RootType.TypeIsSet` a precondition for names in the `ClassName` context without introducing an extra worklist algorithm, because a `ClassName` can set the type of variable and therefore contributes to `RootType.TypeIsSet`. As language designers, we choose to avoid this complexity by introducing the following rule:

- Variable names are not allowed in `ClassName` contexts.

This restriction has no significant effect on the expressive power of NameLan.

The generic lookup invokes `AccessNodesFromQualifier` when dealing with a `QualifiedId`, to obtain the set of bindings in which that applied occurrence must be sought (see Section "Obtain a range from a qualifier" in *Name Analysis Reference Manual*). The default implementation of `AccessNodesFromQualifier` yields the set of bindings owned by the qualifier. If the qualifier is a variable, however, `AccessNodesFromQualifier` must yield the set of bindings owned by that variable's type. Thus, as developers, we need to override the default implementation of `AccessNodesFromQualifier`:

AccessNodesFromQualifier.c[90]==

```
#include "AccessNodesFromQualifier.h"
#include "pdl_gen.h"
NodeTuplePtr
AccessNodesFromQualifier(DefTableKey qualifier, DefTableKey appselector)
{ NodeTuplePtr res = GetOwnedNodeTuple(qualifier, NoNodeTuple);
  if (res == NoNodeTuple)
    res = GetOwnedNodeTuple(GetTypeOf(qualifier, NoKey), NoNodeTuple);
  return res;
}
```

This macro is attached to a product file.

Note that this implementation of `AccessNodesFromQualifier` first assumes that the qualifier owns a set of members. If that assumption is correct, the set members is returned. Only if the qualifier does not own any members does `AccessNodesFromQualifier` obtain the qualifier's type and seek the members of that type. This behavior is necessary because `AccessNodesFromQualifier` is invoked for *every* `QualifiedId` regardless of the context.

`AccessNodesFromQualifier.c` is one of the specification files:

Specification files[91]==

```
AccessNodesFromQualifier.c
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

The only remaining issue is how to report a violation of the rule that a variable name is not allowed in a `ClassName` context. That report should be attached to the leftmost incorrect component of the `ClassName`, which might be either a `SimpleName` or a `QualifiedId`. We'll wrap the necessary computation in a role named `ChkLegalClassName`:

Abstract syntax tree[92]==

```
SYMBOL SimpleName  INHERITS ChkLegalClassName END;
SYMBOL QualifiedId INHERITS ChkLegalClassName END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

`ChkLegalClassName` should report an error if and only if all of the following conditions are satisfied:

- The applied occurrence identifies a binding '`(i, k)`'.
- The entity '`k`' is a variable.
- The applied occurrence is in a `ClassName` context.

Abstract syntax tree[93]==

```
SYMBOL ChkLegalClassName COMPUTE
  IF(AND(AND(
      NOT(EQ(THIS.Key, NoKey)),
      GetIsVariable(THIS.Key, 0)),
      INCLUDING (ClassName.InClassName, RootScope.InClassName)),
    message(
      ERROR,
      CatStrInd ("Must not occur in a class name: ", THIS.Sym),
      0,
      COORDREF));
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The `IsVariable` property must be set to distinguish entities that are variables:

Properties and property computations[94]==

```
IsVariable: int;        /* 1 if the entity is a variable */
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

Abstract syntax tree[95]==

```
SYMBOL VarDefName COMPUTE
  SYNT.GotDefKeyProp += ResetIsVariable(THIS.Key,1);
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Making `GotDefKeyProp` depend on the setting of the `IsVariable` property ensures that the property's value will be available before any computation attempts to access it (see Section "Defining Occurrences" in *Name Analysis Reference Manual*).

Finally, we need to set the `InClassName` attribute that defines the context:

Abstract syntax tree[96]==

```
SYMBOL ClassName, RootScope: InClassName: int;
SYMBOL ClassName COMPUTE SYNT.InClassName = 1; END;
SYMBOL RootScope COMPUTE SYNT.InClassName = 0; END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%TQ > .
```

None of these files will have write permission in your current directory.

1. Verify that type-qualified names are correctly handled by generating a processor called `tq` and applying that processor to `expr.nl`.
2. List the four kinds of entity that can be bound to NameLan identifiers.
3. An identifier bound to an entity of each kind might occur in a qualified name. Write a correct NameLan program in which an identifier bound to each kind of entity occurs in some qualified name. Verify your program using `tq`.
4. State the kinds of entity that can not legally be bound to an identifier occurring in each of the following contexts:
   - after a dot in a qualified name
   - before a dot in a qualified name
   - in a qualified `ClassName`
   - at the end of a type-qualified name

   Briefly explain each of your answers.
5. Write a syntactically correct program with qualified names that are illegal for the reasons you stated above, and check it with `tq`. Are you satisfied by the results? Explain briefly.

4.3 Type-qualified edge names

We have seen that type-qualified names require cooperation between name and type analysis modules, and how they introduce dependence among computations. The type-qualified names in `expr.nl` describe components of an expression in a value context. Suppose that, for some reason, there is a section of code containing a number of references to the members of some computed object. The NameLan `with` statement allows the user to use simple names to refer to the members of a class:

Phrase structure[97]==

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

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

The attributes needed for a `Name` in this context will differ from those needed in other contexts, so we define a new abstract syntax symbol to differentiate the context:

Make contexts of complete names explicit[98]==

```
RULE: Statement ::= 'with' WithName 'do' WithBody END;
RULE: WithName  ::= Name                          END;
```

This macro is defined in definitions 37, 44, 56, 57, 68, 77, and 98.
This macro is invoked in definition 38.

Because the `WithName` may be type-qualified, we need to ensure that all types have been defined before analyzing it:

Ensure that types are defined[99]==

```
RULE: WithName ::= Name COMPUTE
  Name.ContextIsReady += INCLUDING RootType.TypeIsSet;
```

```
END;
```

This macro is invoked in definition 105.

Here is an example in which the `WithName` is a simple variable name:

with.nl[100]==

```
class T { int a, b, c; }
{ float a, c = 1.2;
  T p = new T; p.c = 3;
  T r = new T; r.b = 4;
  a = r.b + c;
  with p do a = r.b + c;
}
```

This macro is attached to a non-product file.

Our scope rule for a NameLan `with` statement is:

- The `WithBody` is an anonymous class whose superclass is the class of the `WithName`.

This rule implies that the `WithBody` must be a range:

Abstract syntax tree[101]==

```
SYMBOL WithBody INHERITS RangeScope END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Recall that inheritance is modeled by a path edge in the scope graph (see Section 2.2 [Inheritance], page 24). We need to add that edge to the graph, but the edge tip is defined by a name that may be type-qualified. Therefore the inheritance edge cannot be constructed until all edges have been added to the graph *and* the types of all typed identifiers have been determined. This is clearly impossible: "all edges have been added" cannot be a precondition for "add an edge".

The paradox can be avoided by using the `OutSideInDeps` role to partition the name analysis of `with.nl` into two problems (see Section "The OutSideInDeps role" in *Name Analysis Reference Manual*).

Partition the program analysis[102]==

```
SYMBOL WithBody INHERITS OutSideInDeps END;
```

This macro is defined in definitions 102 and 104.
This macro is invoked in definition 105.

This statement implies that there is a subgraph '`W`' of the scope graph '`G`', such that no edge may have its tail in '`G-W`' and its tip in '`W`'. Subgraph '`W`' corresponds to the abstract syntax subtree rooted in the `WithBody` node. Given this property of the scope graph, we can complete the name and type analysis of lines 1-4 of `with.nl` without considering the inheritance edge. At that point we will have all of the information necessary to add the inheritance edge and analyze the `WithBody`.

The `OutSideInEdge` role attaches computations to add the inheritance edge. The symbol inheriting `OutSideInEdge` corresponds to the scope graph node that is the tail of the inheritance edge, and the tip is the node obtained from the `Key` of the `WithName` (see Section "The OutSideInDeps role" in *Name Analysis Reference Manual*).

Add the path edge to the graph[103]==

```
SYMBOL WithBody INHERITS OutSideInEdge END;

RULE: Statement ::= 'with' WithName 'do' WithBody COMPUTE
  WithBody.tipEnv = AccessNodesFromQualifier(WithName.Key, NoKey);
END;

RULE: WithName ::= Name COMPUTE
  WithName.Key = Name.Key;
END;
```

This macro is invoked in definition 105.

We need to partition the type analysis as well (see Section "Interaction with type analysis" in *Name Analysis Reference Manual*).

Partition the program analysis[104]==

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

This macro is defined in definitions 102 and 104.
This macro is invoked in definition 105.

Combining the attribute computations and adding them to the other specifications gives us a complete specification from which we can generate a processor that handles programs like `with.nl`:

Abstract syntax tree[105]==

```
Ensure that types are defined[99]
Partition the program analysis[102]
Add the path edge to the graph[103]
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%With > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Use `Bindings.specs` to generate a processor named `with` that will show the bindings for applied occurrences, and apply it to `with.nl`. Is the result what you expected?
2. The variable `r.b` is accessed via a qualified name within the `with` statement of `with.nl`. Briefly explain why you could not implement a construct like the following to allow `r.b` to be accessed via a simple name:

```
with p,r do a = b + c;
```

3. Suppose that a language designer allowed the `WithName` to be a list as in the previous exercise. Provide reasonable scope rules for such a construct.

5 Multiple Scope Graphs

The scope graph is the basic data structure for name analysis (see Section "Fundamentals of Name Analysis" in *Name Analysis Reference Manual*). Its structure embodies the relationships among ranges dictated by scope rules, and its content reflects the set of defining occurrences subject to those rules. There are two reasons that a processor specification may require more than one scope graph:

- The language permits several defining occurrences of the same identifier to bind to different entities in a given range.
- Different constructs in the language obey different scope rules.

In the first case the scope graphs are isomorphic, but the contents vary because they describe different defining occurrences; in the second the structure varies because the scope rules are different. This chapter extends NameLan in two ways, to illustrate the necessary computations.

5.1 Reusing identifiers in the same scope

It's hard to think up names! Suppose that, as language designers, we allow a programmer to use the same identifier to represent a package, a class, a variable, and a method in a single range. If we, as developers, can decide which of these is meant on the basis of context then we can use distinct scope graphs to obtain the proper defining occurrence for each applied occurrence. Here is an example:

mgcd.nl[106]==

```
import gcd.*;
class gcd { int z; }
int gcd;
int gcd(int x, int y) {
  while (x != y) {
    if (x > y) x = x - y;
    else y = y - x;
  }
  return x;
}
{ gcd c;
  gcd = gcd(c.z, w); }

package gcd;
int w;
```

This macro is attached to a non-product file.

The program in `mgcd.nl` contains four applied occurrences of `gcd` with different meanings. You would probably have little difficulty deciding on the basis of context that the first applied occurrence names the package, the second names the class, the third names the variable, and the fourth names the method.

An Eli-generated processor needs four scope graphs to support that intuition, each modeling the bindings for one kind of entity. Because the scope rules are the same for all of the

identifiers naming those entities, the scope graphs will have the same structure but different contents; the graphs themselves are *isomorphic* (see Section "Isomorphic Scope Graphs" in *Name Analysis Reference Manual*).

A single instantiation of the `ScopeGraphs` module will support any number of isomorphic scope graphs, but the scope graph data structures that Eli builds when generating a processor must be capable of handling the maximum number of isomorphic graphs supported by any single instantiation. This number is conveyed by the pre-processor identifier `MaxIsoGraphs` (see Section "Isomorphic Scope Graphs" in *Name Analysis Reference Manual*). We need to override the default value of 1 by placing a directive in `ScopeGraphs.h` (see Section 3.2 [Import on demand], page 38).

ScopeGraphs.h content[107]==

```
#define MaxIsoGraphs  4
```

This macro is defined in definitions 72, 107, and 109.
This macro is invoked in definition 73.

The actual number of isomorphic scope graphs associated with a single instantiation of the module is specified by the `NumberOfIsoGraphs` attribute of the root symbol of the grammar (see Section "Isomorphic Scope Graphs" in *Name Analysis Reference Manual*).

Abstract syntax tree[108]==

```
SYMBOL Collection COMPUTE SYNT.NumberOfIsoGraphs=4; END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53,
65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111,
112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Each graph is indexed by a integer whose value is between 0 and 3. Values larger that 3 can be used to give information about the syntactic context, but indicate that the specific graph to be used either is unknown or must be computed. (We call these *weak* contexts to distinguish them from the *strong* contexts where the specific graph can be determined.)

It's a good idea to use a symbolic name instead of a numeric value when specifying a graph index. Appropriate names can be defined as a C enumeration in `ScopeGraphs.h`:

ScopeGraphs.h content[109]==

```
enum {
  packageGraph, classGraph, variableGraph, methodGraph,
  packageOrClassContext, ambiguousContext
};
```

This macro is defined in definitions 72, 107, and 109.
This macro is invoked in definition 73.

The first four identifiers, whose representations end in `Graph`, index the scope graphs. The remaining identifiers, whose representations end in `Context`, model weak contexts.

Each defining or applied occurrence has a `GraphIndex` attribute. By default, the values of those attributes are set to 0. This default corresponds to the case of a single scope graph.

When the module instantiation supports several isomorphic scope graphs, the developer must override the default values of the `GraphIndex` attributes with the index of the appropriate scope graph. Defining occurrences are strong contexts that are straightforward to specify (note that a parameter is considered to be a variable):

Abstract syntax tree[110]==

```
ATTR GraphIndex: int;

SYMBOL PackageDefName COMPUTE
  INH.GraphIndex = packageGraph;
END;

SYMBOL ClassDefName COMPUTE
  INH.GraphIndex = classGraph;
END;

SYMBOL VarDefName COMPUTE
  INH.GraphIndex = variableGraph;
END;

SYMBOL ParamDefName COMPUTE
  INH.GraphIndex = variableGraph;
END;

SYMBOL MethodDefName COMPUTE
  INH.GraphIndex = methodGraph;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Applied occurrences are always components of the *complete names* that we have defined throughout this document (see Section 2.1 [Qualified names], page 20). As language designers, we must state NameLan rules for the context of each complete name, and then as developers derive LIDO computations to implement them. Here we write just the LIDO computation for each such context, however, leaving it to you to deduce an appropriate NameLan rule:

Abstract syntax tree[111]==

```
RULE: ClassName ::= Name COMPUTE
  Name.GraphIndex = classGraph;
END;

RULE: ExprName  ::= Name COMPUTE
  Name.GraphIndex = variableGraph;
END;

RULE: MethName  ::= Name COMPUTE
  Name.GraphIndex = methodGraph;
END;

RULE: WithName  ::= Name COMPUTE
  Name.GraphIndex = variableGraph;
```

```
END;

RULE: SuperClass ::= WLName COMPUTE
  WLName.GraphIndex = classGraph;
END;

RULE: ImportName ::= WLName COMPUTE
  WLName.GraphIndex = classGraph;
END;

RULE: PCName    ::= WLName COMPUTE
  WLName.GraphIndex = packageOrClassContext;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Note that all but one of these complete names is in a strong context. The only weak context is the `PCName`, which could be either a package name or a class name (see Section 3.2 [Import on Demand], page 38).

Once we know the context of a complete name, we need to analyze the applied occurrences making up that name. The context for a complete name applies only to the rightmost applied occurrence of that complete name. If the complete name is a simple name, the computations are:

Abstract syntax tree[112]==

```
RULE: Name ::= SimpleName COMPUTE
  SimpleName.GraphIndex = Name.GraphIndex;
END;

RULE: WLName ::= SimpleWLName COMPUTE
  SimpleWLName.GraphIndex = WLName.GraphIndex;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The NameLan rules that we, as language designers, have formulated to define the meanings of other applied occurrences come in two parts: definition of the weak contexts of any qualifiers, and then resolution of those weak contexts by selecting possible bindings. The rules for qualifier contexts are:

- A name to the left of the rightmost `.` in a qualified `ClassName` is a package-or-class name.
- A name to the left of the rightmost `.` in a qualified package-or-class name is a package-or-class name.
- A name to the left of the rightmost `.` in a qualified `ExprName` is an *ambiguous* name.
- A name to the left of the rightmost `.` in a qualified `MethName` is an ambiguous name.

- A name to the left of the rightmost `.` in a qualified ambiguous name is an ambiguous name.

Here is a LIDO implementation of these rules:

Abstract syntax tree[113]==

```
RULE: Name ::= Name '.' QualifiedId COMPUTE
  QualifiedId.GraphIndex = Name[1].GraphIndex;
  Name[2].GraphIndex =
    IF(EQ(Name[1].GraphIndex,classGraph),
      packageOrClassContext,
    IF(EQ(Name[1].GraphIndex,packageOrClassContext),
      packageOrClassContext,
    ambiguousContext));
END;

RULE: WLName ::= WLName '.' QualifiedWLId COMPUTE
  QualifiedWLId.GraphIndex = WLName[1].GraphIndex;
  WLName[2].GraphIndex = packageOrClassContext;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The `GraphIndex` attribute of the applied occurrence is passed to the generic lookup as the `GraphIndex` property of the applied occurrence's `UseKey` (see Section "Applied Occurrences" in *Name Analysis Reference Manual*). If the value of the `GraphIndex` property is less than the number of isomorphic scope graphs, `N`, the generic algorithm seeks a defining occurrence in the indexed graph (recall that indexes start at 0). Otherwise, it seeks a defining occurrence independently in each of the graphs, and presents the results in an array `G` of length `N`. If no defining occurrence was found in graph `k`, then `G[k]` contains the value `NoKey`.

If the value of the `GraphIndex` property was greater than or equal to `N`, the generic lookup invokes a C function named `DisambiguateGraphs` before returning (see Section "Deciding among possible bindings" in *Name Analysis Reference Manual*). `DisambiguateGraphs` is given `G`, `N`, and the `UseKey` as arguments, and must return an appropriate value as the final result of the generic lookup.

Eli's default version of `DisambiguateGraphs` checks whether exactly one of the searches yielded a result other than `NoKey`. If so, then it returns that result; otherwise it returns `NoKey`. This implements the rule that wherever the context would allow a simple name to have more than one meaning, the normal scope rules find only one of those meanings. File `mgcd.nl` satisfies that condition, but `ambig.nl` does not:

ambig.nl[114]==

```
import v.*;

class A {
  class C { int x; }
```

```
}

A v;
{ v.C x;
  v.C.x = 0; }

package v;
class C { int x; }
```

This macro is attached to a non-product file.

Consider the two uses of `v` at the beginning of the program block. It is reasonable to interpret the first as an applied occurrence of the package name, since using a variable in the type of another variable is counterintuitive. Interpretation of the second as an applied occurrence of the variable name is also reasonable, although perhaps not as obvious as the first. If NameLan is to allow programs like `ambig.nl`, it must provide appropriate rules for the programmer. Those rules obviously can't be stated in terms of `G[k]`, which is an implementation detail.

The solution is to explain the choice as a reclassification of the context, based on the results of the searches. This reclassification turns a weak context into a strong context, which requires the desired selection. Here are our rules for NameLan:

- Package-or-class names are reclassified as follows:
  - If the package-or-class name is a simple name consisting of a single identifier:
    - If the identifier is in the scope of a class declaration then the package-or-class name is reclassified as a class name.
    - Otherwise the package-or-class name is reclassified as a package name.
  - If the package-or-class name is a qualified name, then it is reclassified as a class name.
- Ambiguous names are reclassified as follows:
  - If the ambiguous name is a simple name consisting of a single identifier:
    - If the identifier is in the scope of a variable or parameter declaration then the ambiguous name is reclassified as a variable name.
    - Otherwise if the ambiguous name is in the scope of a class declaration then the ambiguous name is reclassified as a class name.
    - Otherwise the ambiguous name is reclassified as a package name.
  - If the ambiguous name is a qualified name, the qualifier is first reclassified. Then:
    - If the qualifier is reclassified as a package name or a class name then:
      - If the identifier is a variable name in the qualifier's range then the ambiguous name is reclassified as a variable name.
      - Otherwise if the identifier is a class name in the qualifier's range then the ambiguous name is reclassified as a class name.
      - Otherwise an error is reported.
    - If the qualifier is reclassified as a variable name of type '`T`' then:
      - If the identifier is a variable name in '`T`''s class then the ambiguous name is reclassified as a variable name.

- Otherwise if the identifier is a class name in 'T''s class then the ambiguous name is reclassified as a class name.
- Otherwise an error is reported.

Note that these rules do not require any actual modification of any GraphIndex. All they do is to specify which of the elements of G DisambiguateGraphs should return. There is no requirement that the selected element of G differ from NoKey. If the variable GraphIndex contains the value of the GraphIndex property of the UseKey, here is a C computation that implements the NameLan rules:

Implementation of the NameLan reclassification[115]==

```
if (GraphIndex == packageOrClassContext) {
  if (G[classGraph] != NoKey) return G[classGraph];
  return G[packageGraph];
}

if (GraphIndex == ambiguousContext) {
  if (G[variableGraph] != NoKey) return G[variableGraph];
  if (G[classGraph] != NoKey) return G[classGraph];
  return G[packageGraph];
}
```

This macro is invoked in definition 116.

The simplest way to complete the implementation is to download Eli's version of DisambiguateGraphs.c and then replace the default computation:

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

DisambiguateGraphs.c[116]==

```
#include "deftbl.h"
#include "pdl_gen.h"
#include "ScopeGraphs.h"

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 GraphIndex = GetGraphIndex(app, 0);

  Implementation of the NameLan reclassification[115]

  return NoKey;
}
```

This macro is attached to a product file.

GetGraphIndex is used to obtain the UseKey's GraphIndex property (see Section "Behavior of the basic query operations" in *The Definition Table Module*). The include file

`pdl_gen.h` provides the interface for `GetGraphIndex`, and `ScopeGraphs.h` defines the symbolic constants used in the computation. The final `return` statement avoids a compiler warning.

`DisambiguateGraphs.c` must be included in the list of specifications:

Specification files[117]==

```
DisambiguateGraphs.c
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Kinds > .
```

None of these files will have write permission in your current directory.

1. Explain why all qualifier contexts of `WLName`s are `packageOrClassContext`.
2. Explain why the default implementation of `DisambiguateGraphs.c` suffices for analyzing `mgcd.nl` (see Section "Deciding among possible bindings" in *Name Analysis Reference Manual*).
3. Consider the two uses of `v` at the beginning of `ambig.nl`'s program block.
   a. Give the contents of the array `G` for the first applied occurrence of `v`.
   b. Use the context reclassification rules of NameLan to explain how the context of the first use of `v` should be reclassified.
   c. Show that `DisambiguateGraphs` will return the correct graph index for this applied occurrence of `v`.
   d. Repeat the first three parts of this exercise for the second applied occurrence of `v`.
4. Use `Bindings.specs` to generate a processor and apply it to `ambig.nl`. Did you get the results you expected?

5.2 Constructs obeying different scope rules

Consider the task of reading integer values from a file until the current number would cause the running sum to exceed the first value on the file. The program must print the number of values making up the final sum. The problem here is that there are two conditions under which to exit the loop: either the data runs out or the preset value is exceeded.

As language designers, we can extend NameLan to simplify the solution to this problem:

Phrase structure[118]==

```
Statement:      Loop / 'break' Ident ';'.
Loop:           Ident ':' 'while' '(' Expr ')' Statement.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

Here is a program that uses the extension:

maxsum.nl[119]==

```
import stdio.*;
{ int sum = 0, count = 0, below;
  below = getint();
  loop: while (feof(stdin) == 0) {
    int next = sum + getint();
    if (next > below) break loop;
    sum = next; count = count + 1;
  }
  putint(count);
}

package stdio;
class file { }
file stdin;
int feof(file f) { }
int getint() { }
void putint(int v) { }
```

This macro is attached to a non-product file.

As developers, we need new abstract syntax symbols to distinguish the two new contexts for an identifier (see Section "Representation of identifiers" in *Name Analysis Reference Manual*). The obvious choices are `LabelDef` and `LabelUse`:

Abstract syntax of identifiers[120]==

```
RULE: Loop      ::= LabelDef ':' 'while' '(' Expr ')' Statement END;
RULE: LabelDef  ::= Ident END;

RULE: Statement ::= 'break' LabelUse ';' END;
RULE: LabelUse  ::= Ident END;
```

This macro is defined in definitions 8, 9, 26, 31, 33, 42, 51, and 120.
This macro is invoked in definition 10.

The `Loop` construct is a labeled `while` statement. Execution of a break statement terminates the smallest enclosing `Loop` construct defining the given label. Our scope rule is:

- The scope of a defining occurrence `LabelDef` is the smallest enclosing `Loop`.

The key point for name analysis of labels is that this scope rule requires a scope graph whose structure is completely different from that of the scope graphs we have been using. It therefore cannot be implemented by isomorphism, but requires an additional instantiation of the `ScopeGraphs` and `SGProof` modules (see *Name Analysis Reference Manual*).

Specification files[121]==

```
$/Name/ScopeGraphs.gnrc +instance=LBL_                  :inst
$/Name/SGProof.gnrc     +instance=LBL_+referto=Ident    :inst
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

The `instance` argument `LBL_` is needed to distinguish this instantiation from the one used for analysis of other names (see Section "Basic Scope Rules" in *Name Analysis according to scope rules*). That instantiation argument must prefix each role name used for analysis of labels:

Abstract syntax tree[122]==

```
SYMBOL Loop     INHERITS LBL_RangeScope                  END;
SYMBOL LabelDef INHERITS LBL_IdDefScope                  END;
SYMBOL LabelUse INHERITS LBL_GCSimpleName, LBL_ChkIdUse END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Labels > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Use `Bindings.specs` to generate a processor named `lbl` and apply it to `maxsum.nl`.

   It may happen that Eli does not find an evaluation order for the computations of this specification. More than one instantiation plus type-qualified inheritance relations (e.g. the `with` statement) present too many possibilities. In that case, you need to eliminate some of those possibilities by adding dependence. The following delays the computations of the `LBL_` instance until the computations of the original instance are done. That may help Eli to find an evaluation order:

   ```
   SYMBOL LabelDef COMPUTE
     INH.LBL_GraphIndex=0 <- INCLUDING OutSideInDeps.FPSolved;
   END;
   ```

6 Selecting Acceptable Bindings

Consider the file `clash.nl`:

clash.nl[123]==

```
import fsm.*;
import random.*;
{ int count = 0;
  state = 0;
  while (state != 7) {
    if (ran() < 0.25) advance(1);
    else if (ran() < 0.8) advance(2);
    else advance(3);
    count = count + 1;
  }
}

package fsm;
int state;
void advance(int input) {
  if (input == 1) state = state - 1;
  else if (input == 2) state = state + 1;
}

package random;
int state = 100001;
float ran() {
  state = state * 125;
  state = state - (state / 2796203) * 2796203;
  return state / 2796203.0;
}
```

This macro is attached to a non-product file.

Two entities named `state` are available in the context of the program body, imported on demand from separate packages. Because `state` is ambiguous, no unique binding for either of the two applied occurrences in lines 4 and 5 can be found. See Section 3.2 [Import on demand], page 38, Exercise 3.c, for a discussion of this problem.

File `clash.nl` illustrates a weakness of the package facility. The variable `random.state` must retain its value from one call of `ran` to another, but there is no need for the name to be visible outside of `random`. As language designers, we could address this weakness by adding rules to NameLan allowing the programmer to specify that the binding for `random.state` could not be identified by an applied occurrence of `state` in the context of lines 4 and 5 of `clash.nl`.

This situation typifies an aspect of the name analysis process that we call *acceptability*: `random.state` is an *available* binding for the applied occurrence in line 4 of `clash.nl` according to the general scope rules of NameLan, but the additional rules prevent the entity bound by it from being *acceptable* in that context.

There are two common kinds of additional rules that prevent an available binding from being acceptable in a particular context: *access rules* and *position rules*. Access rules define some directive that a user can attach to a declaration to specify the contexts in which the binding it creates is acceptable. Position rules involve the relative locations of the defining and applied occurrences in the source text. We will cover each in the remainder of this chapter.

6.1 Access rules

As language designers, we can solve the problem of `clash.nl` by introducing a `private` directive into NameLan. Any variable, method, or class can be marked as being private to the containing package:

Phrase structure[124]==

```
Declaration: 'private' VarDecl.
Declaration: 'private' MethodDecl.
Declaration: 'private' ClassDecl.
```

This macro is defined in definitions 2, 25, 30, 32, 41, 50, 55, 67, 76, 97, 118, and 124.
This macro is invoked in definition 3.

The following access rules describe the effect of a `private` directive:

- Let binding '`(i,k)`' be associated with a defining occurrence of '`i`' located in package '`P`' and marked private. An applied occurrence of '`i`' that is not located in package '`P`' cannot identify '`(i,k)`'.
- Let binding '`(i,k)`' be associated with a defining occurrence of '`i`' located in class '`A`' within package '`P`' and marked private. Let '`C`' be a subclass of '`A`'. An applied occurrence of '`i`' within the body of '`C`' cannot identify '`(i,k)`' if '`C`' or any class '`B`' that is a superclass of '`C`' and a subclass of '`A`' is not located in package '`P`'.

Here's how a `private` directive is used to avoid the ambiguity that occurred in `clash.nl`:

private.nl[125]==

```
import fsm.*;
import random.*;
{ int count = 0;
  state = 0;
  while (state != 7) {
    if (ran() < 0.25) advance(1);
    else if (ran() < 0.8) advance(2);
    else advance(3);
    count = count + 1;
  }
}

package fsm;
int state;
void advance(int input) {
  if (input == 1) state = state - 1;
  else if (input == 2) state = state + 1;
```

```
}

package random;
private int state = 100001;
float ran() {
  state = state * 125;
  state = state - (state / 2796203) * 2796203;
  return state / 2796203.0;
}
```

This macro is attached to a non-product file.

The access rules describing the `private` directive have no effect on the structure of the generic lookup algorithm; there are still two available bindings in `private.nl` for the applied occurrence of `state` on line 4. In order to handle access rules, the generic lookup algorithm invokes one of three functions whenever it finds a binding at a scope graph node `n` (see Section "Is the binding acceptable?" in *Name Analysis Reference Manual*). Which of the three is invoked depends on the context in which the search was undertaken and the state of the search:

`isAcceptableSimple(DefTableKey def, DefTableKey app)`
: is invoked in a search for a simple identifier when `n` is the initial node or a node reached by following a parent edge.

`isAcceptableQualified(DefTableKey def, DefTableKey app)`
: is invoked in a search for a qualified identifier when `n` is the initial node.

`isAcceptablePath(DefTableKey def, DefTableKey app, int kind)`
: is invoked in a search for a simple identifier or a qualified identifier when `n` is the tip of a path edge with label `kind`.

The `def` argument is the available binding, and `app` is the `UseKey` of the applied occurrence (see Section "Applied Occurrences" in *Name Analysis Reference Manual*).

A default version of each of these functions, which finds any binding acceptable, is available in the library; the appropriate default version will be used unless the developer overrides it by specifying a C-coded function. As developers, we implement the access rules describing the effect of a `private` directive by providing appropriate implementations for these functions.

Recall that the first two arguments to these functions are definition table keys. Those keys must have properties that will allow the functions to make the appropriate decision (see Section "The Definition Table Module" in *Property Definition Language*).

Properties and property computations[126]==

```
InPackage: DefTableKey; /* Enclosing package */
IsPrivate: int;         /* 1 if the available binding is private */
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

If a binding is found for a simple name in an initial node or a node reached by following a parent edge, then the applied occurrence lies in the range of the defining occurrence. The range of a defining occurrence does not cross a package boundary, so the binding will always be acceptable. We can therefore use the default `isAcceptableSimple`.

If the initial node is a qualifier, a binding found for the qualified identifier will not be acceptable if the applied occurrence is not in the same package as the defining occurrence. This condition is checked by the following computation, which is common to both `isAcceptableQualified` and `isAcceptablePath`:

Common private access check[127]==

```
if (!GetIsPrivate(def, 0)) return AcceptBinding;
defpkg = GetInPackage(def, NoKey);
apppkg = GetInPackage(app, NoKey);
if (defpkg != apppkg) return IgnoreSkipPath;
```

This macro is invoked in definitions 128 and 130.

If the available binding isn't private, then it is acceptable. If the available binding and the applied occurrence are not in the same package, then the binding is not acceptable. The function returns `IgnoreSkipPath` in that case because, although the binding is not acceptable, it hides any other bindings for the same identifier along the current search path (see Section "Is the binding acceptable?" in *Name Analysis Reference Manual*).

At the end of this computation, we know that the entity is private, and that its defining occurrence is in the same package as the applied occurrence. That information is sufficient to accept a binding for a qualified identifier at the initial node:

Check acceptability of a qualified identifier[128]==

```
int
isAcceptableQualified (DefTableKey def, DefTableKey  app)
{ DefTableKey defpkg, apppkg;
  Common private access check[127]
  return AcceptBinding;
}
```

This macro is invoked in definition 138.

Let's see how this works on a contrived example:

accessPack.nl[129]==

```
{ }

package P;
class A { private int x, y; } /* package access */
class B {
  class C extends A {
    void f() { x = y; } /* legal */
  }
  class D extends Q.F {
    void g() { x = A.y; } /* illegal x */
  }
}

package Q;
class F extends P.A {
  void h() { x = P.A.y; } /* illegal x, y */
```

```
}
```

This macro is attached to a non-product file.

First consider the qualified name `A.y` in method `g`. The generic lookup finds a binding for the qualified identifier `y` in the node for the members of class `A`, which is the initial node of the search. It therefore invokes `isAcceptableQualified`, and the arguments would have the following properties:

```
IsPrivate attribute of def = 1
InPackage attribute of def = P
InPackage attribute of app = P
```

You should verify that the result would be `AcceptBinding`.

Now consider the qualified name `P.A.y` in method `h`. The generic lookup finds the same binding for the qualified identifier `y`, but this time the `InPackage` property of the `app` argument is `Q`. You should verify that `isAcceptableQualified` would return `IgnoreSkipPath` in this case.

All of the applied occurrences of `x` in `accessPack.nl` must have their available bindings checked by `isAcceptablePath` because each of those bindings is found at the tip of an inheritance edge. The binding for the applied occurrence of `x` in method `h` can be rejected by the common private access check, but that check will neither accept nor reject the bindings for the applied occurrences in method `f` or method `g`.

Consider the applied occurrence of `x` in method `g` of class `D`. The generic search will find the available binding in class `A` of package `P`, and that binding is marked private. `D` is a subclass of `F`, and `F` is a subclass of `A`. Class `F` does not lie in package `P`. According to the scope rules of NameLan, this means that the applied occurrence of `x` in method `g` cannot identify the binding in class `A`.

An inheritance path consists of a sequence of scope graph nodes owned by classes. The scope graphs module provides a function, `CheckPathsNsp`, that visits every node in the path defined by a starting node, an ending node, and a kind of path edge (see Section "Useful graph operations" in *Name Analysis Reference Manual*). The fourth argument of `CheckPathsNsp` is a user-defined function that `CheckPathsNsp` invokes at each node visit, passing the definition table key of the owner of the node being visited.

In order to verify accessibility of a binding that is neither accepted nor rejected by the common private access check, `isAcceptablePath` must verify that all nodes of the inheritance path are owned by classes within the package containing its declaration:

Check acceptability at the tip of a path edge[130]==

```
int
isAcceptablePath (DefTableKey def, DefTableKey  app, int lab)
{ DefTableKey defpkg, apppkg;
  Common private access check[127]
  pkg = defpkg; /* defpkg is set by the Common private access check */
  if (!CheckPathsNsp(
        GetInClassNode(app, NoNodeTuple),
        GetInClassNode(def, NoNodeTuple),
        lab,
        ChkClass))
    return IgnoreSkipPath;
```

```
  return AcceptBinding;
}
```

This macro is invoked in definition 138.

`ChkClass` is the function that `CheckPathsNsp` uses to verify that a class on the inheritance path is in the package containing the declaration. It requires a global variable `pkg`, which must be set before the path is scanned:

Call-back function to verify inheritance paths[131]==

```
int
ChkClass(DefTableKey cls)
{ if (GetInPackage(cls, NoKey) == pkg) return 1;
  return 0;
}
```

This macro is invoked in definition 138.

In order to specify the beginning of the inheritance path for a particular binding, we need to know the scope graph node owned by the class containing the applied occurrence of the identifier being sought; the end of that inheritance path is the scope graph node containing the binding. The necessary information can be provided by an `InClassNode` property for each identifier occurrence:

Properties and property computations[132]==

```
InClassNode: NodeTuplePtr;
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

Let us now consider how the properties that we have defined are set. We will use the class symbol `DeclContext` to abstract the two contexts, `Declaration` and `DeclStmt`, in which entities are declared. A computation can then reach up to the enclosing `DeclContext` for an attribute, `private`, indicating whether there was a `private` directive. In most cases there will be no such directive, so we provide a default computation for `DeclContext.private`:

Abstract syntax tree[133]==

```
SYMBOL DeclContext: private: int;
SYMBOL DeclContext COMPUTE SYNT.private = 0; END;

SYMBOL Declaration INHERITS DeclContext END;
SYMBOL DeclStmt    INHERITS DeclContext END;

RULE: Declaration ::= 'private' VarDecl COMPUTE
  Declaration.private = 1;
END;

RULE: Declaration ::= 'private' MethodDecl COMPUTE
  Declaration.private = 1;
END;

RULE: Declaration ::= 'private' ClassDecl COMPUTE
  Declaration.private = 1;
```

```
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

For every defining occurrence that might be modified by a `private` directive, we need to set the `IsPrivate` property of the `Key`. To avoid duplication, we will define the necessary computation in a class symbol `DeclaredName` and use LIDO inheritance:

Abstract syntax tree[134]==

```
SYMBOL VarDefName    INHERITS DeclaredName END;
SYMBOL MethodDefName INHERITS DeclaredName END;
SYMBOL ClassDefName  INHERITS DeclaredName END;

SYMBOL DeclaredName COMPUTE
  SYNT.GotDefKeyProp +=
    ResetIsPrivate(THIS.Key, INCLUDING DeclContext.private);
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Note the use of an accumulating computation for the void attribute `GotDefKeyProp` (see Section "Accumulating Computations" in *LIDO – Reference Manual*). This guarantees that the `IsPrivate` property is set before it can be used (see Section "Information access" in *Name Analysis Reference Manual*).

A computation at each declaration sets the `InPackage` property of the entity being declared to the key representing the package containing that declaration (see Chapter 3 [Libraries], page 31).

Abstract syntax tree[135]==

```
SYMBOL DeclaredName COMPUTE
  SYNT.GotDefKeyProp +=
    ResetInPackage(
      THIS.Key,
      INCLUDING (PackageBody.ScopeKey, Collection.ScopeKey));
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Four symbols in our abstract syntax represent applied occurrences. In every one of these four contexts we need to set the `InPackage` property of the `UseKey` to the key representing the package containing the applied occurrence. To avoid duplication, we define the necessary computation in a class symbol `AppliedName` and use LIDO inheritance:

Abstract syntax tree[136]==

```
SYMBOL AppliedName COMPUTE
  SYNT.GotUseKeyProp +=
```

```
    ResetInPackage(
      THIS.UseKey,
      INCLUDING (PackageBody.ScopeKey, Collection.ScopeKey));
END;

SYMBOL SimpleName       INHERITS AppliedName END;
SYMBOL QualifiedId      INHERITS AppliedName END;
SYMBOL SimpleWLName     INHERITS AppliedName END;
SYMBOL QualifiedWLId    INHERITS AppliedName END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The computations to set the `InClassNode` property of the keys for the bindings and applied occurrences are similar to the computations setting the `InPackage` property. In the case of the `InClassNode` property, however, both defining and applied occurrences can lie outside of class definitions.

Abstract syntax tree[137]==

```
SYMBOL DeclaredName COMPUTE
  SYNT.GotDefKeyProp +=
    ResetInClassNode(
      THIS.Key,
      INCLUDING (ClassBody.Env, Collection.Env));
END;

SYMBOL AppliedName COMPUTE
  SYNT.GotUseKeyProp +=
    ResetInClassNode(
      THIS.UseKey,
      INCLUDING (ClassBody.Env, Collection.Env));
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

The C functions must be combined and the global variable `pkg` provided:

AccCtl.c[138]==

```
#include "LangSpecFct.h"
#include "err.h"

DefTableKey pkg;

Check acceptability of a qualified identifier[128]
Call-back function to verify inheritance paths[131]
Check acceptability at the tip of a path edge[130]
```

This macro is attached to a product file.

`AccCtl.c` is a specification file:

Specification files[139]==

`AccCtl.c`

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%AccCtl > .
```

None of these files will have write permission in your current directory. You will need to add write permission in order to do the exercises.

1. Consider the role of the `DeclStmt` context in the NameLan language design.
   a. Do you think that it would be reasonable to use a `private` directive in this context? Explain briefly.
   b. If a `private` directive would never be used in the `DeclStmt` context, why do we bother to have `DeclStmt` inherit `DeclContext`? (Hint: try deleting that inheritance and building a processor.)
2. If a qualifier is the initial node, why is a binding found for the qualified identifier acceptable if the applied occurrence is in the same package as the defining occurrence?
3. Draw the scope graph for `accessPack.nl`, and highlight the inheritance path for the applied occurrence of `x` in method `g`.
4. Use `Bindings.specs` to generate a processor named `acc` that will show bindings for applied occurrences.
   a. Apply `acc` to `private.nl` and verify the bindings.
   b. Apply `acc` to `accessPack.nl` and verify the bindings.
5. Add the following class to package `P` of `accessPack.nl`:

```
class E {
  void i() { Q.F.x = A.y; }
}
```

   a. List the available bindings for the applied occurrence `x` in method `i`. Can this applied occurrence identify any of these bindings? Explain your answer using the scope rules of NameLan.
   b. Which of the `isAcceptable` functions will be called by the generic lookup for each of the bindings in (a)? Explain briefly.
   c. Briefly explain how the `isAcceptable` function reaches its decision in each case.
   d. Apply `acc` to the modified file and verify the predicted behavior.

6.2 Position control

A method body or block consists of a sequence of statements and variable declarations. The rules of NameLan only allow a reference to a variable declared in such a sequence to take

the form of a simple name; it cannot be inherited or used in a qualified name. This means that there is no mechanism by which the variable can be accessed outside of the method body or block in which it was declared. We therefore call these variables *local* variables.

Local variable declarations may appear at arbitrary locations in a method body or block. How should we interpret their meaning in relation to each other and to the statements in that method body or block? The programmer might well consider the variable name to be unknown before its declaration. As language designers, we could formalize that intuition:

- The scope of a defining occurrence `VarDefName` in a `DeclStmt` phrase begins at the `VarDefName` and ends at the end of the smallest enclosing `Block`.
- No two defining occurrences `VarDefName` of the same identifier may have scopes that end at the same point.

Note that the first rule applies only to the special case of a local variable. All other `VarDefName` occurrences still obey the scope rules stated in earlier chapters. The second rule applies to *all* `VarDefName` occurrences. It prevents multiple variable declarations in a range (see Section 1.4 [Error reporting], page 12).

If a scope ends at the end of a `Block` phrase, the smallest abstract syntax subtree encompassing that scope is the corresponding `Block` subtree. That subtree can be chosen as the range of the scope, just as it would be the range of a `VarDefName` scope defined by the rules of the kernel language (see Section "Basic Scope Rules" in *Name analysis according to scope rules*). This means that the attribute computation seeking an available binding is used for all applied occurrences in the `Block` subtree. However, if the available binding is for a local variable, then it may not be acceptable due to the position of the applied occurrence. Here's a trivial example to show how the developer can use position control to implement our new scope rule:

cscope.nl[140]==

```
int i;
{ i = -42;
  float i;
  i = 42;
}
```

This macro is attached to a non-product file.

There are two variables named `i` in `cscope.nl`. The scope of the floating-point variable declared on the third line begins at the end of that line, and ends at the closing brace on the fifth line. Thus the applied occurrence on the second line does not have the same meaning as the applied occurrence on the fourth line.

The search for a binding for the applied occurrence on the second line begins in the scope graph node for the block. That scope graph node contains a defining occurrence on the third line, which must be checked for acceptability. This binding is not acceptable because the applied occurrence does not follow the defining occurrence in the text, and is therefore not in the scope of that defining occurrence. The search must continue in the scope node for the complete text, where an acceptable binding is available.

The search for a binding for the applied occurrence on the fourth line also begins in the scope graph node for the program block. In this case the binding *is* acceptable, because the applied occurrence follows the defining occurrence in the text and is therefore in the scope of that defining occurrence.

In order to compare positions of defining and applied occurrences, we use values of type `CoordPtr` (see Section "Source Text Coordinates and Error Reporting" in *The Eli Library*). By default, the `CoordPtr` value specifies the text line number and column index at which the phrase begins.

Module computations set the `Coord` property of every `IdDefScope.Key` attribute to the `CoordPtr` value for the phrase rooted in the `IdDefScope` node (see Section "Defining Occurrences" in *Name Analysis Reference Manual*). Similar module computations set the `Coord` property of every `UseKey` to the `CoordPtr` value for its phrase (see Section "Applied Occurrences" in *Name Analysis Reference Manual*). We can override the default version of `isAcceptableSimple` with a version that applies the function `earlier` to the `Coord` properties of the keys for the defining and applied occurrences of a local variable to check whether the binding found by the generic lookup is acceptable:

PosCtl.c[141]==

```
#include "LangSpecFct.h"

int
isAcceptableSimple (DefTableKey def, DefTableKey app)
{ if (GetIsLocal(def, 0)) {
    CoordPtr defcoord = GetCoord(def, NoPosition);
    CoordPtr appcoord = GetCoord(app, NoPosition);
    if (!earlier(defcoord, appcoord)) return IgnoreContinue;
  }
  return AcceptBinding;
}
```

This macro is attached to a product file.

See Section "Source Text Coordinates and Error Reporting" in *The Eli Library*, for the details of the `earlier` function.

`PosCtl.c` is an additional specification:

Specification files[142]==

```
PosCtl.c
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

`IsLocal` is a property indicating that the entity is a local variable (see Section "The Definition Table Module" in *Property definition language*).

Properties and property computations[143]==

```
IsLocal: int;
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

Here are computations that will establish the value of the `IsLocal` property of the key of a local variable:

Abstract syntax tree[144]==

```
SYMBOL DeclContext: IsLocal: int;
```

```
SYMBOL Declaration COMPUTE SYNT.IsLocal=0; END;
SYMBOL DeclStmt    COMPUTE SYNT.IsLocal=1; END;

SYMBOL VarDefName COMPUTE
  SYNT.GotDefKeyProp +=
    ResetIsLocal(THIS.Key, INCLUDING DeclContext.IsLocal);
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%PosCtl > .
```

None of these files will have write permission in your current directory.

1. Use file `Bindings.specs` to generate a processor named `loc` that will show bindings of applied occurrences, and apply it to `cscope.nl`, verifying that the bindings for the applied occurrences are correct in both cases.
2. Program `cscope.nl` shows that, if a language allows the scope of a local variable to be a part of a block, it is possible for two applied occurrences of a particular variable identifier to have different meanings within that block. Many language designers consider this to be bad programming style. Suppose that you decide to make processor `loc` issue a message about this example of bad programming style (see Section "Error Reporting" in *Frame Library Reference Manual*).
   a. What severity would you use for that message? Explain briefly.
   b. Alter the specifications given in this section to issue such a message. Generate a new processor, `rpt`, from your modified specification. Test `rpt` on `cscope.nl`.
   c. Delete the first line of `cscope.nl` and apply `rpt` to the resulting program. Is the result what you expected? Do you think that the result is reasonable? Explain briefly.
3. If you were not satisfied with `rpt`'s error reporting, carefully consider the placement of the `message` call. Try to improve the result by saving some information in `isAcceptableSimple` and using it as a condition for invoking `message` in `LookupComplete` (see Section "Initialization and finalization" in *Name Analysis Reference Manual*).

   Test your improvements on both versions of `cscope.nl`.

7 Predefined Identifiers

Most programming languages use pre-defined identifiers to represent a few basic entities. For example, it would be useful for NameLan to have a pre-defined class `Object`. The effect would be as though the following class had a defining occurrence in the range of the compilation:

```
class Object {
  int hashCode() { }
}
```

`Object` would act as the direct superclass of any class that did not specify a direct superclass. The result would be that `Object` is a superclass of every class, and all classes inherit the entities declared in `Object` unless those entities are hidden. A variable of type `Object` could hold a reference to an object of any class. Here is a program that uses these properties:

object.nl[145]==

```
class C {
  int h() { hashCode(); }
}

class D { }

{ C c; c.hashCode();
  D d; d.hashCode();
  Object o; o.hashCode();
}
```

This macro is attached to a non-product file.

This chapter explains how to extend NameLan with a pre-defined `Object` superclass.

Pre-defined identifiers have no actual defining occurrences in the program text, so their bindings must be created by the generated processor. Eli provides a module (`SGPreDefId`) to support specification of pre-defined identifiers (see Section "Pre-defined Identifiers" in *Name Analysis Reference Manual*).

The developer needs to instantiate `SGPreDefId` with the instance parameter that was used in the instantiation of the `ScopeGraphs` module in which the pre-definitions must be implemented. The scope rules for both the pre-defined class `Object` and the entities declared in its body are implemented by the set of four isomporphic scope graphs discussed in Section 5.1 (see Section 5.1 [Reusing identifiers in the same scope], page 55). Those scope graphs are supported by our first instantiation of the `ScopeGraphs` module (see Section "Basic Scope Rules" in *Name analysis according to scope rules*). That instantiation carried no instance parameter, so we also instantiate the `SGPreDefId` module with no instance parameter:

Specification files[146]==

```
$/Name/SGPreDefId.gnrc  +referto=(Predef.d)     :inst
$/Tech/MakeName.gnrc    +instance=Ident         :inst
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

We use the operations provided by `SGPreDefId` to pre-define two identifiers, `Object` and `hashCode`, and the scope graph node tuple owned by the class `Object`. Each predefinition is established by an appropriate macro call (see Section "Pre-defined Identifiers" in *Name Analysis Reference Manual*).

Object The identifier `Object` is bound to the known key `ObjectKey` in the root environment of the scope graph for classes:

Scope graph node
The `NodeTuplePtr` value defining the environment of the class `Object` is created and assigned to `ObjectEnv`:

hashCode The identifier `hashCode` is bound to the known key `hashCodeKey` in the `ObjectEnv` environment of the scope graph for methods:

The three macro calls appear in the file whose name was given by the `SGPreDefId` instantiation's `referto` parameter:

Predef.d[147]==

```
PreDefKeyNdx ("Object", ObjectKey, classGraph)
PreDefNode(ObjectKey, ObjectEnv)
PreDefKeyEnvNdx("hashCode", hashCodeKey, ObjectEnv, methodGraph)
```

This macro is attached to a non-product file.

The developer must define `ObjectKey` and `hashCodeKey` as known keys, with appropriate properties set (see Section "How to declare properties" in *Property definition language*).

Properties and property computations[148]==

```
"ScopeGraphs.h"
ObjectKey       -> GraphIndex={classGraph}, IsType={1};
hashCodeKey     -> GraphIndex={methodGraph};
```

This macro is defined in definitions 79, 94, 126, 132, 143, and 148.
This macro is invoked in definition 80.

See Chapter 5 [Multiple Scope Graphs], page 55, for the `GraphIndex` property. See Section 4.1 [Connect to the Typing module], page 44, for the `IsType` property.

`ObjectEnv` is a `NodeTuplePtr`-valued variable that will hold the generated node tuple owned by the class `Object` (see Section "Pre-defined Identifiers" in *Name Analysis Reference Manual*). The developer must establish this variable in a C file and provide an appropriate header file:

Predef.c[149]==

```
#include "Predef.h"
NodeTuplePtr ObjectEnv; /* Establish the variable ObjectEnv */
```

This macro is attached to a product file.

Predef.h[150]==

```
#ifndef PREDEF_H        /* Prevent multiple inclusions */
#define PREDEF_H
#include "Model.h"      /* Define the NodeTuplePtr type */
extern NodeTuplePtr ObjectEnv;
#endif
```

This macro is attached to a product file.

`ObjectEnv` will be referred to in generated code, so the developer needs to ensure that `Predef.h` is included in that generated code (see Section "Implementing Tree Computations" in *LIDO – Computations in Trees*).

Predef.head[151]==

```
#include "Predef.h"
```

This macro is attached to a product file.

These three files are all additional specifications:

Specification files[152]==

```
Predef.head
Predef.h
Predef.c
```

This macro is defined in definitions 12, 16, 23, 27, 74, 78, 81, 91, 117, 121, 139, 142, 146, and 152.
This macro is invoked in definition 13.

This completes the pre-definition of `Object`. We now need to establish path edges to `Object` from each class using the default inheritance (see Chapter 2 [Classes], page 19). The relevant abstract syntax is:

```
RULE: Inheritance ::= Default   END;
RULE: Default     ::=           END;
```

The computation is close to that for specific super classes (see Section 2.2 [Inheritance], page 24). It uses the `BoundEdge` role instead of the `WLCreateEdge` role because the edge tip is known (see Section "Path edge creation roles" in *Name Analysis Reference Manual*).

Abstract syntax tree[153]==

```
SYMBOL Default INHERITS BoundEdge COMPUTE
  SYNT.tailEnv = INCLUDING Inheritance.SubClassEnv;
  SYNT.tipEnv = ObjectEnv;
END;
```

This macro is defined in definitions 10, 18, 22, 38, 47, 52, 53, 65, 69, 71, 88, 89, 92, 93, 95, 96, 101, 105, 108, 110, 111, 112, 113, 122, 133, 134, 135, 136, 137, 144, and 153.
This macro is invoked in definition 11.

Exercises

These exercises are based on files defined in the Tutorial. To obtain copies of those files in your current directory, enter Eli and give the following command:

```
-> $elipkg/Name/LearnSG%Predef > .
```

None of these files will have write permission in your current directory.

1. Draw the scope graph that will be created for `object.nl`. Which nodes and edges were created by the pre-definition module?
2. Use file `Bindings.specs` to generate a processor named `pd` that will show bindings for applied occurrences, and apply it to `object.nl`. Is the result what you expected?
3. Why does `pd` not report applied occurrences of the predefined identifiers `Object` and `hashCode`?

8 Index

P

Q

R

S

T

W