Semantic Analysis Cookbook

Part 1: Declarations

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Semantic Analysis Cookbook - Part 1: Declarations

Josef Grosch

Oct. 14, 1992

Document No. 29

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Semantic Analysis Cookbook
Part 1: Declarations

1. Introduction

A compiler for a programming language usually has two parts: an analysis part and a synthesis part. The analysis part comprises lexical analysis, syntax analysis, and semantic analysis. The main task of the synthesis part is code-generation. For several years I am attracted by the question: "How to program semantic analysis?" I have asked this question to many people, but except the remark "a good question" I did not receive a good answer so far. In text books on compiler construction I never found a satisfying answer, either.

What do I understand by the notion semantic analysis? The analysis part of a language processor usually consists of two phases: A context-free analysis phase and a context-sensitive analysis phase. The context-free phase is called syntax analysis or parsing and the context-sensitive phase is called semantic analysis. The latter comprises the check of context conditions and the computation of so-called code-generation attributes. These attributes are values to be incorporated into an intermediate representation or the machine code to be generated by the language processor. The generation of an intermediate representation is not considered to be part of semantic analysis. Typical tasks of semantic analysis are name (or declaration) analysis and type checking.

From a logical point of view, the two phases of a language processor support the following scheme: Syntax analysis reconstructs a derivation tree for a program to be processed. Semantic analysis checks conditions in this tree and computes additional information. There are several possibilities to implement this general scheme:

First, Syntax analysis and semantic analysis can be performed at the same time in an interlocked fashion. There is no need to store the program between the two phases in an intermediate data structure. This approach is restricted to one-pass languages such as for example Pascal. Pascal follows the *declare before use* design principle in order to enable this kind of implementation. There are two constructs which are not really one-pass: The use of the reference type in the declaration of pointer types before its declaration and forward gotos. These constructs can be integrated into a one-pass scheme using small additional data structures. Often LL(1) parser generators with an L-attribution mechanism are used to support this kind of implementation. Semantic analysis is programmed by hand and attached to corresponding grammar rules.

Second, Syntax analysis and semantic analysis can be performed one after the other. This requires to store the program in an appropriate data structure such as a syntax tree. This approach removes the restriction to one-pass languages. Often LL(1) or LR(1) parser generators are used to carry out syntax analysis. The parsers execute action statements to build a syntax tree. Semantic analysis operates on the syntax tree. It can be programmed by hand, generated from an attribute grammar using an attribute evaluator generator tool, or constructed by making use of a pattern-matching tool.

The implementation schemes mentioned represent more or less two extreme positions. Many other approaches are possible. One example are multipass compilers where semantic analysis is separated into several passes which are executed sequentially. Every pass communicates a representation of the program including the current results to the next pass. This representation could be a linearized version of the syntax tree which is stored on disk for example.

I will neither discuss all possible solutions to the problem of semantic analysis nor give an objective solution. Instead, I will give a subjective answer which is influenced by my personal background, my personal experience in compiler writing, and the tools for compiler construction that I prefer [GrE90]. I assume the reader has some familiarity with these tools for compiler construction because the examples will be written in the input languages of these tools. I must admit that I am answering the question "How to program semantic analysis?" primarily for myself. If other people find my solution useful then all the better. The goal is to look for a systematic method that allows for an easy implementation of semantic analysis in a rather short time.

The basic idea is to give a collection of recipes or sample solutions for typical tasks in semantic analysis. The collection shall cover all tasks necessary for typical imperative programming languages such as Pascal, Modula-2, and C. It is important to have solutions for the different tasks that are as independent of one another as possible. This is a precondition to support the construction of a semantic analyzer by unrestricted combination of several partial solutions. If there are tasks or solutions that appear often they should be provided in a reusable fashion. Many tasks in semantic analysis are similar or even identical for different languages. This suggests to look for solutions that are independent of any programming language in particular.

In the following, semantic analysis will be based on the second implementation scheme mentioned above. A scanner and a parser construct a syntax tree. Semantic analysis operates on this tree by computing attributes and by checking context conditions. This approach allows for the formulation of all problems in semantic analysis. It avoids the restrictions of one-pass schemes, because a tree can be processed in as many passes as necessary and the subtrees of a node can be visited in any order. Nowadays, the storage consumption for trees is not a problem any more because enough main memory is available.

I will use an abstract syntax tree instead of a concrete syntax tree. I consider an abstract syntax not to be just a "small" abstraction of a concrete syntax. I prefer to say that abstract syntax has nothing to do with concrete syntax. It is a representation of the program that eases semantic analysis. Therefore it should be a small and simple representation. Of course, this statement is an exaggeration. The abstract syntax does have similarities with the concrete one, because it represents the same information. I am using this formulation in order to say that we should not stick too close to concrete syntax. My understanding of abstract syntax may appear a bit extreme. Here is not the space to argue for this opinion.

Currently, I am thinking about the idea of a so-called *normalized* abstract syntax. This notion describes an even simpler form of abstract syntax than I am using now. This idea is probably illustrated best by an example using the source level. The three declarations

```
VAR x, y: INTEGER;

VAR x: INTEGER; y: INTEGER;

VAR x: INTEGER; VAR y: INTEGER;
```

would be represented in a normalized abstract syntax in the same way. The tree representation would reflect the structure of the last pair of declarations. My current style of abstract syntax also introduces as many normalizations as possible. The difference can be seen in the way an abstract syntax tree is constructed. The current mapping of concrete syntax to abstract syntax is a syntax-directed translation which is performed by the tree building action statements executed by a parser. For the construction of a normalized abstract syntax more than a syntax-directed translation is needed. It would either require small transformations triggered by the parser or even a complete transformation phase in between syntax analysis and semantic analysis.

1.1. Possible Implementation Techniques

As already mentioned, there are primarily three ways to implement computations on a tree: One could use hand-written code, an attribute evaluator generated from an attribute grammar, or a pattern-matching tool. I will briefly characterize these possibilities and discuss the advantages and drawbacks of these approaches.

Using hand-written code one can program an arbitrary number of tree traversals that visit the subtrees of a node in any order and execute appropriate computations. One has explicit control on the order of the computations, can use global variables, and can safely produce any side-effects. A disadvantage is that one has to worry about the order of the traversals and computations. There are no checks whether the set of computations is complete.

While hand-written code supports an imperative style of programming, attribute grammars [Knu68, Knu71] advocate a functional style. They promise to overcome the problems of hand-written code by providing formal checks and by automatic determination of an appropriate evaluation order. An attribute grammar is checked for the correct use of synthesized and inherited attributes, whether there are computation rules for all attributes (completeness), and whether there are no cyclic dependencies. While these checks are certainly valuable, the determination of an evaluation order does not relieve of the burden to worry about this order. In larger applications there are often cyclic dependencies among the attributes if the attribute grammar is processed by a tool that handles the class of ordered attribute grammars [Kas80]. In order to remove those cyclic dependencies it has been necessary to worry quite a lot about evaluation order and to introduce additional attributes along with additional computations. These steps solve the problems with the cycles but lead to rather complicated attribute grammars. Those grammars degrade from formal problem specifications to problem implementations which are hard to understand and to maintain.

The use of a pattern-matching tool is in my eyes a gradual improvement of writing code by hand. The underlying mechanism is again the recursive traversal of trees with explicit control of the number of passes and the order of the visits to the subtrees of a node. Pattern-matching can be seen as a comfortable way of writing conditional statements or branch statements, respectively. The advantage of this approach is the provision of a concise notation for the formulation of tree processing problems. However, the fundamental problems of writing code by hand remain. One still has to worry about the order of the traversals and computations and there is no check for the completeness of a specification.

1.2. Proposal

From the drawbacks of the above implementation techniques, I conclude that substantial improvements are necessary. It is certainly of value to have a formal consistency check for the problem specification and to be freed from the determination of the order of the computations. Therefore my approach is based on attribute grammars. Attribute evaluation is done on abstract syntax trees and perhaps on normalized abstract syntax trees. I propose to use the class of well-defined attribute grammars, which is the largest class of attribute grammars. This class promises to solve the problems of the evaluation order. Smaller classes such as ordered attribute grammars are considered to be too restrictive because one has still to bother with evaluation order. Evaluators for well-defined attribute grammars are not as efficient as those for e. g. ordered attribute grammars. For ordered attribute grammars the evaluation order (visit sequence) can be determined statically during the generation of the evaluator. For well-defined attribute grammars the evaluation order must be determined dynamically during the run time of the evaluator. I decided this trade-off between expressive power and efficiency in favor of expressive power. Attribute grammar tools are available that can process well-defined attribute grammars [Groa]. Compared to ordered attribute grammars the increased consumption of memory and run time is relatively small and tolerable with today's hardware.

Furthermore, I will use higher-order attribute grammars [VSK89, Vog93] which can handle tree-valued attributes and that allow the underlying tree to grow dynamically during attribute evaluation. The attribution of trees will be extended to the attribution of graphs. While conventional attribute grammars base computations only on attributes local to a grammar rule, I will allow the access of non-local attributes. The processing of graphs and access to non-local attributes is of interest for handling a definition table which is discussed below. In order to get an automatic determination of an evaluation order, the computations have to be expressed in a functional style. That means all arguments of a computation must be explicitly mentioned and an attribute can not depend on itself. If possible, I will combine techniques for attribute grammars with those of pattern-matching to perform semantic analysis. In every case a pattern-matching tool can be used to describe attribute computations, especially those based on tree-valued attributes.

One of the main problems of semantic analysis is the administration of a definition table. Many people prefer to call this data structure symbol table. It describes the objects of a program by appropriate attributes and holds information about the so-called scopes. I propose not to construct a separate data structure for the definition table but to use certain parts of the abstract syntax tree for that purpose. This has the advantage that the computation of the attributes for the definition table is the same as any other attribute computation. The definition table can be regarded as some kind of spread sheet. Rules describe the relationship among the definition table entries (attributes) in a functional style. The existing attribute evaluator takes care that all entries are computed and it determines automatically an order to achieve this.

2. Method

The main contribution of this work is to present a collection of recipes or sample solutions for typical tasks or problems in semantic analysis. I will start with solutions for name analysis or declaration analysis. For every problem a small language processor or compiler has been implemented in order to provide a clear and simple prototype solution. The compilers have been generated using the Cocktail Toolbox for Compiler Construction [GrE90]. The input specifications for the tools will be reproduced and discussed in the following. The sample compilers use C as implementation language.

All solutions have some parts in common while other parts are specific to the individual problems. The common parts comprise the scanner, the main program, the definition table, and the "make" file. These are explained in chapter 3. The individual parts comprise a parser, an abstract syntax, and an attribute grammar.

The solutions for the separate problems are described in a schematic way using the following structure:

Description
Conditions
Remarks
Concrete Syntax
Attribute Grammar
Attribute Diagram
Input
Messages

It starts with a description of the task or the problem to be solved. This may include a small informal definition of the used sample language. It is followed by a list of context conditions that have to be checked for this problem. Additional remarks might be used to comment any peculiarities. The section "concrete syntax" contains the input for the parser generator *lalr* [GrV]. This includes the context-free grammar that defines the concrete syntax of the sample language and the tree building action statements that map concrete to abstract syntax. The section "attribute grammar" contains the input for the attribute evaluator generator *ag* [Groa, Groa]. It defines the abstract syntax, the necessary attributes, the attribute computation rules, and checks for the context conditions. It is followed by so-called *attribute diagrams*. These diagrams support the visualization of the problem solution by giving a graphical representation of a sample attributed tree. The used formalism is described below. The description of a problem is terminated by a sample input and the corresponding messages resulting from the execution of the language processor.

2.1. Attribute Diagram

An attribute grammar is a generative system for a set of attributed trees. It is a textual specification which is oriented towards grammar rules or node types of a tree, respectively. A rule along with the associated attribute definitions and attribute computations describes the relationships among the attributes local to this rule. These small, local building blocks have to be used to implement algorithms that concern larger tree parts such as the distribution of information over the tree and the check of conditions between distant attributes. The global relations in an attributed tree result implicitly from the application of the attribute grammar mechanism to the set of rules.

An attribute diagram is a graphical representation of an attributed tree. Its purpose is to overcome the drawbacks of attribute grammars that arise from the textual representation and the local, rule oriented view. An attribute diagram is a visual representation of a typical attributed tree that shows the data flow among the attributes and their global relationship.

The following symbols are used in attribute diagrams. Circles denote tree nodes. They are marked with the node type which is eventually abbreviated. Sometimes circles are used for additional data structures such as nodes of type *Env* describing scopes. Rectangles denote attributes. They are attached to tree nodes and bear an abbreviation of their name. The Appendices 1 and 2 explain all used abbreviations. Arrows serve for two purposes. Arrows from circles to circles refer from parent nodes to child nodes. Arrows from rectangles to rectangles or to circles represent attribute values (or dependencies) of tree-valued attributes. These attribute values are pointers referring to tree nodes.

The left attribute diagram of the first illustration in chapter 4 (see page 14) shows a simple list of declarations. There are two kinds of links between the list elements. The arrows among the circles represent the abstract syntax tree. The arrows from the threaded attribute *Objects* (actually the attribute pair *ObjectsIn* and *ObjectsOut* which are abbreviated as *OI* and *OO*) represent a set of objects as building block for a definition table. For a description of threaded attributes see the manual of the attribute evaluator *ag* [Groa]. The first attribute diagram shows the actual targets of the pointers. An abstraction for threaded attributes is depicted in the second attribute diagram given in the middle. This abstraction reflects more the dependencies among the attributes and their corresponding evaluation order. A further abstraction can be found in the right attribute diagram. Unessential details such as the intrinsic attributes *Ident* (abbreviated as *Id*) and *Pos* are omitted. Sometimes, a layout that looks like a chain of attributes will be used. It means that all attributes point to the last element of this chain. The use of this construct will be clear from the context.

3. Common Parts

This section presents the parts common to all the following problem solutions: the scanner, the main program, the definition table, and the "make" file.

3.1. Scanner

The scanners are quite simple and recognize identifiers and fixed tokens such as keywords and delimiters. They are generated from the specification given below using the scanner generator *rex* [Grob]. The specification describes essentially only identifiers. The set of fixed tokens is problem specific. It is extracted automatically out of the concrete syntax and inserted in place of the line starting with INSERT RULES [Groc].

```
EXPORT  {
# include "Idents.h"

INSERT tPosition
INSERT tScanAttribute
}

GLOBAL  {
INSERT ErrorAttribute
}

DEFAULT {
   (void) fprintf (stderr, "%3d, %2d: Error       illegal character: %c\n",
      Attribute.Position.Line, Attribute.Position.Column, * TokenPtr);
}

DEFINE  digit   = {0-9} .
        letter  = {a-z A-Z} .

RULES

INSERT RULES #STD#

#STD# letter (letter | digit) * : {
   Attribute.Ident.Ident = MakeIdent (TokenPtr, TokenLength);
   return Ident;
}
```

3.2. Main Program

The main program given below is rather simple. It mainly initiates the two phases of the language processors: The function *Parse* implements the syntax analysis phase comprising scanning, parsing, and the construction of the abstract syntax tree. The procedure *Eval* implements the semantic analysis phase using an attribute evaluator. The calls of the procedures *StoreMessages* and *WriteMessages* instruct the error handling module to emit a sorted list of error messages.

```
# include "Errors.h"
# include "Parser.h"
# include "Tree.h"
# include "Eval.h"

main ()
{
   StoreMessages (rtrue);
   (void) Parser ();
   BeginEval ();
   Eval (TreeRoot);
   WriteMessages (stderr);
   return 0;
}
```

3.3. Definition Table

The definition table, also called symbol table, is a data structure that is used to describe the objects appearing in a program. This section introduces the notions necessary for this area and briefly sketches a simple implementation.

A *definition table* consists of an ordered set of scopes. *Scopes* are ranges of text in a program such as nested blocks or nested procedures that regulate the visibility and the extent of identifiers and objects. A scope is associated with a set of objects. Sometimes those sets have to be ordered for example in the case of the fields of a record type. *Objects* are the entities appearing in a program such as variables, named constants, procedures, and types. An object is described by a set of appropriate *attributes* that hold suitable values. Every object has at least one attribute describing its identifier. Other attributes are for example the kind of the object (variable, constant, procedure, etc.), the type of a variable, or the value associated with a named constant.

An object appears in a program in two ways: It is either declared or used. A declaration leads to an entry in the definition table. A declaration may be either explicit or implicit in languages such as Fortran or Basic. In general, a declaration does not contain directly all the information needed to complete a description of an object. It might be necessary to consult other entries in the definition table or other parts of the syntax tree, for example in order to compute the value of a named constant.

At the location where an object is used, only its identifier is known, usually. One task of semantic analysis is to determine the object that is denoted by this identifier. In general, this relation is ambiguous at the first view and there are several strategies to unambiguously determine an object depending on the language definition.

In case of nested blocks the object in the innermost block that surrounds the using location is selected. There might be different name spaces for different kinds of objects such as for example variables, types, and labels. If at the using location the kind of the object is known then this information can be used to select an appropriate object. The full power of overloading resolution is used if the signature of a function is consulted in order to unambiguously select an object.

The data structure of a definition table will be implemented as follows: The entries for the object descriptions are the nodes of the abstract syntax tree standing for declarations. The attributes of these descriptions are treated as any other attributes. A set of objects is represented as a linked list of tree nodes. The links are stored in addition to the links already present to implement the tree structure because the tree structure and the set of objects do not correspond in general. A scope is represented by a special node which refers to a set of objects and to a surrounding scope. This node is of a type named *Env*, which is short for *environment*. I will use the notion environment for the part of a definition table that is valid at a certain program location. The set of all nodes representing scopes constitutes the definition table. From the implementation point of view the definition table is not a data structure that is stored separately from the abstract syntax tree. It is spread over the abstract syntax tree. However, from the logical point of view it can be regarded as an abstract data type with certain operations and a hidden implementation.

This implementation of a definition table does not worry about efficiency. It tries to be as simple and clear as possible. Once definition table entries are created their storage space is never released during run time. This is necessary because of the functional nature of the computations and to provide a great deal of expressive power. It has been observed that one can afford this because of today's main memory capacities. I do not follow the stack model of scopes which worries too much about saving storage and it is oriented too much towards one-pass translations. I also do not bother with any kind of garbage collection.

Linked lists imply that the search for an object needs linear time. This might be too slow for large programs with many declared objects. In this case the linked list data structure can be overlaid by a hash table yielding constant time access. The implementation is encapsulated in the definition table module and a user of this module does not have to take care about the underlying internal data structure.

The user of the definition table needs to know the basic structure of the data representation and the available operations. As has been already mentioned, the data structure of the definition table is integrated with the abstract syntax. The implementation of both is supported by the generator for abstract syntax trees *ast* [Groa]. The definition table provides abstract objects which have two attributes: an identifier and a source position. These abstract objects are described by the following excerpt of the specification of an abstract syntax written in the notation of *ast*.

```
Decls          = <
   NoDecl      = .
   Decl        = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var      = ...
      Const    = ...
      Proc     = ...
      RefDecl  = Decl .
   > .
> .
```

The node type *Decls* describes lists of objects, *NoDecl* describes an empty list, and *Decl* describes an abstract object. The child called *Next* of the latter refers to the succeeding list element with respect to the abstract syntax. Concrete objects types such as variable, constant, and procedure can be derived using the extension mechanism of *ast* [Gro90, Grod]. This is equivalent to the definition of subclasses or subtypes in object-oriented languages. The node type *Decls* is augmented by a threaded attribute called *Objects*:

```
Decls = [Objects: tTree THREAD] .
```

This definition actually introduces a pair of attributes called *ObjectsIn* and *ObjectsOut*. Those attributes are used for linking objects into sets.

The node type *RefDecl* is not a concrete object type. It refers to another declaration and it is used for internal purposes of the definition table. Sometimes an object is element of more than one set of objects. In this case it can be linked directly into one set, only. The other sets use nodes of type *RefDecl* that refer to the one real object descriptor.

The node type *Env* describes scopes. From the logical point of view it has two children that refer to a set of objects and a surrounding environment:

```
Env   = Decls IN Env IN .
```

The actual implementation uses the following definition:

```
Env   = [Objects: tTree IN] Env IN .
```

The fundamental operation of the definition table is a lookup function called *Identify*. Its basic signature is:

```
Identify :  Ident × Env → Object
```

or

```
Identify : tIdent × Env → { Decl, NoDecl }
```

Given an identifier and an environment it will search the environment for an object with this identifier. If such an object is found it returns the corresponding object description which is a node of type *Decl*. Otherwise it returns a node of type *NoDecl*.

In reality there exists a set of routines with varying functionality:

```
IdentifyObjects   : Ident × Objects          → Object
IdentifyLocal     : Ident × Env              → Object
IdentifyWhole     : Ident × Env              → Object
IdentifyTail      : Ident × Env × Pos        → Object
IdentifyLocalKind : Ident × Env × Kind       → Object
IdentifyWholeKind : Ident × Env × Kind       → Object
IdentifyTailKind  : Ident × Env × Pos × Kind → Object
```

IdentifyObjects takes an identifier and a set of objects as arguments and it searches only in this set of objects. *IdentifyLocal* takes an environment as argument but it searches only in the innermost scope and disregards all surrounding scopes. *IdentifyWhole* corresponds to the function Identify that has been used above to introduce the principles of the definition table operations. *IdentifyTail* works like IdentifyWhole with the difference, that the source position of the object returned must lie before the source position of the using location which is given by the additional argument named *Pos*. This routine is necessary to describe the *declare before use* principle. *IdentifyLocalKind*, *IdentifyWholeKind*, and *IdentifyTailKind* are similar as the previous routines. They have an additional argument named *Kind* that specifies the kind of the object to be searched for. While the routines without the syllable Kind in their name consider all objects to reside in one name space, the others can be used to handle separate name spaces. The argument *Kind* is an integer value that may refer to a single node type or to a class comprising several node types. In the actual implementation the mentioned routines have the following headers:

```
   tTree IdentifyObjects    (Ident, Objects)
   tTree IdentifyLocal      (Ident, Env)
   tTree IdentifyWhole      (Ident, Env)
   tTree IdentifyTail       (Ident, Env, Pos)
   tTree IdentifyLocalKind  (Ident, Env, Kind)
   tTree IdentifyWholeKind  (Ident, Env, Kind)
   tTree IdentifyTailKind   (Ident, Env, Pos, Kind)

   tIdent      Ident;
   tTree       Objects;
   tTree       Env;
   tPosition   Pos;
   Tree_tKind  Kind;
```

The definition table provides two auxiliary macros which are commonly needed for semantic analysis:

```
   IsBefore (Pos1, Pos2)
   Error (Pos, Text, Ident)
```

The macro *IsBefore* tests two source positions with respect to their order. *Error* calls a routine to report errors found during semantic analysis.

The following is the implementation of the definition table module:

```
# include "Position.h"
# include "Errors.h"

# define IsBefore(Pos1, Pos2) (Compare (Pos1, Pos2) < 0)
# define Error(Pos, Text, Ident) MessageI (Text, xxError, Pos, xxIdent, \
                                          (char *) & Ident)

static tTree IdentifyObjects (Ident, Objects)
   register tIdent      Ident;
   register tTree       Objects;
{
   while (Objects->Kind != kNoDecl)
      if (Objects->Decl.Ident == Ident)
         return Objects->Kind == kRefDecl ? Objects->RefDecl.Decl : Objects;
      else Objects = Objects->Decl.ObjectsIn;
   return mNoDecl ();
}

static tTree IdentifyLocal (Ident, Env)
   register tIdent      Ident;
   register tTree       Env;
{
   if (Env != NoTree) {
      register tTree Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident)
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
   }
   return mNoDecl ();
}

static tTree IdentifyWhole (Ident, Env)
   register tIdent      Ident;
   register tTree       Env;
{
   while (Env != NoTree) {
      register tTree Object;
      Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident)
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
      Env = Env->Env.Env;
   }
   return mNoDecl ();
}

static tTree IdentifyTail (Ident, Env, Pos)
   register tIdent      Ident;
   register tTree       Env;
   tPosition            Pos;
{
   while (Env != NoTree) {
      register tTree Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident && IsBefore (Object->Decl.Pos, Pos))
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
      Env = Env->Env.Env;
   }
   return mNoDecl ();
}

static tTree IdentifyLocalKind (Ident, Env, Kind)
   register tIdent      Ident;
   register tTree       Env;
   register Tree_tKind  Kind;
{
   if (Env != NoTree) {
      register tTree Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident && (Tree_IsType (Object, Kind) ||
            Object->Kind == kRefDecl && Tree_IsType (Object->RefDecl.Decl, Kind)))
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
   }
   return mNoDecl ();
}

static tTree IdentifyWholeKind (Ident, Env, Kind)
   register tIdent      Ident;
   register tTree       Env;
   register Tree_tKind  Kind;
{
   while (Env != NoTree) {
      register tTree Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident && (Tree_IsType (Object, Kind) ||
            Object->Kind == kRefDecl && Tree_IsType (Object->RefDecl.Decl, Kind)))
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
      Env = Env->Env.Env;
   }
   return mNoDecl ();
}

static tTree IdentifyTailKind (Ident, Env, Pos, Kind)
   register tIdent      Ident;
   register tTree       Env;
   tPosition            Pos;
   register Tree_tKind  Kind;
{
   while (Env != NoTree) {
      register tTree Object = Env->Env.Objects;
      while (Object->Kind != kNoDecl)
         if (Object->Decl.Ident == Ident && (Tree_IsType (Object, Kind) ||
            Object->Kind == kRefDecl && Tree_IsType (Object->RefDecl.Decl, Kind)) &&
            IsBefore (Object->Decl.Pos, Pos))
            return Object->Kind == kRefDecl ? Object->RefDecl.Decl : Object;
         else Object = Object->Decl.ObjectsIn;
      Env = Env->Env.Env;
   }
   return mNoDecl ();
}
```

3.4. Makefile

The Makefile controls the invocation of the tools for compiler generation, takes care about the compilation and linking of the compiler parts, and finally runs the executable compiler using a sample input. The structure of the Makefile follows the recommendation given in the appropriate document [Groe] of the Cocktail Toolbox for Compiler Construction.

```
LIB     = $(HOME)/lib
INCDIR  = $(LIB)/include
CFLAGS  = -I$(INCDIR)
CC      = cc -g

SOURCES = Scanner.h Scanner.c Parser.h Parser.c Tree.h Tree.c Eval.h Eval.c l.c
OBJS    = Scanner.o Parser.o Tree.o Eval.o l.o

try:    l
        l < in 2> o
        diff out o

l:      $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) $(LIB)/libreuse.a -o l

Scanner.rpp Parser.lrk: l.prs
        lpp -cxzj l.prs;

Scanner.h Scanner.c:    l.scn Scanner.rpp
        rpp < l.scn | rex -cd;

Parser.h Parser.c:      Parser.lrk
        lark -cdi Parser.lrk;

Tree.h Tree.c:  l.ag
        cg -cdimwDIL5 l.ag;

Eval.h Eval.c:  l.ag
        cg -cDIL5 l.ag;

Parser.o:       Parser.h Scanner.h Tree.h
Eval.o:         Eval.h Tree.h DefTab.c
```

```
Tree.o:          Tree.h
l.o:             Parser.h Tree.h Eval.h

lint:   $(SOURCES)
        lint $(CFLAGS) -u $(SOURCES) | fgrep -v -i Warning

clean:
        rm -f Scanner.[hc] Parser.[hc] Tree.[hc] Eval.[hc]
        rm -f Parser.lrk Scanner.rpp core* *.o *.dbg l o yy*.h

.c.o:
        $(CC) $(CFLAGS) -c $*.c;
```

4. Declarations

The first part of this work will discuss problems in connection with declarations. The goal is to cover most of the typical tasks in name analysis or declaration analysis. All examples in this chapter involve a definition table module.

4.1. Declaration of Objects

Description

The first language under consideration deals with declarations and nothing else. A program consists of a list of abstract declarations, only. An abstract declaration introduces just the identifier of an object. This problem is simple and not very sensible. Its purpose is to serve as introductory example, to familiarize with the used method, and to illustrate the abstractions behind attribute diagrams. Those abstraction have been described in a previous section.

Conditions

- Identifiers may not be declared multiply

Remarks

The threaded attribute *Objects* used for the definition table may seem as overkill compared to a simple synthesized attribute. There are two reasons in favor of a threaded attribute. First, in general there might be several subtrees that contribute objects to a scope (see next problem). Second, the attribution influences the quality of the semantic error messages. The presented solution reports multiply declared identifiers starting at the second appearance. Using a synthesized attribute, all multiply declared identifiers but the last one would be reported as errors.

There are no explicit computations for the attribute *ObjectsOut* at the rules *Decl* and *NoDecl*. Copy rules are generated automatically by the tool *ag* in this case:

```
Decl   = { ObjectsOut := Next:ObjectsOut; }
NoDecl = { ObjectsOut := ObjectsIn;       }
```

The features SELF and DEP which are used in the computation of the attribute *Next:ObjectsIn* may need some explanation. The value SELF refers to the current tree node or in other words it is a pointer to the current node. The computation needed for this attribute would be just:

```
Next:ObjectsIn := SELF;
```

However, this way the dependency on the attribute *ObjectsIn* is lost and therefore a dynamic attribute evaluator for well-defined attribute grammars would just compute the last *ObjectsIn* attribute in a list and fail to compute the other instances. This problem can be fixed by the introduction of a virtual dependency on the attribute *ObjectsIn*. This is achieved with the pseudo function

DEP which returns its first argument. However, all arguments are incorporated into the dependency analysis and they are evaluated before DEP returns its result.

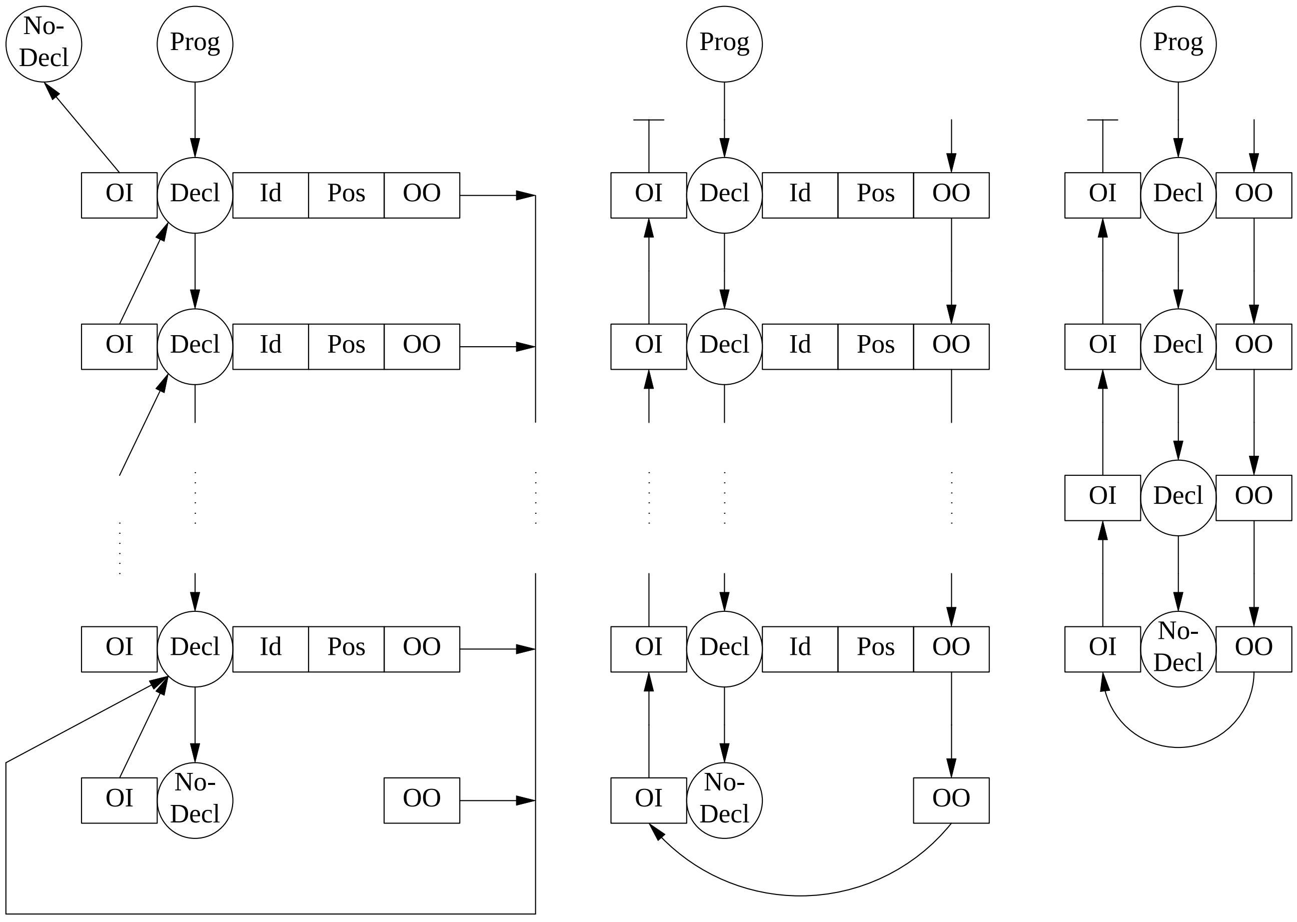
Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = DCL Ident Next: Decls .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Decls:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                               } .
Decl    = { Tree := mDecl (Ident:Ident, Ident:Position, Next:Tree);           } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      RefDecl   = Decl .
   > .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { Decls:ObjectsIn     := mNoDecl ();                                } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);            } .

END DefTab
```

Attribute Diagram



Input

```
DCL x DCL y DCL x DCL z
```

Messages

```
   1, 17: Error       identifier multiply declared: x
```

4.1.1. Sublists

Description

This language is an abstraction where the declarations for one scope are distributed over several subtrees in the abstract syntax tree. The example considers lists of lists of declarations. The threaded attribute *Objects* is definitively necessary in this case.

Conditions

- Identifiers may not be declared multiply

Concrete Syntax

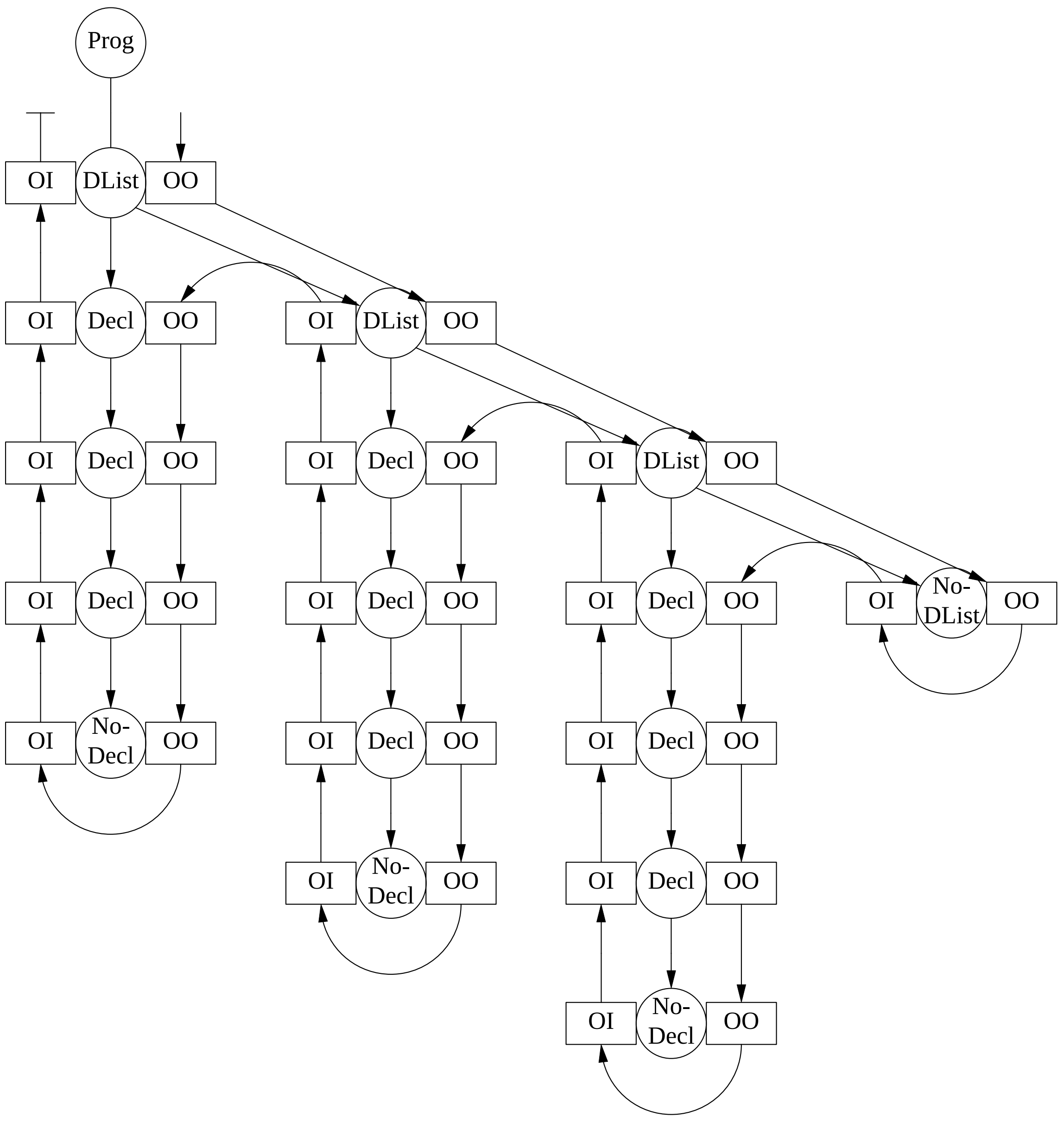
```
PROPERTY INPUT RULE

Prog             = DLists .
DLists           = <
   NoDList       = .
   DList         = LIST Decls Next: DLists .
```

```
> .
Decls           = <
   NoDecl       = .
   Decl         = DCL Ident Next: Decls .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE DLists Decls = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (DLists:Tree); };                             } .
NoDList = { Tree := mNoDList ();                                                } .
DList   = { Tree := mDList (Decls:Tree, Next:Tree);                             } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Decl    = { Tree := mDecl (Ident:Ident, Ident:Position, Next:Tree);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = DLists .
DLists          = <
   NoDList      = .
   DList        = Decls Next: DLists .
> .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      RefDecl   = Decl .
   > .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

DLists Decls    = [Objects: tTree THREAD] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { DLists:ObjectsIn    := mNoDecl ();                                  } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .

END DefTab
```

Attribute Diagram



Input

```
LIST DCL x
LIST DCL y DCL x DCL z
```

Messages

```
   2, 16: Error       identifier multiply declared: x
```

4.2. Use of Objects

Description

This language deals with abstract uses of objects.

Conditions

- Identifiers may not be declared multiply
- Every used identifier must be declared

Remarks

The list of declarations in a program defines a set of objects. At the rule *Prog* this set is denoted by the attribute *Decls:ObjectsOut*. It is the first argument of the call to the function *mEnv* which generates a data structure for a scope. In this case the second argument *NoTree* refers to a surrounding scope that is empty. The scope information is passed to all elements in the statement sequence using a pointer valued attribute called *Env*. The computations for this attribute are copy rules that are generated automatically.

Concrete Syntax

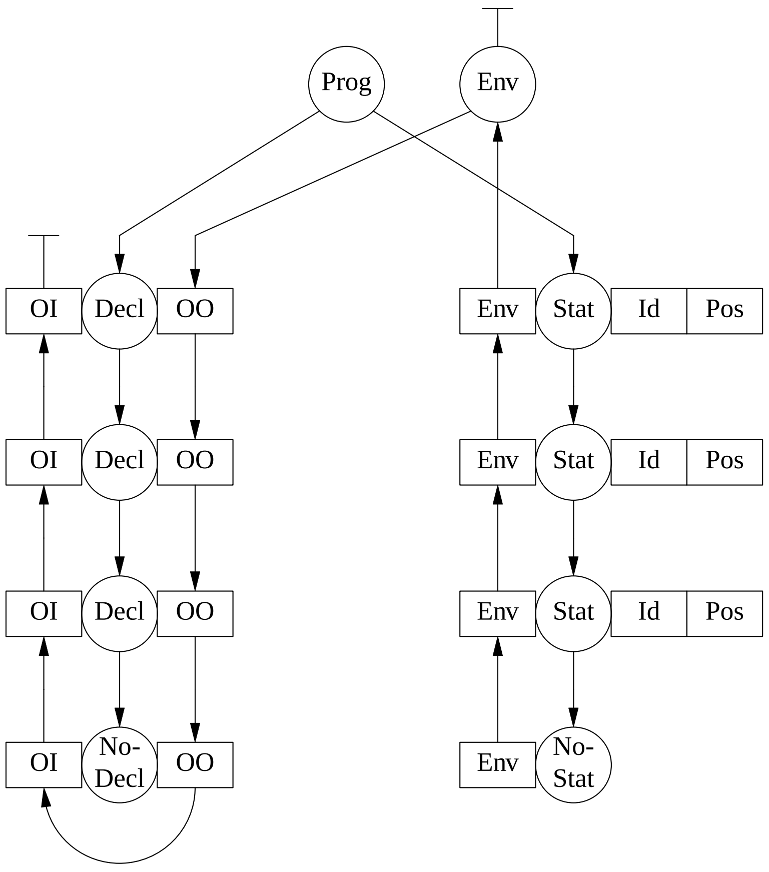
```
PROPERTY INPUT RULE

Prog            = Decls Stats .
Decls           = <
   NoDecl       = .
   Decl         = DCL Ident Next: Decls .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Decls:Tree, Stats:Tree); };                 } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Decl    = { Tree := mDecl (Ident:Ident, Ident:Position, Next:Tree);            } .
NoStat  = { Tree := mNoStat ();                                                } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Decls Stats .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      RefDecl   = Decl .
   > .
> .
```

```
Stats            = <
   NoStat        = .
   Stat          = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls            = [Objects: tTree THREAD] .
Stats            = [Env: tTree] .
Env              = [Objects: tTree IN] Env IN .

RULE

Prog    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, NoTree);             } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                   } .

END DefTab
```

Attribute Diagram



Input

```
DCL x DCL y DCL x DCL z
USE x USE u USE y USE y
```

Messages

```
   1, 17: Error       identifier multiply declared: x
   2, 11: Error       identifier not declared: u
```

4.3. Nested Blocks

This group of languages considers nested blocks. Procedures without parameters are the language constructs serving as blocks.

4.3.1. Visibility from Block Begin to Block End

Description

An identifier may be used in the whole block where it is declared or in enclosed blocks unless there is a declaration with the same identifier. It may be used before or after its declaration. In other words: The *declare before use* principle is not enforced.

Conditions

- Identifiers may not be declared multiply
- Every used identifier must be declared

Remarks

This scope rule is comfortable because a programmer does not have to worry about the order of the declarations. This problem can be delegated to a compiler.

The realization of this scope rule is relatively simple because the definition table supports this kind of comfort. Scope rules such as *declare before use* can be realized as well (see later section) but require an increased effort.

The node type *Decls* has an attribute named *Env* because the declaration of a procedure (node type *Proc*) may contain uses of objects.

Concrete Syntax

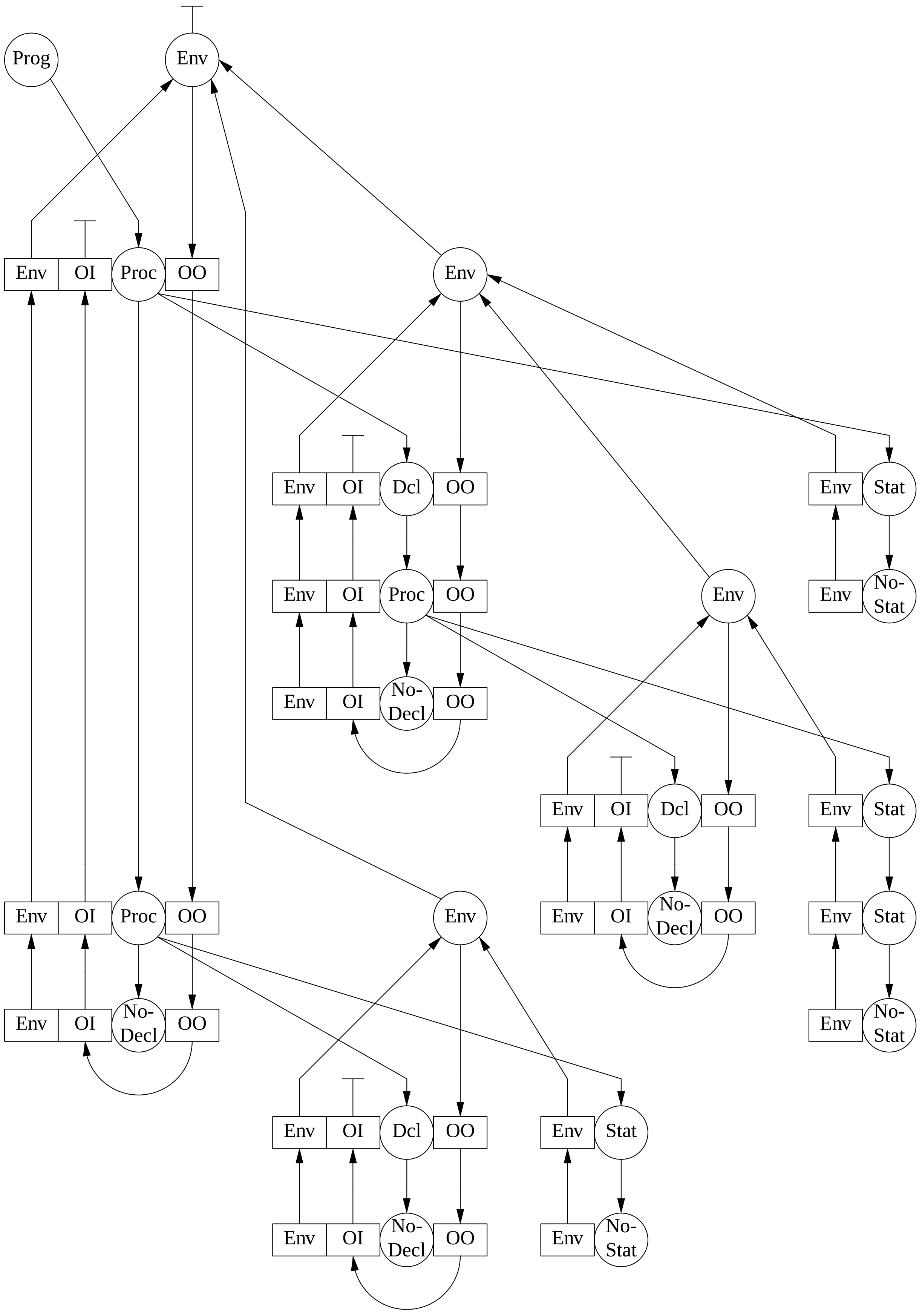
```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .
```

```
RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                                 } .
NoDecl  = { Tree := mNoDecl ();                                                   } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);                } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                           } .
NoStat  = { Tree := mNoStat ();                                                   } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);               } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);                } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                     } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);                } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                     } .

END DefTab
```

Attribute Diagram



Input

```
PROCEDURE p
   DCL x
   PROCEDURE q USE x USE y END
   PROCEDURE r
      PROCEDURE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y
END
```

Messages

```
   3, 26: Error       identifier not declared: y
   9, 14: Error       identifier not declared: y
```

4.3.2. Visibility from Declaration to Block End

Description

An identifier may be used in the block where it is declared in a range starting from its declaration and ending at the end of the block. It may be used in enclosed blocks residing in the mentioned range unless there is a declaration with the same identifier. If an identifier is used in a block before a declaration of the same identifier contained in this block it refers to an object in a surrounding block.

Conditions

- Identifiers may not be declared multiply
- Every used identifier must be declared

Remarks

The languages C and Ada use this scope rule. This strategy seems to be influenced from an implementation that does semantic analysis in one pass and that uses a stack of scopes.

The concrete syntax, the abstract syntax, the attribute diagram, and the test input for this example are the same as in section 4.3.1. In the attribute grammar only the checks at the rule *Stat* are different.

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree
```

```
PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                                 } .
NoDecl  = { Tree := mNoDecl ();                                                   } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);                } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                           } .
NoStat  = { Tree := mNoStat ();                                                   } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);               } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);                } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                     } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);                } .
Stat    = { CHECK IdentifyTail (Ident, Env, Pos)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                     } .

END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
PROCEDURE p
   DCL x
   PROCEDURE q USE x USE y END
   PROCEDURE r
      PROCEDURE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y
END
```

Messages

```
   3, 26: Error       identifier not declared: y
   5, 29: Error       identifier not declared: y
   9, 14: Error       identifier not declared: y
```

4.3.3. Declare Before Use

Description

An identifier must be declared before its use. The description is similar to the previous one with the following exception: It is an error to use an identifier before a declaration of the same identifier in the same block.

Conditions

- Identifiers may not be declared multiply
- Every used identifier must be declared
- The declaration of an object must lie before any use

Remarks

The language Modula-2 uses this scope rule with respect to declarations.

The concrete syntax, the abstract syntax, the attribute diagram, and the test input for this example are the same as in section 4.3.1. In the attribute grammar only the checks at the rule *Stat* are different.

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .
```

```
MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                                      } .
NoDecl  = { Tree := mNoDecl ();                                                       } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);                    } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                               } .
NoStat  = { Tree := mNoStat ();                                                       } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);                   } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Stat            = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);                    } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                         } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);                    } .
Stat    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK IsBefore (Object->Decl.\Pos, Pos)
            => Error (Pos, "identifier used before declaration", Ident);        } .
END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
PROCEDURE p
   DCL x
   PROCEDURE q USE x USE y END
   PROCEDURE r
      PROCEDURE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y
END
```

Messages

```
   3, 26: Error       identifier not declared: y
   5, 23: Error       identifier used before declaration: x
   5, 29: Error       identifier used before declaration: y
   9, 14: Error       identifier not declared: y
```

4.3.4. Forward/Incomplete Declarations

Description

The regulations for *declare before use* are extended by forward or incomplete declarations. Again an object has to be declared before any use. It is sufficient if a forward declaration precedes the using location as long as a complete declaration follows either before or after the using location.

Conditions

- Identifiers may not be declared multiply
- Every used identifier must be declared
- The declaration of an object must lie before any use
- A complete declaration must exist for every forward declaration
- The complete declaration must lie behind the forward declaration

Remarks

The language Pascal uses this scope rule with one exception for the declaration of pointer types.

The implementation allows multiply declarations to a certain degree. A forward declaration and the corresponding complete declaration are treated internally as two separate objects having the same identifier.

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
```

```
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
      Forward   = PROCEDURE Ident FORWARD Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                  } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                              } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);           } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                      } .
Forward = { Tree := mForward (Ident:Ident, Ident:Position, Next:Tree);       } .
NoStat  = { Tree := mNoStat ();                                              } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);          } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      Forward   = .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {
# include "DefTab.c"
static tTree object;
}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Forward Stat    = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);              } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                   } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;                             } .
Dcl     = { CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Proc    = { CHECK object = IdentifyObjects (Ident, ObjectsIn),
                  object->Kind == kNoDecl || object->Kind == kForward
            => Error (Pos, "identifier multiply declared", Ident);              } .
Forward = { Object              := IdentifyLocalKind (Ident, Env, kProc);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "complete declaration missing", Ident) AND_THEN
            CHECK IsBefore (Pos, Object->Proc.\Pos)
            => Error (Pos, "forward follows complete declaration", Ident);      } .
Stat    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK IsBefore (Object->Decl.\Pos, Pos) ||
                  (object = IdentifyWholeKind (Ident, Env, kForward),
                  object->Kind == kForward) && IsBefore (object->Forward.\Pos, Pos)
            => Error (Pos, "identifier used before declaration", Ident);        } .

END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
PROCEDURE p
   PROCEDURE a USE b USE c USE f END
   PROCEDURE b FORWARD
   PROCEDURE c USE b USE c USE f END
   PROCEDURE b END
   PROCEDURE c USE b USE c USE f END
   PROCEDURE b END
   PROCEDURE a FORWARD
   PROCEDURE d FORWARD
   PROCEDURE e FORWARD
   PROCEDURE e FORWARD
END
```

Messages

```
   2, 20: Error       identifier used before declaration: b
   2, 26: Error       identifier used before declaration: c
   2, 32: Error       identifier not declared: f
   4, 26: Error       identifier used before declaration: c
   4, 32: Error       identifier not declared: f
   6, 14: Error       identifier multiply declared: c
   6, 32: Error       identifier not declared: f
   7, 14: Error       identifier multiply declared: b
   8, 14: Error       forward follows complete declaration: a
   9, 14: Error       complete declaration missing: d
  10, 14: Error       complete declaration missing: e
  11, 14: Error       complete declaration missing: e
```

4.4. Implicit Declarations

4.4.1. Local

Description

This language does not require that used identifiers are declared. For undeclared identifiers objects are declared implicitly. These are inserted in the local environment or the actual block, respectively.

Conditions

- Identifiers may not be declared multiply

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                               } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);              } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                         } .
NoStat  = { Tree := mNoStat ();                                                 } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
```

```
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);            } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                 } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);            } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl => {
               tTree t = Env->\Env.Objects;
               Env->\Env.Objects = mDcl (Ident, Pos, NoTree);
               Env->\Env.Objects->Dcl.ObjectsIn = t;
               Error (Pos, "implicit declaration", Ident);
            };                                                                } .

END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
PROCEDURE p
   DCL x
   PROCEDURE q USE x USE y END
   PROCEDURE r
      PROCEDURE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y USE x
END
```

Messages

```
   3, 26: Error       implicit declaration: y
   9, 14: Error       implicit declaration: y
```

4.4.2. Global

Description

This language does not require that used identifiers are declared. For undeclared identifiers objects are declared implicitly. These are inserted in the global environment or the outermost block, respectively.

Conditions

- Identifiers may not be declared multiply

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                              } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);             } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
NoStat  = { Tree := mNoStat ();                                                } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl          = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl        = .
      Proc       = Decls Stats .
      RefDecl    = Decl .
   > .
> .
Stats            = <
   NoStat        = .
   Stat          = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .
```

```
MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls            = [Objects: tTree THREAD] .
Decls Stats      = [Env: tTree] .
Env              = [Objects: tTree IN] Env IN .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);              } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                   } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl => {
               tTree t;
               tTree env = Env;
               while (env->\Env.\Env != NoTree) env = env->\Env.\Env;
               t = env->\Env.Objects;
               env->\Env.Objects = mDcl (Ident, Pos, NoTree);
               env->\Env.Objects->Dcl.ObjectsIn = t;
               Error (Pos, "implicit declaration", Ident);
            };                                                                  } .

END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
PROCEDURE p
   DCL x
   PROCEDURE q USE x USE y END
   PROCEDURE r
      PROCEDURE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y USE x
END
```

Messages

```
   3, 26: Error       implicit declaration: y
```

4.5. Named Blocks

There are programming languages where blocks may be named. Procedures can also be regarded as named blocks.

4.5.1. Qualified Use

Description

Usually objects declared in a surrounding block can not be accessed in an enclosed block if there is a redeclaration with the same identifier. A precondition to overcome this restriction is the existence of named blocks. Then those objects from surrounding blocks can be accessed by qualification.

Conditions

- Identifiers may not be declared multiply
- Every used object must be declared
- The qualifying object must be declared
- The qualifying object must be a procedure
- The object accessed by qualification must exist at the time of access

Remarks

The solution of this problem uses the access of a non-local attribute in the rule *Qualify*. It is necessary to look up an object in the environment of the qualifying object. This is achieved using the following notation:

```
   REMOTE Designator:Object => Proc:NewEnv
```

The attribute *Designator:Object* refers to the tree node describing the qualifying object. This node is of type *Proc*. The attribute *NewEnv* of this node is needed in a computation and the attribute evaluator takes care that this attribute is computed before its use.

Concrete Syntax

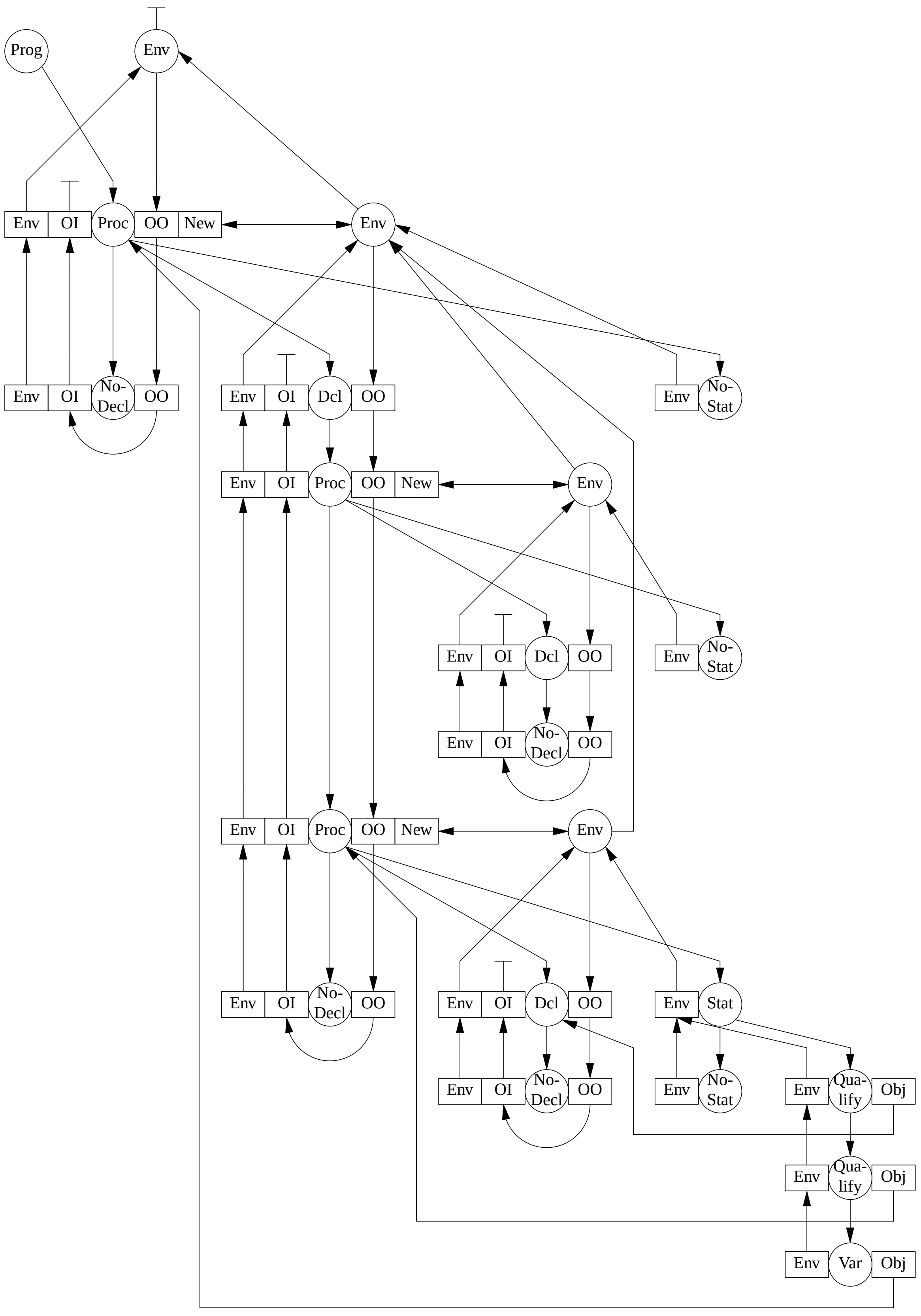
```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Designator Next:Stats .
> .
Designator      = <
   Var          = Ident .
   Qualify      = Designator '.' Ident .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .
```

```
MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats Designator = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                                  } .
NoDecl  = { Tree := mNoDecl ();                                                    } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);                 } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                        Stats:Tree);                                               } .
NoStat  = { Tree := mNoStat ();                                                    } .
Stat    = { Tree := mStat (Designator:Tree, Next:Tree);                            } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position);                            } .
Qualify = { Tree := mQualify (Ident:Ident, Ident:Position, Designator:Tree);       } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}
EXPORT { typedef tTree ttree; }

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Designator Next: Stats .
> .
Designator      = [Ident: tIdent] [Pos: tPosition] <
   Var          = .
   Qualify      = Designator .
> .

MODULE DefTab

EVAL GLOBAL {
# include "DefTab.c"

static rbool IsAccessible (Ident, Env, Object)
   register tIdent      Ident;
   register tTree       Env;
   register tTree       Object;
{
   while (Env != NoTree)
      if (IdentifyLocal (Ident, Env) == Object) return rtrue;
      else Env = Env->Env.Env;
   return rfalse;
}
}

DECLARE

Decls           = [Objects: ttree THREAD] .
Decls Stats Designator  = [Env: ttree INH] .
Proc            = [NewEnv: ttree] .
Env             = [Objects: ttree IN] Env IN [Object: ttree IN] .
Designator      = [Object: ttree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree, SELF);        } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            NewEnv              := mEnv (Decls:ObjectsOut, Env, SELF);
            Stats:Env           := NewEnv;
            Decls:Env           := NewEnv;                                      } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Stat    = { CHECK Designator:Object->Kind == kNoDecl ||
                  IsAccessible (Designator:Ident, Env, Designator:Object)
            => Error (Designator:Pos, "illegal usage", Designator:Ident);       } .
Designator = { Object           := mNoDecl ();                                  } .
Var     = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                   } .
Qualify = { CHECK Designator:Object->Kind == kProc ||
                  Designator:Object->Kind == kNoDecl
            => Error (Pos, "procedure required", Ident);
            Object              := Designator:Object->Kind != kProc ? mNoDecl () :
               IdentifyWhole (Ident, REMOTE Designator:Object => Proc:NewEnv);
            CHECK Designator:Object->Kind != kProc || Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                   } .

END DefTab
```

Attribute Diagram



Input

```
PROCEDURE p
   DCL x
   PROCEDURE q
      DCL x
      USE x USE y
   END
   PROCEDURE r
      PROCEDURE s
         DCL x
         USE x USE y
      END
      DCL x DCL y
      USE x USE y
      USE p.x USE q.x USE r.x USE s.x
      USE p.y USE q.y USE r.y USE s.y USE x.x
      USE q.p.x USE r.q.p.x USE q.p.y USE p.q USE p.q.x
   END
   USE x USE y
END
```

Messages

```
   5, 17: Error       identifier not declared: y
  14, 21: Error       illegal usage: x
  14, 37: Error       illegal usage: x
  15, 13: Error       identifier not declared: y
  15, 21: Error       identifier not declared: y
  15, 45: Error       procedure required: x
  16, 37: Error       identifier not declared: y
  16, 55: Error       illegal usage: x
  18, 14: Error       identifier not declared: y
```

4.6. Kind Checking

The various identifiers in a program can lie in one or in several name spaces.

4.6.1. One Name Space

Description

This language regards all identifiers to lie in one name space. It distinguishes different kinds of objects: variables, named constants, and procedures. There are two different locations for the use of objects. A *use* statement requires a variable or a constant identifier and a *call* statement requires a procedure identifier.

Conditions

- Identifiers may not be declared multiply
- Every used object must be declared
- A use statement requires a variable or a constant identifier
- A call statement requires a procedure identifier

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
```

```
   NoDecl       = .
   Decl         = <
      Var       = VAR Ident Next: Decls .
      Const     = CONST Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = <
      Use       = USE Ident Next: Stats .
      Call      = CALL Ident Next: Stats .
   > .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                              } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position, Next:Tree);             } .
Const   = { Tree := mConst (Ident:Ident, Ident:Position, Next:Tree);           } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
NoStat  = { Tree := mNoStat ();                                                } .
Use     = { Tree := mUse (Next:Tree, Ident:Ident, Ident:Position);             } .
Call    = { Tree := mCall (Next:Tree, Ident:Ident, Ident:Position);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var       = .
      Const     = .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
   > .
> .
```

```
MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Use Call        = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);              } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                   } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Use     = { Object := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kVar || Object->Kind == kConst
            => Error (Pos, "variable or constant required", Ident);             } .
Call    = { Object := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kProc
            => Error (Pos, "procedure required", Ident);                        } .

END DefTab
```

Attribute Diagram

see attribute diagram in the section on "Nested Blocks"

Input

```
PROCEDURE p
   VAR x
   CONST x
   CONST y
   VAR q
   PROCEDURE q
      USE x
      USE y
      USE q
      CALL q
   END
   USE r
   USE p
   CALL z
   CALL x
END
```

Messages

```
   3, 10: Error       identifier multiply declared: x
   6, 14: Error       identifier multiply declared: q
   9, 11: Error       variable or constant required: q
  12,  8: Error       identifier not declared: r
  13,  8: Error       variable or constant required: p
  14,  9: Error       identifier not declared: z
  15,  9: Error       procedure required: x
```

4.6.2. Several Name Spaces

Description

This language regards the identifiers to lie in two name spaces. Variables and named constants constitute one name space and procedures constitute an other one. Again, there are two different locations for the use of objects. A *use* statement requires a variable or a constant identifier and a *call* statement requires a procedure identifier.

Conditions

- Identifiers in one name space may not be declared multiply
- Every used object must be declared
- A use statement requires a variable or a constant identifier
- A call statement requires a procedure identifier

Remarks

The functions *IdentifyWholeKind* and *Tree_IsType* have an argument describing the kind of an object. This argument describes either a single kind of objects (e. g. *kProc*) or a class of object kinds (e. g. *kValue*) which comprises the object kinds having a value (*kVar* and *kConst*).

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Var       = VAR Ident Next: Decls .
      Const     = CONST Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = <
      Use       = USE Ident Next: Stats .
      Call      = CALL Ident Next: Stats .
   > .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .
```

```
RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                              } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position, Next:Tree);             } .
Const   = { Tree := mConst (Ident:Ident, Ident:Position, Next:Tree);           } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
NoStat  = { Tree := mNoStat ();                                                } .
Use     = { Tree := mUse (Next:Tree, Ident:Ident, Ident:Position);             } .
Call    = { Tree := mCall (Next:Tree, Ident:Ident, Ident:Position);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Value     = <
         Var    = .
         Const  = .
      > .
      Proc      = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
   > .
> .

MODULE DefTab

EVAL GLOBAL {
# include "DefTab.c"
static int kind;
}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Use Call        = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);             } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env          := Stats:Env;                                  } .
Decl    = { Next:ObjectsIn     := DEP (SELF, ObjectsIn);
            ObjectsOut         := Next:ObjectsOut;                            } .
Value   = { CHECK ! Tree_IsType (IdentifyObjects (Ident, ObjectsIn), kValue)
            => Error (Pos, "identifier multiply declared", Ident);            } .
Proc    = { CHECK IdentifyObjects (Ident, ObjectsIn)->Kind != kProc
            => Error (Pos, "identifier multiply declared", Ident);            } .
Use     = { Object := IdentifyWholeKind (Ident, Env, kValue);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                 } .
Call    = { Object := IdentifyWholeKind (Ident, Env, kProc);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                 } .

END DefTab
```

Attribute Diagram

see attribute diagram in the section on "Nested Blocks"

Input

```
PROCEDURE p
   VAR x
   CONST x
   CONST y
   VAR q
   PROCEDURE q
      USE x
      USE y
      USE q
      CALL q
   END
   USE r
   USE p
   CALL z
   CALL x
END
```

Messages

```
   3, 10: Error       identifier multiply declared: x
  12,  8: Error       identifier not declared: r
  13,  8: Error       identifier not declared: p
  14,  9: Error       identifier not declared: z
  15,  9: Error       identifier not declared: x
```

4.7. Labels

There are various ways to treat labels serving as targets for goto statements.

4.7.1. Implicit Declaration

Description

In the simplest case the label marking a statement introduces an object into the definition table. It can be regarded as some kind of implicit declaration.

Conditions

- Identifiers for all kinds of objects including labels may not be declared multiply

- Every used object must be declared
- A use statement requires a variable or a constant identifier
- A call statement requires a procedure identifier
- A goto statement requires a label identifier

Remarks

Labels are treated as implicit declarations and therefore corresponding object descriptors are inserted into the current set of objects.

Besides declarations statements may contribute objects to the definition table as well. Therefore both, the node types *Decls* and *Stats* have the threaded attribute *Objects*.

Concrete Syntax

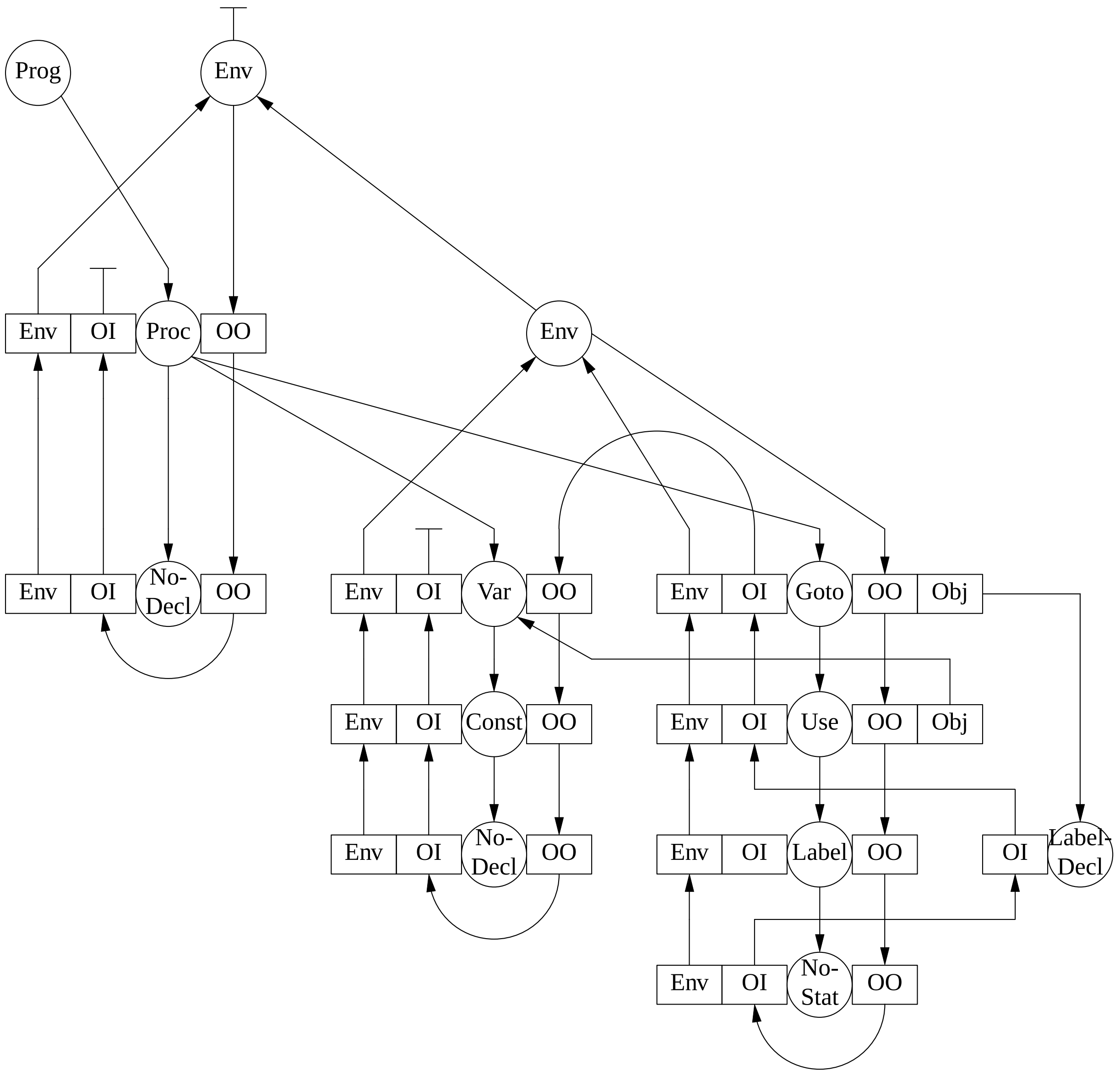
```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Var       = VAR Ident Next: Decls .
      Const     = CONST Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = <
      Use       = USE Ident Next: Stats .
      Call      = CALL Ident Next: Stats .
      Label     = Ident ’:’ Next: Stats .
      Goto      = GOTO Ident Next: Stats .
   > .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                               } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position, Next:Tree);              } .
Const   = { Tree := mConst (Ident:Ident, Ident:Position, Next:Tree);            } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                         } .
NoStat  = { Tree := mNoStat ();                                                 } .
Use     = { Tree := mUse (Next:Tree, Ident:Ident, Ident:Position);              } .
Call    = { Tree := mCall (Next:Tree, Ident:Ident, Ident:Position);             } .
Label   = { Tree := mLabel (Next:Tree, Ident:Ident, Ident:Position);            } .
Goto    = { Tree := mGoto (Next:Tree, Ident:Ident, Ident:Position);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var       = .
      Const     = .
      Proc      = Decls Stats .
      LabelDecl = .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
      Label     = [Ident: tIdent] [Pos: tPosition] .
      Goto      = [Ident: tIdent] [Pos: tPosition] .
   > .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls Stats     = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Use Call Goto   = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);            } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Stats:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                 } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);            } .
Use     = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kVar || Object->Kind == kConst
            => Error (Pos, "variable or constant required", Ident);           } .
Call    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kProc
            => Error (Pos, "procedure required", Ident);                      } .
```

```
Label   = { Next:ObjectsIn      := {
               Next:ObjectsIn   = mLabelDecl (Ident, Pos, NoTree);
               Next:ObjectsIn->LabelDecl.\ObjectsIn = ObjectsIn;
            };
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);           } .
Goto    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kLabelDecl
            => Error (Pos, "label required", Ident);                         } .

END DefTab
```

Attribute Diagram



Input

```
PROCEDURE p
   VAR x
   PROCEDURE q
      VAR y
      x: x: y:
   END
   y:
   USE x USE y USE q USE z
   CALL x CALL y CALL q CALL z
   GOTO x GOTO y GOTO q GOTO z
END
```

Messages

```
   5, 10: Error       identifier multiply declared: x
   5, 13: Error       identifier multiply declared: y
   8, 14: Error       variable or constant required: y
   8, 20: Error       variable or constant required: q
   8, 26: Error       identifier not declared: z
   9,  9: Error       procedure required: x
   9, 16: Error       procedure required: y
   9, 30: Error       identifier not declared: z
  10,  9: Error       label required: x
  10, 23: Error       label required: q
  10, 30: Error       identifier not declared: z
```

4.7.2. Explicit Declaration

Description

In addition to the location where a label marks a statement an explicit declaration for that label may be required (for example in Pascal).

Conditions

- Identifiers for all kinds of objects including labels may not be declared multiply
- Every used object must be declared
- A use statement requires a variable or a constant identifier
- A call statement requires a procedure identifier
- A goto statement requires a label identifier
- A label definition requires a label identifier
- At most one label definition may exist for one declared label

Remarks

The last condition is implemented using a boolean attribute called *Mark* for the node type *LabelDecl*.

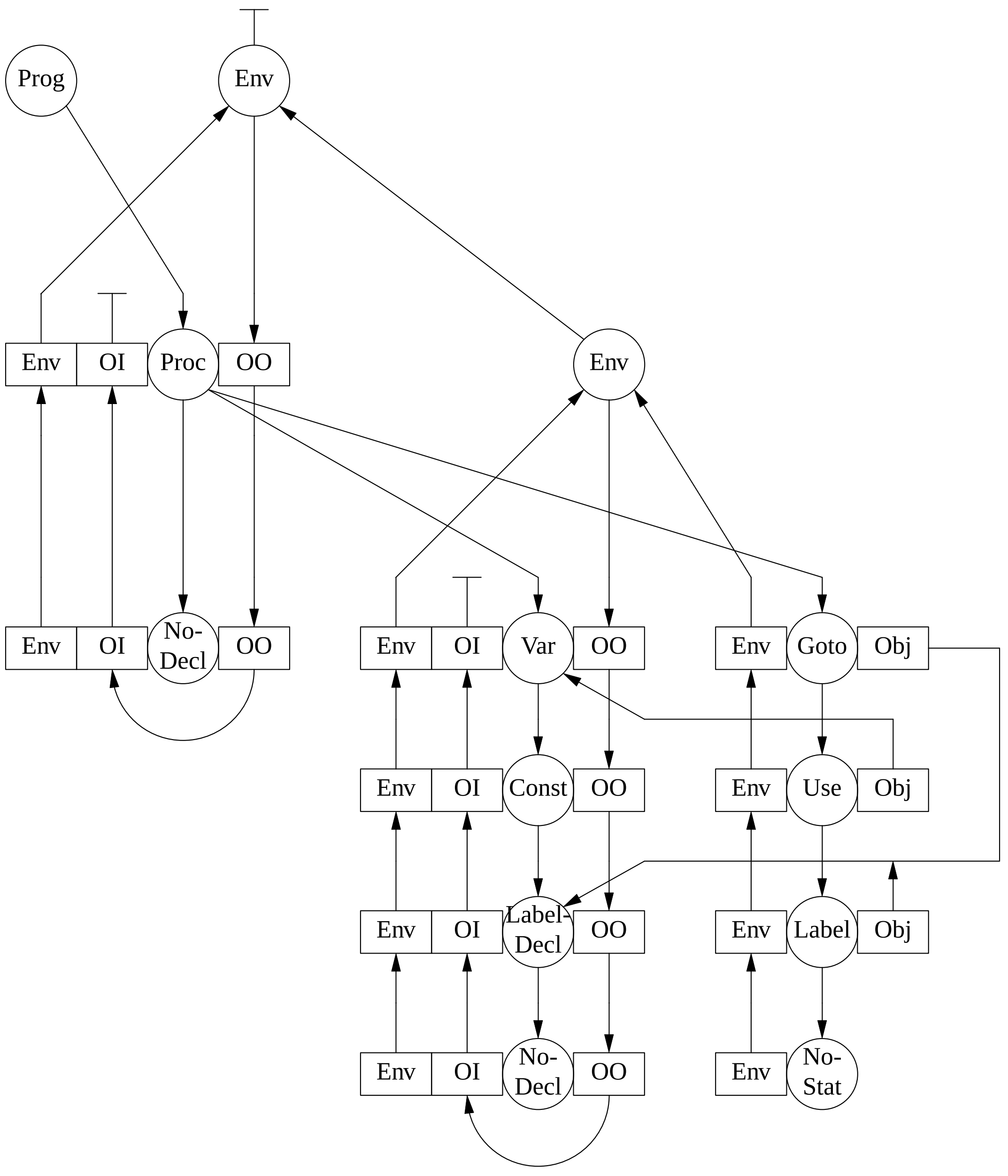
Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Var       = VAR Ident Next: Decls .
      Const     = CONST Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
      LabelDecl = LABEL Ident Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = <
      Use       = USE Ident Next: Stats .
      Call      = CALL Ident Next: Stats .
      Label     = Ident ':' Next: Stats .
      Goto      = GOTO Ident Next: Stats .
   > .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                              } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position, Next:Tree);             } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
LabelDecl={ Tree := mLabelDecl (Ident:Ident, Ident:Position, Next:Tree, rfalse);} .
NoStat  = { Tree := mNoStat ();                                                } .
Use     = { Tree := mUse (Next:Tree, Ident:Ident, Ident:Position);             } .
Call    = { Tree := mCall (Next:Tree, Ident:Ident, Ident:Position);            } .
Label   = { Tree := mLabel (Next:Tree, Ident:Ident, Ident:Position);           } .
Goto    = { Tree := mGoto (Next:Tree, Ident:Ident, Ident:Position);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var       = .
      Const     = .
      Proc      = Decls Stats .
      LabelDecl = [Mark: rbool] .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
      Label     = [Ident: tIdent] [Pos: tPosition] .
      Goto      = [Ident: tIdent] [Pos: tPosition] .
   > .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Use Call Goto
Label           = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);           } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);           } .
Use     = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kVar || Object->Kind == kConst
            => Error (Pos, "variable or constant required", Ident);          } .
Call    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kProc
            => Error (Pos, "procedure required", Ident);                     } .
Label   = { Object              := IdentifyLocal (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kLabelDecl
            => Error (Pos, "label required", Ident)
            => { if (Object->LabelDecl.Mark)
               Error (Pos, "label multiply declared", Ident);
            else
               Object->LabelDecl.Mark = rtrue;
            };                                                               } .
Goto    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kLabelDecl
            => Error (Pos, "label required", Ident);                         } .

END DefTab
```

Attribute Diagram



Input

```
PROCEDURE p
   VAR x
   PROCEDURE q
      VAR y LABEL w LABEL y LABEL z
      x: w: w: y:
   END LABEL y
   y:
   USE x USE y USE q USE z
   CALL x CALL y CALL q CALL z
   GOTO x GOTO y GOTO q GOTO z
END
```

Messages

```
   4, 27: Error       identifier multiply declared: y
   5,  7: Error       identifier not declared: x
   5, 13: Error       label multiply declared: w
   8, 14: Error       variable or constant required: y
   8, 20: Error       variable or constant required: q
   8, 26: Error       identifier not declared: z
   9,  9: Error       procedure required: x
   9, 16: Error       procedure required: y
   9, 30: Error       identifier not declared: z
  10,  9: Error       label required: x
  10, 23: Error       label required: q
  10, 30: Error       identifier not declared: z
```

4.7.3. Jump Checking

Description

Usually it is forbidden to jump from outside into compound statements such as loops or branch statements.

Conditions

- Identifiers for all kinds of objects including labels may not be declared multiply
- Every used object must be declared
- A use statement requires a variable or a constant identifier
- A call statement requires a procedure identifier
- A goto statement requires a label identifier

Remarks

The solution treats compound statements as nested blocks with own scopes. This way the labels of illegal gotos become invisible and are reported as undeclared identifiers. In order to keep the number of scopes small and to speed up the run time of the identification functions new scopes are introduced only if they are absolutely necessary. This is achieved by the function *mENV* from the definition table module.

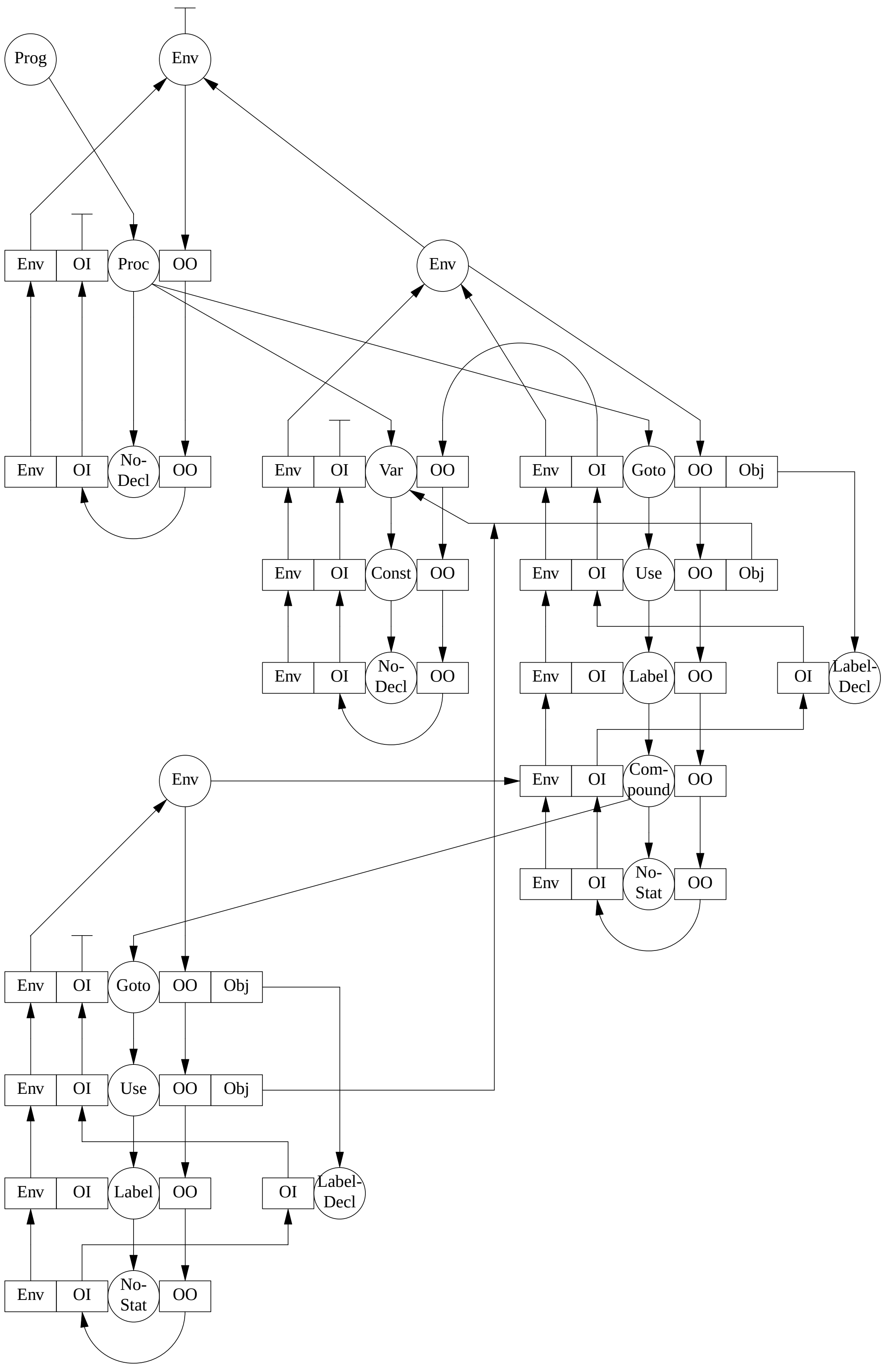
Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = <
      Var       = VAR Ident Next: Decls .
      Const     = CONST Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats 'END' Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = <
      Use       = USE Ident Next: Stats .
      Call      = CALL Ident Next: Stats .
      Label     = Ident ':' Next: Stats .
      Goto      = GOTO Ident Next: Stats .
      Compound  = ’BEGIN’ Stats ’END’ Next: Stats .
   > .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                         } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Proc:Tree); };                                   } .
NoDecl  = { Tree := mNoDecl ();                                                     } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position, Next:Tree);                  } .
Const   = { Tree := mConst (Ident:Ident, Ident:Position, Next:Tree);                } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                             } .
NoStat  = { Tree := mNoStat ();                                                     } .
Use     = { Tree := mUse (Next:Tree, Ident:Ident, Ident:Position);                  } .
Call    = { Tree := mCall (Next:Tree, Ident:Ident, Ident:Position);                 } .
Label   = { Tree := mLabel (Next:Tree, Ident:Ident, Ident:Position);                } .
Goto    = { Tree := mGoto (Next:Tree, Ident:Ident, Ident:Position);                 } .
Compound= { Tree := mCompound (Next:Tree, Stats:Tree);                              } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Proc .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Var       = .
      Const     = .
      Proc      = Decls Stats .
      LabelDecl = .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Next: Stats <
      Use       = [Ident: tIdent] [Pos: tPosition] .
      Call      = [Ident: tIdent] [Pos: tPosition] .
      Label     = [Ident: tIdent] [Pos: tPosition] .
      Goto      = [Ident: tIdent] [Pos: tPosition] .
      Compound  = Stats .
   > .
> .

MODULE DefTab

EVAL GLOBAL {
# include "DefTab.c"
static tTree mENV (Objects, Env)
   register tTree       Objects;
   register tTree       Env;
{
   return Objects->Kind == kNoDecl ? Env : mEnv (Objects, Env);
}
}

DECLARE

Decls Stats     = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Use Call Goto   = [Object: tTree] .

RULE

Prog    = { Proc:ObjectsIn      := mNoDecl ();
            Proc:Env            := mEnv (Proc:ObjectsOut, NoTree);             } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Stats:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                  } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);             } .
Use     = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kVar || Object->Kind == kConst
            => Error (Pos, "variable or constant required", Ident);            } .
Call    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kProc
            => Error (Pos, "procedure required", Ident);                       } .
Label   = { Next:ObjectsIn      := {
               Next:ObjectsIn   = mLabelDecl (Ident, Pos, NoTree);
               Next:ObjectsIn->LabelDecl.\ObjectsIn = ObjectsIn;
            };
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);             } .
Goto    = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK Object->Kind == kLabelDecl
            => Error (Pos, "label required", Ident);                           } .
Compound= { ObjectsOut          := Next:ObjectsOut;
            Stats:ObjectsIn     := mNoDecl ();
            Stats:Env           := mENV (Stats:ObjectsOut, Env);               } .

END DefTab
```

Attribute Diagram



Input

```
PROCEDURE p
   PROCEDURE q
      x: GOTO x GOTO y GOTO z
   END
   y: GOTO x GOTO y GOTO z
   BEGIN
     z: GOTO x GOTO y GOTO z
   END
   GOTO x GOTO y GOTO z
   BEGIN
     GOTO x GOTO y GOTO z
   END
END
```

Messages

```
   3, 29: Error       identifier not declared: z
   5, 12: Error       identifier not declared: x
   5, 26: Error       identifier not declared: z
   7, 14: Error       identifier not declared: x
   9,  9: Error       identifier not declared: x
   9, 23: Error       identifier not declared: z
  11, 11: Error       identifier not declared: x
  11, 25: Error       identifier not declared: z
```

4.8. Modules

4.8.1. Local Modules

Description

So-called local modules may be nested like blocks. However, objects from a surrounding module are not visible in enclosed modules. Such a language may not be sensible. It serves as introductory example for a series of languages dealing with modules.

Conditions

- Identifiers may not be declared multiply
- Every used object must be declared

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Module .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
      Module    = ’MODULE’ Ident Decls Stats ’END’ Next: Decls .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .
```

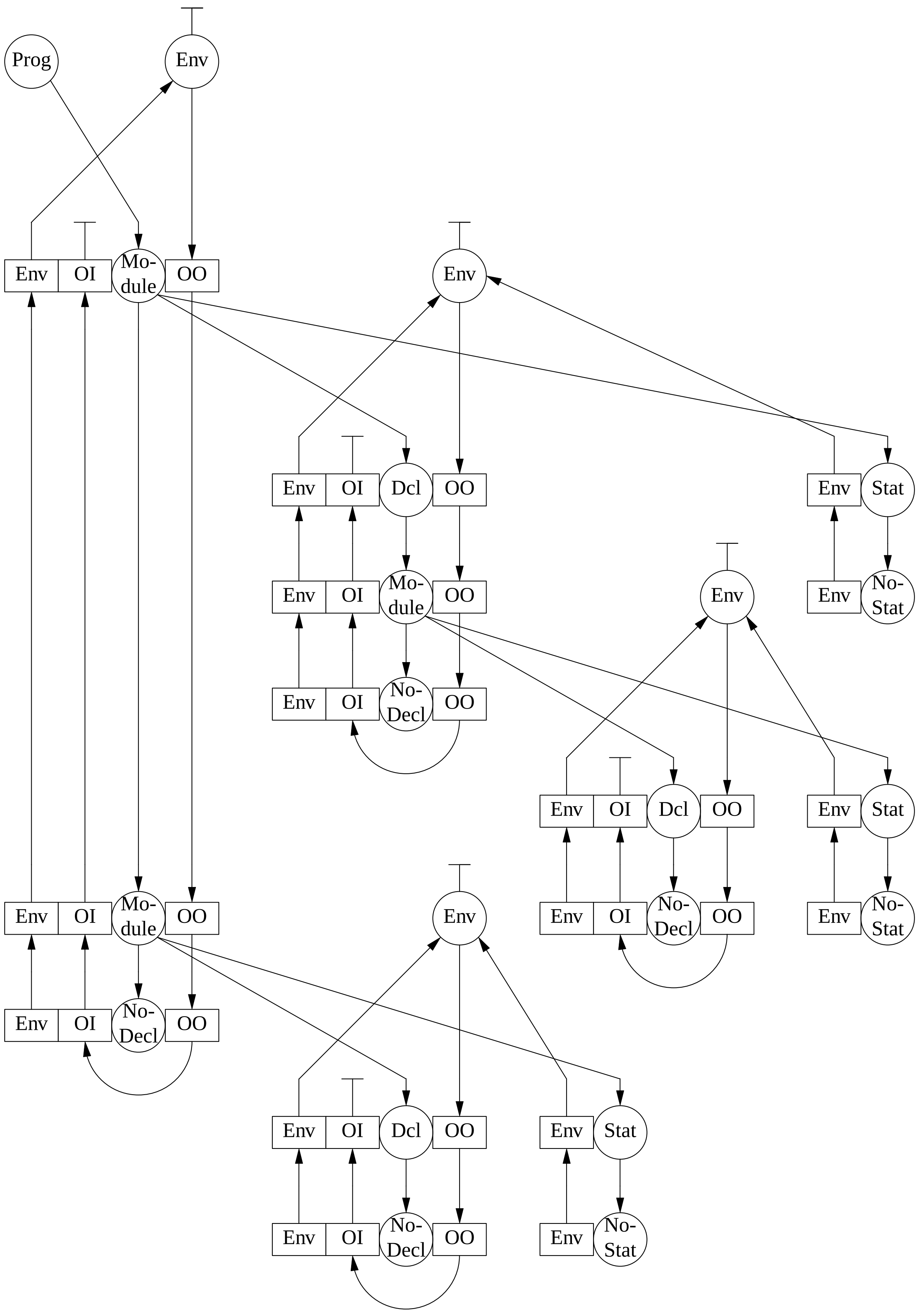
```
MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Module:Tree); };                             } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);              } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                         } .
Module  = { Tree := mModule (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                             Stats:Tree);                                       } .
NoStat  = { Tree := mNoStat ();                                                 } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Module .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      Module    = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .

RULE

Prog    = { Module:ObjectsIn    := mNoDecl ();
            Module:Env          := mEnv (Module:ObjectsOut, NoTree);            } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                   } .
Module  = { Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, NoTree);
            Decls:Env           := Stats:Env;                                   } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
```

```
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);            } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                 } .

END DefTab
```

Attribute Diagram



Input

```
MODULE p
   DCL x
   MODULE q USE x USE y END
   MODULE r
      MODULE s USE x USE y END
      DCL x DCL y
      USE x USE y
   END
   USE x USE y
END
```

Messages

```
   3, 17: Error       identifier not declared: x
   3, 23: Error       identifier not declared: y
   5, 20: Error       identifier not declared: x
   5, 26: Error       identifier not declared: y
   9, 14: Error       identifier not declared: y
```

4.8.1.1. Import Clause

Description

This language adds import statements which explicitly allow objects from surrounding modules to become visible. This version of import statements specify the object to be imported as well as the module from which the object is to be imported. This module is searched for in the outer environment with respect to the import statement. All objects of a module may be imported by other modules.

Conditions

- Identifiers in sets of declared and imported objects may not appear multiply
- Every used object must be declared
- The first identifier in an import clause must refer to a module object

Remarks

Similar to the problem of qualified access using named blocks this solution also requires the access of a non-local attribute. Again it is necessary to look up an object in an environment which is not contained in the current scope hierarchy. This example as well as the following ones demonstrate the comfort of the presented solutions arising from well-defined attribute grammars and access to non-local attributes. Even cyclic import relations are possible, that means one module imports from a second one and vice versa. There is no need to worry about the order of the computation. The generated attribute evaluator takes care of this problem.

Imported objects are included in the set of objects associated with the current block in the same way as are declared objects. However, those objects are linked already into the sets of objects at the place of their declaration. Therefore newly created *reference objects* are used to refer to objects from other scopes using the node type *RefDecl*.

Concrete Syntax

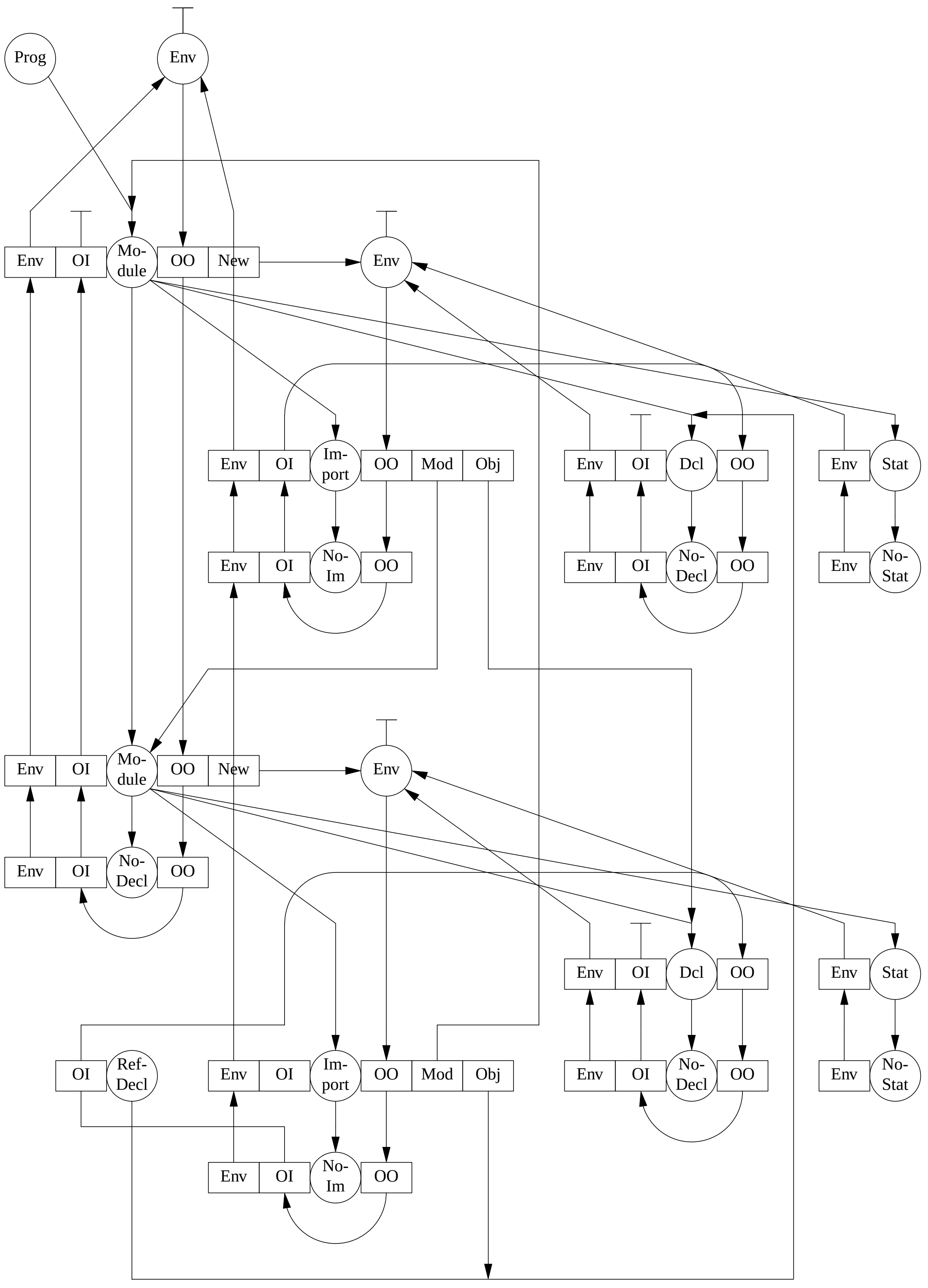
```
PROPERTY INPUT RULE

Prog            = Module .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
      Module    = ’MODULE’ Ident Imports Decls Stats ’END’ Next: Decls .
   > .
> .
Imports         = <
   NoImport     = .
   Import       = FROM Module: Ident ’IMPORT’ Ident Next: Imports .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Imports Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Module:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);             } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
Module  = { Tree := mModule (Ident:Ident, Ident:Position, Next:Tree, Imports:Tree,
                             Decls:Tree, Stats:Tree);                          } .
NoImport= { Tree := mNoImport ();                                              } .
Import  = { Tree := mImport (Module:Ident, Module:Position, Ident:Ident,
                             Ident:Position, Next:Tree);                       } .
NoStat  = { Tree := mNoStat ();                                                } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Module .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      Module    = Imports Decls Stats .
      RefDecl   = Decl .
   > .
> .
Imports         = <
   NoImport     = .
   Import       = [Ident: tIdent] [Pos: tPosition] [ObjIdent: tIdent]
                  [ObjPos: tPosition] Next: Imports .
```

```
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls Imports   = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Proc Module     = [NewEnv: tTree] .
Imports         = [OuterEnv: tTree] .
Env             = [Objects: tTree IN] Env IN .
Import          = [module: tTree] [Object: tTree] .

RULE

Prog    = { Module:ObjectsIn    := mNoDecl ();
            Module:Env          := mEnv (Module:ObjectsOut, NoTree);            } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            NewEnv              := mEnv (Decls:ObjectsOut, Env);
            Stats:Env           := NewEnv;
            Decls:Env           := NewEnv;                                      } .
Module  = { Decls:ObjectsIn     := mNoDecl ();
            Imports:ObjectsIn   := Decls:ObjectsOut;
            NewEnv              := mEnv (Imports:ObjectsOut, NoTree);
            Stats:Env           := NewEnv;
            Decls:Env           := NewEnv;
            Imports:OuterEnv    := Env;                                         } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);              } .
Import  = { module              := IdentifyWhole (Ident, OuterEnv);
            CHECK module->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK module->Kind == kModule
            => Error (Pos, "module required", Ident) AND_THEN
            CHECK Object->Kind != kNoDecl
            => Error (ObjPos, "identifier not declared", ObjIdent) AND_THEN
            CHECK IdentifyObjects (ObjIdent, ObjectsIn)->Kind == kNoDecl
            => Error (ObjPos, "identifier multiply declared", ObjIdent);
            Object      := module->Kind != kModule ? mNoDecl () : IdentifyObjects
               (ObjIdent, REMOTE module->Module.Decls => Decls:ObjectsOut);
            Next:ObjectsIn      := {
               if (Object->Kind != kNoDecl) {
                  Next:ObjectsIn = mRefDecl (ObjIdent, ObjPos, NoTree, Object);
                  Next:ObjectsIn->RefDecl.\ObjectsIn = ObjectsIn;
               } else
                  Next:ObjectsIn = ObjectsIn;
            };                                                                  } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                   } .

END DefTab
```

Attribute Diagram



Input

```
MODULE p
   DCL x
   MODULE q
      FROM r IMPORT x
      FROM r IMPORT y
      FROM r IMPORT z
      DCL y
      USE x USE y USE z
   END
   MODULE r
      FROM q IMPORT x
      FROM q IMPORT y
      FROM q IMPORT z
      FROM x IMPORT x
      FROM p IMPORT x
      DCL x
      USE x USE y USE z
   END
   USE x USE y
END
```

Messages

```
   5, 21: Error       identifier not declared: y
   6, 21: Error       identifier not declared: z
   8, 23: Error       identifier not declared: z
  11, 21: Error       identifier not declared: x
  13, 21: Error       identifier not declared: z
  14, 12: Error       module required: x
  15, 12: Error       identifier not declared: p
  17, 23: Error       identifier not declared: z
  19, 14: Error       identifier not declared: y
```

4.8.1.2. Export Clause

Description

While the previous example allowed all objects of a module to be imported by other modules, this language explicitly controls this. Export statements specify the objects that may be imported by other modules.

Conditions

- Identifiers in sets of declared and imported objects may not appear multiply
- Every used object must be declared
- The first identifier in an import clause must refer to a module object
- The objects appearing in export clauses must be declared

Remarks

The export clauses are used to construct a subset of the declared objects again using the node type *RefDecl*. The import clauses consult this subset in order to check the legality of the import.

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Module .
```

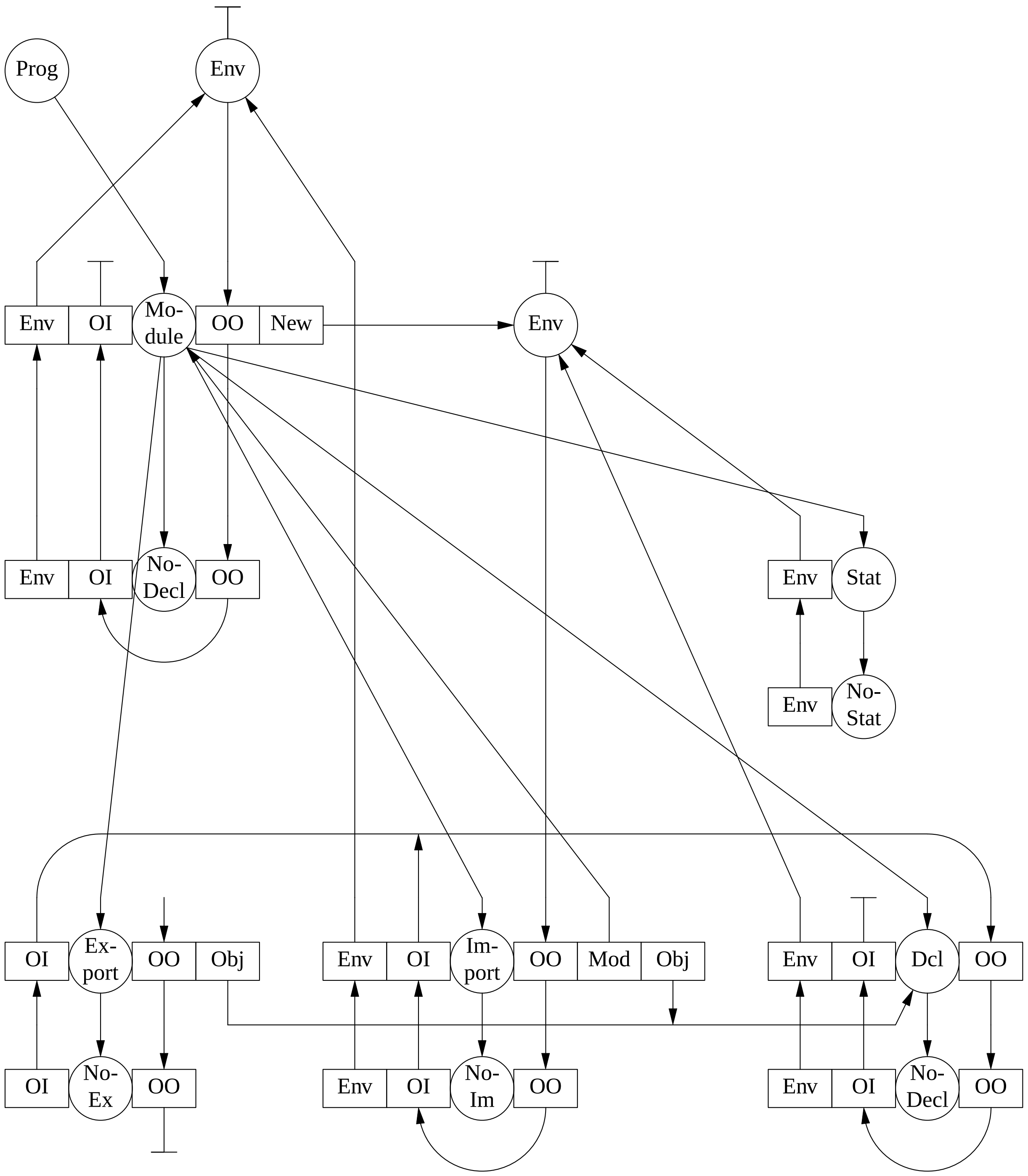
```
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident Next: Decls .
      Proc      = PROCEDURE Ident Decls Stats ’END’ Next: Decls .
      Module    = ’MODULE’ Ident Exports Imports Decls Stats ’END’ Next: Decls .
   > .
> .
Exports         = <
   NoExport     = .
   Export       = ’EXPORT’ Ident Next: Exports .
> .
Imports         = <
   NoImport     = .
   Import       = FROM Module: Ident ’IMPORT’ Ident Next: Imports .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Exports Imports Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Module:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                                } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree);             } .
Proc    = { Tree := mProc (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                           Stats:Tree);                                        } .
Module  = { Tree := mModule (Ident:Ident, Ident:Position, Next:Tree, Exports:Tree,
                             Imports:Tree, Decls:Tree, Stats:Tree);            } .
NoExport= { Tree := mNoExport ();                                              } .
Export  = { Tree := mExport (Ident:Ident, Ident:Position, Next:Tree);          } .
NoImport= { Tree := mNoImport ();                                              } .
Import  = { Tree := mImport (Module:Ident, Module:Position, Ident:Ident,
                             Ident:Position, Next:Tree);                       } .
NoStat  = { Tree := mNoStat ();                                                } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);            } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Module .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = .
      Proc      = Decls Stats .
      Module    = Exports Imports Decls Stats .
      RefDecl   = Decl .
   > .
> .
Exports         = <
   NoExport     = .
   Export       = [Ident: tIdent] [Pos: tPosition] Next: Exports .
> .
Imports         = <
   NoImport     = .
   Import       = [Ident: tIdent] [Pos: tPosition] [ObjIdent: tIdent]
                  [ObjPos: tPosition] Next: Imports .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls Imports Exports   = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Proc Module     = [NewEnv: tTree] .
Imports         = [Env: tTree] .
Env             = [Objects: tTree IN] Env IN .
Import          = [module: tTree] [Object: tTree] .
Export          = [Object: tTree] .

RULE

Prog    = { Module:ObjectsIn    := mNoDecl ();
            Module:Env          := mEnv (Module:ObjectsOut, NoTree);           } .
Proc    = { Decls:ObjectsIn     := mNoDecl ();
            NewEnv              := mEnv (Decls:ObjectsOut, Env);
            Stats:Env           := NewEnv;
            Decls:Env           := NewEnv;                                     } .
Module  = { Decls:ObjectsIn     := mNoDecl ();
            Imports:ObjectsIn   := Decls:ObjectsOut;
            NewEnv              := mEnv (Imports:ObjectsOut, NoTree);
            Stats:Env           := NewEnv;
            Decls:Env           := NewEnv;
            Exports:ObjectsIn   := Decls:ObjectsOut;
            Imports:Env         := Env;                                        } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);             } .
NoExport= { ObjectsOut          := mNoDecl ();                                 } .
Export  = { Object              := IdentifyObjects (Ident, ObjectsIn);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);
            ObjectsOut          := {
               if (Object->Kind != kNoDecl) {
                  ObjectsOut    = mRefDecl (Ident, Pos, NoTree, Object);
                  ObjectsOut->RefDecl.\ObjectsIn = Next:ObjectsOut;
               } else
```

```
                  ObjectsOut    = Next:ObjectsOut;
            };                                                                 } .
Import  = { module              := IdentifyWhole (Ident, Env);
            CHECK module->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK module->Kind == kModule
            => Error (Pos, "module required", Ident) AND_THEN
            CHECK Object->Kind != kNoDecl
            => Error (ObjPos, "identifier not exported", ObjIdent) AND_THEN
            CHECK IdentifyObjects (ObjIdent, ObjectsIn)->Kind == kNoDecl
            => Error (ObjPos, "identifier multiply declared", ObjIdent);
            Object      := module->Kind != kModule ? mNoDecl () : IdentifyObjects
               (ObjIdent, REMOTE module->Module.Exports => Decls:ObjectsOut);
            Next:ObjectsIn      := {
               if (Object->Kind != kNoDecl) {
                  Next:ObjectsIn = mRefDecl (ObjIdent, ObjPos, NoTree, Object);
                  Next:ObjectsIn->RefDecl.\ObjectsIn = ObjectsIn;
               } else
                  Next:ObjectsIn = ObjectsIn;
            };                                                                 } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                  } .

END DefTab
```

Attribute Diagram



Input

```
MODULE p
   DCL x
   MODULE q
      EXPORT y
      FROM r IMPORT x
      FROM r IMPORT y
      FROM r IMPORT z
      DCL y
      USE x USE y USE z
   END
   MODULE r
      EXPORT x
      EXPORT y
      FROM q IMPORT x
      FROM q IMPORT y
      FROM q IMPORT z
      FROM x IMPORT x
      FROM p IMPORT x
      FROM r IMPORT x
      DCL x DCL z
      USE x USE y USE z
   END
   USE x USE y
END
```

Messages

```
   6, 21: Error       identifier not exported: y
   7, 21: Error       identifier not exported: z
   9, 23: Error       identifier not declared: z
  13, 14: Error       identifier not declared: y
  14, 21: Error       identifier not exported: x
  16, 21: Error       identifier not exported: z
  17, 12: Error       module required: x
  18, 12: Error       identifier not declared: p
  19, 21: Error       identifier multiply declared: x
  23, 14: Error       identifier not declared: y
```

4.8.2. Separately Compiled Modules

In the previous examples the modules where contained in one compilation unit. Now every module is treated as a separate compilation unit. The modules are not allowed to be nested any more. Again, the existence of import and export clauses is assumed. A compiler has to inspect not only the source of the module to be compiled but also some information about the modules that are imported in order to perform a complete semantic analysis.

4.8.2.1. Without Symbol Files

Description

One solution for the separate compilation problem is to directly read and analyze the sources of the imported modules. The advantages of this solution are that no files are necessary that describe the exported objects of the modules and the modules may be compiled in any order.

Conditions

- Identifiers in sets of declared and imported objects may not appear multiply
- Every used object must be declared
- The first identifier in an import clause must refer to a module object
- The objects appearing in export clauses must be declared

Remarks

The solution uses a different main program and a different scanner as described in the next sections.

Main Program

The abstract syntax tree for a compilation unit does not contain only the actual module but also the modules that are (transitively) imported. Therefore the abstract syntax can be regarded as a list of trees described by the node type *Progs*. The main program takes care that all needed modules are scanned and parsed and included in the abstract syntax tree. For this purpose a list starting at the variable *UnitList* with elements of the node type *Unit* is maintained. It keeps all modules to be processed. Initially it holds the module that is to be compiled. After one module has been read, the procedure *UpdateList* is used to extend the list with additionally necessary modules. This algorithm iterates until all transitively imported modules have been processed. It is assumed that the modules are stored in files with the same names as the modules.

```
# include <sys/stat.h>
# include "Idents.h"
# include "Errors.h"
# include "Parser.h"
# include "Tree.h"
# include "Eval.h"

static tTree UnitList = NoTree;

static UpdateList ()                    /* extend unit list by imported units   */
{
   tTree Import = TreeRoot->Prog.Imports;
   while (Import->Kind == kImport) {    /* consider all imported units          */
      tIdent Ident = Import->Import.Ident;
      tTree ProgPtr = TreeRoot;
      tTree Unit = UnitList;

      while (ProgPtr->Kind == kProg)    /* has unit already been parsed ?       */
         if (ProgPtr->Prog.Ident == Ident) goto Found;
         else ProgPtr = ProgPtr->Prog.Next;

      while (Unit != NoTree)            /* is unit already in unit list ?       */
         if (Unit->Unit.Ident == Ident) goto Found;
         else Unit = Unit->Unit.Next;
                                        /* add unit to unit list                */
      UnitList = mUnit (Ident, Import->Import.Pos, UnitList);
Found:Import = Import->Import.Next;
   }
}

main (argc, argv)
   int argc; char * argv [];
{
   StoreMessages (rtrue);
   BeginScanner ();
   TreeRoot = mNoDecl ();
   UnitList = mUnit (MakeIdent (argv [1], strlen (argv [1])), NoPosition, UnitList);
   while (UnitList != NULL) {
      char SourceFile [256];
      struct stat buf;

      GetString (UnitList->Unit.Ident, SourceFile);
      Attribute.Position.Ident = UnitList->Unit.Ident;
      if (stat (SourceFile, & buf) < 0) {
         MessageI ("cannot access file", xxError, UnitList->Unit.Pos, xxIdent,
            (char *) & UnitList->Unit.Ident);
         UnitList = UnitList->Unit.Next;
      } else {
         BeginFile (SourceFile);
         (void) Parser ();
         UnitList = UnitList->Unit.Next;
         UpdateList ();
      }
   }
   TreeRoot = mRoot (ReverseTree (TreeRoot));
   BeginEval ();
   Eval (TreeRoot);
   WriteMessages (stderr);
   return 0;
}
```

Scanner

The scanner is distinguished from the normal ones by the definition of the type *tPosition*. This type is extended by a field that describes the source file. The DEFAULT section is adapted to this type. Accordingly, the module *Errors* that reports syntactic and semantic errors has been modified to include the file name into the source position.

```
EXPORT  {
# include "Idents.h"
# include "Position.h"

INSERT tScanAttribute
}

GLOBAL  {
# include "Idents.h"

INSERT ErrorAttribute
}

DEFAULT {
   char FileName [256];
   GetString (Attribute.Position.Ident, FileName);
   (void) fprintf (stderr, "\"%s\", %3d, %2d: Error       illegal character: %c\n",
      FileName, Attribute.Position.Line, Attribute.Position.Column, * TokenPtr);
}

DEFINE  digit   = {0-9} .
        letter  = {a-z A-Z} .

RULES

INSERT RULES #STD#

#STD# letter (letter | digit) * : {
   Attribute.Ident.Ident = MakeIdent (TokenPtr, TokenLength);
   return Ident;
}
```

Concrete Syntax

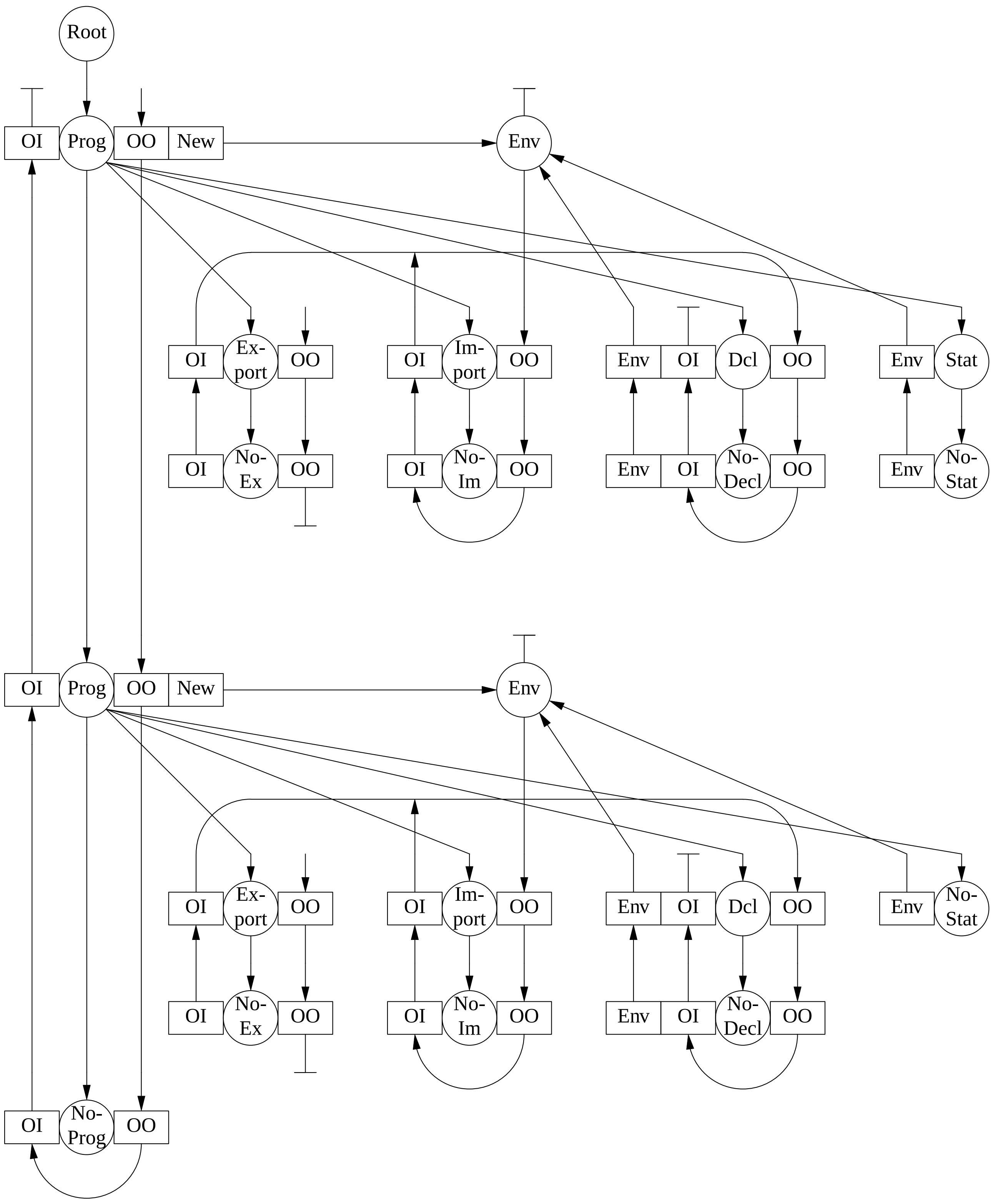
```
PROPERTY INPUT RULE

Prog            = ’MODULE’ Ident Exports Imports Decls Stats ’END’ .
Exports         = <
   NoExport     = .
   Export       = ’EXPORT’ Ident Next: Exports .
```

```
> .
Imports         = <
   NoImport     = .
   Import       = FROM Module: Ident 'IMPORT' Ident Next: Imports .
> .
Decls           = <
   NoDecl       = .
   Decl         = DCL Ident Next: Decls .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Exports Imports Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Ident:Ident, Ident:Position, TreeRoot,
                  Exports:Tree, Imports:Tree, Decls:Tree, Stats:Tree); };       } .
NoExport= { Tree := mNoExport ();                                               } .
Export  = { Tree := mExport (Ident:Ident, Ident:Position, Next:Tree);           } .
NoImport= { Tree := mNoImport ();                                               } .
Import  = { Tree := mImport (Module:Ident, Module:Position, Ident:Ident,
                             Ident:Position, Next:Tree);                        } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Decl    = { Tree := mDecl (Ident:Ident, Ident:Position, Next:Tree);             } .
NoStat  = { Tree := mNoStat ();                                                 } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Root            = Prog .
Exports         = <
   NoExport     = .
   Export       = [Ident: tIdent] [Pos: tPosition] Next: Exports .
> .
Imports         = <
   NoImport     = .
   Import       = [Ident: tIdent] [Pos: tPosition] [ObjIdent: tIdent]
                  [ObjPos: tPosition] Next: Imports .
> .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls REVERSE <
      Prog      = Exports Imports Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .
Unit            = [Ident: tIdent] [Pos: tPosition] Next: Unit .

MODULE DefTab

EVAL EXPORT {# include "DefTab.h"} GLOBAL {# include "DefTab.c"}

DECLARE

Progs Exports Imports Decls     = [Objects: tTree THREAD] .
Decls Stats     = [Env: tTree] .
Prog            = [NewEnv: tTree] .
Env             = [Objects: tTree IN] Env IN .
Import          = [prog: tTree] [Object: tTree] .
Export          = [Object: tTree] .

RULE

Root    = { Prog:ObjectsIn      := mNoDecl ();
            Prog:Env            := NoTree;                              } .
Prog    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            Decls:ObjectsIn     := mNoDecl ();
            Imports:ObjectsIn   := Decls:ObjectsOut;
            NewEnv              := mEnv (Imports:ObjectsOut, NoTree);
            Decls:Env           := NewEnv;
            Stats:Env           := NewEnv;
            Exports:ObjectsIn   := Decls:ObjectsOut;                    } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);      } .
NoExport= { ObjectsOut          := mNoDecl ();                          } .
Export  = { Object              := IdentifyObjects (Ident, ObjectsIn);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);
            ObjectsOut          := {
               if (Object->Kind != kNoDecl) {
                  ObjectsOut    = mRefDecl (Ident, Pos, NoTree, Object);
                  ObjectsOut->RefDecl.\ObjectsIn = Next:ObjectsOut;
               } else
                  ObjectsOut    = Next:ObjectsOut;
            };                                                          } .
Import  = { prog := IdentifyObjects
                       (Ident, REMOTE TreeRoot->Root.Prog => Prog:ObjectsOut);
            CHECK prog->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK prog->Kind == kProg
            => Error (Pos, "module required", Ident) AND_THEN
            CHECK Object->Kind != kNoDecl
            => Error (ObjPos, "identifier not exported", ObjIdent) AND_THEN
            CHECK IdentifyObjects (ObjIdent, ObjectsIn)->Kind == kNoDecl
            => Error (ObjPos, "identifier multiply declared", ObjIdent);
            Object      := prog->Kind != kProg ? mNoDecl () : IdentifyObjects
               (ObjIdent, REMOTE prog->Prog.Exports => Decls:ObjectsOut);
            Next:ObjectsIn      := {
               if (Object->Kind != kNoDecl) {
                  Next:ObjectsIn = mRefDecl (ObjIdent, ObjPos, NoTree, Object);
                  Next:ObjectsIn->RefDecl.\ObjectsIn = ObjectsIn;
               } else
                  Next:ObjectsIn = ObjectsIn;
            };                                                                 } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                  } .

END DefTab
```

Attribute Diagram



Input

```
MODULE q
   EXPORT y
   FROM r IMPORT x
   FROM r IMPORT y
   FROM r IMPORT z
   DCL y
   USE x USE y USE z
END

MODULE r
   EXPORT x
   EXPORT y
   FROM q IMPORT x
   FROM q IMPORT y
   FROM q IMPORT z
   FROM x IMPORT x
   FROM p IMPORT x
   FROM r IMPORT x
   DCL x DCL z
   USE x USE y USE z
END
```

Messages

```
"r",   3, 11: Error       identifier not declared: y
"r",   4, 18: Error       identifier not exported: x
"r",   6, 18: Error       identifier not exported: z
"r",   7,  9: Error       cannot access file: x
"r",   7,  9: Error       identifier not declared: x
"r",   8,  9: Error       cannot access file: p
"r",   8,  9: Error       identifier not declared: p
"r",   9, 18: Error       identifier multiply declared: x
"q",   4, 18: Error       identifier not exported: y
"q",   5, 18: Error       identifier not exported: z
"q",   7, 20: Error       identifier not declared: z
```

4.8.2.2. With Symbol Files

Description

An other solution for the separate compilation problem is to use so-called symbol files. Information about the exported objects of a module is written on a symbol file. If information about imported objects is needed it is read from the symbol files. The advantage of this solution is that scanning and parsing of the sources of the imported modules is not necessary. It is assumed that it is faster to read the symbol files.

Conditions

- Identifiers in sets of declared and imported objects may not appear multiply
- Every used object must be declared
- The first identifier in an import clause must refer to a module object

- The objects appearing in export clauses must be declared

Remarks

The solution uses a different main program and a different scanner. The scanner and the error handler are the same as in the previous example. The main program is described in the next section.

The names of the symbol files are constructed by adding the suffix ".s" to the module names.

Main Program

In principle the strategy is the same as in the previous example. The abstract syntax tree consists of a list of trees for all (transitively) imported modules. At the end of a compilation a symbol file is written. It contains a binary representation of the actual module. The statement part is omitted because only the declaration, import, and export parts are needed. The same iterative algorithm as in the previous example is used. The difference is that binary symbol files are read instead of source files using a scanner and parser. Reading and writing of the binary symbol files is performed with the reader and writer procedures *GetTree* and *PutTree* that are generated by the tool *ast*.

```
# include <sys/stat.h>
# include "Idents.h"
# include "Errors.h"
# include "Parser.h"
# include "Tree.h"
# include "Eval.h"

static tTree UnitList = NoTree;

static UpdateList ()                    /* extend unit list by imported units   */
{
   tTree Import = TreeRoot->Prog.Imports;
   while (Import->Kind == kImport) {    /* consider all imported units          */
      tIdent Ident = Import->Import.Ident;
      tTree ProgPtr = TreeRoot;
      tTree Unit = UnitList;

      while (ProgPtr->Kind == kProg)    /* has unit already been parsed ?       */
         if (ProgPtr->Prog.Ident == Ident) goto Found;
         else ProgPtr = ProgPtr->Prog.Next;

      while (Unit != NoTree)            /* is unit already in unit list ?       */
         if (Unit->Unit.Ident == Ident) goto Found;
         else Unit = Unit->Unit.Next;
                                        /* add unit to unit list                */
      UnitList = mUnit (Ident, Import->Import.Pos, UnitList);
Found:Import = Import->Import.Next;
   }
}

main (argc, argv)
   int argc; char * argv [];
{
   char SourceFile [256];
   FILE * f;

   StoreMessages (rtrue);
   BeginScanner ();
   BeginFile (argv [1]);
   Attribute.Position.Ident = MakeIdent (argv [1], strlen (argv [1]));
   (void) Parser ();
   UpdateList ();
   while (UnitList != NoTree) {
      struct stat buf;

      GetString (UnitList->Unit.Ident, SourceFile);
      (void) strcat (SourceFile, ".s");
      if (stat (SourceFile, & buf) < 0) {
         MessageI ("cannot access symbol file for", xxError, UnitList->Unit.Pos,
            xxIdent, (char *) & UnitList->Unit.Ident);
         UnitList = UnitList->Unit.Next;
      } else {
         tTree Prog;
         f = fopen (SourceFile, "r");
         Prog = GetTree (f);
         (void) fclose (f);
         Prog->Prog.Next = TreeRoot;
         TreeRoot = Prog;
         UnitList = UnitList->Unit.Next;
         UpdateList ();
      }
   }
   TreeRoot = mRoot (ReverseTree (TreeRoot));
   BeginEval ();
   Eval (TreeRoot);
   WriteMessages (stderr);

   {                                    /* write symbol file                    */
      tTree Prog = TreeRoot->Root.Prog;
      (void) strcpy (SourceFile, argv [1]);
      (void) strcat (SourceFile, ".s");
      f = fopen (SourceFile, "w");
      PutTree (f, mProg (Prog->Prog.Ident, Prog->Prog.Pos, mNoDecl (),
         Prog->Prog.Exports, Prog->Prog.Imports, Prog->Prog.Decls, mNoStat ()));
      (void) fclose (f);
   }
   return 0;
}
```

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = ’MODULE’ Ident Exports Imports Decls Stats ’END’ .
Exports         = <
   NoExport     = .
   Export       = ’EXPORT’ Ident Next: Exports .
> .
Imports         = <
   NoImport     = .
   Import       = FROM Module: Ident ’IMPORT’ Ident Next: Imports .
> .
Decls           = <
   NoDecl       = .
   Decl         = DCL Ident Next: Decls .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Ident Next: Stats .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .
```

```
MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Exports Imports Stats = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Ident:Ident, Ident:Position, mNoDecl (),
                  Exports:Tree, Imports:Tree, Decls:Tree, Stats:Tree); };       } .
NoExport= { Tree := mNoExport ();                                               } .
Export  = { Tree := mExport (Ident:Ident, Ident:Position, Next:Tree);           } .
NoImport= { Tree := mNoImport ();                                               } .
Import  = { Tree := mImport (Module:Ident, Module:Position, Ident:Ident,
                       Ident:Position, Next:Tree);                              } .
NoDecl  = { Tree := mNoDecl ();                                                 } .
Decl    = { Tree := mDecl (Ident:Ident, Ident:Position, Next:Tree);             } .
NoStat  = { Tree := mNoStat ();                                                 } .
Stat    = { Tree := mStat (Ident:Ident, Ident:Position, Next:Tree);             } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}
EXPORT { typedef tTree ttree; }
GLOBAL {
# define gettPosition(a) yyGet ((char *) & a, sizeof (a)); yyGetIdent (& a.Ident);
# define puttPosition(a) yyPut ((char *) & a, sizeof (a)); yyPutIdent (a.Ident);
}

PROPERTY INPUT RULE

Root            = Prog .
Exports         = <
   NoExport     = .
   Export       = [Ident: tIdent] [Pos: tPosition] Next: Exports .
> .
Imports         = <
   NoImport     = .
   Import       = [Ident: tIdent] [Pos: tPosition] [ObjIdent: tIdent]
                  [ObjPos: tPosition] Next: Imports .
> .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls REVERSE <
      Prog      = Exports Imports Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = [Ident: tIdent] [Pos: tPosition] Next: Stats .
> .
Unit            = [Ident: tIdent] [Pos: tPosition] Next: Unit .

MODULE DefTab

EVAL EXPORT {# include "DefTab.h"} GLOBAL {# include "DefTab.c"}
```

```
DECLARE

Progs Exports Imports Decls     = [Objects: ttree THREAD] .
Decls Stats     = [Env: ttree] .
Prog            = [NewEnv: ttree] .
Env             = [Objects: ttree IN] Env IN .
Import          = [prog: ttree] [Object: ttree] .
Export          = [Object: ttree] .

RULE

Root    = { Prog:ObjectsIn      := mNoDecl ();
            Prog:Env            := NoTree;                              } .
Prog    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            Decls:ObjectsIn     := mNoDecl ();
            Imports:ObjectsIn   := Decls:ObjectsOut;
            NewEnv              := mEnv (Imports:ObjectsOut, NoTree);
            Decls:Env           := NewEnv;
            Stats:Env           := NewEnv;
            Exports:ObjectsIn   := Decls:ObjectsOut;                    } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);      } .
NoExport= { ObjectsOut          := mNoDecl ();                          } .
Export  = { Object              := IdentifyObjects (Ident, ObjectsIn);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);
            ObjectsOut          := {
               if (Object->Kind != kNoDecl) {
                  ObjectsOut    = mRefDecl (Ident, Pos, NoTree, Object);
                  ObjectsOut->RefDecl.\ObjectsIn = Next:ObjectsOut;
               } else
                  ObjectsOut    = Next:ObjectsOut;
            };                                                          } .
Import  = { prog := IdentifyObjects
                       (Ident, REMOTE TreeRoot->Root.Prog => Prog:ObjectsOut);
            CHECK prog->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident) AND_THEN
            CHECK prog->Kind == kProg
            => Error (Pos, "module required", Ident) AND_THEN
            CHECK Object->Kind != kNoDecl
            => Error (ObjPos, "identifier not exported", ObjIdent) AND_THEN
            CHECK IdentifyObjects (ObjIdent, ObjectsIn)->Kind == kNoDecl
            => Error (ObjPos, "identifier multiply declared", ObjIdent);
            Object      := prog->Kind != kProg ? mNoDecl () : IdentifyObjects
               (ObjIdent, REMOTE prog->Prog.Exports => Decls:ObjectsOut);
            Next:ObjectsIn      := {
               if (Object->Kind != kNoDecl) {
                  Next:ObjectsIn = mRefDecl (ObjIdent, ObjPos, NoTree, Object);
                  Next:ObjectsIn->RefDecl.\ObjectsIn = ObjectsIn;
               } else
                  Next:ObjectsIn = ObjectsIn;
            };                                                          } .
Stat    = { CHECK IdentifyWhole (Ident, Env)->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);           } .

END DefTab
```

Attribute Diagram

see previous attribute diagram

Input

```
MODULE q
   EXPORT y
   FROM r IMPORT x
   FROM r IMPORT y
   FROM r IMPORT z
   DCL y
   USE x USE y USE z
END

MODULE r
   EXPORT x
   EXPORT y
   FROM q IMPORT x
   FROM q IMPORT y
   FROM q IMPORT z
   FROM x IMPORT x
   FROM p IMPORT x
   FROM r IMPORT x
   DCL x DCL z
   USE x USE y USE z
END
```

Messages

```
"r",   3, 11: Error       identifier not declared: y
"r",   4, 18: Error       identifier not exported: x
"r",   6, 18: Error       identifier not exported: z
"r",   7,  9: Error       cannot access symbol file for: x
"r",   7,  9: Error       identifier not declared: x
"r",   8,  9: Error       cannot access symbol file for: p
"r",   8,  9: Error       identifier not declared: p
"r",   9, 18: Error       identifier multiply declared: x
"q",   4, 18: Error       identifier not exported: y
"q",   5, 18: Error       identifier not exported: z
```

4.9. Classes

The features offered by object-oriented languages are not standardized in any way and vary widely. This section describes two simple abstractions of single and multiple inheritance. Both languages have classes consisting of attributes (instance variables) and methods. Subclasses inherit the entities of the superclasses and may overwrite the inherited methods by giving own definitions. Classes are not treated like modules that restrict the visibility for objects. Instead, they are considered as blocks. Therefore methods may access objects from outside the class similar to the access of objects in surrounding blocks. In this case there are two places to look for non-local objects: the superclass and the surrounding block. This conflict is resolved by looking in the superclass first and in the surrounding block next. The abstract declarations for objects other than classes are extended by a type that has to refer to a class.

4.9.1. Single Inheritance

Description

Single inheritance means that a class may at most have one superclass.

Conditions

- Identifiers may not be declared multiply
- The type object in a declaration must be declared
- The type object in a declaration must refer to a class
- The super class object in a class declaration must be declared
- The super class object in a class declaration must refer to a class
- Inherited objects may not be redeclared except for methods
- Every used object must be declared

Remarks

As explained above, there are two non-local scopes an object can be searched in: the super class and the surrounding block. In order to describe this situation the definition of the node type *Env* in the definition table is extended as follows:

```
   Envs  = <
      Env      = [Objects: tTree IN] Env: Envs IN .
      Env2     = Env1: Envs IN Env2: Envs IN .
   > .
```

The type *Env* is as before and refers to a set of objects and a surrounding scope. The type *Env2* refers to two non-local scopes.

Concrete Syntax

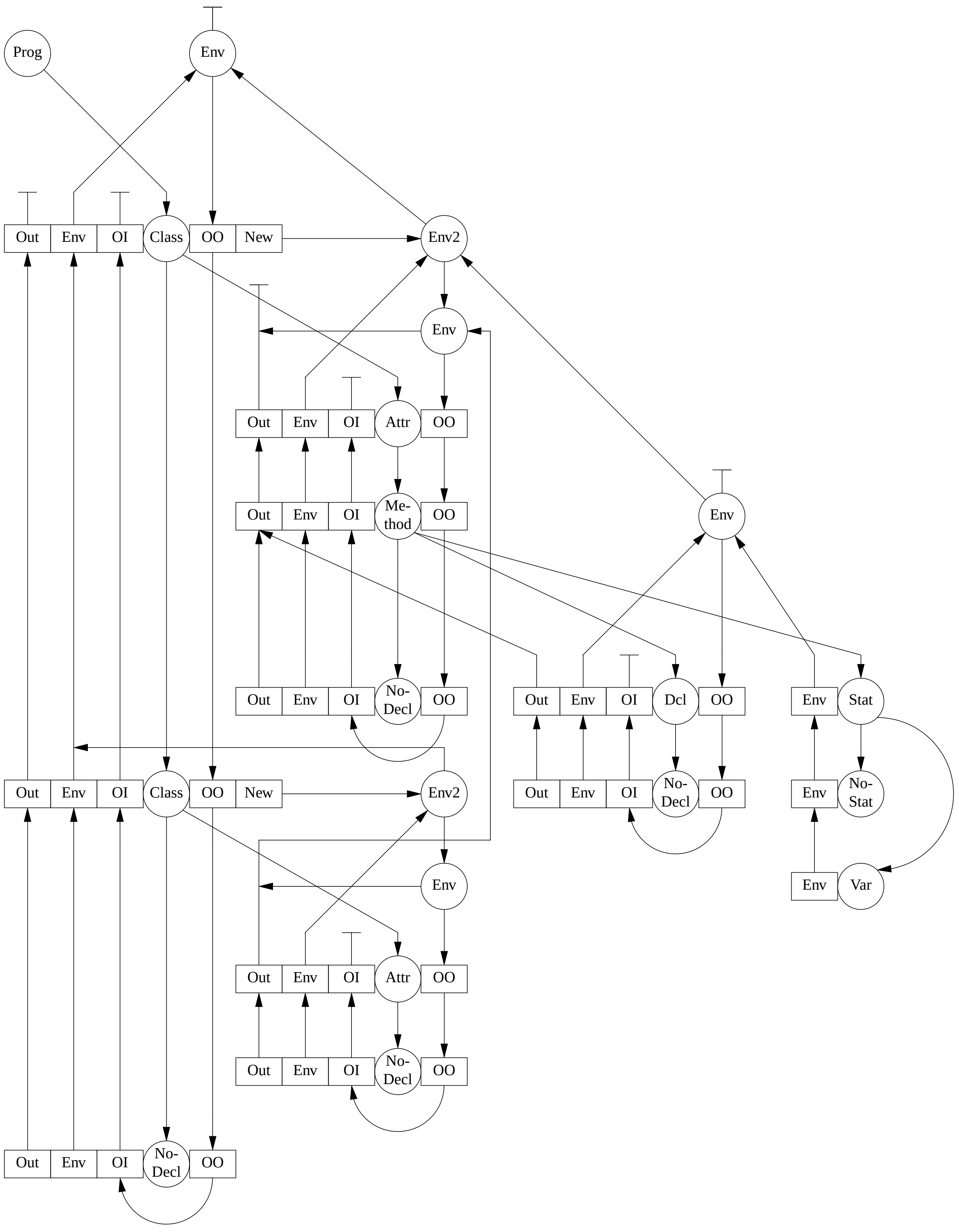
```
PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident ':' Class: Ident Next: Decls .
      Class     = CLASS Ident Super Features 'END' Next: Decls .
   > .
> .
Super           = <
   NoSuper      = .
   OneSuper     = '(' Ident ')' .
> .
Features        = <
   NoFeature    = .
   Attribute    = Ident Next: Features .
   Method       = PROCEDURE Ident Decls Stats 'END' Next: Features .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Designator Next:Stats .
> .
Designator      = <
   Var          = Ident .
   Qualify      = Designator ’.’ Ident .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                     } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Features Stats Designator = [Tree: tTree] .
        Super = [SuperId: tIdent] [Pos: tPosition] .

RULE

Prog    = { => { TreeRoot = mProg (Decls:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                               } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree, Class:Ident,
                          Class:Position);                                    } .
Class   = { Tree := mClass (Ident:Ident, Ident:Position, Next:Tree, Super:SuperId,
                            Super:Pos, Features:Tree);                        } .
NoSuper = { SuperId := NoIdent;         Pos := NoPosition;                    } .
OneSuper= { SuperId := Ident:Ident;     Pos := Ident:Position;                } .
NoFeature={ Tree := mNoDecl ();                                               } .
Attribute={ Tree := mAttribute (Ident:Ident, Ident:Position, Next:Tree);       } .
Method  = { Tree := mMethod (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                             Stats:Tree);                                     } .
NoStat  = { Tree := mNoStat ();                                               } .
Stat    = { Tree := mStat (Designator:Tree, Next:Tree);                       } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position);                       } .
Qualify = { Tree := mQualify (Ident:Ident, Ident:Position, Designator:Tree);    } .

END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = [Class: tIdent] [ClassPos: tPosition] .
      Class     = [Super: tIdent] [SuperPos: tPosition] Features: Decls .
      Attribute = .
      Method    = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Stats           = <
   NoStat       = .
   Stat         = Designator Next: Stats .
> .
Designator      = [Ident: tIdent] [Pos: tPosition] <
   Var          = .
   Qualify      = Designator .
> .

MODULE DefTab
```

```
EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Stats Designator  = [Env: tTree INH] .
Decls           = [OuterEnv: tTree] .
Class           = [NewEnv: tTree] .
Decl            = [Object: tTree] .
Designator      = [Object: tTree] .
Qualify         = [Class: tTree] .

RULE

Envs    = <
   Env  = [Objects: tTree IN] Env: Envs IN .
   Env2 = Env1: Envs IN Env2: Envs IN .
> .
Prog    = { Decls:ObjectsIn     := mNoDecl ();
            Decls:OuterEnv      := NoTree;
            Decls:Env           := mEnv (Decls:ObjectsOut, NoTree);           } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            Object              := mNoDecl ();
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
            => Error (Pos, "identifier multiply declared", Ident);            } .
Dcl     = { Object := IdentifyWhole (Class, Env);
            CHECK Object->Kind != kNoDecl
            => Error (ClassPos, "identifier not declared", Class) AND_THEN
            CHECK Object->Kind == kClass
            => Error (ClassPos, "class required", Class);                     } .
Class   = { Object := IdentifyWhole (Super, Env);
            CHECK Super == NoIdent || Object->Kind != kNoDecl
            => Error (SuperPos, "identifier not declared", Super) AND_THEN
            CHECK Super == NoIdent || Object->Kind == kClass
            => Error (SuperPos, "class required", Super);
            Features:ObjectsIn  := mNoDecl ();
            Features:OuterEnv   := Super == NoIdent || Object->Kind == kNoDecl ?
                                   NoTree : REMOTE Object => Class:NewEnv;
            NewEnv := mEnv2 (mEnv (Features:ObjectsOut, Features:OuterEnv), Env);
            Features:Env        := NewEnv;                                    } .
Attribute={ Object := IdentifyWhole (Ident, OuterEnv);
            CHECK Object->Kind == kNoDecl
            => Error (Pos, "identifier already inherited", Ident);            } .
Method  = { Object := IdentifyWhole (Ident, OuterEnv);
            CHECK Object->Kind == kNoDecl || Object->Kind == kMethod
            => Error (Pos, "identifier already inherited", Ident);
            Decls:ObjectsIn     := mNoDecl ();
            Stats:Env           := mEnv (Decls:ObjectsOut, Env);
            Decls:Env           := Stats:Env;                                 } .
Designator={Object              := mNoDecl ();                                } .
Var     = { Object              := IdentifyWhole (Ident, Env);
            CHECK Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                 } .
Qualify = { CHECK Designator:Object->Kind == kDcl ||
                  Designator:Object->Kind == kNoDecl
            => Error (Pos, "object required", Ident);
            Class               := Designator:Object->Kind == kDcl ?
               REMOTE Designator:Object => Dcl:Object : mNoDecl ();
            Object              := Class->Kind == kClass ?
               IdentifyWhole (Ident, REMOTE Class => Class:NewEnv) : mNoDecl ();
            CHECK Designator:Object->Kind != kDcl || Class->Kind != kClass ||
               Object->Kind != kNoDecl
            => Error (Pos, "identifier not declared", Ident);                  } .
END DefTab
```

Attribute Diagram



Input

```
CLASS c
   x
   y
   PROCEDURE p END
   PROCEDURE q END
END

CLASS d (c)
   y
   z
   z
   PROCEDURE q USE z USE x USE q USE p USE r USE C USE c USE H USE C.x USE D.x
               USE D.q USE C.r USE r.C END
   PROCEDURE r END
   PROCEDURE r END
   PROCEDURE y END
   PROCEDURE x END
END

DCL C: c
DCL D: d
DCL E: d
CLASS e END
CLASS e (g) END
CLASS f (C) END
DCL E: c
DCL F: g
DCL G: C
```

Messages

```
   9,  4: Error       identifier already inherited: y
  11,  4: Error       identifier multiply declared: z
  12, 62: Error       identifier not declared: H
  13, 30: Error       identifier not declared: r
  13, 38: Error       object required: C
  15, 14: Error       identifier multiply declared: r
  16, 14: Error       identifier multiply declared: y
  16, 14: Error       identifier already inherited: y
  17, 14: Error       identifier already inherited: x
  24,  7: Error       identifier multiply declared: e
  24, 10: Error       identifier not declared: g
  25, 10: Error       class required: C
  26,  5: Error       identifier multiply declared: E
  27,  8: Error       identifier not declared: g
  28,  8: Error       class required: C
```

4.9.2. Multiple Inheritance

Description

Multiple inheritance allows for an arbitrary number of superclasses.

Conditions

see previous section

Remarks

see previous section

Concrete Syntax

```
PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = <
      Dcl       = DCL Ident ’:’ Class: Ident Next: Decls .
      Class     = CLASS Ident ’(’ Supers ’)’ Features ’END’ Next: Decls .
   > .
> .
Supers          = <
   NoSuper      = .
   Super        = Ident Next: Supers .
> .
Features        = <
   NoFeature    = .
   Attribute    = Ident Next: Features .
   Method       = PROCEDURE Ident Decls Stats ’END’ Next: Features .
> .
Stats           = <
   NoStat       = .
   Stat         = USE Designator Next:Stats .
> .
Designator      = <
   Var          = Ident .
   Qualify      = Designator ’.’ Ident .
> .

Ident           : [Ident: tIdent] { Ident       := NoIdent;                    } .

MODULE Tree

PARSER GLOBAL   {# include "Tree.h"}

DECLARE Decls Features Supers Stats Designator = [Tree: tTree] .

RULE

Prog    = { => { TreeRoot = mProg (Decls:Tree); };                            } .
NoDecl  = { Tree := mNoDecl ();                                               } .
Dcl     = { Tree := mDcl (Ident:Ident, Ident:Position, Next:Tree, Class:Ident,
                          Class:Position);                                    } .
Class   = { Tree := mClass (Ident:Ident, Ident:Position, Next:Tree, Supers:Tree,
                            Features:Tree);                                   } .
NoSuper = { Tree := mNoSuper ();                                              } .
Super   = { Tree := mSuper (Ident:Ident, Ident:Position, Next:Tree);          } .
NoFeature={ Tree := mNoDecl ();                                               } .
Attribute={ Tree := mAttribute (Ident:Ident, Ident:Position, Next:Tree);      } .
Method  = { Tree := mMethod (Ident:Ident, Ident:Position, Next:Tree, Decls:Tree,
                             Stats:Tree);                                     } .
NoStat  = { Tree := mNoStat ();                                               } .
Stat    = { Tree := mStat (Designator:Tree, Next:Tree);                       } .
Var     = { Tree := mVar (Ident:Ident, Ident:Position);                       } .
Qualify = { Tree := mQualify (Ident:Ident, Ident:Position, Designator:Tree);  } .
```

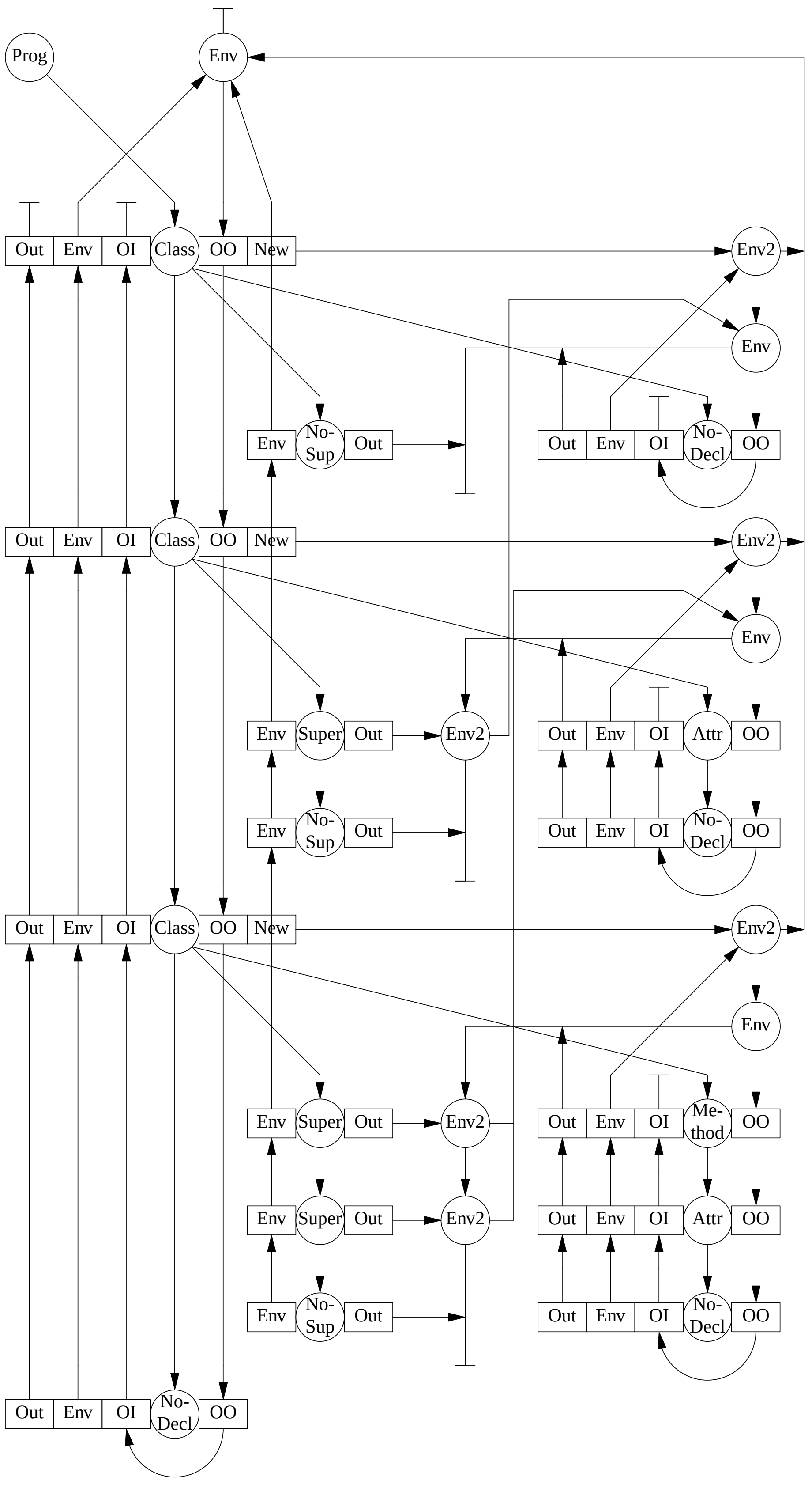
```
END Tree
```

Attribute Grammar

```
TREE IMPORT {
# include "Idents.h"
# include "Scanner.h"
}

PROPERTY INPUT RULE

Prog            = Decls .
Decls           = <
   NoDecl       = .
   Decl         = [Ident: tIdent] [Pos: tPosition] Next: Decls <
      Dcl       = [Class: tIdent] [ClassPos: tPosition] .
      Class     = Supers Features: Decls .
      Attribute = .
      Method    = Decls Stats .
      RefDecl   = Decl .
   > .
> .
Supers          = <
   NoSuper      = .
   Super        = [Ident: tIdent] [Pos: tPosition] Next: Supers .
> .
Stats           = <
   NoStat       = .
   Stat         = Designator Next: Stats .
> .
Designator      = [Ident: tIdent] [Pos: tPosition] <
   Var          = .
   Qualify      = Designator .
> .

MODULE DefTab

EVAL GLOBAL {# include "DefTab.c"}

DECLARE

Decls           = [Objects: tTree THREAD] .
Decls Supers Stats Designator   = [Env: tTree INH] .
Decls           = [OuterEnv: tTree] .
Class           = [NewEnv: tTree] .
Decl Super Designator   = [Object: tTree] .
Qualify         = [Class: tTree] .
Supers          = [EnvOut: tTree] .

RULE

Envs    = <
   Env  = [Objects: tTree IN] Env: Envs IN .
   Env2 = Env1: Envs IN Env2: Envs IN .
> .
Prog    = { Decls:ObjectsIn     := mNoDecl ();
            Decls:OuterEnv      := NoTree;
            Decls:Env           := mEnv (Decls:ObjectsOut, NoTree);             } .
Decl    = { Next:ObjectsIn      := DEP (SELF, ObjectsIn);
            ObjectsOut          := Next:ObjectsOut;
            Object              := mNoDecl ();
            CHECK IdentifyObjects (Ident, ObjectsIn)->Kind == kNoDecl
```

```
               => Error (Pos, "identifier multiply declared", Ident);            } .
Dcl      = { Object := IdentifyWhole (Class, Env);
               CHECK Object->Kind != kNoDecl
               => Error (ClassPos, "identifier not declared", Class) AND_THEN
               CHECK Object->Kind == kClass
               => Error (ClassPos, "class required", Class);                     } .
Class    = {
               Features:ObjectsIn  := mNoDecl ();
               Features:OuterEnv   := Supers:EnvOut;
               NewEnv := mEnv2 (mEnv (Features:ObjectsOut, Supers:EnvOut), Env);
               Features:Env        := NewEnv;                                    } .
Supers   = { EnvOut              := NoTree;                                      } .
Super    = { Object              := IdentifyWhole (Ident, Env);
               CHECK Object->Kind != kNoDecl
               => Error (Pos, "identifier not declared", Ident) AND_THEN
               CHECK Object->Kind == kClass
               => Error (Pos, "class required", Ident);
               EnvOut              := Object->Kind != kClass ? Next:EnvOut :
                  mEnv2 (REMOTE Object => Class:NewEnv, Next:EnvOut);            } .
Attribute={ Object               := IdentifyWhole (Ident, OuterEnv);
               CHECK Object->Kind == kNoDecl
               => Error (Pos, "identifier already inherited", Ident);            } .
Method   = { Object              := IdentifyWhole (Ident, OuterEnv);
               CHECK Object->Kind == kNoDecl || Object->Kind == kMethod
               => Error (Pos, "identifier already inherited", Ident);
               Decls:ObjectsIn     := mNoDecl ();
               Stats:Env           := mEnv (Decls:ObjectsOut, Env);
               Decls:Env           := Stats:Env;                                 } .
Designator={Object               := mNoDecl ();                                  } .
Var      = { Object              := IdentifyWhole (Ident, Env);
               CHECK Object->Kind != kNoDecl
               => Error (Pos, "identifier not declared", Ident);                 } .
Qualify  = { CHECK Designator:Object->Kind == kDcl ||
                     Designator:Object->Kind == kNoDecl
               => Error (Pos, "object required", Ident);
               Class               := Designator:Object->Kind == kDcl ?
                  REMOTE Designator:Object => Dcl:Object : mNoDecl ();
               Object              := Class->Kind == kClass ?
                  IdentifyWhole (Ident, REMOTE Class => Class:NewEnv) : mNoDecl ();
               CHECK Designator:Object->Kind != kDcl || Class->Kind != kClass ||
                  Object->Kind != kNoDecl
               => Error (Pos, "identifier not declared", Ident);                 } .

END DefTab
```

Attribute Diagram



Input

```
CLASS a ()
   x
   y
   PROCEDURE p END
   PROCEDURE q END
END

CLASS b (a)
   z
   PROCEDURE q END
   PROCEDURE r END
END

CLASS c (a)
   z
   PROCEDURE q END
   PROCEDURE r END
END

CLASS d (b c)
   y
   z
   u
   PROCEDURE q END
   PROCEDURE r END
END
```

Messages

```
  21,  4: Error       identifier already inherited: y
  22,  4: Error       identifier already inherited: z
```

Appendix 1: Abbreviations for Node Types

Attr	Attribute
Class	
Compound	
Const	
DList	
Dcl	
Decl	
Env	
Env2	
Export	
Goto	
Import	
Label	
LabelDecl	
Method	
Module	
NoDList	
NoDecl	
NoEx	NoExport
NoIm	NoImport
NoProg	
NoStat	
NoSup	NoSuper
Proc	
Prog	
Qualify	
RefDecl	
Root	
Stat	
Super	
Use	
Var	

Appendix 2: Abbreviations for Attribute Names

Env	Env
Id	Ident
Mod	Module
New	NewEnv
OI	ObjectsIn
OO	ObjectsOut
Obj	Object

Out	OuterEnv, EnvOut
Pos	Pos

References

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, *LNCS 477*, (Oct. 1990), 106-116, Springer Verlag.

[Gro90] J. Grosch, Object-Oriented Attribute Grammars, in *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmanci and E. Gelenbe (ed.), Cappadocia, Nevsehir, Turkey, Oct. 1990, 807-816.

[Gro92] J. Grosch, Transformation of Attributed Trees Using Pattern Matching, *LNCS 641*, (Oct. 1992), 1-15, Springer Verlag.

[Groa] J. Grosch, Ag - An Attribute Evaluator Generator, Cocktail Document No. 16, CoCoLab Germany.

[Grob] J. Grosch, Puma - A Generator for the Transformation of Attributed Trees, Cocktail Document No. 26, CoCoLab Germany.

[GrV] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Cocktail Document No. 8, CoCoLab Germany.

[Groa] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Cocktail Document No. 15, CoCoLab Germany.

[Grob] J. Grosch, Rex - A Scanner Generator, Cocktail Document No. 5, CoCoLab Germany.

[Groc] J. Grosch, Preprocessors, Cocktail Document No. 24, CoCoLab Germany.

[Grod] J. Grosch, Multiple Inheritance in Object-Oriented Attribute Grammars, Cocktail Document No. 28, CoCoLab Germany.

[Groe] J. Grosch, Toolbox Introduction, Cocktail Document No. 25, CoCoLab Germany.

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13*, 3 (1980), 229-256.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2,* 2 (June 1968), 127-146.

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

[Vog93] H. H. Vogt, *Higher Order Attribute Grammars*, PhD Thesis, University of Utrecht, Feb. 1993.

Contents