Incremental Semantic Analysis

Görel Hedin

CODEN: LUTEDX/(TECS-1003)/1-276/(1992)

Lund, March 1992

Department of Computer Science
Lund University
Box118
S-221 00 Lund, Sweden

E-mail: Gorel.Hedin@dna.lth.se

Front cover: The god Tor with his hammer Mjølner

Abstract

Semantic analysis is a central part of the compilation process. The main subproblems include name analysis, type checking, and detection of static-semantic errors. In an interactive programming environment it is useful to perform the semantic analysis incrementally, keeping the static-semantic information up to date while the program is edited. This allows advanced browsing and editing facilities to be implemented, based on the semantic information. Furthermore, incremental semantic analysis is a prerequisite for making also the rest of the compilation process incremental in order to reduce the turnaround time between editing and execution.

This work is directed towards incremental semantic analysis for object-oriented programming languages. These languages have comparatively complex static-semantics which could not be adequately handled with earlier techniques such as attribute grammars.

The main contribution of this work is a new technique for developing incremental semantic analyzers: *Door Attribute Grammars*. This technique extends standard attribute grammars by allowing objects and references to be specified as part of the attribution of a syntax tree. This extension results in space-efficient attributions for which incremental updates can be performed efficiently. In particular, the complex naming semantics of object-oriented languages can be handled in a straight forward way by attributing the tree with explicit visibility graphs built using objects and references.

The price for using objects and references in an attribution is that non-local attribute dependencies are introduced which prevent incremental attribute evaluators to be generated completely automatically from the grammar. We solve this problem by splitting the grammar in two parts: one part (the main grammar) which can be treated by automatic methods, and another part (the door package) for which a manual, but systematic, implementation technique is developed. A door package can implement general aspects of a family of programming languages. To specify a new language in the supported family it suffices to write a main grammar, using the door package as a tool box.

The techniques have been developed and tested in practice. A complete incrementally compiling environment has been built: *Mjølner/Orm*, which currently supports the major part of Simula.

Acknowledgments

The research reported in this thesis was carried out within the Programming Environments group at the Department of Computer Science at Lund University. First of all, I want to thank Boris Magnusson, my advisor and leader of this group. He introduced me to the field of object-oriented programming languages and compilers. His support and insight has been invaluable for me throughout my thesis work, not the least during this last intense phase.

The experimental work with the Orm system was done jointly in our group, and I want to thank all who participated in this development. It has been most stimulating and enjoyable to work with you. Thank you Mats Bengtsson, Lars-Ove Dahlin, Göran Fries, Anders Gustavsson, Sten Minör, Dan Oscarsson, and Magnus Taube. In particular, I want to express my gratitude to Lars-Ove Dahlin who has been my partner in the semantic analysis business. He designed and implemented the semantic kernel in Orm. The lookup technique based on search paths and traces is due to him. I would also like to thank the newer members of our team.

The work with Orm was carried out as part of the Nordic Mjølner project. The participation in this project has been very stimulating and I want to thank the other Mjølner participants for their encouragement and interest in my work.

I also want to thank my other colleagues at the department for their help in various ways. Special thanks to Sten Henriksson for reading an early draft of my thesis and giving valuable comments. Thanks to Oskar Permvall for being a living dictionary and answering many strange questions. Thanks to Anne-Marie Westerberg for helping me with various practical things.

Financially, this work has been supported by Nordisk Industrifond (The Nordic Fund for Technology and Industrial Development) and NUTEK (The National Swedish Board for Technical Development).

Contents

Chapter 1
Introduction

The development of high-performance low-cost graphical workstations during the last decade has made it possible to assign considerable amounts of computing power to advanced user interfaces. This development has revolutionized the way people work with computers. Today we are accustomed to interactive applications where information is presented and manipulated in an intuitive way. In order to build such interactive applications, *incremental computation* is fundamental: the information presented to the user is modified according to the user's actions, and derived information is updated in the process. The specific application area of interest in this thesis is interactive systems for program development.

The use of interactive and incremental techniques can simplify and speed up the programming process in several respects: By allowing the programmer to browse and manipulate programs in terms of their inherent structure, by performing compilation and linking automatically as needed, and by giving integrated support across editing and execution, allowing the programmer to switch easily between these activities. A basic approach to building such systems is *structure-oriented programming environments* [DEFH87] where programs are represented internally as abstract syntax trees described by a context-free grammar. A central component in such environments is an *incremental semantic analyzer* which computes and incrementally updates static-semantic information derived from the program. The static-semantic information is represented as an attribution of the abstract syntax tree and is incrementally updated while the user edits the program. This allows advanced browsing and editing facilities to be implemented, based on the semantic information. Furthermore, incremental semantic analysis is a prerequisite for making also the rest of the compilation process incremental in order to reduce the turnaround time between editing and execution.

Goals

The goal of the research presented in this thesis was to develop techniques allowing efficient incremental static-semantic analyzers to be constructed. In particular, we were interested in enabling such analysis to be performed for *object-oriented languages*. The static-semantic rules for object-oriented languages are comparatively complex, and existing techniques for developing incremental analyzers were not adequate for these languages.

The research presented here has been conducted within the *Mjølner project*, an experimental research project on building programming environments for object-oriented languages [DLMM87]. The project ran from 1986 through 1991 and involved universities and industrial partners in Denmark, Finland, Norway, and Sweden.

One of the environments built in this project is *Orm* [MHM+90], a complete incrementally compiling environment, currently supporting the major part of Simula [DMN68]. This system was implemented by the author and colleagues at the Department of Computer Science at Lund University. The system has been used on an experimental basis in courses on introductory programming, compiler construction, and programming environments, at Lund University and a few other universities. The problems which needed to be addressed in order to build environments like Orm constituted the motivation for the development of the techniques proposed in this thesis.

Method

The most well-studied high-level technique for developing incremental analyzers is the one based on *attribute grammars* [Knu68]. Incremental static-semantic analyzers can be generated automatically from such grammars [Rep84]. The principle idea of attribute grammars is very useful and attractive for the implementation of incremental systems: the attribution of a syntax tree is described declaratively, and an incrementally updating attribute evaluator can be automatically derived from the specification. This gives robust implementations which are easy to change and maintain.

However, *incremental name analysis*, an important part of incremental static-semantic analysis, cannot be done effectively using the standard approach of [Rep84]. These problems are even more pronounced for object-oriented languages, due to their more complex naming rules. Attribute grammars are not well suited for describing such rules and this leads to poor incremental behavior, even when using updating algorithms which are optimal within the context of attribute grammars. We find the most serious problem with attribute grammars to be that the attributions they can define are too limited to be practical.

The approach used in this thesis has been to develop a new technique for incremental computation which preserves the declarative property of standard AGs, but extends the range of attributions which can be described. This is done by extending the attribute grammar formalism to support specification of *objects* and *references* as attributes of the syntax tree. This is a radical step away from standard AGs which restrict the attributions to have value semantics only. The proposed technique is called *Door Attribute Grammars*, after the special “door objects” acting as an interface between the syntax tree and its object attribution. The use of objects and references allows very flexible structuring of the static-semantic information and makes it possible to describe the static-semantics of object-oriented languages in a straight-forward manner. The resulting attributions are space efficient and suited for efficient incremental updates.

The introduction of objects and references in the syntax tree attribution results in non-local dependencies which, in general, prevent attribute evaluators to be generated automatically from the grammar. We have addressed this problem by splitting the Door AG into a *main grammar* and a *door package*. The main grammar is very similar to a standard AG and can be implemented automatically by standard methods. The door package isolates the non-local dependencies and must be implemented by hand. We have developed a systematic implementation technique to support this manual construction.

Door packages are independent of the context-free grammar, and can be designed to handle general static-semantic properties of a family of languages. This allows a door package to be used as a tool box by many different main grammars, in order to implement different languages. Thus, although the door package must be implemented by hand, the same door package can be used for many programming languages, and attribute evaluators for each of these languages can be generated automatically from the main grammars.

An object-oriented variant of standard AGs has been developed to be used for specifying the main grammars of Door AGs. This allows the modelling and specialization principles of object-oriented programming to be applied to grammars, resulting in more compact and readable specifications compared to the traditional formalism.

Results

The main contribution of this thesis is the Door Attribute Grammar formalism and the implementation techniques developed for constructing incremental evaluators for such grammars. The techniques have been tested in practice. A precursory form of the Door AGs is used in the Orm system which supports the major part of Simula. The Door AGs and the implementation techniques as presented in this thesis have also been implemented and tested for the key problems appearing in static-semantic analysis for object-oriented languages. These applications of Door AGs show that the technique is suitable for practical construction of highly interactive program development environments.

1.1 Thesis outline

This thesis is structured in four major parts: background, object-oriented attribute grammars, door attribute grammars, and conclusions.

Background

- Chapter 2: The Orm Environment
  An overview is given of the Orm programming environment with particular emphasis on the aspects related to static-semantics. The requirements on incremental techniques for building such environments are discussed.

- Chapter 3: Incremental Static-Semantic Analysis
  Major problems in incremental static-semantic analysis are reviewed with particular emphasis on the problems appearing for object-oriented languages. Existing techniques for incremental name analysis are reviewed.
- Chapter 4: Standard Attribute Grammars
  Attribute grammars are reviewed and the problems of using them as a tool for incremental static-semantic analysis are identified.
- Chapter 5: A Basic Object-Oriented Specification Language
  A basic object-oriented specification language, OOSL, is introduced. This language will be used for specifying object-oriented AGs and Door AGs. The language was developed specifically for this thesis and is useful both for specifying the declarative grammars and the imperative incremental update algorithms.

Object-oriented Attribute Grammars

- Chapter 6: Object-Oriented Attribute Grammars
  An object-oriented reformulation of standard AGs is introduced. It is shown how the advantages of object-oriented description techniques can be applied to grammars, in particular how behavior can be defined at suitable levels of generalization.
- Chapter 7: Attribute Evaluation Techniques
  Techniques for demand-driven and 1-visit data-driven evaluation of standard AGs are discussed. These techniques will be used as the basis for attribute evaluation in the main grammars of Door AGs. It is shown how the implementation of evaluation algorithms is simplified by using an object-oriented implementation language. A new technique employing static skipping of visit instructions is introduced.

Door Attribute Grammars

- Chapter 8: Door Attribute Grammars
  Door Attribute Grammars are introduced. An example is given of how Door AGs can be used to specify the static-semantics of a simple block structured language. The technique is compared to standard AGs.
- Chapter 9: Door AG Implementation, part I
  The principles for implementation of visit-oriented incremental evaluators for Door AGs are described. A systematic technique for analyzing non-local dependencies is developed.
- Chapter 10: Door AG Implementation, part II
  An incremental visit-oriented evaluation algorithm for Door AGs is given and a systematic technique for constructing visit procedures for door packages is developed. It is shown how incremental algorithms for standard AGs can be adapted for evaluation of main grammars.

- Chapter 11: Advanced Attributions
  It is shown how some central problems arising in object-oriented languages can be handled by Door AGs. Problems treated include subclassing, remote access, and type-checking of reference assignments.

Conclusions

- Chapter 12: Evaluation
  This chapter reports on practical experience from using Door AGs in the Orm environment. We discuss actual time consumption and estimates of space consumption.
- Chapter 13: Conclusions and Future Work
  This chapter summarizes the contributions of this work and some possibilities for future work are suggested.

Chapter 2
The Orm Environment

This chapter gives an overview of the Orm programming environment with focus on issues related to incremental static-semantic analysis. The requirements on incremental techniques for building such environments are discussed.

2.1 Introduction

Orm is an interactive programming environment based on structure-oriented editing and incremental compilation. The environment includes both a programming level and a meta level. The programming level supports program development, from editing to execution. The compilation tasks such as static-semantic analysis, code generation, and loading are all done incrementally and automatically as needed by the system. The meta level supports structure-oriented editing of grammars. These grammars describe the abstract syntax, concrete syntax, static semantics, and code generation for a specific language. The Orm environment is aimed towards object-oriented languages and the current version supports the major part of Simula. This is in contrast to earlier structure-oriented environments like the Cornell Program Synthesizer [TR81], Gandalf [MF81], DICE [Fri84], and Pecan [Rei84], which are all aimed towards procedural languages like C and Pascal.

The major goal of Orm is to provide advanced interactive support for program development, in particular:

- Allow programs to be edited and browsed at a semantic level, and not only at a syntactic or textual level.
- Allow editing and execution to be mixed freely, and also to continue execution after program changes.
- Allow interactive observation of the object structures present during execution.
- Support the full software development cycle including design and documentation, version and variant control.

An additional goal of Orm is to support interactive *language* development, by allowing the user to edit both programs and grammars at the same time in order to interactively try out changes to a language.

Interaction in Orm can be said to be “object-oriented” rather than “tool-oriented”. The goal is to allow the user to interact directly with programs, executions, and grammars, rather than via explicit tools. To support this “object-oriented” interaction style, the user interface of Orm is based on direct manipulation. Objects of interest to the user are presented as icons/windows, similar in flavor to the Star office automation system [SIKV82] and to its successors like the Macintosh finder. In contrast to Star and Macintosh, the interface of Orm is based on a *hierarchical* window system [Osc89] which allows windows (objects) to be shown in their local context. For example, a program window can contain class windows which in turn can contain procedure windows [HM88].

In addition to the support for programming, which we will discuss in more detail below, Orm supports version- and revision control of the programs and grammars created by the system [Gus90]. There is also a future goal of supporting programming of real-time systems. This has motivated research on real-time garbage collection algorithms with low predictable response times [Ben90].

2.1.1 Implementation status

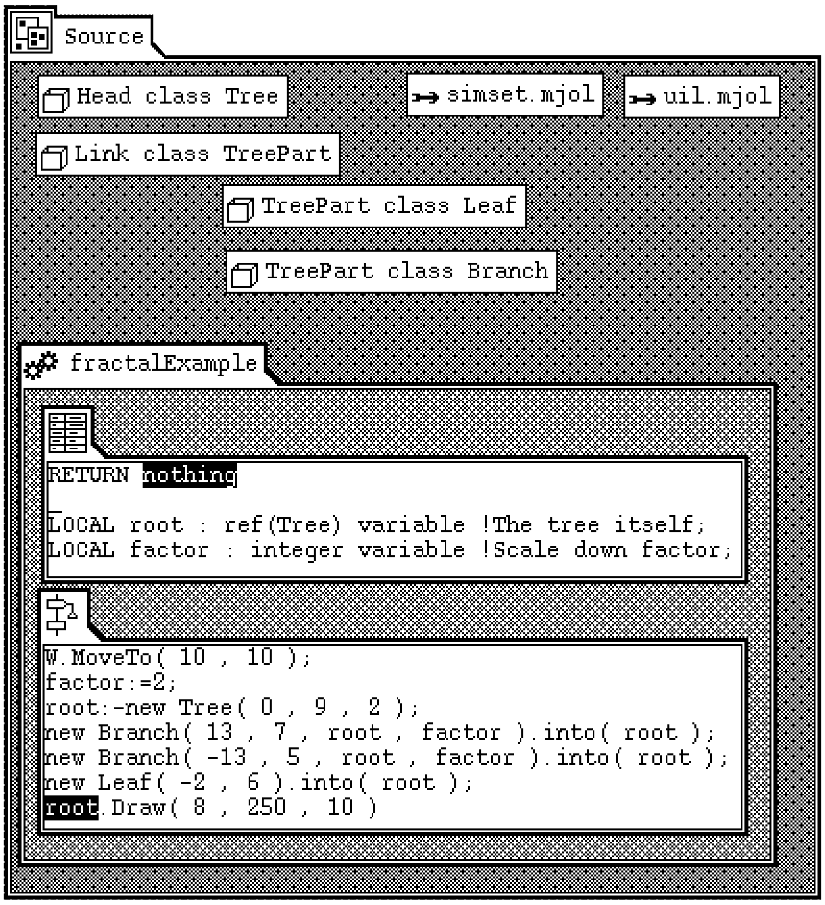
Orm is implemented in Simula and runs on SUN SPARC-stations. The implementation consists of approximately 100 000 lines of code. It is a complete environment supporting both editing and execution of programs. The executing program runs in a separate UNIX process and communicates with the environment over a pipe. Currently, the execution is performed by an interpreter, interpreting intermediate three-address instructions, but a binary run-time system is under development. The largest programs written so far in Orm are about 1000 lines. The largest grammar is the grammar for Simula, consisting of about 100 productions.

2.2 Programming level

2.2.1 Source program

A program in Orm is presented to the user as a hierarchy of nested classes and procedures, each shown as a window. The window nesting reflects the block nesting structure of the program. Local declarations and statements are presented inside the blocks as a textual unparsing of syntax trees. Three editing mechanisms are available: menu-driven structure-oriented editing, textual input which is incrementally parsed, and context-sensitive editing which will be further explained below. Incomplete parts of the syntax trees are represented by placeholders, shown as question marks in the unparsed text.

Static-semantic information such as name bindings, type information, and static-semantic errors, is maintained incrementally by a static-semantic analyzer. The incremental analysis is performed after each single edit step. The most recent static-semantic information is thus always available to be used in editing and browsing. Normally, the incremental analysis is immediate and not noticed by the user, even for changes to declarations. Static-semantic errors are presented by unobtrusive markings on the unparsed text for the erroneous constructs. The markings disappear automatically when the user corrects the error. Explanations of errors are available via a menu command.



```
RETURN nothing

LOCAL root : ref(Tree) variable !The tree itself;
LOCAL factor : integer variable !Scale down factor;
```

```
W.MoveTo( 10 , 10 );
factor:=2;
root:-new Tree( 0 , 9 , 2 );
new Branch( 13 , 7 , root , factor ).into( root );
new Branch( -13 , 5 , root , factor ).into( root );
new Leaf( -2 , 6 ).into( root );
root.Draw( 8 , 250 , 10 )
```

Figure 2.1 A source program in Orm

2.2.1.1 Example program in Orm

Figure 2.1 shows an example program in Orm. The program draws fractal trees. Four classes appear in the program:

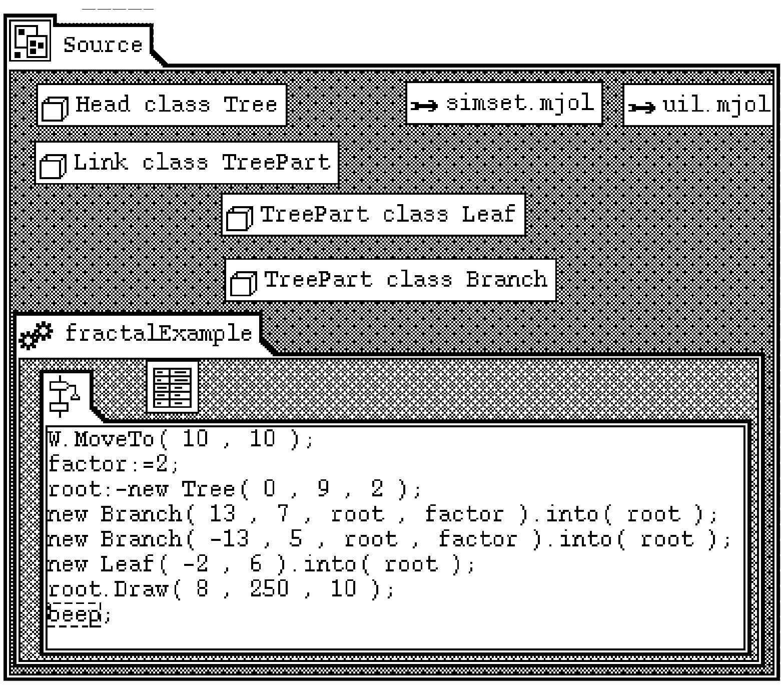
- class `Tree` which models a fractal tree
- class `TreePart` which models a part of a fractal tree (an abstract class)
- classes `Branch` and `Leaf`, both subclasses of `TreePart`.

The program imports two modules: `simset.mjol` which contains the standard linked list facility of Simula (classes `Head` and `Link`), and `uil.mjol` which is a graphical drawing package. The program furthermore contains a procedure `fractalExample` which constructs a fractal tree out of `Branch` and `Leaf` components and tells it to draw itself. (The fractal tree draws itself recursively, smaller for each recursion, until a certain threshold.) The class windows in Figure 2.1 are all closed, but opening any of them would reveal inner structure such as procedures, local variables, and class statement body.

2.2.1.2 Static-semantic errors

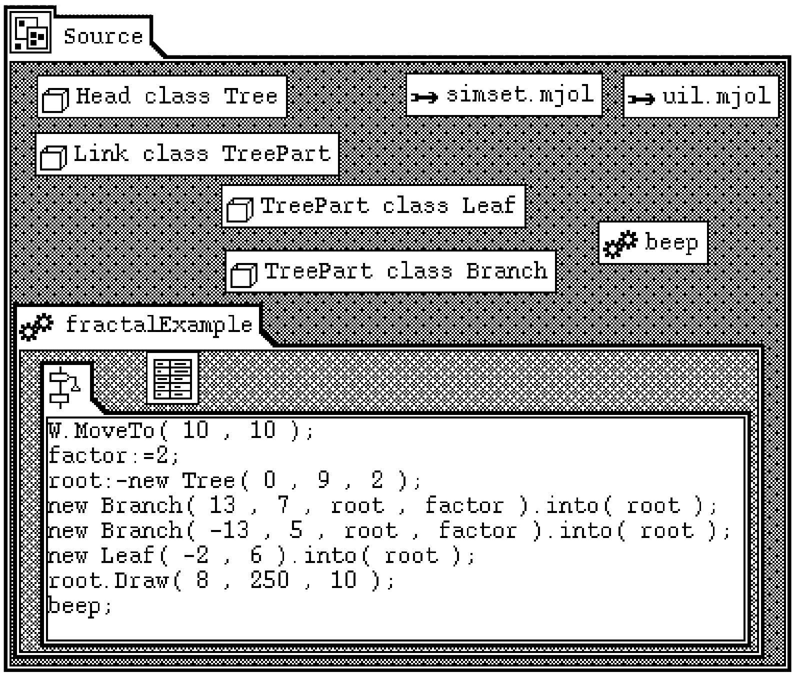
Figure 2.2 shows an example of a static-semantic error. A call to a procedure `beep` has been added at the bottom of the procedure `fractalExample`. However, the procedure has not yet been declared, so the error is marked on the screen by a dashed rectangle surrounding the call.

When a new procedure `beep` is added to the program, as shown in Figure 2.3, the mark on the call disappears immediately. The new procedure is added at the global program level and is thus visible throughout the whole program, including the four classes. Nevertheless, the response time for the incremental analysis is not noticeable by the user. This is because the work done by the incremental analyzer is related only to the actual number of uses of the new procedure, and is independent of the size of the whole program. One of the major goals of the work reported in this thesis is to find techniques which make such immediate response possible, regardless of the size of the program.



```
W.MoveTo( 10 , 10 );
factor:=2;
root:-new Tree( 0 , 9 , 2 );
new Branch( 13 , 7 , root , factor ).into( root );
new Branch( -13 , 5 , root , factor ).into( root );
new Leaf( -2 , 6 ).into( root );
root.Draw( 8 , 250 , 10 );
beep;
```

Figure 2.2 A static-semantic error

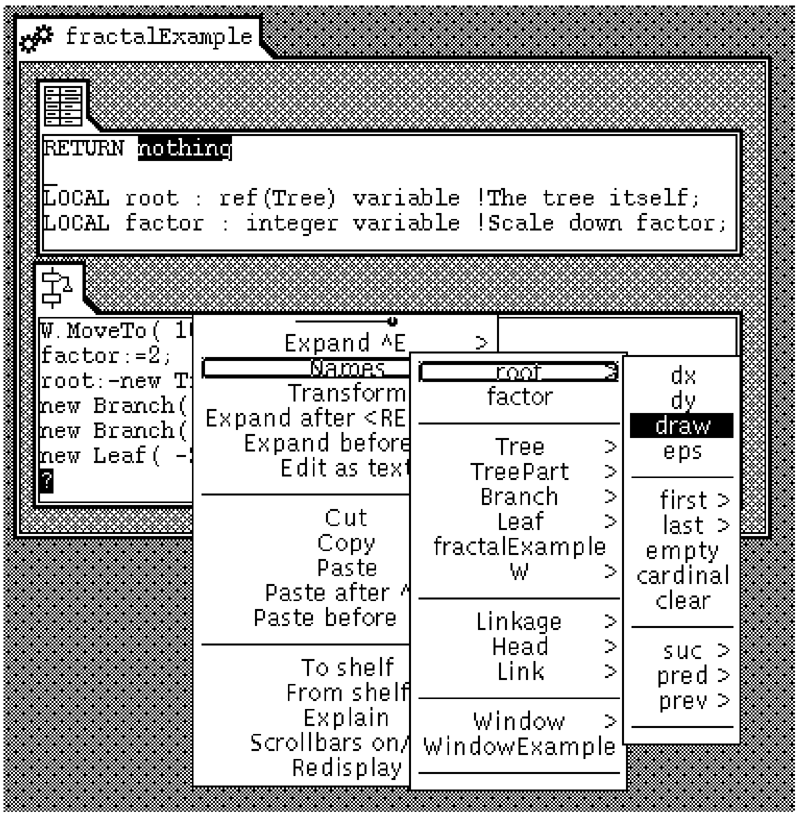


```
W.MoveTo( 10 , 10 );
factor:=2;
root:-new Tree( 0 , 9 , 2 );
new Branch( 13 , 7 , root , factor ).into( root );
new Branch( -13 , 5 , root , factor ).into( root );
new Leaf( -2 , 6 ).into( root );
root.Draw( 8 , 250 , 10 );
beep;
```

Figure 2.3 Correction of static-semantic error

2.2.1.3 Context-sensitive editing

Context-sensitive editing in Orm is centered around a “Names” menu which gives a list of all declared names visible at the current edit focus [Hed92]. This editing mechanism provides a powerful way of constructing dot-expressions, procedure calls, etc. It also works as a simple browsing mechanism. Figure 2.4 shows an example of context-sensitive editing in Orm.



```
RETURN nothing

LOCAL root : ref(Tree) variable !The tree itself;
LOCAL factor : integer variable !Scale down factor;
```

```
W.MoveTo( 1
factor:=2;
root:-new T
new Branch(
new Branch(
new Leaf( -
?
```

Figure 2.4 Context-sensitive editing in Orm

The edit focus is located at the last statement in the procedure (the question-mark placeholder). The edit menu (`Expand/Names/Transform/...`) is a hierarchical menu. Pulling right at “`Names`” gives a sub menu with a list of all declared names visible at the edit focus. Selecting any of these names (`root/factor/...`) replaces the edit focus by the selected name. In the figure, we have instead pulled further right at `root`, bringing up yet another sub menu (`dx/dy/draw/...`). This menu shows the accessible variables and procedures of the root object. I.e., since `root` is declared as a reference qualified by class `Tree`, this menu shows all the variables and procedures declared in `Tree`. By selecting `draw` in this sub menu, the placeholder is replaced by the statement

```
root.draw(?,?,?)
```

where the question marks are new placeholders for the actual parameters of the procedure `draw`.

This mechanism for context-sensitive editing is an example of the kind of advanced editing support which is possible to obtain in an integrated incremental system, where static-semantic information is kept up to date after each edit operation.

2.2.2 Program execution

The user can switch freely between editing and execution of the program. Code generation and linking is done automatically and incrementally as needed. A program does not have to be complete or free from static-semantic errors to be executed. Placeholders and erroneous constructs simply give the effect of a breakpoint in the execution.

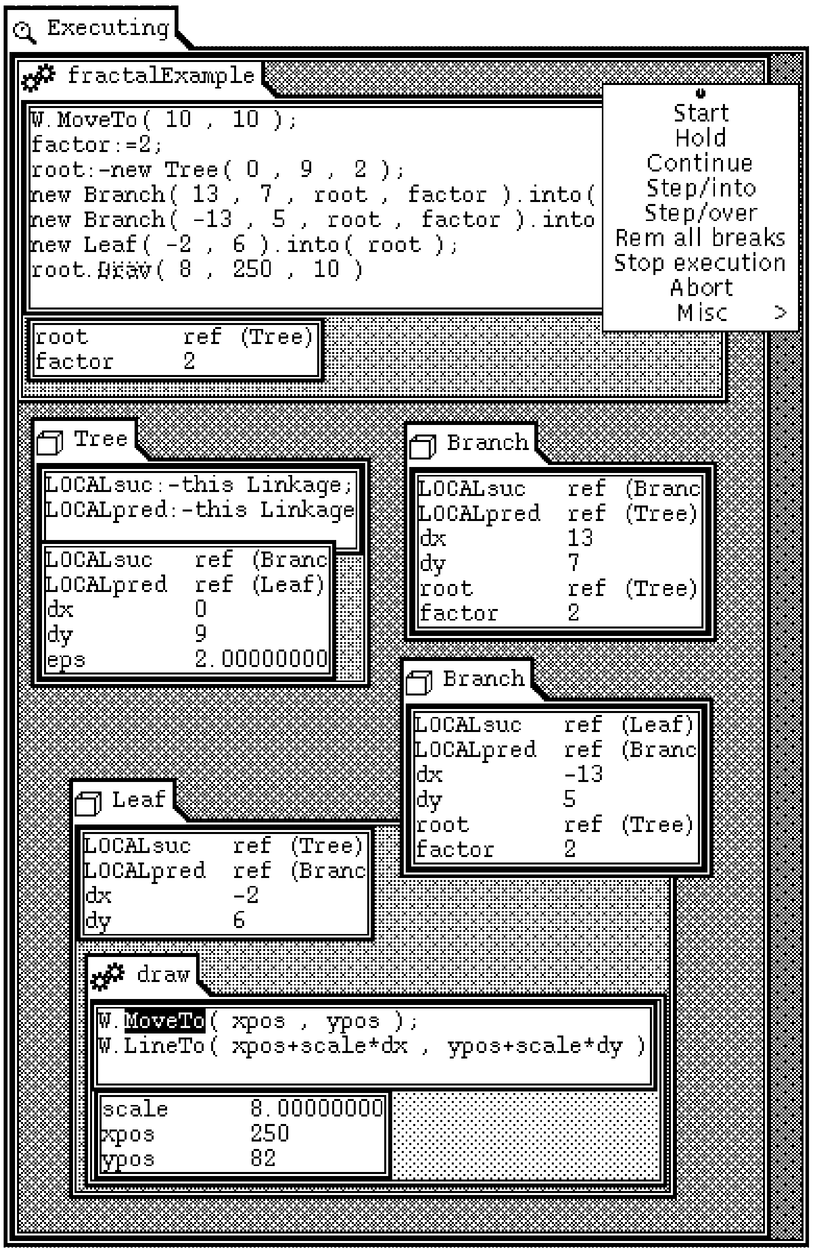
Objects and activation records created during program execution are presented as windows, similar to the class and procedure windows in the source program [THM87]. Also in the execution, the window hierarchy reflects the block nesting structure. For example, the execution of a call "`obj.draw`" is shown as a procedure activation window for `draw` nested inside an object window for the object denoted by `obj`. Local declarations and statements are presented in a similar way as in the source program. In the statement part, the current execution point is highlighted. In the declaration part, the actual values of local variables are shown. For reference variables, the value field shows the actual qualification of the referenced object, and works like a hypertext link button which links to the window of the referenced object.

2.2.2.1 Execution example

Figure 2.5 shows an example execution state of the program of Figure 2.1. The execution window contains an activation of the `fractalExample` procedure and four objects: a `Tree` object, two `Branch` objects and a `Leaf` object. Inside the `Leaf` object is an activation of the procedure `draw` for this object. The current execution point is at the call of `MoveTo`, which is highlighted in the procedure `draw`. Each procedure activation and object has local variables whose current values are shown. Clicking on a reference variable will either briefly highlight the corresponding object window, or bring up a new window (if the corresponding object window was not already on the screen).

2.2.2.2 Continued execution after changes to program

It is a goal of Orm to allow continued execution after program changes. Such functionality can be very useful when developing programs in an exploratory fashion. Another setting where this can be useful, or even necessary, is the maintenance of persistently executing programs. Changes to code which is not active are straight-forward to handle. But, in the general case, changes to executing programs give rise to consistency and version issues, and also to synchronization issues in case of on-the-fly updates to a running program. Some environments like INTERLISP, Gandalf, and DICE support continued execution after changes by transforming procedure acti-



```
W.MoveTo( 10 , 10 );
factor:=2;
root:-new Tree( 0 , 9 , 2 );
new Branch( 13 , 7 , root , factor ).into(
new Branch( -13 , 5 , root , factor ).into
new Leaf( -2 , 6 ).into( root );
root.Draw( 8 , 250 , 10 )
```

```
root        ref (Tree)
factor      2
```

```
LOCALsuc:-this Linkage;
LOCALpred:-this Linkage
```

```
LOCALsuc    ref (Branc
LOCALpred   ref (Leaf)
dx          0
dy          9
eps         2.00000000
```

```
LOCALsuc    ref (Branc
LOCALpred   ref (Tree)
dx          13
dy          7
root        ref (Tree)
factor      2
```

```
LOCALsuc    ref (Leaf)
LOCALpred   ref (Branc
dx          -13
dy          5
root        ref (Tree)
factor      2
```

```
LOCALsuc    ref (Tree)
LOCALpred   ref (Branc
dx          -2
dy          6
```

```
W.MoveTo( xpos , ypos );
W.LineTo( xpos+scale*dx , ypos+scale*dy )
```

```
scale       8.00000000
xpos        250
ypos        82
```

Figure 2.5 A program execution state in Orm

vations or popping the activation stack in difficult cases [Fri83]. At present, Orm does not yet support continued execution, but requires the execution to be restarted after any change to the program source. However, an earlier prototype of Orm did support continued execution. The default scheme used was to let existing instances continue to execute the old code, but let new instances execute the new code version [HM86], [HM87]. Transformations between the old and the new version were made on request.

2.3 Meta level

The Orm programming environment is driven by a number of grammars, controlling the abstract syntax, concrete syntax, static semantics, and code generation. These grammars are represented in the same way as programs, i.e., as abstract syntax trees, and can be edited within the environment itself. The compilation tools: the structure-oriented editor, the static-semantic analyzer, and the code generator all *interpret* grammars (or slightly preprocessed representations for the two latter tools), which allows changes to the grammars to be tried out more or less directly on a sample program. In this way, Orm supports interactive development of language-based environments.

2.3.1 Abstract and concrete syntax

The abstract syntax for a language is described by a BNF style grammar. The concrete syntax is described by a parallel document, giving the unparsing specification for each production in the abstract syntax.

The structure-oriented editor used in Orm interprets the grammars for abstract and concrete syntax *directly*, in their abstract syntax tree form. The editor uses these grammars to control the syntax of a target program also represented in the same abstract syntax tree form. By cascading a series of editor instances working on grammars and programs it is possible to change the grammars and immediately see the effects in a sample program [Min90]. This way, Orm can be used as a highly interactive laboratory for language development.

This technique has been used not only for the development of grammars for programming languages, like Simula, but also for the development of the grammar formalisms themselves, which are then described by meta-grammars. All these grammars have in fact been developed within the system itself, except for two small meta-grammars needed to bootstrap the system.

2.3.2 Static semantics

The grammar for static semantics is based on attribute grammars, but extended with a kernel of primitives making use of objects and side-effects in a controlled manner [Hed88]. These extensions allow efficient incremental processing to be implemented. This method is a precursor to the Door Attribute Grammars treated in this thesis.

The functionality implemented by the primitives in the kernel are, for example:

- adding a declaration to a symbol table
- changing the type of a declaration
- binding an identifier according to a given declaration environment
- mark a static-semantic error on the screen

To preserve the declarative nature of attribute grammars, the primitives are formulated as invariants rather than as imperative operations. The static-semantic grammar is very similar to an ordinary attribute grammar, but makes use of these primitives instead of the usual approach of building symbol tables and environments as attributes. The primitives themselves make use of objects and side-effects, and cannot be expressed in terms of attribute grammars.

The static-semantic analyzer is implemented as an incremental attribute evaluator running a visit-oriented evaluation algorithm. The visit sequences which drive the evaluation are generated from the grammar by sorting the attributes topologically according to their local dependencies. The visit sequences are represented as object structures which are *interpreted* by the attribute evaluator. However, the kernel primitives are compiled and executed directly. This gives a suitable mix of flexibility and efficiency. The grammar can easily be changed and tried out on programs without having to recompile and link the Orm system. Nevertheless, the incremental evaluation is sufficiently efficient for practical use because the "inner loops" of the incremental processing are performed in the compiled kernel.

2.3.3 Code generation

Code generation in Orm is done incrementally with the granularity of a block. Each block results in a template (a structural description of the block) and a code object (a sequence of instructions). Since there is no need for feedback to the user, the templates and code objects do not have to be kept up-to-date with the source program at all times. The blocks needing new template and/or code are only marked by a flag during the incremental static-semantic analysis. When the user gives the command to execute the program, information about which blocks are changed is downloaded to the runtime system. New code for the changed blocks is generated and downloaded incrementally on demand from the runtime system as the program is executed. In principle, this process could be speeded up by doing code generation and loading in advance in a background process, similar to what was done in the Magpie system [DMS84]. This is, however, currently not implemented in Orm.

The code generation grammar is an attribute grammar defining the code for a block as a sequence of three-address instructions. The front-end of the code generator is an exhaustive attribute evaluator which evaluates the code attributes for a block at a time. The resulting sequence of instructions is then translated to native code by a code generator back-end. The code generation grammar makes use of attributes defined in the static-semantic grammar, e.g. identifier bindings, types, and error information, and is a comparatively straight-forward mapping from syntax to instructions.

Similarly to the static-semantic analyzer, the code generator front-end *interprets* a preprocessed form of the code generation grammar. It is therefore possible to modify the code generator grammar interactively and try it out on a sample program. The code generator back-end is, on the other hand, a program created using a traditional generative approach. Generating code for a new target machine thus requires re-generation and re-compilation of the back-end.

2.4 Requirements on incremental techniques

The incremental static-semantic analyzer plays a central role in an environment like Orm. It derives static-semantic information from the source program, and maintains this information as the program is edited. The static-semantic information is essential for many other components of the environment. For example:

- Error messages are communicated to the editor to be displayed in the unparsed presentation of the program.
- Symbol table information is made available to the context-sensitive editor to produce menus of visible names.
- Bindings between identifiers and declarations can be made available to facilities for browsing and “masterscope” [TM81] in the environment.
- Symbol tables, bindings, type, and error information is used by the code generator.

Ideally, the time for updating the static-semantic information after each edit step should be so low that the user does not notice any delay. A simple approach to this problem would be to use a fast machine and compute all the static-semantic information from scratch after each edit step. However, this exhaustive approach does not scale up. Regardless of how fast the machine is, it will always be possible to create a larger program for which the exhaustive strategy does not give the proper response time. A more general solution to the problem is to use incremental techniques and recompute only affected parts of the static-semantic information. To scale up, the response time for an incremental technique must depend only on the size of the affected parts, and be independent on the program size (or at least grow very slowly with the program size).

For any incremental analysis technique to be successful, it is necessary that a small change to the program gives a correspondingly small change in the derived static-semantic information. To obtain this property, the actual structure of the information is of paramount importance. It is necessary to find information structures which are suitable for incremental updating. Furthermore, the information should be structured so that the affected parts for a given change correspond in size to what the user intuitively finds reasonable.

The hardware platform for interactive applications is personal workstations where the processing power is completely at the disposal of the user. In contrast to the old mainframes, the cost for processing power does not depend on how much of the processing power is actually used. This implies that if an incremental updating technique is fast enough to give a response time which is not noticeable by the user, it is not meaningful to optimize the technique further. Since an incremental computation usually operates on a small amount of data, it is often acceptable to use algorithms with worse performance than would be acceptable for batch tools. Flexibility, simplicity of the algorithm, and low space consumption may be more important than low time consumption. For example, it may be acceptable to use linear searching to do symbol table look-up, whereas faster algorithms would be preferable for batch compilers.

Memory consumption, on the other hand, is useful to try to keep small. It lies in the nature of interactive applications that they consume large amounts of memory in order to keep information readily available to the user. There is a tendency that no matter how much memory is added to a workstation, the user will make good use of it, by running more applications on larger data. As long as reasonable response times can be met by the system it is therefore better to consume time than memory. In addition, maintaining more information does not necessarily decrease response time since it takes time to update the added information as well.

To sum up, the desired properties of incremental techniques are the following:

- Low, preferably unnoticeable, response times for common editing operations, independent of program size.
- The response time should stand in proportion to the amount of information the user perceives as affected.
- Space consumption should be kept low.

The key to obtaining a system with these properties is to find structures for static-semantic information which are suitable for incremental updates and for which changes match the changes perceived by a user. The first issue addressed by this thesis is therefore to analyze what problems occur in incremental static-semantic analysis for object-oriented languages. We then propose a technique for describing and updating such structures. Although the main motivation for the work reported in this thesis is to handle the specific problems appearing for object-oriented languages, the proposed technique to solve these problems: Door Attribute Grammars, is not limited to this application area. It is a general technique for describing and updating information derived from syntax trees.

Chapter 3
Incremental Static-Semantic Analysis

This chapter discusses the static-semantic analysis problems appearing for object-oriented languages, and techniques for solving these problems incrementally. Since there are many different views on what object-oriented programming is, we also give some remarks on the views taken in this thesis.

3.1 Introduction

Static program analysis is the process of analyzing a program and deriving context-dependent information from its syntax tree. This is in contrast to *dynamic* analysis which deals with analysis of a program execution. In this thesis, we are particularly concerned with *static-semantic* analysis problems, and not so much with other static analysis problems such as code generation and data flow problems. Static-semantic analysis includes the following subproblems:

- *Name analysis*. Each name application is bound to the corresponding name declaration according to the scope rules of the language.
- *Type analysis*. Each expression is associated with a type.
- *Error detection*. Detection of violations of static-semantic rules ("compile-time errors").

In *incremental* static-semantic analysis, name bindings, type information, and error information is incrementally kept up to date as the user edits the program.

This chapter discusses techniques for incremental static-semantic analysis with particular emphasis on object-oriented programming languages. Static-semantic analysis for these languages is more complex than for the procedural languages treated in standard textbooks on compiler construction. In particular, name analysis for object-oriented languages is substantially more complex, due to the combination of block structure and subclassing. Incremental analysis is correspondingly more complex for object-oriented languages.

The rest of this chapter is organized as follows.

- §3.2 discusses the view on object-oriented programming taken in this thesis.
- §3.3 describes in some depth the name analysis problems occurring in object-oriented languages
- §3.4 reviews and compares methods for incremental name analysis
- §3.5 and §3.6 describe type analysis and error detection problems particular to object-oriented languages
- §3.7 discusses briefly different techniques for incremental updating

3.2 Object-oriented programming

Object-oriented programming began with the Simula-67 programming language [DMN68]. This language was the inspiration source of many newer languages including Smalltalk [GR83], C++ [Str86], and Eiffel [Mey88]. The intense current interest in everything with the label “object-oriented” has diluted the term and made it applicable to almost everything. Perhaps a better term for the original use of object-oriented programming techniques, such as subclassing and virtual procedures, would be “class-oriented programming”.

Even within “class-oriented programming”, there are diverging views on what classes and the related mechanisms of subclassing and virtual procedures are good for. These mechanisms are interesting from many different aspects: modelling, code reuse, data protection, parallelism, etc. By putting different aspects in the center, different object-oriented languages and schools have emerged. In this thesis we follow the school developed along with the Simula and BETA [KMMN87] languages, and which is sometimes referred to as the “Scandiavian school” of object-oriented programming [Coo88]. This school emphasizes modelling: Classes are used for modelling concepts and subclassing for modelling specialized concepts. We will not go into details of the philosophy behind this school, but simply point out a few views relevant for this thesis.

- *Types*. Classes are types and static typing is important because it makes programs more descriptive and easier to understand. Further, by performing static type-checks, many errors in the program are caught early without having to run the program. However, in order to not hamper power of expression, dynamic type-checks are needed in some cases, as discussed in [MMM90].
- *Subclassing*. Subclassing is a concept specialization mechanism, rather than a mere code sharing mechanism. This view is closely connected to that of regarding classes as types and subclasses as subtypes. An object of a subtype should be possible to use wherever an object of its supertype is allowed. Code sharing usually comes out as a nice side-effect of specialization, but other code sharing mechanisms such as aggregation are often better from a modelling point of view.

- *Multiple inheritance*. The value of multiple inheritance is questionable. For object-oriented languages which include multiple inheritance, the motivation has usually been code sharing and not modelling. From a modelling and typing point of view, the full implications of multiple inheritance are unclear.
- *Block structure*. Block structure, or lexical nesting, is an important modularization mechanism as it allows definitions to be localized. In object-oriented programming, nesting is useful in several interesting ways [Mad87]. Few object-oriented languages, other than Simula and BETA, have unlimited block structure allowing, for example, classes to be nested inside classes. However, most object-oriented languages can be seen as having at least a limited block structure where an outer (sometimes implicit) level contains the classes, and each class contains a number of procedures. Some languages have additional block structure, e.g. Smalltalk has a local block concept used inside the procedures (methods) and C++ has a limited form of nested classes.
- *Data protection*. In some views on object-oriented programming, access to a variable in an object from another object is forbidden or considered bad programming style. This is not the view in the Scandinavian school. The view here is that the individual object is seldom the right level for data protection. I.e., it is perfectly fine to access variables in an object directly from another object. Usually, several objects are closely tied together, e.g. in forming a larger aggregate, and “protecting” them from mutual access gives no benefit. The issue of data protection is often more relevant on a larger granularity, such as a module. This view is similar to the one presented in [Szy92].

These views have influenced the way object-oriented techniques are used in this thesis. They have also affected which static-semantic analysis problems are treated in detail. Literature describing the Scandinavian school of object-oriented programming include [Nyg86], [KM88], [MM88], [KMMN91].

3.2.1 Object terminology

We make a sharp distinction between *objects* and *values*. A value is an *immutable* entity whereas objects are *mutable*. Although objects are mutable, each object has a unique immutable *identity*. Object identities are values, but they are different from the values normally used in mathematics in that they denote mutable entities. To distinguish object identities from the numerical values, sets, cartesian products, and other values normally used in mathematics, we refer to the latter as *regular values*.

By having access to an object identity value, the mutable contents of the denoted object can be accessed. An expression or attribute holding an object identity value is called a *reference*.

We will use the class and type system introduced by Simula and the notation used in [MMM90]. Each object is an instance of a class. Classes are arranged in a subclass hierarchy. We use the symbol $\subset$ for the (transitive) relation *subclass of* and the symbol $\supset$ for the (transitive) relation *superclass of*. I.e., we write $\mathsf{B} \subset \mathsf{A}$ to indicate that B is a direct or indirect subclass of A. This notation reflects the extension of classes. I.e., the set of all possible `B` objects is a subset of the set of all possible `A` objects.

Each reference `r` has a *formal qualification*, written `qual(r)` (a class). The formal qualification restricts the set of objects which `r` is allowed to denote. The type system guarantees that in any object structure, the reference `r` will denote either an object of a class `D` such that `D` $\subset$ `qual(r)`, or it will have the special value `NONE`, denoting no real object. For technical purposes, `NONE` is considered to be the identity of an object of class `NOCLASS` which is not in the class hierarchy. The class of the object denoted by `r` is called the *actual qualification* of `r`, and is written `qual(object(r))`. This type system is used in many object-oriented languages including Simula, BETA, C++, and Eiffel. Smalltalk adheres partially to this type system: References have actual but no formal qualifications.

We differ between dynamic and static reference attributes as in BETA. A *dynamic* reference attribute has the value `NONE` at object creation time and can later be changed to denote other objects. A *static* reference attribute of an object `x` denotes another object `y` created automatically as part of creating `x`. A static reference can never be changed to denote another object. `y` is said to be a *part-object* of `x`. We also say that `x` is the *owner* of `y`.

3.3 Name analysis

We now turn back to the main topic of this chapter: the static-semantic analysis problems which occur for object-oriented languages.

In name analysis, each *name application* is associated with the corresponding *name declaration*, and information about the declared name, for example its type, is made available to the name application. The association between a name application and its name declaration is called a *binding*. The process of finding the appropriate binding for a name application, according to the scope rules of the language, is called *lookup*. Lookup can be performed by a function which takes as parameters the name and a *declarative environment* of the name application. The declarative environment contains information about which name declarations are visible at the name application site.

Name analysis is a fundamental problem in static analysis - nearly all other static analysis problems require the name analysis to have been done. Name analysis is further one of the hardest static analysis problems to handle incrementally because of the non-local dependencies introduced by using names. In object-oriented languages, the name analysis problem is more complex than in procedural languages, because of the combination of subclassing and block structured visibility rules.

A general model for describing and implementing name analysis is Garrison's Inheritance Graph Model [Gar87]. We use a similar but simpler model, suitable for languages based on Algol-like block structure. This model assumes that the order of declarations within a block is of no importance. It handles the usual shadowing principle: that a declaration in one block will shadow other declarations with the same name in outer blocks. With *block* we mean here a syntactic construct which introduces a new name space. A block is also a template for run-time units (block instances or *objects*). Examples of blocks are Algol block statements, Simula classes and procedures, and Pascal records. Name analysis applies to all kinds of named entities, e.g. classes, procedures, and variables. Although the dynamic semantics for accesses may differ substantially (e.g. between a procedure call and a variable access), the static name analysis is done in the same way for all entities. The model makes use of a visibility graph defined as follows.

3-1 Definition Visibility graph

A *visibility graph* is a directed acyclic graph $G = (V, E)$. The vertex set V is divided into three disjoint sets T, P, and $\{null\}$ as follows.

- T is a set of *table vertices*. A table vertex represents a local symbol table giving access to the declarations of one block. It has no outgoing edges.
- P is a set of *path vertices*. A path vertex represents a specific combination of tables. It has one or more outgoing edges, $e_1 .. e_n$, $n \geq 1$. The edges are ordered, representing the precedence of combination with respect to shadowing. I.e., a declaration accessible via an edge e_k will shadow declarations of the same name accessible via edges $(e_{k+1} .. e_n)$.
- *Null* is a distinguished vertex with no outgoing edges. It is used as a sentinel.

end 3-1

The *lookup sequence* of a vertex v, denoted by $LS(v) = (t_1 : t_2 : ... : t_m)$, is defined as the sequence of table vertices obtained by doing "in-line substitution" in the following way. A vertex v with outgoing edges $e_1 .. e_n$, $n \geq 1$, ending in vertices $v_1 .. v_n$, is substituted by the sequence $v_1 : v_2 : ... : v_n$. The distinguished vertex *null* is ignored if encountered. This results in a sequence of only table vertices. A table will occur several times in the sequence if it is reachable via several edge paths. Since the graph is acyclic, the lookup sequences are finite.

We will sometimes use an *expanded form* of the lookup sequence which includes the set of declared names for each block. The following notation is used: $(t_1 \{...\} : t_2 \{...\} : ... : t_m \{...\})$. Empty sets are dropped in the notation.

Let $X = (x_1 : ... : x_n)$ be a lookup sequence. The subsequence $(x_1 : ... : x_k)$, $1 \leq k \leq n$, is said to be the x_k*-prefix* sequence of X. Analogously, the subsequence $(x_k : ... : x_n)$, $1 \leq k \leq n$, is said to be the x_k*-suffix* sequence of X.

To implement name analysis, a visibility graph is constructed for the program to be analyzed. The graph contains a table vertex for each block in the program. Path vertices are added to combine the tables in useful ways. Each name application is then associated with one of the path vertices in the graph. This path vertex is said to be the *lookup vertex* of the name application. The lookup vertex represents the declarative environment of the name application, and gives access to the declarations of all names visible at the name application site. Lookup can be implemented as a simple recursive procedure which looks in the tables encountered in a left-to-right traversal of the graph, starting at the lookup vertex. The tables are thus visited in the order of the lookup sequence.

3.3.1 Block Structure

The visibility graph of a block-structured program can be constructed as follows:

3-2 **Construction** Visibility graph for block structure

For each block b in the program a table vertex t_b and a path vertex s_b is added to the graph. The path vertex s_b is called the *static path* and has two outgoing edges. The first edge ends in t_b. The second edge ends in the static path of the enclosing block. For the outermost block, the second edge ends the distinguished vertex *null*.

end 3-2

The static path vertices serve two roles: To be used in construction of the static path vertices of enclosed blocks, and to be used as lookup vertices. An example is shown in Figure 3.1. Table vertices are shown as large boxes. Path vertices are shown as smaller boxes with one slot for each outgoing edge. The slots are ordered; the leftmost slot corresponding to the first outgoing edge. An outgoing edge ending in the *null* vertex is shown as a diagonal line through the slot.

The static path vertex s_b of a block b is used as the lookup vertex of the name applications inside b. For example, the name application x (in the “x:=1” assignment) has the lookup vertex s_{D} which has the expanded lookup sequence (t_{D} : t_{C} : t_{A} {x}). The declaration of x is thus found in t_{A}.

For plain block structure, there is a one-to-one correspondence between table vertices and path vertices. A simpler visibility model could have been used here. However, in the following we will see plenty of examples where there is no such one-to-one correspondence, and where it is necessary to distinguish between tables and paths.

```
begin (A)

  integer x;

  begin (B)
  end;

  begin (C)

    begin (D)
      x := 1;
    end;

    begin (E)
    end;

  end;

end;
```

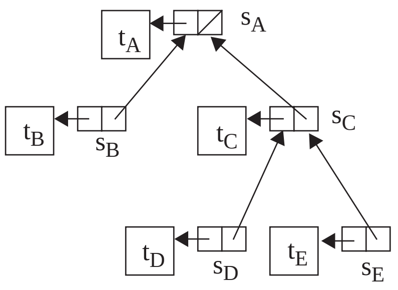


Figure 3.1 Block Structure

3.3.2 Subclassing and block structure

Object-oriented languages have more complex visibility rules because of the combination of subclassing and block structure. Consider the following Simula program and its associated visibility graph.

```
begin (A)

  class B; (B)
  begin
    integer x;
  end;

  B class C; (C)
  begin

    begin (D)
      x := 1;
    end;

    begin (E)
    end;

  end;

end;
```

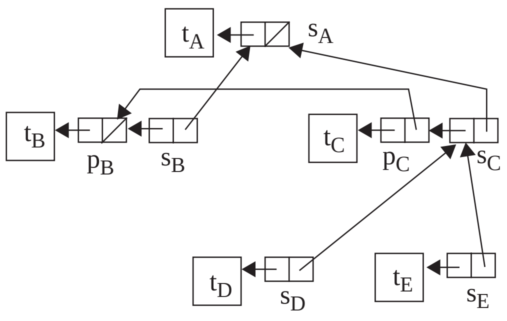


Figure 3.2 Subclassing Combined with Block Structure

The expanded lookup sequence of s_D is in this case ($t_D : t_C : t_B$ {x} : t_A {B, C}) and the declaration of x is found in t_B as appropriate. The graph is constructed as follows.

3-3 Construction Visibility graph for combined subclassing and block structure

There are two kinds of blocks: simple blocks (procedures or Algol block statements) and classes. For simple blocks, the same construction rules are used as in the plain block-structure case (construction 3-2). For a class *c*, one table vertex t_c and two path vertices p_c and s_c are added to the visibility graph. The prefix path p_c has two outgoing edges. The first ends in t_c and the second ends in the prefix path of the superclass. If the class has no superclass the second edge ends in the *null* vertex. The static path s_c also has two outgoing edges. The first ends in p_c and the second in the static path of the enclosing block.

end 3-3

The prefix path vertex here serves the role of being used in the construction of other prefix and static path vertices. In §3.3.3 we will see how it is associated with name applications in remote accesses.

The construction of visibility graphs for subclasses in combination with block structure works also in the case of nested classes. An example Simula program with nested classes is the following one.

```
begin (A)

  class B; (B)
  begin

    integer x;

    class BL; (BL)
    begin
    end;

  end;

  B class C; (C)
  begin

    BL class CL; (CL)
    begin
      x := 1;
    end;

  end;

end;
```

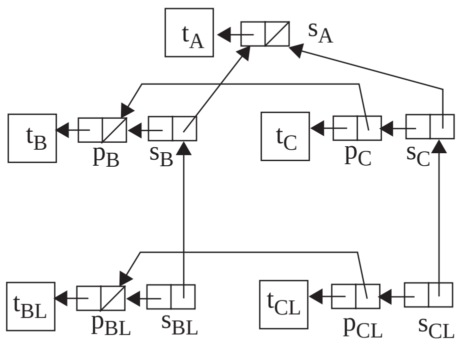


Figure 3.3 Nested Classes

The expanded lookup sequence of $s_{\mathtt{CL}}$ is ($t_{\mathtt{CL}}$: $t_{\mathtt{BL}}$: $t_{\mathtt{C}}$ {`CL`} : $t_{\mathtt{B}}$ {`x`, `BL`} : $t_{\mathtt{A}}$ {`B`, `C`}) and the declaration of `x` is thus found in $t_{\mathtt{B}}$.

A straight-forward construction of the visibility graph for an erroneous program with cyclic subclassing would lead to a cyclic visibility graph. This must be prevented since our definition of visibility graphs (3-1) allows only acyclic graphs. One way to solve this problem is to select one of the classes on the cycle, and treat it as if it had no superclass. This breaks the cycle. The selected class can be associated with a static-semantic error to indicate that its superclass is considered invalid. Which of the classes on the cycle to select can be an error handling policy, as will be discussed in more detail in §3.6. In the example below, class `B` has been selected to break the cycle, and the visibility graph is constructed as if `B` had no superclass. The dashed line shows the edge which would be present if the cycle was not broken. Class `B` is marked with a static-semantic error "*`invalid superclass`*" in the program.

```
begin (A)

  C class B; (B)invalid superclass
  begin
  end;

  B class C; (C)
  begin
  end;

end;
```

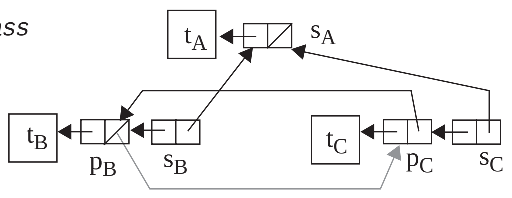


Figure 3.4 Cyclic Subclassing. Breaking the cycle in the visibility graph.

3.3.3 Remote access

Remote access is a visibility construct which gives access to entities in explicitly referred objects. Simula (and also many other languages) use dot-notation for remote access. A remote access has the following syntactic form:

<remote access> ::= <object expression> . <selector>

The *object expression* can itself be a remote access, thus leading to a series of remote accesses. The *selector* is a name application whose declarative environment depends on the type (class) of the object expression. The prefix path vertex of the class describes this declarative environment. Consider the following program:

```
begin (A)

  class B; (B)
  begin
    integer x;
  end;

  B class C; (C)
  begin

    begin (D)
      x := 1;
    end;

    begin (E)
    end;

  end;

  ref (C) rC;
  rC.x := 2;
end;
```

Figure 3.5 Program Including a Remote Access

This is the same program as Figure 3.2, with the addition of a reference variable `rC` and an assignment statement including a remote access. These additions do not affect the visibility graph, which is thus the same as in Figure 3.2. The class of the object expression `rC` is `C`, and the lookup vertex of the selector `x` is consequently the prefix path vertex $p_{\mathtt{C}}$. The expanded lookup sequence of $p_{\mathtt{C}}$ is ($t_{\mathtt{C}}$: $t_{\mathtt{B}}$ {`x`}) and the declaration of `x` is thus found in $t_{\mathtt{B}}$.

3.3.4 The Inspect Statement in SIMULA

The *inspect* statement of Simula (called “connection statement” in the language definition for Simula) “opens” an object so the entities of the object can be accessed directly by name applications inside the inspect statement. In case the inspected object does not contain declarations of the name applications, the declarative environment outside the inspect statement applies. The “with” statement of Pascal works in a similar way. The inspect statement is not typical for object-oriented languages. Nevertheless, it is a useful construct in certain situations. Since it has interesting visibility rules we have included a discussion of it here.

To represent the declarative environment of name applications inside the inspect statement, special *inspect path vertices* need to be added to the visibility graph, as follows:

3-4 **Construction** Inspect path vertex

For each inspect statement, inspecting an object of class *C*, an inspect path vertex is constructed with two outgoing edges. The first edge ends in p_C, and the second in the path vertex applicable outside the inspect statement.

end 3-4

The following program example shows an inspect path vertex *v*.

```
begin (A)

  class B; (B)
  begin
    integer x;
  end;

  B class C; (C)
  begin
  end;

  ref (C) rC;

  begin (D)

    integer y;

    inspect rC do
    begin
      x := y;
    end;

  end;

end;
```

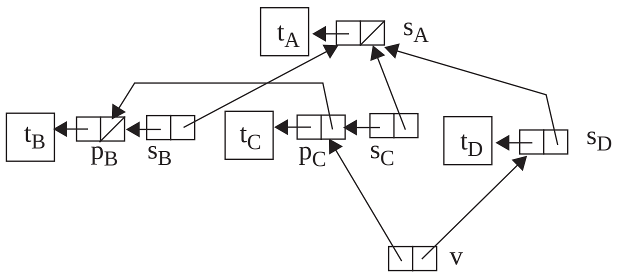


Figure 3.6 An Inspect Path Vertex

The expanded lookup sequence of the inspect path vertex *v* is (t_C : t_B {`x`} : t_D {`y`} : t_A {`B`, `C`, `rC`}). Thus, we see that both `x` and `y` are accessible inside the inspect statement.

A variant on the inspect statement contains “when”-clauses to allow an object to be inspected as an object of a specific class. This allows the inspect statement to be used as a case-statement, dispatching on the type of the object. For this variant, an inspect path vertex is needed for each “when”-clause.

3.3.5 Virtual Classes in BETA

The *virtual* concept in BETA is far more general than in most other object-oriented languages. In particular, it applies not only to procedures, but also to classes [MM89]. If the type (class) of a reference attribute is a virtual class, the actual type of the attribute depends on where the attribute is used. The special type mechanism “like current” in Eiffel works similarly. An important use of BETA’s virtual classes is in general classes such as set, vector, and list, where they can be used to achieve an effect similar to type parameters.

The example below illustrates a simple use of virtual classes in BETA. The virtual class `V` is declared inside class `A` and is extended inside `A`’s subclass `B`. The reference `rV` denotes a “part-object”, i.e. an object automatically generated at the same time as the enclosing `A` object. Inside class `A`, all we know is that `rV` denotes (at least) a `V` object. Thus, we know that `rV` has an `x` attribute, but we cannot know if it also has other attributes, e.g. `y`. Inside class `B`, we know that `rV` actually denotes an object of (at least) the extended `V` definition, which is seen as an anonymous subclass of `V`. Here, we know that `rV` has both an `x` and a `y` attribute.

```
(#
   A: class
   (#  V: virtual class (# x: @integer #);
       rV: @V;
   do
       1 -> rV.x;
       INNER;
   #);

   B: class A
   (#  V: extended class (# y: @integer #);
   do
       rV.x -> rV.y;
   #);
#)
```

Figure 3.7 Reference to object of virtual class in BETA

The type of an `rV` application depends on where the application occurs. Inside `A`, the type is `V`. Inside `B`, the type is the extended `V`. We differ between the *declared* type of `rV`, which is `V`, and the *actual* type of an `rV` application, which depends on where the application occurs. The declaration of `rV` can be found using ordinary lookup. The actual type can then be found by a function ACTUAL(*ApplBlk*, *Decl*), where *ApplBlk* is the block containing the application, and *Decl* is the declaration of `rV`. This function traverses the block, class, and declaration structure in order to find the actual type. We do not include the details of this function as it is rather complex. The important thing to note here is that the addition of virtual classes and the use of them as virtual types does not affect the way the visibility graph is constructed. It only affects the way actual types of name applications are located.

The situation becomes more complicated if virtual classes are used as superclasses. Consider the example below. In this case, the virtual class `V` has a subclass `W`. Inside class `A`, `W` is known to have a superclass which is at least `V`. The object denoted by `rW` is thus known to have at least the `x` and the `y` attributes. In class `B`, the virtual class `V` is extended. Thus, inside `B`, `W` is known to have a superclass which is at least the extended `V`, and `rW` is known to have at least the `x`, `y`, and `z` attributes. The class `W` is virtual in the sense that its superclass is not fixed. Whenever an extension to `V` is introduced, this means an implicit introduction of an actual definition of `W`. This actual definition, `W'`, is a new anonymous class with its own prefix vertex $p_{\mathtt{W'}}$, with edges ending in ($t_{\mathtt{W}}$,

$p_{V'}$) , as shown in Figure 3.9. The selectors in the `rW` remote accesses in `B` are associated with the $p_{W'}$ vertex. Since `W'` is not an explicit class it has no enclosed entities. There is therefore no need for any table vertex or static path vertex for `W'`.

```
(#  (M)
    A: class
    (#  V: virtual class (#x: @integer; do 1 -> x; #);
        W: class V (#y: @integer #);
        rW: @W;
    do  rW.x -> rW.y;
        INNER;
    #);

    B: class A
    (#  V: extended class (#z: @integer #); (V')
    do  rW.x -> rW.y -> rW.z;
    #);
#)
```

Figure 3.8 Virtual superclass in BETA

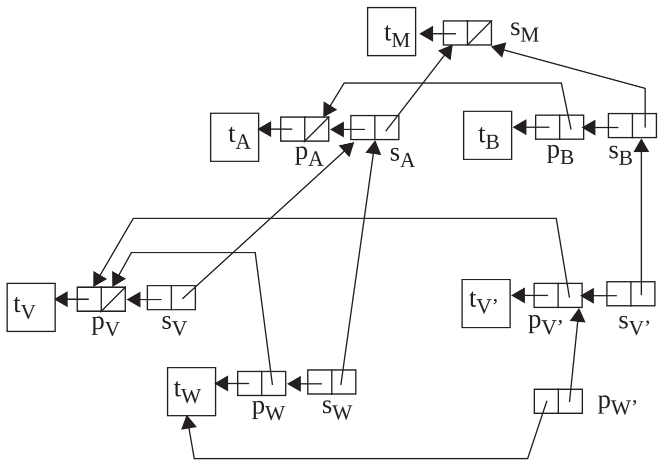


Figure 3.9 Visibility graph for virtual superclass

To summarize, the presence of a virtual class affects the visibility graph if the virtual class is used as a superclass. If the virtual class is only used as a type on references, the visibility graph construction of 3-3 is sufficient.

3.3.6 Cyclic Dependencies

Types, bindings, symbol tables, and visibility graphs depend on each other. In several situations the dependencies are inherently cyclic. We will here mention some of these situations in an intuitive way.

- *Arbitrary declaration/application order.* This means that within a block, the declarations contribute to the symbol table of the block, but they can also utilize the information in the symbol table, e.g. if their type makes use of a name application which could be declared in the block. This leads to a cyclic dependency between the symbol table and the declaration, as in the figure below.

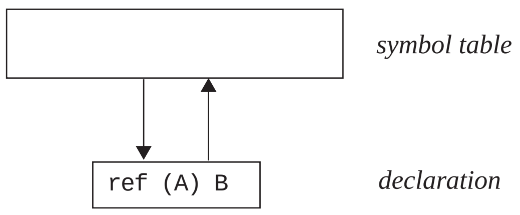


Figure 3.10 Cyclic dependencies between symbol table and declaration.

- *Reference variables*. In a reference variable declaration "`ref (x) y`", `x` must be the name of a class. Clearly, the meaning of `y` depends on what "`ref (x)`" means, and the name application `x` in this construct depends on the declaration of `x`. The following example is correct according to the context-free syntax, but constitutes a static-semantic error since `a` is a reference variable and not a class. In this case, the dependencies lead to a cycle:

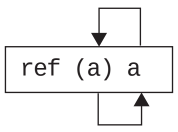


Figure 3.11 Cyclic dependencies within a reference variable.

- Cyclic subclassing. As mentioned earlier, cyclic subclassing must be especially taken care of to avoid cycles in the visibility graph. However, even if the visibility graph is constructed to not contain any cycle, there is still a cyclic dependency between the classes since changing the superclass of either of them will affect which identifiers are visible in the other one:

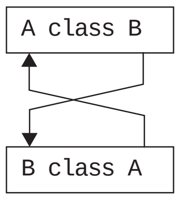


Figure 3.12 Cyclic dependencies between classes.

3.3.7 Other features and variations

As noted by Garrison [Gar87], there are about as many subtle variations on visibility rules as there are languages. Our goal here has not been to describe them all, but to describe some basic typical rules for block structured and object-oriented visibility mechanisms. Examples of other mechanisms which influence visibility are overloading, multiple inheritance, visibility restrictions, and significance of declaration-application order.

- *Overloading*. In overloading, it is not only the name which influences declaration lookup, but also the operand types. The lookup function would need to be modified to take this into account. However, the same visibility graphs as described above can be used.
- *Multiple inheritance*. The visibility graphs above can be extended in a straight-forward way to handle multiple inheritance. Instead of having only one edge ending in the prefix path vertex of the superclass, a path vertex of a class needs a set of outgoing edges, one for each superclass. The ordering of edges would resolve name clashes in an ad-hoc left-to-right manner. The workings of the lookup function would have to be changed if other name clash resolution schemes were desired.
- *Visibility restrictions*. Many languages have visibility restriction mechanisms. These restrictions can either be defined by the language, or can be possible to express explicitly in the program. E.g., in Simula, attributes can be explicitly declared as “hidden” or “protected” which restricts access from subclasses or from outside the class, respectively. Visibility restrictions can be seen as “filters” which apply during lookup.
- *Declaration-application order*. In some languages, the order of declarations is significant, and a declaration must precede all its applications in the program text. Such restrictions are mainly motivated by implementation reasons, with the goal of being able to write one-pass batch compilers. From a language design point of view, such restrictions make little sense. As noted by Garrison [Gar87], restricting declaration order leads to very complex visibility graphs. In incremental compilers, restricting the declaration order does not simplify the

implementation. On the contrary, the restrictions add new dependencies which have a negative effect on the possible performance of the incremental compiler. Thus, we see only negative effects of restricting declaration order, and do not treat such languages further in this thesis.

3.4 Incremental name analysis

In incremental name analysis, the bindings between applied and declared names, and the information transmitted via bindings, are kept up to date as the program is changed. Adding or removing a name application affects only the binding for that particular name application. Changes to declarations, on the other hand, can have widespread consequences, affecting many name applications. We distinguish between the following kinds of changes to declarations:

I *Add a declaration*.

II *Remove a declaration*.

III *Change a declaration*. With this we mean simple changes which do not affect the bindings or the visibility graph. A typical example is changing the type of a variable declaration.

IV *Change the visibility graph*. Some changes to declarations can affect the visibility graph. A typical example is replacing the superclass of a class declaration.

For each of these changes there is a set A of directly affected name applications which either have to be rebound, or for which the transmitted declaration information has changed. Changes II and III are comparatively easy to handle, as they affect only the name applications bound to the changed declaration. If there is some suitable data structure for following bindings from name declarations to name applications, the affected sites are easily located. Adding a declaration (I), is considerably more complicated. Here, the affected sites can either be name applications for which there was earlier no declaration, or name applications bound to declarations which become shadowed by the new declaration.

Changes which affect the visibility graph (IV), are also more difficult to handle. Suppose an outgoing edge of a path vertex v has been replaced. Let R be the set of name applications which have a lookup vertex from which v is reachable. The affected name applications A is a subset of R. For example, consider changing the superclass of a class declaration c in a language with block structure, subclassing, and remote access. The change affects the visibility graph by replacing the second edge in the prefix path p_c. Here, R consists of the following name applications:

- Name applications lexically inside class c or its subclasses.
- Selectors in remote accesses where the type of the object-expression is c or a subclass to c.

We will now review a number of methods which are used in different systems for handling the above changes and comment on their time complexity and applicability. The methods will be characterized in terms of the cardinals of the following sets:

A	The set of directly affected name applications which either have to be rebound, or for which the information transmitted across its binding has changed.
R	The set of name applications which have a lookup vertex from which changed declarations or changed edges in the visibility graph are reachable.
T	The set of nodes in the whole syntax tree.

These sets are related as $A \subseteq R \subseteq T$. The lower bound for any method is $O(|A|)$. Some methods take as much time as $O(|T|)$. Clearly, such methods do not scale up. Most notably, the methods resulting from standard attribute grammars are $O(|T|)$. Methods which are better than $O(|T|)$ usually require extra bookkeeping data and operations. Typically, such bookkeeping is performed for each name application and the extra time is thus spread out during the incremental analysis. However, space is usually a scarce resource in incremental systems, and since name applications are very common in programs - perhaps up to half of the nodes in a syntax tree are name applications - the extra bookkeeping space should be minimized. What is acceptable in practice depends on many factors: the frequency of different changes, the usual structure and size of programs, the actual implementation and hardware, how large space consumption is acceptable, how long maximum delay for any change is acceptable, etc. One should also note that applying one method for one kind of program change will often affect the performance at other program changes. Compromises are thus necessary.

1. *Give Up*. This method does incremental analysis only for changes in statements and expressions, but recompiles the whole program if one of the changes I-IV occurs. Clearly, the method is $O(|T|)$ and it does not scale up. The Cornell Program Synthesizer [TR81] used this approach.

2. *Search Smallest Subtree*. The idea here is to find the smallest subtree outside which there can be no affected name applications. Each name application within this area is then checked, typically by redoing the lookup and comparing the result with the previous binding. In a plain block structured language this means that the search can be restricted to the block subtree containing the changed declaration, and the time complexity is in this case $O(|R|)$. However, in the more general case, e.g. for object-oriented languages, the visibility graph may have edges which do not follow the structure of the syntax tree. In such cases, the method may degenerate to be $O(|T|)$ for most changes of type I-IV. For example, since a declaration inside a class can be accessed by name applications syntactically outside the class (by remote access or subclassing), this in effect leads to that the smallest subtree containing possible name applications coincides with the whole syntax tree. Thus, in general, the method is $O(|T|)$ and does not scale up. Optimal incremental evaluation for standard attribute grammars [Rep82], [Yeh83] behaves this way in practice, although the attribute evaluation method is implicit and not formulated as an explicit search.

3. *Follow reverse edges*. The idea with this method is to follow the visibility graph edges in the reverse direction to find syntax subtrees which may contain affected name applications. This method is similar to the previous one, but achieves $O(|R|)$ complexity without requiring the visibility graph edges to follow the syntax tree structure.

4. *Maintain Cross-References*. By maintaining a set of references at each declaration to all its name applications, changes II and III can be handled in $O(|A|)$. The bookkeeping space amounts to one cross-reference for each name application.

5. *Search Environment*. Some simple languages have the property that a declaration d in a block b can only shadow declarations visible via the static path of b. We call this the *shadows-visible* property. Languages with block structure, subclassing and non-nested classes have this property. For such languages, the following ad-hoc method can be used to handle addition of declarations (I).

   > Consider a new declaration d in a block b. Search the declarations in $LS(s_b)$. For example, in the case of pure block structure, search the declarations in the enclosing blocks. If a declaration d' with the same name as d is found, locate the name applications bound to d', e.g. using cross-reference information. Of these name applications, the ones which have t_b in their lookup sequence are affected.

   This method, although formulated only in the special case of pure block structure, was used by Johnson and Fischer in [JF82]. However, this method does not work for languages with more complex visibility constructs, e.g. languages with nested classes and inspect statements. To see this, consider the following lookup sequences of the program with nested classes in Figure 3.3.

   $LS(s_{BL}) = (t_{BL} : t_B : t_A)$

   $LS(s_{CL}) = (t_{CL} : t_{BL} : t_C : t_B : t_A)$

   We see that the table t_C occurs after t_{BL} in the latter sequence, but not in the former. Thus, a declaration in the block BL may shadow a declaration in C although the declarations in C are not visible from the declarative environment s_{BL}. The program is thus not shadows-visible. If the Search Environment method is applied it will not find affected name applications bound via the s_{CL} vertex. By a similar example, comparing the lookup sequences of s_D and v in Figure 3.6, it is seen that inspect statements may also result in programs which are not shadows-visible and where the method thus fails.

   Another drawback with the Search Environment method is that it may locate more name applications than necessary. In many practical situations, the method is, however, close to $O(|A|)$. The same bookkeeping information can be used as in the Cross-References method.

6. *Maintain Traces*. A general method for handling additions of declarations (III) is to make use of “traces”. The idea is to let each name application leave traces in the symbol tables traversed during lookup. A trace includes information both about the name of the searched identifier and about the location of the name application which has tried to bind to that symbol table. When adding a new declaration to a symbol table, the trace information is inspected to find name applications which have previously tried to bind to that name in that symbol table, but failed to find a matching declaration.

   The cost of adding a new declaration using this method is $O(|A|)$ if the trace information contains direct information about the location of the affected name applications. There is also a bookkeeping cost when binding a name application a. This cost is proportional to the number of symbol tables l traversed during lookup before finding a matching declaration. The book-keeping cost in time is thus proportional to the time for lookup and should not be any problem in an incremental system.

However, the overhead in space may be more serious. If half of the number of syntax nodes in the tree are name applications this leads to a space overhead of $O(l_{ave}(|T| / 2))$ where l_{ave} is the average value for l. Depending on the value of l_{ave} and the actual space cost for each trace, this could be a problem in practice. In the worst case l is equal to the length of the look-up sequence for a. This happens if the matching declaration is a global declaration or if a is undeclared. For practical purposes, the maximum length of a lookup sequence could be estimated to around 10, corresponding to 3 static block levels and 7 subclass levels.

If trace information is added also to the symbol tables where matching declarations are found, these traces are equivalent to cross-references and the method can thus be extended to handle also removals and changes to declarations (II, III).

The method can also handle changes to the visibility graph (IV) if traces are maintained also in connection to the visibility graph *edges*. Suppose the endpoint of an edge is changed from a vertex v to another vertex w. The affected name applications can then be found by inspecting all the trace information associated with the changed edge. In principle, the set of name applications B found this way may be a superset of A since some of the name applications bound via the changed edge may end up being bound to the same name declaration also after the change. For example, if the superclass of a class declaration is replaced, name applications within the class which are bound to global declarations will leave traces at the class-superclass edge, but are not necessarily affected by a replacement of this edge. In practice, it is reasonable to assume that local accesses are much more common than global accesses and that $|A \cap B|$ is much smaller than $|A|$. This method is then $O(|A|)$ also for changes of type IV.

The Mjølner/Orm system performs incremental name analysis based on this method. A slightly simpler variant was used in a precursory system for simpler block-structured languages [MM85], [Min85]. Several other systems also use variants of this method. Hoover describes a technique for handling aggregate values [Hoo87], suitable for incremental name analysis for block-structured languages, where *key trees* serve the role of traces. Vorthmann developed a system for incremental name analysis for general graph based visibility rules [VL88], [Vor90a]. This system uses a technique very similar to the one used in Orm, representing graph edge endpoints by *views*, which maintain the trace information. Vorthmann uses the term *bread crumbs* for the trace technique.

The following table summarizes the worst-case time complexity and applicability of the above methods.

	I — Add declaration	II — Remove declaration	III— Change declaration	IV— Change vis. graph
1. Give up	$O(\|T\|)$	$O(\|T\|)$	$O(\|T\|)$	$O(\|T\|)$
2. Subtree*	$O(\|T\|)$	$O(\|T\|)$	$O(\|T\|)$	$O(\|T\|)$
3. Rev. edges	$O(\|R\|)$	$O(\|R\|)$	$O(\|R\|)$	$O(\|R\|)$
4. Cross-refs.	—	$O(\|A\|)$	$O(\|A\|)$	—
5. Search env.**	$O(\|A\|)$	—	—	—
6. Traces***	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$	$O(\|A\|)$

*Effect of using standard AGs for incremental name analysis
**Only applicable to languages with the "shadows-visibile" property
***Method used in Mjølner/Orm.

Figure 3.13 Complexity of incremental name analysis methods

3.4.1 Compromises between time and space

As mentioned above, the "Maintain traces" method has a space overhead of $O(l_{ave}(|T|/2))$ where l_{ave} is the average number of symbol tables traversed during lookup. In practice, it can be motivated to reduce this overhead and store less information at the expense of a somewhat longer time for finding the affected name applications.

Compromise used in Orm

In the Orm system, the traces include information only for quickly identifying which blocks actually contain affected name applications. These blocks are then searched linearly. This reduces the space needed for trace information but still gives a reasonable response time since most blocks are rather small. This approach is thus a compromise between searching and maintaining information.

A hybrid method

If space costs must be kept very low, another alternative could be to use cross-references (method 4) for deletions and changes of declarations (II, III), reverse edges (method 3) for changes to the visibility graph (IV), and a variant of maintain traces (method 6) for additions of declarations (I). The latter method would keep only a reference count in the symbol tables as trace information, counting the number of name applications which tried but failed to bind to a given name in a given symbol table. This combination of methods would have the advantage of reducing the space overhead for trace information substantially. For addition of declarations, it would still be able to handle the very common case of $|A|= 0$ in constant time ($O(|A|)$). The disadvantages would be that in the $|A|> 0$ case and for changes to the visibility graph, method 3 would have to be used. However, since these changes occur comparatively infrequently, the disadvantages of the $O(|R|)$ response of method 3 might be outweighed by the advantages of a lower space consumption.

Special handling of accesses to global standard names

An interesting possibility is to consider handling global accesses differently from non-global accesses. If the program is dominated by local accesses, l_{ave} is approximately 1 and the space overhead $O(|T| / 2)$. On the other hand, if the program is dominated by global accesses and the maximum length of lookup sequences is, say, 10, the space overhead could be as much as $O(5|T|)$. If the space for one trace is the same as the space for a syntax node, this would mean that the traces would take up 5 times the space of the syntax tree. Although programs are not likely to be dominated by global accesses, this indicates that there is a lot of space to save if global accesses are comparatively frequent.

It is reasonable to expect most programs to be dominated by local accesses, but there may also be a fair amount of global accesses. For example, standard functions and types can be considered to be declared at the global level. Suppose e.g. that the name “integer” is not a reserved word in the language, but can in principle be declared in the program and thereby hide the standard use of this name. There may then be a rather large proportion of name applications which are bound to such standard global names. It could be worthwhile to treat these name applications differently: First, the trace information needed is large since traces for these name applications have to be added along the whole lookup sequence. Second, the user is not likely to add declarations of these names, although it is allowed by the language. One possibility to handle this in a more space efficient way would be to keep a hash table of all such standard names and check it when a declaration is added. If such a name was re-declared by the user, it would take time to find the affected name applications. On the other hand, this allows the trace information to be avoided for all applications of these standard names.

3.4.2 Incremental name analysis, summary

This section has reviewed a number of methods for performing incremental name analysis. It can be noted that evaluators for standard attribute grammars correspond to the “Search smallest subtree” method, a method with poor performance, $O(|T|)$ in the worst case, where $|T|$ is the number of nodes in the syntax tree. The best method reviewed is the “Maintain Traces” method which achieves the lowest bound, $O(|A|)$, for all four types of changes. ($|A|$ is the number of affected name applications.) The Mjølner/Orm environment uses a variant of this method.

In implementing incremental name analysis in practice, it can be motivated to make tradeoffs between time and space. A couple of examples have been given to illustrate this.

3.5 Type analysis

In type analysis, each name application and expression in a program is associated with a type. We use the term *type* in a rather broad sense, covering also terms like “kind”, “mode”, and “formal qualification”. The type information is used for certain kinds of name analysis (e.g. remote access as in §3.3.3), for code generation, and for type checking. The type system of Simula is rather heterogeneous as it includes both simple built-in types such as Integer and Boolean, complex built-in types such as Text, and user-defined types such as the classes appearing in a program. We can regard also other entities, such as procedures and Algol block statements, as types. For error handling, it is useful to extend the type system with a type *Unknown*, which can be used for incomplete and erroneous expressions, e.g. name applications for which there is no declaration.

The classes defined in a Simula program form a forest of trees, according to the class (type) hierarchy. In type checking, it is useful to extend this forest to a complete lattice with a top and a bottom. In the lattice, a subclass appears below its superclass. The top, *`ObjectClass`*, is an implicit class which corresponds to a Smalltalk-like “class Object”. Although there is no explicit class `Object` in Simula, classes which have no explicit superclass can be considered to be subclasses of the implicit `ObjectClass`. The bottom of the lattice, `NoClass`, models the fictitious class of the reference value `NONE`. Although `NoClass` is not related to any other class, it can, for type checking purposes, be seen as a subclass of all other classes, and thus be placed as the bottom of the lattice. The example below shows the class lattice for a Simula program.

```
begin
   class A;
   begin
   end;

   A class B;
   begin
   end;

   A class C;
   begin
   end;
end;
```

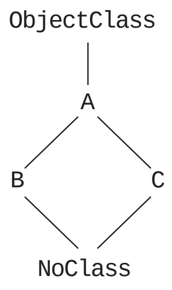


Figure 3.14 A Class Lattice

Each class type introduces a unique reference type. Type-checking reference expressions involves comparing classes in terms of the class lattice. The reference assignment statement `rX :- rY` illustrates this. Both `rX` and `rY` should have reference types. Let `qual(r)` be the formal qualification of a reference `r`. For Simula, the formal qualification of a reference is always known at compile-time. There are then the following cases:

1. `qual(rX)` is above or equal to `qual(rY)`. In this case, the assignment is always legal at run time (its legality is statically checkable).

2. `qual(rX)` is below (and not equal to) `qual(rY)`. In this case, it depends on the actual qualification of `rY` if the assignment is legal. A run-time test is needed to check the actual qualification of `rY`, and if it fails, the assignment will result in a run-time error.
3. `qual(rX)` and `qual(rY)` are incomparable, i.e. neither one of them is below the other and they are not equal. In this case, the assignment can never be legal at run time (it is a static-semantic error).

Class comparisons are needed to check also other Simula constructs, e.g. *is*, *in*, *qua*, and *inspect*. Other languages, like BETA, Eiffel, and C++, make use of class comparisons in similar ways.

In the previous section it was discussed how a change of superclass led to the replacement of an edge in the visibility graph. Similarly, this change leads to replacement of an edge in the class lattice. Class comparisons depending on this change must then be recomputed.

3.6 Error detection

An incremental static-semantic analyzer can detect and report static-semantic errors continuously as the program is edited. Examples of such errors are:

- Missing declaration of name application.
- Multiple declarations of the same name in the same block.
- Illegal type of an expression with respect to its context.
- Cyclic subclassing.

In an interactive system, error detection can be *history-dependent*, i.e. it is possible to let the order of program changes influence the error detection behavior. Consider a block with multiple declarations of the name *a*. By taking the editing history into account, the oldest declaration of *a* can be considered the "real" one, whereas the other ones are considered as faulty. Thus, new errors are associated with the latest changes, rather than with previously existing code. This is advantageous both because the error can be reported in terms of the current editing context, and because it minimizes the necessary incremental re-analysis (adding a new declaration with the same name as an existing one does not lead to any rebinding of name applications). Naturally, if the "real" declaration is removed, the next oldest comes into effect, and rebinding will occur. Cyclic subclassing can be handled in a similar way, by regarding one of the classes as causing the cycle, and treating it semantically as if it had *ObjectClass* as its superclass. The class treated this way can be chosen depending on the editing history.

Error reporting can be done in several ways in an interactive system. One way is to highlight the erroneous language constructions on the screen. Another way is to maintain a list of errors in a separate window. Regardless of how it is done there is a need for a communication mechanism in the incremental analyzer which updates the report as the program is analyzed.

3.7 Techniques for incremental analysis

A number of techniques have been proposed for implementing incremental static-semantic analysis. One line of development is based on *semantic action routines*, i.e., procedures connected to productions. The Gandalf system uses this approach [Med82]. The advantage of the action routine approach is that it allows efficient incremental static-semantic analysis to be implemented. The disadvantage is that the action routines are difficult and error-prone to implement. The editor calls the action routines according to given patterns, but it is up to the programmer of the routines to make sure they implement the correct actions in the correct order.

There are also some language-specific systems with completely hand-coded incremental static-semantic analyzers. In particular, the Rational system [WL86] which is a commercial environment for Ada. This system is interesting as it is actually used for large practical programming projects. Although the implementation technique used is ad hoc, it proves that it is possible to use incremental techniques for practical systems in industrial settings.

The by far most influential technique for incremental static-semantic analysis is the one based on *attribute grammars*. This techniques was introduced by Reps, Teitelbaum, and Demers [DRT81], [RTD83], [Rep84] and their work also resulted in a comprehensive practical system: the Cornell Synthesizer Generator which supports generation of language-based editors from attribute grammar specifications [RT88].

The major benefit of attribute grammars in incremental systems is that they allow an attribution of a syntax tree to be specified declaratively. General algorithms can be employed to automatically build the attribution and to update it in the event of changes to the syntax tree. Attribute grammars thus free the compiler implementor from programming explicitly *how* the attribution is to be built. It is sufficient to specify *what* the attribution should be like when it is correct. This is especially attractive in incremental systems, since there may be complex dependencies between attributes which would make hand-coding of updating the attribution very complex and error-prone.

The disadvantage of attribute grammars is that they are poorly suited for some of the problems in static-semantic analysis. In particular, name analysis is difficult to express in attribute grammars, and results in poor efficiency for incremental systems. These problems are even more pronounced for object-oriented languages which have comparatively complex naming semantics.

Many proposals have been made for improving the attribute grammar technique in various ways, with the goal of keeping the declarative nature of the specification while achieving efficient incremental evaluators. However, these suggested improvements are primarily directed towards procedural languages and do not solve the problems for object-oriented languages.

The next chapter covers attribute grammars in more detail, pinpoints the problems with this technique, and discusses earlier suggested enhancements.

3.8 Summary

We have shown how a simple visibility-graph based model for name analysis can be used to describe important name analysis problems occurring in object-oriented languages: block structure, subclassing, remote access, nested classes, Simula's inspect statement, and BETA's virtual classes. Problems like overloading and visibility restrictions have not been taken into account, but could be added to the basic model.

We have reviewed a number of existing methods for performing incremental name analysis and compared their worst-case time complexity. The best method reviewed is the "Maintain Traces" method which achieves the lowest bound for all the investigated edit cases: add declaration, remove declaration, change declaration, change visibility graph. However, the space overhead is larger for this method than for the other methods. In a practical system, it can be motivated to make tradeoffs between time and space and use hybrid methods.

Type checking of object-oriented programs involves comparison of class types, arranged according to the class hierarchy. Such comparisons need to be performed at each reference assignment, and also at some other language constructs. In the incremental situation it must be taken into account that changes to the class hierarchy may affect the results of such comparisons.

For detection of static-semantic errors, we have proposed the use of a history-dependent policy. Such a policy associates errors with the latest program changes in case there are multiple causes of the error. For example, adding a new declaration of an already existing name causes the new declaration to be considered erroneous, whereas the old one remains in effect. We find this a highly desirable behavior of an interactive system.

Chapter 4
Standard Attribute Grammars

This chapter reviews standard attribute grammars, incremental attribute evaluation, and the problems of using standard attribute grammars in the incremental setting.

4.1 Introduction

Attribute grammars were introduced by Knuth in 1968 [Knu68]. The idea gave rise to intense research activity, both in developing the theory and in experimental compiler construction. An extensive bibliography on attribute grammars was published in 1988 [DJL88]. The attribute grammars in their original form as introduced by Knuth will be referred to as *standard* attribute grammars, to distinguish them from other variations and extensions such as the Door Attribute Grammars introduced later in this thesis.

4.2 Definitions and notation

A *standard attribute grammar* G is an extension of a context-free grammar which associates with each nonterminal X a set of attributes $A(X)$, and with each production p: $X_0 ::= X_1 \,..\, X_{n(p)}$ a set of equations $E(p)$. In a syntax tree T of G, each nonterminal instance (syntax node) x of a nonterminal X will have instances of the attributes in $A(X)$. An attribute $a \in A(X)$ is denoted by $X.a$ and an attribute instance of a in a node x of X is denoted by $x.a$.

An equation in $E(p)$ has the following form:

$$a_0 = f(a_1, \ldots, a_m)$$

where $a_0 \,..\, a_m$ are attributes in $\bigcup A(X_k)$, $0 \le k \le n(p)$, and f is a *semantic function* defining the value of a_0 in terms of the arguments $a_1 \,..\, a_m$. The equations of a production apply to all instances of the production, and $a_0 \,..\, a_m$ then denote attribute instances. Both in the case of attributes and attribute instances, a_0 is said to *depend* on $a_1 \,..\, a_m$. A *dependency graph* $D(T)$ is a directed graph whose vertices are attribute instances in T, and where there is an edge from a to b iff b

depends on a. In practice, the semantic function is often the identity function. In this case, the equation is referred to as a *copy equation*.

Each set $A(X)$ is divided into two disjoint sets: a set of *synthesized* attributes $S(X)$ and a set of *inherited* attributes $I(X)$. G is said to be *well-formed* if the start nonterminal does not have any inherited attributes, and if, for each production p, $E(p)$ contains equations defining all the synthesized attributes of X_0, all the inherited attributes of X_k, $1 \leq k \leq n(p)$, and no other equations. If G is well-formed, each attribute instance in any syntax tree of G will have exactly one defining equation. We will only consider well-formed attribute grammars in this thesis, unless explicitly stated otherwise.

An attribute instance $x.a$ has either the special value *null*, or a non-null value. If the value is non-null, the attribute is *available*. An attribute instance $x.a$ is *evaluated* by applying its semantic function to the arguments and assigning the resulting value to $x.a$. Such evaluation may only be performed if all the arguments are available. An attribute instance $x.a$ is *consistent* if all its arguments are available and if its defining equation holds, i.e. if the value of $x.a$ equals its semantic function applied to the arguments. The tree T is *consistent* if all its attribute instances are consistent. G describes an equation system for T, where all the attribute instances are variables in the equation system. If T is consistent, its attribution is a *solution* to the equation system.

An attribute grammar is said to be *well-defined* if every possible syntax tree has at least one solution. If every possible syntax tree has exactly one solution, the grammar is said to be *uniquely-defined*. If the grammar is well-defined, but some syntax trees have more than one solution, the grammar is said to be *underdetermined*.

An *exhaustive* evaluator takes as input a syntax tree T where all the attribute instances are null, and evaluates semantic functions until T is consistent. An *incremental* evaluator restores consistency in a tree T which was initially consistent, but was then modified by syntactic changes.

4.3 AG classes and evaluation techniques

There are two principle kinds of evaluation techniques: data-driven and demand-driven evaluation. In *data-driven evaluation*, which is the usual technique, attribute instances are represented by memory cells, whose values can be read and stored. To obtain an attribution, the attribute instances are evaluated in topological order, i.e. according to a topological sort of the dependency graph. In *demand-driven evaluation*, each attribute is represented by its semantic function. The attribute values are thus not stored using this technique. Accessing the value of an attribute is implemented by calling its semantic function. This actually obviates the whole evaluation process - all attributes are automatically available and consistent. The drawback of this techniques is that it may be extremely slow since the same semantic function can be called many times. In the worst case, the time complexity is exponential in the total number of attributes. *Lazy attribute evaluation* is a variant on demand-driven evaluation which stores the value of an attribute the first time its semantic function is called, and uses that value in subsequent calls. This technique is time-optimal (linear in the number of attributes). Data-driven and demand-driven techniques can be freely combined by selecting some attributes to be *data attributes*, treated by data-driven techniques, and others to be *demand attributes*, treated by demand-driven techniques. This can be a useful way to trade evaluation time (in demand-driven evaluation) for space (used by the attributes in data-driven evaluation). In incremental systems it may even be possible to save both space and time by using demand or lazy attributes: If an attribute value is not always needed, time is saved by not having to update it at incremental evaluation. Lazy attributes only have to be marked as invalid but do not need to be recomputed until they are actually needed. Examples of incremental systems employing demand and lazy attribute evaluation schemes include [LMOW88] and [Hud91].

A data-driven evaluation algorithm can be either dynamic or static, depending on when the evaluation order is computed. A *dynamic* algorithm builds and sorts the dependency graph at evaluation time, whereas a *static* algorithm analyzes the attribute dependencies in the grammar at evaluator construction time, and can then evaluate any tree without doing dependency analysis. Static evaluators are in practice much faster than dynamic evaluators and also consume much less storage.

Attribute grammars are classified according to the dependencies between attributes. The classes of interest in this thesis are the following:

- *Non-circular.* An attribute grammar is non-circular if $D(T)$ of every possible syntax tree T is acyclic. All non-circular grammars are uniquely-defined, i.e., there exists exactly one solution for each possible syntax tree.
- *Ordered*. An attribute grammar is ordered (OAG), according to Kastens definition [Kas80]. This is a subclass of the non-circular grammars.
- *1-visit*. This is a subclass of the ordered attribute grammars. In [DJL88] these grammars are called “simple-1-sweep”.
- *Circular.* An attribute grammar is circular if, for some T, $D(T)$ has a cycle. Circular grammars may be well-defined under certain circumstances.

General non-circular AGs require dynamic evaluators. The first time-optimal incremental algorithm for this general class was a dynamic change propagation algorithm by Reps [Rep82]. Ordered AGs by Kastens [Kas80] is an important class of grammars for which static evaluators can be applied. Ordered AGs are sufficient for many practical applications. E.g., compilers for complex existing programming languages like Ada have been implemented using ordered AGs [KHZ82]. Incremental versions of Kastens’ algorithm for ordered AGs have been presented by Yeh [Yeh83] and Reps [Rep84], and the version of this algorithm which is used in the Cornell Synthesizer Generator is presented in detail in [RT88].

Kastens’ evaluator for ordered AGs (and its incremental versions) is based on a very simple principle of *visit sequences*. A visit sequence is a sequence of instructions of the following three types:

- *EVAL*(a)   Evaluate the attribute instance a.

- *VISIT*(i, r) Visit the i'th descendant node for the r'th time.
- *RETURN*(r) Return to the ancestor node for the r'th time.

At evaluator construction time a visit sequence is computed for each production according to the attribute dependencies of the grammar. At evaluation time a simple evaluator walks the tree and visits descendant nodes and ancestor nodes according to the *VISIT* and *RETURN* instructions, and evaluates attributes according to the *EVAL* instructions. Such an evaluator can be programmed using co-routines, recursive procedures, or a finite state automaton.

We use the term *n-visit* AG to mean an ordered AG which requires at most n visits to any node. I.e., n is equal to the maximum value of r occurring in any instruction in the visit sequences. The special case of 1-visit grammars is of particular interest for this thesis. Standard 1-visit AGs are useful only for rather simple applications such as one-pass compilers. A more complex language like Simula would require at least a 3-visit AG in order to handle that class declarations, variable declarations, and name applications of these declarations may occur in any order. However, as will be discussed in detail in §8.11, the Door attribute grammars introduced in this thesis often have less complex dependencies than standard AGs and 1-visit dependencies are often sufficient even for quite complex languages like Simula. 1-visit grammars are attractive because evaluators for them are even simpler to implement than general evaluators for ordered attribute grammars. Using object-oriented techniques, both exhaustive and incremental evaluators can be implemented in a very simple way for such grammars, as will be demonstrated in Chapter 7.

In Knuth's original definition of attribute grammars, circular AGs were considered erroneous. Most subsequent work has dealt only with non-circular AGs or subclasses thereof. However, it is quite possible to consider circular AGs, and several practical problems can be expressed in a simpler way if circularities are allowed. Typical examples are live-analysis in data-flow, and arbitrary declaration/application order in static semantics. Incremental algorithms for handling circular attribute grammars have been developed by e.g. Farrow [Far86] and Jones [JS86], [Jon90].

In contrast to non-circular AGs, a circular AG does not necessarily have a solution for every syntax tree. A tree with a cycle in its dependency graph may have zero, one, or many solutions. The attributes on a cycle can be evaluated iteratively, and if the consecutively computed values converge, a solution is found. Under certain circumstances it is possible to decide if a circular grammar always has such converging behavior. The usual approach is to arrange the possible values of each attribute on the cycle in a lattice. If the semantic functions are monotone and if the lattices are of finite height the iteration is guaranteed to converge. Furthermore, by using the bottom elements as start values, the iterative evaluation is guaranteed to find the least solution (the least fixed point). The least solution is usually the only interesting solution, and by choosing it as *the* solution, the circular grammar is uniquely-defined.

As discussed in §3.3.6 there are several constructs in object-oriented languages which lead to cyclic dependencies, at least on an informal intuitive level. It is reasonable to expect that these dependencies are simpler to express in a formal specification language which allows cyclic dependencies than in one which does not allow cycles.

4.4 Limitations of standard AGs

Standard AGs make it possible to define attributions of syntax trees. However, the nature of these attributions and the ways they can be defined is limited by the definition of standard AGs. In particular:

I *Regular values*. The attributes of individual syntax nodes are limited to have regular values, and must not be object identity values, denoting objects whose contents can be altered (as a consequence of syntax tree modifications).

II *Whole attributes*. A semantic function always defines a whole attribute, never just part of it.

III *Simple assertions*. An AG equation is an assertion which states something about the attribution and which can be true or false. Taking a more general view on attributions, a *solution* is an attribution for which all assertions are true. In this perspective, AGs restrict the assertions to be of a very simple form.

IV *Local dependencies*. The attributes in an equation always involve only attributes belonging to the nodes of a single production, i.e. all attribute dependencies are *local* (with respect to the syntax tree). If there is a dependency between attributes of two nodes distant from each other in the syntax tree, such a dependency has to be expressed by a chain of dependencies via all intermediate nodes. There is thus no way to express a direct non-local dependency.

V *Rigid dependencies*. The dependencies between attributes are rigid in the sense that they are completely governed by the form of the syntax tree and not at all by the values of the attributes. Also, the dependencies are rigid in the sense that an attribute is always considered to depend on *whole* other attributes, not just on parts of them.

VI *Uniquely-defined*. Standard non-circular attribute grammars are always uniquely-defined. In some situations underdetermined grammars would be preferrable. For example, this is useful to handle history-dependent error reporting as discussed in §3.6.

As an example of what it can mean to relax the above restrictions, consider defining name analysis of an object-oriented language like Simula. By following the visibility graph approach of §3.3, the following attributes can be used:

- Each declaration node has a declaration object which contains the name, represented as a text string, and the type, represented as a reference to a type object.
- Each block node has a symbol-table vertex object which contains a set of references to the declaration objects of declaration nodes appearing in the declaration part of the block. A block node also has a static path vertex object, containing references to other vertex objects (representing the outgoing edges in the visibility graph).
  Each class node has in addition a prefix path vertex object.
- Each name application node has: a declarative environment, represented by a reference to one of the vertex objects; a binding, represented by a reference to the appropriate declaration object (according to the name and the declarative environment); and a type attribute, represented by a reference to the same type object as is referred in the declaration object.

It is clear that this kind of attribution cannot be described by a standard AG. We see immediately that (I) is relaxed. For example, the contents of the set in a symbol-table object is altered if declaration nodes are added or removed. In defining such an attribution, it is useful to relax also the other limitations. For example:

- An equation could define the type of a declaration object. Since the declaration object is an attribute of a declaration node, this means defining only part of an attribute, thus relaxing (II).
- To state which declaration objects belong to a specific symbol table, a membership assertion could be used, thus relaxing (III).
- Via the binding reference attribute in the name applications, each name application is made directly dependent on the corresponding name declaration, thus relaxing (IV).
- The dependencies from name declarations to name applications are flexible in the sense that they may change due to changes in the attribution, even if the syntax tree structure between these nodes remains intact. For example, if the name of the declaration or application is changed, or if the structure of the visibility graph is changed (e.g. by changing the superclass of a class), the bindings (and thereby the dependencies) will change. Thus, (V) is relaxed.
- To handle multiple declarations of the same name, the symbol table contents could be defined to include declaration objects in such a way that all the declaration objects would have unique names. Thus, if there were more than one declaration object with the same name, only one of them would be included in the symbol table. This leads to an underdetermined grammar since it does not express which of the declaration objects to include. Thus, (VI) is relaxed.

One of the main goals of the work presented in this thesis has been to develop a formalism which allows these kinds of relaxations. An additional requirement on such a formalism is that it should be reasonably simple to develop efficient incremental attribute evaluators for a given specification in the formalism. The Door attribute grammars, presented in Chapter 8 and onwards is the proposed solution to meet these goals.

4.5 On optimality

The performance of an incremental evaluation algorithm is usually characterized by the number of attribute evaluations performed after a modification to the syntax tree [Rep82]. The set of attributes requiring new values, due to the syntax tree modification, is referred to as AFFECTED. Each attribute evaluation is considered to take unit time. Clearly, the lower bound is |AFFECTED|. If the algorithm uses $O(|\text{AFFECTED}|)$ time, it is said to be (asymptotically) *optimal*. It is with respect to this measure that the algorithm for non-circular AGs of Reps [Rep82] and the algorithm for ordered AGs of Yeh [Yeh83] are optimal. We refer to this measure as the *traditional* performance measure.

However, the traditional measure is useful only for comparing evaluation algorithms within one formalism (e.g. standard attribute grammars). It is not useful for comparing the performance between evaluators based on different formalisms. The problem with the traditional measure is that it assumes that all attributes defined in an attribute grammar are interesting. This is usually not the case. A better measure is obtained if we only consider *interesting* attributes, i.e. attributes which are of interest to maintain with respect to the application, and disregard additional attributes introduced for technical reasons. We refer to this measure as the *interesting* performance measure. A formalism, e.g. standard attribute grammars, may force us to use a lot of uninteresting attributes, simply in order to define the interesting ones. Now, let AFFECTED mean only the *interesting* attributes which require new values. If the interesting measure is used, the traditional algorithms are no longer optimal.

To illustrate this, consider maintaining only a type attribute for each name application, but no other attributes. To define the type attributes in a standard attribute grammar, additional (uninteresting) attributes are needed which propagate the type from each name declaration node to all its name applications. This can only be done by defining a large dictionary-like attribute which maps names to types and which is propagated throughout the scope of the declaration. The traditional algorithms store and update also these uninteresting dictionary attributes. Suppose the type is changed of a globally visible declaration for which there are n name applications. AFFECTED is in this case the type attributes of these n name applications, and $|\text{AFFECTED}| = n$. This change leads to all dictionary attributes receiving new values. The updating of the dictionary values can actually be done in a comparatively efficient way since many of them can share implementation. Worse is, however, that *all* type attributes in the whole program are re-evaluated, since they depend on a dictionary attribute which has changed value. The performance in this case is thus $O(|T|)$ rather than $O(|n|)$ and clearly very far from optimal.

4.6 Related work

Several methods have been proposed for overcoming the problems of standard AGs in the incremental setting. Some approaches aim at developing improved evaluation schemes while staying within the standard AG formalism. Others extend the standard AG formalism with new constructs to allow improved description and evaluation. There are also some approaches which abandon standard AGs altogether and propose other declarative formalisms. Most approaches are directed towards improving incremental name analysis for block structured languages. A few methods have been applied to modular languages, but we have not found any method applied to object-oriented languages, except for the author’s earlier work. Most of the proposed methods are based on dynamic attribute evaluation algorithms rather than on the more efficient static algorithms.

Extending AGs with additional constructs

Johnson and Fischer suggested extending AGs with *context-sensitive relation sets* [JF82], a mechanism allowing a declaration site to be linked to its application sites based on block-structured scope rules. The mechanism allowed information to flow directly over these links and special procedures were developed to handle addition and deletion of declarations. The mechanism was later developed into *non-local productions* [JF85]. This approach is more formal but unfortunately has the same efficiency problems as standard AGs for additions and deletions of declarations.

Beshers and Campbell suggested extending standard AGs by *maintained* and *constructor attributes* [BC85], [Bes87]. A maintained attribute is an instance of a data type with a given set of operations. Instead of defining the value of the maintained attribute with an equation, the value is defined by constructor attributes which each corresponds to an operation pair (an action and a retraction). A construction attribute is associated to the nearest maintained attribute on the path to the root of the syntax tree. The value of a maintained attribute is equal to the value obtained by applying the action operation of all its associated constructor attributes. *Trigger clauses* serve a similar purpose as the context-sensitive relation sets of Johnson and Fischer. The technique allows name analysis of block structured languages to be defined by using maintained attributes to represent symbol tables, constructor attributes for adding declarations to the symbol tables, and trigger clauses to allow information flow directly from declaration sites to application sites. The evaluation algorithm is based on the dynamic algorithm of [Rep82].

Kaiser introduces *action equations* [Kai85], an extension to AGs primarily intended for specification of dynamic semantics. Some of the action equation constructs are relevant also for specifying static semantics. *Set-valued attributes* can be defined to model symbol tables, a *membership* construct can be used to assert the membership of a declaration in a symbol table set, and a *propagation* construct to link declaration sites to application sites. The evaluation algorithm is based on the dynamic algorithm of [Rep82].

Improving evaluation of standard AGs

Hoover et al. [Hoo86], [HT86b], [Hoo87] and Reps et al. [RMT86] have the goal of improving incremental evaluation without extending or changing the attribute grammar formalism as such. The technique used is to provide special evaluation support for a pre-defined dictionary data type supporting block structured scope rules and for accelerating change propagation by supporting non-local dependencies. In [RMT86] the latter is done by allowing access to attributes of ancestors in the syntax tree, while Hoover uses a more general approach of bypassing copy equations. Evaluation in [RMT86] is based on the dynamic algorithm of [Rep82]. Peckham [Pec90a] defines a subclass of AGs which can handle the non-local dependencies of [RMT86] using a static evaluation algorithm. Hoover develops a new dynamic evaluation algorithm, *approximate topological ordering*, based on assigning priorities to attributes. Although this algorithm is dynamic and although it may be sub-optimal, it is reported to perform well in practice.

Support for advanced name analysis

All the above methods have in common that they support incremental name analysis for languages with block structure. None of them supports specification of languages with more advanced scope rules like object-oriented languages.

The author proposed extending standard AGs with *operational constraints* [Hed88]. An operational constraint expresses an invariant and is associated with a pair of operations. An evaluation operation enforces the invariant and a de-evaluation operation undoes the side-effects of the previous evaluation operation. The operational constraints are in this respect similar to the constructor attributes of Beshers. A kernel handling incremental name analysis for Simula-like object oriented languages was implemented and interfaced to the operational constraint mechanism. This technique was used to implement the incremental static-semantic checker of Mjølner/Orm [MHM+90]. Evaluation is based on a static algorithm.

Vorthmann uses a similar but more language independent approach by extending standard AGs by a special language designed for handling name analysis [VL88], [Vor90a]. This *naming specification language*, NSL, allows specification of scopes, scope connections, name declarations and applications. The approach is quite flexible and can be used to implement the naming semantics of languages like Modula-2, Pascal, and C. The evaluation algorithm is based on the dynamic algorithm of [Rep82]. The principle ways in which scopes can be combined is equivalent to the visibility graphs used in §3.3. Therefore, the NSL approach should be suited also for implementing object-oriented languages. However, NSL has not been applied to object-oriented languages in practice and apparently some mechanisms are missing in order to implement such languages. In particular, there is no mechanism for dealing with erroneously cyclic subclassing, as was discussed in §3.3.2 and there is no mechanism for comparing class types which is needed to do type checking, as was discussed in §3.5.

Other approaches based on context-free grammars

Demers et al. suggest a declarative message-passing formalism as an alternative to attribute grammars [DRZ85]. Instead of propagating symbol table attributes to declarations and uses, the declarations and uses send messages to the symbol table to request or assert information. However, since messages may flow only along the syntax tree structure (apart from replies which may flow directly back to the request site), the approach seems suitable only for simple block structured languages and not for languages with more advanced scope rules.

Horwitz and Teitelbaum suggest combining attribute grammars with a relational database [HT86a]. Attributes can depend on the values in database relations and the database relations can depend on attribute values (provided no circularity is introduced). Symbol tables can be maintained as relations and it is reported that the approach may prove more cost effective after changes to declarations than the pure attribute grammar approach.

Bahlke and Snelting have built a generator system PSG, which is based on a concept of *context relations* instead of attributes [BS86]. Instead of propagating symbol table information down to all use sites, as is normal in attribute grammars, the sets of “still-possible attribute values” for use sites are collected in relations which are propagated upwards in the tree. By performing unification on such relations it is possible to detect type-checking errors even for incomplete program fragments where declarations are missing. A special scope-analysis specification language is used which offers various built-in concepts, but is not a general mechanism. In [Sne91] it is reported that incremental type inferencers for Ada and Fortran 8x have been generated based on this approach.

Ballance has developed the notion of *logical constraint grammars* where contextual constraints are expressed by annotating productions in a context-free grammar with goals written in a logic programming language [Bal89]. This allows declarative specification of logic databases of facts which can be used to model symbol tables. Incremental evaluation is accomplished by a consistency manager which detects inconsistencies between the syntax tree and derived data and invokes the backtracking evaluator to (re-)attempt goals. These ideas have been implemented in the system *Pan* [BGV92] and the technique has been used to specify the static-semantics of Modula-2.

Approaches based on graphs

Attribute grammars and the related methods described above are all based on attributing a syntax tree in one way or the other. A different approach is to start out with a graph substrate rather than a syntax tree. This is useful if the underlying edited structure is a graph rather than an abstract syntax tree. Applications include consistency checking of module interfaces. Alpern et al. have developed a formalism for specifying attributed directed graphs [ACR+88] and incremental attribute evaluation algorithms for such graphs [ACR+87]. Kaplan and Goering suggest a similar approach but based on graph grammars [KG89]. Graph grammars describe graphs derived from strings, analogously to how context-free grammars describe parse trees derived from strings [ENR83].

Other approaches to incrementality

A completely different approach to incrementality is that of making a batch computation incremental. The idea here is to start out with a batch algorithm (or function) which computes a result from an input, and to derive a corresponding incremental version of this algorithm which updates the result after changed inputs (rather than re-computing the result from scratch). The work of [YS91], [PT89], and [SH91] is in this direction. This approach is not directly comparable to attribute grammars since the idea of AGs is to describe invariant properties of the resulting attribution. I.e., the initial description is not a batch algorithm. Rather, the approach of making batch computations incremental is orthogonal to that of declarative approaches like attribute grammars.

4.7 Summary

We have reviewed the standard attribute grammar formalism and its properties in the incremental setting. We find the fundamental idea of attribute grammars to be very useful and attractive for the implementation of incremental systems: the attribution of a syntax tree is described declaratively, and an incrementally updating attribute evaluator can be automatically derived from the specification. This gives robust implementations which are easy to change and maintain. Although the technique has serious limitations in other respects, it would be preferable to overcome these limitations while preserving the idea of a declarative description of the attribution.

We find the most serious problem with attribute grammars to be that the attributions they can define are too limited to be practical. We have listed a number of problems which need to be overcome in order to solve the static-semantic analysis problems reviewed in Chapter 3 in an adequate manner.

We have also reviewed a number of existing methods for improving attribute grammars and related methods for incremental evaluation. However, except for the author’s own earlier work, none of these approaches is reported to have been applied successfully to object-oriented languages.

Chapter 5
A Basic Object-Oriented Specification Language

In this chapter we introduce a basic object-oriented specification language developed specifically for this thesis. For the sake of convenience, it will be called *OOSL*. OOSL has basic object-oriented features similar to those of Simula. It will be extended by grammar related features as needed during the rest of the thesis. One important feature of OOSL is that it distinguishes firmly between declarative and imperative constructs. In specifying grammars, only the declarative constructs are used. The imperative constructs are used only in the implementation of attribute evaluators. All constructs of OOSL are straight-forward to implement in any object-oriented language.

It is not necessary to read this chapter thoroughly to understand the subsequent chapters. It should suffice to skim through it to get an idea of what constructs are included, and then return later to more detailed reading if the syntax is not obvious enough.

5.1 Declarative constructs

5.1.1 Classes and subclasses

Classes in OOSL are arranged in a single-inheritance hierarchy. A declaration of a class defines the class name, the name of the superclass (optional), and the class body which is a list of declarations (also optional). If the superclass is not given, the class is considered to be a subclass of a most general class `ANYCLASS`. The class may also have a formal parameter part discussed in more detail in §5.1.11.

```
<class-decl> ::=
    <class-id> ‘:’ ‘class’ [<formal-par-part>] [<superclass>]
```

```
       [ ‘{’ <decl-body> ‘}’]

   <superclass> ::= <superclass-id>
   <decl-body> ::= (<decl>, ‘;’)*
```

for example

```
   A: class B { };
```

For clarity, we will use boldface for the most important keywords. Comments are written as `(* this is a comment *)`.

5.1.2 Variable declarations

Although the term “variable” suggests imperative programming, variables are relevant also for declarative specifications. It is possible to define the value of a variable declaratively rather than imperatively. This is exactly what is done in attribute grammars, where the attributes correspond to variables.

A variable may be of one of the built-in types: integer, boolean, and string. In addition, a variable may be a reference. The following syntax is used:

```
   <decl> ::= <var-decl>

   <var-decl> ::= <var-id> ‘:’ <type>

   <type> ::= ‘integer’ | ‘boolean’ | ‘string’ | ‘ref’ <class>

   <class> ::= <class-id>
```

for example

```
   i: integer;
   b: boolean;
   s: string;
   r: ref A;
```

5.1.3 Objects

Objects can be introduced in a declarative way by declaring static references, as in BETA. The object denoted by the static reference is created automatically at the same time as the object declaring the static reference.

```
   <decl> ::= <stat-ref-decl>

   <stat-ref-decl> ::=
       <stat-ref-id> ‘:’ ‘object’ [<class>] [ ‘{’ <decl-body> ‘}’]
```

Notice that an object may (as in BETA) have a body. If it has a body, it is considered to be an instance of an anonymous class which is a subclass of the `<class>` (or a subclass of `ANYCLASS` if no `<class>` is given). For example:

```
A: class
{   x: object C;

    y: object D
    {   a: integer;
    };
};
```

For each `A` object which is created, there will be a `C` object and an extended `D` object created as well. These objects are denoted by the static references `x` and `y` respectively. The object denoted by `y` is an instance of an anonymous subclass of `D`.

5.1.4 Virtual functions

Functions in OOSL are applicative in the sense that they have no side-effects and two calls will yield the same result when called with the same parameters in a given object configuration. However, if objects are changed between two calls, the function results may differ. Function results and parameters may be references as well as regular values.

Functions are virtual and may be overridden in subclasses. A virtual function *specification* declares the name, result type, and parameters. It may optionally also contain an *implementation* defining the result value. A separate function implementation contains only a definition of the result value. In defining the function result, the value assignment operator "`:=`" is used for regular-valued functions and the reference assignment operator "`:-`" is used for functions returning references.

```
<decl> ::= <func-spec> | <func-impl>

<func-spec> ::=
    <func-id> ':' 'func' <type> ['(' (<form-par>, ',')+ ')']
    [(':=' | ':-') <expr>]

<form-par> ::= <param-id> ':' <type>

<func-impl> ::=
    'impl' <func-id> (':=' | ':-') <expr>
```

For example,

```
A: class
{   f: func ref C (x: integer) :- c0;
};
```

```
B: class A
{  impl f :-
       if x>0
       then c1
       else c2;
};
```

Here, the function `f` is both specified and implemented in class `A`. In the subclass `B` an implementation of `f` is given which overrides the implementation in `A`.

5.1.5 Super

Similar to Smalltalk, an implementation of a function `f` declared in a class `C` may call the implementation of `f` declared in the superclass of `C`. This is done by calling `f` via the special reference `super`. The following syntax is used:

```
<exp> ::= <super-exp>

<super-exp> ::= ‘super’
```

The following example illustrates the use of `super`:

```
A: class
{  f: func integer (x: integer);
   impl f := x*x;              (* 1 *)
};

B: class A;

C: class B
{  impl f :=                   (* 2 *)
       if x=0
       then 1
       else super.f(x);
};
```

Suppose `f` is called for a `C` object. The implementation at `(* 2 *)` is then invoked. If the parameter `x` is not 0 the function calls `f` via `super`. This results in an invocation of the function implementation of `f` which is closest above in the class hierarchy, i.e. the implementation at `(* 1 *)`.

5.1.6 Regular value expressions

For arithmetic expressions, boolean expressions, and relations, we use the usual syntax used in most programming languages. Also for the conditional expression we use the usual syntax:

```
<exp> ::= <cond-exp>

<cond-exp> ::= ‘if’ <exp> ‘then’ <exp> ‘else’ <exp>
```

5.1.7 Reference expressions

For simple expressions involving references to objects, we use the syntax of Simula:

```
<exp> ::=  <ref-eq> | <ref-neq> | <in-exp> | <this-exp> |
           <none-exp> | <remote-access>

<ref-eq> ::= <exp> ‘==’ <exp>
<ref-neq> ::= <exp> ‘=/=’ <exp>
<in-exp> ::= <exp> ‘in’ <class-id>
<this-exp> ::= ‘this’ <class-id>
<none-exp> ::= ‘NONE’
<remote-access> ::= <exp> ‘.’ (<id> | <call>)
```

For example,

- “`r1 == r2`” is true if `r1` and `r2` denote the same object.
- “`r1 =/= r2`” is true if `r1` and `r2` denote different objects.
- “`r` **`in`** `C`” is true if `r` denotes an object which is at least of class `C`. I.e., the object is of a class `D` such that $D \subseteq C$.
- “**`this`** `C`” denotes the `C` object in which the expression occurs.
- “`NONE`” is the special object identity value denoting “no object”.
- “`r.a`” is a remote access to the entity `a` of the object denoted by `r`. The remotely accessed entity may be either an identifier (e.g. for a variable), or a function call. If the resulting value of a remote access is an object identity, it may be used as the left operand of another remote access. For example, “`r.f(x).c`” is an access to `c` in the object denoted by the reference returned by the function call `f(x)` of the object denoted by the reference `r`.

5.1.8 Let-expression

OOSL includes let-expressions. A let-expression defines the value of a local variable which may be accessed in the body of the let-expression. The value of the let-expression is the same as the value of its body. The local variable is not explicitly declared, but will have the same type as the expression used to define its value. We use a special lexical identifier for such local variables, starting with a “`$`”-sign. As for functions, the “`:=`” or the “`:-`” operator is used depending on if the variable has a regular value or if it is a reference.

```
<exp> ::= <dollar-id> | <let-exp>

<let-exp> ::= ‘let’ <dollar-id> (‘:=’ | ‘:-’) <exp> in <exp>
```

For example,

```
let $X := a*b in
   $X *($X-1)
```

5.1.9 Inspect-expression

OOSL includes an “inspect-expression” which does case analysis on the actual qualification of an object. The inspect-expression is inspired by Simula’s inspect statement, but works as an expression rather than an imperative statement. Another difference from Simula is that the inspect-expression uses an explicit variable for the inspected object. Thus, entities in the inspected object must be accessed by remote access via this inspect variable, instead of directly as in Simula. In our view, this leads to easier reading of the code.

```
<exp> ::= <inspect-exp>

<inspect-exp> ::=
    ‘inspect’ <dollar-id> ‘:-’ <exp>
    (‘when’ <class-id> ‘do’ <exp>)*
    ‘otherwise’ <exp>
```

An inspect-expression thus has the following form:

```
inspect $X :- r
when C1 do e1
when C2 do e2
...
when Cn do en
otherwise e
```

Here, `r`, which must be a reference expression, denotes an object called the *inspected object*. Like in a let-expression, the local variable `$X` is defined to denote this object. Depending on the actual class of the inspected object and the classes $C_1..C_n$, one of the when-clauses or the otherwise-clause will apply. Let `A` be the actual qualification of `r`. The first when-clause for which $A \subseteq C_k$ holds is the one which applies, and e_k is the resulting value of the inspect-expression. If there is no when-clause for which $A \subseteq C_k$ holds, the otherwise-clause applies, and `e` is the resulting value of the inspect-expression. Inside e_k, `$X` has the formal qualification C_k and `$X` is guaranteed to actually denote a real object (i.e. it cannot be `NONE`). Inside `e`, `$X` has the same formal qualification as the reference-valued expression `r`.

In our OOSL specifications, the most common use of the inspect-expression is to do safe access to attributes via references which can be `NONE` at run time. For example, consider a reference variable `rA` and a class `A` with an attribute `t` as follows:

```
rA: ref A;

A: class;
{   t: integer;
};
```

If the variable `rA` sometimes is `NONE` and sometimes denotes a real `A` object, this has to be checked before accessing the attribute `t` via the reference. This can be done by using an inspect-expression as follows:

```
    inspect $X :- rA
    when A do $X.x
    otherwise 0
```

The result of the expression is `rA.x` if `rA` denotes a real object, and 0 if `rA` is `NONE`.

5.1.10 Loop expression

The *loop-expression* is in principle equivalent to a recursive function, but is a convenient way of writing such functions in line:

```
<exp> ::= <loop-exp> | <next-exp>

<loop-exp> ::=
    ‘loop’ <dollar-id> (‘:=’ | ‘:-’) <exp> ‘do’ <exp>

<next-exp> ::= ‘next’ <dollar-id> (‘:=’ | ‘:-’) <exp>
```

The form of the loop-expression is thus

```
loop $X := start do
    body
```

As in a let-expression, the local variable `$X` is assigned the value of the start expression and can be used inside the `body` expression. The result of the loop-expression is the value of its `body`. The `body` may contain a “next-expression” of the form

```
next $X := e
```

The value of the next-expression is the value of the enclosing loop-expression with `$X`=`e` instead of `$X`=`start`. I.e., the loop is “restarted” with a new value for `$X`.

For example, the loop construct can be used to find a particular element in a linked list as follows:

```
loop $X :- list.first do
    if $X.data = 5
    then $X
    else next $X :- $X.suc;
```

The loop-expression is equivalent to calling a function

```
f(start)
```

which is defined as

```
f: func T1 ($X: T2) := body
```

where a “next” expression is replaced by the recursive call

```
f(e)
```

Just as for tail-recursive functions, the loop-expression can actually be implemented as an imperative loop if all next-expressions occur in tail-recursive positions. I.e., if the next-expressions are the resulting values of the body and not used in further computations. In our examples, we will only use tail-recursive loop-expressions.

5.1.11 Generic classes

It is useful to be able to parameterize collection classes (sets, lists, etc) with the types of their member elements. Some object-oriented languages support this, e.g. Eiffel by means of “generic classes” [Mey88] and BETA by means of “virtual patterns” [MM89] . Introducing such mechanisms has some consequences for static type checking as will be discussed in section §5.5.3. For OOSL, we take an approach similar to the “constrained generic classes” of Eiffel [Mey92].

A formal class *T* can be declared as a parameter of a class *C*. The parameter *T* and its least formal qualification is given within square brackets in the declaration of *C* and can be used inside *C* to define the types of attributes, function parameters and results. *C* is said to be a *generic class*.

The formal parameter part of a class declaration has the following syntax:

```
<formal-par-part> ::=
    ‘[’
    ( <par-class-id> ‘:’ ‘class’ <qual-id>, ‘,’ )+
    ‘]’
```

and for the `<class>` nonterminal used in declarations of reference variables, parameters, and static references, we add the following alternative productions as well (in addition to the production in §5.1.1):

```
<class> ::= <par-class-id>
<class> ::= <class-id> <actual-par-part>

<actual-par-part> ::= ‘[’ ( <class>, ‘,’ )+ ‘]’
```

For example

```
A: class [T: class T2]
{   b: ref T;
    f: func ref T (x: ref T);
}
```

Here, the formal parameter `T` is specified to be at least class `T2`, i.e., `T` ⊆ `T2`. This makes it possible to use attributes of `T2` inside class `A` when dealing with objects of the parameter class `T`. In using class `A` as the type of a reference, an actual class parameter is supplied. For example

```
rU: ref A[U]
```

where the actual parameter `U` is the name of another class. `U` must be either `T2` or a subclass of `T2`.

5.2 Applicative classes

In addition to simple values like integers and booleans, there is a need also to express structured values. For these, we will use the same techniques as in functional languages, i.e. use objects and references to form structures which represent the values, but use these structures only in restricted ways so that the immutability of values is not violated (referential transparency). More precisely:

- An object used to represent a value must not be changed
- Testing equality of object identities is not sufficient for determining value equality

The use of objects and references is essential in functional programming to save storage for values which are equal, or which share common substructure, and to allow fast construction of new values. Data types with these characteristics are sometimes called *applicative data types* [Mye84]. In analogy, we use the term *applicative class* to mean a class whose objects are used to represent values. It could be interesting to develop special purpose syntax for applicative classes to enforce correct use of them. However, since such development is outside the scope of this thesis, we will only add a comment `(* applicative *)` at the declaration of a class to indicate that it is applicative. By convention, we will use these classes as follows:

- New objects of applicative classes may be created by the "new" operator.
- A reference to an object of an applicative class must not be tested for identity. Instead the applicative class must provide a function for testing equality.
- The contents of an object of an applicative class must never be changed.

The "new" operator used to construct a new object has the following syntax:

```
<exp> ::= <new-exp>

<new-exp> ::= 'new' <class>
```

For example, the expression "**new** A" results in a reference to a new A object. The "new" construct has, in principle, a side-effect (creating a new object) and two executions of it will result in references denoting different objects. However, this is transparent when used for applicative classes, since the reference values are not tested for identity.

Applicative classes are specified with an interface of functions which can be used to construct values, compare values, inspect parts of values, etc. These functions may very well be implemented using imperative techniques as long as this is transparent to the usage of the values. For example, if a function needs to construct a new value, the state of the new value can be set using imperative code. The function implementations may also compare references, for example to quickly determine value equality for the cases where two values happen to be represented by the same object.

As an example of an applicative class, consider set values. We define a general applicative class `Set` with the following interface.

```
Set: class [T: class ANYCLASS] (* applicative *)
{ empty: func boolean;
  contains: func boolean (e: ref T);
  add: func ref Set[T] (e: ref T);
  union: func ref Set[T] (s: ref Set[T]);
  equal: func boolean (s: ref Set[T]);
}
```

Figure 5.1 Interface to class Set

A `Set` object models a set of references to `T` objects. The function `empty` returns `true` if the set is empty. The function `contains` returns true if the set contains the element `e`. The expression "`s.add(e)`" returns a new set object which contains the same elements as `s` and in addition the `e` element. I.e., the `add` function does *not* change the internal state of `S`. Similarly, the expression "`s1.union(s1)`" returns a new set object containing the union of the elements in `s1` and `s2`, without changing the `s1` or `s2` objects. The expression "`s1.equal(s2)`" returns `true` if `s1` and `s2` represent the same set. The expression "**`new`** `Set[T]`" returns an empty set of references to `T` objects.

Note that although `Set` is an applicative class, the class parameter `T` needs not be an applicative class. Since references are immutable, the value of a set is not changed although the internal state of an object denoted by a reference in the set may change. Thus, the value of a set of references to non-applicative objects is not a regular value, but it is a value in the same sense as references are values. I.e., the set is immutable, but mutable information is accessible via the references in the set.

An object *x* of an applicative class represents a regular value only if all objects reachable via *x* are also applicative.

5.3 Imperative constructs

5.3.1 Statements

OOSL contains the usual kinds of statements: assignment, conditional statement and while statement. In addition, an inspect statement is used, similar to Simula's but using a local variable for the inspected object, as in the inspect-expression case.

The assignment comes in two variants (as in Simula): a value assignment for regular values, and a reference assignment for references:

```
<stmt> ::= <assign-stmt> | <cond-stmt> | <while-stmt> |
           <inspect-stmt>

<stmt-list> ::= (<stmt>, ‘;’)*

<assign-stmt> ::= <val-assign-stmt> | <ref-assign-stmt>

<val-assign-stmt> ::= <exp> ‘:=’ <exp>

<ref-assign-stmt> ::= <exp> ‘:-’ <exp>

<cond-stmt> ::=
   ‘if’ <exp>
   ‘then’ <stmt-list>
   [‘else’ <stmt-list>]
   ‘end’ ‘if’

<while-stmt> ::=
   ‘while’ <exp>
   ‘do’ <stmt-list>
   ‘end’ ‘while’

<inspect-stmt> ::=
   ‘inspect’ <dollar-id> ‘:-’ <exp>
   (‘when’ <class-id> ‘do’ <stmt-list>)*
   ‘otherwise’ <stmt-list>
   ‘end’ ‘inspect’
```

The left-hand side of an assignment statement can be a rather complex expression involving “this”, “remote-access”, etc. This is the reason why we have used the non-terminal <exp> for the left-hand side of the assignments, rather than a more specific non-terminal. To be correct, the expression appearing on the left side of an assignment must be an assignable entity (e.g. a variable).

5.3.2 Virtual procedures

Procedures in OOSL are similar to the functions in the way the *virtual* mechanism works and in the separation of specification from implementation. A virtual procedure *specification* declares the name, possible result type, and parameters. A virtual procedure *implementation* contains a list of statements. If the procedure returns a value, this is done by an assignment statement, with the procedure name on the left-hand side (as in Algol/Simula).

```
<decl> ::= <proc-spec> | <proc-impl>

<proc-spec> ::=
   <proc-id> ‘:’ ‘proc’ [ <type> ]
   [‘(’ (<form-par>, ‘,’)+ ‘)’]
   [‘{’<imp-body>‘}’]

<proc-impl> ::=
   ‘impl’ <proc-id> ‘{’<imp-body>‘}’

<imp-body> ::=
   (<decl> ’;’)* <stmt-list>
```

For example,

```
A: class
{   p: proc ref C (x: integer)
    {   ...
        p :- ...;
    };
};

B: class A
{   impl p
    {   ...
        p :- ...;
    };
};
```

Here, the procedure `p` is both specified and implemented in class `A`. In the subclass `B` an implementation of `p` is given which overrides the implementation in `A`.

A procedure call may appear either as an expression or a statement, depending on if it returns a value or not:

```
<stmt> ::= <proc-call>

<exp> ::= <proc-call>

<proc-call> ::=
    <proc-id> ['(' (<exp>, ',')+ ')']
```

5.3.3 Iterators

Many object-oriented languages have some mechanism by which iterators can be implemented. For example, the INNER mechanism or the quasi-parallel sequencing of Simula and BETA can be used. In Smalltalk, the “block”-mechanism can be used. Since OOSL includes none of these constructs we have instead included an explicit iterator construct, a simplified version of the construct in CLU [LSAS77].

The iterator construct in OOSL is similar to the procedure construct in that the specification may be separated from the implementation. The following syntax is used for defining an iterator:

```
<decl> ::= <iter-spec> | <iter-impl>

<iter-spec> ::=
    <iter-id> ':' 'iterator' <type>
    ['{'<proc-body>'}']

<iter-impl> ::=
    'impl' <iter-id> '{'<imp-body>'}'

<stmt> ::= <yield-stmt>

<yield-stmt> ::= 'yield' <exp>
```

For example, an iterator `each` which iterates over all the elements of a `List` object can be defined as follows:

```
List: class
{   ...
    each: iterator ref Element;
    ...
    impl each
    {   e: ref Element;
        e :- first;
        while e =/= none do
            yield e;
            e :- e.suc;
        end while;
    };
};
```

At an invocation of the iterator, the iterator is executed similar to a procedure, but at each `yield` statement, control is passed back to the caller. When control is passed back to the iterator again, the execution continues after the `yield` statement. The value of the yield expression (`e`) is passed to the caller at each iteration.

A call to an iterator has the following syntax:

```
<stmt> ::= <for-stmt>

<for-stmt> ::=
    ‘for’ <dollar-id> (‘:=’ | ‘:-’) <exp> ‘do’
    <stmt-list>
    ‘end’ ‘for’
```

For example, the `each` iterator of `List` can be used to iterate over all the elements in a list and compute a sum as follows:

```
sum: integer;
sum := 0;
for $x :- aList.each do
    sum := sum + $x.val;
end for;
```

The local variable `$x` is set to the value returned by the `yield` statement of the iterator. For each yield, `$x` receives a new value and the statements after `do` are executed. When the iterator runs through its end, the execution continues after the end of the `for` statement.

5.4 Modularization

OOSL allows declarations local to a class to be defined syntactically outside the body of the class. This is done by using a *separate clause* which adds local declarations to an already existing class. This allows different aspects of an OOSL specification to be described separately, similar to how it is done in the SSL language, the specification language for the Cornell Synthesizer Generator [RT84].

A *module* in OOSL is a list of declarations. An OOSL specification can be split into an arbitrary number of modules. The OOSL specification is simply the sum of all the declarations in all the modules with no regards taken to ordering, either within or between modules. OOSL modules are primarily intended as “separate understanding” modules rather than modules for separate compilation.

The mechanism of separate clauses is important because it allows different aspects of a class to be written in different modules. In this thesis we will primarily use separate clauses in order to separate the declarative parts of an OOSL specification (the grammar) from the imperative parts (implementation of visit procedures for attribute evaluation). The separation mechanism is also useful for separating interface from implementation and for describing different aspects of a grammar in different modules, e.g. name analysis separate from error detection.

Modules and separate clauses have the following syntax:

```
<module> ::= (<decl>, ‘;’)*

<decl> ::= <sep-clause>

<sep-clause> ::= ‘addto’ <id> ‘{’ <decl-body> ‘}’
```

For example, instead of writing

```
C: class
{   f: func integer := ...;
};
```

this class could be split into two parts. One declaring the class and the interface to the function:

```
C: class
{   f: func integer;
};
```

and another part adding the implementation of the function in a separate clause:

```
addto C
{   impl f := ...;
};
```

These two parts can be placed in different modules.

5.5 Type-checking issues in OOSL

Most, but not all, type-checking can be done statically in OOSL. To make sure our specifications will not give rise to run-time type-checking errors, we will make use of some simple conventions as described below. These conventions are adhered to in the examples of the subsequent chapters.

In principle, the following run-time errors (related to type-checking) may occur in OOSL:

- Reference is `NONE` in a remote access
- Missing virtual procedure/function implementation
- Violation of the relation between formal and actual qualification at assignments and calls.

5.5.1 Reference to NONE

In a specification, we want to be certain that a remote access is not done on references which are `NONE` since that would constitute a run-time error. To accomplish this we will by default assume that all references are intended to denote real objects rather than `NONE`. In some cases, it can be convenient to allow a reference variable to have the value `NONE`, or a function or procedure to return `NONE`. For these cases we will mark the declaration of the variable, function, or procedure with the comment `(* may be NONE *)` or `(* may return NONE *)`. In using these references we will take special care. We will not use remote access on these references, but use the inspect expression or inspect statement. This allows safe handling of the reference since the “otherwise” clause of the inspect construct will be executed in case the reference is actually `NONE`.

5.5.2 Missing virtual implementation

It is sometimes useful to specify a virtual procedure or function in a class, without giving an implementation of it in that class. This is often the case for “abstract” classes which are not intended to be instantiated. If such a class would be instantiated, and the procedure or function called, this would result in a run-time error. We say that a class is *incomplete* if it has such specifications without implementations. We will prevent run-time errors of this type as follows: A comment `(* abstract *)` is added to the declaration of classes which are intended to be incomplete. Classes marked this way will not be instantiated. Further, classes which are not marked as abstract must be complete.

5.5.3 Qualification violation at assignment

The type system used in OOSL is based on the qualification relation stated in §3.2.1:

> A reference with formal qualification `C` must have an actual qualification `D` such that `D` $\subseteq$ `C` or `D` = `NOCLASS` (i.e. reference to `NONE`).

To ensure that this relation always holds, it must be checked at each reference assignment. (In this respect, parameters passed into functions and procedures can also be viewed as assignments.) Although most such checking can be done statically, there are some situations when this is not possible. These issues have been a discussion topic at recent conferences on object-oriented programming, initiated by a paper by Cook [Coo89].

The principles for qualification-based type-checking and the consequences for static/dynamic type checking are treated in [MMM90]. To summarize the consequences of these principles, applied to OOSL, consider a reference assignment “`r1 :- r2`”. In most cases the correctness of the assignment can be checked statically. There are two cases when full information is not available statically (disregarding possible information from data-flow analysis):

I The formal qualification of `r2` is less than the formal qualification of `r1`. In this case, the assignment *may* be correct (if the actual qualification of `r2` is equal to or greater than the formal qualification of `r1`).

II `r1` is declared as a reference of a formal type of a generic class. In general, the formal qualification of `r1` is then unknown at compile-time.

Case I: For this case, we simply regard such assignments in OOSL as illegal. We have not found the need for such assignments in our applications of OOSL. If such need would arise, it is always possible to use the “inspect” construct to write the assignment in a way which can be statically type-checked.

Case II: Consider the following generic class:

```
A: class[T: class T2]
{   r: ref T;
    p: proc(s: ref T);
};
```

Here, the formal class `T` may have different actual values for different `A` objects. I.e., to know the formal qualification of the `r` and `s` references, it is necessary to know the actual qualification of the `A` object (which includes information about the actual parameter for `T`). This information is available statically for part-objects, but not for objects references via dynamic references. For example:

```
rstat: object A[U];
rstat.r := ...;   (* statically checkable *)
rstat.p(...);     (* statically checkable *)

rdyn: ref A[U];
rdyn.r := ...;    (* requires dynamic check *)
rdyn.p(...);      (* requires dynamic check *)
```

An assignment to `rstat.r` can thus be type-checked statically. Likewise, the “assignment” to the parameter `s` at a call to `rstat.p` can be checked statically. On the other hand, if an `A` object is accessed via a dynamic reference, the actual qualification of the `A` object is in general not known statically, and a dynamic type-check is needed for assignments to `r` and `s`.

In our applications of OOSL, we have found it sufficient to do this kind of assignments via static references (part-objects), and thus obtain statically checkable specifications.

5.6 Summary

OOSL is an object-oriented specification language with both declarative and imperative features. The declarative features include classes, generic classes, objects, variables, functions, and expressions. The imperative features include statements, procedures, and iterators. Additional grammar-related features will be added in subsequent chapters.

A modularization mechanism, in the form of separate clauses, allows different aspects of classes to be declared separately. This will be utilized in subsequent chapters for specifying grammars and incremental attribute evaluators. The grammars will be specified using only declarative OOSL constructs. The attribute evaluators will extend the grammar by imperative constructs, resulting in executable incremental evaluators. Thus, the declarative specification is *extended* (rather than translated) into an executable program.

OOSL is a general-purpose notation for specification and programming. A few simple conventions have been given to obtain specific use of certain language constructs. In particular, conventions for writing “applicative classes” and conventions for avoiding run-time type-checking errors have been introduced. These conventions are useful for general-purpose specification/programming as well, but are not supported by existing languages. It could be an interesting area of further research to construct some mechanism for supporting these conventions and allowing them to be checked statically.

Chapter 6
Object-Oriented Attribute Grammars

Attribute grammars are descriptions of consistently attributed syntax trees. Taking an object-oriented perspective, the nodes in the syntax tree are objects, and a language can be described by classes organized in a specialization hierarchy. In this chapter we extend OOSL to allow specification of *object-oriented attribute grammars*, supporting this view.

6.1 Introduction

For a long time, probably since the introduction of Simula, an object-oriented view on grammars has been used as a practical programming technique in many systems. Recently, explicit object-oriented formalisms for context-free grammars have been proposed in several forms, e.g. [Nør87], [CNS87], [MN88], [TTTI88]. In this chapter we bring the attributes and equations of attribute grammars into such an object-oriented framework, introducing *object-oriented attribute grammars*. The presentation here is based on earlier papers by the author [Hed88], [Hed89], although these papers used a slightly different specification language (not OOSL).

Object-oriented AGs are equivalent to standard AGs, i.e. all attributes have regular values as in standard AGs. Although regular values may be implemented by objects of applicative classes (as described in the previous chapter), it is only the syntax nodes that are conceptually regarded as objects. The examples in this chapter therefore model name analysis in the traditional AG style. In Chapter 8, the object-oriented AGs will be extended to “Door AGs” where true objects can be part of the attribution, allowing name analysis to be based on explicit visibility graphs.

Although object-oriented AGs are equivalent to standard AGs there are differences in notation which make certain things easier to express in object-oriented AGs. Attributes and equations are defined in node classes and inherited along the classification hierarchy, similarly to variables and procedures in object-oriented programming. The classification hierarchy represents a specialization hierarchy of language concepts. Behavior (in the form of equations) can be defined at suitable levels of generalization and default behavior can be overridden in specialized node classes. This allows specifications to be written in a more compact and readable way than is possible in traditional AG formalisms.

Attributes declared in a most general node class will appear in all nodes. Such attributes can be accessed by tools which have no detailed knowledge of the language. The object-oriented notation plays an important role here, since it allows the attributes to be specified very easily. In a traditional notation, the grammar would have to be cluttered with trivial attribute declarations and equations, and one would simply not think of using attributes for these purposes.

6.1.1 Conflicting terminology

In combining the ideas of object-orientation and attribute grammars, the term “inherited” may be a source of confusion since it is used in both areas with different meanings. Henceforth, we use *inherited* in the sense of attribute grammars. To refer to inheritance in the object-oriented sense, we will use the term *oo-inherited*.

6.1.2 Outline of chapter

The rest of this chapter is organized as follows: §6.2 and §6.3 extend OOSL with constructs for defining context-free grammars, attributes and equations. §6.4 gives an example of an object-oriented AG by specifying a simple desk calculator. §6.5 gives examples of how the classification hierarchy can be expanded and how behavior can be defined at general levels. §6.6 discusses the use of local attributes. §6.7 discusses criteria for well-formedness of object-oriented AGs. §6.8 comments on possible extensions, including multiple-inheritance. §6.9 discusses some related approaches to object-oriented grammars, and §6.10 gives a summary of the chapter.

6.2 Context-free grammars

The object-oriented specification style lends itself to express context-free grammars. For example, instead of defining a non-terminal `statement` and two productions `while-statement` and `if-statement`, one can define `statement` as a class, and `while-statement` and `if-statement` as subclasses, i.e. specialized statements. The nodes of a syntax tree are then class instances (objects) rather than “non-terminal instances labelled by productions”. We extend OOSL with the notion of *node classes* intended for description and construction of abstract syntax trees. If X is a non-terminal and p is a production with left-hand side X, then X and p will be formulated as node classes and p will be a subclass of X. A traditional CFG thus corresponds to a two-level class hierarchy. The object-oriented formulation in addition allows the class hierarchy to be further expanded, both by forming more general classes and more specialized classes. Later in this chapter, we will show how the use of general classes allows specification of AGs to be simplified.

6.2.1 Alternation and construction classes

OOSL differs between different kinds of node classes similar to how the GRAMPS system [CI84] differs between production rules. The basic kinds of node classes are *alternation* node classes and *construction* node classes. Alternation classes are used for abstract language constructs whereas the construction classes are used for concrete language constructs with a specific structure of son nodes. I.e., alternation classes correspond to non-terminals and construction classes to productions. More general node classes are also modeled by alternation classes.

The following syntax is used:

```
<alt-node-decl> ::=
    <alt-class-id> ‘:’ ‘alt’ [<alt-superclass-id>]
    [ ‘{’ <decl-body> ‘}’]

<cons-node-decl> ::=
    <cons-class-id> ‘:’ ‘cons’ [<alt-superclass-id>]
    ‘(’ (<son-decl>, ‘,’)* ‘)’
    [ ‘{’ <decl-body> ‘}’]

<son-decl> ::= <son-id> ‘:’ ‘ref’ <node-class-id>
```

For example:

```
Stmt: alt;
WhileStmt: cons Stmt(cond: ref Exp, doPart: ref Stmt);
```

The optional superclass must always be an alternation class. If no superclass is explicitly given, the class is considered to be a subclass of a most general node class `ANYNODE`. The class `ANYNODE` is in turn considered a subclass of `ANYCLASS`.

The declarations of references to son nodes in a construction class corresponds to the “right-hand-side” of a production. In a legal syntax tree, these references will denote real objects (i.e., they will not be `NONE`).

The node classes may, like ordinary classes, have a body of declarations. This will be used for defining attributions and evaluation of attributions.

6.2.2 Lexemes

Lexemes, such as identifiers and literal values, can *in principle* be defined using alternation and construction classes, by including the lexical definitions in the context-free syntax. However, that would be highly impractical and we therefore introduce a special kind of node class for lexemes:

```
<lexeme-node-decl> ::=
    <lexeme-class-id> ‘:’ ‘lex’
    ‘(’ <attr-id> ‘:’ <type> ‘)’
```

A lexeme class cannot be declared as subclass to any other class, but is considered to be a subclass of the most general node class `ANYNODE`. Further, a lexeme class does not have a body. It is considered a purely lexical entity.

In the rest of this thesis we will use the following two predefined lexeme classes for handling identifiers and integer constants:

```
ID: lex (ident: string);
INT: lex (val: integer);
```

6.2.3 Lists

Lists can, in principle, also be defined using alternation and construction class. However, in practical systems it is useful to have special list constructs, modeling a node with a variable number of sons of the same class. In OOSL this is done by introducing a list node class:

```
<list-node-decl> ::=
   <list-class-id> ‘:’ ‘list’ <alt-superclass-id>
   ‘(’ <son-id> ‘:’ ‘ref’ <node-class-id> ‘)’
```

For example:

```
StmtList: list (s: ref Stmt)
```

Here, a `StmtList` node has a variable number of sons of class `Stmt`.

In the examples in this thesis we will use lexeme and list classes whenever this is handy. However, in the algorithms and discussions we will often, for simplicity, ignore the lexeme and list classes, and treat only the basic node classes (alternations and constructions). Extending the algorithms to handle also lexeme and list classes is trivial.

6.2.4 Completing Classes

In structure-oriented programming environments it is necessary to handle incomplete syntax trees. Usually, this is done by extending the grammar by so called *completing* productions, i.e. one nullary production for each non-terminal. This way, syntax trees considered incomplete by the user can be considered complete by the system. The same technique can be used in object-oriented CFGs, i.e., extending the set of node classes such that for an alternation class `A`, appearing on the right-hand side of a construction class, there is a completing nullary construction class “`NullA:` **`cons`** `A( )`”.

It would be possible to let the class `A` itself act as the completing class and thus also allow instances of alternation classes in the syntax tree. This would be attractive from a code-sharing view. However, this may sometimes lead to conflicts between defining the behavior of the class in its role as completing class and the behavior of the class in its role as superclass of other classes. An example of this conflict is given in §6.5.2. Henceforth, we will therefore use explicit nullary construction classes as completing classes. By convention, these classes will be named `NullA` for an alternation class `A`. If no such class is explicitly given in the examples, we will assume there is such a class definition elsewhere.

The use of nullary construction classes as completing classes means that only construction classes (and lexeme and list classes) will be used for generating syntax node objects. All alternation classes are abstract classes which are never instantiated (other than as part of a more specialized object).

6.3 Attributes and equations

An *object-oriented AG* is an object-oriented CFG where each node class is extended with attribute and equation declarations. Attributes are similar to variables in that they are oo-inherited to subclasses. Equations are similar to virtual procedures in that they can override other equations. An equation in a class *C*, overrides equations in superclasses of *C* defining the same attribute.

In traditional AGs, the attributes are declared in non-terminals and the equations in productions. In object-oriented AGs, attributes and equations can be declared in both alternation and construction classes. This gives new interesting possibilities. In particular, it allows general behavior to be defined in general classes and to be overridden by specialized behavior in subclasses. This is very useful in many cases, as discussed in more detail in §6.5.

6.3.1 Attributes

Attribute declarations are similar to variable declarations, but preceded with a keyword to indicate if the attribute is inherited, synthesized, or local. A local attribute means here an attribute which must be defined in the node itself, just like a synthesized attribute, but which may not be accessed by the father node. The distinction between inherited, synthesized, and local attributes could, in principle, be derived from the ways the equations use the attributes, but we prefer this to be explicit in the declarations.

```
<attr-decl> ::=
   ( ‘inh’ | ‘syn’ | ‘loc’ ) <attr-id> ‘:’ <type>
```

For example:

```
inh a1: ref B;
syn a2: ref C;
loc a3: boolean;
```

Note that it is possible to define attributes with reference types. However, in a standard object-oriented AG, all references must denote objects representing regular values, i.e. the classes used for reference types must be applicative classes.

6.3.2 Equations

An equation defines the value of a synthesized or local attribute in the node itself, or an inherited attribute of one of the son nodes. Equations have the form of assignment statements, but are preceded by a keyword “**eq**” to indicate the difference. As usual for AGs, the order of the equations is completely irrelevant.

```
<eq-decl> ::= ‘eq’ <attr> (‘:=’ | ‘:-’) <exp>
<attr> ::= <attr-id> | <son-id> ‘.’ <attr-id>
```

For example:

```
eq s.a1 :- a1;
eq a3 := true;
```

6.3.3 Collective equations

We introduce *collective equations* to make better use of the possibilities in object-oriented AGs to define general behavior. A collective equation defines an inherited attribute for all sons of a given class. Collective equations are used in alternation classes where the exact number and types of sons are not known:

```
<coll-eq-decl> ::=
    ‘eq’ ‘son’ <node-class-id> ‘.’ <attr> (‘:=’ | ‘:-’) <exp>
```

For example,

```
eq son C.x := true;
```

Here, “**son** `C`” denotes any son node declared to be of the node class `C` (or a subclass of `C`). The equation defines the attribute `x` of these son nodes. Collective equations make it possible to define general behavior for propagating information downwards in the syntax tree. Suppose the above collective equation appears in an alternation class `A` and that `A` has a construction subclass:

```
B: cons A (t1: ref D1, t2: ref D2)
```

where `D1` and `D2` are subclasses to `C`. The collective equation is in this case equivalent to two ordinary equations in `B`:

```
eq t1.x := true
eq t2.x := true
```

6.4 An example: Desk calculator

As an example of a traditional AG expressed in OOSL, we will use a slightly simplified and adapted variant of the desk calculator example in [RT84].

The calculator grammar describes integer expressions, using integer constants, let-expressions, and normal arithmetic operators such as addition and subtraction. To handle the identifiers introduced by let-expressions, symbol tables are introduced to keep track of identifiers and associate them to their values. The symbol table type is defined in OOSL by using an applicative class with the following interface:

```
Table: class (* applicative *)
{ add: func ref Table(id: string, val: integer);
  lookup: func integer(id: string);
  found: func boolean(id: string);
};

emptyTab: object Table;
```

Figure 6.1 Interface to data type for symbol tables

A call “`tab.add(id, val)`” returns a reference to a new `Table` object which includes the association pair (`id`, `val`) in addition to the contents of `tab`. The function `lookup` returns the value associated with the `id` (or 0 if the `id` is not in the table). The `found` function returns true if the `id` is in the table. The object `emptyTab` is an empty table.

The desk calculator grammar can now be defined as shown in Figure 6.1. The specification is very similar to the specification in Reps’ paper. One notable difference is the use of the collective equation in `Exp`. This equation defines that the `env` attribute of an expression is propagated to all of its expression son nodes. This default behavior applies to most expressions. In a traditional notation, each normal expression would need one equation stating this behavior for each son node. For example, the `Sum` expression would need two equations defining the `env` attributes of its two sons. In the object-oriented AG such repeated equations are not needed, the default behavior is specified once and for all in the `Exp` class. The `Let` expression is the only class which needs to override this behavior. It overrides the default behavior by the equation “**`eq`** `inExp.env :- ...`”. Note that the default behavior still applies to the other son (`defExp`).

```
Calc: cons (e: ref Exp)
{ loc result: integer;
  eq e.env :- emptyTab;
  eq result := e.val;
};

Exp: alt
{ inh env: ref Table;
  syn val: integer;
  eq son Exp.env :- env;
};

NullExp: cons Exp ( )
{ eq val := 0;
};

Sum: cons Exp
  (lop: ref Exp, rop: ref Exp)
{ eq val := lop.val + rop.val;
};

Diff: cons Exp
  (lop: ref Exp, rop: ref Exp)
{ eq val := lop.val - rop.val;
};
```

```
Let: cons Exp
  (letId: ref ID,
  defExp: ref Exp,
  inExp: ref Exp)
{ eq val := inExp.val;
  eq inExp.env :-
    env.add(letId.ident,
            defExp.val);
};

Use: cons Exp (useId: ref ID)
{ loc error: boolean;
  eq val :=
      env.lookup(useId.ident);
  eq error := not
      env.found(useId.ident);
};

Const: cons Exp
  (constInt: ref INT)
{ eq val := constInt.val
};
```

Figure 6.2 OOSL specification of a desk calculator

6.5 Defining general behavior

One of the main benefits of object-oriented description techniques is that the classification hierarchy allows properties of objects to be described at suitable levels of generalization. In a traditional AG, all attributes are declared at the level of non-terminals, and all equations at the level of productions. In contrast, the object-oriented AGs allows a class hierarchy with an arbitrary number of levels, where attributes and equations can be declared at any level of generalization. In this section we will give a couple of examples of how this can be utilized.

The general behavior is often defined in general alternation classes which are motivated only from the static semantics and not from the context-free syntax. Thus, they never appear on the right-hand side of construction classes. We refer to such classes as *behavior classes*.

6.5.1 Propagation of environment information

Standard attribute grammars are usually cluttered with copy equations, most of which simply serve the role of propagating information from one place in the tree to another. A typical example is the propagation of declarative information throughout the tree. Object-oriented grammars give a possibility of describing such behavior on a general level, rather than specifically for each single language construct.

For example, in the desk calculator grammar, the collective equation in the Exp class defines the general propagation behavior of the env attribute. In a standard AG each construction class would instead need one copy equation for each Exp son node. The desk calculator grammar is, however, very simple in that it includes only one alternation class, Exp. This made it possible to define the general behavior directly in Exp. In more complex grammars behavior may need to be described at a more general level. This can be done by introducing behavior classes which are superclasses of existing alternation classes. For the few language constructs which do not follow the general behavior, the behavior can be overridden in the corresponding subclass, just as in the desk calculator grammar.

Consider specification of name analysis for a block-structured language (in the traditional AG style). Similar to the desk calculator grammar, an inherited attribute env represents the declarative environment. The env attribute is propagated throughout the syntax tree in order to reach all name applications. The name applications use the env attribute to look up the corresponding declaration and associated type information. Some syntax nodes affect the declarative environment, but most nodes simply pass the env attribute to all the son nodes by copy equations. In an object-oriented AG this general behavior can be described as shown in the example below. The behavior class Node models syntax nodes in general and is therefore the superclass of all other node classes in the grammar (excluding the lexeme classes). Node is specialized into the behavior classes Root and Descendant. Root models the possible root nodes and can be seen as a generalization of start non-terminals. Descendant models all other nodes, i.e. all nodes which have a father node in the syntax tree. All Descendant nodes have an inherited env attribute (1). The general behavior is to propagate the same env value to all son nodes. This is described by the collective equation (2). A root node does not have any env attribute, but propagates the value of an empty environment to its son nodes (3).

```
Node: alt;

Descendant: alt Node
{   inh env: ref Environment;              (* 1 *)
    eq son Descendant.env :- env;          (* 2 *)
};

Root: alt Node
{   eq son Descendant.env :- emptyEnv;     (* 3 *)
};
```

This general behavior is suitable for most language constructs, e.g. while-statements, if-statements, arithmetic expressions, etc. In a standard AG each such construct would need to explicitly define one copy equation for each son node. In the object-oriented AG above, the collective equation (2) gives the same result.

For language constructs which do alter the declarative environment, explicit equations need to be given. E.g., the Algol-like block statement below defines a new declarative environment in terms of the enclosing environment and the declarations local to the block. The equation (4) overrides the collective equation (2) for the `StmtList` son node. However, the collective equation still applies to the `DeclList` son node.

```
BeginBlock: cons Descendant
    (dl: ref DeclList, sl: ref StmtList)
{   eq sl.env :- f(env, dl.localenv);      (* 4 *)
};
```

Similarly, the collective equation can be overridden in other language constructs which change the declarative environment, e.g. procedures, classes, remote access expressions, and inspect-statements. However, for the great majority of language constructs, the general behavior applies.

6.5.2 Left-values of actual out-parameters

As an example of general behavior of a synthesized attribute we consider the use of out-parameters. Out-parameters are assigned a value inside a procedure and the actual parameter in a call must therefore have a *left-value* (i.e. it must denote a location which can be assigned a value). This is easy to check as in equation (1) below, by letting each expression have a synthesized attribute `hasLeftValue`.

```
ActualParam: cons Descendant (e: ref Exp);
{   inh isOutParam: boolean;
    eq leftValError: boolean;
    eq leftValError :=                     (* 1 *)
        isOutParam and not e.hasLeftValue;
};
```

Most expressions do not have a left-value. For example, none of the many relations and arithmetic expressions have a left-value. The general behavior of expressions can in this case be defined in the Expr alternation class as follows:

```
Exp: alt Descendant;
{   syn hasLeftValue: boolean;
    eq hasLeftValue:= false;               (* 2 *)
};
```

One of the few expressions which do have a left-value is name applications denoting variables. The class for name applications thus needs an equation which overrides (2), e.g. as below in (3).

```
Use: cons Exp (useId: ref ID);
{   ...
    eq hasLeftValue:= f(env, useId.ident); (* 3 *)
};
```

Here `f` should be a function which returns `true` if the name application denotes a variable, another out-parameter, or another assignable entity.

It can be noted here that syntactically incomplete expressions should be defined as having left-values. Otherwise, an incomplete expression at the place of an actual out-parameter would result in an irrelevant error message. The completing class for expressions therefore needs to override the equation (2):

```
NullExp: cons Exp ()
{   eq hasLeftValue:= true;                (* 4 *)
};
```

We have here an example of when the behavior of a completing class is different from the behavior of its superclass, thus motivating the use of explicit completing classes as discussed in §6.2.4.

6.6 Local attributes

Local attributes are like synthesized attributes in that they are defined in the node itself. They differ from synthesized attributes in that they are not allowed to be accessed by equations of the father node. The explicit distinction between synthesized and local attributes will be utilized in the construction of attribute evaluators as explained in §7.3.4.

It should be noted we use the term “local” in a slightly different sense than is done in SSL, the specification language of the Cornell Synthesizer Generator. In SSL, “local” means local to a production. Such local attributes were introduced in SSL to allow nodes of individual productions to have individual attributes, which is not possible in the AGs of Knuth. An SSL local attribute corresponds, in an object-oriented AG, to declaring a local or synthesized attribute in a construction class. The locality in the sense of SSL can thus be achieved with synthesized attributes in object-oriented AGs. Locality in object-oriented AGs means protection from access from other nodes, and this can apply to attributes declared at any level of generalization. Thus, attributes in alternation classes can also be declared local, although this is less common than in construction classes.

A typical use of local attributes is in the modelling of static-semantic errors, like the attribute `error` in the `Use` class in Figure 6.1.

Another use of local attributes is in connection with demand attributes. In the example in §6.5.1, most of the `env` attributes are defined by the collective copy equation. Most of these attributes are not particularly interesting to store since they can be computed very easily when needed. One might therefore want to implement the `env` attribute as a demand attribute. For the few cases where the copy equation is overridden and a new environment value is computed in a non-trivial manner, a local attribute can be added to store the value. E.g., the definition of the `BeginBlock` could be changed to:

```
BeginBlock: cons Descendant
    (dl: ref DeclList, sl: ref StmtList)
{   storedEnv: ref Environment;
    eq storedEnv :- f(env, dl.localenv)
    eq sl.env :- storedEnv;
};
```

A third use of local attributes is for specifying information to be accessed by external tools such as code generators or facilities in the user interface. This is discussed more in §7.4.1.

6.7 Well-formed object-oriented AGs

An attribute grammar is well-formed if, for each possible syntax tree, each attribute instance has exactly one defining equation. In deciding if an object-oriented AG is well-formed, similar conditions are used as for standard AGs. I.e., classes used for generating root nodes must not have any inherited attributes, and each construction class must have defining equations for all its synthesized and local attributes and for all the inherited attributes of all its son nodes.

In deciding well-formedness, there is one particular difficulty which occurs for object-oriented AGs which does not occur for standard AGs. The problem lies in deciding what inherited attributes a son node has. Because of oo-inheritance, it is possible to construct grammars where this cannot be determined statically. We will rule out such grammars by adding a special rule.

Consider a construction class C with son nodes

t_1 : **ref** X_1
...
t_n : **ref** X_n

The actual qualifications of the son nodes $t_1 \,..\, t_n$ are not known statically. An actual son node t_k may be of a more specialized class Z which is a subclass to X_k. Suppose Z declares an inherited attribute. This would imply that the set of equations needed in C is not statically known, but would have to be dynamically adapted, depending on the actual qualification of t_k. Although this would be possible in principle, it would be highly impractical. We therefore put the following additional requirement on object-oriented AGs:

6-1 **Condition** Additional condition for well-formedness of object-oriented attribute grammars.

Let `A` be a node class. If there exists a superclass `S` ⊃ `A` which is used in declaring a son node reference “`t:` **`ref`** `S`” for some construction class, then `A` must not declare any inherited attributes.

end 6-1

This condition guarantees that the set of equations necessary in each construction class is statically known. In standard AGs this condition is automatically fulfilled since productions (corresponding to subclasses) cannot declare inherited attributes.

The condition does not cause any practical problems when designing a grammar. If one finds that an inherited attribute a is needed for a class Z which is subclass of a class X occurring on the right hand side of a construction class, one simply has to declare a in X instead of in Z. This will have the effect that all other subclasses of X will also have this attribute although they will not use it.

6.8 Possible extensions

There are many ways to define object-oriented grammars. We will now discuss a couple of possible extensions to the object-oriented AGs defined in this thesis.

6.8.1 Subclasses of construction classes

In our definition of object-oriented AGs, construction classes cannot have subclasses. However, such specialization could be useful in some situations. Consider binary arithmetic operators. The general properties of such expressions could be described in a construction class as follows:

```
BinaryArithExpr: cons Expr
    (leftOp: ref Expr, rightOp:ref Expr)
{   typesOK: boolean;
    eq tp :=
        if leftOp.tp = RealType or rightOp.tp = RealType
        then RealType
        else IntegerType;
    eq typesOK :=
        isArithmetic(leftOp.tp) and isArithmetic(rightOp.tp);
}
```

It would be useful to specialize this class into subclasses `Add`, `Sub`, `Mul`, and `Div`. This way, the type checking could be described at the general level in `BinaryArithExpr` and would not have to be repeated in each of the specialized classes. It would be straight-forward to extend our formalism to allow such specialization of construction classes, but for simplicity we have not included this possibility.

6.8.2 Multiple inheritance

Our object-oriented AG definition is based on single inheritance. We have not found any need for multiple inheritance grammars in practice. Nevertheless, the question naturally arises of what the consequences would be of extending the formalism to allow multiple inheritance. As in all other object-oriented systems, multiple inheritance leads to name clash problems and the need for more complex implementation techniques. For object-oriented grammars, there is an additional difficulty which arises.

Consider the following fragment of a well-formed grammar.

```
A: alt;

B: alt {inh x: integer;};

C: cons (s: ref A);
```

Suppose we would like to introduce a new class `D` which is a subclass of both `A` and `B`. This class would naturally have access to the `x` attribute which is oo-inherited from `B` and could use this attribute to define a local attribute as follows:

```
D: cons A, B ()
{   loc y: integer;
    eq y := f(x);
};
```

However, adding this class has the consequence that the `s` son of a `C` node can actually be a `D` node (since `D` is a subclass of `A`). In such a tree, there would be no defining equation for the inherited attribute `x` of the son node since `C` does not contain such an equation. The grammar is thus no longer well-formed.

This problem is similar in nature to the problem mentioned in §6.7. The class `D` could actually be viewed as violating the condition 6-1: Suppose `D` is initially a subclass only of `A`. Declaring `D` as a subclass also of `B` implicitly adds the inherited attribute x to `D`. This violates the condition 6-1 since `A` (which is a superclass of `D`) is used for declaring a son node (the son “`s: `**`ref`**` A`” in class `C`).

Although the problem is similar to the one mentioned earlier for single-inheritance, there is in this case no way of moving the declaration of the problematic `x` attribute in order to make the grammar well-formed. The problem has to be solved by changing the class hierarchy, for example by adding a mutual superclass `E` to classes `A` and `B` and moving the declaration of `x` to `E`. This would imply that `C` must declare an equation defining the `x` attribute, since `A` nodes now also have this attribute.

6.9 Related approaches

The traditional nonterminal/production formalism for context-free grammars, and the BNF and extended BNF variants, are primarily parsing-oriented. Their main purpose is to describe the set of strings belonging to a language, in order to allow the strings to be recognized (parsed). In an interactive environment parsing is of secondary interest. It is only one of several possible tools in the editor of the environment. Most work in grammar-based programming environments instead use grammar formalisms which describe abstract syntax trees, emphasizing that each node in the syntax tree is a typed data object.

The Metal formalism [KLMM83] used in the Mentor project [DHKL84] is based on tree algebras, describing the grammar by *sorts* (also called phyla) and *operators*. The Synthesizer Generator [RT84] also uses the phylum/operator terminology and the systems developed within the Gandalf project [Not85] uses a similar class/operator terminology. A sort is equivalent to a nonterminal and an operator to a production. The main difference is that the operators are named, whereas productions are usually anonymous. Because the operators are named, it is natural to view them as node types. In the algebraic terminology, a sort is a *set* of operators and corresponds to a type union. In relation to object-oriented programming, subsets can be interpreted as subclassing. Further, if two sorts have a non-empty intersection and one is not a subset of the other, this corresponds to multiple inheritance. Metal allows such grammars, but since it handles only context-free syntax and not attributions, this does not cause the problems described in §6.8.2. The Synthesizer Generator, on the other hand, restricts all sorts to be disjoint, thus making the formalism equivalent to the nonterminal/production formalism, or a two-level single-inheritance scheme.

Other type-based approaches to context-free grammars include the GRAMPS system (GRAmmar-based MetaProgramming Scheme) [CI84]. This system makes use of four kinds of production rules: *construction* rules, *alternation* rules, *repetition* rules, and *lexical* rules. These rules are similar to usual BNF rules, but structured in such a way that each production rule defines a syntactic type. The GRAMPS formalism has inspired the similar use of classes in OOSL. GRAMPS is, however, not object-oriented and deals only with the context-free grammar and not attributions.

Explicitly object-oriented approaches to context-free grammars include the *hierarchical grammars* of Nørmark [Nør87] (inspired by the phylum/operator approach), the ASDL system [CNS87], [KS89], based on subtype specialization and variant record generalization, the *structured CFGs* by Madsen and Nørgaard [MN88](inspired by the GRAMPS approach), and the *nodeclass* approach of Tenma et al. [TTTI88] (inspired by the AND/OR rules of BNF). Koskimies also gives an object-oriented interpretation of BNF-style grammars in [Kos88], and defines *SI-structured* (Single Inheritance) and *MI-structured* (Multiple Inheritance) CFGs in [Kos91] which are object-oriented interpretations of traditional BNF-style grammars. Although some of these object-oriented approaches to context-free grammars make use of node attributes, they are all based on operational computation rather than AG-based definition of the attribute values.

The object-oriented approach to attribute grammars described in this thesis is based on earlier work by the author presented in [Hed88] and in more detail in [Hed89]. A similar approach was suggested later by Grosch [Gro90]. One difference is that Grosch does not distinguish between construction classes and alternation classes. Attributes, equations, and son references are all oo-inherited to subclasses. The motivation of Grosch for introducing object-oriented techniques is compactness of specification and compactness of the resulting syntax trees. OO-inheritance of attributes is mentioned, but not emphasized, and there is no mechanism for collective equations allowing general behavior to be described. The formalism is used in a compiler generator system for exhaustive evaluation.

6.10 Summary

This chapter has extended OOSL with constructs for specifying attribute grammars in an object-oriented manner. Four new kinds of classes were added: *alternations*, *constructions*, *lists*, and *lexemes*, together referred to as *node classes*. Notation for declaring attributes and equations was also added.

The object-oriented approach allows attributes and equations to be declared at any level in the class hierarchy, thereby allowing behavior to be defined at suitable levels of generalization. Some examples were given on how this can be utilized.

A discussion was also given on the problems of augmenting an object-oriented AG with multiple inheritance. It turns out that such a grammar is, in general, not well-formed.

Chapter 7
Attribute Evaluation Techniques

Object-oriented AGs are equivalent to standard AGs and evaluation of object-oriented AGs can therefore be done using standard algorithms. But by formulating the algorithms using object-oriented techniques, the implementation can in many cases be simplified. In this chapter, we will look at some evaluation techniques and show how they can be implemented in an object-oriented language. These techniques will be used as a basis for the algorithms introduced in Chapter 10, treating incremental evaluation of Door Attribute Grammars.

The evaluation algorithms we will treat in this chapter are the following:

- A demand-driven algorithm for general non-circular AGs
- An exhaustive data-driven algorithm for 1-visit AGs
- An incremental data-driven algorithm, also for 1-visit AGs.

We show how these algorithms can be programmed using object-oriented techniques. We also give a very simple technique for computing the visit sequences for 1-visit AGs. This can be done in a much simpler way than for the more general Ordered AGs. The incremental data-driven algorithm employs a new technique of *static* skipping of visit sequence instructions, rather than the usual dynamic skipping techniques. This makes it possible to avoid expensive attribute value comparisons needed for the dynamic skipping algorithms. Although this technique may give non-optimal evaluation for a standard AG, this non-optimality is normally irrelevant for Door AG evaluation.

We base the algorithms for incremental evaluation on the usual subtree replacement editing model where a subtree OLD is replaced by an unattributed subtree NEW. To restore consistency all three evaluation algorithms described are used: Exhaustive evaluation is performed on the new subtree, incremental evaluation is performed to propagate changes and re-evaluate affected attributes, and demand-driven evaluation is used for those attributes which are implemented as demand attributes rather than data attributes. In §7.4 the effects of combining data and demand evaluation are discussed in more detail.

7.1 Demand-driven evaluation

The principle of demand-driven evaluation is very simple; each attribute is represented by its semantic function. This principle can be applied to all non-circular attribute grammars, although it results in non-optimal evaluation. In Engelfriet's survey of evaluation methods [Eng84], the demand-driven algorithm is referred to as "P4". Engelfriet also gave an optimal version of this algorithm, employing "lazy" evaluation, referred to as "P5". The lazy demand-driven algorithm was first described by Jalili in [Jal85] and Jourdan gave a Lisp implementation of this algorithm in [Jou84].

In this section we present object-oriented versions of these algorithms. The resulting programs are remarkably simple because of the use of virtual functions. Synthesized attributes can be mapped directly to virtual functions. Inherited attributes can be implemented by virtual functions in the father node, but since the father node may have many sons of the same class, an extra parameter is needed in this function to let the father node decide which of the equations to apply.

7-1 **Construction** Demand-driven evaluator

Given an object-oriented AG in OOSL, a demand-driven evaluator, equivalent to Engelfriet's "P4" evaluator, can be implemented in OOSL as follows.

- *Reference to father.* Each node is equipped with a reference to its father node by adding a reference variable to the most general node class `ANYNODE`:
```
addto ANYNODE
{   father: ref ANYNODE; (* NONE for the root node *)
};
```
  The `father` reference denotes the father node in the syntax tree. The father of the root node is `NONE`.

- *Synthesized attributes*. A declaration of a synthesized attribute `aSyn` of type `T` in a class `C` is replaced by a virtual function specification in class `C`:
```
addto C
{   aSyn: func T;
};
```
- *Equations for synthesized attributes*. An equation defining a synthesized attribute `aSyn` as the expression `e` in a class `C` is replaced by an implementation of the virtual function `aSyn`:
```
addto C
{   impl aSyn := e;
};
```

- *Local attributes.* Declarations and equations for local attributes are implemented in the same way as synthesized attributes.

- *Inherited attributes*. A declaration of an inherited attribute `aInh` of type `T` in a class `C` is replaced by a function specification and implementation in class `C`:
```
addto C
{   aInh: func T := father.C_aInh(this ANYNODE);
};
```

Note that the remote access "`father.C_aInh...`" can be safely done since `father` will only be `NONE` for the root node, and the root node has no inherited attributes.

The function `C_aInh` is declared as a virtual function specification in class `ANYNODE`:

```
addto ANYNODE
{   C_aInh: func T(s: ref ANYNODE);
};
```

- *Collective equations*. A collective equation "`son C.aInh := e`" in an alternation class `D` is replaced by an implementation in class `D` of the virtual function `C_aInh`:

```
addto D
{   impl C_aInh := e;
};
```

- *Equations for inherited attributes*. Consider a construction class `D` which declares equations defining the attribute `aInh` of son nodes of class `C`. These equations are replaced by an implementation of the virtual function `C_aInh`. The parameter `s` in the specification of this function is used to do case analysis on which equation to apply. If no equation declared in `D` applies, the defining equation must be in the superclass, and in this case **`super`** is called.

  For example, suppose `D` has four sons of class `C`: `t1`, `t2`, `t3`, and `t4`, and two equations:

```
    eq t1.aInh := e1;
    eq t2.aInh := e2;
```

  In this case, these equations are replaced by the following function implementation:

```
addto D
{   impl C_aInh :=
        if s == t1
        then e1
        else   if s == t2
               then e2
               else super.C_aInh(s);
};
```

end 7-1

7.1.1 An example

The example below shows the resulting demand-evaluator for the desk calculator example of §6.4. Note how similar the implementation is to the original grammar.

```
addto ANYNODE
{ father: ref ANYNODE;
  (* NONE for the root node *)

  Exp_env: func ref Table
    (s: ref ANYNODE);
};

addto Calc
{ result: func integer;
  impl Exp_env :- emptyTab;
  impl result := e.val;
};

addto Exp
{ env: func ref Table:-
    father.Exp_env
      (this ANYNODE);
  val: func integer;
  impl Exp_env :- env;
};

addto NullExp
{ impl val := 0;
};

addto Sum
{ impl val :=
    lop.val + rop.val;
};
```

```
addto Diff
{ impl val :=
    lop.val - rop.val;;
};

addto Let
{ impl val := inExp.val;
  impl Exp_Env :-
    if s == inExp
    then
      env.add(letId.ident,
              defExp.val)
    else
      super.Exp_env(s);
};

addto Use
{ error: func boolean;
  impl val :=
    env.lookup(useId.ident);
  impl error := not
    env.found(useId.ident)
};

addto Const
{ impl val := constInt.val;
};
```

Figure 7.1 Demand-driven evaluator for desk calculator

7.1.2 Lazy evaluation

A lazy evaluator works like a demand evaluator, but stores the attribute value the first time the attribute is accessed. At subsequent accesses, the stored value is returned directly, instead of applying the semantic function. This makes the algorithm optimal. A mark bit for each attribute is used to check if the attribute has been evaluated previously. In addition, another mark bit can be used to check for circularity at evaluation time. This functionality is easily incorporated into the object-oriented evaluator as shown below. However, in the rest of this thesis, we will not make use of lazy evaluation.

7-2 **Construction** Lazy evaluator

A lazy evaluator, equivalent to Engelfriet's "P5" evaluator, and including the circularity check of Jalilis evaluator [Jal85] can be constructed by modifying the demand-driven evaluator 7-1 as follows:

- *Additional instance variables*. For each attribute declaration `a` in a class `C`, add a declaration of a variable `a_value` which will be used to store the attribute value. Also introduce a boolean variable `computed_a` which is true if `a` is stored in `a_value`, and a boolean variable `computing_a` which is true if the value of `a` is under computation. Initially (before evaluation), all the boolean variables are false:

```
addto C
{   a_value: T;
    computed_a: boolean;
    computing_a: boolean;
};
```

- *Attribute procedures*. In 7-1 each attribute `a: T` declared in a class `C` was implemented by a function in `C`

```
a: func T := e;
```

  where `e` was the right hand side of the equation defining `a` (in case of a synthesized or local attribute) or a call to a function in the father node (in case of an inherited attribute). In the lazy evaluator, this function is replaced by a procedure implemented as follows.

```
addto C
{   a: proc T;
    {   if computing_a then
            error("Circularity in grammar");
        if computed_a then
            a := a_value
        else
            computing_a := true;
            a_value := e;
            computing_a := false;
            a := a_value;
        end if;
    };
};
```

end 7-2

7.2 Exhaustive 1-visit evaluation

We now turn to data-driven evaluation techniques. A simplified form of Kastens' algorithms for Ordered AGs (OAGs) [Kas80] can be used for 1-visit grammars. For these grammars, the visit sequences will have only one segment. In addition, the computation of visit sequences can be substantially simplified, compared to the more general OAG case. In this section we show how an exhaustive 1-visit evaluator based on these techniques can be implemented in OOSL.

7.2.1 Total attribute and equation sets

Whereas implementation of demand-driven evaluators could be done more or less directly by replacing attribute and equation declarations, the implementation of data-driven evalutors requires that we consider the *total* set of attributes and equations of a node, taking oo-inheritance and overriding into account. We therefore define $A(\mathtt{C})$ as the total set of attributes that a `C` node has, and $E(\mathtt{C})$ as the total set of equations applying to a `C` node. We say a class `C` *declares* an attribute or an equation if the attribute declaration or equation appears in the declaration of class `C` rather than in the declaration of any of the superclasses $\{\mathtt{S} \mid \mathtt{S} \supset \mathtt{C}\}$. Further, we say that a class `C` *has* an attribute or equation, if it is a member of $A(\mathtt{C})$ or $E(\mathtt{C})$ respectively. These sets are defined as follows:

7-3 Definition Total set of attributes

The total set of attributes $A(\mathtt{C})$ of a node class `C` is the union of all attributes declared in any of the classes $\{\mathtt{X} \mid \mathtt{X} \supseteq \mathtt{C}\}$.

end 7-3

7-4 Definition Total set of equations

The total set of equations $E(\mathtt{C})$ of a node class `C` is the union of all equations declared in any of the classes $\{\mathtt{X} \mid \mathtt{X} \supseteq \mathtt{C}\}$, subject to overriding and replacement of collective equations as follows:

- Let e be an ordinary equation "**`eq`** `a := ...`" declared in a class $\mathtt{D} \supseteq \mathtt{C}$. The equation e is a member of $E(\mathtt{C})$ if it is not overridden in `C`, i.e., if there is no other equation e' defining `a` and which is declared in a class `D′` such that $\mathtt{D} \supset \mathtt{D'} \supseteq \mathtt{C}$.
- Let e be a collective equation "**`eq son`** `S.a :=...`" declared in a class $\mathtt{D} \supseteq \mathtt{C}$. If `C` is an alternation class, e does not give rise to any equations in $E(\mathtt{C})$. If `C` is a construction class with son nodes

```
t_1 : ref X_1
...
t_n : ref X_n
```

then for each son node $\mathtt{t}_k$ such that $\mathtt{S} \supseteq \mathtt{X}_k$, we construct a corresponding ordinary equation e' "**`eq`** $\mathtt{t}_k$`.a := ...`". The equation e' is a member of $E(\mathtt{C})$ if it is not overridden, i.e., if there is no equation e'' which also defines $\mathtt{t}_k\mathtt{.a}$ and which is declared in a class `D′` such that $\mathtt{D} \supset \mathtt{D'} \supseteq \mathtt{C}$.

end 7-4

7.2.2 Dependency graphs

Visit sequences are computed by approximating the dependency graphs of all possible syntax trees by dependency graphs for the productions (construction classes) in the grammar and then doing a topological sort on these graphs. The construction of the production dependency graphs is rather complicated for OAGs. For 1-visit grammars considerable simplifications are possible.

1-visit grammars have the property that an inherited attribute in a node is never dependent on a synthesized attribute in the same node. Otherwise, the node would have to be visited twice: one visit to calculate the synthesized attribute, then an intermediate visit to the father node to calculate the inherited attribute, then a second visit to use the inherited attribute. When constructing the dependency graph for a construction class C with son nodes

t_1 : **ref** X_1
...
t_n : **ref** X_n

we can assume that for each of the son nodes t_k, all synthesized attributes of t_k depend on all the inherited attributes of t_k. Because of the above mentioned property, this cannot result in cyclic dependency graphs. Therefore, to construct the dependency graph for a construction class, no transitive dependency analysis is necessary. It is sufficient to analyze the equations in the construction class itself.

7-5 **Construction** Dependency graph of construction class.

Given a construction class C with son nodes

t_1 : **ref** X_1
...
t_n : **ref** X_n

we construct its dependency graph $DG(C)$.

Vertices.
$DG(C)$ has vertices $\{v(\texttt{inh}), v(\texttt{syn}), v(t_1) .. v(t_n), v(a_1) .. v(a_m)\}$ where

- $v(\texttt{inh})$ represents the inherited attributes of C
- $v(\texttt{syn})$ represents the synthesized attributes of C
- $v(t_j)$ represents the attributes of the son node t_j
- $v(a_k)$ represents the attribute a_k defined by the k'th equation in $E(C)$.

Edges.
An equation "**eq** a :=. . b . ." in $E(C)$, gives rise to the following edges:

- If b is defined by another equation in $E(C)$, an edge $(v(b), v(a))$.
- If b is a synthesized attribute of son t_j, an edge $(v(t_j), v(a))$.
- If a is an inherited attribute of son t_j, an edge $(v(a), v(t_j))$.
- If a is a synthesized attribute in $A(C)$, an edge $(v(a), v(\texttt{syn}))$.

- If b is an inherited attribute in $A(C)$, an edge ($v($**`inh`**$)$, $v(a)$).

end 7-5

If all resulting dependency graphs are acyclic, the grammar is 1-visit.

The figure below shows the dependency graphs for the construction classes of the desk calculator grammar.

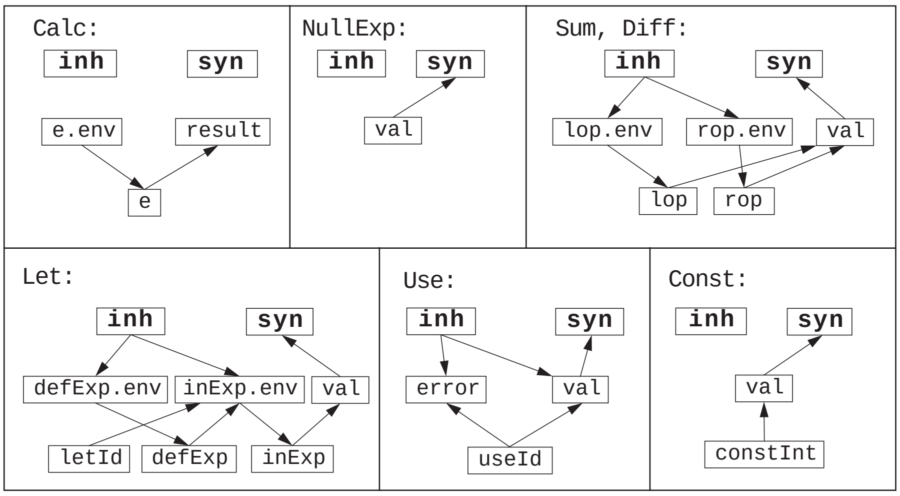


Figure 7.2 Dependency graphs for desk calculator example

7.2.3 Exhaustive Visit Sequences

For an Ordered AG, a *visit sequence* is a sequence of EVAL, VISIT, and RETURN instructions (§4.3). We will refer to the visit sequences used in exhaustive evaluation as *exhaustive* visit sequences. As will be explained later, we will use slightly different visit sequences for the incremental evaluation. The exhaustive visit sequences contain instructions only of the following kinds:

EVAL *a*	meaning: evaluate attribute *a*
VISIT *t*	meaning: visit son node *t*

We do not use explicit RETURN instructions. For OAGs, RETURN instructions are used to mark the end of a segment in a sequence, but since visit sequences for 1-visit AGs consist of only one segment, this instruction is not needed. We can view all sequences as ending in an implicit RETURN instruction which returns control to the father node.

A visit sequence is obtained by sorting the vertices of a dependency graph into a *vertex sequence* and then translating the vertex sequence to a visit sequence as follows:

7-6 Construction Exhaustive visit sequences

Let Sort(G) be a function which sorts all the vertices of the graph G into a topologically ordered sequence. The exhaustive visit sequence of a construction class C is denoted by $EVS(C)$ and is constructed as follows

$$EVS(C) := \quad \text{Translate}(\text{Sort}(DG(C))$$

end 7-6

The function Translate is defined as follows:

7-7 Function Translate(S)

Given a vertex sequence S, the function Translate(S) produces a visit sequence as follows:

- A vertex $v(a)$ for an attribute a is replaced by a corresponding instruction EVAL a.
- A vertex $v(t)$ for a son node t, where t is a lexeme, is removed from the sequence.
- A vertex $v(t)$ for a son node t, where t is *not* a lexeme, is replaced by an instruction VISIT t.
- The v(**`inh`**) and v(**`syn`**) vertices are removed from the sequence.

end 7-7

Note that lexeme nodes do not need to be visited since they do not contain any equations. A vertex for a lexeme son node need therefore not result in any visit instruction.

Figure 7.3 shows the exhaustive visit sequences for the desk calculator example.

```
EVS(Calc)=
  ( EVAL e.env,
    VISIT e,
    EVAL result )

EVS(NullExp)=
  ( EVAL val )

EVS(Sum)= EVS(Diff)=
  ( EVAL lop.env,
    VISIT lop,
    EVAL rop.env,
    VISIT rop,
    EVAL val )
```

```
EVS(Let)=
  ( EVAL defExp.env,
    VISIT defExp,
    EVAL inExp.env,
    VISIT inExp,
    EVAL val )

EVS(Use)=
  ( EVAL error,
    EVAL val )

EVS(Const)=
  ( EVAL val )
```

Figure 7.3 Exhaustive visit sequences for desk calculator example

7.2.4 Construction of Exhaustive Evaluator

Given the exhaustive visit sequences it is very simple to program an exhaustive evaluator in OOSL:

7-8 **Construction** Exhaustive data-driven evaluator

Given an object oriented AG and the exhaustive visit sequences for its construction classes, an exhaustive data-driven evaluator can be implemented in OOSL as follows:

- *Evaluation procedure.* A virtual procedure `exhVisit` is declared in the general class `ANYNODE`:

```
addto ANYNODE
{   exhVisit: proc;
};
```

- *Attribute declarations*. An attribute declaration (synthesized, inherited, or local) is interpreted as variable declaration.
- *Visit sequences*. For each construction class, an implementation of the virtual procedure `exhVisit` is given. This implementation is a straight-forward translation from the exhaustive visit sequence for the construction class. Each EVAL instruction is implemented as an assignment statement (interpreting the equation as an assignment statement). Each VISIT instruction of a son s is implemented as a call

```
s.exhVisit
```

end 7-8

7.2.5 An example

The example below shows the resulting exhaustive data-driven evaluator for the desk calculator example. Given a syntax tree with root node `r`, the tree is evaluated by calling `r.exhVisit`.

```
addto ANYNODE
{ exhVisit: proc;
};

addto Calc
{ impl exhVisit
  { e.env :- emptyTab;
    e.exhVisit;
    result := e.val;
  };
};

addto NullExp
{ impl exhVisit
  { val := 0;
  };
};

addto Sum
{ impl exhVisit
  { lop.env :- env;
    lop.exhVisit;
    rop.env :- env;
    rop.exhVisit;
    val := lop.val + rop.val;
  };
};

addto Diff
{ impl exhVisit
  { lop.env :- env;
    lop.exhVisit;
    rop.env :- env;
    rop.exhVisit;
    val := lop.val - rop.val;
  };
};
```

```
addto Let
{ impl exhVisit
  { defExp.env :- env;
    defExp.exhVisit;
    inExp.env :-
      env.add(letId.ident,
              defExp.val);
    inExp.exhVisit;
    val := inExp.val;
  };
};


addto Use
{ impl exhVisit
  { error := not
      env.found(useId.ident);
    val :=
      env.lookup(useId.ident);
  };
};

addto Const
{ impl exhVisit
  { val := constInt.val;
  };
};
```

Figure 7.4 Exhaustive data-driven evaluator for desk calculator

7.3 Incremental 1-visit evaluation

We base incremental evaluation on the usual subtree replacement editing model. Let *T* be a consistently attributed syntax tree, *OLD* a subtree in *T*, and *NEW* an un-attributed syntax tree which may syntactically replace *OLD* in *T*. The incremental evaluation problem is to obtain a new consistent attribution after *OLD* has been replaced by *NEW*.

7.3.1 Standard Incremental OAG Evaluators

Yeh published an incremental algorithm for OAGs in [Yeh83]. The evaluator is very similar to the stack automaton exhaustive evaluator by Kastens [Kas80]. To make the technique incremental, three modifications were made:

- Before the incremental evaluation is started, a stack configuration is computed which corresponds to the first visit (in an exhaustive evaluation) of the root of the new subtree. Thus, after each subtree replacement, such a stack configuration has to be computed.
- Attributes to be evaluated are kept track of by a marking mechanism. At an EVAL instruction the attribute is evaluated only if it is marked. After evaluating an attribute its new value is compared with the old value, and if they differ, all successor attributes are marked. Thus, it is necessary to have dependency information available at evaluation time. This information can be computed statically, however.
- Techniques are employed for determining if certain visits can be skipped, or if the evaluation can be stopped altogether. These checks are done dynamically (at evaluation time) using information about which attributes have changed values.

Reps published a similar incremental algorithm for OAGs in his thesis [Rep84]. Another version of this algorithm, which uses less bookkeeping information, was published in [RT88]. In contrast to Yeh, Reps uses a finite automaton rather than a stack automaton. The finite automaton implementation uses explicit instructions to visit the father node instead of popping a stack. An advantage of this in the incremental setting is that no initial stack configuration needs to be computed.

The algorithm in [RT88] does dynamic visit-skipping in an elegant way. A boolean flag in each syntax node is used to determine if a visit to that node can be skipped or not. After evaluating a synthesized or inherited attribute, the new value is compared to the old one. If the values differ, the flag of the father node or the appropriate son node is set. In this way, an asymptotically optimal algorithm is achieved without having the extra overhead of marking individual attributes. However, markings of individual attributes is suggested as an optimization.

7.3.2 An evaluator based on incremental sequences

The standard incremental evaluation algorithms discussed in the previous section rely on comparing old and new attribute values in order to limit the evaluation propagation. Clearly, such value comparison is irrelevant for demand attributes since their values are not saved anywhere. In Door Attribute Grammars, which will be introduced in the next chapter, very many attributes are demand attributes. The standard algorithms are therefore unnecessarily complex in the context of Door AGs.

In this section we describe a new incremental evaluation algorithm for 1-visit AGs which is *not* based on attribute value comparisons. The algorithm instead limits evaluation propagation by using information determined *statically* from the grammar. The advantage of this approach is that it simplifies the evaluation substantially. If applied to a grammar with many data attributes, the algorithm will be non-optimal in general. It will not recognize when attribute values have converged, but may continue to evaluate arbitrarily many attributes which already have correct values. However, if the grammar is such that convergence is unlikely, this algorithm may perform well, even for a standard AG with only data attributes. For example, it will perform nearly optimal for the desk-calculator grammar. Actually, in this case it may well run faster than the standard algorithms since it does not spend any time on comparing attribute values. However, the main motivation for introducing this algorithm is not for running it on standard AGs, but to make use of it in the algorithms introduced in Chapter 10, for evaluation of Door AGs.

The idea is to do static dependency analysis to avoid unnecessary visits and re-evaluation of attributes. This is done by using a set of visit sequences for each construction class, one for each adjacent node. We refer to these visit sequences as *incremental* visit sequences since they are used in the incremental evaluator. An incremental visit sequence gives the sequence of instructions to perform when an adjacent node executes an instruction to visit the node in question. The sequence to execute when the control comes from the father node is called the *incremental father sequence*. The sequence to execute when the control comes from a son node is called an *incremental son sequence*. In addition to the EVAL and VISIT instructions the following instruction is also used:

VISITFATHER	meaning: visit the father node

This instruction is different from the implicit RETURN instruction ending all sequences. In incremental evaluation the control starts at the point of subtree replacement, rather than at the root as in exhaustive evaluation. To visit ancestor nodes of the replaced subtree, the explicit VISITFATHER instruction is needed. The implicit RETURN instruction at the end of the incremental father sequence means (as for the exhaustive sequence) return to the father node. For a son sequence executed at a visit from a son *t*, on the other hand, the implicit RETURN instruction means return to the son *t*.

In constructing an incremental son sequence, we assume that all the synthesized attributes of that son node have changed. In constructing an incremental father sequence, we assume that all the inherited attributes of the node have changed. This corresponds to approximating all the definitions in an adjacent node by one vertex in the dependency graph (a son vertex $v(t)$ or the $v(\texttt{inh})$ vertex) as was done in §7.2.2. The incremental visit sequences are constructed as follows:

7-9 Construction Incremental visit sequences

Let v belong to a dependency graph DG and let SortReachable(v) be a function which returns a topologically ordered sequence S of all vertices in DG which are reachable from v (excluding v itself).

For a construction class C with son nodes

t_1 : **`ref`** X_1
. . .
t_n : **`ref`** X_n

n incremental son sequences, denoted $IVS(C, t_1)$ . . $IVS(C, t_n)$, and one incremental father sequence, denoted $IVF(C)$, are constructed as follows:

```
IVF(C) :=          Translate(SortReachable(v(inh)))

IVS(C, t_k) :=     let S = SortReachable(t_k) in
                       let VS = Translate(S) in
                           if v(syn) ∈ S then
                               Append(VS, VISITFATHER)
                           else
                               VS
```

where Append(VS, $instr$) is a function which appends the instruction $instr$ at the end of the visit sequence VS.

end 7-9

Note that VISITFATHER instructions occur only at the end of incremental son sequences. The VISITFATHER instruction can, without loss of generality, be placed last rather than at the place of v(**`syn`**) since v(**`syn`**) has no outgoing edges.

```
IVF(Calc)=
  ( )

IVS(Calc, e)=
  ( EVAL result )

IVF(NullExp)=
  ( )

IVF(Sum)=
IVF(Diff)=
  ( EVAL lop.env,
    VISIT lop,
    EVAL rop.env,
    VISIT rop,
    EVAL val )
```

```
IVS(Sum, lopp)=
IVS(Sum, rop)=
IVS(Diff, lop)=
IVS(Diff, rop)=
  ( EVAL val,
    VISITFATHER )

IVF(Let)=
  ( EVAL defExp.env,
    VISIT defExp,
    EVAL inExp.env,
    VISIT inExp,
    EVAL val )

IVS(Let, letId)=
IVS(Let, defExp)=
  ( EVAL inExp.env,
    VISIT inExp,
    EVAL val,
    VISITFATHER )
```

```
IVS(Let, inExp)=
  ( EVAL val,
    VISITFATHER )

IVF(Use)=
  ( EVAL error,
    EVAL val )

IVS(Use, useId)=
  ( EVAL error,
    EVAL val,
    VISITFATHER )

IVF(Const)=
  ( )

IVS(Const, constInt)=
  ( EVAL val,
    VISITFATHER )
```

Figure 7.5 Incremental visit sequences for the desk calculator

Figure 7.5 shows the incremental visit sequences for the desk calculator example.

7.3.3 Construction of Incremental Evaluator

An incremental evaluator can be constructed in OOSL as follows.

7-10 **Construction** Incremental data-driven evaluator

Given an object-oriented AG and the incremental visit sequences for its construction classes, an incremental data-driven evaluator can be implemented in OOSL by extending the exhaustive evaluator of 7-8 as follows:

- *Evaluation procedures.* Two virtual procedures `incFatherVisit` and `incSonVisit` are specified in the general class `ANYNODE`. A reference to the father node is also declared (as for the demand-driven evaluator):

```
addto ANYNODE
{   father: ref ANYNODE; (* NONE for the root node *)
    incFatherVisit: proc;
    incSonVisit: proc(s: ref ANYNODE);
};
```

  The procedure `incFatherVisit` models a visit from the father node. The procedure `incSonVisit` models a visit from the son node `s`.

- *The father sequence*. For each construction class, an implementation of the virtual procedure `incFatherVisit` is given. This implementation is a straight-forward translation from the incremental father sequence for the construction class. Each EVAL instruction is implemented as a corresponding assignment statement. Each VISIT instruction to a son node `s` is implemented as a call

```
    s.incFatherVisit;
```

- *The incremental sequences*. For each construction class, an implementation of the virtual procedure `incSonVisit` is given. The parameter `s` is used to do case analysis on which incremental son sequence to apply. The EVAL and VISIT instructions are implemented in OOSL as in the father sequence case. Each VISITFATHER instruction is implemented as a call

```
    father.incSonVisit(this ANYNODE);
```

end 7-10

After a replacement of a subtree with root OLD by an un-attributed subtree with root NEW in a tree, attribute consistency is restored by the following procedure:

7-11 **Procedure** RestoreConsistency(OLD, NEW)

```
RestoreConsistency: proc
```

```
                (OLD: ref ANYNODE, NEW: ref ANYNODE);
            {   Copy the values of inherited attributes in OLD
                to the inherited attributes in NEW.
                NEW.exhVisit;
                NEW.father.incSonVisit(NEW);
            }
```

end 7-11

The example below shows the implementation of incremental data-driven evaluation for the construction class `Let` in the desk calculator example.

```
addto Let
{ impl incFatherVisit
  { defExp.env :- env;
    defExp.incFatherVisit;
    inExp.env :- env.add(letId.ident, defExp.val);
    inExp.incFatherVisit;
    val := inExp.val;
  };

  impl incSonVisit
  {
    if S == letId or S == defExp then
      inExp.env :- env.add(letId.ident, defExp.val);
      inExp.incFatherVisit;
      val := inExp.val;
      father.incSonVisit(this ANYNODE);
    else if S == inExp then
      val := inExp.val;
      father.incSonVisit(this ANYNODE);
    end if;
  };
}
```

Figure 7.6 Incremental data-driven evaluation for the construction class Let.

7.3.4 Static vs. dynamic skipping of instructions

The incremental algorithms of Yeh and Reps discussed in §7.3.1 use the exhaustive visit sequences also for incremental evaluation. At the first visit to a *C* node from its father node, the complete exhaustive sequence of *C* is executed. At the first visit from a son node *t* the execution is started after the first VISIT *t* instruction. For 1-visit grammars, the Yeh and Reps algorithms correspond to an incremental sequence algorithm using the following *tail sequences*:

TVF(*C*) := EVS(*C*)

TVS(*C*, *t*) := Append(MatchTail(EVS(*C*), VISIT *t*), VISITFATHER)

where the function MatchTail(*VS*, *instr*) returns the subsequence of *VS* starting with the instruction following *instr.* The incremental visit sequences are subsets of the corresponding tail sequences. If an instruction in a tail sequence does not appear in the corresponding incremental sequence we say the instruction has been *statically skipped*. The algorithms of Yeh and Reps instead rely on dynamic skipping, using markers on attributes and nodes.

If an EVAL instruction is skipped this saves the work of one attribute evaluation. Skipping a VISIT instruction will save re-evaluation in a whole subtree. Skipping a VISITFATHER instruction terminates the evaluation all together since the VISITFATHER instruction is always the last instruction in a son sequence.

There are two effects which may cause our algorithm based on static skipping to evaluate more attributes than an algorithm based on dynamic skipping:

- *Value convergence*. If re-evaluation of an attribute results in the same value as before, the static skipping algorithm does not recognize this, but will re-evaluate all successor attributes, both direct and transitive ones.
- *Parallel attributes*. Two attributes a and b of a node are said to be *parallel* if they are both synthesized or both inherited. In the dependency graph for a node class C, parallel attributes which are defined by a given neighbor node are approximated by the same vertex. I.e., all synthesized attributes of a son node t are represented by the vertex $v(t)$. Likewise, all the inherited attributes of C are represented by the vertex $v($**`inh`**$)$. Due to this approximation, the re-evaluation of an attribute a will cause all successors of an attribute b, where b is parallel to a, to be re-evaluated.

Our algorithm is primarily intended to be used as a part of an evaluator for Door Attribute Grammars, rather than as an evaluator for standard AGs. In Door AGs the above sources of sub-optimality have a very limited effect, as will be discussed in §10.8.4.

In comparing the incremental visit sequences to the tail sequences for the desk calculator grammar we can note that most of the father sequences are the same as the corresponding tail sequences (i.e. no instructions have been skipped) and that most of the son sequences have not skipped the VISITFATHER instruction. As will be discussed in §10.8.3, there will be a higher degree of skipping in Door AGs. The distinction between local and synthesized attributes is important for allowing skipping of VISITFATHER instructions. If the local attributes were treated as synthesized attributes this would lead to many cases of parallel attributes which could cause unnecessary evaluations.

7.4 Combining data and demand attributes

Incremental systems store data to allow fast updates. However, storage consumption can be a bottleneck in these systems so it is important to store only the most important information. One way to reduce the storage consumption in an AG system is to implement some of the attributes as demand attributes (i.e. functions) rather than as data attributes (i.e. stored values). To use demand attributes instead of data attributes is not always a time/space tradeoff. On one extreme, demand attributes may degrade performance if they are accessed often, thus using less space but more time. On the other extreme, demand attributes may save both time and space. For attributes which are seldom used, the cost of incrementally maintaining their values may be more than the cost of accessing them as demand attributes. Some attribute instances may even be in a syntactic context where they are not accessed at all. The grammar in §6.5.2 is an example of this: The `hasLeftValue` attribute is present in all `Exp` nodes, but only a few of these nodes will have a father node actually using this attribute.

The data and demand-driven evaluation approaches can be combined freely. Demand attributes are implemented as described in §7.1. In implementing the incremental data-driven evaluator, the demand attributes must be considered during dependency analysis, but when constructing the visit sequences, the evaluation instructions for these attributes are simply removed.

7.4.1 Externally accessed demand attributes

Given an incrementally maintained attribution, it can be useful to make this information available to programming environment tools like editors and debuggers. This can be done by letting the tools call virtual functions of individual nodes. For example, a context-sensitive editor might have use for asking if an expression can be replaced by an expression of type `t` without causing a type-checking error. This could be implemented by a boolean function `typeAcceptable(t)` defined for expressions. The function can be defined in terms of attributes, either existing ones or additional ones added specifically for this purpose, as follows:

```
Exp: alt
{   inh allowedTp: ref Type; (* external *)
    typeAcceptable: func boolean (t: ref Type)
        := t == allowedTp;
};

Add: cons Exp(lop: ref Exp, rop: ref Exp)
{   eq lop.allowedTp :- intType;
    eq rop.allowedTp :- intType;
};
```

The attribute `allowedTp` is an *external attribute*. With this we mean a demand attribute which implements a service needed by an external tool, but which is not used in the definition of any data attribute. An external attribute can be accessed only by other external attributes, by functions, and by external tools. The reason for distinguishing between external and ordinary attributes is that the external attributes need not be considered in dependency analysis since they have no dependent data attributes.

7.5 Summary

This chapter has described how some attribute evaluation algorithms can be implemented using object-oriented techniques. Three basic algorithms have been treated: demand-driven evaluation, exhaustive data-driven evaluation for 1-visit grammars, and incremental data-driven evaluation for 1-visit grammars. All of these are very simple to implement in an object-oriented language by making use of virtual functions and procedures. In the incremental algorithm a new technique for static skipping of instructions was introduced. This was accomplished by using several visit sequences for each node class instead of only one. Static skipping avoids value comparisons at evaluation time, but results, in general, in sub-optimal evaluation. This sub-optimality has little or no effect for grammars where value convergence is unlikely, and for grammars with many demand attributes. The primary motivation for the static skipping technique is that it is useful in the evaluation of Door Attribute Grammars as will be described in subsequent chapters.

Chapter 8
Door Attribute Grammars

In this chapter we introduce Door Attribute Grammars, an extension to standard AGs allowing declarative specification of attributions containing objects and references. This enables attributions which are suitable for incremental update to be specified. In particular, visibility graphs for object-oriented languages, such as those described in §3.3, can be defined explicitly in Door AGs. We show how Door AG specifications are written and give an example of specifying name analysis for a simple block structured language. More advanced examples for object-oriented languages are given in Chapter 11.

8.1 Introduction

The notion of Attribute Grammars can be generalized by viewing an attribute grammar as a specification (A, I) where A defines which attributes the attribution consists of, and where I is a set of invariants, stating truths about the attribution. A syntax tree is *consistently attributed* if the syntax tree has all the attributes declared in A and if all the invariants in I are fulfilled. For a standard AG, A is the set of declarations of inherited, synthesized, and local attributes, and I is the set of equations defining the values of these attributes.

A Door AG extends a standard AG both in what kind of attributes can be defined and in what kind of invariants can be given. For a Door AG, A includes not only the attribute declarations of standard AGs, but also declarations of *semantic objects* and *door objects* which may have their own attributes. The semantic objects may be used for context-dependent object structures such as visibility graphs. Door objects serve as connections between syntax nodes and semantic objects. The set of invariants I for a Door AG includes equations defining attribute values, exactly as in standard AGs, but also another kind of invariants called *conditions*, which can be used for defining the members of collection-valued attributes.

An important difference between Door AGs and standard AGs is that Door AGs allow reference attributes to denote any object, i.e. any node, door, or semantic object. These objects are, in general, *mutable* since their attributes may change values as a consequence of changes to the syntax tree. Door AGs thus allow references to denote mutable objects. In standard AGs, all attributes must have regular values, and references are therefore allowed only if they denote immutable objects (with the goal of space-efficient representation of regular values). The introduction of references to mutable objects is essential in order to model explicit visibility graphs and to implement the best incremental name analysis methods. The price for allowing such references is that non-local attribute dependencies are introduced. In general, this prevents evaluators to be completely automatically generated from the grammar. The reasons for this will be discussed in more detail in §10.10.

If evaluators cannot be generated automatically from a grammar, one of the most important benefits of the attribute grammar approach is lost. One of the major design goals of Door AGs has therefore been to be able to separate those parts which can be treated automatically from those which require manual treatment. This is done by splitting a Door AG specification into two parts: a *door package* and a *main grammar*. The door package contains the classes defining door objects and semantic objects and must be implemented manually. The main grammar contains the node classes and is very similar to a standard AG. It can be implemented automatically using techniques based on those for standard AGs. Although the implementation of door packages is manual, it can be done in a highly systematic fashion as will be shown in detail in chapters 9 and 10. A door package can be designed to handle some important aspects of a family of programming languages, e.g., name analysis and type systems. To specify a language it suffices to write a main grammar, using the door package as a tool box. The main grammar can then be processed and an incremental evaluator can be constructed automatically.

8.2 Nodes, doors, and semantic objects

A Door AG defines attributed syntax trees built out of three kinds of objects: *syntax nodes*, making up the syntax tree; *semantic objects*, which may be used to model context-dependent structures; and *door objects* which serve as interfaces between the syntax nodes and the semantic objects. The Door AG consists of class definitions for these objects. Syntax nodes may own door objects which in turn may own semantic objects. Thus, via static references, the doors and semantic objects can be seen as an extension of the syntax tree (Figure 8.1). The objects may have dynamic reference attributes, thus turning the tree into a graph (Figure 8.2).

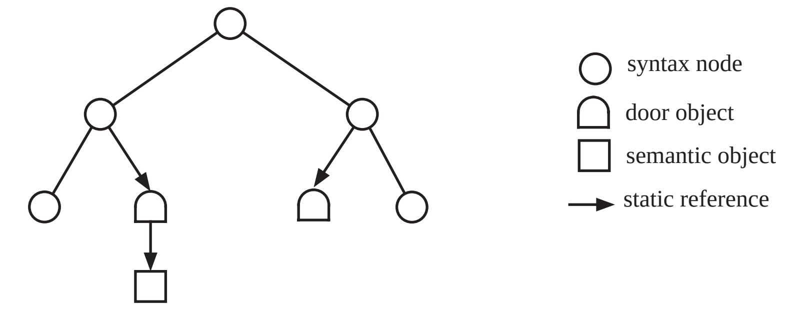


Figure 8.1 Syntax tree extended with doors and semantic objects

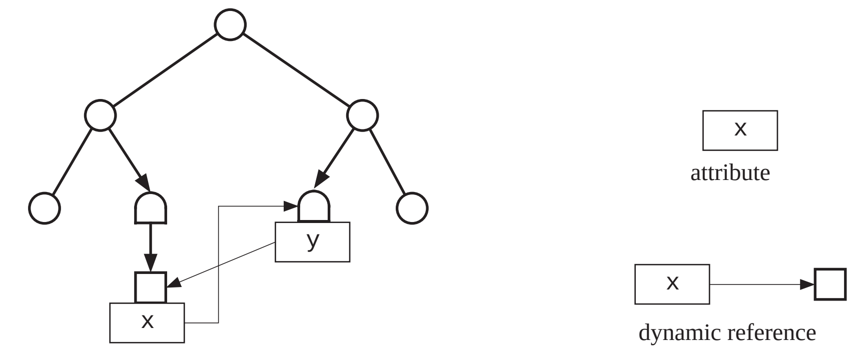


Figure 8.2 Attributed syntax tree

Semantic objects are instances of ordinary OOSL classes, which we will henceforth refer to as *semantic classes*. Nodes are instances of *node classes*, as introduced in Chapter 6, and doors are instances of special *door classes*. A semantic class may declare local attributes, virtual functions, and part-objects (of semantic classes).

Door classes have similarities to both semantic classes and node classes. The graphical depiction of the door objects is intended to indicate this. A door is similar to a node in that it may have inherited and synthesized attributes. This allows a door to be treated exactly like a son node by its owning syntax node. I.e., a syntax node must define the inherited attributes of its door objects, and it may use the synthesized attributes of its door objects to define other attributes. Similarly, a door must define its synthesized attributes, and it may use its inherited attributes to define its synthesized and other attributes.

A door is similar to a semantic object in that it may declare local attributes, virtual functions, and part-objects (of semantic classes). It is the responsibility of the door to declare equations defining the values of the local attributes in both the door itself and all its part-objects (including transitive part-objects). Some part-objects may, however, be so called *collection* objects. The contents of these objects are defined by *conditions* rather than equations, as will be described in §8.3.

A Door AG may also contain *constant* semantic object definitions. Such semantic objects are declared globally and are not owned by other objects.

Formally, a Door Attribute Grammar G is the combination of a main grammar G_M and a door package G_D. The main grammar is a tuple $G_M = (N, C_M)$ where N is a set of node classes and C_M a set of constant semantic objects. The door package is a triple $G_D = (D, S, C_D)$, where D is a set of door classes, S a set of semantic classes, and C_D a set of constant semantic objects. The Door AG is the quadruple $G = (N, D, S, C)$ where $C = C_M \cup C_D$.

8.2.1 OOSL extensions

Node classes for Door AGs are specified in OOSL in the same basic way as for the object-oriented AGs of Chapter 6. Semantic classes are specified as the normal classes of OOSL in Chapter 5, using the keyword “class”. Door classes are specified using the keyword “door” as follows:

```
<door-class-decl> ::=
    <door-class-id> ‘:’ ‘door’ [<door-superclass-id>]
    [ ‘{’ <decl-body> ‘}’]
```

The optional superclass of a door class must be another door class. If no superclass is explicitly given, the class is considered to be a subclass of a most general door class `ANYDOOR`. The class `ANYDOOR` is in turn considered a subclass of `ANYCLASS`. The top level of the OOSL class hierarchy is shown in Figure 8.3.

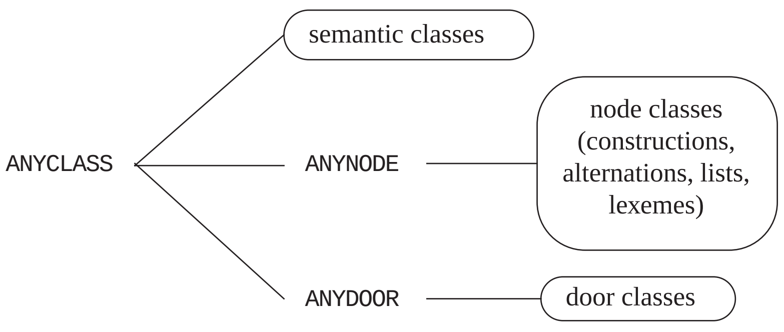


Figure 8.3 Top of OOSL class hierarchy

Semantic objects, constant or not, are specified using static references as explained in Chapter 5. Door objects are specified using the keyword “doorobject” as follows:

```
<stat-door-ref-decl> ::=
    <stat-door-ref-id> ‘:’ ‘doorobject’ <door-class-id>
```

For example:

```
D: door
{  inh x: integer;
};

N: cons
{  rD: doorobject D;
   eq rD.x := 3;
};
```

Here, `rD` is a static reference to a `D` door object. I.e., each object of the node class `N` has a part-object of door class `D`. The node defines the inherited attribute `x` of its door object. Figure 8.4 shows a syntax tree attributed according to the above definition.

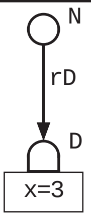


Figure 8.4 Attributed syntax tree

8.3 Collections and conditions

A semantic object may be declared as a *collection*. The content of a collection object is defined by *conditions* which appear in other door objects, distributed over the syntax tree. This is in contrast to other semantic objects whose contents are defined by equations in the owning door.

A condition declared in a door class `D` contributes to the total definition of a collection object by stating that an object `x` is a member of the collection. The object `x` must be either the `D` object itself or one of its part-objects (direct or indirect). Formally, a condition is simply a boolean expression which may be true or false. This allows a condition to state memberships conditionally. By using recursive functions, it is also possible to let a condition state an arbitrary number of memberships. In a consistently attributed syntax tree, all conditions must be true. The definition of collections follow a “closed world assumption” in that if the membership of an object `y` does not follow as a logical consequence from any condition, then `y` is defined to *not* be a member of the collection.

The following syntax is used.

```
<collection-decl> ::= ‘collection’ <stat-ref-decl>
<condition-decl> ::= <cond-id> ‘:’ ‘cond’ <exp>
```

As an example, consider the specification of symbol tables. Suppose there is a general class `List` which has a boolean function `contains(x)` which returns true if `x` is a member of the list:

```
List: class[T: class ANYCLASS]
{   contains: func boolean(x: ref T);
};
```

A class `Entry` is specified to model symbol table entry objects and a class `SymbolTable` to model the symbol table. The class `SymbolTable` declares a `List` object as a collection in order to collect suitable `Entry` objects:

```
Entry: class
{   ...
};

SymbolTable: class
{   collection entries: object List[Entry];
};
```

To use these semantic classes in a Door AG, we define one door class `D1` introducing a `SymbolTable` object, and another door class `D2` introducing an `Entry` object. The class `D2` registers the `Entry` object as a member of the list of a `SymbolTable` by using a condition:

```
D1: door
{   myTable: object SymbolTable;
    syn tbl: ref SymbolTable;
    eq tbl :- myTable;
};

D2: door
{   inh tbl: ref SymbolTable;
    myEntry: object Entry;
    reg: cond tbl.entries.contains(myEntry);
};
```

The condition `reg` defines that `myEntry` is a member of the list `tbl.entries`. The reason for giving conditions a name (e.g. `reg`) is merely that it makes it easier to refer to a given condition when analyzing specifications. The condition names are not used in the Door AG specification itself.

By inherited and synthesized reference attributes, information about a `SymbolTable` object can be transmitted from a `D1` object to one or several `D2` objects. Figure 8.5 shows an example syntax tree attributed according to the above definitions. In this example, a reference to the `D1` object has been transmitted via the syntax tree to two `D2` objects which each register an `Entry` object in the `SymbolTable` list.

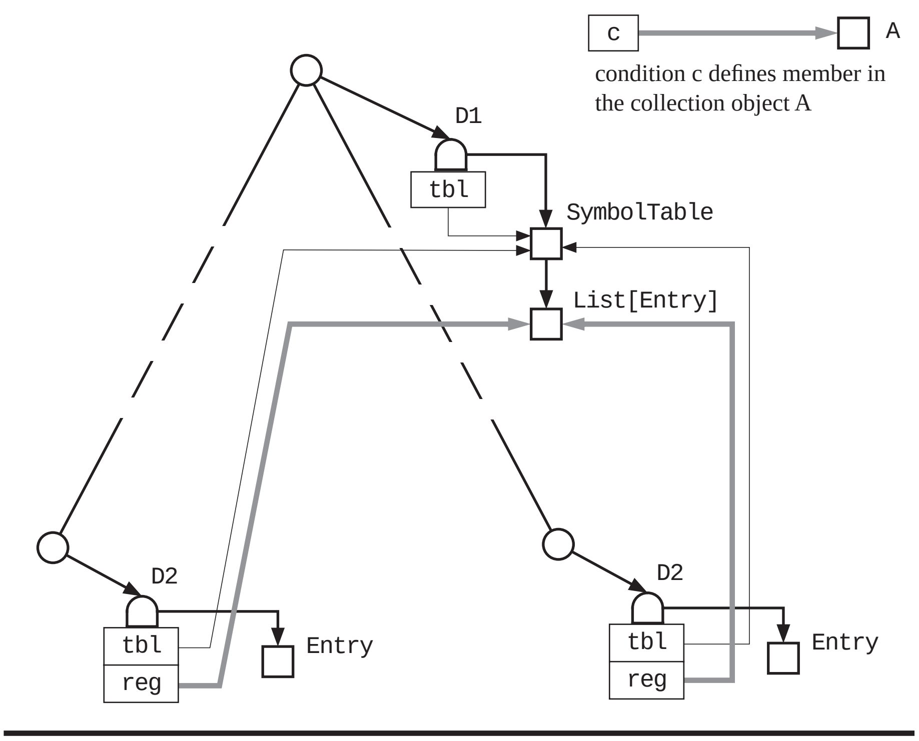


Figure 8.5 Members of a collection defined using conditions

8.4 Aggregates

The depiction of an attributed syntax tree can become quite complex if all details are included. We will use the term *aggregate* for a set of objects connected by static references. Depictions of attributed syntax trees can be simplified by collapsing aggregates to a single graphic symbol.

As an example, Figure 8.5 can be simplified as below. Here, the `SymbolTable` and `List` objects are collapsed to an aggregate, and each `D2` and its corresponding `Entry` object are also collapsed to aggregates. An additional simplification has been made by not showing the `tbl` attributes.

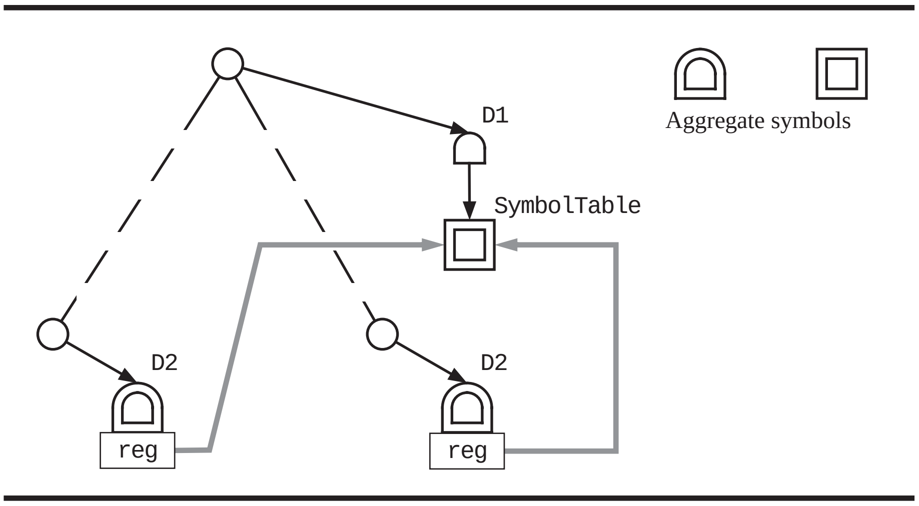


Figure 8.6 Simplifying depictions by using aggregate symbols

8.5 Fix attributes and functions

8.5.1 Fix attributes

Attributes may be declared as *fix*. This means that the definition of the attribute must be such that the value of the attribute never needs to be updated. I.e., once a fix attribute has obtained a value, it will need no future re-evaluation. The reason for introducing fix attributes is to simplify the construction of evaluators for door packages. Fix attributes also allow a higher degree of instruction skipping, as will be discussed in §10.9.1.

We assume the usual editing model based on subtree replacements. In this model, the lifetime of a node always spans the lifetime of its son nodes. Suppose a node *n* defines a fix inherited attribute *a* of a son node. Because of the subtree replacement editing model, the lifetime of the attributes in *n* will span the lifetime of *a* and the equation defining *a* may therefore use fix attributes in *n*. The equation may also use synthesized attributes of door nodes owned by *n*, since a part-object has the same lifetime as its owning object. The equation may, however, not use synthesized attributes of a son node of *n*, since replacement of the son node may result in a new value for the synthesized attribute. This could lead to a new value for *a* which would violate its fix property.

To declare an attribute as fix, the keyword “fix” is added at the end of the declaration:

```
<fix-attr-decl> ::=
    ( ‘inh’ | ‘syn’ | ‘loc’ ) <attr-id> ‘:’ <type> ‘fix’
```

Typically, fix attributes are used to propagate information from one door in the syntax tree to another door further down in the syntax tree. For example, consider the example on SymbolTable objects of §8.3. The synthesized and inherited tbl attributes can be declared fix as follows:

```
D1: door
{   myTable: object SymbolTable;
    syn tbl: ref SymbolTable fix;   (* 1 *)
    eq tbl :- myTable;              (* 2 *)
};

D2: door
{   inh tbl: ref SymbolTable fix;   (* 3 *)
    myEntry: object Entry;
    reg: cond tbl.entries.contains(myEntry);
};
```

The synthesized attribute tbl at (* 1 *) is declared as fix. Clearly, the equation defining this attribute (* 2 *) fulfills this since myTable is a static reference and thus never changes. A syntax node *n* owning a D1 door may use the synthesized tbl reference to define fix inherited attributes of son nodes. The reference may be propagated throughout the subtree of *n* while retaining the fix property. However, the reference cannot be propagated up to the father node of *n* without losing the fix property. Since the D2 requires its inherited tbl attribute to be fix (* 3 *), the reference can only be propagated to D2 doors within the subtree rooted at *n*. Thus, the use of fix attributes restricts the way the door package can be used by a main grammar.

8.5.2 Fix functions

A function may also be declared as fix. This means that the function must return the same value for a given set of parameters, regardless of changes to the syntax tree and consequent changes to the attribution. An implementation of a fix function may only make use of constant information and of other fix attributes and functions. If a function is declared as fix, all implementations of it must fulfill this requirement. To declare a function as fix, the keyword “fix” is added after the function specification:

```
<fix-func-spec> ::=
    <func-id> ‘:’ ‘func’ <type>
    [‘(’ (<form-par>, ‘,’)+ ‘)’]
    ‘fix’
```

For example:

```
f: func boolean fix;
```

8.6 Non-local dependencies

The introduction of reference attributes in Door AGs leads to non-local dependencies. This section will define what is meant by such dependencies and give an example of how they can occur.

Henceforth, we will use the term *access dependency* for the dependencies usually considered in standard AGs. Chapter 9 will consider slightly different kinds of dependencies.

8-1 Definition Access dependency

Let *a* be an attribute defined by an equation *e*. For each attribute *b* accessed by the right hand side of *e*, there is an *access dependency* from *b* to *a*, and *a* is said to be *access-dependent* on *b*.

end 8-1

In a standard AG, all access dependencies are *local*. I.e., they occur between attributes of the same or neighbor syntax nodes. With *neighbor* syntax nodes we mean nodes related as father and son. The concept of local access dependencies is extended to Door AGs by considering the *extended syntax tree*, i.e. the syntax tree extended by door objects and semantic objects. In the extended syntax tree, a door object and all its semantic part-objects are considered to be neighbors to each other and to the syntax node owning the door object. Local and non-local access dependencies in a Door AG are defined as follows:

8-2 Definition Local/non-local access dependency

Let *a* be an attribute which is access-dependent on another attribute *b*. Let x_a be the object declaring *a* and x_b the object declaring *b*. The dependency is said to be a *local access dependency* iff x_a and x_b are the same object or x_a and x_b are neighbors in the extended syntax tree. Otherwise, the dependency is said to be a *non-local access dependency.*

end 8-2

As an example of a non-local access dependency, consider the following classes.

```
A: class
{   loc x: integer;
};

D1: door
{   inh ix: integer;
    syn rA1: ref A;
    myA: object A;
    eq myA.x := ix;       (* 1 *)
    eq rA1 :- myA;        (* 2 *)
};

D2: door
{   inh rA2: ref A;
    syn sx: integer;
    eq sx := rA2.x;       (* 3 *)
};
```

A `D1` door uses the inherited attribute `ix` to define the `x` attribute of its `A` part-object `(* 1 *)`. A main grammar declaring a `D1` door gets access to the `A` object by the synthesized reference attribute `rA1` `(* 2 *)`. This reference may be propagated (using normal copy equations) to another part of the syntax tree into the `rA2` attribute of a `D2` door. The `D2` door accesses the `x` attribute of the `A` object and copies this value to the synthesized attribute `sx` `(* 3 *)`.

Figure 8.7 illustrates this example. References to the `A` object are propagated from `rA1` along the syntax tree to `rA2`. In contrast, the information from `x` to `sx` flows directly from the `A` object to the `D2` door although these two objects may be located far from each other in the syntax tree. The attribute `sx` is also dependent on `rA2`. I.e., changing either `x` or `rA2` will affect `sx`.

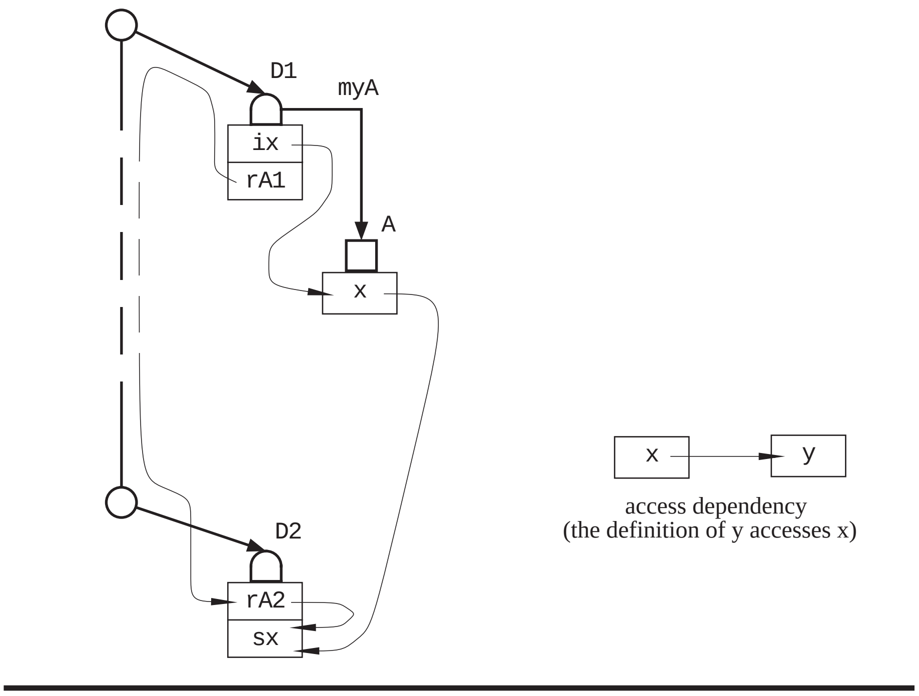


Figure 8.7 Access dependencies

From the definition above, it is clear that the dependency from `ix` to `x` (by equation `(* 1 *)`) is a local access dependency, whereas the dependency from `x` to `sx` (by equation `(* 3 *)`) is a non-local access dependency. The copy equations which must be present in the syntax tree in order to propagate the `A` reference from the `D1` door to the `D2` door give rise to a chain of local access dependencies.

Note that the equation `(* 2 *)` does not give rise to any dependency since `myA` is a static reference and not an attribute (in the attribute grammar sense).

Note also that a change to the attribute `x` does not affect any of the references denoting the `A` object (e.g. `myA`, `rA1`, and `rA2`). The values of these attributes is the object identity of the `A` object, which is immutable and unaffected by any changes to the contents of the object.

The non-local dependencies between attributes can be mapped to non-local dependencies between objects. We are here only interested in syntax nodes and door objects and see the semantic part-objects more as passive entities. I.e., an attribute of a semantic part-object is mapped to the owning door object rather than to the semantic object in which it is declared. This implies that the dependency from `x` to `sx` is mapped to a dependency from the `D1` object to the `D2` object.

In principle, it would make sense to put the equation `(* 3 *)` directly in a syntax node instead of in the door `D2`. There is an important reason why this is not done. The reason is that we want no non-local access dependencies involving syntax nodes. Putting the equation `(* 3 *)` in a syntax node would give a non-local access dependency from the `A` object to that syntax node. We want all non-local access dependencies to be mapped on “door-to-door” dependencies. This is necessary in order to be able to automatically generate evaluators for the main grammars. This restriction is accomplished by the following rule:

8-3 **Rule** Access via reference attributes in syntax nodes

Syntax nodes may not access mutable information via reference attributes. The only information which a syntax node may access via a reference attribute is the *immutable* information, i.e., the object identity itself, and fix functions in the denoted object.

end 8-3

Although access to mutable information via reference attributes is not allowed directly in the syntax nodes, such accesses can always be performed indirectly by introducing a door object performing the access.

8.7 Data and demand attributes

In Door AGs, most of the important information is maintained in doors and semantic objects. The attributes of syntax nodes are used mainly for propagating information between doors and for simple computations which take little time to recompute when needed. For this reason, we consider all attributes in the syntax nodes as *demand* attributes by default. In addition, both the synthesized and the inherited attributes of doors are considered as demand attributes. It is thus only the local attributes of semantic objects and doors which are stored as data attributes.

For syntax nodes, attributes can be explicitly specified as data attributes by adding the keyword “data” after the attribute declaration. For semantic objects and doors there is no need for a corresponding possibility to explicitly specify attributes as “demand” attributes, since parameterless virtual functions are equivalent to local demand attributes.

The following items summarize the default scheme for data/demand attributes:

- All attributes (synthesized, inherited, and local) of syntax nodes are by default implemented as *demand* attributes.
- All synthesized and inherited attributes of door objects are implemented as *demand* attributes.
- All local attributes of door objects and semantic objects are implemented as *data* attributes.

The difference between data and demand attributes does not affect the meaning of a Door AG. It is only a question of time and space consumption. However, when the goal is to develop practical incremental attribute evaluators, we find it very important that the grammar designer has full control over what part of the attribution is stored. In particular, it is important to not be forced to store large numbers of attributes simply to be able to propagate information from one part of the syntax tree to another.

8.8 Summary of graphical symbols

The previous sections have introduced a number of graphical symbols used for depicting attributed syntax trees. Figure 8.8 summarizes this graphical notation.

Note that an attribute dependency often points in the opposite direction of a dynamic reference. This is because a dynamic reference attribute of an object *a* which denotes another object *b* makes it possible to define attributes in *a* using attributes in *b*.

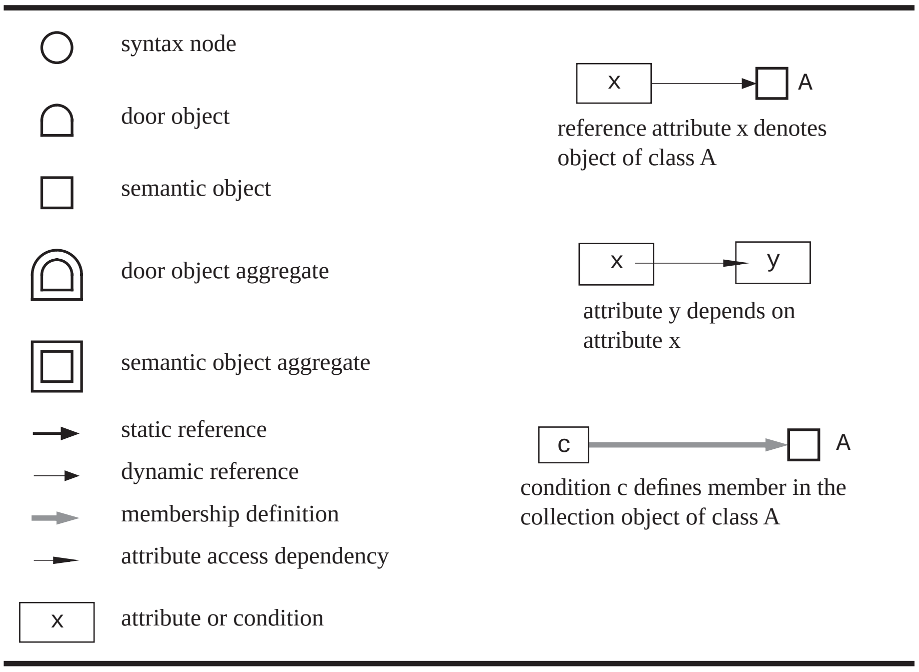

syntax node
door object
semantic object
door object aggregate
semantic object aggregate
static reference
dynamic reference
membership definition
attribute access dependency
x
attribute or condition
x
A
reference attribute x denotes object of class A
x
y
attribute y depends on attribute x
c
A
condition c defines member in the collection object of class A

Figure 8.8 Graphical notation for attributed syntax trees

8.9 An example Door AG

As an example of a Door AG we will show the construction of a door package supporting name analysis for nested blocks. The static semantics of a small Algol like language will be specified using this door package. The language includes nested blocks, integer and boolean variables, assignment statements, and integer constants. An example program in this language may be:

```
begin
   integer x;
   x := 1;
end;
```

Figure 8.9 shows an attributed syntax tree for this program. The main goal of the attribution is to bind name applications to name declarations. Name applications are represented by *`UseDoor`* objects and name declarations by *`DeclDoor`* objects. Bindings are represented by the reference attribute `binding` in a `UseDoor` which denotes the appropriate `DeclDoor` (see the lower part of the figure).

Blocks are represented by *`BlockDoor`* objects. Each `BlockDoor` has a `SymbolTable` object which collects the declarations of the block. `SymbolTable` objects are connected by objects modelling the vertices of visibility graphs as described in §3.3. The class `TwoPath` models a path vertex with two outgoing edges (`first` and `second`). Each `BlockDoor` has a `TwoPath` object to model the combination of local and enclosing scope. In this example, the block is the topmost block in the program. The attribute `second` therefore denotes the constant object `emptyPath` which models the null vertex.

In addition to the `UseDoor`, `DeclDoor`, and `BlockDoor`, the example shows a *`RootDoor`*. The `RootDoor` is used at the root of the syntax tree to provide a suitable definition for the enclosing scope for the first block. There will thus only be one `RootDoor` object in a syntax tree.

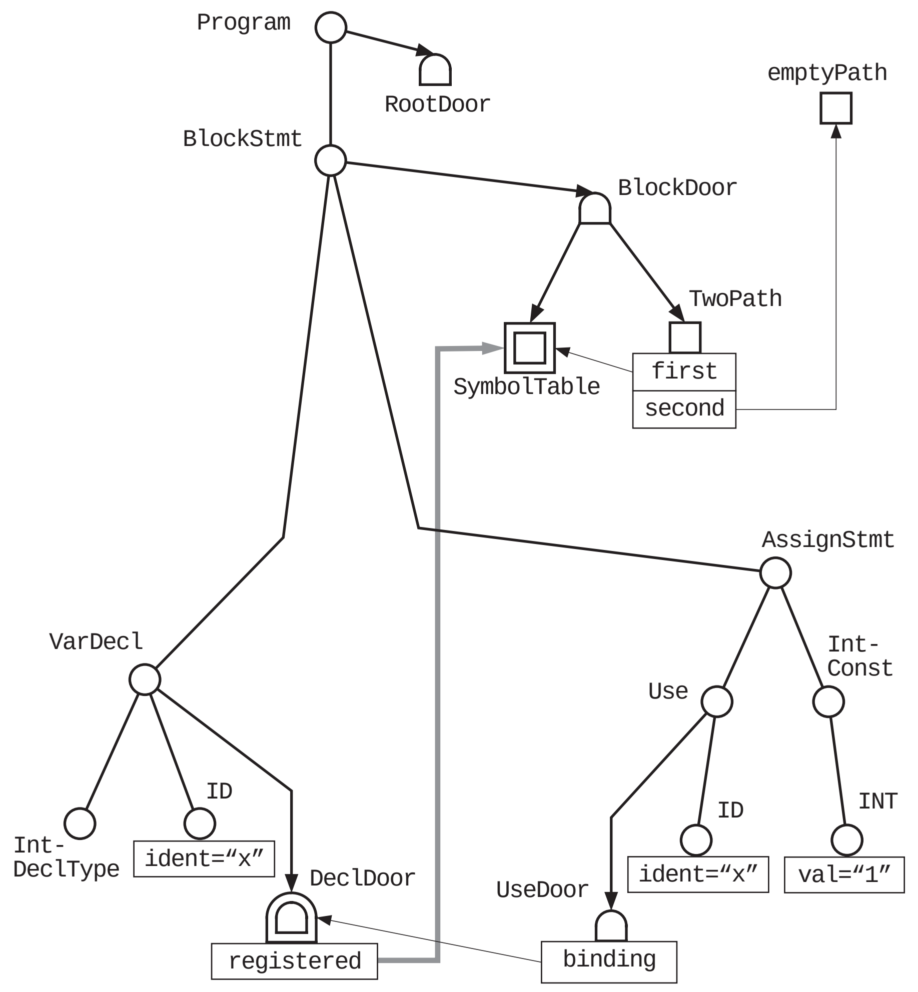


Figure 8.9 Attributed syntax tree

To define and incrementally maintain an attribution such as the one above, several additional attributes are needed. There are two main groups of attributes not shown in the figure. One group is all the inherited and synthesized attributes of the nodes and doors needed for communicating information from one part of the syntax tree to another. The other group is the attributes needed for efficient incremental evaluation.

For example, information about the `SymbolTable` object needs to be propagated down to the `DeclDoor` so it can register itself properly. Similarly, information about the `TwoPath` object needs to be propagated down to the `UseDoor` object so it can use this information to set its binding attribute properly. This propagation can be done by synthesized and inherited attributes in the nodes and doors. Since these attributes are by default demand attributes, they take up no storage.

As an example of information needed during incremental evaluation, consider changing the `ident` attribute of the `VarDecl ID` node from “`x`” to “`y`”. Such a change implies that the `binding` attribute of the `UseDoor` needs to be updated to denote some other object. There is thus a need of additional attributes which make it possible to locate the affected `UseDoor` objects efficiently after such a change.

In this section, we consider only the attributes needed for *defining* the above attribution. The additional attributes needed for efficient incremental evaluation will be discussed in Chapter 9.

The Door AG is divided into a main grammar and a door package as described in §8.2. The door package consists of definitions of the four door classes `RootDoor`, `BlockDoor`, `DeclDoor`, and `UseDoor`, and of definitions of associated semantic classes and constant objects. The main grammar consists of node classes for a specific language.

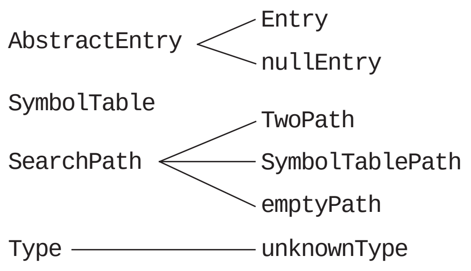


Figure 8.10 Specialization hierarchy of semantic classes and objects

8.9.1 Semantic classes and constant objects

The semantic classes and constant objects of interest and their specialization relationships are shown in Figure 8.10.

- The class `Entry` models a declared identifier. The `nullEntry` is a constant object representing an undeclared identifier. The class `AbstractEntry` is a generalization of these two.
- The class `SymbolTable` collects `Entry` objects.

- The class SearchPath models a vertex in a visibility graph. Its specialization TwoPath models a path vertex with two outgoing edges. The SymbolTablePath models a table vertex and connects to a SymbolTable object. The emptyPath object models the null vertex.
- The class Type models the general concept of a type in a programming language. One constant specialization is given: unknownType, which is used for modelling the types of undeclared identifiers. Other constant objects which are specializations of Type can be defined in the main grammar, in order to suit the needs of the specified programming language.

8.9.1.1 Linked lists

The door package makes use of a separate package for linked lists to implement the collection of Entry objects in a SymbolTable object. The linked list package has the following interface:

```
List: class[T: class Element]
{ contains: func boolean(e: ref T);
  first: func ref T; (* may return NONE *)
  sucOf: func ref T(e: ref T); (* may return NONE *)
  predOf: func ref T(e: ref T);(* may return NONE *)
};

Element: class;
```

The class List models a list head and the class Element a list element. The function contains returns true if e is in the list. The function first returns the first element in the list, or NONE if the list is empty. The sucOf and predOf functions return the successor and predecessor of an element e in the list, or NONE if e is the last or first element, respectively.

8.9.1.2 AbstractEntry, Entry, nullEntry

The class AbstractEntry and its specializations are defined as below. A reference qualified by AbstractEntry may be used to represent the binding of an identifier. It will denote an Entry object if the identifier is declared or the nullEntry object otherwise. The virtual function getTp can be used to retrieve the type of the identifier. `AbstractEntry` is defined as a subclass of `Element` in order to be able to put `Entry` objects into linked lists.

```
AbstractEntry: class Element (* abstract *)
{ getTp: func ref Type;
};

Entry: class AbstractEntry
{ loc ident: string;
  loc tp: ref Type;
  impl getTp :- tp;
};

nullEntry: object AbstractEntry
{ impl getTp :- unknownType;
};
```

8.9.1.3 SymbolTable

The class `SymbolTable` collects `Entry` objects using a collection part-object of class `List[Entry]`. A function `lookup` traverses the collection to find an `Entry` object for a given identifier.

```
SymbolTable: class
{ collection entries: object List[Entry];
  lookup: func ref Entry(ident: string)
           (* may return NONE *)
    :-  loop $L :- entries.first do
          inspect $E :- $L
          when Entry do
            if $E.ident = ident
            then $E
            else next $L :- entries.sucOf($L)
          otherwise NONE;
};
```

8.9.1.4 SearchPath

The class `SearchPath` specifies a virtual function `lookup` which traverses the visibility graph reachable from that point to find the declaration of a given identifier. The function is implemented in different ways in the specializations of `SearchPath`.

```
SearchPath: class (* abstract *)
{ lookup: func ref Entry(ident: string);
         (* may return NONE *)
};

TwoPath: class SearchPath
{ loc first: ref SearchPath;
  loc second: ref SearchPath;
  impl lookup :-
    inspect $E :- first.lookup(ident)
    when Entry do $E
    otherwise second.lookup(ident);
};

SymbolTablePath: class SearchPath
{ loc table: ref SymbolTable;
  impl lookup :- table.lookup(ident);
};

emptyPath: object SearchPath
{ impl lookup :- NONE;
};
```

8.9.1.5 Type

The definition of the semantic class `Type` and its specialization `unknownType` is as follows.

```
Type: class;
unknownType: object Type;
```

8.9.2 Door classes

8.9.2.1 RootDoor

The door class `RootDoor` simply defines a synthesized attribute which denotes the constant semantic object `emptyPath`. This attribute is intended to be used as the enclosing environment for the topmost block:

```
RootDoor: door
{ syn rootPath: ref SearchPath fix;
  eq rootPath :- emptyPath;
};
```

8.9.2.2 BlockDoor

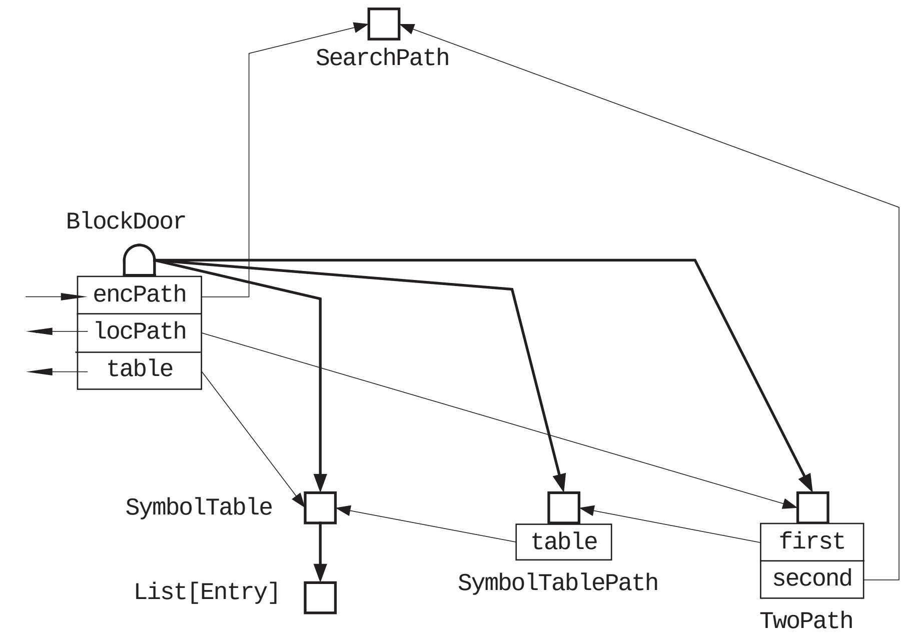
The door class `BlockDoor` defines three part-objects: a `SymbolTable`, a `SymbolTablePath`, and a `TwoPath` object. The `TwoPath` object models the static path vertex according to the construction of visibility graphs for block structure given in §3.3.1. The part objects of `BlockDoor` are connected to each other and to a `SearchPath` object representing the enclosing environment. Information about the latter object is obtained via an inherited reference attribute of the door.

The door also defines two synthesized reference attributes. One denoting the `SymbolTable` object and one denoting the `TwoPath` object. The `SymbolTable` reference can be propagated to other parts of the syntax tree allowing `DeclDoor` objects to register `Entry` objects as members of the `SymbolTable` collection. The `TwoPath` reference can be propagated to other parts of the syntax tree allowing other blocks to be constructed using this one as enclosing environment, and allowing `UseDoor` objects to look up declarations of identifiers.

The definition of the `BlockDoor` class is as follows:

```
BlockDoor: door
{ inh encPath: ref SearchPath fix;
  syn locPath: ref SearchPath fix;
  syn table: ref SymbolTable fix;
  theTable: object SymbolTable;
  staticPath: object TwoPath;
  theTablePath: object SymbolTablePath;
  eq staticPath.first :- theTablePath;
  eq staticPath.second :- encPath;
  eq theTablePath.table :- theTable;
  eq table :- theTable;
  eq locPath :- staticPath;
};
```

The following figure shows a `BlockDoor` attributed according to the above definition. Its synthesized and inherited attributes are shown as well, although they are demand attributes and not actually stored. The incoming and outgoing dependency edges show the information flow for these attributes. The `SearchPath` object will actually be a specialization of class `SearchPath`. It will either be the constant `emptyPath` or a `TwoPath` object of another `BlockDoor`.

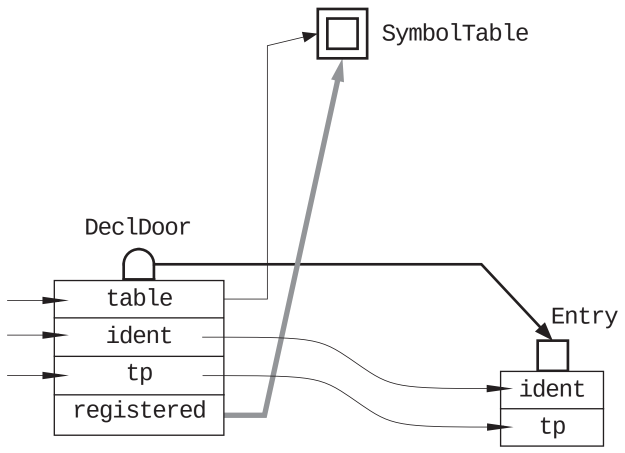


8.9.2.3 DeclDoor

The `DeclDoor` class declares an `Entry` part-object which it registers in a `SymbolTable` by declaring a condition. Information about the `SymbolTable` object is obtained via an inherited reference attribute. The door also has two additional inherited attributes containing the name and the type of the declaration. These attributes are used for defining the corresponding attributes in the `Entry` object:

```
DeclDoor: door
{ inh table: ref SymbolTable fix;
  inh ident: string;
  inh tp: ref Type;
  theEntry: object Entry;
  registered: cond table.entries.contains(theEntry);
  eq theEntry.ident := ident;
  eq theEntry.tp :- tp;
};
```

The following figure shows a `DeclDoor` attributed according to the above definition. The dependency edges show the information flow from the inherited `ident` and `tp` attributes to the corresponding attributes in the `Entry` object.



8.9.2.4 *UseDoor*

The `UseDoor` looks up the appropriate `Entry` object for an identifier by calling the `lookup` function of a `SearchPath` object. Both the identifier and the reference to the `SearchPath` are inherited attributes of the door. The result of the `lookup` function is used for defining a local attribute `binding` in the door. In case there is no matching `Entry` object, the `binding` attribute is defined to denote the `nullEntry` object. The `binding` attribute is also defined to denote the `nullEntry` object if the identifier is the empty string. This reason for this is that the `ident` attribute of un-expanded `ID` nodes in the syntax tree is assumed to have the value of the empty string. If a normal binding would be attempted for the empty string this could lead to binding an applied un-expanded identifier to a declared un-expanded identifier, which is probably not a desirable behavior of the door package.

The `binding` attribute is used for retrieving the type of the identifier. A reference denoting the appropriate `Type` object is made available to the syntax tree through a synthesized attribute of the door. In addition, a synthesized attribute `declared` is defined which is true if the identifier has a proper declaration.

```
UseDoor: door
{ inh path: ref SearchPath;
  inh ident: string;
  syn tp: ref Type;
  syn declared: boolean;
  loc binding: ref AbstractEntry;
  eq binding :-
      if ident = “”
      then nullEntry
      else
        inspect $E :- path.lookup(ident)
        when Entry do $E
        otherwise nullEntry;
  eq tp :- binding.getTp;
  eq declared := binding =/= nullEntry;
};
```

The following figure shows an example of a `UseDoor` attributed according to the above definition. The inherited `SearchPath` attribute denotes a `TwoPath` object which is in turn connected to another `TwoPath` object of an enclosing block. The function `lookup` finds the appropriate `Entry` object by first searching the lower `SymbolTable` collection and, when not found there, continues searching in the upper `SymbolTable` collection. The figure also shows the dependency edge from the `tp` attribute of the bound `Entry` object to the corresponding synthesized attribute of the door.

The inherited attribute `path` is not declared as fix. It may thus change during incremental evaluation to denote other `SearchPath` objects. This allows the `UseDoor` to be used not only in simple Algol-like languages, but also in languages with remote access. This will be discussed in §11.3.2.2.

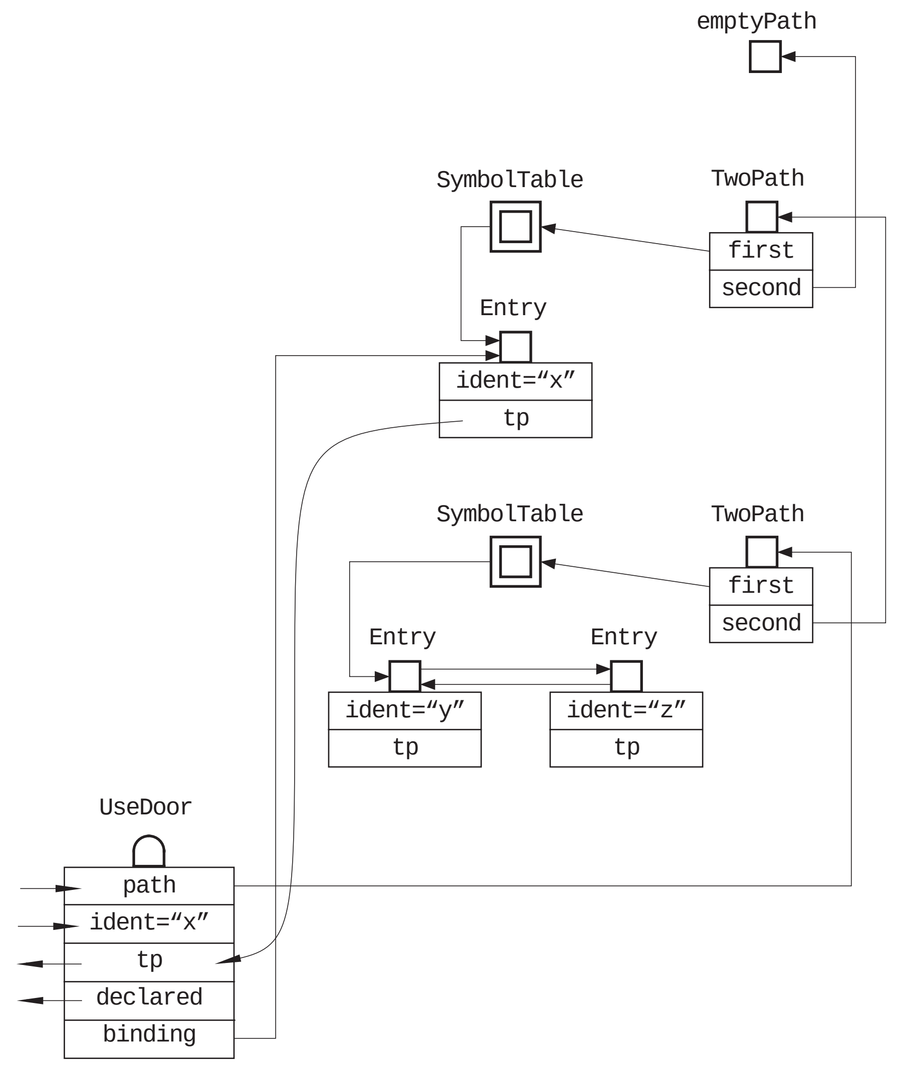


Figure 8.11 A `UseDoor` bound to an `Entry` object in an enclosing block

8.9.3 The door package interface

The definitions of the previous two sections constitute the door package. However, only a small part of this information is actually used directly by a main grammar. An interface can be extracted from the door package, containing only those definitions which are needed to write a main grammar:

```
SearchPath: class; (* non-instantiable *)

SymbolTable: class; (* non-instantiable *)

Type: class;
unknownType: object Type;

RootDoor: door
{ syn rootPath: ref SearchPath fix;
};

BlockDoor: door
{ inh encPath: ref SearchPath fix;
  syn locPath: ref SearchPath fix;
  syn table: ref SymbolTable fix;
};

DeclDoor: door
{ inh table: ref SymbolTable fix;
  inh tp: ref Type;
  inh ident: string;
};

UseDoor: door
{ inh path: ref SearchPath;
  inh ident: string;
  syn tp: ref Type;
  syn declared: boolean;
};
```

Figure 8.12 Door package interface

The essence of the door package interface is the inherited and synthesized attributes of the door classes. In addition, the interface includes the types (semantic classes) of these attributes. The interface also includes constant objects which are of interest when defining the main grammar.

Note that the interface includes only the abstract semantic class `SearchPath` and not its specializations. Note also that the `AbstractEntry` and its specializations are completely hidden. In principle, an equivalent door package could be constructed which has the same interface as above, and the same black box behavior, but a different internal specification, involving other semantic classes.

Both `SearchPath` and `SymbolTable` are marked as `(* non-instantiable *)` in the interface. This means that the main grammar may not declare constant semantic objects of these classes. Because of this restriction, the main grammar cannot construct its own constant semantic objects of these types and feed them into the door package. It can only get access to objects of these classes through synthesized reference attributes of the door package itself.

The Type class, on the other hand, can freely be used by the main grammar for declaring semantic constants to suit the needs of a particular language. For example, constant Type objects may be declared in order to represent integers and booleans.

```
intType: object Type;
boolType: object Type;

Program: cons (s: ref Stmt)
{ r: doorobject RootDoor;
  eq s.path :- r.rootPath;
};

Stmt: alt
{ inh path: ref SearchPath fix;
};

NullStmt: cons Stmt();

BlockStmt: cons Stmt
  (d: ref Decl, s: ref Stmt)
{ b: doorobject BlockDoor;
  eq b.encPath :- path;
  eq d.table :- b.table;
  eq s.path :- b.locPath;
};

Decl: alt
{ inh table:
    ref SymbolTable fix;
};

NullDecl: cons Decl();

VarDecl: cons Decl
  (dt: ref DeclType,
  declId: ref ID)
{ d: doorobject DeclDoor;
  eq d.table :- table;
  eq d.ident := declId.ident;
  eq d.tp :- dt.tp;
};

DeclType: alt
{ syn tp: ref Type;
};

NullDeclType: cons DeclType()
{ eq tp :- unknownType;
};
```

```
IntDeclType: cons DeclType()
{ eq tp :- intType;
};

BoolDeclType: cons  DeclType()
{ eq tp :- boolType;
};

AssignStmt: cons Stmt
  (to: ref Use, from: ref Exp)
{ loc error: boolean;
  eq to.path :- path;
  eq from.path :- path;
  eq error := not
    (to.tp == unknownType or
    from.tp == unknownType or
    to.tp == from.tp);
};

Exp: alt
{ inh path: ref SearchPath;
  syn tp: ref Type;
};

NullExp: cons Exp()
{ eq tp :- unknownType;
};

Use: cons Exp (useId: ref ID)
{ u: doorobject UseDoor;
  loc error: boolean;
  eq u.path :- path;
  eq u.ident := useId.ident;
  eq tp :- u.tp;
  eq error :=
      useId.ident <> “”
      and not u.declared;
};

IntConst: cons Exp
  (n: ref INT)
{ eq tp :- intType;
};
```

Figure 8.13 An example main grammar

8.9.4 The main grammar

A main grammar for a small Algol-like language is given in Figure 8.13. It uses the door package to define the attribution of Figure 8.9. Two constant `Type` objects are declared: `intType` and `boolType`, to represent the types used in this small language.

The grammar also does static semantic error checking, by defining boolean `error` attributes.

- The `Use` node checks if the applied identifier is declared. However, if the identifier is the empty string, this is not considered an error since empty strings represent un-expanded `ID` nodes.
- The `AssignStmt` node checks if the left and right hand sides are type compatible. If either one of these types is the `unknownType`, this indicates an undeclared identifier or an un-expanded `ID` or expression, and is not considered an error.

Figure 8.14 shows the attributed syntax tree of Figure 8.9 again, but now with all the attributes of the main grammar and the synthesized and inherited attributes of the door objects, and with all of the semantic part-objects collapsed into aggregates. The dependency edges between attributes show the information flow through the syntax tree. It can be noted that all of these attributes are demand attributes, and thus take up no storage.

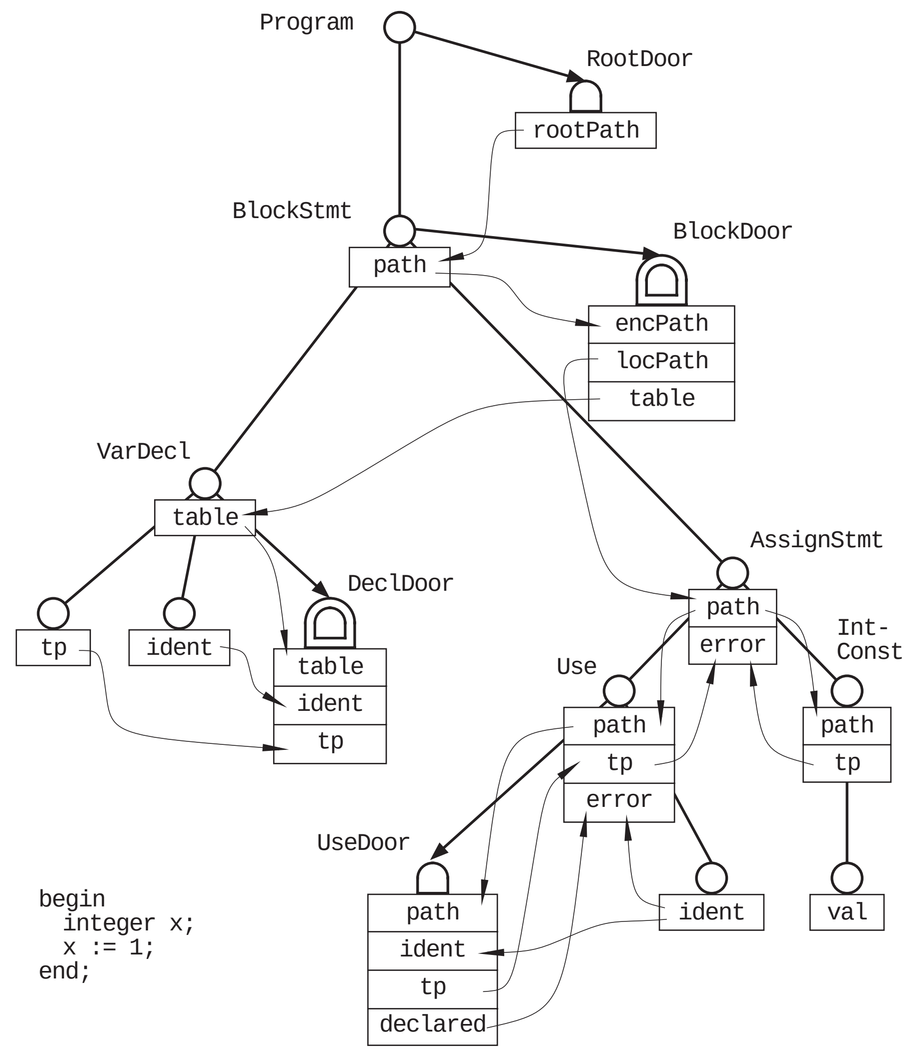


Figure 8.14 Information flow in an example attributed syntax tree

8.10 Underdetermined grammars

A standard AG *solution* of a syntax tree is an attribution for which all equations of the grammar hold (§4.2). In analogy, for a Door AG, a solution of a syntax tree is an attribution for which all equations hold and for which all conditions are true.

Non-circular standard AGs are always *uniquely-defined*, i.e. there is exactly one solution for each possible syntax tree. In constrast, it is both possible and useful to construct Door AGs which are *underdetermined*, i.e., some syntax trees may have multiple solutions. One reason to use an underdetermined grammar is to make it possible to implement history-dependent error reporting, as discussed in §3.6. Circular standard AGs can, in principle, also be underdetermined (see §4.3). However, to the author's knowledge, this has not been utilized in practical applications.

It is the use of *collections* in Door AGs which makes it possible to construct interesting underdetermined grammars. A direct way of constructing an underdetermined grammar is simply to use an ordered collection, e.g. a list. The conditions define which elements should be members of the list, but not in which order the elements should appear in the list. There are thus as many solutions as there are permutations of the elements. The values of other attributes may depend on the actual order, so quite different solutions can result from this underdeterminedness.

As an example, the Door AG of §8.9 is underdetermined in exactly this way. In this grammar, the list of `Entry` objects in a `SymbolTable` is ordered, but the order is not defined in the grammar. The order is of importance at lookup, where the first `Entry` for a given identifier is returned. If a block contains more than one declaration for the same identifier, the lookup function can return different `Entry` objects depending on the order in the list and the binding of a name application can thus differ in different solutions. Consider the following program:

```
begin
   integer x;
   boolean x;
   x := 1;
end;
```

The syntax tree for this program (according to the Door AG of §8.9) has two solutions, as illustrated in Figure 8.15.

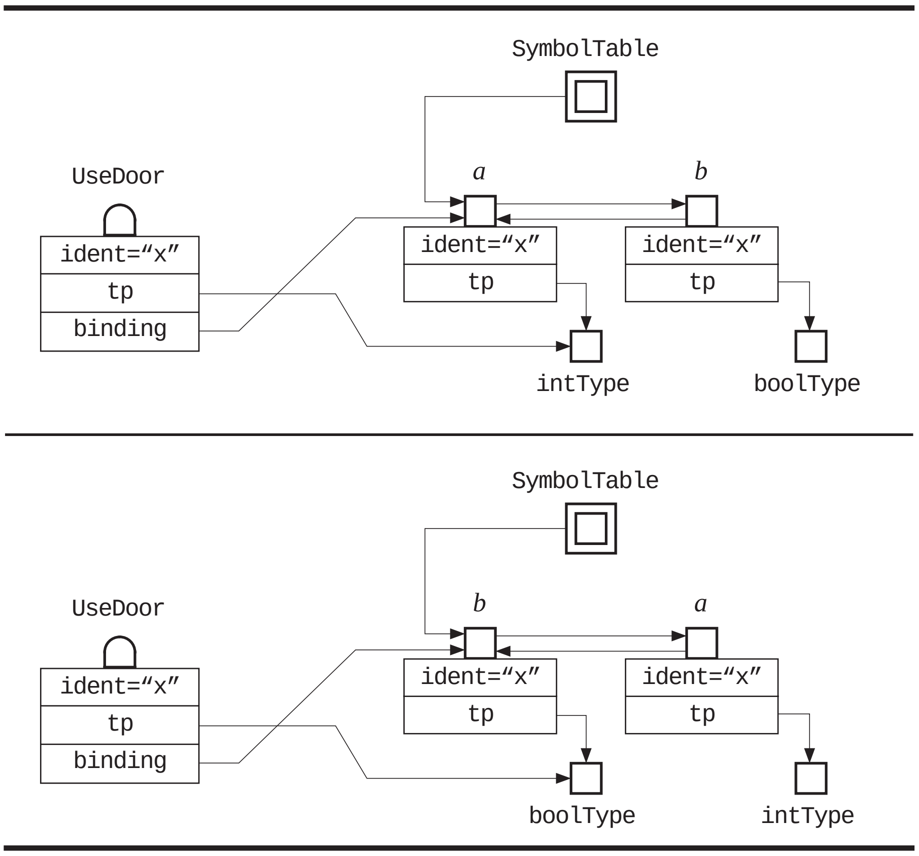


Figure 8.15 Two solutions of an underdetermined syntax tree

The two `Entry` objects *a* and *b* appear in different order in the two solutions. This leads to that both the `binding` and the `tp` attribute of the `UseDoor` denote different objects in the two solutions. Thus, in the first solution, the name application `x` is considered to be an integer and in the second it is considered to be a boolean.

In any given solution, preference is thus given to one of the `Entry` objects for a given identifier, and name applications cannot be bound to the other ones. I.e., it is only one of the `Entry` objects for a given identifier which is actually visible, although all of them are present in the symbol table. A natural extension of the door package would be to add a synthesized boolean attribute `visible` to `DeclDoor` which is defined to be true if the `Entry` object is not preceeded by any other `Entry` object for the same identifier. This attribute could then be used in a main grammar to define an error message for declarations that are not visible. Such an extension will be discussed in more detail in Chapter 11.

The above method of using ordered collections is not the only way to construct an underdetermined Door AG. Underdeterminism can be achieved also if only unordered collections are used, by defining reference attributes of the collected objects in a certain way. An example of this will be given in Chapter 11 to show how cyclic subclassing can be handled by a Door AG.

Note that it is perfectly possible to specify uniquely-determined Door AGs rather than underdetermined ones. For example, other door packages can be constructed which handle multiple declarations in the traditional uniquely-determined manner. I.e., by either defining all declarations that have namesakes in the same block as “invisible”, or by defining the syntactically foremost declaration as the visible one.

8.11 Comparison to standard AGs

In §4.4, we discussed a number of limitations of standard AGs. These limitations are all relaxed by the extensions introduced in Door AGs. Below, we again list these limitations and comment briefly on how they are relaxed.

I *Regular values*. While attributes of nodes in standard AGs must have regular values, the attributes of nodes, doors, and semantic objects in Door AGs may be *references*.

II *Whole attributes*. An equation in a standard AG always defines a *whole* attribute value. In a Door AG, large attribute values are replaced by semantic objects. These objects have their own attributes which are defined individually by equations or conditions. This allows a finer granularity of definition than what is possible in standard AGs.

III *Simple assertions*. Standard AGs have only one kind of assertions, namely *equations*. Door AGs have, in addition, *conditions* which allow the membership of elements in collection objects to be expressed.

IV *Local dependencies*. In a standard AG, non-local dependencies have to be expressed by a chain of local dependencies. There is no way to express a direct non-local dependency. In contrast, the use of references in Door AGs allows direct non-local dependencies to be expressed, simply by accessing the contents of an object denoted by a reference.

V *Rigid dependencies*. In a standard AG, all attribute dependencies are completely governed by the form of the syntax tree. The attribute dependencies of a Door AG are more flexible in that they may be governed also by attribute *values*. This will be discussed in greater detail in Chapter 9. The dependencies in Door AGs are also less rigid than those of standard AGs because they allow an attribute to depend on a small part of a structure, rather than on the structure as a whole. This is possible since structures can be built out of objects, and an attribute can depend on an attribute of another object. In a standard AG, all structures must be regular values, and an attribute must depend on the whole value.

VI *Uniquely-defined*. Whereas standard non-circular AGs are always uniquely-defined, a Door AG can be *underdetermined* in useful ways, allowing history-dependent error reporting to be implemented.

One of the most important advantages of Door AGs compared to standard AGs is the fine granularity of definition which can be obtained. It is possible to let the equations and conditions define a very small amount of information each. This is in contrast to standard AGs where one is forced to let some equations define very large information structures. The finer granularity in Door AGs allows the size of AFFECTED (the set of affected attributes after a syntactic change) to be dramatically decreased for some important problems and therefore makes it possible to implement much more efficient incremental evaluators.

In comparing the example of §8.9 with a usual standard AG for the same language, there are some important similarities and differences to be noted. First, the use of the search path objects resembles the use of “environment” attributes often used in standard AGs. New path/environment attributes are computed at each block level and, using inherited attributes, propagated down throughout the statement part to reach name applications. The important difference here is that the path attributes are *references* whereas the environment attributes of standard AGs have large complex *regular values*, including all possibly interesting information about all visible declarations. This leads to dramatic differences for the size of AFFECTED after a change. Consider changing the name of a declaration. The path attributes in the Door AG example are unaffected by this change since they still denote the same search path objects. In a corresponding standard AG, on the other hand, *all* environment attributes containing information about the changed declaration are affected. The size of AFFECTED is here proportional to the size of the syntax subtree which is within the scope of the changed declaration, and may thus be very large.

Another difference between Door AGs and standard AGs is in how environments can be constructed. In the Door AG example a reference to the symbol table object is propagated by inherited attributes from the block to the declarations. Each declaration can then individually define, by using a condition, that its entry is a member of the symbol table list. This makes the declarations independent of each other, and syntactic changes to one declaration does not affect other declarations. In a standard AG an environment is typically built by arranging the declarations in a list and using partial environment attributes for accumulating the total environment information. This makes the declarations dependent on each other and increases the size of AFFECTED.

If name declarations and applications may occur in any order, a standard AG needs several partial attributes to build up the total information in “passes”. The resulting standard AG is complicated, low-level, and the use of “passes” increases the grammar complexity, for example from 1-visit to OAG. The Door AG, on the other hand, is completely insensitive to the order of declarations. Dependencies between name declarations and name applications are handled in the same way regardless of where in the syntax tree the name applications occur.

To sum up, solving incremental name analysis in Door AGs instead of in standard AGs leads to:

- Dramatic decrease of the size of AFFECTED after changes to declarations. For example, from a number proportional to the size of the syntax tree to a very small number, close to zero. (This is the case for adding a global declaration of a name not previously used in the program.)

- Simpler local attribute dependencies, for example a reduction from OAG to 1-visit.
- Simpler specification without intermediate “pass” attributes whose values are uninteresting in the final attribution.

8.12 Summary

We have introduced Door Attribute Grammars and given a simple example to show how they can be used for specifying visibility-graph based name analysis. Although this example showed only how to define visibility graphs for simple block structure, it is straight-forward to extend the example to define more complex visibility graphs such as those needed for object-oriented languages. This will be discussed in greater detail in Chapter 11. In this chapter it will also be shown how doors can be used as an interface mechanism towards an external environment such as a window system or other tools in a programming environment.

A Door AG is an extension of a standard AG. These extensions can be summarized as follows:

- A syntax node can be extended with part-objects. A part-object owned directly by a syntax node is called a *door*, and transitively owned objects are called *semantic objects*.
- A semantic object can be specified as a *collection*, meaning that it is a collection of member objects, and the members are defined non-locally by *conditions*.
- Attributes may be *references*. I.e. they may have object identity values, denoting other nodes, doors, or semantic objects.

These extensions allow attributions containing objects and references to be specified in a straight-forward way. The use of objects and references allows visibility graphs to be modelled explicitly and allows the best incremental name analysis methods to be implemented. The price for using references is that non-local attribute dependencies are introduced which prevent evaluators to be generated completely automatically from the grammar. The solution to this problem has been to separate those parts of a Door AG which can be treated automatically (the *main grammar*) from those which require manual treatment (the *door package*). The following two chapters explain how evaluators for Door AGs can be implemented.

Chapter 9
Door AG Implementation, part I

This chapter and the next one describe how an evaluator for Door AGs can be implemented, based on an incremental visit-oriented technique. This chapter describes dependency analysis while the next one describes the evaluation algorithm and the construction of visit procedures.

9.1 Evaluation principle

The attribute evaluator is implemented as a global object with operations to be called by the editor. Basic operations are: replace a subtree, insert/delete a subtree in a list, and evaluate a whole new syntax tree. We will only discuss the replace subtree operation in detail since the other operations can be seen as special cases of this operation.

It is the task of the evaluator to update the attribution after each change to the syntax tree. The evaluator performs this update by calling visit procedures in the syntax nodes and door objects. A visit procedure may propagate the evaluation by calling visit procedures of other syntax nodes and door objects. For the syntax nodes, this propagation is always local, from node to node along the syntax tree. The door objects, on the other hand, may propagate the evaluation by calling visit procedures of other door objects, located anywhere in the syntax tree.

Figure 9.1 illustrates this. The user replaces the type of a variable declaration. The editor handles this change by calling the replace operation of the evaluator with the old and the new type subtrees as arguments. The evaluator in turn calls a visit procedure of the syntax node at the replacement point. From this point, the evaluation is propagated along the syntax tree into the `DeclDoor` object connected to the variable declaration. From the `DeclDoor` the evaluation propagates directly to the `UseDoor` objects for the name applications using the variable. From these objects, the evaluation propagates into the syntax tree where the type attributes of the expressions using the variable are updated.

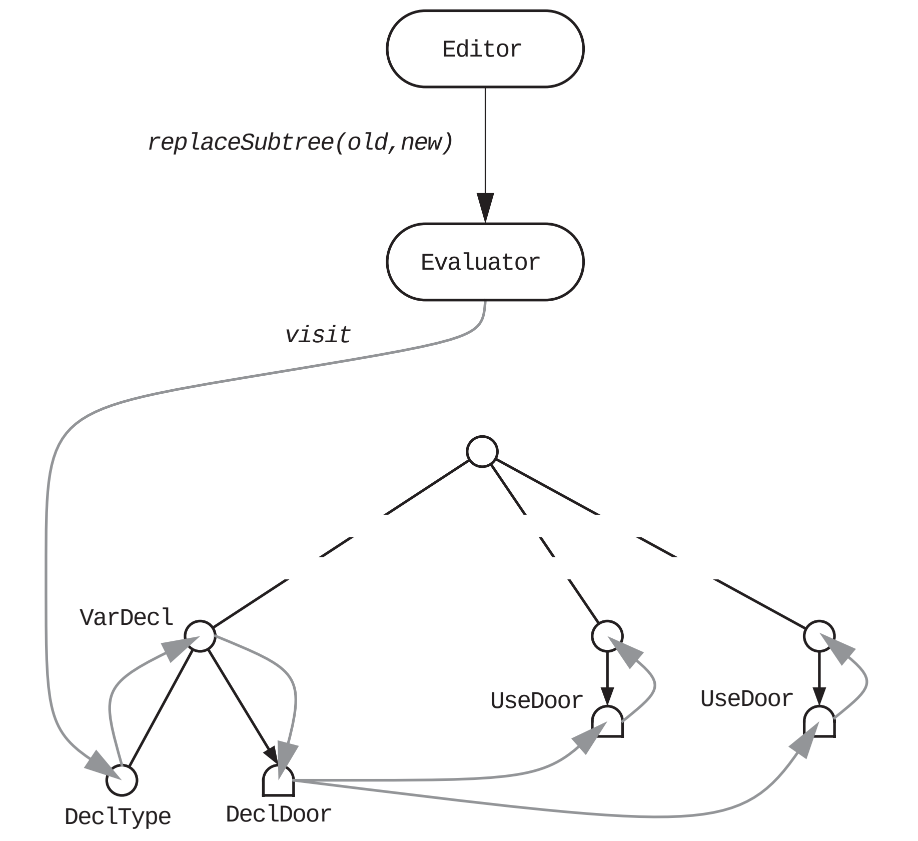


Figure 9.1 Propagation of evaluation by visit procedures

This is a slightly simplified description: the evaluator actually does a little more than calling just one visit procedure. There are also several types of visit procedures which are called under different circumstances. Nevertheless, this description contains the essence of the evaluation process. Implementation of an incremental evaluator for a Door AG consists of implementing a number of visit procedures for each node class and door class in the grammar.

9.2 Implementation steps

The separation of a Door AG into a main grammar and a door package is very important from an implementation point of view: Those parts of a grammar which can be treated automatically are isolated from those which require manual implementation. This allows door packages to be viewed as tool sets which extend standard AGs. Advanced facilities for common problems in static-semantics such as name analysis, type checking, and error detection, can be implemented in a door package which is used by many main grammars describing different languages.

Main grammars can be implemented by adapting existing techniques for standard AGs. We will base the implementation on the 1-visit technique for standard AGs described in Chapter 7, but it is also possible to use other algorithms. The implementation of main grammars is described in §10.8.

The implementation of door packages is substantially more complicated. We have developed a manual, but systematic, method for implementation. It involves analysis of the non-local dependencies which occur between door objects, and the design of information structures allowing the dependent doors to be located efficiently at evaluation time. After the dependency analysis, the construction of visit procedures for the door classes is comparatively straight-forward.

The door dependency analysis is treated in this chapter. As part of this analysis, additional so called *dependency attributes* are added to the door classes. These attributes define the additional information structures which allow dependent doors to be located efficiently. This explicit addition of dependency attributes is one of the keys to the generality and efficiency of Door AGs. The door package implementor is free to choose which dependency attributes to use and it is therefore possible to achieve performance close to hand coded and to tune the space and time consumption as desired. The dependency attributes are defined in the same way as the normal attributes, i.e. by equations or conditions. The addition of such attributes may add new non-local dependencies, and the construction of dependency graphs is thus an iterative process.

Chapter 10 describes the details of the evaluator and how the door visit procedures are constructed. The global evaluator object is general in that it contains no specific information about the door package. It only assumes the existence of a few virtual visit procedures in the node and door classes.

Figure 9.1 illustrates the steps involved in specifying and implementing a Door AG. Here we have split the door package into an interface and a body. The interface contains only the door class declarations with their inherited and synthesized attributes, as in the example of §8.9.3. This is the only information needed in order to write and implement a main grammar using the door package. During the dependency analysis, the door classes are extended with dependency attributes, dependency functions, and procedures which implement evaluation and de-evaluation of the conditions. The dependency functions and condition procedures are called by the visit procedures which are added later.

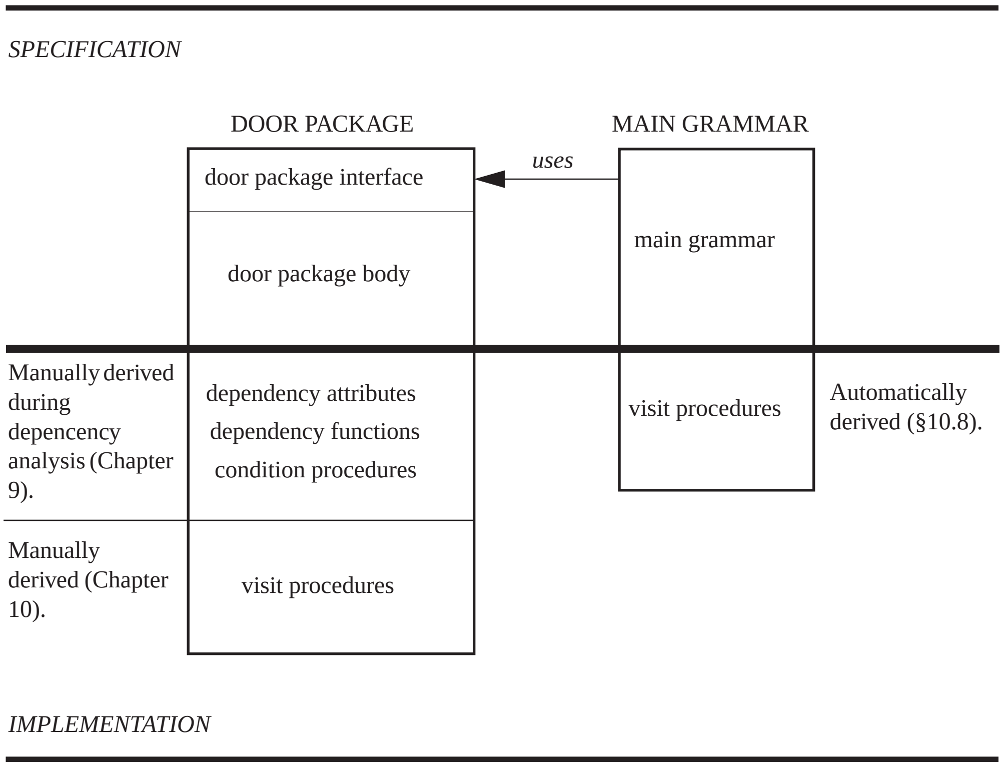


Figure 9.2 Specification and implementation of Door AGs

9.3 Dependencies

9.3.1 Invariants and evaluation

We will use the term *invariant* to mean either an equation or a condition. An invariant is in one of the two states `unevaluated` or `evaluated`. All invariants of a new syntax node or door object are initially in the `unevaluated` state. An invariant which is in the `evaluated` state is either *consistent* or *inconsistent*:

9-1 **Definition** Consistent invariants

- An equation which is in the `evaluated` state is said to be *consistent* if its defined attribute has the same value as the right hand side of the equation.
- A condition which is in the `evaluated` state is said to be *consistent* if the condition expression has the value *true*.

An invariant which is in the `evaluated` state, but is not consistent, is said to be *inconsistent*.

end 9-1

This definition is a straight-forward extension of the notions of “available” and “consistent” for attributes of standard AGs (see §4.2). Let *e* be an equation defining the value of an attribute *a*. If *e* is *consistent*, this corresponds the attribute *a* being *consistent* in the standard AG terminology. If *e* is in the `evaluated` state, this corresponds to *a* being *available* in the standard AG terminology.

Invariants are subject to evaluation, de-evaluation, and re-evaluation. With *evaluation* of an invariant we mean executing actions to make the invariant consistent. For an equation this means evaluating the right-hand side and assigning it to the defined attribute. For a condition, a special *evaluation procedure* has to be implemented which (possibly conditionally) inserts an element into a collection.

Invariants can be evaluated, de-evaluated, and re-evaluated. With *evaluation* of an invariant we mean executing actions to make the invariant consistent. For an equation this means evaluating the right-hand side and assigning it to the defined attribute. For a condition, a special *evaluation procedure* has to be implemented which (possibly conditionally) inserts an element into a collection.

With *de-evaluation* of an invariant we mean executing actions to remove possible earlier side-effects from the previous evaluation. For conditions, a special *de-evaluation procedure* has to be implemented which (possibly conditionally) removes an element from a collection. For equations, the evaluation has no side-effects other than setting the value of the defined attribute. No explicit de-evaluation operation is therefore needed for equations. Nevertheless, it is sometimes useful to consider the de-evaluation of an equation as an implicit action which makes the value of the defined attribute inaccessible.

With *re-evaluation* of an invariant we simply mean a de-evaluation followed by an evaluation.

Evaluation may only be applied to invariants in the `unevaluated` state and brings the invariant into the `evaluated` state. Conversely, de-evaluation may only be applied to invariants in the `evaluated` state, and brings the invariant into the `unevaluated` state. The figure below depicts these legal state transitions:

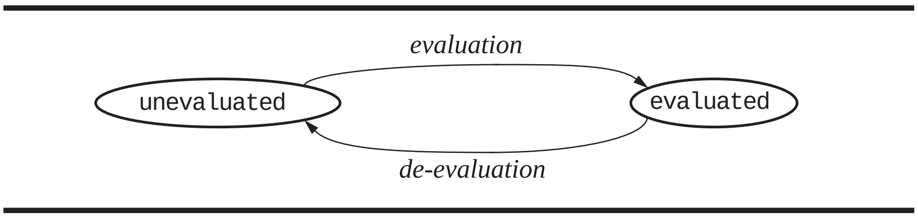


Figure 9.3 State transition diagram for invariants

9.3.2 Dependencies and events

9-2 **Definition** Dependency

A *dependency* is a directed relation (*a*, *b*) from an invariant *a* to another invariant *b*, meaning that the evaluation or de-evaluation of *a* may cause *b* to become inconsistent. The invariant *b* is said to be *dependent* on *a*.

end 9-2

A dependency will also be written (*x.a*, *y.b*) meaning that *a* and *b* are invariants of the objects *x* and *y* respectively. The notation (*x.a*, *Y.b*) refers to the *set* of dependencies outgoing from *x.a* which end in an invariant *y.b* such that *y* is an object of the class *Y*.

The evaluation or de-evaluation of an invariant will be referred to as an *event*. An event is written (*action*, *e*), where *action* is an evaluation action, either `eval` for evaluation or `deeval` for de-evaluation, and *e* is an invariant. The dependency set of an event (*action*, *e*) is the set of invariants which may become inconsistent if *action* is applied to the invariant *e*. If the dependency sets of (`eval`, *e*) and (`deeval`, *e*) differ, the invariants in the (`eval`, *e*) set are said to be *eval-dependent* on *e*, and the invariants in the (`deeval`, *e*) set are said to be *deeval-dependent* on *e*.

In analogy to definition 8-2 in §8.6, a dependency (*a*, *b*) is said to be *local* if *a* and *b* are declared in the same or neighbor objects in the extended syntax tree. Otherwise the dependency is said to be *non-local*. Likewise, an event (*action*, *x.a*) is said to be *non-local* to an object *y* if *x* is an object non-neighbor to *y*.

From §8.6 we recall that non-local dependencies can occur only between door objects. For non-local dependencies we distinguish between *static dependencies* and *evaluation-time dependencies*. A static dependency is a dependency (*X.a*, *Y.b*) where *X* and *Y* are door classes. For an object *x* of class *X*, the static dependency corresponds to a set of *k* evaluation-time dependencies {(*x.a*, y_1*.b*) . . (*x.a*, y_n*.b*)}, $n \geq 0$, where y_1 . . y_n are objects of class *Y*.

Note that we consider dependencies as occurring between invariants. This is in contrast to most techniques for standard AGs which consider dependencies as occurring between attributes.

However, this difference is purely technical since there is a one-to-one correspondence between attributes and equations. For convenience, we will use the terms “attribute” and “equation” interchangeably in connection to evaluation and dependencies. E.g., “evaluation of an attribute *a*” is equivalent to “evaluation of the equation defining *a*”. Likewise, “the attribute *a* is dependent on the attribute *b*” is equivalent to “the equation defining *a* is dependent on the equation defining *b*”.

9.3.3 Actual dependencies

The usual way to decide if an event can cause another invariant to become inconsistent is to analyze accesses. In §8.6 we defined access-dependencies for attributes. However, access-dependencies are sometimes unnecessarily pessimistic. In particular, this can be the case for certain non-local dependencies. Suppose an attribute *a* is access-dependent on another attribute *b*. By taking attribute values into account, it is in some cases possible to deduce from the attribute definitions that a given event (*action*, *b*) will only cause a subset of the access-dependent invariants to actually become inconsistent. If this is the case, the invariants in the subset are said to be *actually dependent* on *b*:

9-3 **Definition** Actual dependency set

Let (*action*, *b*) be an event with the access-dependency set *P*. If it can be deduced from the Door AG that this event will cause only a certain subset *Q* of *P* to become inconsistent, then *Q* is referred to as the *actual dependency set*. Each invariant in *Q* is said to be *actually dependent* on *b*.

end 9-3

Thus, an access-dependency does not always imply an actual dependency. This can be utilized in order to avoid unnecessary re-evaluation of access-dependent invariants.

For example, the lookup function used in defining the `binding` attribute of a `UseDoor` (in the example of §8.9) may search through many symbol tables and access the `ident` attributes of many `Entry` objects. The `binding` attribute in a `UseDoor` can thus be access-dependent on very many `ident` attributes. However, changing any of these `ident` attributes will only cause the binding to become inconsistent if it is changed to the same identifier as the one of the `UseDoor`. Thus, for a given change to an `ident` attribute, the set of actually dependent `binding` attributes is only a subset of the set of access-dependent `binding` attributes. By considering actual dependencies instead of access-dependencies, many unnecessary re-evaluations of `binding` attributes can be avoided.

9.3.4 Non-local dependencies

An invariant in a door class which accesses mutable information in non-neighbor objects has incoming non-local dependencies. Such invariants are called *receiving invariants*. Invariants defining information which is accessed non-locally have outgoing non-local dependencies. Such invariants are called *sending invariants*. During incremental evaluation, changes are propagated from sending invariants to receiving invariants.

To perform this non-local change propagation it is necessary to have some mechanism for finding the set of receiving invariants for each static dependency. I.e., given a static dependency ($X.a$, $Y.b$) and an object x of class X, there is a need for a way of obtaining the set $\{y_1, .. y_n\}$ of objects of class Y for which there are evaluation-time dependencies ($x.a$, $y_k.b$).

This problem is solved by adding *dependency functions* to the door class *X*. Each dependency function returns a set of door objects for a given receiving invariant. The dependency functions make use of ordinary attributes to compute the set of dependent doors. In order to perform this computation efficiently, additional attributes usually need to be added to the grammar.

9.3.5 Conditions

In the implementation of a door package, each condition of a door class *D* is associated with an *evaluation procedure* and a *de-evaluation procedure*. These procedures are located in the class *D* and may access attributes of *D* and its part-objects. However, the de-evaluation procedure will be called by the evaluator at times when non-fix inherited attributes have inconsistent values. Therefore, the de-evaluation procedure must not access such attributes. For simplicity, we restrict both the evaluation procedure and the de-evaluation procedure to only use local attributes and fix inherited attributes.

The access of attributes in the evaluation and de-evaluation procedures cause access-dependencies to conditions:

9-4 **Definition** Access-dependencies to conditions

Let *c* be a condition. For each attribute *b* accessed by the evaluation or de-evaluation procedures for *c*, there is an *access dependency* from *b* to *c*, and *c* is said to be access-dependent on *b*.

end 9-4

A condition may also have dependent invariants. Let *t* be a collection object and c a condition which defines (possibly conditionally) a member in *t*. Invariants accessing information in the collection *t* may be dependent on the condition *c*. An invariant accessing information in a collection usually accesses many of the member elements, but only in order to find one of them with certain properties. For these dependencies it is therefore usually useful to consider actual dependencies rather than access-dependencies.

9.3.6 Local copy attributes

In some cases, the values of non-fix inherited attributes need to be stored in the door to be available at a later time. One example of this was in order to de-evaluate conditions, as was discussed in §9.3.5. Such de-evaluation must be done using the same information as the previous evaluation, and the values of non-fix inherited attributes used at the previous evaluation must therefore be stored in the door. We do this simply by adding a *local copy attribute*, i.e. a local attribute which is defined to have the same value as the non-fix inherited attribute.

If a local copy attribute is added, we require all invariants in the door which depend on the copied attribute to be revised to use the local attribute instead of the inherited one. This insures that all the invariants in the door have been computed using the same inherited information at any given time during evaluation. This simplifies the dependency analysis.

9.4 Door dependency graphs

In dependency analysis of a door package, we build one dependency graph per door class. This dependency graph shows the local dependencies between invariants of the door, as well as non-local dependencies to and from invariants in other doors.

A door dependency graph contains one *inherit* vertex *v*(*a*) for each inherited attribute *a*, one *synthesize* vertex *v*(*b*) for each synthesized attribute *b*, and one *local* vertex *v*(*c*) for each local attribute *c* or condition *c* defined by the door. In addition, the graph contains *send* and *receive* vertices modelling the non-local dependencies. A receive vertex *v*(*L*) has a label *L*. The receive vertex is connected to a number of vertices for receiving invariants (often only one). A send vertex is denoted *v*(*L*, *D*, *f*) where *L* is the label of a receive vertex for the door class *D* and *f* is a dependency function returning a set of *D* objects. A send vertex represents a set of non-local dependents, namely the set of invariants reachable from *v*(*L*) in the door objects of class *D* returned by the function *f*.

The following graphical symbols will be used in door dependency graphs:

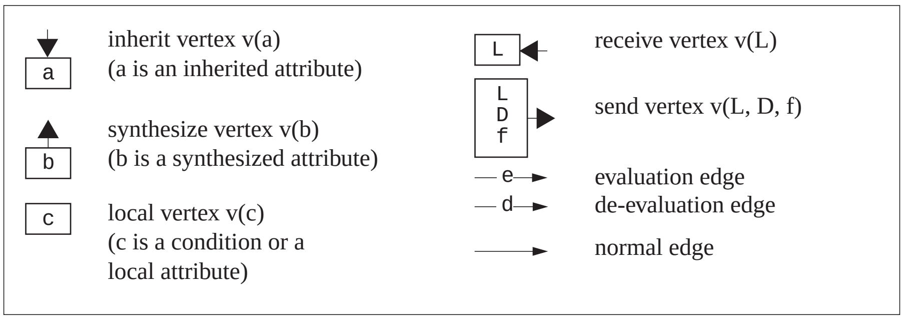


Figure 9.4 Symbols used in dependency graphs

Dependencies are represented in the graphs by edges. An *evaluation edge* (*a*, *b*), labelled by `e`, represents an eval dependency. This means that there is a dependency from *a* to *b* if the invariant *a* is evaluated. A *de-evaluation edge* (*a*, *b*), labelled by `d`, correspondingly represents a de-eval dependency and means that there is a dependency from *a* to *b* if the invariant *a* is de-evaluated. A *normal* un-labelled edge is equivalent to both an evaluation edge and a de-evaluation edge.

9.4.1 Unordered collections

In order to implement efficient dependency functions, additional attributes can be added to the grammar. Often, it is useful to add a collection object which keeps track of dependent doors as an unordered collection. We will make use of the following collection class for this purpose:

```
UnorderedCollection: class[T: class ANYCLASS]
{   contains: func boolean(e: ref T);
    contents: func ref Set[T];
    add: proc(e: ref T);
    remove: proc(e: ref T);
};
```

The class `UnorderedCollection` models a collection of references to `T` objects. It is very similar to the class `Set`, but in contrast to `Set` it is not an applicative class. I.e., its set of references may change as a consequence of changes to the syntax tree. However, the function `contents` returns a corresponding set value (a reference to a `Set` object). The set value obtained by a call at one point in time cannot change even if elements are later added to or removed from the collection. This is important since the result of the dependency function must be a set value which must not be affected by subsequent evaluation in the syntax tree.

The `add` and `remove` procedures can be used to implement evaluation and de-evaluation procedures for conditions defining elements in the collections.

9.4.2 A simple example of dependency graphs

As a simple example, we will show how dependency graphs can be constructed for the following door package:

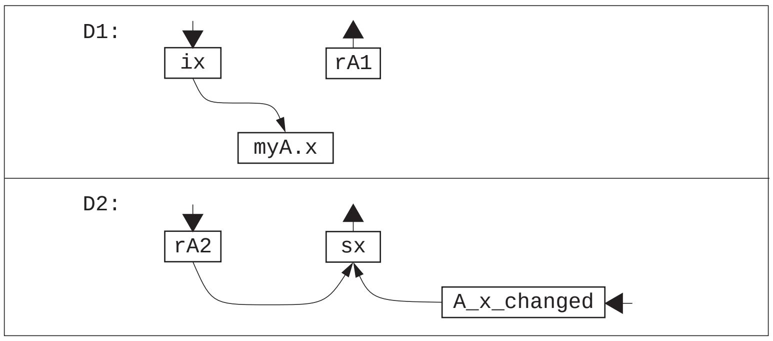
```
A: class
{   loc x: integer;
};

D1: door
{   inh ix: integer;
    syn rA1: ref A;
    myA: object A;
    eq myA.x := ix;
    eq rA1 :- myA;
};

D2: door
{   inh rA2: ref A;
    syn sx: integer;
    eq sx := rA2.x;
};
```

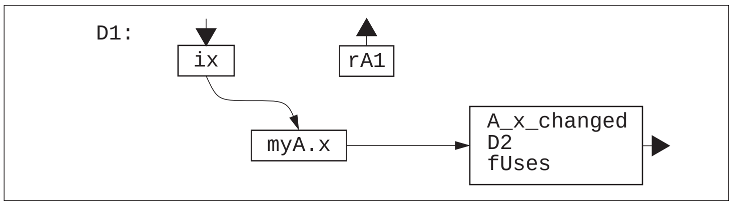
This is the same example door package as of §8.6.

The first step in the construction is to add vertices for invariants and inherited attributes and to add edges for local dependencies. In addition, for each receiving invariant, a receive vertex is inserted. This yields the following graphs:



The class D2 has one receiving invariant, namely the equation defining sx. This equation uses non-local information: it uses the x attribute in the A object denoted by the inherited reference rA2. This dependency is represented by a receive vertex labelled A_x_changed.

The next step is to add send vertices matching the receive vertices. In this case, the receive vertex represents a static dependency (D1.myA.x, D2.sx). The D1 door should therefore have a send vertex *v*(A_x_changed, D2, fUses) where fUses is a function which computes the set of affected D2 doors. The resulting graph for D1 is the following:



The next step is to define the function fUses of class D1. This function should return a set of references to all D2 doors for which the rA2 attribute denotes the myA object of the D1 door. One possibility would be to search the whole syntax tree for finding D2 objects fulfilling these criteria. A more efficient solution is to add dependency attributes which keep track of the D2 objects. This can be done by adding a collection object to D1 and let each D2 object register in the proper collection object by using a condition. The function fUses can then be defined simply as the set contents of the collection object:

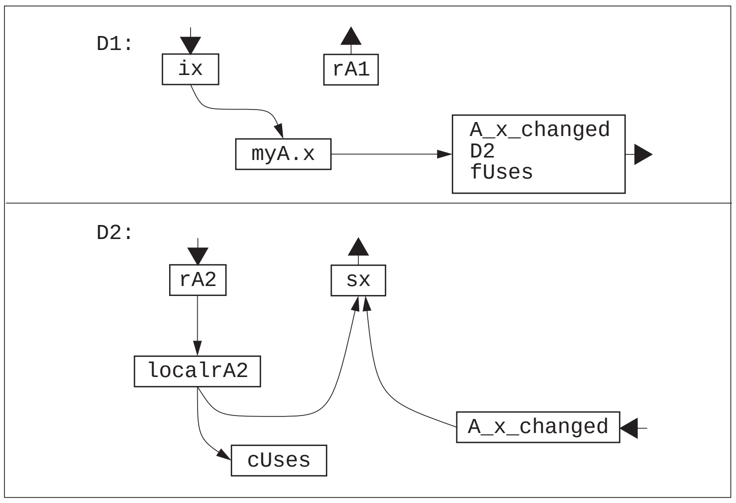
```
addto D1
{   collection uses: object UnorderedCollection[D2];
    fUses: func ref Set[D2] :- uses.contents;
};

addto D2
{   loc localrA2: ref A2; (* local copy attribute *)
```

```
        eq localrA2 :- rA2;
        eq sx := localrA2.x;(* revised to use local copy *)
        cUses: cond localrA2.uses.contains(this D2);
        evalCUses: proc
        {   localrA2.uses.add(this D2);
        };
        deEvalCUses: proc
        {   localrA2.uses.remove(this D2);
        };
    };
```

The two procedures `evalCUses` and `deEvalCUses` are the evaluation and de-evaluation procedures for the condition `cUses`. Since these procedures may not use the non-fix inherited attribute `rA2` directly, a local copy attribute has been added according to §9.3.6. The equation defining `sx` is also revised to use this local copy. Since the new invariant `cUses` has been added to `D2`, the dependency graph for `D2` must be revised accordingly. This implies adding a new vertex *v*(`cUses`) and a dependency from *v*(`rA2`) to *v*(`cUses`).

The resulting dependency graphs for the door package are as follows:



9.4.3 Restricted use of attributes in dependency functions

There are certain restrictions on which attributes may be used in the implementation of a dependency function. The dependency functions of a door object are always called *prior* to the evaluation or de-evaluation of invariants in the object. Depending on the current evaluation state, the attributes of the door object will be un-evaluated or evaluated, consistent or inconsistent. Which attributes are allowed to be used can be deduced from the door dependency graph by analyzing the incoming dependencies to the send vertices using the dependency function.

Consider a send vertex *v*(`L`, `D2`, `f`) of a door class `D1`. The function `f` is then a function in `D1`. The implementation of `f` can access any collection part-objects in `D1`, but access to *attributes* of `D1` and its part-objects is subject to certain restrictions. The restrictions depend on the kind of dependency edges ending in *v*(`L`, `D2`, `f`). Recall that an edge can be an evaluation edge, a de-evaluation edge, or a normal edge (equivalent to both an evaluation and a de-evaluation edge). The following cases apply:

I There is an evaluation edge (*v*(`a`), *v*(`L`, `D2`, `f`)) where `a` is an inherited attribute or a local invariant. In this case, `f` will be called by the evaluator when no local attributes in the door or its semantic part-objects have yet been evaluated. In this case, `f` must not access these attributes. It may, however, access inherited attributes if no other restrictions apply.

II There is a de-evaluation edge (*v*(`a`), *v*(`L`, `D2`, `f`)) where `a` is an inherited attribute or local invariant, or there is an edge (*v*(`L`), *v*(`L`, `D2`, `f`)) where *v*(`L`) is a receive vertex. In this case, `f` will be called by the evaluator when the door is fully evaluated and when all invariants are consistent, except for those depending on non-fix inherited attributes. In this case, `f` must not access any non-fix inherited attribute. It may, however, access all local attributes of the door and its semantic part-objects and all fix inherited attributes, if no other restrictions apply.

For example, the function `fUses` in the example of §9.4.2 is used in a send vertex *v*(`A_x_changed`, `D2`, `fUses`). Since this vertex has an incoming normal edge (*v*(`myA.x`), *v*(`A_x_changed`, `D2`, `fUses`)), both the above restrictions apply. The implementation of `fUses` is therefore not allowed to access inherited attributes or local attributes of `D1` and its part objects. It may, however, access the part-objects themselves. The implementation of `fUses` accesses only the collection object `uses`. Thus, the restrictions above are adhered to.

9.4.4 Construction of dependency graphs

The construction of the door dependency graphs involves design decisions such as which send and receive vertices to add and how to implement the dependency functions. The construction is in principle an iterative process since the addition of new attributes to implement the dependency functions efficiently may lead to additional non-local dependencies. Although we provide no complete formal algorithm for constructing the door dependency graphs, it is possible to construct them in a systematic manner as follows:

9-5 **Construction** Door dependency graphs

I *Implement conditions*. For each condition, implement evaluation and de-evaluation procedures. Possibly add new local attributes to perform these operations without using non-fix inherited attributes.

II *Local analysis*. For each door class *D*, construct a dependency graph *DG*(*D*) with one inherit vertex for each inherited attribute, one synthesize vertex for each synthesized attribute, one local vertex for each local attribute defined by an equation in *D*, and one local vertex for each condition in *D*. Do local dependency analysis on the invariants of *D* and add edges corresponding to local access-dependencies. If an invariant r accesses non-fix non-local information, add a receive vertex with a new label L and add an edge from $v(L)$ to $v(r)$. The invariant r is said to be a *receiving invariant*.

III *Add send vertices*. Consider each added receive vertex $v(L)$ and its corresponding receiving invariant r of a door class D_r. Analyze the door package to find the set of invariants S whose evaluation or de-evaluation can cause a non-local instance of r to become inconsistent. Consider each invariant s in S. Let D_s be the door class of s. Construct a send vertex $v(L, D_r, f)$ where f is a dependency function returning the appropriate set of D_r doors. Declare f in D_s if it is a new function (implementation can wait until step V). Add the vertex $v(L, D_r, f)$ to the dependency graph of D_s, unless such a vertex already exists in the graph. If evaluation (de-evaluation) of s can cause r to become inconsistent, add an evaluation (de-evaluation) edge from $v(s)$ to $v(L, D_r, f)$. (If both evaluation and de-evaluation can cause inconsistency, add a normal edge instead.)

IV *Simplify send/receive vertices*. Consider two receive vertices $v(L_1)$ and $v(L_2)$ of a door class D_r. Suppose there is a door class D_s with two send vertices $v(L_1, D_r, f)$ and $v(L_2, D_r, f)$. Two edges (x_1, y_1), (x_2, y_2) are said to be *equivalent* if $x_1{=}x_2$ and if the edges have the same label. If the sets of incoming edges to the two send vertices are equivalent, then simplification of the dependency graphs is possible as follows: A new receive vertex $v(L_3)$ is added to D_r and an edge $(v(L_3), x)$ is added for each edge $(v(L_1), x)$ or $(v(L_2), x)$. The two send vertices are then collapsed and replaced by a send vertex $v(L_3, D_r, f)$. Each of the receive vertices $v(L_1)$ and $v(L_2)$ is removed unless it is referred to by another send vertex.

V *Implement dependency functions*. Implement the dependency functions used by the send vertices. If this involves addition of attributes and invariants, go back to step I and repeat the construction for the added attributes and invariants.

end 9-5

9.5 Analysis of example door package

We will now construct the dependency graphs for the door package of §8.9 according to construction 9-5.

9.5.1 Implement conditions (step I)

Step I is to implement evaluation and de-evaluation procedures for the conditions in the door classes. In the example door package there is one condition: `registered` in `DeclDoor` (§8.9.2.3) which states the membership of an `Entry` object in a `List` object:

```
DeclDoor: door
{   ...
    registered: cond table.entries.contains(theEntry);
    ...
};
```

To implement this condition, we assume that the class of `table.entries`, i.e. `List`, has two procedures `add` and `remove`. The procedure `add` adds an element to the end of the list and the procedure `remove` removes an element from the list:

```
addto List
{   add: proc(e: ref T);
    remove: proc(e: ref T);
};
```

The condition can then be implemented by extending the `DeclDoor` class by the following two procedures:

```
addto DeclDoor
{   evalRegistered: proc
    {   table.entries.add(theEntry);
    };
    deEvalRegistered: proc
    {   table.entries.remove(theEntry);
    };
};
```

Since the inherited attribute `table` is fix, this is in agreement with the rules for attribute access in evaluation/de-evaluation procedures as stated in §9.3.5. If `table` had been non-fix, an additional local attribute would have had to be added to `DeclDoor`, and the evaluation procedures be revised to use this attribute instead of `table`.

9.5.2 Local analysis (step II)

Step II is to analyze the local access-dependencies for the door classes and to identify the invariants depending on non-local information. This analysis for the four doors `RootDoor`, `BlockDoor`, `DeclDoor`, and `UseDoor` of §8.9.2 results in the dependency graphs shown in Figure 9.5.

For example, consider the invariants of `UseDoor` (§8.9.2.4):

```
UseDoor: door
{   ...
    eq binding :-                          (* 1 *)
            if ident = “”
            then nullEntry
            else
                inspect $E :- path.lookup(ident)
                when Entry do $E
                otherwise nullEntry;
    eq tp :- binding.getTp;                (* 2 *)
    eq declared := binding =/= nullEntry;  (* 3 *)
};
```

The equation defining `binding (* 1 *)` depends locally on the attributes `ident` and `path`. In addition, the `binding` depends on what is returned by the function `lookup`. This is a non-fix function using non-local information and a receive vertex *v*(`lookupChanged`) is added to reflect this non-local dependency.

The equation defining `tp (* 2 *)` depends locally on `binding` and non-locally on what is returned by the non-fix function `getTp`. A receive vertex *v*(`getTpChanged`) is added to reflect this non-local dependency.

For equation `(* 3 *)` there is only a local dependency from `binding` to `declared`.

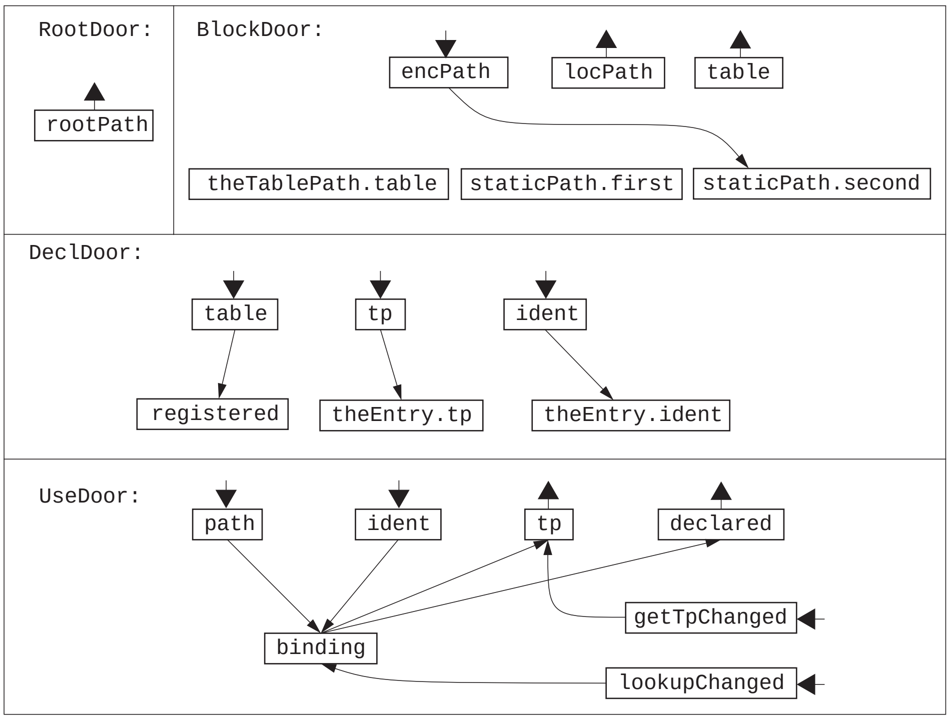


Figure 9.5 Dependency graphs after step II

9.5.3 Add send vertices (step III)

In this step, send vertices are added to match the receive vertices *v*(`getTpChanged`) and *v*(`lookupChanged`).

9.5.3.1 Send vertices for getTpChanged

The receive vertex *v*(`getTpChanged`) for `UseDoor` represents non-local events which may affect the result of the function call `binding.getTp` and thereby cause the `tp` attribute of `UseDoor` to become inconsistent. The function `getTp` has two implementations: one in `nullEntry` and one in `Entry`. The implementation in `nullEntry` returns a constant value and can therefore not lead to any non-local dependencies. The implementation of `getTp` in `Entry` (see §8.9.1.2) depends on the `tp` attribute of class `Entry`. This attribute is defined in `DeclDoor` by the equation

```
eq theEntry.tp :- tp;
```

There is therefore a static dependency

(`DeclDoor.theEntry.tp`, `UseDoor.tp`)

For a given `DeclDoor` object `d`, the dependent objects are those `UseDoor` objects whose `binding` attribute denotes `d.theEntry`. To find these objects at evaluation time, we add a dependency function `fUses` to `DeclDoor`:

```
addto DeclDoor
{   fUses: func ref Set[UseDoor];
}
```

We defer the implementation of this function until §9.5.5.

A new send vertex *v*(`getTpChanged`, `UseDoor`, `fUses`) and an edge from *v*(`theEntry.tp`) to this send vertex are then added to the dependency graph of `DeclDoor` (see the edge marked by `(1)` in Figure 9.6).

9.5.3.2 Send vertices for lookupChanged

The receive vertex *v*(`lookupChanged`) for `UseDoor` represents non-local events which may affect the result of the function call `path.lookup` and thereby cause the `binding` attribute of `UseDoor` to become inconsistent.

According to the definitions of classes `SymbolTable` and `SearchPath` (§8.9.1.3, §8.9.1.4), the result of a call to `lookup` of a `SearchPath` object `p` depends on the following non-local information:

1. which `SymbolTable` objects are found via `p`
2. which `Entry` objects are found in the lists of these `SymbolTable` objects
3. the values of the `ident` attributes in these `Entry` objects
4. the order of the `Entry` objects in the lists

We will now consider each of these sources of non-local dependencies in turn.

Symbol table objects

The SymbolTable objects found via a given SearchPath object can in fact not change in this door package since all connections between SearchPath objects are defined using fix information. This is seen from the definition of BlockDoor (§8.9.2.2).

Entry objects

The Entry objects found in the list of a SymbolTable object may change due to evaluations and de-evaluations of the condition registered in DeclDoor. There is thus a static dependency

(DeclDoor.registered, UseDoor.binding).

The dependency sets of evaluating and de-evaluating this condition are different. Let d be a DeclDoor object. If the condition registered for d is *de-evaluated*, the d.theEntry object is removed from the symbol table. This invalidates the binding in UseDoor objects whose binding attribute denotes d.theEntry. This is exactly the set of objects returned by the fUses function of DeclDoor which was defined to handle the getTpChanged dependency. We use the same function to construct a send vertex *v*(lookupChanged, UseDoor, fUses) which is added to the dependency graph of DeclDoor. A *de-evaluation edge* is added from *v*(registered) to this send vertex (edge (2) in Figure 9.6).

In *evaluating* the condition registered of d, the d.theEntry object is added to the symbol table. This affects the binding attribute of another set of UseDoor objects. Consider computing the binding attribute of a UseDoor object u. The u.path.lookup function looks for Entry objects in a series of symbol tables until either a matching entry is found in a symbol table s_n, or there are no more symbol tables on the path. Let $s_1 \,..\, s_{n-1}$ be the symbol tables where no matching entry was found. We say the UseDoor object has *attempted* to bind to these symbol tables. If an entry object e is added to any of these symbol tables, and e.ident = u.ident, then this will make u.binding inconsistent. Evaluating the condition registered of d will thus affect all UseDoor objects u which have attempted to bind to d.table and for which u.ident = d.theEntry.ident. To find these objects at evaluation time, we add a dependency function fAttempted to DeclDoor:

```
addto DeclDoor
{   fAttempted: func ref Set[UseDoor];
}
```

Again, the implementation of this function is deferred to §9.5.5.

A new send vertex *v*(lookupChanged, UseDoor, fAttempted) and an evaluation edge from *v*(registered) to this send vertex are then added to the dependency graph of DeclDoor (edge (3) in Figure 9.6).

Ident values

The value of the `ident` attribute of an `Entry` object is defined by the equation

```
    eq theEntry.ident := ident
```

in `DeclDoor`. Evaluation or de-evaluation of this equation has the same effect as the evaluation or de-evaluation of the `registered` condition. *De-evaluating* the equation means making the old value of `ident` inaccessible, and will affect the `binding` attribute of the `UseDoor` objects returned by the `fUses` function. *Evaluating* the equation means making a new value accessible and this will affect the `binding` attribute of the `UseDoor` objects returned by the `fAttempted` function. Thus, the dependency graph of `DeclDoor` is updated by adding a de-evaluation edge from *v*(`theEntry.ident`) to *v*(`lookupChanged`, `UseDoor`, `fUses`) and an evaluation edge from *v*(`theEntry.ident`) to *v*(`lookupChanged`, `UseDoor`, `fAttempted`) (edges `(4)` and `(5)` in Figure 9.6).

Entry order

The order of the entry objects in a symbol table list matters only if there are namesake declarations in the block, i.e. if two or more entries have the same `ident` value. In this case, the `lookup` function will return the first of these entries. For the present discussion, we assume that this does not occur. We will return to the issue of namesake declarations in Chapter 11.

Step III in the dependency graph construction has thus resulted in three new send vertices in the graph for `DeclDoor` as shown below:

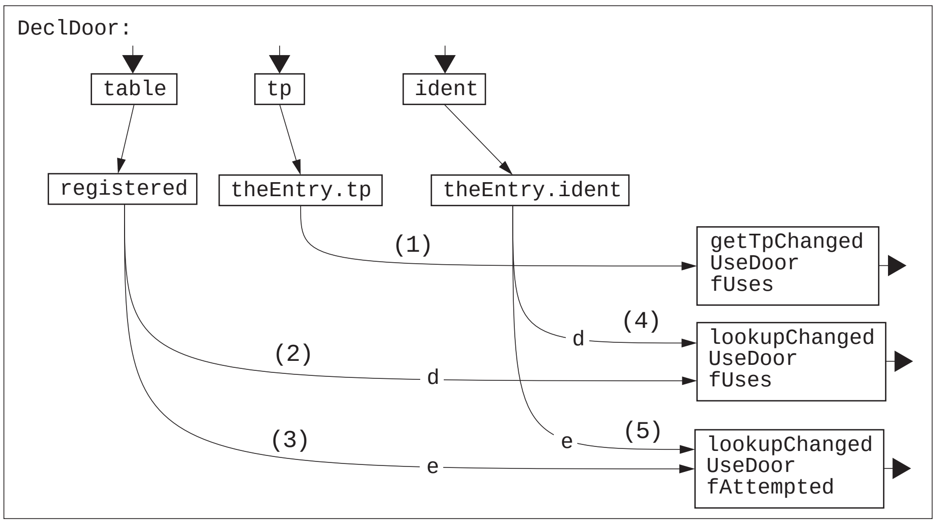

DeclDoor:
table
tp
ident
registered
theEntry.tp
theEntry.ident
(1)
getTpChanged UseDoor fUses
(4)
d
lookupChanged UseDoor fUses
(2)
d
(3)
e
(5)
e
lookupChanged UseDoor fAttempted

Figure 9.6 Send vertices added to DeclDoor graph

9.5.4 Simplify send / receive vertices (step IV)

No simplification of send/receive vertices is possible in our example: The two send vertices *v*(getTpChanged, UseDoor, fUses) and *v*(lookupChanged, UseDoor, fUses) fulfill the first criterion of two matching vertices $v(L_1, D_r, f)$ and $v(L_2, D_r, f)$, but the second criterion of equivalent incoming edges for these vertices is not fulfilled.

9.5.5 Implement dependency functions (step V)

In this step, the two dependency functions fUses and fAttempted declared in DeclDoor are implemented.

9.5.5.1 The function fUses

The function fUses of class DeclDoor should return the set of UseDoor objects whose binding attribute denotes the Entry object of the DeclDoor. To implement this function we add a collection object uses to class Entry which keeps track of these UseDoor objects. The collection object is declared as an UnorderedCollection (see §9.4.1). A condition cUses is added to UseDoor to define which UseDoor objects are members of which collections. The fUses function can then be defined simply as the set contents of the uses object:

```
addto Entry
{   collection uses: UnorderedCollection[UseDoor];
};

addto DeclDoor
{   impl fUses :- theEntry.uses.contents;
};

addto UseDoor
{   cUses: cond
        inspect $B :- binding
        when Entry do $B.uses.contains(this UseDoor)
        otherwise true;
};
```

The condition defines the membership conditionally: the UseDoor is declared as a member of a uses collection only if binding actually denotes an Entry object. The binding attribute might denote the nullEntry object (which is not of class Entry) in which case the UseDoor will not be member of any uses collection. The “inspect”-expression used in the condition does case analysis on binding to separate these two cases.

The function fUses occurs in the send vertex *v*(getTpChanged, UseDoor, fUses) which has an incoming normal dependency edge, corresponding to both an evaluation and a de-evaluation edge. Therefore, the implementation of fUses may not make use of local or inherited attributes, but only of collection objects (as stated in §9.4.3). This requirement is met by the above implementation.

The evaluation and de-evaluation procedures for the cUses condition can be implemented by using the add and remove procedures of class UnorderedCollection:

```
addto UseDoor
{   evalCUses: proc
    {   inspect $B :- binding
        when Entry do $B.uses.add(this UseDoor)
        end inspect;
    };
    deEvalCUses: proc
    {   inspect $B :- binding
        when Entry do $B.uses.remove(this UseDoor)
        end inspect;
    };
};
```

These procedures use the “inspect”-statement to conditionally add and remove the UseDoor from the collection. The procedures use no inherited attributes at all, so this is in agreement with the rules for attribute access in evaluation/de-evaluation procedures as stated in §9.3.5.

9.5.5.2 The function fAttempted

Consider a DeclDoor object d. The function fAttempted of d should return the set of UseDoor objects which have attempted to bind to d.table, and for which the ident attribute is equal to a certain value *v*. Recall that fAttempted is called in connection to the evaluation of the condition registered and the evaluation of the attribute theEntry.ident. The value *v* is the new value of the attribute theEntry.ident. However, the function fAttempted is called before any evaluation starts in the DeclDoor. At this point in time, the inherited attributes are available, but the attribute theEntry.ident is not (or contains an old value). The new value *v* can instead be found in the inherited attribute d.ident.

To implement the function fAttempted we will make use of a class Dictionary with the following interface:

```
Dictionary: class [T: class ANYCLASS]
{
    associationAt: func ref T(key: string);
                                     (* may return NONE *)
};
```

A dictionary associates objects with strings. The function associationAt returns the object associated with a given string. If there is no object associated with the string, the function returns NONE. The objects we will associate with strings will be of class UnorderedCollection[UseDoor].

A collection object attempted of class Dictionary will be added to class SymbolTable. The function associationAt(ident) will return an unordered collection of UseDoor objects which have attempted to bind to the SymbolTable using the identifier ident. It is then straight-forward to implement the function fAttempted:

```
addto SymbolTable
{   collection attempted: object
        Dictionary[UnorderedCollection[UseDoor]];
};

emptyUseDoorSet: object Set[UseDoor];

addto DeclDoor
{   impl fAttempted :-
        inspect $C :- table.attempted.associationAt(ident) do
        when UnorderedCollection[UseDoor] do $C.contents
        otherwise emptyUseDoorSet;
};
```

Since the function `associationAt` may return `NONE`, an inspect-expression is used to take care of this case. If the function returns an `UnorderedCollection` object, the "when" clause applies, and the set contents of that object is returned. If the function returns `NONE`, the "otherwise" clause applies and the constant semantic object `emptyUseDoorSet` is returned.

The function `fAttempted` occurs only in the send vertex *v*(`lookupChanged`, `UseDoor`, `fAttempted`) which has only an incoming evaluation edge. According to the rules of §9.4.3 the implementation of `fAttempted` may therefore use inherited but not local attributes. As seen above, the implementation uses the `table` and the `ident` attributes which are both inherited.

The contents of the collection `attempted` is defined by a condition `cAttempted` in `UseDoor`. The evaluation procedure of `cAttempted` should add the `UseDoor` to the `attempted` collection of each symbol table occurring on its path, up to but not including the symbol table containing the entry which the `UseDoor` is bound to. To implement the evaluation and de-evaluation procedures some additional attributes are needed. We add a local attribute `table` to class `Entry`, making it possible to find the symbol table of an entry object in an efficient way:

```
addto Entry
{   loc table: ref SymbolTable;
};

addto DeclDoor
{   eq theEntry.table :- table;
};
```

Further, we add two local copy attributes `localPath` and `localIdent` to `UseDoor` to be able to implement the `cAttempted` condition without using the non-fix inherited attributes `path` and `ident` (in order to adhere to the rule in §9.3.5):

```
addto UseDoor
{   loc localPath: ref SearchPath;
    loc localIdent: string;
    eq localPath :- path;
    eq localIdent := string;
};
```

The definition of the condition `cAttempted` and the implementation of the evaluation and de-evaluation procedures is straight-forward, but results in rather lengthy definition. For brevity, we have therefore hidden the details in the functions `f`, `g`, and `h` whose implementation has been left out.

```
addto UseDoor
{   cAttempted: cond
        if localIdent <> “”
        then f(localPath, localIdent, binding)
        else true;

    evalCAttempted: proc
    {   if localIdent <> “”
        then g(localPath, localIdent, binding);
    };

    deEvalCAttempted: proc
    {   if localIdent <> “”
        then h(localPath, localIdent, binding);
    };
};
```

9.5.5.3 Comment

The implementation of the dependency functions `fUses` and `fAttempted` corresponds exactly to solving the incremental name analysis problems I, II and III treated in §3.4.

The problems II and III (remove and change declaration) correspond to the *v*(`lookupChanged`, `UseDoor`, `fUses`) and *v*(`getTpChanged`, `UseDoor`, `fUses`) send vertices respectively. The `fUses` function solves these problems by the `uses` collection in each `Entry` object. This is an implementation of method 4 (maintain cross-references).

The problem I (add a declaration) corresponds to the *v*(`lookupChanged`, `UseDoor`, `fAttempted`) send vertex. The `fAttempted` function solves this problem by the `attempted` collection in each `SymbolTable` object. This is an implementation of method 6 (maintain traces).

The last problem IV (change the visibility graph) does not occur for this simple door package since all edges in the visibility graph (the `first` and `second` attributes of `TwoPath` objects) are defined using fix attributes and can thus not change.

An alternative implementation of `fUses` and `fAttempted` could have used the same “maintain traces” method for all three problems. This could have been done by defining the `attempted` collection of a symbol table to include also collections of the `UseDoor` objects bound to entries in the table.

It was argued in §3.4.1 that it can be motivated to trade space for time in implementing incremental name analysis. Some examples were given of how the space overhead could be reduced by not maintaining full trace information, and instead combine with some searching to find the affected name applications. This would correspond to another implementation of the `fAttempted` function and the `attempted` collection. An alternative implementation could let the `attempted` collection store only information about what blocks contain name applications attempting to bind to certain identifiers. The `fAttempted` function would then have to search the syntax trees of these blocks to find the affected name applications.

9.5.6 Repeat construction for added invariants

During step V, invariants were added to the door classes. The construction of the dependency graph therefore has to be repeated for these additions.

The following conditions were added:

- `cUses` of class `UseDoor`
- `cAttempted` of class `UseDoor`

and the following equations:

- `theEntry.table` of class `DeclDoor`
- `localPath` of class `UseDoor` (a local copy attribute)
- `localIdent` of class `UseDoor` (a local copy attribute)

The invariants in `UseDoor` are revised to use the local copy attributes instead of the inherited attributes directly, as discussed in §9.3.6.

Both of the conditions depend on non-local information. Following the steps of construction 9-5, we would add a receive vertex for each of the conditions and send vertices which match these receive vertices. However, it is easily seen that the conditions depend on exactly the same non-local information as the `binding` attribute. Therefore, simplification of send and receive vertices according to step IV lead to graphs with no new send and receive vertices. Instead, edges (*v*(`lookupChanged`), *v*(`cUses`)) and (*v*(`lookupChanged`), *v*(`cAttempted`) are added.

The vertices for the equations are straight-forward to add since they depend only on local information.

The final resulting dependency graphs are depicted below:

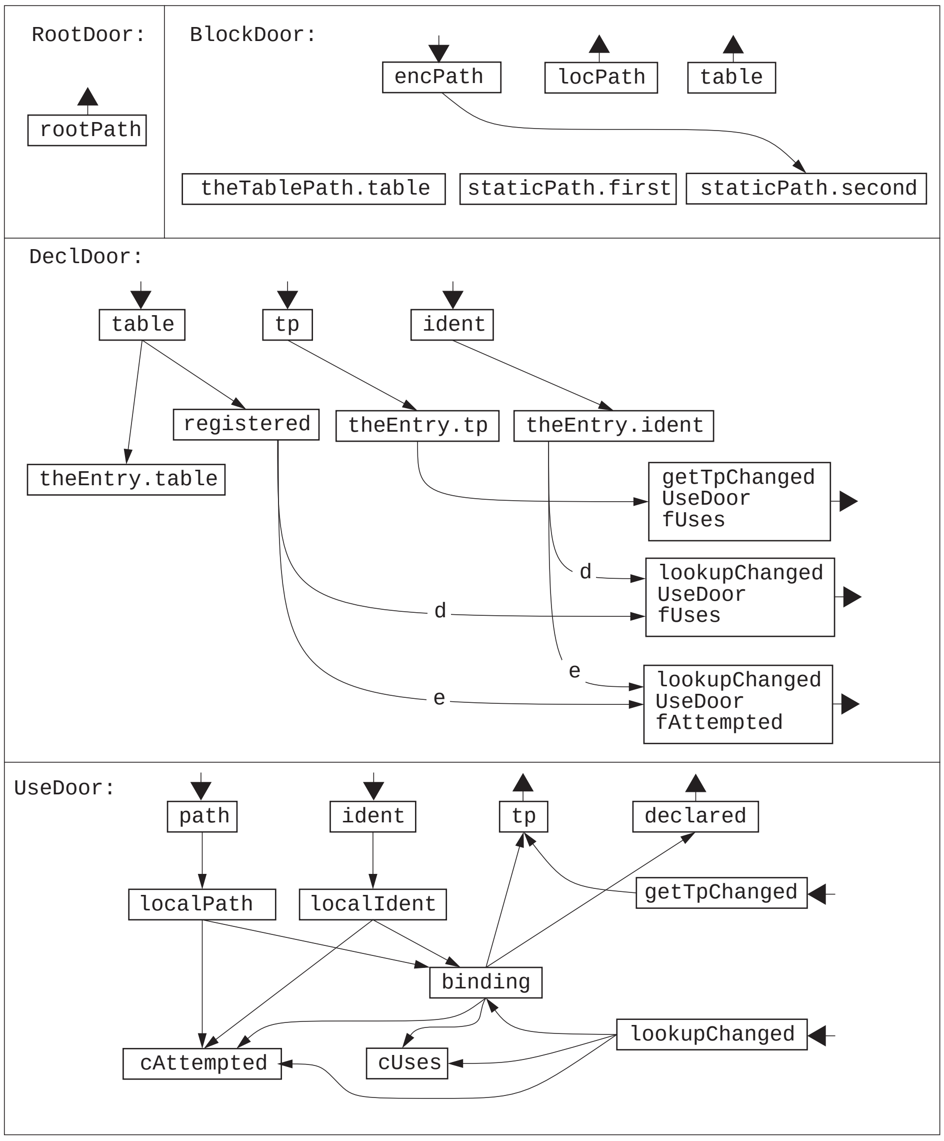


Figure 9.7 Final dependency graphs

9.6 Summary

We have described a technique for constructing dependency graphs for door packages. One graph is constructed for each door class. A graph shows the local dependencies between inherited attributes and local invariants of a door. It also shows the non-local dependencies to other doors. *Send vertices* represent outgoing non-local dependencies and *receive vertices* incoming non-local dependencies. At evaluation time, a send vertex corresponds to a *set* of receiving door objects of a given class. *Dependency functions* are added to the sending door classes and have the responsibility to return the set of receiving door objects. To implement the dependency functions efficiently, additional attributes must usually be added.

The technique for constructing dependency graphs is systematic but not automatic. The non-local dependencies are analyzed manually, and the implementation of the dependency functions is manual. The manual implementation of dependency functions makes it possible to trade space for time, in order to achieve desired performance. The local analysis is straight-forward to automate, and automatic or semi-automatic techniques also for the non-local analysis would be desirable. This is an area for future research.

As an example of dependency graph construction, the door package of §8.9 was analyzed. Dependency functions were constructed for this package to achieve incremental name analysis according to the best methods treated in §3.4.

Chapter 10
Door AG Implementation, part II

This chapter describes the incremental evaluation algorithm and the construction of visit procedures. A door package is implemented by extending each door class with a set of visit procedures, according to the door dependency graphs constructed in Chapter 9. A main grammar is implemented by extending the node classes with visit procedures, using the same technique as was used for standard AGs in Chapter 7. It is also shown how circular dependencies are handled.

10.1 Outline of evaluator algorithm

The Door AG evaluator is an extension to the evaluator for 1-visit standard AGs presented in §7.3. The basic editing operation is the same: a subtree replacement where an old subtree is replaced by a new completely unevaluated subtree.

The standard AG evaluator performs three steps to restore consistency. The first step is the actual syntactic subtree replacement and copying of attribute values for the inherited attributes at the replacement point. The second step is an exhaustive evaluation of the new subtree. The third step is an incremental evaluation, starting at the point of subtree replacement.

For a Door AG, this operation needs to be extended. First, the old subtree may contain conditions stating memberships in collection objects located in the rest of the tree. These conditions must be de-evaluated to remove the corresponding elements. Second, evaluation which propagates into doors may need to be propagated to other non-local doors, according to the send vertices. Third, evaluation which propagates non-locally to a door may propagate into the syntax tree again, via the synthesized attributes of the door.

To handle these additional issues, the Door AG evaluator works in five steps as follows:

10-1 **Outline** Door AG evaluator algorithm

I *Exhaustive de-evaluation phase*. The conditions in the doors of the old subtree are de-evaluated.

II *Subtree Replacement*. The old subtree is replaced by the new subtree. Each inherited data attribute of the root of the new subtree is assigned a value by copying the corresponding value from the root of the old subtree.

III *Exhaustive evaluation phase*. The new subtree is evaluated exhaustively.

IV *Local incremental phase*. Incremental evaluation proceeds in the syntax tree, starting at the successors of the synthesized attributes of the root of the new subtree.

V *Non-local incremental phase*. Incremental evaluation is started in the syntax tree at the successors of the synthesized attributes of each non-locally affected door.

end 10-1

Step I handles the de-evaluation of conditions in the doors of the old subtree. This step is done before the actual subtree replacement so the de-evaluation procedures of the conditions can access the inherited attributes of its doors. Recall that the inherited attributes of doors and all attributes in the syntax tree are implemented as demand attributes (by default). Thus, to access the inherited attributes of the doors, access to the remaining syntax tree may be necessary.

Step II performs the subtree replacement and is exactly the same as the corresponding step in the standard AG evaluator. However, the copying of inherited attributes is usually an empty operation since all attributes in the syntax tree are (by default) demand attributes.

Steps III and IV also correspond exactly to the exhaustive and incremental phases of the standard AG evaluator. During these steps, the evaluation may propagate from the syntax tree out to door objects. In this case, evaluation is propagated over to the receiving doors of non-local dependencies, but not back into the syntax tree at these receiving doors. Instead, references to the receiving doors are saved in a global work list so that evaluation can be propagated into the syntax tree at these doors at a later point in time.

Step V deals with the doors on the work list. For each of these doors, evaluation is propagated into the syntax tree to successor attributes of the synthesized attributes of the door. During this phase, the evaluation may again propagate out to door objects, and additional receiving door objects can be added to the work list. If a door object is already on the work list, it is not added again. Step V goes on as long as there are any doors left on the work list.

10.1.1 Scattered evaluations

We will refer to the evaluation starting at a specific point in the syntax tree as the execution of an *evaluation thread*. In step V of the evaluator outline, evaluation threads are started at door objects on the work list, i.e. at points which may be scattered all over the syntax tree. In our algorithm, these evaluation threads are executed in sequence. If a given attribute a depends on two door objects on the work list, this will lead to a being evaluated twice. In principle, the threads could be *coordinated* in order to avoid such duplicate evaluation.

This problem is analogous to the problem of *multiple subtree replacement*. Several methods have been proposed for dealing with this problem for standard AGs, e.g. [RMT86], [YK88], [Pec90b], [Vor90b], [FKT90] and the same methods could be applied to step V of our Door AG evaluator. However, the situation is slightly different in Door AGs than in standard AGs:

First, an evaluation thread in a Door AG is usually very short. Attributes are propagated from a door through the syntax tree to another door, but more seldom long distances in the syntax tree. This is in contrast to standard AGs where at least some evaluation threads are very long, passing through the entire syntax tree. Second, if evaluation is propagated via a non-local dependency to a door, the evaluation thread starting at that door is queued up on the work list. Thus, even if evaluation is propagated twice to a door via non-local dependencies, this will result only in one evaluation thread starting at that door. For these reasons, coordination is not as critical in Door AGs as in standard AGs. We have not found it motivated to improve the algorithm in this respect.

Consider the example in Figure 10.1.

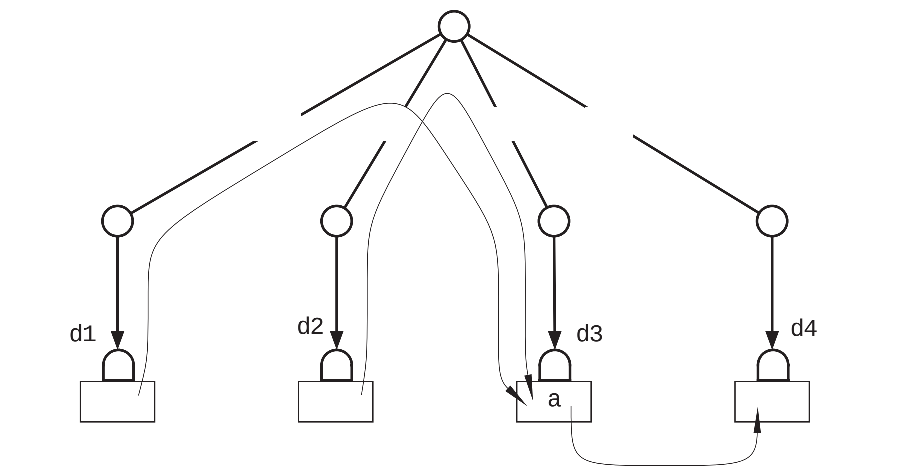


Figure 10.1 Multiple evaluation of the same attribute

The door `d3` is locally dependent on both `d1` and `d2`. The door `d4` is non-locally dependent on `d3`. Suppose the work list contains the two doors `d1` and `d2`. When executing the evaluation thread starting at `d1`, evaluation is propagated through the syntax tree to `d3`. The change may then propagate non-locally to `d4` which is added to the work list. When the evaluation thread of `d1` is finished, a new evaluation thread is started at `d2`. This evaluation also propagates to `d3`, and the change may again propagate to the non-locally dependent door `d4`. Since `d4` is already on the work list, it is not added again. Thus, the uncoordinated evaluation of the threads starting at `d1` and `d2` leads to multiple evaluation of the attributes in `d3`, but not to multiple evaluation of the thread starting at `d4`. At this point, the evaluation is again coordinated.

10.2 Types of visit procedures

A visit procedure schedules evaluation of local invariants with calls to visit procedures of other objects. For syntax nodes, the same types of visit procedures are used as described in §7.3: `exhVisit`, `incFatherVisit`, and `incSonVisit`. In addition, a syntax node can be visited from its door part-objects. This is implemented by an additional visit procedure `incDoorVisit` which is implemented analogously to `incSonVisit`. I.e., the procedure `incDoorVisit(d)` schedules evaluation of invariants dependent on the synthesized attributes of the door part-object `d`.

10.2.1 Visits to doors from syntax nodes

A door node can be visited from its owning syntax node during the exhaustive de-evaluation phase (step I), during the exhaustive evaluation phase (step III), or during one of the incremental phases (steps IV and V). These three types of visits are implemented by three visit procedures `exhDeEvalVisit`, `exhEvalVisit`, and `incOwnerVisit`:

- The procedure `exhDeEvalVisit` models a visit from the owning node during the exhaustive de-evaluation phase. The procedure de-evaluates all conditions in the door.
- The procedure `exhEvalVisit` models a visit from the owning node during the exhaustive evaluation phase. The procedure schedules evaluation of all invariants in the door.
- The procedure `incOwnerVisit` models a visit from the owning node during one of the incremental evaluation phases. The non-fix inherited attributes of the door may have new values, and the procedure schedules re-evaluation of their dependent invariants.

10.2.2 Visits to doors from other doors

A door may also be visited from other doors via non-local dependencies, according to the receive vertices of the dependency graph of the door. For each receive vertex *v*(`L`) a pair of visit procedures `deEvalL` and `evalL` are implemented:

- The procedure `deEvalL` schedules de-evaluation of all conditions in the door which depend on the receive vertex *v*(`L`).
- The procedure `evalL` schedules evaluation of all invariants in the door which depend on the receive vertex *v*(`L`).

10.2.3 Summary of visit procedures

To summarize, the visit procedures needed by all nodes and doors are the following:

```
addto ANYNODE
{   exhVisit: proc;
    incFatherVisit: proc;
    incSonVisit: proc (s: ref ANYNODE);
    incDoorVisit: proc (d: ref ANYDOOR);
};

addto ANYDOOR
{   exhEvalVisit: proc;
    exhDeEvalVisit: proc;
    incOwnerVisit: proc;
};
```

In addition, a receiving door class `D` with *n* receive vertices will have a pair of visit procedures for each of these vertices:.

```
addto D
{   deEvalL1: proc;
    evalL1: proc;
    ...
    deEvalLn: proc;
    evalLn: proc;
};
```

The construction of the visit procedures of door classes will be treated in detail in §10.4.

10.2.4 Extension to OOSL

The door visit procedures have a more complex structure than the visit procedures for standard AGs. In order to make it clearer which statements correspond to evaluations and which statements have other purposes, we introduce two new OOSL statements: **`eval`** and **`deeval`** for the evaluation and de-evaluation of invariants, respectively. The syntax is as follows:

```
<eval-stmt> ::= ‘eval’ (<attr-id> | <cond-id>)
<deeval-stmt> ::= ‘deeval’ <cond-id>
```

The **`eval`** statement is shorthand for the assignment statement evaluating an attribute, or a call to the evaluation procedure of a condition.

The **`deeval`** statement is shorthand for the de-evaluation procedure of a condition.

For example, consider the following equation and condition:

```
eq a := f(s,t)
c: cond q.contains(r)
```

A de-evaluation of `c`, followed by an evaluation of `a` and `c` can be written using deeval and eval statements as follows:

```
deeval c
eval a
eval c
```

This is equivalent to the following statements (given that the evaluation and de-evaluation procedures for `c` are named `evalC` and `deEvalC` respectively):

```
deEvalC;
a := f(s,t);
```

```
evalC;
```

10.3 The evaluator object

Our Door AG evaluator is implemented as a global object called `evaluator`. The `evaluator` contains a procedure `replaceSubtree` to be called by the editor when a given subtree is to be replaced. The procedure `replaceSubtree` both actually replaces the subtree and restores consistency in the attribution of the whole syntax tree. The `evaluator` object also contains the work list of door objects to be evaluated during the non-local incremental phase. Between calls to `replaceSubtree`, the work list is empty.

10.3.1 Work list implementation

The work list is implemented as an `OrderedCollection` object with the following interface:

```
OrderedCollection: class[T: class ANYCLASS]
{   empty: func boolean;
    add: proc(e: ref T);
    addSet: proc(s: ref Set[T]);
    removeIfFound: proc(e: ref T);
    removeFirst: proc ref T;
};
```

This class models an ordered collection of `T` objects. The function `empty` returns true if the collection is empty. The procedure `add` adds an object `e` at the end of the collection (unless the object is already in the collection). The procedure `addSet` adds each of the objects in a set at the end of the collection (except those already in the collection). The procedure `removeIfFound` removes a given object from the collection if it is found there. The procedure `removeFirst` removes the first object and returns a reference to it.

The evaluator also makes use of `Set` objects. The interface to class `Set` (whose interface was given in Figure 5.1) is extended with an iterator `each` which iterates over all the elements in the set:

```
addto Set
{   each: iterator ref T;
};
```

10.3.2 Functionality in nodes and doors

The evaluator assumes some additional functionality in syntax nodes and doors:

```
addto ANYNODE
{   father: ref ANYNODE; (* NONE for the root node *)
    replaceBy: proc(n: ref ANYNODE);
    copyInheritedAttributesFrom: proc(n: ref ANYNODE);
    allDoors: func ref Set[ANYDOOR];
};

addto ANYDOOR
```

```
{   owner: **ref** ANYNODE;
};
```

The reference `father` of a syntax node denotes the father node, or `NONE` in case of the root node. The procedure `replaceBy` replaces the subtree rooted at the node by another subtree rooted at node `n`. The procedure `copyInheritedAttributesFrom` assigns a value to each of the inherited data attributes in the node, using the values of the corresponding attributes in `n`. This procedure will be empty for most node classes since attributes in the syntax nodes are usually demand attributes. The function `allDoors` returns a set of references to all doors owned by any of the nodes in the subtree rooted at the node.

The reference `owner` of a door object denotes the owning syntax node.

10.3.3 Evaluation algorithm

The `evaluator` object contains a procedure `replaceSubtree` which can be called by the editor. We assume that the editor calls this procedure only for syntactically legal replacements and never for the root syntax node. It is also assumed that the attribution is already in a consistent state when `replaceSubtree` is called. The procedure replaces the subtree and updates the attribution incrementally to a new consistent state.

This definition of the `replaceSubtree` procedure is a straight-forward implementation of the algorithm outlined in 10-1. The only detail which was not mentioned earlier is the statement during step I which removes a door from the work list. The reason for this statement will be explained later in §10.5.3.

```
evaluator: object
{ worklist: object OrderedCollection[ANYDOOR];

  replaceSubtree: proc
    (oldNode: ref ANYNODE, newNode: ref ANYNODE)
  { d: ref ANYDOOR;

    (* I: Exhaustive de-evaluation phase *)
    for $d :- oldNode.allDoors.each do
      worklist.removeIfFound($d);
      $d.exhDeEvalVisit;
    end for;

    (* II: Subtree Replacement *)
    oldNode.replaceBy(newNode);
    newNode.copyInheritedAttributesFrom(oldNode);

    (* III: Exhaustive evaluation phase *)
    newNode.exhVisit;

    (* IV: Local incremental phase *)
    newNode.father.incSonVisit(newNode);

    (* V: Non-local incremental phase *)
    while not worklist.empty do
      d :- worklist.removeFirst;
      d.owner.incDoorVisit(d);
    end while;
  };
};
```

Figure 10.2 Subtree replacement algorithm

10.4 Construction of door visit procedures

10.4.1 Basic outline for door visit procedures

The visit procedures for the door classes are constructed from the dependency graphs and they all use the same basic algorithm outline. The visit procedures schedule de-evaluation and/or evaluation of local invariants. In addition, they call visit procedures of dependent doors and add these doors to the work list of the evaluator. Let *D* be a door class. The basic outline for the visit procedures of *D* is as follows:

10-2 **Algorithm outline** Basic door visit procedure

1. Compute the set of dependent doors.
2. Call `deEvalL` for each dependent door.
3. Local de-evaluation of conditions in *D*.
4. Local evaluation of invariants in *D*.
5. Call `evalL` for each dependent door.
6. Add the dependent doors to the work list of the evaluator.

end 10-2

The above outline applies directly to the three visit procedures present in all door classes: `exhEvalVisit`, `exhDeEvalVisit`, and `incOwnerVisit`. The `deEvalL` and `evalL` visit procedures can be seen as two halves of the same visit. These procedures are implemented by dividing the above pattern, so the `deEvalL` procedure implements steps 1 to 3 and the `evalL` procedure the steps 4 to 6.

The `deEvalL` procedures are called *before* local evaluation, to allow the non-locally dependent doors to de-evaluate conditions using the old attribute values in *D*. The `evalL` procedures are called *after* local evaluation to allow the non-local dependents to use the new attribute values. Notice that a (`deEvalL`, `evalL`) procedure pair may have its own non-local dependents, and these may have additional non-local dependents, and so on. As can be seen from the algorithm outline, all non-locally dependent conditions, including transitive ones, will be de-evaluated before any evaluation of invariants starts.

The example door package of §8.9 for Algol-like block structure does not contain any transitive non-local dependencies. However, one example of such dependencies will be given in §11.2, which treats subclassing.

10.4.2 Visit procedure characteristics

Each visit procedure for a door *D* can be characterized by three sets of vertices from the dependency graph for *D*. In this context we consider each procedure pair (`deEvalL`, `evalL`) for a receive vertex v(`L`) as one unit. The three *characteristic sets* are the following:

- a set of condition vertices S_{cond}. (With "condition" vertex we mean a local vertex $v(c)$ for a condition c.)
- a set of local vertices S_{loc}
- a set of send vertices S_{send}

In step 3 of the basic outline above, the visit procedure should de-evaluate the conditions corresponding to S_{cond}. The order of de-evaluation is arbitrary since the conditions are independent of one another. In step 4 the visit procedure should evaluate the invariants corresponding to S_{loc}. These evaluations should be done in topological order according to the dependency graph. In step 1 the visit procedure should compute the set of dependent doors according to the dependency functions of the send vertices in S_{send}. In steps 2 and 5 the `deEval` and `eval` procedures for the appropriate receive vertices should be called for these non-local doors, and in step 6 the doors should be added to the evaluator work list. The characteristic sets for the different kinds of visit procedures are summarized in the table below.

	S_{cond}	S_{loc}	S_{send}
`exhDeEval-Visit`	all condition vertices	$\varnothing$	all send vertices with incoming de-eval edge reachable from a condition vertex
`exhEval-Visit`	$\varnothing$	all local vertices	all send vertices with incoming eval edge
`incOwner-Visit`	all condition vertices reachable from vertices for inherited non-fix attributes	all local vertices reachable from vertices for inherited non-fix attributes	all send vertices reachable from vertices for inherited non-fix attributes
`deEval/ eval L`	all condition vertices reachable from the receive vertex v(`L`)	all local vertices reachable from the receive vertex v(`L`)	all send vertices reachable from the receive vertex v(`L`)

Figure 10.3 Visit procedure characteristics

The basic algorithm for a visit procedure outlined in 10-2 can now be formulated more precisely in terms of the characteristic sets:

10-3 **Algorithm** Basic door visit procedure

Given the characteristic sets S_{cond}, S_{loc}, and S_{send} for a door visit procedure *p* of a door class *D*, the algorithm of *p* is as follows:

1. Let v(`L`$_1$, `D`$_1$, `f`$_1$) . . v(`L`$_n$, `D`$_n$, `f`$_n$) be the vertices in S_{send}. Compute *n* dependency sets `DP`$_k$, $1 \le k \le n$, of non-locally dependent doors by calling the corresponding dependency function:

   ```
   DP_k :- f_k;
   ```

   where `DP`$_k$ is a local variable in *p* (or in *D* for `deEval`/`eval` pairs) declared as

   ```
   DP_k: ref Set[D_k];
   ```

2. For each dependency set `DP`$_k$, $1 \le k \le n$, call the appropriate `deEval` procedure for each of its doors:

   ```
   for $d :- DP_k.each do $d.deEvalL_k; end for
   ```

3. For each condition `c` in S_{cond}, de-evaluate `c`:

   ```
   deeval c;
   ```

4. For each invariant `e` in S_{loc}, evaluate `e`:

   ```
   eval e;
   ```

   The invariants should be evaluated in topological order according to the dependency graph.

5. For each dependency set `DP`$_k$, $1 \le k \le n$, call the appropriate `eval` procedure for each of its doors:

   ```
   for $d :- DP_k.each do $d.evalL_k; end for
   ```

6. Add each dependency set `DP`$_k$, $1 \le k \le n$, to the work list of the evaluator:

   ```
   evaluator.worklist.addSet(DP_k);
   ```

end 10-3

Note that in step 1, the local variables for the dependency sets are normally declared in the visit procedure. This works for the `exhDeEvalVisit`, `exhEvalVisit`, and `incOwnerVisit` procedures. However, it does not work for a `deEval`/`eval` pair since the variables are computed in the `deEval` procedure and used in the `eval` procedure. For this case, the dependency set variables are instead stored in the door itself. An example of this is given in §11.2.7 (§Figure 11.19).

The characteristic sets for a given door class *D* and the basic algorithm for a given visit procedure *p* can be computed automatically from the dependency graph of *D*. However, there are some additional issues which need to be taken into account and which may call for modifications to the basic algorithm above. In particular:

- The dependency sets `DP`$_k$, $1 \le k \le n$, may overlap.
- Attributes can be tested for convergence to avoid unnecessary change propagation.
- Additional code can be added to affect which of several consistent solutions is chosen, in the case of underdetermined grammars.

To deal with these issues, the basic algorithm of 10-3 has to be modified. This will be treated in §10.6.

10.5 Door evaluation states

During evaluation, a door can be in one of the following states: `InNewTree`, `Evaluated`, `DeEvaluated(L)`, `InOldTree`, and `Busy`. In each of these states, the following holds for the states of the invariants in the door (we recall from §9.3.1 that an invariant can be either in the `uneval-uated` or `evaluated` state):

- When a door is in the `InNewTree` state, no invariants of the door have yet been evaluated. I.e., all the invariants are in the `unevaluated` state.
- When a door is in the `Evaluated` state, all its invariants are also in the `evaluated` state.
- When a door is in the `DeEvaluated(L)` state, where *v*(`L`) is a receive vertex of the door dependency graph, the conditions depending on *v*(`L`) of the door are in the `unevaluated` state, whereas the rest of the invariants in the door remain in the `evaluated` state.
- When a door is in the `InOldTree` state all its conditions are in the `unevaluated` state.

The evaluation states for the doors serve as pre- and post-conditions for the visit procedures according to the following state transition graph. The door is in the `Busy` state during each of these transitions:

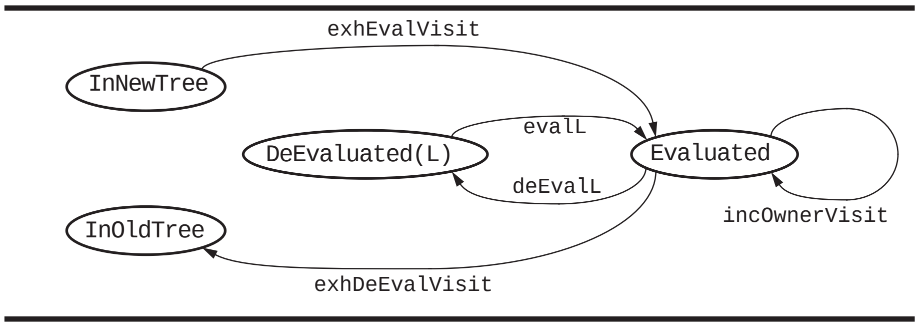


Figure 10.4 Evaluation states for doors

We will now discuss under what circumstances these preconditions are fulfilled.

10.5.1 Calls from the evaluator algorithm

In a consistently attributed syntax tree, all door objects are in the `Evaluated` state. At a subtree replacement, the doors in the new subtree are initially in the `InNewTree` state. The evaluator algorithm of 10-1 should terminate by leaving all the doors in the old subtree in the `InOldTree` state and all doors in the modified syntax tree in the `Evaluated` state.

We observe that the net effect of executing one of the procedures `exhDeEvalVisit`, `exhEvalVisit`, and `incOwnerVisit` of a door does not change the state of other doors. This is clear from the basic visit procedure algorithm 10-2 since a `deEvalL` call is always matched by an `evalL` call.

It is now easy to verify that the evaluator algorithm of 10-1 fulfills the preconditions of the called visit procedures and that the algorithm terminates with all doors in the `Evaluated` state:

Step I brings all doors in the old subtree from the `Evaluated` state to the `InOldTree` state by calling the `exhDeEvalVisit` procedure of these doors. Step III brings all doors in the new subtree from the `InNewTree` state to the `Evaluated` state by calling the `exhEvalVisit` procedure of these doors. After this step, all doors in the syntax tree are in the `Evaluated` state. During the subsequent incremental evaluation steps IV and V, the visit procedures of the syntax nodes will call the `incOwnerVisit` procedure of dependent doors. Calling this procedure for a door does not change its door state (other than temporarily to `Busy` during the call), and all doors in the syntax tree are therefore in the `Evaluated` state when the evaluation algorithm terminates.

10.5.2 Calls from inside visit procedures

Each door visit procedure must fulfill the preconditions of the `deEvalL` and `evalL` procedures it calls. We will now investigate the requirements this puts on the implementation of the visit procedures. We do this by considering the incremental and exhaustive evaluation phases separately.

Incremental phases

First, consider evaluation during one of the incremental phases IV or V in the evaluation algorithm 10-1. During these phases, all door objects are in the `Evaluated` state except for during the call of the `incOwnerVisit` procedure of a door.

Let d be a door object visited via the `incOwnerVisit` procedure during one of these phases. Let $\mathtt{DP}_1(d)\,..\,\mathtt{DP}_n(d)$ be the sets of non-locally dependent doors, according to step 1 of the basic algorithm 10-3. Clearly, these sets must be disjoint. Otherwise, a `deEvalL` procedure would be called for a door already in the `DeEvaluated` state which would violate the precondition of the `deEvalL` procedure. Similarly, if there are transitive non-local dependencies, the sets $\mathtt{DP}_1(d)\,..\,\mathtt{DP}_n(d)$ must be disjoint to all the dependency sets $\mathtt{DP}_1(d')\,..\,\mathtt{DP}_n(d')$ where d' is a door non-locally dependent on d (directly or transitively). Furthermore, the door object itself, d, must not be a member of any of these dependency sets. Otherwise, a `deEvalL` procedure would be called for d while in its `Busy` state.

For any dependent door $d' \neq d$ which occurs in only one of the direct and transitive dependency sets, the algorithm 10-3 fulfills the preconditions: Step 2 calls `deEvalL` when d' is in the `Evaluated` state, bringing it into the `DeEvaluated` state. Step 2 later calls the corresponding `evalL` procedure, bringing d' back to the `Evaluated` state.

Thus, if all the direct and transitive dependency sets are disjoint and *d* is not a member of any of them, then the preconditions of the called visit procedures are fulfilled.

Exhaustive phases

Now, consider the exhaustive phases I and III. During these phases some doors will be in the `InOldTree` or `InNewTree` states respectively. The dependency functions must not return objects in these states, since this would lead to violation of the preconditions when calling the `deEvalL` procedures of those doors. In addition, the non-local dependency sets must be disjoint, just as for the incremental phases.

Consider first the exhaustive evaluation in phase III. A door *d* is in the `InNewTree` state if it has not yet been evaluated. In this case, *d* cannot yet be a member of any collection object of another door, since it is only conditions in *d* which can add *d* to collection objects of other doors. Thus, dependency functions which consult the information in collection objects, as described in §9.5.5, cannot return sets containing objects in the `InNewTree` state. However, if the dependency function performs a search in the syntax tree to find affected doors, it must explicitly test the state of the encountered door objects in order to avoid returning door objects in the `InNewTree` state.

The situation is analogous in phase I. Here, doors in the `InOldTree` state have been de-evaluated so they can no longer be members of collections in other doors. Thus, dependency functions consulting information in collections cannot return doors which are in the `InOldTree` state.

10.5.3 Visit order during exhaustive de-evaluation

During the exhaustive de-evaluation (phase I), changes may be propagated via non-local dependencies from one door in the old subtree to other doors also in the old subtree. This can happen if the receiving doors have not yet been visited by the `exhDeEvalVisit` procedure, i.e. they are still in the `Evaluated` state. The normal action for a visit procedure which propagates to non-local dependents is to add the receiving doors to the work list (step 6 of algorithm 10-3). However, receiving door objects in the old subtree should not remain on the work list after phase I since this would result in meaningless evaluation in the old subtree during phase V. Such evaluation would also break the precondition of the `incOwnerVisit` procedures of these doors, since they are in the `InOldTree` state after the de-evaluation phase.

We have chosen to solve this problem as follows: The visit procedures add receiving doors as usual, according to step 6, regardless of if the doors are in the old subtree or not. If a door of the old subtree is added to the work list, it will be removed just before it is de-evaluated. This is the reason for the statement

```
    worklist.removeIfFound($d);
```

in phase I of the evaluator algorithm.

10.5.4 Meeting visit procedure preconditions. Summary.

The preconditions of the procedures `exhEvalVisit`, `exhDeEvalVisit`, and `incOwnerVisit` are met as was shown in §10.5.1.

To ensure that the preconditions of the procedures `deEvalL` and `evalL` are met, the following must be observed in the implementation of door visit procedures:

Given a door object *d* and a door visit procedure *p*, the preconditions of the `deEvalL` and `evalL` procedures called by *p* are met if:

1. all direct and transitive sets of dependent doors are disjoint, and *d* itself is not a member of any of these sets, and
2. all dependency functions consult collection object information rather than search the syntax tree

If (1) does not hold, the algorithm of *p* has to be adapted. This will be discussed in §10.6.1.

If (2) does not hold, i.e. if the dependency functions search the syntax tree, an explicit state flag can be stored in each door object and the dependency function can test this flag to return only objects in the appropriate state.

10.6 Modifications to the basic visit procedure algorithm

10.6.1 Overlapping dependency sets

As mentioned in the previous sections, the non-local dependency sets computed by a door visit procedure may overlap. In this case, the basic algorithm of 10-3 has to be modified in order to avoid violation of the legal state transitions of doors.

Consider the non-local dependency sets $\mathtt{DP}_1$ . . $\mathtt{DP}_n$ computed according to the send vertices $v(\mathtt{L}_1, \mathtt{D}_1, \mathtt{f}_1)$ . . $v(\mathtt{L}_n, \mathtt{D}_n, \mathtt{f}_n)$ in step 1 of the basic algorithm 10-3. Clearly, if $\mathtt{D}_1$ . . $\mathtt{D}_n$ are all different door classes, the dependency sets will all be disjoint. If this is not the case, the door package has to be more closely examined to determine if the dependency sets can overlap, and if so, how this can be amended. If any of the receive vertices $v(\mathtt{L}_k)$ in turn has dependent send vertices, this implies transitive non-local dependency sets which have to be examined as well.

One situation which may occur is the following: Consider two vertices $v(\mathtt{L}_j, \mathtt{D}_j, \mathtt{f}_j)$ and $v(\mathtt{L}_k, \mathtt{D}_k, \mathtt{f}_k)$ where $\mathtt{L}_j \neq \mathtt{L}_k$, $\mathtt{D}_j = \mathtt{D}_k$, and $\mathtt{f}_j = \mathtt{f}_k$. We say that a receive vertex $v(\mathtt{L}_1)$ is *covered* by another receive vertex v(L$_2$) if all vertices reachable from v(L$_1$) are reachable also from v(L$_2$). If L$_j$ is covered by L$_k$, then the dependency set DP$_j$ can simply be dropped since all evaluation according to DP$_j$ is covered by the set DP$_k$. This particular example occurs in `DeclDoor` where the send vertices v(`getTpChanged`, `UseDoor`, `fUses`) and v(`lookupChanged`, `UseDoor`, `fUses`) are related exactly like this.

Another example, also occurring in `DeclDoor`, is the vertices v(`lookupChanged`, `UseDoor`, `fUses`) and v(`lookupChanged`, `UseDoor`, `fAttempted`). Although both these dependency sets contain `UseDoor` objects, they will not overlap because the sets returned by `fUses` and `fAttempted` will always be disjoint. In this case, there is thus no problem of overlapping dependency sets.

10.6.2 Testing attribute values for convergence

The re-evaluation of a local attribute defined in a door may result in the same value as before. In this case, it is not necessary to re-evaluate dependent invariants. In principle, general techniques can be used to automatically generate code for detecting such value convergence and the visit procedure algorithm can be modified accordingly. However, in our examples we will only employ such tests in a few special cases.

The `incOwnerVisit` procedure re-evaluates the invariants which depend on non-fix inherited attributes. In the case where all directly dependent invariants are copy equations defining local attributes, an additional step 0 will be added to the visit procedure. This step tests the values of the non-fix inherited attributes against the local attributes defined by the copy equations. If all of these attributes converge, the rest of the visit procedure is skipped. If only a subset of the non-fix inherited attributes have new values, the visit procedure will perform only a corresponding subset of its normal actions. These convergence tests improve performance by avoiding unnecessary re-evaluation. In addition, such tests are necessary to handle circular dependencies, as treated in §10.9.

Another case of value convergence may occur when calling the `deEvalL` and `evalL` visit procedures of non-locally dependent doors. If the synthesized attributes are unchanged, although they depend on the re-evaluated invariants, the evaluation does not need to be propagated out into the syntax tree. I.e., it is unnecessary to add the non-local door to the evaluator work list.

10.6.3 Affecting the solution for underdetermined grammars

As discussed in §8.10, a Door AG can be made *underdetermined*. For example, the order of a list collection can be left undefined by the grammar. For such grammars, any permutation of the member elements is consistent with the Door AG definition and the actual permutation will depend on the order of evaluation. It is possible to add additional code to the visit procedures to explicitly control the permutation. This may be necessary in order to obtain a particular history-dependent behavior. An example of this is given in §11.1 where additional code is added to the visit procedures of `DeclDoor` to handle multiple declarations of the same identifier.

10.7 Visit procedures for the example door package

In this section we will implement visit procedures for the four door classes `RootDoor`, `BlockDoor`, `DeclDoor`, and `UseDoor` in the example door package of §8.9. The characteristic sets for each visit procedure are computed from the dependency graphs in Figure 9.7. The visit procedures are then implemented according to algorithm 10-3, with possible modifications as described in §10.6.

10.7.1 RootDoor

The dependency graph for class `RootDoor` contains only a synthesize vertex, and all the characteristic sets of its visit procedures are therefore empty. The visit procedure implementations are consequently empty as well.

10.7.2 BlockDoor

The dependency graph for class `BlockDoor` gives the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
`exhDeEvalVisit`	∅	∅	∅
`exhEvalVisit`	∅	{ *v*(`theTablePath.table`), *v*(`staticPath.first`), *v*(`staticPath.second`) }	∅
`incOwnerVisit`	∅	∅	∅

The characteristic sets for the procedures `exhDeEvalVisit` and `incOwnerVisit` are all empty, and these procedures are consequently empty as well. For the procedure `exhEvalVisit`, only the S_{loc} set is non-empty. The resulting implementation therefore only contains step 4 of the basic algorithm:

```
addto BlockDoor
{ impl exhEvalVisit
  { (* Step 4. Evaluate local invariants *)
    eval theTablePath.table;
    eval staticPath.first;
    eval staticPath.second;
  };
```

10.7.3 DeclDoor

The class `DeclDoor` is the most complex of the door classes since it has non-locally dependent doors. All three visit procedures `exhDeEvalVisit`, `exhEvalVisit`, and `incOwnerVisit` have non-empty implementations.

10.7.3.1 Procedure exhDeEvalVisit

The procedure `exhDeEvalVisit` for class `DeclDoor` has the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhDeEval-Visit	{ *v*(registered) }	∅	{ *v*(lookupChanged, UseDoor, fUses) }

The basic algorithm gives the following implementation of procedure `exhDeEvalVisit`:

```
addto DeclDoor
{ impl exhDeEvalVisit
  { boundUses: ref Set[UseDoor]; (* dependency set *)

    (* Step 1. Compute dependent doors *)
    boundUses :- fUses;

    (* Step 2. De-evaluate dependent doors *)
    for $d :- boundUses.each do $d.deEvalLookupChanged; end for;

    (* Step 3. De-evaluate local conditions *)
    deeval registered;

    (* Step 5. Evaluate dependent doors *)
    for $d :- boundUses.each do $d.evalLookupChanged; end for;

    (* Step 6. Add dependent doors to work list*)
    evaluator.worklist.addSet(boundUses);
  };
};
```

10.7.3.2 Procedure exhEvalVisit

The procedure `exhEvalVisit` for class `DeclDoor` has the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhEval-Visit	∅	{ *v*(theEntry.table), *v*(registered), *v*(theEntry.tp), *v*(theEntry.ident) }	{ *v*(getTpChanged, UseDoor,fUses), *v*(lookupChanged, UseDoor, fAttempted) }

The functions `fUses` and `fAttempted` will always return disjoint sets. This follows from the grammar since a `UseDoor` object which is bound to an entry object cannot at the same time be a member of the `attempted` collection in the symbol table containing the entry. Thus, the two send vertices in S_{send} represent disjoint dependency sets. The implementation of the `exhEvalVisit` procedure is therefore a direct application of the basic algorithm:

```
addto DeclDoor
{ impl exhEvalVisit
  { tpUses: ref Set[UseDoor]; (* dependency set *)
    attemptedUses: ref Set[UseDoor]; (* dependency set *)

    (* Step 1. Compute dependent doors *)
    tpUses :- fUses;
    attemptedUses :- fAttempted;

    (* Step 2. De-evaluate dependent doors *)
    for $d :- tpUses.each do $d.deEvalGetTpChanged; end for;
    for $d :- attemptedUses.each do
      $d.deEvalLookupChanged;
    end for;

    (* Step 4. Evaluate local invariants *)
    eval theEntry.table;
    eval registered;
    eval theEntry.tp;
    eval theEntry.ident;

    (* Step 5. Evaluate dependent doors *)
    for $d :- tpUses.each do $d.evalGetTpChanged; end for;
    for $d :- attemptedUses.each do
      $d.evalLookupChanged;
    end for;

    (* Step 6. Add dependent doors to work list*)
    evaluator.worklist.addSet(tpUses);
    evaluator.worklist.addSet(attemptedUses);
  };
};
```

It can be deduced from the grammar that the `tpUses` dependency set will always be empty. This is because there can only be `UseDoor` objects bound to the entry if the entry is in the symbol table list, which is not the case at the beginning of procedure `exhEvalVisit`. The visit procedure above can thus be simplified by removing the code associated with `tpUses`.

10.7.3.3 Procedure incOwnerVisit

The procedure `incOwnerVisit` for class `DeclDoor` has the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
`incOwner-Visit`	∅	`{ `*v*`(theEntry.tp),` *v*`(theEntry.ident) }`	`{ `*v*`(getTpChanged, UseDoor, fUses),` *v*`(lookupChanged, UseDoor, fUses),` *v*`(lookupChanged, UseDoor, fAttempted) }`

In this procedure, the step 0 discussed in §10.6.2 is added to test for convergence of the two non-fix inherited attributes `tp` and `ident` by comparing them to `theEntry.tp` and `theEntry.ident`. For each possible outcome of these comparisons, refined characteristic sets S_{loc} and S_{send} can be computed by treating the unchanged attributes as fix inherited attributes. This results in the following characteristic sets:

	S_{loc}	S_{send}
`a) tp unchanged ident unchanged`	∅	∅
`b) tp changed ident unchanged`	`{ `*v*`(theEntry.tp) }`	`{ `*v*`(getTpChanged, UseDoor, fUses) }`
`c) tp unchanged ident changed`	`{ `*v*`(theEntry.ident) }`	`{ `*v*`(lookupChanged, UseDoor, fUses),` *v*`(lookupChanged, UseDoor, fAttempted) }`
`d) tp changed ident changed`	`{ `*v*`(theEntry.tp),` *v*`(theEntry.ident) }`	`{ `*v*`(getTpChanged, UseDoor, fUses),` *v*`(lookupChanged, UseDoor, fUses),` *v*`(lookupChanged, UseDoor, fAttempted) }`

The visit procedure is modified to handle these four cases.

In case (a), where both `tp` and `ident` are unchanged, the rest of the visit procedure is skipped.

In case (d), where both attributes are changed, there is a dependency set overlap between *v*(`getTpChanged`, `UseDoor`, `fUses`) and *v*(`lookupChanged`, `UseDoor`, `fUses`), since the same dependency function is used for these vertices. However, since the receive vertex *v*(`getTpChanged`) is covered by the receive vertex *v*(`lookupChanged`) in the dependency graph for `UseDoor`, the dependency set corresponding to the send vertex *v*(`getTpChanged`, `UseDoor`, `fUses`) can simply be dropped in the implementation of `exhDeEvalVisit`, as discussed in §10.6.1.

In steps 2, 5, and 6, we will make use of the following constant object, modelling an empty set of `UseDoor` objects.

```
emptyUseDoorSet: object Set[UseDoor];
```

The procedure `incOwnerVisit` can now be implemented as follows:

```
addto DeclDoor
{ impl incOwnerVisit
  { tpUses: ref Set[UseDoor]; (* dependency set *)
    boundUses: ref Set[UseDoor]; (* dependency set *)
    attemptedUses: ref Set[UseDoor]; (* dependency set *)
    tpUnchanged: boolean;
    identUnchanged: boolean;

    (* Step 0. Check for value convergence *)
    tpUnchanged := tp == theEntry.tp;
    identUnchanged := ident = theEntry.ident;

    if tpUnchanged and identUnchanged then
      (* case (a) - skip the rest *)
    else
      (* Step 1. Compute dependent doors *)
      tpUses :- fUses;
      boundUses :- fUses;
      attemptedUses :- fAttempted;

      (* Adjust dependency sets according to cases (b,c,d) *)
      if identUnchanged then (* case (b) *)
        boundUses :- emptyUseDoorSet;
        attemptedUses :- emptyUseDoorSet;
      else if tpUnchanged then (* case (c) *)
        tpUses :- emptyUseDoorSet;
      else (* case (d) *)
        tpUses :- emptyUseDoorSet;
      end if;

      (* Step 2. De-evaluate dependent doors *)
      for $d :- tpUses.each do $d.deEvalGetTpChanged; end for;
      for $d :- boundUses.each do $d.deEvalLookupChanged; end for;
      for $d :- attemptedUses.each do
        $d.deEvalLookupChanged;
      end for;

      (* Step 4. Evaluate local non-fix invariants *)
      if not tpUnchanged then eval theEntry.tp; end if;
      if not identUnchanged then eval theEntry.ident; end if;

      (* Step 5. Evaluate dependent doors *)
      for $d :- tpUses.each do $d.evalGetTpChanged; end for;
      for $d :- boundUses.each do $d.evalLookupChanged; end for;
      for $d :- attemptedUses.each do
        $d.evalLookupChanged;
      end for;

      (* Step 6. Add dependent doors to work list *)
      evaluator.worklist.addSet(tpUses);
      evaluator.worklist.addSet(boundUses);
      evaluator.worklist.addSet(attemptedUses);
    end if;
  };
};
```

10.7.4 UseDoor

The dependency graph for class `UseDoor` has two receive vertices *v*(`getTpChanged`) and *v*(`lookupChanged`). Thus, in addition to the procedures `exhDeEvalVisit`, `exhEvalVisit`, and `incOwnerVisit`, the two visit procedure pairs (`deEvalGetTpChanged`, `evalGetTpChanged`) and (`deEvalLookupChanged`, `evalLookupChanged`) need to be implemented. The dependency graph for class `UseDoor` gives the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhDeEval-Visit	{ *v*(cUses), *v*(cAttempted) }	∅	∅
exhEval-Visit	∅	{ *v*(localPath), *v*(localIdent), *v*(binding), *v*(cUses), *v*(cAttempted) }	∅
incOwner-Visit	{ *v*(cUses), *v*(cAttempted) }	{ *v*(localPath), *v*(localIdent), *v*(binding), *v*(cUses), *v*(cAttempted) }	∅
deEval/Eval getTpChanged	∅	∅	∅
deEval/Eval lookupChanged	{ *v*(cUses), *v*(cAttempted) }	{ *v*(localPath), *v*(localIdent), *v*(binding), *v*(cUses), *v*(cAttempted) }	∅

The procedures are all straight-forward implementations of the basic algorithm:

```
addto UseDoor
{ impl exhDeEvalVisit
  { (* Step 3. De-evaluate local conditions *)
    deeval cUses;
    deeval cAttempted;
  };

  impl exhEvalVisit
  { (* Step 4. Evaluate local invariants *)
    eval localPath;
    eval localIdent;
    eval binding;
    eval cUses;
    eval cAttempted;
  };

  impl incOwnerVisit
  { (* Step 3. De-evaluate local conditions *)
    deeval cUses;
    deeval cAttempted;

    (* Step 4. Evaluate local invariants *)
    eval localPath;
    eval localIdent;
    eval binding;
    eval cUses;
    eval cAttempted;
  };

  deEvalGetTpChanged: proc
  { (* empty *)
  };

  evalGetTpChanged: proc
  { (* empty *)
  };

  deEvalLookupChanged: proc
  { (* Step 3. De-evaluate local conditions *)
    deeval cUses;
    deeval cAttempted;
  };

  evalLookupChanged: proc
  { (* Step 4. Evaluate local invariants *)
    eval binding;
    eval cUses;
    eval cAttempted;
  };
};
```

10.8 Evaluation of main grammar

10.8.1 Main grammar classification

The main grammar of a Door AG can be classified according to standard AG classes such as 1-visit, OAG, non-circular, and circular. This is done by considering each door class as an ordinary node class and by ignoring all send and receive vertices of the door class dependency graphs. While 1-visit standard AGs are applicable only to very simple 1-pass languages, the class of 1-visit main grammars is sufficient for a much larger range of practical languages. The reason for this is that problems like name analysis which give rise to complex dependencies in standard AGs are handled by doors and semantic objects in Door AGs. The remaining dependencies in the main grammar are very simple. For example, languages like Algol and Simula, where order of declaration is irrelevant, can be described by Door AGs with 1-visit main grammars. Describing these languages in standard AGs requires an OAG.

Our evaluation technique for Door AGs handles 1-visit main grammars, and the evaluation technique used is the very simple one based on static skipping which was introduced in Chapter 7. We find it interesting that it is possible to use such simple implementation techniques and yet achieve an efficient incremental system for a complex language like Simula.

Nevertheless, it is straight-forward to adapt our evaluation technique to other existing standard AG algorithms in order to handle more complex main grammars. In §10.8.6 we will show how this can be done.

Another interesting thing to note is that a door package may have non-local dependencies which lead to circular chains of attribute dependencies when combined with a main grammar. However, since the non-local dependencies are irrelevant for attribute evaluation in the main grammar, this does not affect the main grammar complexity. Thus, simple 1-visit evaluation techniques can be used for the main grammar even if the Door AG as a whole is circular. Examples of circular dependencies are given in §10.9.

10.8.2 Extensions to standard AG method

In order to evaluate the main grammar, the visit procedures `exhVisit`, `incFatherVisit`, `incSonVisit`, and `incDoorVisit` need to be implemented for the node classes. This can be done automatically. The `exhVisit`, `incFatherVisit`, and `incSonVisit` procedures are implemented as described in Chapter 7, with certain extensions as described below. The `incDoorVisit` procedure is implemented analogously to the `incSonVisit` procedure, simply by considering the door objects declared in a node class as another kind of son nodes.

10.8.2.1 Dependency graphs

The dependency graph construction of 7-5 is extended to handle door part-objects. The doors are treated exactly like son nodes, i.e., a vertex *v*(`d`) is added to the graph of a node class for each door object `d` declared in the node class. Edges to and from door vertices are added in exactly the same way as for son nodes, according to the use of synthesized attributes and definitions of inherited attributes of the doors.

10.8.2.2 Exhaustive evaluation

The implementation of the procedure `exhVisit` described in 7-8 is extended to handle door part-objects. Each VISIT instruction corresponding to a door vertex *v*(`d`) is implemented as a call

```
d.exhEvalVisit
```

10.8.2.3 Incremental evaluation

The implementation of the procedures `incFatherVisit` and `incSonVisit` is modified to handle fix attributes and door part-objects.

We recall from §8.5.1 that a fix attribute receives a value during exhaustive evaluation, but cannot change during incremental evaluation. I.e., during incremental evaluation, the fix attributes are equivalent to constants. This implies that these attributes need not be considered in the dependency analysis for incremental evaluation.

A subset of the edges in the dependency graphs constructed according to algorithm 7-5, correspond to use or definition of fix attributes. In the construction of the incremental visit sequences, according to 7-9, these edges are removed from the graph before performing the topological sort.

The implementation of the procedures `incFatherVisit` and `incSonVisit` as described in 7-10 is extended to handle door part-objects. Each VISIT instruction corresponding to a door vertex *v*(`d`) is implemented as a call

```
d.incOwnerVisit
```

The procedure `incDoorVisit` is implemented exactly analogous to the procedure `incSonVisit`, but dispatches on door part objects instead of on son nodes.

10.8.2.4 Demand attributes

The synthesized and inherited attributes of a door class are demand attributes and can be implemented in the same way as demand attributes of standard AGs as treated in §7.1.

10.8.3 Visit procedures for the main grammar example

As an example, consider the main grammar example in Chapter 8 (defined in §8.9.4). The dependency graphs for the construction classes are shown in Figure 10.5.

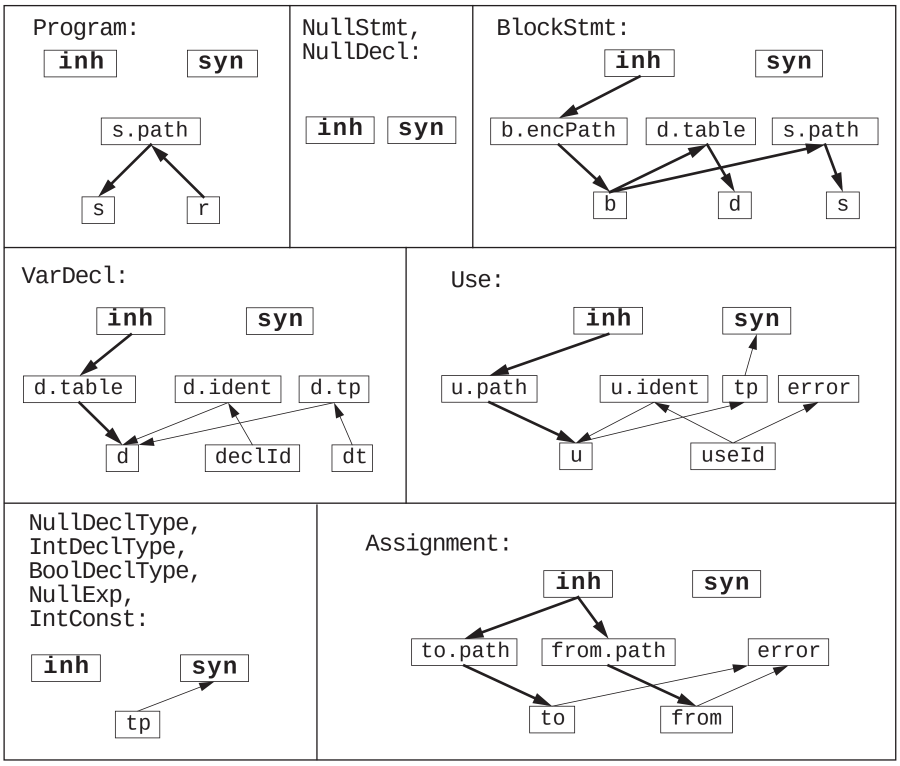


Figure 10.5 Dependency graphs for main grammar. Dependency edges corresponding to use or definition of fix attributes are drawn with thicker lines.

The visit procedures are straight-forward to construct from these dependency graphs, using the algorithms in 7-8 and 7-10, modified as described in the previous section. Note that all the attributes in the main grammar are demand attributes, and therefore dropped in the visit procedures:

```
addto Program
{ impl exhVisit
  { r.exhEvalVisit;
    s.exhVisit;
  };
};

addto BlockStmt
{ impl exhVisit
  { b.exhEvalVisit;
    d.exhVisit;
    s.exhVisit;
  };
};

addto VarDecl
{ impl exhVisit
  { dt.exhVisit;
    d.exhEvalVisit;
  };

  impl incSonVisit
  { d.incOwnerVisit;
  };
};
```

```
addto Use
{ impl exhVisit
  { u.exhEvalVisit;
  };

  impl incSonVisit
  { u.incOwnerVisit;
    father.incSonVisit
      (this ANYNODE);
  };

  impl incDoorVisit
  { father.incSonVisit
      (this ANYNODE);
  };
};

addto Assignment
{ impl exhVisit
  { to.exhVisit;
    from.exhVisit;
  };
};
```

Figure 10.6 Visit procedures for main grammar

As shown in the above figure, the visit procedures for the main grammar are extremely simple and short. Empty procedures need not be implemented since they can rely on empty default implementations in class `ANYNODE`.

The incremental procedures (`incFatherVisit`, `incSonVisit`, and `incDoorVisit`) are often empty or very simple due to the heavy use of fix attributes. This leads to a large amount of instruction skipping compared to the `exhVisit` procedure. For example, consider the dependency graph for `BlockStmt`. All edges in this graph represent uses or definitions of fix attributes, and all incremental visit procedures are consequently empty for the `BlockStmt` class.

10.8.4 Effects of static skipping

The evaluation algorithm used for main grammars is based on the static skipping algorithm described in Chapter 7. As discussed in §7.3.4, this algorithm is sub-optimal and may evaluate more attributes than an algorithm based on dynamic skipping. On the other hand, it may in some cases be faster than a dynamic skipping algorithm since it avoids comparisons of attribute values. The primary advantage of the static skipping algorithm is its simplicity and the ease with which it can be implemented.

The static skipping algorithm does not compare old and new attribute values and can therefore not stop change propagation if attribute values converge. For a standard AG, this can be a serious limitation and lead to poor incremental performance. However, a main grammar of a Door AG is radically different in character from a standard AG. Because non-local information is propagated via doors, the dependency paths in the syntax tree can be much shorter than in a standard AG. Convergence tests are performed in the door visit procedures and can thus prevent unnecessary non-local propagation.

For example, consider the dependency graphs of the main grammar in Figure 10.5. Changes to the identifier or type of a `VarDecl` node will only propagate into `d` (the `DeclDoor`). A corresponding node class in a standard AG would have a synthesized attribute which would be dependent on such changes, and the changes would be propagated to the father node and further on through the syntax tree. A change to the identifier of a `Use` node will propagate into `u` (the `UseDoor`) and also via the `tp` attribute up to the father node. If the grammar had contained some more advanced expressions, such a change could propagate further up in the syntax tree, but not further up than to the enclosing statement, since the statements have no synthesized attributes. Although this main grammar is extremely simple, it is representative for the way the standard static-semantic problems of name analysis, type checking, and error checking can be specified in a Door AG.

For static-semantic checking using Door AGs, the sub-optimal effects of the static skipping algorithm are thus very limited.

10.8.5 Effects of using demand attributes

The use of demand attributes in the main grammar saves storage. However, there is also a potential danger of using such attributes since evaluation time can grow very quickly for certain kinds of uses.

Consider evaluation of a `UseDoor`. This involves an access to the demand attribute `path`. Such access leads to a series of function calls, typically one for each syntax node on the way up to the nearest node with a `BlockDoor`. This is usually not a long distance - perhaps 10 nodes at the most. This gives an overhead of 10 function calls per evaluation of a `UseDoor`. If only few `UseDoor` objects are evaluated, which should be the common case in an incremental system, this overhead should be no problem. However, one should take care in defining the equations and code inside the `UseDoor` so the `path` attribute is accessed only once during the evaluation. This is the case for our implementation of `UseDoor` since a local copy of `path` is stored in the door. In evaluating the door, the demand attribute is accessed only once to assign a value to the local copy. Thereafter the local copy is accessed.

A potentially more dangerous use of demand attributes is when using them for synthesized attributes. Consider a type attribute of an expression which is defined in terms of the type attributes of its son nodes, and the type attributes of these son nodes are defined in terms of their son nodes and so on. Thus, access to one type attribute could result in a number of function calls proportional to the size of the whole subtree. It could even be much worse, if each such function accesses the types of its son nodes more than once, in which case there would be an exponential growth of the number of function calls.

To be useful in practice, it is necessary that demand attributes are not used in situations where the subtrees can grow large, and where the same attribute is accessed more than once. Type-checking is usually not problematic from this point of view. First of all, the size of subtrees is bound by the size of the largest statement since statements do not have synthesized type attributes. Usually, at least in object-oriented programming, the number of syntax nodes within a statement subtree (not counting sub-statements) is rather small. In addition, it is usually possible to avoid accessing the same attribute more than once by using let-expressions in the definition of the equations.

Nevertheless, there is a potential time-sink in using demand attributes, and one must be careful when writing the grammar in order to avoid situations where the same attribute function is called over and over again. It could be useful to develop a scheme for caching demand attributes during a thread of evaluation.

10.8.6 Using standard evaluation algorithms

We have chosen to use the statically skipping 1-visit algorithm for implementation of main grammars because it is very simple and yet sufficient for practical problems. However, any incremental evaluation algorithm for standard AGs could, in fact, be adapted and used for the main grammar, by a simple adaptation of the Door AG evaluator. The only restriction is that the main grammar must treat the door objects as 1-visit nodes. I.e., a synthesized attribute of a door object must not be used (directly or transitively) to define an inherited attribute of the same door object.

Consider any incremental evaluation algorithm for standard AGs. For most such algorithms it is straight-forward to construct the two following procedures: `StandardNew(old, new)` and `StandardChanged(n)` with the following semantics:

- `StandardNew(old: ref ANYNODE, new: ref ANYNODE)` is called when a subtree of a consistently attributed syntax tree has been replaced by a new completely un-attributed subtree. The reference `old` denotes the root of the replaced subtree and `new` denotes the root of the new subtree. The procedure restores consistency in the syntax tree.
- `StandardChanged(n: ref ANYNODE)` is called when all attributes in the whole syntax tree are consistent except for the immediate successors of the synthesized attributes of a node `n`. The procedure restores consistency in the syntax tree.

These procedures must then be adapted to handle evaluation which propagates into door objects. For dependency analysis, only the door package interface needs to be considered. The door classes are considered as a special kind of node classes where all the synthesized attributes depend on all the inherited attributes of the door. I.e., all door classes are inherently 1-visit. The procedures above are adapted as follows:

- `AdaptedStandardNew(old: ref ANYNODE, new: ref ANYNODE)` is an adaptation of `StandardNew`: When evaluation is propagated into a door object, the procedure `incOwnerVisit` is called, unless the door is in the subtree rooted at new and it is the first time the door is visited. In this case the procedure `exhEvalVisit` is called instead.
- `AdaptedStandardChanged(n: ref ANYDOOR)` is an adaptation of `StandardNew`: The parameter is here a door object. When evaluation is propagated into another door object, the procedure `incOwnerVisit` is called.

```
evaluator: object
{ worklist: object OrderedCollection[ANYDOOR];

  replaceSubtree: proc
    (oldNode: ref ANYNODE, newNode: ref ANYNODE)
  { d: ref ANYDOOR;

    (* I: Exhaustive de-evaluation phase *)
    for $d :- oldNode.allDoors.each do
      if worklist.contains($d) then
        worklist.remove($d);
      end if;
      $d.exhDeEvalVisit;
    end for;

    (* II: Subtree Replacement *)
    oldNode.replaceBy(newNode);

    (* III, IV: “Standard” evaluation phase *)
    AdaptedStandardNew(oldNode, newNode);

    (* V: Non-local incremental phase *)
    while not worklist.empty do
      d :- worklist.removeFirst;
      AdaptedStandardChanged(d);
    end while;
  };
};
```

Figure 10.7 Door AG evaluation based on standard algorithm

Figure 10.7 shows how the Door AG evaluator of Figure 10.2 can be adapted to standard algorithm, by collapsing phases III and IV into one “standard” phase, where `AdaptedStandardNew` is called, and by calling the `AdaptedStandardChanged` procedure in phase V. Thus, the algorithm in Figure 10.7 can evaluate Door AGs with main grammars of any standard AG class, given that the door objects are treated as 1-visit nodes. To handle several visits to door objects, each of the door visit procedures `exhEvalVisit` and `incOwnerVisit` would have to be refined into a set of visit procedures.

10.9 Circular dependencies

There are several situations in static semantic checking which lead to circular chains of information dependencies, at least intuitively. This was discussed in §3.3.6. When specifying these static semantic problems in Door AGs, some of these intuitively circular dependencies also lead to actual circular dependencies between attributes in the attributed syntax tree. Fortunately, these circular dependencies are very simple to handle in Door AGs.

The collections and conditions used in Door AGs are inherently cyclic in the following way: Information about a collection (e.g. a reference to it) is propagated to a number of door objects distributed in the syntax tree. By defining members in the collection, using conditions, information is propagated back from these door objects to the collection. While this does not directly introduce circular dependencies in the dependency graphs, it defines a circular path along which other information can be propagated giving true circular attribute dependencies.

In comparison to the intuitive circular dependencies discussed in §3.3.6, the first issue, *arbitrary declaration / application order*, can be solved in Door AGs simply by using conditions and collections, as described in the example of §8.9. In the case of *cyclic subclassing*, the circularity is explicitly broken in order to avoid constructing cyclic visibility graphs. This will be shown in §11.2. The case of *reference variables* leads to true circular chains of attribute dependencies, but this circularity converges immediately and is straight-forward to handle as will be shown in §11.3.3.

Potential circular dependencies can easily be recognized from the door dependency graphs by matching send and receive vertices. Let `D1` be a door class with a send vertex *v*(`L`, `D2`, `f`) and `D2` a door class with a receive vertex *v*(`L`). If there is a dependency path from an inherited attribute `a` of `D1` to the send vertex, and a path from the receive vertex to a synthesized attribute `b` of `D2`, then this is a potential cyclic dependency. Figure 10.8 illustrates this.

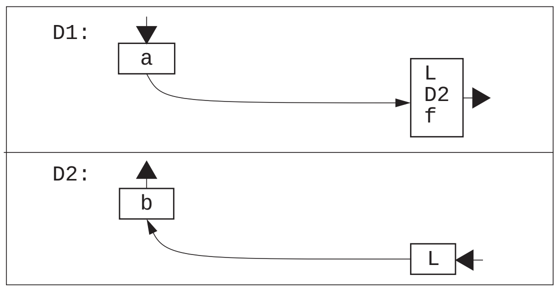


Figure 10.8 Dependency graphs with potential cycle

A cycle is obtained if the main grammar defines the attribute `a` by using the value of the `b` attribute as illustrated in Figure 10.9.

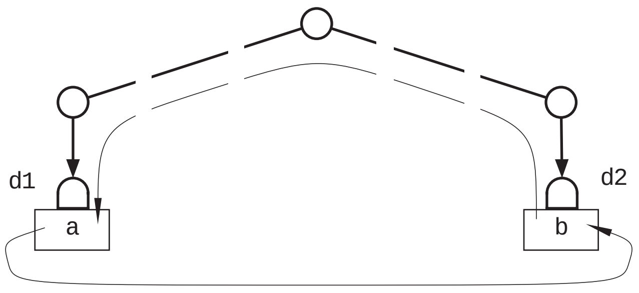


Figure 10.9 Circular chain of dependencies

A circular chain of dependencies which passes through a non-local dependency like this can be handled in a very simple way. The chain passes via the inherited attribute `a` of a door object of class `D1`. Provided that the evaluation converges, the evaluation loop is stopped simply by inserting a convergence test on `a` in the `incOwnerVisit` procedure of `D1`, exactly as described in §10.6.2.

A circular dependency as the one above can appear as the result of using collections and conditions. Consider Figure 10.10. The door `d3` has a collection part-object and a reference to this object is passed through the syntax tree to both `d1` and `d2`. Hence, the dependencies from `collRef3` to `collRef1` and `collRef2`. The condition `c` in `d1` defines the object `d1` as a member of the collection. This makes the information in `d1` available to `d2` and results in the non-local dependency from `a` to `b`. When evaluating this tree exhaustively, the doors are evaluated according to the local dependencies in the syntax tree: `d3`, `d2`, `d1`. When `d2` is first evaluated, `d1` is not yet a member of the collection. When `d1` is evaluated later, it adds itself to the collection and discovers that there is a non-local dependency from `d1` to `d2`. The door `d2` is then re-evaluated and the change will propagate around the dependency cycle until the attribute `a` of `d1` converges.

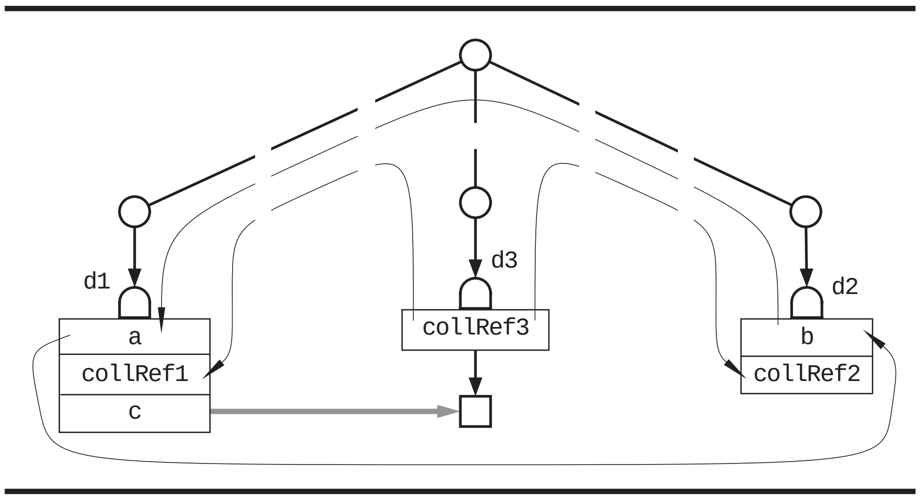


Figure 10.10 Circular dependency appearing as a result of using a collection.

Of the intuitive cyclic dependencies discussed in §3.3.6, it is only the one concerning reference variables which actually leads to circular attribute dependencies. For this problem, the attributes on the cycle converge immediately (after a single cycle). This will be discussed in §11.3.3.

10.9.1 Example: adding like-types

As a simple example of circular dependencies, we will extend our example language of §8.9.4 with "like-types" as in Eiffel [Mey88]. An example program in this extended language may be the following:

```
begin
   x: integer;
   y: like x;
end;
```

This means that `y` has the same type as `x`, i.e. it is also an integer. Like-types can be introduced by adding the following specialization of the `DeclType` node class:

```
LikeDeclType: cons DeclType(u: ref Use);
{  eq u.path :- path;
   eq tp :- u.tp;
};
```

This allows the type of a `VarDecl` to be declared as a like-type. In addition, the grammar must be extended to propagate a `path` attribute from `BlockStmt` down to the declaration part, so it can reach the `Use` node son of the `LikeDeclType` node.

The introduction of like-types leads to potential cyclic dependencies. For example, it is syntactically correct (although static-semantically erroneous) to write:

```
begin
   x: like x;
end;
```

Here, the type of `x` is declared to be the same as the type of `x`, an obviously cyclic definition. Figure 10.11 shows the circular chain of dependencies for this declaration.

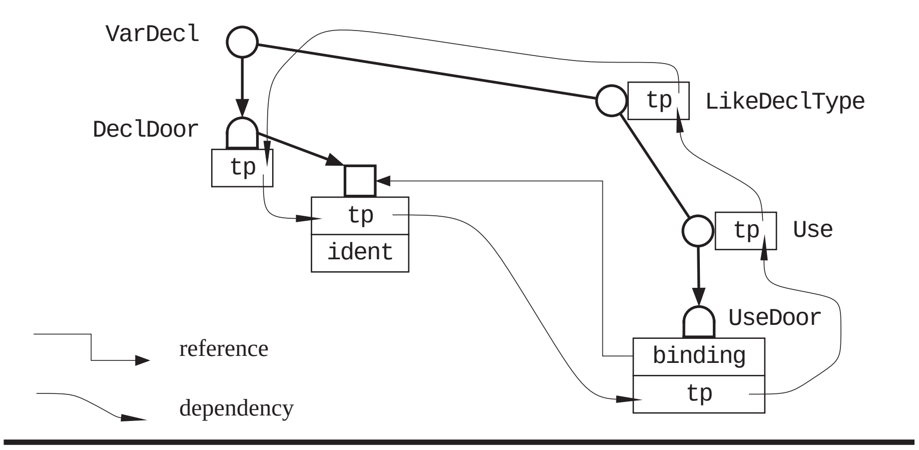


Figure 10.11 Circular definition of like-type

The `incOwnerVisit` procedure of `DeclDoor` contains a convergence test on the inherited attribute `tp` (see §10.7.3.3). Thus, since all the equations defining the `tp` attributes are copy equations, the evaluation will terminate. However, there are many attribute solutions to this syntax tree. All the `tp` attributes on the cycle must be equal, but the grammar is underdetermined in that it does not define which value to use in case of a cycle. Any `Type` value will in this case be consistent with the equations, e.g. `intType`, `boolType`, or `unknownType`. The actual solution will depend on the order of evaluation.

For example, exhaustive evaluation of the whole declaration will yield the value `unknownType` for all the `tp` attributes. On the other hand, if a declaration "`x: integer`" is changed to "`x: like x`" by replacing the `DeclType` subtree of the declaration, this will result in the value `intType` for all the `tp` attributes.

Although the incremental evaluation will work and terminate with a correct attribution (according to the grammar), this history-sensitive behavior is probably not desirable for this particular type-checking problem. A better behavior would be if the introduction of a cycle lead to an error message, and if all the `tp` attributes on the cycle got the value `unknownType`. This is possible to accomplish, but requires extensions to the door package. A similar problem is the detection of cyclic subclassing which must be handled in order to define name analysis for subclassing. This will be discussed in §11.2. A similar solution can be used to detect cyclic like-types.

10.10 Possibilities for automatizing the implementation

Our proposed method for implementing door packages is systematic, but manual. Some parts of this construction could be performed automatically while other parts are inherently manual:

- The analysis of local access-dependencies is analogous to normal attribute dependency analysis and could be performed automatically.
- The non-local dependency analysis, on the other hand, is probably very difficult to automatize in general since it involves finding actual dependencies rather than only access dependencies and this requires reasoning about the semantics of the specification. One could, however, imagine automatic support for detecting the existence non-local access-dependencies and allow manual refinement of these dependencies.
- The manual implementation of dependency functions allows arbitrary time/space tradeoffs to be done. Nevertheless, it may be possible to develop automatic support for default implementations, at least in simple cases.
- The implementation of evaluation and de-evaluation procedures for conditions is inherently manual. However, it might be possible to find some suitable “standard” conditions and collections and express them in a way suitable for automatization.
- The generation of the basic visit procedures could be done completely automatically. However, as we have seen in some examples, modifications to these basic algorithms are sometimes needed.
- The addition of extra code to the visit procedures in order to control underdeterminedness of the grammar is inherently manual.
- The detection of overlapping dependency sets is probably very difficult in general, but a pessimistic automatic detection of possible cases of overlap would be straight-forward.
- A pessimistic automatic detection of possible circularities via non-local dependencies is straight forward. Also, in the simple case where all the invariants on the circularity are copy equations, convergence is guaranteed.

It would be an interesting area of future research to build a semi-automatic system which gives automatic support for certain parts of the implementation, as indicated in the list above. Such a system should keep track of both automatic and manual implementation steps in order to handle changes to the door package without having to redo the manual steps. For example, suppose an attribute was added to a door class, and the dependency analysis and all other implementation steps could be done automatically for this attribute. The system should then update the visit procedures without destroying earlier manual additions or changes to these procedures.

10.11 Summary

We have described a visit-oriented evaluator for Door-AGs. A number of visit procedures are constructed for each door class. Each of these procedures is characterized by three “characteristic sets" computed from the dependency graph of the door. Given these characteristic sets, the basic algorithm for the procedure is straight-forward to generate. However, modifications of the basic algorithm may be needed for some visit procedures. In particular, overlapping non-local dependency sets must be detected and the algorithm modified accordingly. In addition, convergence tests on inherited attributes can be added to increase efficiency and to allow circular dependencies to be handled.

Evaluators for the main grammars can be constructed automatically by using simple adaptions of standard AG algorithms. We have shown how the simple 1-visit static-skipping algorithm of §7.3 can be adapted to evaluation of main grammars. This algorithm has the advantage of being very simple to implement, yet sufficiently powerful for practical problems.

The collections and conditions of Door AGs introduce implicit circular dependencies. Along these circular paths, explicit circular attribute dependencies can occur. Such circular dependencies are straight-forward to handle, simply by relying on convergence tests in the door visit procedures. The circular dependencies are useful for a variety of static semantic checking problems, including handling reference variables and cyclic subclassing.

Chapter 11
Advanced Attributions

In this chapter we will give some more advanced examples of how Door AGs can be used in order to specify full-blown object-oriented languages. The simple door package of the previous chapters is extended to handle the major static-semantic problems treated in Chapter 3. All these extensions have been implemented and tested. In particular, the following problems are treated:

1. Multiple declarations of the same identifier in the same block
2. Name analysis in the presence of subclassing
3. References and remote access
4. Type-checking reference assignments
5. Error presentation

The solutions to problems 1 and 2 both utilize an underdetermined grammar in order to achieve history-dependent error handling. Problem 2 leads to transitive non-local dependencies. Problem 3 leads to cyclic dependencies. Problem 5 shows how a door class can be used as an interface to external components in the interactive environment.

In §11.6 we sketch how procedures and parameters can be added, and comment on the possibilities for supporting the advanced virtual class concept of BETA.

11.1 Multiple declarations

In the implementation of the door package of §8.9 we assumed that all the entry objects in a symbol table have different names (see §9.5.3.2 - “Entry order”). In this section, the door package will be extended to detect multiple declarations of the same identifier.

We will adopt the history-dependent error-detection policy outlined in §3.6. I.e., if there are multiple declarations of the same identifier in the same block, one of these will be considered to be correct, and the others faulty. Which one is considered correct depends on the editing history.

More precisely, one of the multiple declarations is considered to be *visible*, whereas the other declarations of the same name are considered to be hidden by the visible declaration. Uses of the name can only be bound to the visible declaration and not to the hidden ones. Additional actions will be inserted in the visit procedures to give older declarations precedence over newer declarations. I.e., adding a new declaration with the same name as an existing one will cause the new declaration to be considered the erroneous one. Similarly, changing the name of a declaration to take on an already existing name, will cause the changed declaration to be considered the erroneous one.

11.1.1 The attribute visible

From the definition of the door package, it is clear that if there are multiple declarations of the same name, it is the one occurring first in the symbol table list which will be considered the visible one, since the `lookup` function of the symbol table will return this declaration entry. We model the visible/hidden status of a declaration by adding a synthesized attribute `visible` to `DeclDoor`:

```
addto DeclDoor
{   syn visible: boolean;
    eq visible := table.lookup(theEntry.ident) == theEntry;
};
```

I.e., the declaration is visible if the `lookup` function of the symbol table returns the `Entry` object owned by `DeclDoor`, and not another `Entry` object with the same name. The synthesized attribute `visible` can be used by the main grammar to report the non-visible declarations as erroneous.

To implement the history-dependent error-detection policy, the procedure `incOwnerVisit` of `DeclDoor` (§10.7.3.3) is modified to affect the permutation order of the `Entry` objects in the symbol table list: Each time the `ident` attribute of the `DeclDoor` is updated (in step 4 of the visit procedure), the `Entry` object is moved to the end of the list as follows:

```
(* Step 4. Evaluate local non-fix invariants *)
if not tpUnchanged then
    eval theEntry.tp;
end if;
if not identUnchanged then
    eval theEntry.ident;
    table.entries.remove(theEntry);
    table.entries.add(theEntry);
end if;
```

The procedure `add` of class `List` adds the element to the *end* of the list (§9.5.2). By removing and adding the `Entry` object it is thus moved to the end of the list. This is an example of a visit procedure modification to affect the solution for underdetermined grammars as discussed in §10.6.3.

The result of this modification is that each time the name of a declaration is edited, the declaration will be moved to the end of the symbol table list and will thus become hidden by other declarations of the same (new) name.

11.1.2 Dependency analysis

We now redo the dependency analysis of `DeclDoor`, with respect to the new attribute `visible`, and take into consideration that there may be several declaration entries with the same identifier in a symbol table.

The new attribute `visible` depends locally on the attribute `theEntry.ident`. In addition, it depends non-locally on the existence of other `Entry` objects with the same identifier which precedes it in the symbol table list. This is modelled by a receive vertex *v*(`VisStatus`) (see Figure 11.2).

Let `d1` and `d2` be two `DeclDoor` objects with `Entry` objects `e1` and `e2` respectively. Suppose that `e2` is a direct or indirect successor of `e1` in the symbol table list, and that `e1.ident` = `e2.ident` = *id*, as shown in Figure 11.1. Suppose furthermore that there is no other `Entry` object with this identifier located between `e1` and `e2`, or before `e1`. Thus, `e1` is visible and `e2` is hidden by `e1`.

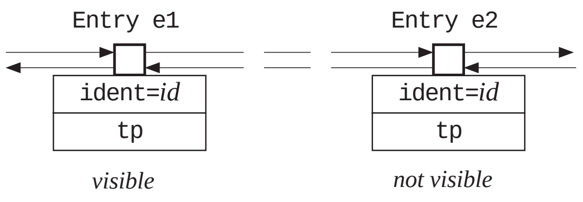


Figure 11.1 Declaration hides other declaration with the same name

The de-evaluation of the attribute `e1.ident` will cause `e2` to become visible, since `e2` then becomes the first entry with `ident` = *id* in the list. This is modelled by adding a send vertex *v*(`VisStatus`, `DeclDoor`, `fNext`) to the dependency graph of `DeclDoor` and adding a de-evaluation edge from *v*(`theEntry.ident`) to this vertex. For the situation in Figure 11.1, the function `fNext` should return a singleton set containing `d2`. In other situations, i.e. if `e1` is not the first entry with `ident` = *id*, or if there is no entry `e2`, then `fNext` should return the empty set. A similar argument for the condition `registered` leads to a de-evaluation edge from *v*(`registered`) to *v*(`VisStatus`, `DeclDoor`, `fNext`).

Now consider *evaluating* the attribute `theEntry.ident` of `d1` again. If the order between the elements is not changed, this would make `e2` invisible again. However, because of the additional code added to `incOwnerVisit`, an `Entry` object for which the `ident` attribute is evaluated is always placed at the end of the symbol table list. Thus, the evaluation of `theEntry.ident` does not have any `DeclDoor` dependents. Similarly, evaluation of the condition `registered` will add the `Entry` at the end of the symbol table list, and no `DeclDoor` objects are affected.

Figure 11.2 shows the resulting dependency graph for `DeclDoor`.

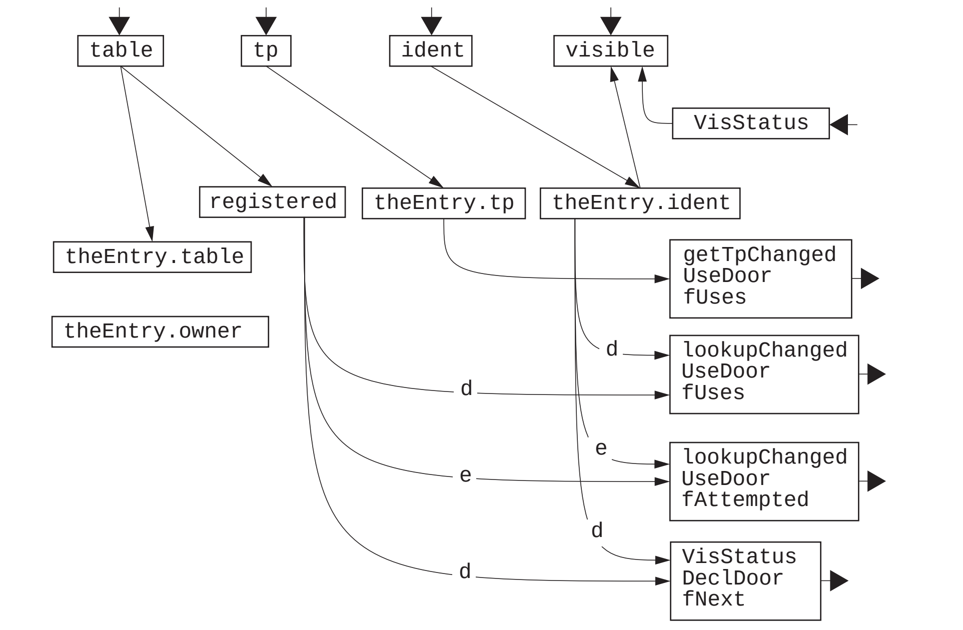


Figure 11.2 Dependency graph for `DeclDoor`

The dependency function `fNext` is straight-forward to implement, but it requires adding an additional reference attribute to class `Entry`, which denotes the `DeclDoor` object owning the `Entry`:

```
addto Entry
{   loc owner: ref DeclDoor;
};

addto DeclDoor
{   eq theEntry.owner :- this DeclDoor;
};
```

The additions needed to the door visit procedures of `DeclDoor` follow exactly the basic door visit procedure algorithm of 10-3. For brevity, these details (and the exact implementation of `fNext`) are left out.

11.2 Name analysis in presence of subclassing

The simple door package of §8.9 supports Algol style scope rules by means of the `BlockDoor`. We will now extend this package to support also classes and subclassing by adding a new door `ClassDoor`. We recall from §3.3.2 that name analysis in the presence of subclassing can be done by attaching two path vertices to each class: a *prefix path* and a *static path*. This is in contrast to the simple (Algol) blocks which have only a static path. The `ClassDoor` will thus have part objects for both a prefix path and a static path whereas the `BlockDoor` has only a static path.

11.2.1 The ClassDoor and its part objects

Classes can be modelled by a door `ClassDoor` which is similar to `BlockDoor` but has two `SearchPath` part-objects instead of only one, as illustrated in Figure 11.3. The prefix path vertex has two outgoing edges: the first edge ends in the symbol table of the class, and the second edge ends in the prefix path of the superclass (*a*). Thus, the symbol tables of a class and all its superclasses are reachable via the prefix path. The static path also has two outgoing edges: the first edge ends in the prefix path of the class and the second edge ends in the static path of the enclosing block (*b*). Thus, the static path gives access to the symbol tables of the class and all its superclasses, to the enclosing block, to the superclasses of the enclosing block (if it is also a class), to the enclosing block of the enclosing block, and so on. The specification of `ClassDoor` is shown in §11.2.5.

This visibility graph model handles not only ordinary class hierarchies, but also arbitrary combinations of classes and block structure. In §3.3.2 an example was given of nested classes (Figure 3.3). Figure 11.4 shows a corresponding attributed syntax tree which embeds the visibility graph in the attribution using objects and references. Similar to Figure 3.3, a diagonal line represents a reference attribute denoting the `emptyPath` object.

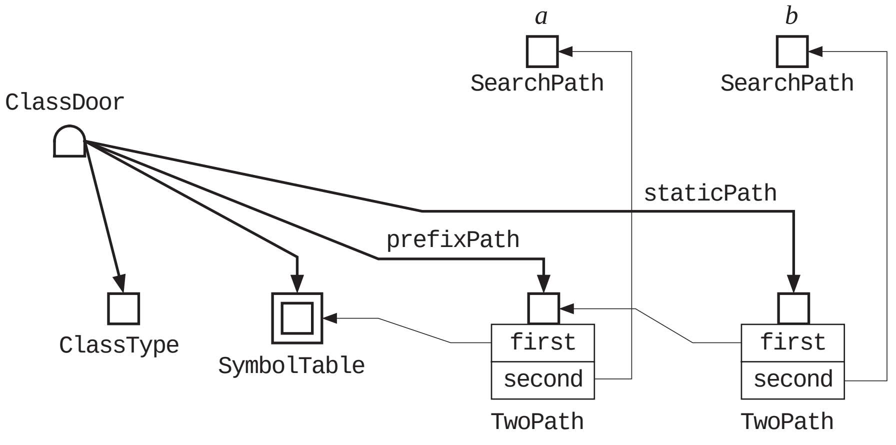


Figure 11.3 A `ClassDoor` and its part objects

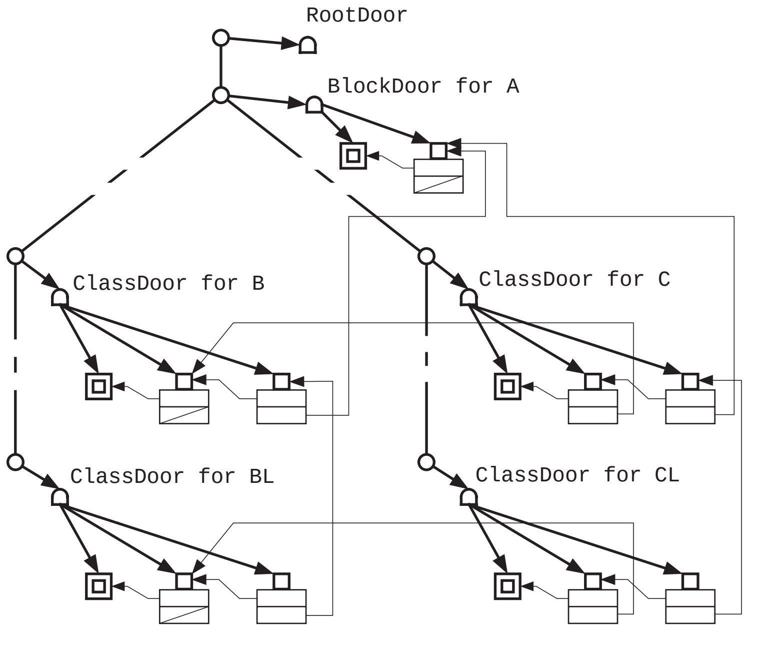


Figure 11.4 Attributed syntax tree for program with nested classes.

11.2.2 Class types

In addition to the two paths and the symbol table, the `ClassDoor` also has a `ClassType` part object. `ClassType` objects are used for type checking of classes. In the attributed syntax tree of a given user program there will be one `ClassType` object for each class in the user program. A `ClassType` object has a reference attribute `prefixClass`, denoting the `ClassType` object of the superclass. In this way, the `ClassType` objects form a forest of trees modelling the class hierarchy in the user program. As discussed in §3.5, it is useful to extend this forest to a single tree by adding a most general class “ObjectClass” which is considered to be a superclass of all the user program classes. It is also useful to introduce a class “NoClass” modelling the class of the value “NONE”. Although “NoClass” is not related to the other classes, it is for some purposes useful to regard it as a subclass of all other classes, thus extending the tree to a lattice. We model the types of “ObjectClass” and “NoClass” by two constant semantic objects `objectClassType` and `noClassType`. The generalization of these constant objects and the class `ClassType` is modelled by an abstract class `AbstractClassType`. These classes and objects are all specializations of the class `Type` introduced in the basic door package (§8.9.1), and the resulting specialization hierarchy for `Type` is shown in Figure 11.5.

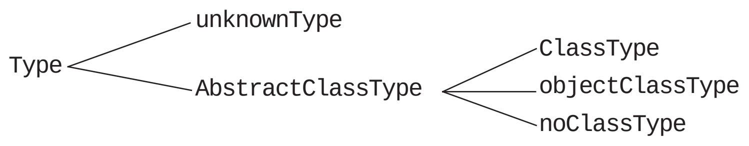


Figure 11.5 Extensions to `Type` class hierarchy

The specification of `AbstractClassType` and its specializations is shown in Figure 11.6. The function call `x.subclassOrEqual(y)` returns true if `x` is a subclass of `y`, or if `x` and `y` are the same class. The function also returns true if `x` is the constant object `noClassType`, to indicate that “NoClass” is regarded as a subclass of all other classes. The function call `x.getPrefixPath` returns the prefix path of the class `x`. The prefix path is defined as `emptyPath` for the `object-ClassType` and the `noClassType`.

`ClassType` objects are linked together to a class hierarchy by the local attribute `prefixClass`. The `ClassType` also has a local attribute `owner` used for defining the `getPrefixPath` function.

```
AbstractClassType: class Type (* abstract *)
{ subclassOrEqual: func boolean (ct: ref AbstractClassType);
  getPrefixPath: func ref SearchPath fix;
};

ClassType: class AbstractClassType
{ loc prefixClass: ref AbstractClassType;
  loc owner: ref ClassDoor;
  impl getPrefixPath :- owner.prefixPath;
  impl subclassOrEqual :=
    if ct == this ClassType
    then true
    else
      if ct == noClassType
      then false
      else prefixClass.subclassOrEqual(ct);
}

objectClassType: object AbstractClassType
{ impl getPrefixPath :- emptyPath;
  impl subclassOrEqual := ct == objectClassType;
};

noClassType: object AbstractClassType;
{ impl getPrefixPath :- emptyPath;
  impl subclassOrEqual := true;
};
```

Figure 11.6 Specification of class types

11.2.3 ClassDoor interface

The objects *a* and *b* of Figure 11.3 have to be communicated to the `ClassDoor` by inherited attributes. For *b* (the static path of the enclosing block), we use an inherited attribute `encPath`, exactly as for the `BlockDoor`. To obtain a reference to *a* (the prefix path of the superclass), we add an inherited attribute

```
syn syntPrefixClass: ref AbstractClassType
```

which is the superclass according to the user program syntax tree. The prefix path is obtained by calling the function `getPrefixClass` of this attribute.

Similar to `BlockDoor`, the `ClassDoor` has synthesized attributes `locPath` and `table`, to be used in an analogous way. The `ClassDoor` also has a synthesized attribute `classTp` which denotes the `ClassType` part object. Finally, as discussed in §3.3.2, cyclic subclassing must be detected. For this type of static semantic error we will use a history-dependent error-detection policy, similar to the handling of multiple declarations: The class declaration whose prefix was last changed is considered as causing the cycle. A synthesized attribute `causesCycle` will be true for this class declaration. The resulting interface of `ClassDoor` is shown in Figure 11.7.

```
ClassDoor: door
{ inh encPath: ref SearchPath fix;
  inh syntPrefixClass: ref AbstractClassType;
  syn locPath: ref SearchPath fix;
  syn table: ref SymbolTable fix;
  syn classTp: ref ClassType fix;
  syn causesCycle: boolean;
}
```

Figure 11.7 Interface to `ClassDoor`

11.2.4 Main grammar extension

The main grammar of §8.9.4 can be extended to include class declarations by adding a node class `ClassDecl` which makes use of a `ClassDoor` as shown in Figure 11.8.

```
ClassDecl: cons Decl
  (prefix: ref Use, declID: ref ID, b: ref Block)
{ cDoor: doorobject ClassDoor;
  dDoor: doorobject DeclDoor;
  loc prefixTypeError: boolean;
  loc cyclicClassError: boolean;
  eq cDoor.encPath :- path;
  eq b.path :- cDoor.locPath;
  eq b.table :- cDoor.table;
  eq cDoor.syntPrefixClass :-                (* 1 *)
      if prefix.tp in AbstractClassType
      then prefix.tp
      else objectClassType;
  eq dDoor.tp :- cDoor.classTp;
  eq dDoor.ident :- declId.ident;
  eq dDoor.table :- table;
  eq prefixTypeError :=                      (* 2 *)
       not prefix.tp in AbstractClassType;
  eq cyclicClassError := cDoor.causesCycle;    (* 3 *)
};
```

Figure 11.8 Specifying class declarations in a main grammar

A `ClassDecl` node has three son nodes: `prefix` (a `Use` node for the name application of the prefix class), `declID` (the name of the declared class), and `b` (the body of the class). The `ClassDecl` further declares two door objects: `cDoor` (a `ClassDoor`) and `dDoor` (a `DeclDoor`). The bulk of the equations are similar to those of the node classes `BlockStmt` and `VarDecl`. However, some of the equations in `ClassDecl` deserve some comments:

One type of static semantic error which can occur in `ClassDecl` is that the `prefix` may be a name application denoting some other entity than a class, e.g. an integer as in the following erroneous Simula program:

```
begin
   integer x;

   x class A;    (* Error: Prefix is not a class *)
   begin
   end;
end;
```

This error is detected by equation `(* 2 *)` which checks if the `tp` attribute of the son `prefix` is at least an `AbstractClassType` object. The equation `(* 1 *)` also takes this situation into account by defining the inherited attribute `syntPrefixClass` of the `ClassDoor` as `objectClassType` in case the `tp` attribute of `prefix` is not at least an `AbstractClassType`.

A second type of static semantic error which can occur in `ClassDoor` is cyclic subclassing as in the following erroneous Simula program:

```
A class B;    (* B is a subclass of A *)
begin
end;

B class A;    (* A is a subclass of B *)
begin
end;
```

In equation `(* 3 *)`, the attribute `causesCycle` is defined as true for only one of the classes in a cycle. Which one depends on the editing history.

11.2.5 Specification of ClassDoor

A first version of the `ClassDoor` specification is shown in Figure 11.9. This definition will be

```
ClassDoor: door
{ inh encPath: ref SearchPath fix;
  inh syntPrefixClass: ref AbstractClassType;
  syn locPath: ref SearchPath fix;
  syn table: ref SymbolTable fix;
  syn classTp: ref ClassType fix;
  syn causesCycle: boolean;

  theClassType: object ClassType;
  eq classTp :- theClassType;
  eq theClassType.owner :- this ClassDoor;
  eq theClassType.prefixClass :-               (* 1 *)
    if syntPrefixClass.subclassOrEqual(theClassType)
    then objectClassType
    else syntPrefixClass;

  theTable: object SymbolTable;
  theTablePath: object SymbolTablePath;
  eq table :- theTable;
  eq theTablePath.table :- theTable;

  prefixPath: object TwoPath;
  eq prefixPath.first :- theTablePath;
  eq prefixPath.second :-                      (* 2 *)
      theClassType.prefixClass.getPrefixPath;

  staticPath: object TwoPath;
  eq locPath :- staticPath;
  eq staticPath.first :- prefixPath;
  eq staticPath.second :- encPath;

  eq causesCycle :=                            (* 3 *)
    syntPrefixClass =/= theClassType.prefixClass;
};
```

Figure 11.9 First version of `ClassDoor` specification

slightly refined during the dependency analysis. Most of the equations are straight-forward, defining the `SearchPath` connections in a similar way as for the `BlockDoor`. However, equations `(* 1 *)`, `(* 2 *)`, and `(* 3 *)` deserve some comments.

Equation `(* 1 *)` defines the attribute `theClassType.prefixClass`. Normally, this attribute will get the value of the inherited attribute `syntPrefixClass`. However, in case this would introduce a cyclic class hierarchy, the `prefixClass` attribute is instead assigned a reference to the constant object `objectClassType`. Thus, this equation guarantees that `ClassType` objects chained together by the reference attribute `prefixClass` cannot form a cyclic chain. This is necessary since the function `subclassOrEqual` would loop otherwise.

Equation `(* 2 *)` defines the attribute `second` of `prefixPath` in terms of the `prefixClass` attribute. Since the `prefixClass` attributes cannot form a cycle, this prevents also the `prefixPath` objects to be connected in a cycle. This is necessary since a cyclic visibility graph would cause the function `lookup` to loop.

Equation `(* 3 *)` defines the attribute `causesCycle` as *true* if the `syntPrefixClass` attribute differs from `theClassType.prefixClass`. This indicates a syntactic cycle in the subclass hierarchy which is broken at this particular `ClassDoor`. For other `ClassDoor` objects on the same syntactic subclass cycle the `causesCycle` attribute will have the value *false*. For `ClassDoor` objects which are not on any cycle, the `causesCycle` attribute will also have the value *false*.

11.2.6 Dependency analysis

The dependency analysis of `ClassDoor` involves two kinds of non-local dependencies. One concerning cyclic subclassing and one concerning changes to edges of the visibility graph.

11.2.6.1 Cyclic subclasses

The attribute `ClassType.prefixClass` depends on non-local information. The equation defining this attribute (`(* 1 *)` in Figure 11.9) accesses the `subclassOrEqual` function which in turn accesses the `prefixClass` attribute of other `ClassType` objects. There is thus a static dependency

(`ClassType.prefixClass`, `ClassType.prefixClass`).

This non-local dependency is modelled by a receive vertex *v*(`cycStatus`) and a send vertex *v*(`cycStatus`, `ClassDoor`, `fCyclic`) in the dependency graph for `ClassDoor` as shown in Figure 11.10.

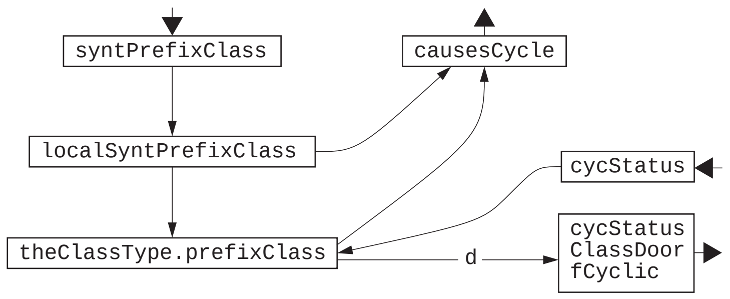


Figure 11.10 Partial dependency graph for `ClassDoor`

A local copy attribute `localSyntPrefixClass` has been added. This will allow the dependency function `fCyclic` to determine if other `ClassDoor` objects are considered to cause a cycle without consulting non-fix inherited attributes whose values may have changed after the latest evaluation of the `ClassDoor` objects. The equations defining `theClassType.prefixClass` and `causesCycle` are updated accordingly as shown in Figure 11.11.

```
addto ClassDoor
{ loc localSyntPrefixClass: ref AbstractClassType;
  eq localSyntPrefixClass :- syntPrefixClass;
  eq theClassType.prefixClass :-              (* 1' *)
    if localSyntPrefixClass.subclassOrEqual(theClassType)
    then objectClassType
    else localSyntPrefixClass;
  eq causesCycle :=                           (* 3' *)
    localSyntPrefixClass =/= theClassType.prefixClass;
}
```

Figure 11.11 Extension and refinement of `ClassDoor` by the local copy attribute `localSyntPrefixClass`

It is possible for a change to the inherited attribute `syntPrefixClass` of a `ClassDoor` *x* to break or introduce a syntactic subclass cycle. If a cycle is introduced, the cause of the cycle will be associated with *x*. If a cycle is broken, the cause can be either in *x* or in any of the other `ClassDoor` objects involved in the cycle. In the latter case, there is a non-local dependency to another `ClassDoor` object *y* (the one on the cycle whose `causesCycle` attribute is true). Since it is only the *breaking* of a cycle which can affect non-local doors, the edge from *v*(`theClassType.prefixClass`) to the send vertex is a *de-evaluation* edge rather than a normal edge.

The de-evaluation of `theClassType.prefixClass` will break a cycle caused by another `ClassDoor` object *y* iff:

- *y* is a superclass of *x*, and
- *y*'s attribute `causesCycle` is true, and
- the syntactic superclass of *y* is a subclass or equal to *x*

The dependency function `fCyclic` should in this case return a singleton set containing *y*. Otherwise, `fCyclic` should return an empty set. To find a superclass causing a cycle, we use a recursive function `findCycleCauser` in class `AbstractClassType` as shown in Figure 11.12. The implementation of the function `fCyclic` is given in Figure 11.13.

The implementation of `findCycleCauser` in `ClassType` accesses the synthesized attribute `causesCycle`, which in turn (since it is a demand attribute) accesses the local copy of the inherited attribute `syntPrefixClass`. By accessing the local copy instead of the inherited attribute directly, the `causesCycle` attribute is insured to return the value used in the latest evaluation of the `ClassDoor`, and not be dependent on possible changes or re-evaluations in the enclosing syntax tree.

```
addto AbstractClassType
{ findCycleCauser: func ref ClassType (* may return NONE *)
    :- NONE;
};

addto ClassType
{ impl findCycleCauser :-
    if owner.causesCycle
    then this ClassType
    else prefixClass.findCycleCauser;
};
```

Figure 11.12 The function `findCycleCauser`

```
emptyClassDoorSet: object Set[ClassDoor];

addto ClassDoor
{ fCyclic: func ref Set[ClassDoor] :-
    inspect $ct :-
      theClassType.prefixClass.findCycleCauser
    when ClassType do
      if $ct.owner.localSyntPrefixClass.
              subclassOrEqual(theClassType)
      then emptyClassDoorSet.add($ct.owner)
      else emptyClassDoorSet
    otherwise emptyClassDoorSet;
};
```

Figure 11.13 Extension of `ClassDoor` by the dependency function `fCyclic`

11.2.6.2 Changes to visibility graph edges

The attribute `prefixPath.second` in `ClassDoor` depends on the non-fix inherited attribute `syntPrefixClass` and may thus change during incremental evaluation. This attribute models an edge in the visibility graph and a change to it may affect the bindings of `UseDoor` objects. This outgoing non-local dependency in `ClassDoor` is modelled by a send vertex *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`), where `fAttemptedEdge` is a dependency function returning the set of `UseDoor` objects which have attempted to bind via the `prefixPath.second` edge.

We will keep track of the affected `UseDoor` objects by maintaining a collection of `UseDoor` objects for this visibility graph edge. To program this in a simple way, we introduce a new class `WatchPath` which is a specialization of `SearchPath` as shown in Figure 11.14. A `WatchPath` object models a path vertex with one outgoing edge. The collection `attemptedEdge` in `WatchPath` collects all `UseDoor` objects which have attempted to bind via the `WatchPath` object. Thus, UseDoor objects attempting to bind via a given edge can be collected simply by redefining that edge to go via a WatchPath object.

A condition cAttemptedEdge is added to UseDoor to maintain the attemptedEdge collections of WatchPath objects. The condition simply states that the UseDoor object is a member of the attemptedEdge collection of all WatchPath objects encountered during lookup. This condition has dependencies similar to the cUses and cAttempted conditions, and is straight-forward to add to UseDoor.

```
WatchPath: class SearchPath
{ loc path: ref SearchPath;
  collection: attemptedEdge:
      object UnorderedCollection[UseDoor];
  impl lookup :- path.lookup(ident);
};

addto UseDoor
{ cAttemptedEdge: cond ...;
};
```

Figure 11.14 The class WatchPath

Figure 11.15 shows how the edge outgoing from prefixPath.second in ClassDoor is redefined to go via a WatchPath object, and how the dependency function fAttemptedEdge is implemented. This function corresponds to solving the incremental name analysis problem IV (change the visibility graph) treated in §3.4. The solution, using WatchPath objects, is an implementation of method 6 (maintain traces).

```
addto ClassDoor
{ theWatchPath: object WatchPath;
  eq prefixPath.second :- theWatchPath;        (* 2' *)
  eq theWatchPath.path :-
       theClassType.prefixClass.getPrefixPath;

  fAttemptedEdge: func ref Set[UseDoor] :-
    theWatchPath.attemptedEdge.contents;
};
```

Figure 11.15 Extension and refinement of ClassDoor by theWatchPath and the dependency function fAttemptedEdge

The final dependency graph for ClassDoor is shown in Figure 11.16.

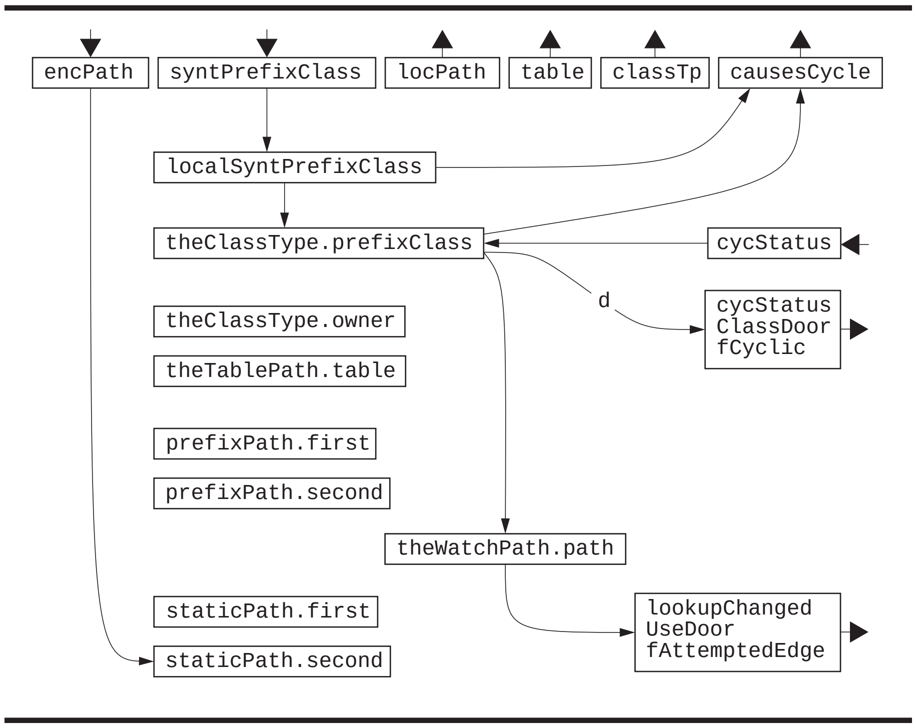


Figure 11.16 Dependency graph for `ClassDoor`

11.2.7 Visit procedures

The visit procedures `exhDeEvalVisit` and `exhEvalVisit` for `ClassDoor` are implemented according to the basic visit procedure algorithm. The visit procedure `incOwnerVisit` has transitive non-local dependents (via the visit procedures for `cycStatus`) which overlap the direct non-local dependents. The `incOwnerVisit` must thus be modified in order to not violate the preconditions of the visit procedures of its dependent doors. In all other respects, the `incOwnerVisit` and the `eval`/`deEval` pair for `cycStatus` are implemented according to the basic algorithm.

The characteristic sets for `incOwnerVisit` and `cycStatus` are as follows:

	S_{cond}	S_{loc}	S_{send}
`incOwner-Visit`	∅	{ *v*(`localSyntPrefixClass`), *v*(`theClassType.prefixClass`), *v*(`theWatchPath.path`) }	{ *v*(`cycStatus`, `ClassDoor`, `fCyclic`), *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`) }
`deEval/Eval cycStatus`	∅	{ *v*(`theClassType.prefixClass`), *v*(`theWatchPath.path`) }	{ *v*(`cycStatus`, `ClassDoor`, `fCyclic`), *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`) }

The dependency set computed by `fCyclic` will actually always be empty when called from the `deEvalCycStatus` procedure. This is because `fCyclic` will only be called for a `ClassDoor` object *y* which causes a cycle. The attribute `theClassType.prefixClass` of *y* will in this case denote `objectClassType`, and the function `fCyclic` of *y* will return the empty set. We therefore remove the send vertex *v*(`cycStatus`, `ClassDoor`, `fCyclic`) from S_{send} of `cycStatus` and obtain the following simplified characteristic sets:

	S_{cond}	S_{loc}	S_{send}
`incOwner-Visit`	∅	{ *v*(`localSyntPrefixClass`), *v*(`theClassType.prefixClass`), *v*(`theWatchPath.path`) }	{ *v*(`cycStatus`, `ClassDoor`, `fCyclic`), *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`) }
`deEval/Eval cycStatus`	∅	{ *v*(`theClassType.prefixClass`), *v*(`theWatchPath.path`) }	{ *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`) }

Overlapping dependency sets can occur for an `incOwnerVisit` procedure. Consider a `ClassDoor` object `X` with two dependency sets `X.fCyclic` and `X.fAttemptedEdge`. If `X.fCyclic` contains another `ClassDoor` object `Y`, the dependency sets `X.fAttemptedEdge` and `Y.fAttemptedEdge` may overlap. The basic algorithm for `incOwnerVisit` must be modified to ensure that the `deEval` procedure is called only once for each of these dependent doors.

As an actual example of overlapping dependency sets, consider the (erroneous) Simula program of Figure 11.17. This program contains a cyclic subclassing hierarchy (`Y` is declared as a subclass of `X` and `X` is declared as a subclass of `Y`). Class `Y` (the uppermost class) is considered to cause the cycle.

```
            begin
               X class Y; (* Error: Class Y causes cycle *)
               begin
                  p := q;(* Error: p and q are undeclared *)
               end;
 edit point
 ──────────►  [Y] class X;
               begin
                  r := s;(* Error: r and s are undeclared *)
               end;

               class Z;
               begin
                  integer p, q, r, s;
               end;
            end;
```

Figure 11.17 Breaking a syntactic subclass cycle by changing `Y` to `Z`.

Consider changing the program by replacing the prefix of class `X` (the middle class) from `Y` to `Z`. This change would break the syntactic subclassing cycle and result in a correct Simula program where all of the name applications `p`, `q`, `r`, and `s` become bound to the corresponding declarations in `Z`. Such an edit results in a call to the `incOwnerVisit` procedure of `ClassDoor` `X`, giving

```
X.fCyclic = {Y}
```

Since the name applications `p`, `q`, `r`, and `s` are all undeclared, the dependency sets of `UseDoor` objects for `X` and `Y` are

```
Y.fAttemptedEdge = {p, q, r, s}
   X.fAttemptedEdge = {r, s}
```

and these dependency sets are thus overlapping.

The `incOwnerVisit` procedure can be modified in a simple way in order to handle this overlap. The idea is to delay the computation of `Y.fAttemptedEdge` until all the `UseDoor` objects of `X.fAttemptedEdge` have been de-evaluated. At this point, the overlapping `UseDoor` objects will no longer be part of the set computed by `Y.fAttemptedEdge`, since the de-evaluation of these `UseDoor` objects have removed them from all `attemptedEdge` collections. Figure 11.18 shows the resulting implementation of `incOwnerVisit`.

The `incOwnerVisit` procedure is constructed according to the basic algorithm, but special care has been taken to order the two de-evaluation iterations of step 2 so that the `deEvalLook-upChanged` procedure of the `UseDoor` objects in `attemptedSet` is called before the `deEvalCycStatus` procedure of the `ClassDoor` objects in `cycStatusSet` (i.e. `Y`) is called. Thus, the dependency set of `UseDoor` objects computed by `Y` will not contain any of the `UseDoor` objects de-evaluated so far. The reverse ordering of Step 2 would lead to overlapping dependency sets and violate the legal evaluation state transitions for the overlapping `UseDoor` objects.

```
addto ClassDoor
{ impl incOwnerVisit
  { cycStatusSet: ref Set[ClassDoor]; (* dependency set *)
    attemptedSet: ref Set[ClassDoor]; (* dependency set *)

    (* Step 1. Compute dependent doors *)
    cycStatusSet :- fCyclic;
    attemptedSet :- fAttemptedEdge;

    (* Step 2. De-evaluate dependent doors *)
    for $d :- attemptedSet.each do $d.deEvalLookupChanged; end for;
    for $d :- cycStatusSet.each do $d.deEvalCycStatus; end for;

    (* Step 4. Evaluate local invariants *)
    eval localSyntPrefixClass;
    eval theClassType.prefixClass;
    eval theWatchPath.path;

    (* Step 5. Evaluate dependent doors *)
    for $d :- cycStatusSet.each do $d.evalCycStatus; end for;
    for $d :- attemptedSet.each do $d.evalLookupChanged; end for;

    (* Step 6. Add dependent doors to work list *)
    evaluator.worklist.addSet(cycStatusSet);
    evaluator.worklist.addSet(attemptedSet);
  };
};
```

Figure 11.18 Procedure `incOwnerVisit` for `ClassDoor`

The visit procedures `deEvalCycStatus` and `evalCycStatus` are implemented according to the basic algorithm. The dependency sets of these procedures must be stored outside the procedures as mentioned in the discussion of algorithm 10-3. This is accomplished simply by storing them in the `ClassDoor` itself as shown in Figure 11.19.

```
addto ClassDoor
{ (* dependency set for cycStatus deEval/eval procedures *)
  attemptedSet: ref Set[ClassDoor];

  deEvalCycStatus: proc
  { (* Step 1. Compute dependent doors *)
    attemptedSet :- fAttemptedEdge;

    (* Step 2. De-evaluate dependent doors *)
    for $d :- attemptedSet.each do $d.deEvalLookupChanged; end for;
  };

  evalCycStatus: proc
  { (* Step 4. Evaluate local invariants *)
    eval theClassType.prefixClass;
    eval theWatchPath.path;

    (* Step 5. Evaluate dependent doors *)
    for $d :- attemptedSet.each do $d.evalLookupChanged; end for;

    (* Step 6. Add dependent doors to work list *)
    evaluator.worklist.addSet(attemptedSet);
  };
};
```

Figure 11.19 Procedures `deEvalCycStatus` and `evalCycStatus`

11.2.8 Circular dependencies

En erroneous program with a cyclic class hierarchy corresponds to an intuitive cyclic chain of dependencies, as discussed in §3.3.6. However, although the `ClassDoor` handles and breaks such cycles, the attributes in a `ClassDoor` are themselves not involved in any cyclic attribute dependencies. I.e., introducing and/or breaking cycles in the class hierarchy of the user program does not lead to cyclic attribute evaluation.

Section §10.9 discussed how potential cyclic dependencies could be identified by matching send and receive vertices in the door dependency graphs. Doing this for the `ClassDoor` indicates a potential cycle between two or more `ClassDoor` objects via the `cycStatus` send and receive vertices. However, from the definition of the dependency function `fCyclic` it is clear that such a cycle cannot actually occur. This was discussed in §11.2.7 where we noted that the `fCyclic` dependency set for a receiving `ClassDoor` is always empty.

11.3 References and remote access

References and remote access are simple to handle based on the facilities for classes of the previous section. Only some small extensions are needed to the door package.

11.3.1 Extensions to door package

To handle reference types, we extend the `Type` class hierarchy by new specializations which parallel those for class types as shown in Figure 11.20. There is a one-to-one correspondence between class types and references types. Given a class type, the corresponding reference type can be accessed and vice versa. This is implemented by a function `getRefType` in `AbstractClassType` and a function `getClassType` in `AbstractRefType`. The reference types are specified in Figure 11.21 and the additions to the class types in Figure 11.22.

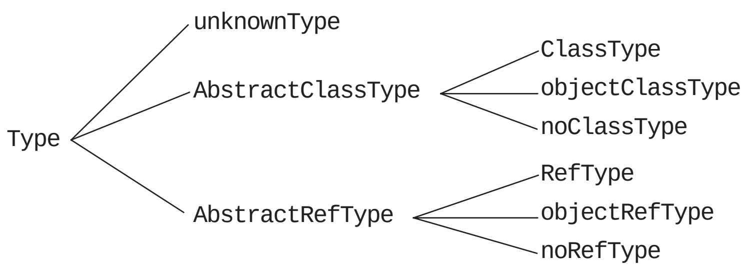


Figure 11.20 Extensions to `Type` class hierarchy

```
AbstractRefType: class Type (* abstract *)
{ getClassType: func ref AbstractClassType fix;
};

RefType: class AbstractRefType
{ loc classTp: ref ClassType;
  impl getClassType :- classTp;
};

objectRefType: object AbstractRefType
{ impl getClassType :- objectClassType;
};

noRefType: object AbstractRefType;
{ impl getClassType :- noClassType;
};
```

Figure 11.21 Specifications of reference types

```
addto AbstractClassType
{ getRefType: func ref AbstractRefType fix;
};

addto ClassType
{ refTp: object RefType;
  impl getRefType :- refTp;
};

addto objectClassType
{ impl getRefType :- objectRefType;
};

addto noClassType
{ impl getRefType :- noRefType;
};
```

Figure 11.22 Additions to class types

Note that `ClassType` declares a `RefType` part object and that `RefType` has a local attribute `classTp`. This implies that the `ClassDoor` must be extended to define the local attribute of `RefType` objects as shown in Figure 11.23. A corresponding evaluation statement must be added to step 4 of the procedure `exhEvalVisit` of `ClassDoor`. This completes the extension to the basic door package to handle classes and subclasses.

```
addto ClassDoor
{ eq theClassType.refTp.classTp :- theClassType;
};
```

Figure 11.23 Addition to `ClassDoor`

11.3.2 Extensions to the main grammar

11.3.2.1 Declaration of reference variables

The main grammar can be extended to allow declarations of reference variables simply by adding a new specialization of the node class `DeclType`. This new node class, `RefDeclType`, is shown in Figure 11.24. The `Use` son node should be an identifier bound to a class declaration, i.e. the `tp` attribute of the `Use` node should be at least an `AbstractClassType`. A local attribute `error` detects if this is the case or not `(* 1 *)`. The resulting type of the `RefDeclType` node is defined as the reference type corresponding to the class type of the `Use` node (or `objectRefType` in case of an error) `(* 2 *)`. Note that the call to `getRefType` is legal since this is a fix function. (Recall that calls to non-fix functions via reference attributes are legal only in the door package.)

```
RefDeclType: cons DeclType(cls: ref Use)
{ loc error: boolean;
  eq error := not cls.tp in AbstractClassType;  (* 1 *)
  eq tp :-                                          (* 2 *)
      inspect $c :- cls.tp
      when AbstractClassType do $c.getRefType
      otherwise objectRefType;
};
```

Figure 11.24 Node class `RefDeclType`

11.3.2.2 Remote access

The principles for remote access were described in §3.3.3. For a remote access

```
a.b
```

the binding of `b` depends on the type of `a`. In Simula, `a` must be a reference variable, or a procedure returning a reference. The left-hand side of the remote access (to the left of the dot) can also be a more complex expression, e.g. another remote access, a “qua”, or a “this” expression.

We extend the main grammar by a node class `RemoteAccess` which has an expression on the left-hand side and a `Use` node on the right hand side (Figure 11.25). A static semantic error occurs if the type of the left hand side is not a reference type. The local attribute `error` detects if this is the case or not `(* 1 *)`.

To bind the right-hand side in a suitable environment, the `path` attribute of the right-hand side is defined as the prefix path of the class of the left hand side reference `(* 2 *)`. By this equation, the right-hand-side `path` attribute is dependent on the `tp` attribute of the left hand side, and will receive a new value if the `tp` attribute changes. The right-hand-side `path` is thus a non-fix attribute. The `Use` node will use this non-fix attribute to define the `path` attribute of its `UseDoor`. It was for this reason we did not require the inherited attribute `path` of `UseDoor` to be a fix attribute in §8.9.2.4. This allows the `UseDoor` to be used also for remote accesses.

The resulting type of the remote access expression is the same as the type of the right hand side `(* 3 *)`.

```
RemoteAccess: cons Exp(lhs: ref Exp, rhs: ref Use)
{ loc error: boolean;
  eq error := not lhs.tp in AbstractRefType;   (* 1 *)
  eq rhs.path :-                               (* 2 *)
      inspect $r :- lhs.tp
      when AbstractRefType do $r.getClassType.getPrefixPath
      otherwise objectClassType.getPrefixPath;
  eq tp :- rhs.tp;                             (* 3 *)
};
```

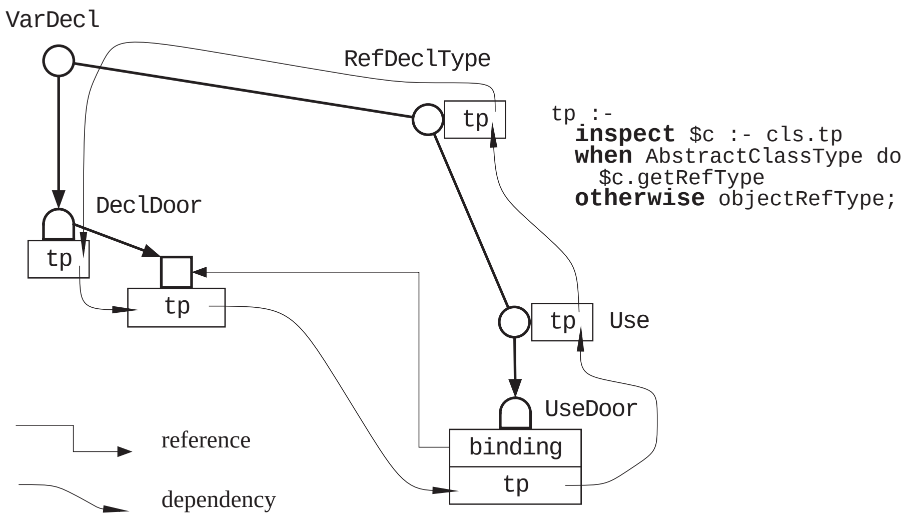
Figure 11.25 Node class RemoteAccess

11.3.3 Circular dependencies

It was noted in §3.3.6 that an erroneous (but syntactically correct) declaration of a reference variable:

```
ref (a) a;
```

leads to an intuitively cyclic dependency. An attributed syntax tree for this declaration, according to the main grammar extended by `RefDeclType` in §11.3.2.1, results in a corresponding circular chain of attribute dependencies as shown in Figure 11.26. The evaluation converges and results in a suitable attribution and error-detection.



```
tp :-
  inspect $c :- cls.tp
  when AbstractClassType do
    $c.getRefType
  otherwise objectRefType;
```

Figure 11.26 Circular dependency chain for erroneous declaration of reference

This circularity is similar to the one for “like-types” discussed in §10.9.1, but in contrast to the “liketype” example, this circularity has exactly one solution, i.e. the grammar is determined for this problem: If one or more declarations of reference variables are involved in a cycle, the type of all these declarations will be `objectRefType`. This is insured by the equation defining `tp` in `RefDeclType` as shown in the figure. The rest of the equations on the dependency chain are copy equations. Furthermore, the `error` attribute of the `RefDeclType` will be true for all the declarations on the cycle since the `UseDoor` should be bound to a declaration of a class and not a declaration of a reference variable.

The evaluation converges after only one cycle in the evaluation. Figure 11.27 shows the visit procedure calls resulting from adding the erroneous declaration.

```
evaluator.replaceSubtree
  (* Exhaustive phase *)
  VarDecl.exhVisit
    RefDeclType.exhVisit
      Use.exhVisit
        UseDoor.exhEvalVisit
          eval binding (->  UseDoor.binding == nullEntry,
                            UseDoor.tp == unknownType)
    DeclDoor.exhEvalVisit
      UseDoor.deEvalLookupChanged
      eval theEntry.ident
      eval theEntry.tp (-> theEntry.tp == objectRefType)
      UseDoor.evalLookupChanged
        eval binding (->  UseDoor.binding == DeclDoor.theEntry,
                          UseDoor.tp == objectRefType)

  (* Non-local incremental phase *)
  Use.incDoorVisit
    RefDeclType.incSonVisit
      VarDecl.incSonVisit
        DeclDoor.incOwnerVisit
          (* converged value for tp *)
```

Figure 11.27 Visit procedure calls during evaluation of cyclic dependency

11.4 Type-checking reference assignments

In §3.5 we discussed type checking for object-oriented languages, and noted that this involves comparison of formal qualifications.

To type-check a reference assignment, `rA :- rB`, the formal qualifications of `rA` and `rB` can be compared by using the function `subclassOrEqual` in `AbstractClassType`. However, since this function is non-fix it must not be called directly from the main grammar, but only from inside a door object. A new door class `CompareClassDoor` is therefore added to the door package. A straight-forward way of designing this door would be to give it two inherited attributes for the two class types to be compared, and one synthesized attribute for the result of the `subclas-`

sOrEqual function called on one of the inherited attributes. However, this would require the node class for the reference assignment to use two door objects since comparisons are needed in both directions. Instead, we will use another design where the CompareClassDoor does both comparisons.

```
CompareClassDoor: door
{ inh shouldBeGenClass: ref AbstractClassType;
  inh shouldBeSpecClass: ref AbstractClassType;
  syn reverseOrder: boolean;
  syn incomparable: boolean;
  orderOK: func boolean :=
      shouldBeSpecClass.subclassOrEqual(shouldBeGenClass);
  eq reverseOrder :=
      if orderOK
      then false
      else
        shouldBeGenClass.subclassOrEqual(shouldBeSpecClass);
  eq incomparable :=
      if orderOK
      then false
      else not
        shouldBeGenClass.subclassOrEqual(shouldBeSpecClass);
};
```

Figure 11.28 CompareClassDoor

Figure 11.28 shows the specification of CompareClassDoor. The two inherited attributes shouldBeGenClass and shouldBeSpecClass stand for “should be the general class” and “should be the specialized class”. The idea is that when doing a class comparison in type-checking, the normal (and statically checkable) case is that one of the classes is more general than the other. As an example, in the Simula reference assignment, the left-hand side should be more general than the right-hand side. The synthesized attribute reverseOrder is defined as true, if shouldBeGenClass is, in fact, a specialization of shouldBeSpecClass. The synthesized attribute incomparable is defined as true if the two classes are incomparable, i.e. if neither class is more general or special than the other one.

11.4.1 Extension to main grammar

The reference assignment in Simula can be modelled as shown in Figure 11.29. The node class RefAssignStmt uses the CompareClassDoor to compare the formal qualifications of the left-hand side and the right-hand side according to the cases listed in §3.5. The local attributes errorLhs, errorRhs, errorInc detect static semantic errors in the assignment. The local attribute `rtCheckNeeded` detects if a run-time check is needed to test the actual qualification of the right-hand side before assignment.

```
RefAssignStmt: cons Stmt(lhs: ref Exp, rhs: ref Exp)
{ loc errorLhs: boolean; (* True if lhs not a reference *)
  loc errorRhs: boolean; (* True if rhs not a reference *)
  loc errorInc: boolean; (* True if lhs incomparable to rhs *)
  loc rtCheckNeeded: boolean; (* True if runtime check needed *)
  cDoor: doorobject CompareClassDoor;
  eq cDoor.shouldBeGenClass :-
    inspect $r :- lhs.tp
    when AbstractRefType do $r.getClassType
    otherwise objectClassType;
  eq cDoor.shouldBeSpecClass :-
    inspect $r :- rhs.tp
    when AbstractRefType do $r.getClassType
    otherwise noClassType;
  eq errorLhs := not lhs.tp in AbstractRefType;
  eq errorRhs := not rhs.tp in AbstractRefType;
  eq errorInc := cDoor.incomparable;
  eq rtCheckNeeded := cDoor.reverseOrder;
};
```

Figure 11.29 Extension to main grammar: RefAssignStmt

11.4.2 Dependency analysis

We now do a dependency analysis for the `CompareClassDoor` (Figure 11.28).

Both the synthesized attributes `reverseOrder` and `incomparable` of the `CompareClassDoor` depend on the `prefixClass` attributes of `ClassType` objects. We model this non-local dependency by a receive vertex *v*(`compare`) in `CompareClassDoor` and a send vertex *v*(`compare`, `CompareClassDoor`, `fComps`) in `ClassDoor`. The `CompareClassDoor` objects which depend on the `prefixClass` attribute of a given `ClassType` object can be kept track of in a collection object `comps` in the `ClassType` object, and a condition `cComps` can be added to `CompareClassDoor` to maintain these collections.

In deducing which `prefixClass` attributes a given `CompareClassDoor` object `x` depends on, it is useful to consider actual dependencies rather than access dependencies. We first note that the values of the synthesized attributes of `x` cannot change unless any of the `prefixClass` attributes corresponding to a lattice edge between `x.shouldBeGenClass` and `x.shouldBeSpecClass` changes. We also note that if any of these inherited attributes denotes either of the constant objects `objectClassType` or `noClassType`, changes to the `prefixClass` attributes cannot affect the values of the synthesized attributes.

Local copy attributes for the inherited attributes `shouldBeGenClass` and `shouldBeSpecClass` are added to allow de-evaluation of the `cComps` condition. The additions are shown in Figure

11.30 and the resulting dependency graphs in Figure 11.31 and Figure 11.32. For brevity, the definition of `cComps` is only sketched.

```
addto ClassType
{ comps: object UnorderedCollection[CompareClassDoor];
}

addto CompareClassDoor
{ loc localShouldBeGenClass: ref AbstractClassType;
  loc localShouldBeSpecClass: ref AbstractClassType;
  eq localShouldBeGenClass :- shouldBeGenClass;
  eq localShouldBeSpecClass :- shouldBeSpecClass;
  cComps: cond
    if localShouldBeGenClass == objectClassType or
        localShouldBeGenClass == noClassType or
        localShouldBeSpecClass == objectClassType or
        localShouldBeSpecClass == noClassType
    then true
    else ...; (*  member of ClassType.comps for lattice edges
                  between localShouldBeGenClass and
                  localShouldBeSpecClass *)
};

addto ClassDoor
{ fComps: func ref Set[CompareClassDoor]
    :- theClassDoor.comps.contents;
};
```

Figure 11.30 Additions due to dependency analysis of CompareClassDoor

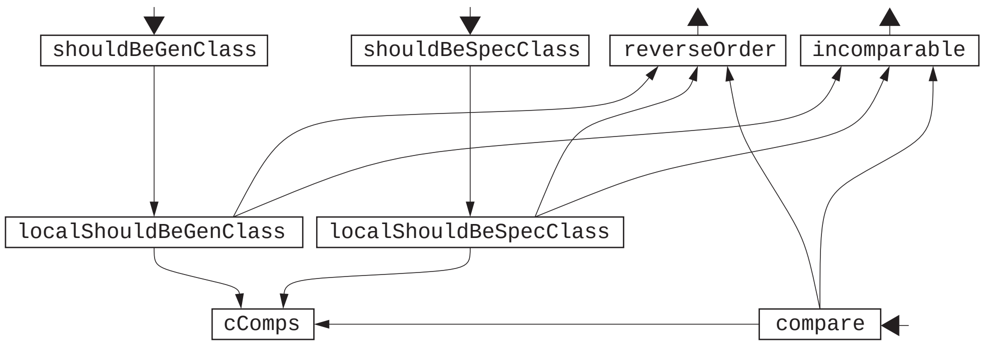


Figure 11.31 Dependency graph for `CompareClassDoor`

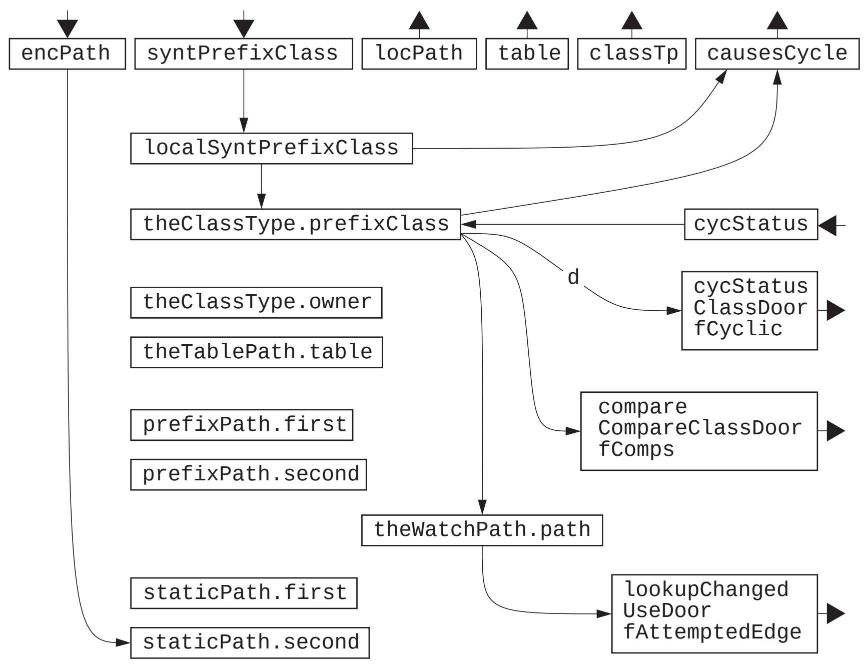


Figure 11.32 Updated dependency graph for ClassDoor

The visit procedures for `CompareClassDoor` are straight-forward implementations according to the basic visit procedure algorithm. The extensions to the visit procedure for `ClassDoor` are also straight-forward.

The send vertex *v*(`compare`, `CompareClassDoor`, `fComps`) leads to a dependency set overlap for `incOwnerVisit` which is analogous to the dependency set overlap caused by the send vertex *v*(`lookupChanged`, `UseDoor`, `fAttemptedEdge`). This overlap can be handled in the same manner as described in §11.2.7.

11.5 Error detection

In our examples so far, detection of static-semantic errors has been specified by local boolean attributes in the main grammar. In an interactive environment, it should be possible to monitor the errors during program editing, e.g. by marking the erroneous constructs in the unparsed presentation of the program, or by maintaining a list of errors in a separate window. In either case, the presentation needs to be updated according to the incremental updates of the attribution. This can be accomplished by using doors as an interface mechanism to the incremental evaluation.

The visit procedures of these doors are implemented in order to achieve controlled side-effects in external components such as the editor and window system.

A door class `ErrorDoor` can be specified which has two inherited attributes: a boolean attribute `error` and a string attribute `message`. The informal semantics is the following: if `error` has the value true, there will be a presentation of the error in the appropriate window, associating the owner node of the `ErrorDoor` with the string `message` (an error message). The visit procedures can be written to add and remove the error presentation as appropriate, e.g. as shown in Figure 11.33. The `errorPresenter` used in this implementation could be a global object connected to the structure-oriented editor.

```
ErrorDoor: door
{ inh error: boolean;
  inh message: string fix;

  loc mark: ref ErrorMarking; (* May be NONE *)

  impl exhEvalVisit
  { if error then
      mark :- errorPresenter.AddMarker(owner, message);
    else
      mark :- NONE;
    end if;
  };

  impl exhDeEvalVisit
  { if mark =/= NONE then
      errorPresenter.RemoveMarker(mark);
      mark :- NONE;
    end if;
  };

  impl incOwnerVisit
  { if error and mark == NONE then
      mark :- errorPresenter.AddMarker(owner, message)
    else if not error and mark =/= NONE then
      mark :- errorPresenter.RemoveMarker(mark)
      mark :- NONE;
    end if;
  };
};
```

Figure 11.33 Class `ErrorDoor`

11.6 Discussion

The door package resulting from the extensions introduced in this chapter handles the major static-semantic problems occurring in object-oriented languages such as nested block structure, classes, reference variables, and remote access. In order to cover a full object-oriented language it is also necessary to support procedures and parameters. In §11.6.1 we sketch how this can be done. In §11.6.2 we discuss the possibility of supporting virtual classes in BETA.

11.6.1 Procedures and parameters

The most important functionality missing in our door package is the possibility for specifying procedures and parameters. This functionality is, however, straight-forward to add. The following sketch corresponds approximately to the facilities of the door package used in the Orm system:

- A door class `ProcDoor`, similar to `BlockDoor` and `ClassDoor`
- A class `ProcType`, similar to `ClassType`. The `ProcType` contains information about the parameter types and the return type.
- A door class `ParamDeclDoor`, for specifying the formal type for a parameter at a given index
- A door class `ParamUseDoor`, for retrieving the formal type of a parameter at a given index
- A class `Kind` which can be specialized to a number of constant objects, e.g. `ConstantKind`, `VariableKind`, `ReferenceParamKind`, `ValueParamKind`, etc.
- Extend the `DeclDoor` and `UseDoor` with a `Kind` attribute.

The use of virtual procedures in addition to ordinary non-virtual procedures has only a small effect on the needed functionality. Although virtual and non-virtual procedures have different dynamic semantics, the static-semantic analysis is done in practically the same manner for both these kinds of procedures. The main difference from a static-semantic point of view is that a virtual procedure may have implementations in subclasses which match the virtual specification. A virtual procedure implementation plays the role of both a name application and a name declaration. It is a name application in that it matches a name declared in a superclass. It is a name declaration in that there must be no other declarations with the same name in the same block. In many object-oriented languages, including Simula, a virtual procedure specification has to be explicitly specified in the syntax, whereas an ordinary procedure declaration will be either a non-virtual procedure or a virtual procedure implementation, depending on if there is a matching virtual specification in a superclass or not. This behavior can be built into the `ProcDoor`.

11.6.2 Supporting virtual classes in BETA

The Door Attribute Grammars as introduced in this thesis have the property that each semantic object is either constant or a part-object owned (transitively) by a syntax node. All semantic objects are thus *static* in the sense that their existence is goverened by the syntax tree alone. One could also imagine *dynamic* objects, whose existence depends on attribute values. Some form of dynamic objects is needed in order to handle the virtual superclasses of BETA. The declaration of an extended virtual class which is used as a superclass, leads to the implicit construction of new path vertices for subclasses to the virtual superclass, as discussed in §3.3.5. Since the number of such subclasses is not limited by the grammar, these new path vertices cannot be declared as transitive part objects of the syntax node. The existence of such an implicit path vertex depends both on the existence of the subclass in question, and on the existence of the extended superclass. If any of these declarations is removed, the implicit path vertex should be removed as well. Clearly, there is a need here for some mechanism for adding and removing semantic objects dynamically, as a consequence of other attribute values. We have not investigated this issue, but our guess is that it should be possible to add such a mechanism. However, this is an area for future research.

The ordinary use of virtual classes in BETA (corresponding to parametrized types), does not require dynamic objects to be introduced. This use of virtual classes can be based on the same visibility graphs as those used in our door package (combined block-structure and subclassing). This was also discussed in §3.3.5. The main difficulty when implementing ordinary virtual classes is that a kind of double lookup is needed when binding an identifier: one ordinary lookup to find the formal declaration of the name application, and an additional to find the actual declaration. To handle this, the definition of `binding` in `UseDoor` would need to be modified. Although we have not tried to specify this, we see no reason why it should not be possible. It is also an area of future study to actually verify this.

11.7 Summary

In this chapter, the basic example door package of §8.9 has been extended to handle the major problems in static-semantics appearing for object-oriented languages. The resulting door package handles name analysis according to the visibility graph based method of §3.3. Arbitrary combination of block structure and subclassing is supported. Cyclic class hierarchies in the user program are detected and resolved by breaking the cycle at one of the classes. Reference variables, remote access, and type checking of reference assignment is supported. The door package makes use of history-dependent error detection for errors like multiple declaration of identifiers and cyclic subclassing. A door class was introduced for communicating error information to external components. All the examples have been implemented and tested in practice.

Chapter 12
Evaluation

This chapter reports on practical experience from using Door AGs in the Orm environment. We discuss actual time consumption and estimates of space consumption.

12.1 Introduction

The incremental static-semantic analyzer in the Orm programming environment is based on a precursory form of Door AGs. The kernel of semantic primitives used in Orm corresponds to a door package, and the grammar used for specifying static-semantics corresponds to a main grammar. The example door package developed in Chapters 8 to 11 is essentially a reformulation of the most important parts of the Orm kernel as a formal door package. In addition, the example door package has been implemented exactly according to the techniques presented in this thesis. This implementation was done in a separate testbed, and is not yet integrated into the Orm system.

12.2 Main grammar for Simula

The main grammar for Simula used in Orm contains around 100 node classes of which around half are related to declarations, half to expressions, and a few to statements. Examples of supported declarations include classes, procedures, parameters to classes and procedures, arrays, simple types like boolean, integer, real, and character. Examples of supported statements include value assignment, reference assignment, if-statement, while-statement, for-statement, procedure call. Examples of supported expressions include remote access, arithmetic expressions (+, -, ...), boolean expressions (and, or, ...), relations (=, <, >, ...), and reference expressions (in, is, qua, this, new...), in all around 40 expressions. The major part of Simula is covered, but a few constructs have been left out such as goto, labels, and switches, block statements, name parameters, and explicit attribute protection (hidden/protected). The resulting main grammar for Simula has 1-visit attribute dependencies although there are no restrictions on the order in which the declarations may occur.

12.3 Modules in Orm

Orm Simula differs from standard Simula by supporting a module concept which is slightly more general than the "external" construct in standard Simula. The module concept in Orm makes it possible to define fragments of programs as separately stored (and compiled) units. A module is in many ways similar to a statically allocated object. It may contain classes, procedures, local variables, and code. Importing a module is semantically similar to inspecting an object: all the declarations in the module become visible in the block which contains the import.

Modules can be understood in terms of Door AGs by considering connecting the syntax trees of two modules via two doors as shown in Figure 12.1. However, the module concept used in Orm is currently hand-coded, and a formalization of this use of doors remains an area of future research. Bindings across module boundaries are not handled in the same way as bindings within a module. For example, bindings which occur via an import-export door do not leave traces in the exporting module. Instead, the import door maintains information about the bindings in order to facilitate updates when the imported module is changed. The exporting module does not contain any information about its importing modules.

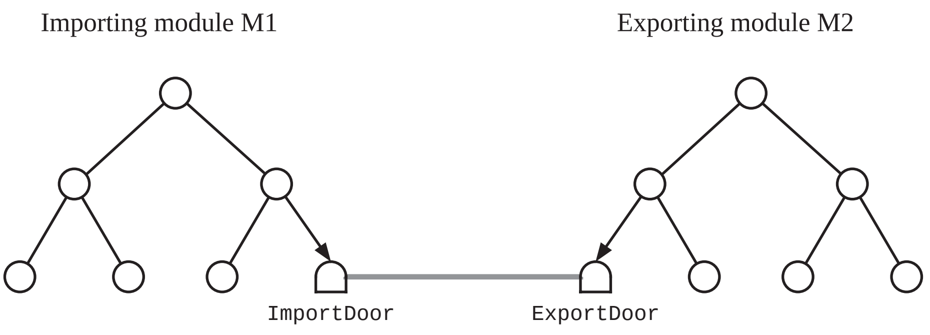


Figure 12.1 Connecting modules by doors

Interactive and incremental support for changes across module boundaries is currently rather limited in Orm. An edited module is always linked to stored versions of its imported modules. I.e., there is no support for editing two connected modules where changes in one module affects the other module directly. Suppose a module M1 imports another module M2. If a new version of M2 is stored, this does not affect M1 which will continue to use the old version of M2. To switch to the new version, the user has to give an explicit command to the M1 module to update itself to import the newest version of M2. In the current implementation of Orm, this is implemented internally by removing the import of the old version followed by inserting an import to the new version.

There are several ways in which the support for modules could be improved. One important issue is to allow connected editing of two or more modules. I.e., changes to one of these modules should immediately cause corresponding updates in the other modules. In particular, this would be useful when the same person works simultaneously with a number of related modules. One could also consider connecting modules edited by different users as it is done in the Mercury system [KKM87].

Another interesting issue is to limit the re-evaluation at module updates. The current technique of removing and inserting an import in order to switch to the newest version of an imported module could be considerably improved. An improved strategy would be to identify which entities in M2 were actually changed, and re-evaluate only dependents on these entities. This would be similar in nature to the idea of “smart recompilation” suggested by Tichy [Tic86].

12.4 Time consumption

For most edit operations, the incremental static-semantic analysis in Orm does not cause noticeable delays. This is in spite of several factors in the current implementation of Orm which could be substantially speeded up in a system intended for production use. In particular, the attribute expressions in the main grammar are interpreted rather than compiled.

For changes to statements, there are no non-local dependents, and the time for incremental re-evaluation is directly proportional to the size of the change. Adding or deleting a statement does not cause any noticeable delay.

For changes to declarations, the time for re-evaluation depends on the number of affected use sites. Changes to declarations thus have the potential of causing arbitrary long delay times since there may be an arbitrary number of use sites affected by the change. Nevertheless, most changes to declarations affect only a small number of use sites, and in these cases, the time for re-evaluation is not noticeable. In particular, a very common operation is to add a new declaration which is not yet used anywhere. Although the declaration may be visible throughout the program, there are no actual non-local dependents, and the time for re-evaluation is practically zero.

12.5 Space consumption

Space consumption in Orm is currently unnecessarily high because the implementation efforts have been directed towards adding interesting functionality rather than optimizing the system. Simple changes could give space savings of several factors. For this reason, we have not made any measurements on the space consumption of Orm. Instead, we have calculated estimated space costs for the example door package of Chapters 8 to 11. These costs are based on some assumptions on the structure and static-semantic characteristics of average programs for which we only have preliminary estimates. The resulting figures should therefore be taken only as a rough indication. Nevertheless, we have tried to make realistic assumptions which are more on the pessimistic than the optimistic side.

12.5.1 Static-semantic characteristics

The space cost for the semantic attribution of a given program depends on the syntactic structure of the program, on a few static-semantic characteristics, and on the basic costs for objects and references etc.

The dominant factor is the number of name applications and their distance from the corresponding declaration. When binding a name application, a number of blocks are passed during lookup (passed without finding a match). We split this number into two quantities: `SBLK` and `PBLK`, where `SBLK` is the number of blocks passed in the "static direction", i.e. upwards along the nesting hierarchy, and `PBLK` is the number of blocks passed in the "prefix direction", i.e. upwards along the class hierarchy. The total number of blocks passed during lookup (without finding a match) is `PBLK+SBLK`. For reference assignments and other constructs which require comparison of formal qualifications, we use the quantity `PDIST` for the number of prefix class edges between the two compared qualifications.

Measurements on a large body of actual programs should be made in order to give accurate estimates of the average values of these quantities. In the lack of such measurements we instead make the following assumptions about the name applications:

- 50% are accesses to local variables and parameters (`SBLK=0`, `PBLK=0`).
- 20% are accesses to instance variables from inside a procedure (`SBLK=1`). We estimate `PBLK=1`.
- 20% are remote accesses to variables and procedures (`SBLK=0`). We estimate `PBLK=1`.
- 10% are references to other entities. (Typically class names occurring in declarations of reference variables.) We estimate `SBLK=3`, `PBLK=6`, corresponding to an estimated average depth of 4 in the nesting hierarchy and 7 in the class hierarchy.

Furthermore, we estimate the average of `PDIST` to be `1`. We do not think any of these estimates is optimistic. These estimates give averages according to the table below.

		estimated average
`SBLK`	`# of blocks passed in "static" direction`	0.5
`PBLK`	`# of blocks passed in "prefix" direction`	1
`PDIST`	`# of prefix class edges between compared classes`	1

12.5.2 Cost for objects and references

The space cost also depends on the implementation of objects and dynamic and static references attributes. We assume the door package is implemented in an object-oriented language with garbage collection. A reasonable implementation of such a language could use 12 bytes in object overhead (`OBJ`): one pointer to the class template, one pointer for the static link, and one word for garbage collection information. The static link is needed if the object or its inner blocks (e.g. virtual procedure implementations) access global information outside the object. If no such information is accessed (or if some other access method is used), the object overhead could be reduced by removing the static link. Depending on the garbage collection algorithm, it might be possible to also remove the word used for garbage collection information.

Dynamic (`DYN`) and static (`STAT`) reference attributes are typically implemented by pointers (4 bytes). It would be possible to reduce also this cost by inlining part-objects (giving `STAT=0`). The table below shows the costs assumed in the following discussion.

		assumed cost in bytes
`OBJ`	`overhead per object`	12
`DYN`	`dynamic reference attribute`	4
`STAT`	`static reference (to part-object)`	4

12.5.3 Cost for doors

To calculate the space costs for the doors we must also estimate the space for collection memberships as stated by the conditions in the Door AG. There are two kinds of collection objects used: `UnorderedCollection` (used for most collections) and `Dictionary` (used for collecting the `UseDoor` objects which have attempted to bind to a `SymbolTable`). We will use the quantities `TRACE` and `KEYTRACE` to model memberships in these collections.

A reasonably space-efficient implementation of the `UnorderedCollection` would be to use a linked list of small arrays of, say, 4 elements in each. The cost for a `TRACE` membership would then be `(OBJ+DYN+4DYN)/4`, i.e. object overhead + link reference + the 4 membership references, and splitting the total on 4 memberships. The same technique could be used for `KEYTRACE`, but storing also a string reference for each member, giving a cost of `(OBJ+DYN+4DYN+4DYN)/4`.

The `UnorderedCollection` object would have an overhead of `OBJ+DYN+2TRACE` (the latter is the cost for an average half empty array), and, similarly, the `Dictionary` object an overhead of `OBJ+DYN+2KEYTRACE`. The table below summarizes these costs.

			bytes
`TRACE`	`membership in UnorderedCollection`	`(OBJ+5DYN)/4`	8
`KEYTRACE`	`membership in Dictionary`	`(OBJ+9DYN)/4`	12
`UNCOLL`	`overhead for UnorderedCollection`	`OBJ+DYN+2TRACE`	32
`DICT`	`overhead for Dictionary`	`OBJ+DYN+2KEYTRACE`	40

Given these quantities, it is simple to calculate the space costs for the doors as follows:

```
overhead for door object                  1 OBJ
static reference from owner node to door  1 STAT
dynamic reference from door to owner      1 DYN
local attributes in door                  x DYN
conditions in door                        y TRACE + z KEYTRACE
for each part object:
   static reference to part object        1 STAT
   object overhead for part object        1 OBJ
   local attrs of part object             ...
   part objects of part object            ...
```

The following table shows the resulting costs and the estimates in bytes.

	Space cost	bytes
RootDoor	OBJ + STAT + DYN	20
BlockDoor	5OBJ + 6STAT + 6DYN + DICT	148
ClassDoor	8OBJ + 11STAT + 12DYN + DICT + 2UNCOLL	292
DeclDoor	2OBJ + 3STAT + 7DYN + UNCOLL	96
UseDoor	OBJ + STAT + 4DYN + (SBLK+PBLK)*KEYTRACE + PBLK*TRACE + TRACE	66
CompareClassDoor	OBJ + STAT + 3DYN + PDIST*TRACE	40

12.5.4 Cost for syntax trees

To estimate the space cost for the syntax tree of an average program, we have made some measurements on Simula programs. The measurements were made on a total of approximately 15000 lines of Simula code. In order to compare the textual representation with a reasonable syntax tree representation, the following quantities were measured: textfile size (in lines and bytes), number of tokens, number of identifiers, average length of identifiers, and average number of occurrences of the same identifier. The table below summarizes these measurements.

	Average over 15000 lines of Simula code
Size of textfile in bytes per line	30 bytes / line
Number of tokens per line	4 tokens / line
% of tokens which were identifiers	35%
Average length of identifier	10 characters
Average number of occurrences of the same identifier	10

It is reasonable to assume that the number of syntax nodes in an abstract syntax tree is the same, or slightly larger than the number of tokens in the text representation. We have made some preliminary measurements which confirm this. We have estimated this factor to 1.35 to take into account that name applications are represented by two nodes: a `Use` and an `ID` node, according to the main grammar of §8.9.4, whereas they correspond to only a single token.

The cost for the average token is then

```
OBJ+2DYN+0.35(OBJ+2DYN+DYN+(OBJ+10)/10) = 29 bytes
```

This corresponds to a syntax node object + father and son references + an average 35% of an `ID` son node which has a reference to a text string object of 10 characters, shared by 10 other `ID` nodes.

The space cost for an abstract syntax tree is thus about 4 times as high as the space cost for the corresponding textual representation. This is in approximate agreement with other reported figures. For example, the Rational environment for Ada is reported to have a corresponding factor of 4.5 and a cost of 20 bytes per syntax node [WL86].

12.5.5 Cost for semantic attribution

To compute the actual space cost for the semantic attribution, we have measured the number of tokens corresponding to the different door classes. The weights in the table below are the measured proportions of these tokens. These measurements were made on the 15000 lines of Simula code used in the measurements of the previous section. The average space cost per door for one token is computed by multiplying this weight with the cost for one door:

Door	Corresponding token	Weight	bytes per door	bytes per token
`UseDoor`	`Name application`	30%	66	19.8
`DeclDoor`	`Name declaration`	5%	96	4.8
`ClassDoor`	`Class`	0,3%	292	0.88
`BlockDoor`	`Procedure`	1,5%	148	2.22
`CompareClassDoor`	`Reference assignment`	3%	40	1.20
				SUM=29

The total average cost for the semantic attribution amounts to 29 bytes per token. This is, incidentally, the same figure as for the syntactic representation. I.e., the semantic attribution takes up the same amount of space as the syntax tree. For comparison, the Rational environment is reported to use around 25% less space for the semantic attributes than for the syntax tree, which would correspond to 15 bytes per syntax node for their system.

12.6 Comparison to other work

We have reviewed other techniques for incremental semantic analysis in §3.7 and §4.6. To our knowledge, none of these methods has been applied successfully to object-oriented languages. The most advanced languages which have been specified and implemented using other methods are modular languages like Modula-2 [Vor90a], [BGV92]. The subclassing feature of object-oriented languages introduces a recursive element in name analysis which does not follow the syntax tree. Many approaches have specific support for nested scopes which follow the syntax tree, e.g. [JF82], [BC85], [Hoo87], but fail to handle more complex combinations. The naming specification language NSL of Vorthmann [Vor90a] appears to be able to describe scopes for object-oriented languages, but has not been applied to such languages. Furthermore, NSL lacks facilities for handling erroneous cyclic subclassing and type checking of reference assignments.

The ability to specify objects and references declaratively seems to be unique for Door AGs. Most other systems adopt a value-oriented specification language, although objects and references are often used internally for implementing higher-level constructs and for speeding up evaluation. The constructs in Door AGs bear similarities to constructs of other attribute-grammar based methods. E.g. the collections and conditions in Door AGs are similar to the set-valued attributes and membership constructs of [Kai85]. The door classes have similarities to the maintained and constructor attributes of [BC85]. However, the explicit use of objects and references in Door AGs make the technique more general and allows description of more advanced attributions.

The use of history-dependent error checking also seems to be unique for Door AGs. Other systems intended for static-semantic checking require the grammar to specify exactly one attribution for each possible syntax tree and cannot handle such history-dependencies.

Circular dependencies via non-local dependencies are easily handled in Door AGs, simply by inserting a convergence test in one of the door visit procedures. We find it important and even essential to be able to handle such dependencies. If circular dependencies are not allowed some problems will be very difficult to define, leading to more complex attributions, less suited for incremental update. However, few other methods allow circular dependencies. Those which do, e.g. [Far86] and [Jon90], are not directed towards solving advanced scope handling.

Space consumption seems to be a neglected area in the field of attribute grammars and few papers report actual figures on this. Attribute grammars have a reputation of being very space-intensive. For example, Kiong and Welsh [KW92] developed a hand-coded incremental semantic analyzer for Pascal which is reported to use 8 times less storage for the semantic attribution than does the Pascal editor supplied with the Cornell Synthesizer Generator version 1.0.1. When developing the Door AG technique it was an important goal to allow space-efficient attributions to be defined. In the previous section we gave estimates on space consumption which indicate that the space consumption is in fact low, approaching that of hand-coded systems.

12.7 Summary

Our experience from implementing Door AGs shows that the technique is useful in practice for constructing highly interactive program development environments. The resulting response time after program modifications is very low in Orm, and usually not noticeable by the user. This is in spite of the fact that the system interprets the attribute expressions in the main grammar. This confirms the view we took in §2.4, where we stated that by using appropriate incremental techniques, the amount of data which needs to be recomputed is small after each change, and one does not have to use the fastest most optimized methods to recompute this data.

Space consumption, on the other hand, is important to try to keep low. Our estimate of the space consumption for a Door AG based system gives a rough indication of about the same amount of space for the semantic attributes as for the syntax tree, i.e. a total of 30+30=60 bytes per average attributed syntax node, corresponding to a factor 8 larger than a text representation. These estimates were based on reasonable, probably pessimistic assumptions about syntactic and static-semantic properties of average programs and costs in the underlying implementation language. These figures are on a par with the space consumption of hand-coded commercial incremental systems.

Chapter 13
Conclusions and Future Work

13.1 Contributions

The main contribution of this thesis is a new technique for implementing incremental static-semantic analyzers: *Door Attribute Grammars*. The motivation for developing this technique was to be able to handle object-oriented languages. Requirements on the technique included that it should allow fine-grained incremental updating in order to keep the static-semantic information up to date after each single edit operation performed by the user, it should allow space efficient representation of the static-semantic information, and the technique should scale up in order to handle large programs.

Door Attribute Grammars extend standard attribute grammars by allowing objects and references to be specified as part of the attribution of a syntax tree. This allows the comparatively complex static-semantics of object-oriented languages to be described in a straight-forward manner, including problems like name analysis in the presence of subclassing, remote access, and type checking of reference assignments. The resulting attributions are space efficient and suited for incremental updates. In particular, the best methods for incremental name analysis can be used, resulting in response times proportional to the number of affected use sites after a change to a declaration. In practice, the number of affected use sites is small and the response time is un-noticeable by the user even when changing a global declaration in a large program.

We have built a complete incrementally compiling environment: *Mjølner/Orm*, which currently supports the major part of Simula. This system is based on a precursory form of Door AGs. The Door AGs as presented in this thesis have also been tested in practice for a number of key problems. This practical experience shows that the technique fulfills the requirements and can be used in practice for constructing highly interactive program development environments.

Although the motivation for developing Door AGs was to handle object-oriented languages, the technique is general and can be applied to any language based on a context-free grammar.

The introduction of objects and references in the attribution is a radical step away from standard AGs which are based solely on value semantics. Nevertheless, the Door AG formalism preserves the principle idea of standard AGs, namely a declarative description which states the invariant properties of a correct attribution.

The rest of this section summarizes the most interesting aspects on Door AGs.

Elements in Door AGs

A Door AG is an extension of a standard AG. The extensions can be summarized as follows:

- A syntax node can be extended with part-objects. A part-object owned directly by a syntax node is called a *door*, and transitively owned objects are called *semantic objects*.
- A semantic object can be specified as a *collection*, meaning that it is a collection of member objects, and the members are defined non-locally by *conditions*.
- Attributes may be *references*. I.e. they may have object identity values, denoting other nodes, doors, or semantic objects.

The attributes of doors and semantic objects are defined in the same way as the attributes of syntax nodes in standard AGs: by means of equations. One of the most important advantages of Door AGs compared to standard AGs is the fine granularity of definition which can be obtained. It is possible to let the equations and conditions define a very small amount of information each. This is in contrast to standard AGs where one is forced to let some equations define very large information structures. The finer granularity in Door AGs allows the size of AFFECTED (the set of affected attributes after a syntactic change) to be dramatically decreased for some important problems and therefore makes it possible to implement much more efficient incremental evaluators.

Applications of Door AGs

The use of objects and references for attributing syntax trees makes it is possible to implement name analysis in a straight-forward way, using explicit visibility graphs, symbol tables, and references between identifier declaration and application sites. This gives compact and simple attributions which are suitable for incremental updates and for access from external tools.

To exemplify the applicability of Door AGs and to show that the suggested implementation technique works, an advanced example door package was described in Chapter 11. The door package handles the major static-semantic problems occurring in object-oriented languages: It supports arbitrary combination of block structure and subclassing, including nested classes. Cyclic class hierarchies in the user program are detected and resolved. The door package also supports reference variables, remote access, and type checking of reference assignments.

Main grammars and door packages

The price for allowing objects and references in the attribution is that non-local dependencies are introduced. In general, this prevents attribute evaluators to be automatically generated from the grammar. This problem is addressed in Door AGs by splitting the grammar into a main grammar and a door package. The main grammar is essentially equivalent to a standard attribute grammar and can be evaluated exhaustively and incrementally using standard methods. The door package isolates the non-local dependencies and must be implemented by hand. However, a systematic technique has been developed for constructing evaluators for door packages. This technique involves doing a static dependency analysis of the door package and constructing a dependency graph for each door class. Additional dependency attributes and functions are added to the door classes to allow efficient propagation of non-local dependencies at evaluation time. The technique allows systematic construction of visit procedures from the dependency graphs and the resulting attribute evaluator is very efficient since it is based on a static visit-oriented technique.

Door packages as tool box extensions to standard AGs

The specification and implementation of door packages constitutes a systematic way of constructing tool box extensions to standard AGs. Door packages which implement some general aspects of a family of programming languages can be used as a tool box by many different main grammars, in order to implement different languages. This approach is used in the Orm system where the static-semantic grammars for different languages are specified by using a kernel of semantic primitives corresponding to a door package. The current semantic kernel in Orm is designed to cover the basic language constructs in object-oriented languages: classes, procedures, subclassing, reference variables, and remote access. An interesting future challenge is to design door packages which cover general aspects of a broader range of programming languages.

Door classes can also be used as a general interface mechanism, in order to connect an attributed syntax tree to external components and trigger events in these components as a result of changes in the attribution. This was illustrated in §11.5 where a door class `ErrorDoor` was implemented to monitor static semantic errors. The visit procedures of the door class were implemented to achieve suitable side-effects in the window system (displaying and removing error messages) as a result of changes in the attribution.

Simplicity of main grammars

Problems like name analysis which give rise to complex attribute dependencies in standard AGs are handled by objects and references in Door AGs. This has the effect that the remaining attribute dependencies in the main grammar are very simple. For example, languages like Algol and Simula, which allow an arbitrary order of declaration, can be described by Door AGs with 1-visit main grammars. Describing these languages in standard AGs would require an Ordered AG.

The simple 1-visit dependencies in main grammars makes it possible to use very simple and efficient evaluation techniques. We have developed a visit-oriented evaluation technique for 1-visit grammars based on static skipping. This technique does not compare attribute values, but skips evaluation instructions and visits on the basis of a static approximation of the dependencies. While such an algorithm is in principle sub-optimal, it works well in practice for the main grammar of a Door AG. We find it interesting that it is possible to use such simple implementation techniques and yet achieve an efficient incremental system for a complex language like Simula.

Interpretation of grammars

The Orm system *interprets* visit sequences for main grammars rather than running compiled visit procedures. This allows the grammar to be changed easily and tried out on programs without having to recompile and link the Orm system. In this way, Orm supports interactive development of language-based environments. Although the visit sequences are interpreted, the incremental evaluation is sufficiently efficient for practical use because the “inner loops” of the incremental processing are performed in the door package which is a compiled part of Orm.

Circular dependencies

There are several situations in static semantic checking which intuitively lead to circular chains of dependencies. In particular: arbitrary declaration order, cyclic subclassing, and declaration of reference variables. All these problems are straight-forward to specify in Door AGs, and one of them (reference variables) actually leads to a circular chain of attribute dependencies via a non-local dependency. Such circularities are easily handled in Door AGs simply by inserting a convergence test. In the case of reference variables, the circular evaluation converges immediately, after a single evaluation cycle.

Circularities via non-local dependencies do not affect the complexity of the main grammars. Thus, simple 1-visit evaluation techniques can be used for the main grammar even if the Door AG as a whole is circular.

History-dependent error checking

We have advocated the use of *history-dependent error checking*. I.e., if a static-semantic error has several possible causes, the latest edited part of the syntax tree is regarded as causing the error. Typical examples when this kind of error handling is useful is cyclic subclassing and multiple declarations of the same name. The use of history-dependent error checking for these problems has the consequence that mistakes made at one point in the program will not cause previously correct parts of the program to suddenly be considered erroneous. For example, adding a new declaration of an already existing name causes the new declaration to be considered erroneous, whereas the old one remains in effect. We find this a highly desirable behavior of an interactive system. In addition to associating the errors with the latest edited parts of the program, this technique leads to less re-evaluation than if all possible causes of the error should be considered as actually erroneous.

Although the error may be associated with one part of the program, it may very well be corrected by editing another part of the program. One could even implement interactive support for finding other potential causes of an error, given a current erroneous site.

History-dependent error checking is accomplished in Door AGs by using *underdetermined* grammars. I.e. grammars for which some aspect of the attribution is not uniquely defined. This means that there are syntax trees for which there is more than one valid attribution. Which solution is actually chosen will depend on the order of evaluation and can, if desired, be controlled by inserting additional actions in the visit procedures of the door classes.

Object-oriented attribute grammars

The main grammars of Door AGs are based on an object-oriented variant of standard AGs presented in Chapter 6. In this object-oriented formulation of standard AGs, the syntax nodes are viewed as objects of classes organized in a specialization hierarchy. Behavior (in the form of attributes and equations) can be defined at suitable levels of generalization and default equations can be overridden in specialized node classes. This allows the grammar to be written in a more compact and readable way than is possible in traditional AG formalisms.

Another advantage of object-oriented AGs is that demand attributes can be easily implemented since they are essentially equivalent to virtual functions. We have found demand attributes to be very useful in interactive environments since they do not occupy any space. In fact, all attributes of the main grammar of a Door AG are by default implemented as demand attributes since most of these attributes are defined by copy equations and very simple to compute when needed.

13.2 Future work

Our experience so far with Door AGs shows that important problems in incremental static-semantic analysis can be solved in a satisfactory manner by this technique. Nevertheless, more practical work should be done on developing door packages and testing them in practice.

Another interesting possibility is to work further in the direction of interactive support for language development and do incremental static-semantic checking also during editing of the grammars themselves. The grammar formalisms used in Orm are based on object-oriented concepts, similar to OOSL, and the static-semantic checking should not be very different from checking object-oriented programming languages.

In addition to applying the Door AG technique to various problems, there are several ways in which the technique itself could be further developed. Below we discuss some of these possibilities.

Cutting and pasting large subtrees

The incremental attribute evaluation algorithm presented in this thesis assumes that a new subtree is always completely un-evaluated before insertion and the old subtree is completely de-evaluated before removing it from the syntax tree. This may be undesirable for large subtrees such as whole classes and procedures since the evaluation/de-evaluation of such large subtrees may lead to noticeable response times. Actually, the Orm system has a block clipboard facility which supports moving and copying fully attributed blocks between the program and the clipboard. This functionality has, however, not yet been formalized in the Door AG model. To do this, the notion of fix attributes needs to be refined. Moving a block to a new context implies that the fix inherited attributes may get new values. The "fix" attributes are thus fix only during incremental evaluation, but may change values at a subtree replacement. The attribute evaluation algorithms for Door AGs need to be generalized to support this.

Dynamic objects

In Door AGs, as presented in this thesis, the semantic objects are always part objects owned transitively by a syntax node. I.e., the existence of all semantic objects is determined from the syntax tree alone. For some advanced language constructs, this may be insufficient. It may be necessary to introduce new semantic objects during evaluation, and make the existence of some semantic objects depend on the attribute values. For example, as discussed in §3.3.5 and §11.6.2, extending a virtual class in BETA leads to the implicit introduction of actual definitions of subclasses to the virtual class. In our visibility graph model for name analysis this means that the existence of some `SearchPath` objects will depend on identifier bindings. To express this in Door AGs, some notion of dynamic semantic objects would need to be introduced.

Another aspect on the static scheme for attaching objects in Door AGs is that it may result in attaching objects which are not always needed. For example, consider merging reference assignment and value assignment to a single syntactic construct (as several languages do). In the present Door AGs, a `CompareClassDoor` object has to be declared for the assignment node class, to take care of comparisons for reference types. But this door object would be unnecessary if the types of the assignment subcomponents were actually value types, rather than reference types. It would be useful to introduce some dynamic scheme for attaching the door object only if it was actually needed, i.e. attaching it depending on the attribute values.

An interesting solution to this problem could be to add support for semantically controlled replacement of *syntax nodes*. E.g., one could imagine a general node class `Assignment` which was specialized into three subclasses: `SyntacticAssignment`, `ValueAssignment`, and `RefAssignment`. The structure-oriented editor (or parser) would always construct a syntactic assignment, but depending on attribute values, the node would be replaced in the process of semantic evaluation by a value assignment node or a reference assignment node. Besides saving storage, such a scheme has the potential of leading to simplifications of the attribute grammar in case the same syntactic (context-free) construct has many possible semantic interpretations.

General door packages

An interesting area of future research is to investigate if it is possible to construct door packages which are applicable to a broad range of languages. Rather than having a door for a block statement and a door for a class, as in our example door package, such a door package should contain more primitive doors which could be combined to model block statements, single-inheritance classes, multiple-inheritance classes, etc. The work of Vorthmann is in this direction [VL88], [Vor90a]. His language NSL supports specification of advanced name analysis, although it is not sufficiently general to handle object-oriented languages. It could be interesting to try to merge the ideas in NSL with Door AGs.

Using the syntax tree itself as a symbol table

In the example door package, special semantic objects are introduced for representing symbol tables and declaration entries. In principle, this information is available in the syntax tree as well. The symbol table is essentially a copy of a declaration list in the syntax tree. Space reductions would be possible by avoiding this copying. It would be valuable to develop general mechanisms which allowed the doors to utilize syntax nodes as semantic objects instead of specifying their own part objects. If possible, this should be done without making the door package dependent on the structure of the main grammar, in order to be able to use the same door package for many languages.

Chapter 14
Bibliography

[ACR+87] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of attributed graphs. Technical Report CS-87-29, Brown University, Dept. of Computer Science, Providence, R.I., December 1987.

[ACR+88] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Graph attribution as a specification paradigm. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 121–129, Boston, Ma., November 1988. ACM. SIGPLAN Notices 24(2).

[Bal89] R. A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, Computer Science Division – EECS, Univ. of California, Berkeley, 1989. TR UCB/CSD 89/548.

[BC85] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 34–42, Seattle, Wa., 1985. ACM. SIGPLAN Notices, 20(7).

[Ben90] M. Bengtsson. *Real-Time Compacting Garbage Collection Algorithms*. Licentiate thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 1990.

[Bes87] G. M. Beshers. *Regular Right Part Grammars and Maintained and Constructor Attributes in Language Based Editors*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Il., 1987.

[BGV92] R. A. Ballance, S. L. Graham, and M. L. Van de Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[BS86] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[CI84] R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, January 1984.

[CNS87] M.-L. Christ-Neumann and H.-W. Schmidt. ASDL - an object-oriented specification language for syntax-directed environments. In H.K. Nichols and D. Simplson, editors, *ESEC'87 Proceedings*, volume 289 of *Lecture Notes in Computer Science*, pages 71–79. Springer-Verlag, 1987.

[Coo88] S. Cook. Impressions of ECOOP'88. *Journal of Object-Oriented Programming*, 1(4):42–43, 1988.

[Coo89] W. R. Cook. A proposal for making Eiffel type-safe. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 57–70, Nottingham, U.K., July 1989. Cambridge University Press.

[DEFH87] S. A. Dart, R. J. Ellison, P. H. Feiler, and A. N. Habermann. Software development environments. *IEEE Computer*, pages 18–28, November 1987.

[DHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. In D. B. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, 1984.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars. Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[DLMM87] H. P. Dahle, M. Löfgren, O. L. Madsen, and B. Magnusson. The Mjølner project. In *Software Tools: Improving Applications: Proceedings of the conference held at Software Tools 87*, pages 81–87, London, June 1987. Online Publications.

[DMN68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. NCC Publ. S-2, Norwegian Computing Centre, Oslo, May 1968. Revised 1970 (Publ. S-22), 1972, and 1984. Swedish Standard SS 63 61 14, 1987.

[DMS84] N. M. Delisle, D. E. Menicosy, and M. D. Schwartz. Viewing a programming environment as a single tool. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, pages 49–56, Montreal, Canada, 1984. ACM. SIGPLAN Notices, 19(6).

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116. ACM, January 1981.

[DRZ85] A. Demers, A. Rogers, and F. K. Zadeck. Attribute propagation by message passing. In *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 43–59, Seattle, Wa., 1985. ACM.

SIGPLAN Notices, 20(7).

[Eng84] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–137. Cambridge University Press, 1984.

[ENR83] H. Ehrig, M. Nagl, and G. Rozenberg, editors. *Graph-Grammars and their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983.

[Far86] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, Palo Alto, Ca., July 1986. ACM. SIGPLAN Notices, 21(7).

[FKT90] A. Feng, T. Kikuno, and K. Torii. Incremental attribute evaluation for multiple subtree replacements in structure-oriented environments. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 192–206, Paris, September 1990. Springer-Verlag.

[Fri83] P. Fritzson. Symbolic debugging through incremental compilation in an integrated environment. *The Journal of Systems and Software*, (3):285–294, 1983.

[Fri84] P. Fritzson. *Towards a Distributed Programming Environment based on Incremental Compilation*. PhD thesis, Linköping University, Linköping, Sweden, 1984.

[Gar87] P. E. Garrison. *Modeling and Implementation of Visibility in Programming Languages*. PhD thesis, University of California, Berkeley, Ca., December 1987. Tech. Rep. UCB/CSC 88/400.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80. The Language and its Implementation*. Addison-Wesley, 1983.

[Gro90] J. Grosch. Object-oriented attribute grammars. In A. E. Harmanci and E. Gelenbe, editors, *Proceedings of the 5th International Symposium on Computer and Information Sciences (ISCIS V)*, pages 807–816, Cappadocia, Nevsehir, Turkey, August 1990.

[Gus90] A. Gustavsson. *Software Configuration Management in an Integrated Environment*. Licentiate thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 1990.

[Hed88] G. Hedin. Incremental attribute evaluation with side-effects. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation (2nd CCHSC Workshop)*, volume 371 of *Lecture Notes in Computer Science*, pages 175–189, Berlin, GDR, October 1988. Springer-Verlag.

[Hed89] G. Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 329–345, Nottingham, U.K., July 1989. Cambridge University Press.

[Hed91] G. Hedin. Incremental static semantic analysis for object-oriented languages using Door attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 374–379, Prague, June 1991. Springer-Verlag.

[Hed92] G. Hedin. Context-sensitive editing in Orm. In K. Systä et al., editors, *Proceedings of the Nordic Workshop on Programming Environment Research*, Tampere, Finland, January 1992. Tampere University of Technology. Software Syst. Lab. TR 14.

[HM86] G. Hedin and B. Magnusson. Incremental execution in a programming environment based on compilation. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, pages 480–491, Honolulu, Hi., January 1986.

[HM87] G. Hedin and B. Magnusson. Supporting exploratory programming in Simula. In *Proceedings of the 15th Simula User's Conference*, pages 73–88, St. Helliers, Jersey, September 1987. Association of Simula Users.

[HM88] G. Hedin and B. Magnusson. The Mjølner environment: Direct interaction with abstractions. In S. Gjessing and K. Nygaard, editors, *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 41–54, Oslo, August 1988. Springer-Verlag.

[Hoo86] R. Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, St. Petersburg, Fl., January 1986. ACM.

[Hoo87] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, Ithaca, N.Y., May 1987. Tech. Rep. 87-836.

[HT86a] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, 1986.

[HT86b] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50, Palo Alto, Ca., July 1986. ACM. SIGPLAN Notices, 21(7).

[Hud91] S. E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, 1991.

[Jal85] F. Jalili. A general incremental evaluator for attribute grammars. *Science of Computer Programming*, (5):83–96, 1985.

[JF82] G. F. Johnson and C. N. Fischer. Non-syntactic attribute flow in language based editors. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 185–195, Albuquerque, N.M., January 1982. ACM.

[JF85] G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 141–151, New Orleans, La., January 1985. ACM.

[Jon90] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, 12(3):429–462, 1990.

[Jou84] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, 1984.

[JS86] L. G. Jones and J. Simon. Hierarchical VLSI design systems based on attribute grammars. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 58–69, St. Petersburg, Fl., January 1986. ACM.

[Kai85] G. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., May 1985. CMU-CS-85-131.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[KG89] S. M. Kaplan and S. K. Goering. Priority controlled incremental attribute evaluation in attributed graph grammars. In J. Diaz and F. Orejas, editors, *TAPSOFT '89 Vol. 1*, volume 351 of *Lecture Notes in Computer Science*, pages 306–320, Barcelona, Spain, March 1989. Springer-Verlag.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[KKM87] G. Kaiser, S. M. Kaplan, and J. Micallef. Multiuser, distributed language-based environments. *IEEE Software*, pages 58–67, November 1987.

[KLMM83] G. Kahn, B. Lang, B. Mélese, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[KM88] J. L. Knudsen and O. L. Madsen. Teaching object-oriented programming is more than teaching object-oriented programming languages. In S. Gjessing and K. Nygaard, editors, *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 21–40, Oslo, August 1988. Springer-Verlag.

[KMMN87] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, 1987.

[KMMN91] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Preprint of book, Aarhus University, Denmark, 1991.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[Kos88] K. Koskimies. Software engineering aspects in language implementation. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation (2nd CCHSC Workshop)*, volume 371 of *Lecture Notes in Computer Science*, pages 39–51, Berlin, GDR, October 1988. Springer-Verlag.

[Kos91] K. Koskimies. Object-orientation in attribute grammars. In H. Alblas and B. Melichar, editors, *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 297–329, Prague, June 1991. Springer-Verlag.

[KS89] B. Krämer and H.-W. Schmidt. Developing integrated environments with ASDL. *IEEE Software*, pages 98–107, January 1989.

[KW92] D. Kiong and J. Welsh. Incremental semantic evaluation in language-based editors. *Software–Practice and Experience*, 22(2):111–135, February 1992.

[LMOW88] P. Lipps, U. Möncke, M. Olk, and R. Wilhelm. Attribute (re)evaluation in OPTRAN. *Acta Informatica*, 26:213–239, 1988.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[Mad87] O. L. Madsen. Block structure and object-oriented languages. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[Med82] R. Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., March 1982.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MF81] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Trans. on Software Eng.*, 7(5):472–482, September 1981.

[Min85] S. Minör. Structures for incremental semantic analysis in the III programming environment. Technical Report LU-CS-TR:85-15, Lund University, Dept. of Computer Science, Lund, Sweden, November 1985.

[Min90] S. Minör. *On Structure-Oriented Editing*. PhD thesis, Lund University, Lund, Sweden, 1990.

[MHM+90] B. Magnusson, G. Hedin, S. Minör, et al. An overview of the Mjølner/Orm environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pages 635–646, Paris, June 1990. Angkor.

[MM85] B. Magnusson and S. Minör. III an integrated interactive incremental programming environment based on compilation. In *1985 ACM SIGSMALL Symposium on Small Systems*, pages 235–244, Danvers, Ma., May 1985. ACM.

[MM88] O. L. Madsen and B. Møller-Pedersen. What object-oriented programming may be — and what it does not have to be. In S. Gjessing and K. Nygaard, editors, *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 1–20, Oslo, August 1988. Springer-Verlag.

[MM89] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In N. Meyrowitz, editor, *OOPSLA'89 Conference Proceedings*, pages 397–406, New Orleans, La., October 1989. ACM. SIGPLAN Notices, 24(10).

[MMM90] O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong typing in object-oriented languages revisited. In N. Meyrowitz, editor, *OOPSLA/ECOOP'90 Conference Proceedings*, pages 140–150, Ottawa, October 1990. ACM. SIGPLAN Notices, 25(10).

[MN88] O. L. Madsen and C. Nørgaard. An object-oriented metaprogramming system. In *Proceedings of the 21th Annual Hawaii International Conference on System Sciences*, pages 406–415, Honolulu, Hi., January 1988.

[Mye84] E. W. Myers. Efficient applicative data types. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, Salt Lake City, Ut., January 1984. ACM.

[Nør87] K. Nørmark. *Transformations and Abstract Presentations in a Language Development Environment*. PhD thesis, Aarhus University, Aarhus, Denmark, January 1987.

[Not85] D. Notkin. The GANDALF project. *The Journal of Systems and Software*, 5(2):91–105, May 1985.

[Nyg86] K. Nygaard. Basic concepts in object oriented programming. *SIGPLAN Notices*, 21(10), October 1986.

[Osc89] D. Oscarsson. HOPE: Hierarchical object-oriented presentation environment. Reference manual, Lund University, Dept. of Computer Science, 1989.

[Pec90a] S. B. Peckham. *Incremental Attribute Evaluation and Multiple Subtree Replacements*. PhD thesis, Cornell University, Ithaca, N.Y., February 1990.

[Pec90b] S. B. Peckham. Globally partitionable attribute grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 327–342, Paris, September 1990. Springer-Verlag.

[PT89] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, Austin, Tx., January 1989. ACM.

[Rei84] S. P. Reiss. Graphical program development with PECAN program development system. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, pages 30–41, Montreal, Canada, 1984. ACM. SIGPLAN Notices, 19(6).

[Rep82] T. W. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 169–176, Albuquerque, N.M., January 1982. ACM.

[Rep84] T. Reps. *Generating Language-Based Environments*. MIT Press, Cambridge, Ma., 1984.

[RMT86] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language-based editors. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, St. Petersburg, Fl., January 1986. ACM.

[RT84] T. Reps and T. Teitelbaum. The synthesizer generator. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, Pittsburgh, Pa., May 1984. ACM. SIGPLAN Notices, 19(5).

[RT87] T. Reps and T. Teitelbaum. Language processing in program editors. *IEEE Computer*, pages 29–40, November 1987.

[RT88] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator. A System for Constructing Language-Based Editors*. Springer-Verlag, 1988.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[SH91] R. S. Sundaresh and P. Hudak. Incremental computation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, Orlando, Fl., January 1991. ACM.

[SIKV82] D. C. Smith, C. Irby, R. Kimball, and B. Verplank. Designing the Star user interface. *BYTE*, 7(4):242–282, April 1982.

[Sne91] G. Snelting. The calculus of context relations. *Acta Informatica*, 28:411–445, 1991.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Szy92] C. A. Szyperski. Import is not inheritance. Why we need both: Modules and Classes. To appear in ECOOP'92, Utrecht, June 1992.

[THM87] M. Taube, G. Hedin, and B. Magnusson. The Mjølner observation tool: An object-oriented view on programs and program execution. In *Proceedings of the 15th Simula User's Conference*, pages 99–108, St. Helliers, Jersey, September 1987. Association of Simula Users.

[Tic86] W. F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[TM81] W. Teitelman and L. Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–34, April 1981.

[TR81] T. Teitelbaum and T. Reps. The Cornell program synthesizer. *Communications of the ACM*, 24(9):563–573, September 1981.

[TTTI88] T. Tenma, H. Tsubotani, M. Tanaka, and T. Ichikawa. A system for generating language-oriented editors. *IEEE Transactions on Software Engineering*, 14(8):1098–1108, August 1988.

[VL88] S. A. Vorthmann and R. J. LeBlanc. A naming specification language for syntax-directed editors. In *Proceedings: IEEE Computer Society 1988 International Conference on Computer Languages*, pages 250–257, Miami, Fl., October 1988. IEEE Computer Society Press.

[Vor90a] S. A. Vorthmann. *Syntax-Directed Editor Support for Incremental Consistency Maintenance*. PhD thesis, Georgia Institute of Technology, Atlanta, Ga., June 1990. TR GIT-ICS-90/03.

[Vor90b] S. A. Vorthmann. Coordinated incremental attribute evaluation on a DR-threaded tree. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 207–221, Paris, September 1990. Springer-Verlag.

[WL86] T. Wilcox and H. Larsen. The interactive and incremental compilation of ADA using Diana. Internal report, Rational, 1986.

[Yeh83] D. Yeh. On incremental evaluation of ordered attribute grammars. *BIT*, 23(3):308–320, 1983.

[YK88] D. Yeh and U. Kastens. Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Notices*, 23(12):45–50, December 1988.

[YS91] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991.

Index