Applications of Door Attribute Grammars

Werner Burghard and Magnus Nilsson

LUNDFD6-NFCS:94-5055

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Applications of Door Attribute Grammars

by

Werner Burghard and **Magnus Nilsson**

Lund University
June 1994

(Examensarbete i Datalogi)

Department of Computer Science
Lund University

CODEN: LUNDFD6/NFCS-5055/1-162/94

Department of Computer Science
Lund University
Box 118
S-221 00 Lund, Sweden

Abstract

As computers get faster and the tasks for computers to solve get more complex, the demands on program development increase. These demands are met by developing new high level languages or by building suitable programming environments around a programming language. The heart of such an environment is the compiler. Semantic analysis has a key function in any compiler, so methods and techniques in this area play an important role in the development of new programming environments. The increased use of high powered workstations makes the research on incremental semantic analysis particularly important.

This work is about specifying and implementing language constructs for incremental systems. The specification method used is Door attribute grammars – an extended attribute grammar suitable for specifying efficient incremental systems. The OOSL language is used for specifications. All suggested constructs are implemented using a metasystem developed in this work – the Embla system. Using these techniques several language constructs where specified and implemented, such as if-, while-, and with-statements; procedure usage; and user defined types.

The Embla metasystem is described in detail, explaining the structure of the system and how this system can be used by different tools, such as editors, debuggers, and pretty-printers.

Also included in this work is a short survey over different specification methods, including context free grammars, attribute grammars, object-oriented attribute grammars, and Door attribute grammars.

Contents

1 Introduction

The area of compilers and computer languages is a fast moving field that always interests computer scientists. The debate on which programming language that is the best is a sometimes heated discussion that has not been solved yet, and probably never will. One cannot even define what such a language would look like. Should it be large and cover many areas, small and flexible, object-oriented or procedural? Object-orientation has for the last years been the most dominant paradigm, and also this work has a strong connection to it.

While computer languages stir up the most talk, tools and programming environments come a close second. Which compiler is superior? What about linkers, debuggers, class browsers, pretty-printers, etc.? A full fettered programming environment is more unusual, Smalltalk has one and so does most lisp-environments. For most languages, however, the traditional development cycle is mostly used, with compilation, linking and execution as separate steps performed by separate tools. In the latter years one can see more and more programming environments reach the market. From this one might conclude that the development of new languages is put on hold while the environment is in focus.

This work concerns incremental compilation. To compile incrementally is to compile only the changes that has occurred since the last compilation. To do this, the compiler maintains information about the program at all times. This information is a valuable asset for the programmer, since he or she can use it to work faster and make fewer errors.

Incremental programming environments allow many features not normally available in programming environments. The incremental compiler maintains a compiled program at all times. When the programmer makes a change to the program an incremental compilation is triggered, either by the environment itself or by the programmer. Since the compiler only compiles the small changes made to the program the compilation is instant and hopefully not noticeable. The environment has access to the compiled program at all times, which gives the environment unique possibilities. It can for example know which identifiers that are reachable from a certain point in the program, and thus present these to the programmer in, for instance, a menu. Since compilations are instant, so is the error reporting. The programmer can correct syntactic and semantic mistakes when they happen.

Incremental compilers make use of the time the programmer spends thinking about the problem and is immediately available to the programmer when he or she is ready to make the required changes. The negative effect of this is the lost coffee breaks for the programmer, due to less time spent waiting for the compilation to finish.

The work presented in this paper concerns the techniques for performing static semantic analysis in an incremental environment. The term incremental environment does not have to be a programming environment, but can be any program that demands immediate response to changes. An example of this could be a WYSIWYG-application.

Goals

The original goals for this project was:

- To rewrite the existing testbench[1] in BETA.
- To integrate an existing syntax directed editor.
- To extend the door package.
- To write a main-grammar generator.

Since the original goals were defined, some changes have been made. The main-grammar generator was left aside and so was the existing syntax directed editor due to practical problems. Research was done in integrating the Sif editor of the Mjølner BETA system, but finally we choose to write our own syntax tree editor as a side project. The rewrite of the testbench took up a large proportion of the time devoted to the project, something we had not anticipated.

Achievements

During the course of this project we have come in touch with many different problems and research areas. This has been interesting and educational, but also slowed the fulfilment of the original goals presented above. In spite of this we

[1] The original testbench, written by Görel Hedin in Simula, was used to test theories and language constructs introduced in [Hed92].

have achieved some results. The most important are the Embla metaprogramming system, a grammar based environment with incremental evaluator, and the specification and implementation of the Ask language. Below we give a short explanation of the areas we have studied during the project.

AG and Door AG

Attribute grammars and Door attribute grammars are the base for analysing and specifying the computer languages we have studied. Door Attribute grammars were first introduced in [Hed92] and both AG and Door AG are explained in chapter 4 and 5.

BETA and its Modularization Techniques.

The Embla system has been implemented using the BETA programming language [MMN93]. In the process of learning to master this many faceted language, we have struggled with the philosophy of the process of analysing, designing, and implementing a computer program. This has given us many insights and valuable approaches that we will carry with us. Our impressions and experiences of the BETA programming language, its modularization system, and associated tools are summarised in chapter 14.

Incremental Semantic Analysis

Incremental semantic analysis, which forms the basis of incremental compilers, is a difficult and demanding subject. As with the whole area of grammar based techniques it is described with extensive use of formalism. We have spent a lot time in trying to understand the knots and bolts of incremental semantic analysis.

The Mjølner Metaprogramming System

The Mjølner Metaprogramming System, or MPS [MPS92], is a view of metaprograms as programs that manipulate other programs. Examples of metaprograms are compilers and pretty-printers. In order for different metaprograms to manipulate a program, the program must be represented in such a way that the metaprograms can understand it. In most cases that representation is the source code itself, however in MPS that common representation is the abstract syntax tree, AST. In MPS one tool can parse the source code into an AST that the other tools can understand. That AST can be opened by another tool that can add or withdraw information from the AST. It can also use information in the AST placed there by another tool.

The Embla System

We started out with a testbench written in Simula by Görel Hedin. The testbench had a rudimentary syntax tree editor, a full incremental evaluator for a small object oriented language, and a door package. Our task was to rewrite the testbench in BETA, improve the system interface, and to extend the language it supported. The rewritten testbench became the Embla metaprogramming system and the language was named Ask. The goal of the rewrite was to restructure the testbench to use the powerful features found in the BETA programming language. By improving the system interface the integration of existing tools, such as Sif the syntax directed editor found in MPS, would be possible. The new interface also made it possible to write tools that used the Embla metaprogramming system at different levels. From the most language independent level to the level that gives complete access to the implementation of a language.

As a side project we developed a syntax tree editor, WMEdit, that used the Embla system to allow graphical editing of a program. Another goal of the Embla system was to organise the system in such a way that the extensions needed to support a new language would be as simple as possible.

The Ask Language

The Ask language is a small object oriented language, which supports a number of semantically difficult constructs. It is an extension to the language that the testbench originally supported. In order to specify the language we have used OOSL, the object oriented specification language, introduced in [Hed92]. From the OOSL specification we have implemented the Ask language for the Embla metaprogramming environment.

How to Read This Document

This document is structured in three parts:

- Part 1 is a grammar survey, introducing the formalism that this work is based upon. This part covers, context-free grammars, attribute grammars, object-oriented attribute grammars, and finally in chapter 5 Door attribute grammars.

- Part 2 discusses the structure and implementation of a metaprogramming system, using Door attribute grammars. This part also covers the Embla system and the foundations of the Ask language.

- Part 3 contains specification and analysis of the language extensions to Ask. Chapter 14 summarises our experiences using the BETA programming language.

PART 1 GRAMMAR SURVEY

This part is about grammars used in computer science and especially those used in compiler techniques. Our aim is to explain the “whys” and “hows” of specifying programming languages using grammars. Let us start by looking at a dictionary definition of the word *grammar*.

> **grammar** *n.* the science dealing with the systematic rules of a language, its forms, inflections and syntax, and the art of using them correctly [...] ‖ the applying of these rules ‖ the system of forms and syntactical usages characteristic of any language ‖ a book of grammar [...] [Web92].

There is some ambiguity in the term, meaning the science of dealing with language rules, the usage of language rules when speaking, and a description of these rules. We will use the term *grammar* to mean the rules and elements constituting a (programming) language.

The linguist often talks about different types of grammars, for instance:

- *Descriptive grammar* – a description containing rules of how a language is actually spoken.
- *Prescriptive grammar* – a description containing rules of how to speak correctly according to some preferences.
- *Teaching grammar* – a collection of language rules and word definitions used as an aid in learning a new language.

Trying to accomplish a complete description grammar the linguists developed a formal method for describing natural languages. These formal methods have then been used in computer science for programming language specifications. They could constitute something that can be called a *definition grammar*, i.e., a grammar stating the correct usage of a (programming) language. Specifying a (programming) language with a grammar has several advantages such as simplifying the construction of compiler and translators for the language.

In this short survey we cover three types of grammars: chapter 3 covers *context-free grammars* (*CFG*), chapter 4 *attribute grammars* (*AG*), and chapter 5 *Door attribute grammars* (*Door AG*). First in chapter 2 we introduce some concepts and terms used later.

What do we mean by language? Many different answers are conceivable, e.g., natural languages in a broad sense, a specific natural language such as English, sign language such as the American Sign Language, query languages such as SQL, formal specification languages such as **Z** or programming languages such as BETA. For most people the term language probably means *natural language*. If a person is involved in computer science the term language will most certainly mean some formal language. Natural languages are those languages we use when communicating between people. The characteristics of these languages are that they evolve naturally over a long time, are often large and complex, and are rich in ambiguities. On the other hand the formal languages are always designed by man, are often small, and are hopefully designed in a way that ambiguities are avoided.

Among the formal languages our main interest lies in the group of programming languages. These are languages constructed for the sole purpose of programming computers, i.e., instructing computers what to do.

Syntax and Semantics

Although very different, all languages have some properties in common. Every language has its *syntax* and *semantics*. Syntax and semantics are terms used in describing languages. In the syntax of a language the form or rules of sentence formation are described. For natural languages, syntax tell us how to build phrases and sentences from the vocabulary of the language. E.g., a sentence in English can be built using a noun phrase (NP) followed by a verb phrase (VP) where the noun phrase is made of an article (Art) and a noun (N) and the verb phrase is constructed from a verb (V) and a noun, as in the example:

```
The boy counted cars.
   NP        VP
 Art  N     V     N
```

For programming languages, syntax describes the rules for forming correct sentences. For example an assignment in Simula [DMN68] can be constructed using an identifier (ID) followed by an assignment operator (":=") and an expression. The expression can be built using two identifiers and a multiplication ("*") operator, as in:

```
Y  :=  X * X;
ID     ID  ID
```

Most languages are different in syntax. The following two examples, in the programming languages BETA [MMN93] and Simula, are syntactically very different but perform the same computation. We say that they are *semantically* equivalent, performing the operation of assigning the square of the identifier *X* to the identifier *Y*.

Y := X*X;
X -> (# i: @integer **enter** i **exit** i*i #) -> Y;

The semantics of a language describes the meaning of phrases in the language. In natural languages, semantics deals with word meaning, i.e., the meaning of words as described in a lexicon, and phrase and sentence meaning, i.e., the meaning of a whole sentence or phrase. In programming languages, semantics describes what the program means, i.e., what computation is performed when the program is executed. For example the statements above has the semantics of returning the square of a supplied integer.

Syntax and semantics are not the only properties of a language. For natural languages we have phonology – the sound system, morphology – the rules of word formation, and the lexicon – the words included in the language. For programming languages other properties are conceivable such as characters or tokens (the "words" of the programming language).

There is no sharp border between syntax and semantics. Which parts of a language description that are in the syntax or in the semantics can vary. For programming languages a common example of this is type matching. Take for instance the expression *X*2* in the programming language Pascal [Wir71] and assume that the identifier *X* is declared of type *char*. This is erroneous but the error can be either syntactic or semantic:

- If it is a syntactic error then the program must be considered as a whole, i.e. the identifier *X* is used inconsistently with its declaration.
- If this is a semantic error the expression will fail since the semantics of the language defines that both operands to multiplication must be a number.

For programming languages both syntax and semantics can be subdivided further. Some terms frequently used are:

- *Abstract syntax* – the hierarchical structure of the language without concern for the actual symbols used.
- *Concrete syntax* – both the hierarchical structure and the symbols.
- *Static semantics* – the parts of the semantics that can be checked at compilation time.

Compiler Overview

In this section we recapitulate some elementary concepts in compiler techniques. The tasks of a compiler are often broken down in four different phases (figure 2.1):

- *Lexical analysis* – translates the stream of characters in the program into *tokens* – single units representing the "words" of the language (see chapter 3).
- *Syntactic analysis* – translates the tokens into a hierarchical structure. This phase uses the *context-free grammar* to structure the tokens into an *abstract syntax tree* (*AST*) (see chapter 3).
- *Semantic analysis* – Checks for static semantic errors and evaluates all attributes in the AST (see chapter 4).
- *Code Generation* – translates the attributed AST into code for the target machine.

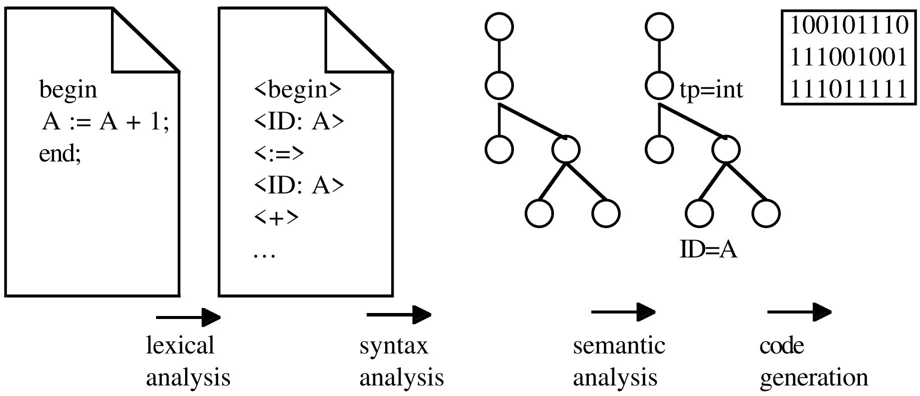


Figure 2.1: The four major phases in a compilation.

This is a somewhat simplified description, ignoring phases such as intermediate code generation and code optimisation. Grammars can be used in all the four phases.

The compilation of programs is traditionally done in batch, meaning that if the source program is changed the whole program must be recompiled. A small change in the program can lead to time-consuming recompilation of the whole program. If we can find efficient methods to isolate the parts of the program that have changed, and only recompile these parts, a lot of time can be saved. These methods can also be used to construct systems where the program is recompiled automatically after a change. This is the idea behind *incremental compilation*.

The later parts of this report (chapter 5 and onwards) is about methods, systems, and examples all using incremental compilation techniques. Furthermore, this report is about the semantic analysis phase. We assume that both lexical and syntax analysis are already done, and ignore the code generation phase altogether.

3 CFG and AST

In this chapter we look at *context-free grammars* (CFG). We explore the possibilities of context-free grammars in specifying the syntax and structure of programming languages. Other terms introduced in this chapter are *concrete* and *abstract syntax*, *syntax trees*, *BNF-notation*, *Chomsky Hierarchy*, *object-oriented context-free grammar*, and *structured context-free grammar*.

Let us start by an informal description of a context-free grammar. As an example we use the following small program in Pascal:

```
A: integer;
A := 0;
while A < 10 do
   A := A + 1;
```

In this program we have a variable declaration, an assignment, and a while-statement containing an assignment. After lexical analysis this will be a stream of tokens. All items surrounded with brackets (“<” and “>”) are considered single tokens. There is no structure in the program at this point.

```
<ID:A> <:> <integer> <;> <ID:A> <:=> <NUM:0> <;>
<while> <ID:A> <<> <NUM:10> <do> <ID:A> <:=> <ID:A>
<+> <NUM:10> <;>
```

Now, the aim of the syntax analysis is to structure this into a representation that reflects our view of the program, i.e., a variable declaration, an assignment, and a while-statement containing an assignment. One way of doing this is to represent the program as a *syntax tree*.

A syntax tree is a data structure containing leaves, called terminals, and internal nodes, called nonterminals. The leaves correspond to the actual tokens in the program. A word about the nonterminals: these are something we introduce in the process of structuring the program, and are not found in the program. For instance, we decide that the variable declaration has an identifier and a type. To model this we introduce the nonterminals *Decl*, *ID*, and *DeclType*. Figures 3.1

and 3.2 show the resulting syntax trees of our example program, where nodes in boldface represents terminals:

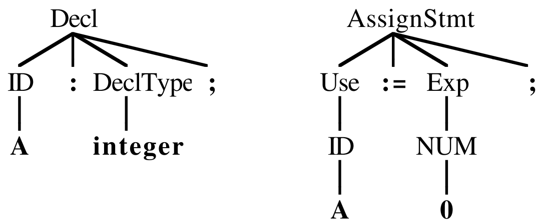


Figure 3.1: Syntax trees for the variable declaration and assign-statement.

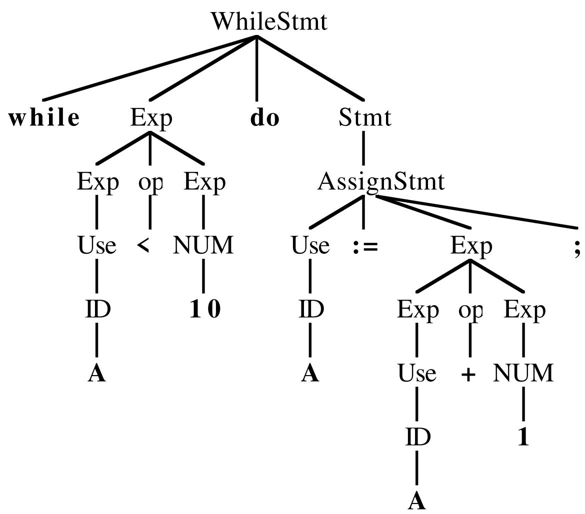


Figure 3.2: Syntax tree for the while-statement.

Trees are not the most practical way of representing the structuring rules of our programming language, so we need to be able to write these rules in another form. This can be done using something called a *context-free grammar* (*CFG*)[2].

A few words about the notation: A widely used notation for specifying CFG's is the *BNF form*, Backus-Naur form or Backus-Normal form. Here the symbol "::=" mean something like "have sons", nonterminals are written within brackets ("<" and ">"), and terminals are written in boldface[3]. To specify alternatives the symbol "|" is used. For example, a rule stating that the nonterminal *A* can as sons

[2]See the section "The Chomsky Hierarchy" for a formal definition of context-free grammar.

[3]The notation of terminals vary in different literature.

have either the nonterminal *B* followed by the terminal *b* or only the terminal *b*, would look like <A> ::= <B> **b | b**. Only one nonterminal is allowed on the left side.

We can now state the syntactic rules for our programming language in the BNF notation:

<Decl>	::=	<ID> **:** <DeclType> **;**
<DeclType>	::=	**integer**
<AssignStmt>	::=	<Use> **:=** <Exp> **;**
<Use>	::=	<ID>
<Exp>	::=	<Exp> <op> <Exp> | <Use> | <NUM>
<Op>	::=	**+ | <**
<WhileStmt>	::=	**while** <Exp> **do** <Stmt>
<Stmt>	::=	<AssignStmt> | <WhileStmt>

Note that we have not stated the rules for the nonterminals <ID> and <NUM>. <ID> denotes all valid identifiers and <NUM> denotes all integer constants that can be used in the program. The integer constants could be specified by the following rule:

<NUM> ::= **-MAXINT… | -3 | -2 | -1 | 0 | 1 | 2 | 3 | … MAXINT**

Recall the first example from chapter two. Breaking up a sentence in the English language ("The boy counted cars") in noun phrase (NP) and verb phrase (VP), and then the noun phrase into an article (Art) and a noun (N) and the verb phrase into a verb (V) and a noun. This can be represented with the following syntax tree and context-free grammar:

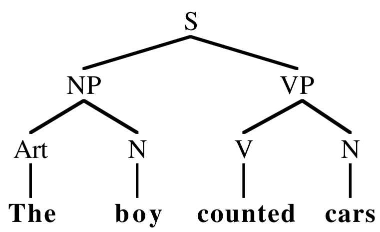


Figure 3.3: Syntax tree for a sentence.

<S>	::=	<NP> <VP>
<NP>	::=	<Art> <N>
<VP>	::=	<V> <N>
<Art>	::=	**The**
<N>	::=	**boy | cars**
<V>	::=	**counted**

Using this limited grammar we see that the sentence “The cars counted boy ” is syntactically correct although it is semantically erroneous.

Going back to the CFG for our example program, we see that this is a very limited grammar allowing only integer declarations, assign- and while-statements, and addition and less-than operations. The grammar can easily be extended by changing the rules. If we for instance would like to add boolean, real, and character declarations; an if-statement; and some more operators, this can be done by the following changes:

<DeclType> ::= **integer | boolean | real | char**
<Stmt> ::= <AssignStmt> | <WhileStmt> | <IfStmt>
<IfStmt> ::= **if** <exp> **then** <Stmt>
<Op> ::= **+ | - | * | / | < | > | = | <>**

It should be noted that this is not a complete or even correct specification of Pascal, it is only a simple example used to introduce context-free grammars. See [Wir71] for a complete description of Pascal.

Abstract Syntax Trees

The syntax-trees shown i figures 3.1 and 3.2 are called *concrete* syntax trees or *parse-trees*. These are trees containing all information about what the program looked like, including all tokens. Often much information in the concrete syntax tree is used only in lexical analysis an parsing, so a more compact representation could omit this information. This is true for the *abstract syntax tree* (*AST*). Figures 3.4 and 3.5 show the abstract syntax tree for our variable declaration, assign-statement, and while-statement.

Figure 3.4: Abstract syntax trees for the variable declaration and assign-statement.

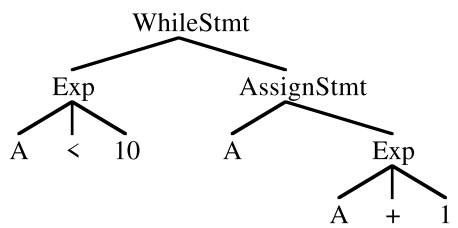


Figure 3.5: Abstract syntax trees for while-statement.

These trees differ from the concrete syntax trees in two ways:

- All syntactic information, i.e., all information used only for lexical analysis, is removed. In the above example the nodes for **:**, **:=**, **;**, **while**, and **do** are removed.
- All nodes having only one son are replaced with that son.

Both forms of syntax trees can be used as the internal representation after syntax analysis. In the rest of this report we will make extensive use of abstract syntax trees and we use the term syntax tree to mean abstract syntax tree.

Different concrete syntax trees can lead to the same abstract syntax tree. The program for the while-statement in the example above can in C be written as:

```
while (A < 10)
   A++;
```

or in ADA as:

```
while A < 10 loop
  A := A + 1;
end loop;
```

The concrete syntax trees for these examples are different but both can be represented by the abstract syntax tree in figure 3.5.

The Chomsky Hierarchy

In the above sections we talked about languages and grammars in a very informal way. In this section we make a short presentation of formal languages and the Chomsky hierarchy, leading up to a formal definition of context free grammars.

First we need to define the terms *alphabet*, *string, formal language*, and *grammar*.

Definition Alphabet
An alphabet is any finite set of symbols. •

Definition String
A string over some alphabet is a finite sequence of symbols drawn from that alphabet. •

Definition Formal Language[4]

A formal language denotes any set of strings over some fixed alphabet. •

For example, given the alphabet $\Sigma = \{a,b\}$, *abba* is a string over Σ, and $L=\{a,b,ab,abba\}$ is a language over Σ. In addition to this we could add a set of rules to determine what strings of symbols from Σ are legal words of L, and define the term *grammar*:

Definition Grammar

An alphabet and a set of rules for forming strings. •

One way of specifying these rules is simply to list all strings in the language. So for the language L the rules could say that *a*, *b*, *ab*, and *abba* are valid strings. This specification method, called a *list grammar*, is not very useful because we often need to specify infinite languages. Take for instance a grammar containing a rule, stated informal as, all strings beginning with one or more *a* and finished with one *b* is in the language. Lets call this language L_1. To list all strings starting with *ab*, *aab*, *aaab* … would not be a good idea. We clearly need another method so we introduce *production system*.

Definition Production system

A production system consists of two alphabets and a set of rules for forming legal strings. The alphabets are V_T the set of *terminals* and V_N the set of *nonterminals*. The set of nonterminals contains a start symbol, S. The rules are also called *productions*. •

To clarify this let us look at the example above. The informal rule for the language L_1 can now be specified using two rules as:

$$S \to aA$$
$$A \to aA \mid b$$

S is the start symbol meaning "start here", i.e., use this rule first. a and b are the terminals, i.e., the symbols used in the output string. S and A are nonterminals, these symbols are introduced as syntactic variables in the productions. The first rule, $S \to aA$, states that from the nonterminal S the terminal symbol a followed by the nonterminal symbol A can be derived. The second rule, $A \to aA \mid b$, says that from the nonterminal symbol A the terminal symbol a followed by the nonterminal symbol A or only the terminal symbol b can be derived. Note that the vertical bar ("|") means or. Applying these rules, starting from the start symbol S, all strings beginning with one or more a and ending with one b can be de-

[4]In this section we use the term language to mean formal language.

rived. Now using the production system the grammar for the language L_1 can be written as:

G_{L1} = The terminals: $V_T = \{a, b\}$.
The nonterminals: $V_N = \{S, A\}$, where S is the start symbol.
The production rules: $\{S \rightarrow aA\ , A \rightarrow aA \mid b\}$.

In the rest of this section we use the following notation[5]:

- Terminals: lower-case letters early in the alphabet, such as *a*, *b*, *c*; operator symbols such as +, -, *; punctuation symbols such as parentheses, comma etc.; the digits 0-9; boldface strings such as **while**, **id**.
- Nonterminals: upper-case letters early in the alphabet, such as *A*, *B*, *C* ; the letter *S* is used as the start symbol.
- The empty string: the Greek letter epsilon, ε.
- Strings of terminals: lower-case letters late in the alphabet, such as *u*, *v*, *w*, *x*, *y*, *z*.
- Strings of grammar symbols: lower-case Greek letters α, β, γ. Strings of grammar symbols are strings including both terminals and nonterminals.
- Alphabet: V_T denotes the set of terminal symbols; V_N denotes the set of nonterminal symbols; $V = V_T \cup V_N$ denotes the set of all symbols.
- Repetitions: we use the notation a^* to mean *a* repeated zero or more times and a^+ to mean *a* repeated one or more times; V^* means string from the alphabet *V* containing zero or more symbols, and V^+ means string from the alphabet *V* containing one or more symbols.

By adding more restrictions to the definition of grammars it is possible to define a language hierarchy. This hierarchy is normally called the Chomsky hierarchy after its inventor the American linguist Noam Chomsky. In the hierarchy the languages with higher type number form subclasses of languages of lower type[6] (see figure 3.6).

The first, and the most powerful, grammars are type 0 or *unrestricted grammars*. In these grammars the only change to the original definition of grammar is that the left side of a production must contain at least one nonterminal. That is, each production is of the form $\alpha \rightarrow \beta$ where $\alpha, \beta \in V^*$ and α contains at least one nonterminal. Unrestricted grammars are also called *phrase structure grammars*.

---

[5]This is the same notation as used in [ASU86].

[6]This classification can be done slightly different. Here we follow the presentation in [HU69].

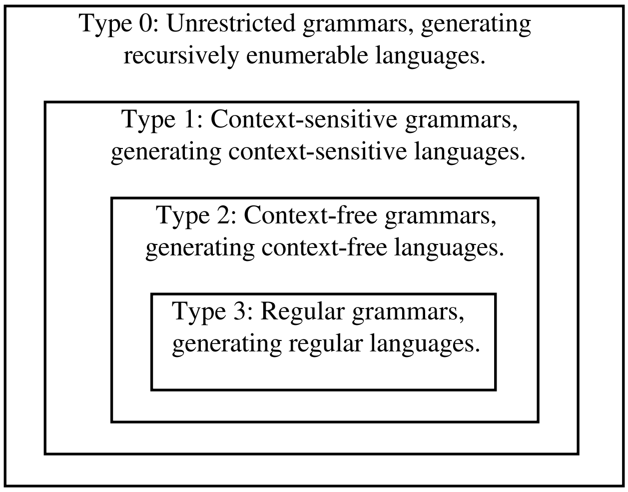


Figure 3.6: The Chomsky hierarchy.

The next restriction to make is to restrict the productions so that, for any production, the length of the left side is less than or equal to the length of the right side. This class of grammars is called type 1 or *context-sensitive grammars* (CSG). Put more formally each production must be of the form:

$$\alpha A\gamma \rightarrow \alpha\beta\gamma \text{ where } A \in V_N,\ \alpha,\gamma \in V^* \text{ and } \beta \in V^+$$

In type 2 grammars, also called *context-free grammars* (CFG), the restriction on the production is that each production must have a nonterminal on the left side. Formally, all productions are of the form $A \rightarrow \alpha$ where $A \in V_N$, and $\alpha \in V^*$.

In the last class, type 3 grammars or *regular grammars*, the restriction is that, for all productions, the left side must be a nonterminal and the right side can contain at most one nonterminal, that is, all productions have the form $A \rightarrow uBv$ or $A \rightarrow u$ where $A,B \in V_N$, and $u,v \in V_T{}^*$.

The languages generated by these grammars are called *recursively enumerable*, *context-sensitive* (*CSL*), *context-free* (*CFL*), and *regular*. It is possible to include the empty string (ε) in the definition of context-free language, context-sensitive language, and regular language. This can be done by adding the following rule (S is the start symbol):

$S \rightarrow \varepsilon$ and S does not occur to the right of any production.

For a thorough description of these grammars and the languages they define see [HU69], [HU79], and[Ray86].

One result of the restrictions applied to the grammars is that the power, i.e., the number of strings recognised, decreases. This means that it is easier to recognise a string belonging to a context-free grammar than it is to recognise a string for an unrestricted grammar. The most suitable grammars for use in conjunction with programming languages are regular grammars and context-free grammars. Regular grammars are often used in specifying the lexical structure of a programming language, i.e., the rules explaining how to assemble individual characters in a program text to atomic symbols called *tokens*. As seen in the previous section, context-free grammars can be used in specifying the syntax of programming languages.

Putting all this together we can define context-free grammars as:

> **Definition** Context-free grammar (CFG)
> A CFG consists of two alphabets and a set of rules (productions) for forming legal strings. The alphabets are V_T the set of *terminals* and V_N the set of *nonterminals*. The set of nonterminals contains a start symbol, S. All rules have a nonterminal on the left side and a sequence of nonterminals and/or terminals on the right side. This can formally be noted as a 4-tuple G = (V_T, V_N, S, P) where P is the set of rules. •

One word can be said about the productions. All productions are rules of the form $A \rightarrow \alpha$ where A is a nonterminal and α is a string consisting of nonterminals and/or terminals. A production, $A \rightarrow \alpha$, would mean that the symbol (nonterminal) A can be replaced by the string α. If more than one rule is applicable for the symbol A, this can be specified using the bar character ("|"), as in $A \rightarrow \alpha \mid \beta \mid \gamma$.

Structural Ambiguity

When applied to a string the syntax rules of a language describe several different things, for instance:

- The grammaticality of the string, i.e., is this a valid string in the language.
- The order of symbols in the string.
- The structure of the string.

The last two points say that given the rules used to identify the string as part of the language and the order in which they apply, we can tell in which order the individual symbols occur in the string, and we can figure out the *syntactical structure* of the string. The term syntactic structure can best be explained using an example from natural language. Consider the sentence:

"The boy saw the man with the telescope."

This could mean two things:

1. The boy sees a man through his telescope or

2. The boy sees a man and the man has a telescope.

Thus the sentence has two different syntactic structures. Another example of structural ambiguities is the expression $a + b * c$. This can mean either calculate the sum of a and b and the multiply this by c or it can mean (and usually does) multiply b and c and then add a. These structures can be pictured graphically in the following abstract syntax trees:

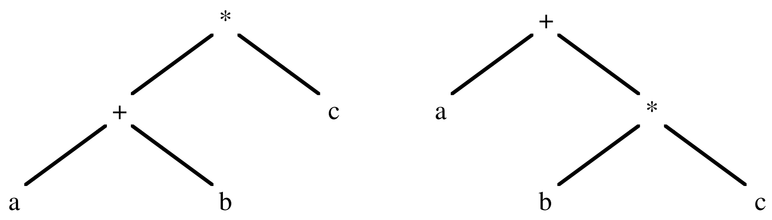


Figure 3.7: Different representation of the expression a+b*c.

We can define rules for an unambiguous grammar that will recognise the expression and will use the standard operator precedence (* before +). Using BNF notation the rules can look like:

<exp> ::= <exp> **+** <term> | <term>	(* rule 1 *)
<term> ::= <term> ***** <factor> | <factor>	(* rule 2 *)
<factor> ::= **symbol**	(* rule 3 *)

symbol is a terminal representing some character. Now using the rules "backwards" we can show that our string is an expression.

$a + b * c$	→ <factor> + <factor> * <factor>	using rule 3 on a, b, and c
	→ <term> + <term> * <factor>	using rule 2 on the first two <factor>
	→ <term> + <term>	using rule 2 on <term> * <factor>
	→ <exp> + <term>	using rule 1 on the first <term>
	→ <exp>	using rule 1 on <exp> + <term>

The concrete syntax tree for this new unambiguous CFG is pictured in figure 3.8. Note that the right-hand side of a production corresponds to sons of the left-hand side node. In figure 3.8 the production <exp> ::= <exp> **+** <term> show up in the concrete syntax tree as a node exp with three sons exp, +, and term.

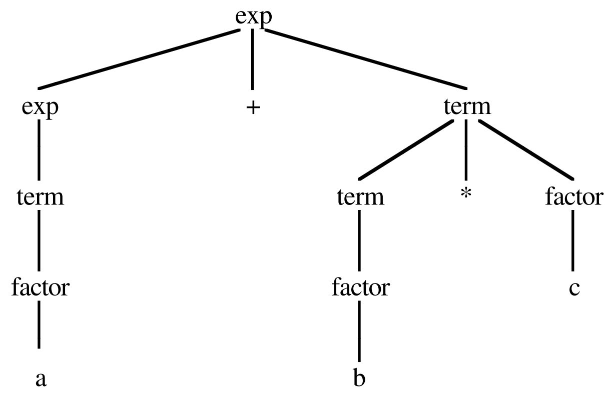


Figure 3.8: Concrete syntax tree for a+b*c (obeying standard operator precedence).

Other Types of CFG

By modifying the context free grammar definition, it is possible to define other variants of context free grammars. We present here two different but related types, *object-oriented grammars* – using the all present ideas of object orientation, and *structured grammars* – used in the Mjølner BETA System [MPS92].

Object Orientation and Context Free Grammars

Object-orientation (*OO*) is a term widely used today in many areas of computer science, e.g. object-oriented programming languages, object-oriented databases, and object-oriented operating systems. The ideas of object orientation originated from the programming language Simula. The goal of object-orientation techniques is to obtain better structuring mechanisms. The problem to be solved is divided into classes. These classes are structured in class hierarchies. Inheritance is used to transfer properties from super classes down to subclasses. The solution of the problem consists of creating and executing instances (called objects) of the classes. There are two types of inheritance: single inheritance meaning that a class can only inherit properties from one superclass; and multiple inheritance meaning that a class can inherit from multiple superclasses. An common example of (multiple) inheritance is the concept of amphibian vehicle sharing properties from both cars and boats.

This OO structuring can be done for two main reasons: “code reuse” and “modelling the real world”[7]. If “code reuse” is the reason, the classes are constructed so that each class forms a well structured part of the problem and can be

[7]The Scandinavian school of object-orientation.

(re)used in different applications. If "modelling the real world" is the reason, the classes are constructed to model some aspect of the real world.

Object-orientation can also be used to structure a CFG specification, as described in [Kos91][8]. Here nonterminals correspond to classes, and nodes in the syntax tree correspond to objects. We divide the nonterminals into two categories, *basic nonterminals* and *superclass nonterminals*. Basic nonterminals are those nonterminals having only a single production and superclass nonterminals are those having a set of chain productions, i.e., productions of the type $A \rightarrow B \mid C \mid D$. The latter category is used to determine class hierarchy. This is done by assigning, for each chain production, that a production of type $A \rightarrow B$ means that A is a superclass of B. The productions for the basic nonterminals give rise to component objects for the left-hand side nonterminal. This means that if the CFG contains a production $A \rightarrow B\ \ C\ \ D$ the (basic) nonterminal A will contain three component objects and these objects will at least be of class B, C, and D respectively. Seen another way, if the (basic) nonterminal occurs in a syntax tree it will have three sons and these sons will at least be of class B, C, and D.

Example 3.1: Expressions and OO context-free grammar.

All context-free grammars can be converted into object-oriented context-free grammars. As an example let us look at the expression $a + b * c$ introduced above. The CFG for our limited expressions was:

<exp> ::= <exp> **+** <term> | <term>
<term> ::= <term> ***** <factor> | <factor>
<factor> ::= **symbol**

To turn this specification into an object-oriented CFG we break up the productions for <exp> and <term> introducing the basic nonterminals <sum> and <mult>.

<exp> ::= <sum> | <term>
<sum> ::= <exp> <addop> <term>
<term> ::= <mult> | <factor>
<mult> ::= <term> <mulop> <factor>
<addop> ::= **+**
<mulop> ::= *****
<factor> ::= **symbol**

---

[8]Although this description is from [Kos91], object orientation, or related notation, in language specification/implementation has been suggested in several different papers. See [Kos91] for more information.

The concrete syntax tree for $a + b * c$. would then look like:

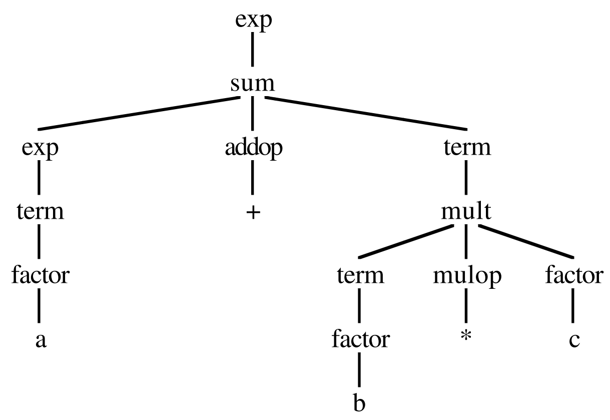


Figure 3.9: Concrete syntax tree for a+b*c using an OO context-free grammar.

Using the chain rules for <exp>, <term>, <mulop>, <addop>, and <factor> we can construct the class hierarchy for this example.

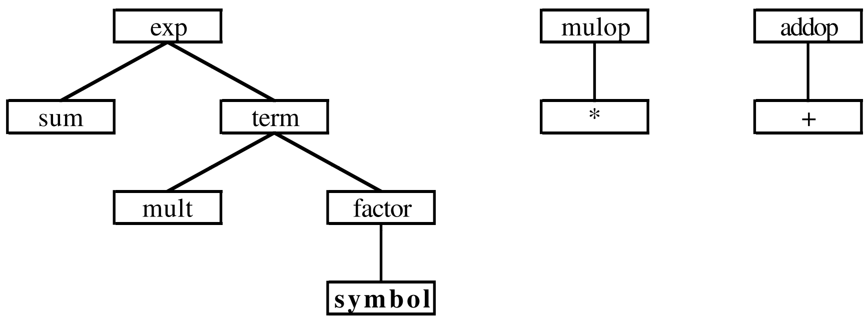


Figure 3.10: Class hierarchy derived from the OO context-free grammar.

The two remaining rules are used to construct the needed component objects for <sum> and <mult>.

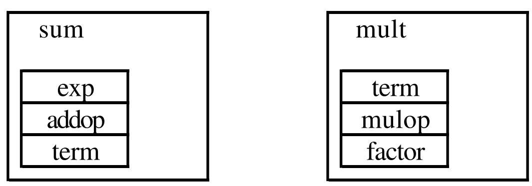


Figure 3.11: Component objects derived from the OO context-free grammar.

This means that if we have a <mult> node in the syntax tree this node will be a subclass of <term>, and <term> will be a subclass of <exp>. The <mult> node will have three sons in the syntax tree, a <term>, a <mulop>, and a <factor> node. Things will be even more complicated when we introduce attributes in the section on object oriented attribute grammars. •

Without going into too much formality we now state the rules that makes a context-free grammar an object-oriented context-free grammar.

1 . The CFG part concerning the superclass nonterminals (i.e. the chain rules) must be cycle-free. This is to avoid cyclic class hierarchies.

2 . All nonterminals must either be used directly in some derivation of a terminal or a subclass to the nonterminal must be used in some derivation of a terminal.[9]

3 . For all nonterminals there must be only one production or if there is more than one production the right-hand side of the production must be a single nonterminal. That is, all nonterminals are either basic nonterminals or superclass nonterminals.[10]

4 . If we only allow single inheritance, then if we have two productions $A \rightarrow B$ and $C \rightarrow B$ this implies that $A = C$.[11]

Structured CFG

This special version of CFG is defined in [MPS92] and used extensively in the Mjølner BETA System. A *structured CFG* is a context-free grammar where the productions satisfy a certain structure. Each nonterminal must be defined by exactly one of the following rules:

1. Alternation rule: a rule of the form <*A0*> ::| <*A1*> | <*A2*> | … | <*An*> where <*An*> are nonterminal symbols. The "::|" corresponds to "::=" in the BNF notation and the vertical bar means or (as in BNF).

2. Constructor rule: a rule of the form <*A0*> ::= *w0* <*t1:A1*> *w1* … <*tn:An*> *wn* where <*A0*> and <*ti*:*Ai*> are nonterminals and *wn* are possibly empty strings of terminal symbols. As expected the rule means <*A0*> derives the string on the right of "::=". *ti* stands for tag-name and

---

9 A CFG with this property is called *pseudo-reduced* in [Kos91].

10 A pseudo-reduced CFG with this property is called *MI-Structured* in [Kos91].

11 A MI-Structured CFG with this property is called *SI-Structured* in [Kos91].

are used to distinguish between different nonterminal occurrences. All tag-names must be different.

3. List rules are used to specify that a nonterminal can derive a list of strings. There are two list rules <*A*> ::+ <*B*> *w* and <*A*> ::* <*B*> *w*. Where <*A*> and <*B*> are nonterminals and *w* is a possibly empty string of terminals. The rule means <*A*> generates a list of <*B*> separated by *w*. "::*" means zero or more elements in the list and "::+" means one or more elements.

4. Optional rule, a rule of the form <*A*> ::? <*B*> meaning that the nonterminal <*A*> may derive the same string as the nonterminal <*B*> or the empty string.

Nonterminals defined by the alternation rule correspond to superclass nonterminals in OO-CFG, and nonterminals defined using the constructor rule correspond to basic nonterminals. The main reason for using structured CFG in the Mjølner BETA System is to make it possible to generate class and object definitions automatically from the context-free grammar.

For a large example of a structured CFG see [MMN93] where the whole of the BETA programming language is specified using structured CFG.

CFG and Compilers

By specifying the syntax of a programming language we can design programs that, given the grammar of the language, can generate a compiler for that language. As we shall see in the next section CFG has several limitations, so the concept of automatic compiler generation from formal specifications is only useful to limited extent for CFG. This idea will play an important role in later sections when the formal notation is extended to *attribute grammars* and *Door-attribute grammars*.

A formal specification of the programming language syntax can also (and have often done so) serve as an aid to compiler designers in implementing compilers.

Limitations of CFG Specifications

As the name suggests context-free grammars cannot be used to specify things that require that we take the context into consideration. Such rules are very common in programming languages and are called *contextual constraints*. Example of these types of rules are:

- An identifier can only be used in the block where it is declared.

- In an assignment the left-hand and the right-hand sides must be of the same type.

Using a context-sensitive grammar we could formally specify these rules. However this has been found to be impractical. Instead we use CFG to specify the syntax of the language, and all rules not possible to specify in CFG are moved to the semantics of the language. Then we use an extended CFG, an attribute grammar (AG), to specify these rules. Attribute grammars is the subject of the next chapter.

4 Attribute Grammars

To overcome some of the limitations of context-free grammars we present *attribute grammars* (*AG*). First introduced in [Knu68], attribute grammars are context-free grammars to which we add extra information to all symbols, and add rules on how to find this information to all productions. The information we add is normally called *attributes* and the rules to obtain the information (evaluate the attributes) are called *semantic rules*.

The outline of this chapter is as follows: First we present the use of attribute grammars in two examples. Then we give a formal definition of attribute grammar. The next two sections explains *dependency graphs* and how to *evaluate* (find value for) the attributes. Thereafter we discuss two variants of attribute grammars, namely *ordered attribute grammars* (*OAG*) and *object-oriented attribute grammars* (*OO AG*). At the end of the chapter we present some of the limitations of attribute grammars.

Two Examples Using Attribute Grammars

The first example is a small attribute grammar capable of generating strings containing the symbol *a* or *b*.

Example 4.1: Small string

Starting off with the context-free grammar $G = (\{a, b\}, \{S, A\}, S, \{S \rightarrow A, A \rightarrow a \mid b\})$, that is, the terminals *a* and *b*, the nonterminals *S* and *A*, the start symbol *S* and the two productions $S \rightarrow A$ and $A \rightarrow a \mid b$. We add two attributes *what* and *str*. *what* is an attribute containing an explanatory string and *str* is an attribute containing the string we generate. These attributes are added to each symbol in the following way: to *S* we add *str*, to *A* we add *what* and *str*, and to *a* and *b* we add *what*.

Now we have to state the semantic rules, i.e., how the attributes get their values. These semantic rules are added to the productions.

$$
\begin{array}{lll}
S \rightarrow A & : & A.what := \text{“A small example”} \\
 & & S.str := A.str \\
A \rightarrow a & : & a.what := A.what \\
 & & A.str := \text{“a”} \\
A \rightarrow b & : & b.what := A.what \\
 & & A.str := \text{“b”}
\end{array}
$$

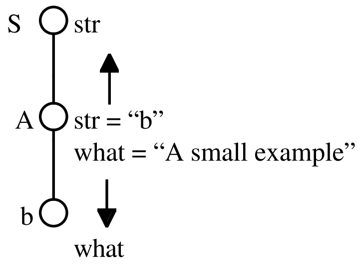


Figure 4.1: Attributed syntax tree for the string b, containing an inherited attribute, *what*, and a synthesized attribute, *str*.

Looking at the semantic rules we can see that the attribute *what* flows from the start symbol to the terminal symbols, i.e., down in a syntax tree. This means $A.what$ is assigned to a text in the production $S \rightarrow A$ and in the production $\mathrm{A} \rightarrow b$ the attribute is copied to b using the semantic rule $b.what := A.what$. Attributes that get their value from attributes in the father node are called *inherited attributes*.

The attribute *str* is assigned from the terminals and up to the start symbol, it is assigned upwards in the syntax tree. $A.str$ is given a value in the production $\mathrm{A} \rightarrow b$ and is copied up to $S.str$ in the production $S \rightarrow A$. Attributes that get their value from attributes in the son nodes are called *synthesized attributes* (see figure 4.1). •

Example 4.2: Binary numbers

To clarify things further we present a larger example[12]. If we want to formally define the binary notation for numbers we can do this by using an attribute grammar. The underlying context-free grammar is defined as:

$$
\begin{array}{l}
V_T = \{0,\ 1,\ \cdot\} \\
V_N = \{B, L, N\}, N \text{ is the start symbol.} \\
P = \{\quad B \rightarrow 0 \mid 1, \\
\qquad\quad L \rightarrow B \mid L\ B, \\
\qquad\quad N \rightarrow L \mid L \cdot L\}
\end{array}
$$

[12]Example originally from [Knu68].

The terminals are the binary numbers ("0" and "1") and the decimal point (".").
The nonterminals are B, standing for bit; L, for lists of bits; and N, for number.

The productions state that:

- A bit is either "0" or "1" ($B \rightarrow 0 \mid 1$).
- A list is either a bit or a list followed by a bit ($L \rightarrow B \mid L\,B$).
- A number is either a list or two lists separated by a decimal point ($N \rightarrow L \mid L \cdot L$).

Now we want to extend this CFG so that for each string of binary numbers we associate a decimal value corresponding to the string's binary value. For instance, the binary string 1101·01 is equal to the decimal value 13.25. A synthesized value attribute (v) is used to propagate the decimal value up to the root node (N). The CFG is extended with attributes in the following way:

> B has a rational attribute value, v, and an integer attribute scale, s.
> L has two integer attributes: scale, s, and length, l, and a rational attribute value, v.
> N has a rational attribute value, v.

v and l are synthesized attributes and s is an inherited attribute. The value attribute (v) is used to store the decimal value for each bit. Scale (s) denotes which position the bit has in the number and length (l) is the position in the list. To this we add the semantic rules to the productions.

p1:	$B \rightarrow 0$	$B.v := 0$
p2:	$B \rightarrow 1$	$B.v := 2^{B.s}$
p3:	$L \rightarrow B$	$L.v := B.v$, $B.s := L.s$ and $L.l := 1$
p4:	$L_1 \rightarrow L_2 B$	$L_1.v := L_2.v + B.v$, $B.s := L_1.s$, $L_2.s := L_1.s + 1$ and $L_1.l := L_2.l + 1$
p5:	$N \rightarrow L$	$N.v := L.v$ and $L.s := 0$
p6:	$N \rightarrow L_1 \cdot L_2$	$N.v := L_1.v + L_2.v$, $L_1.s := 0$ and $L_2.s := -L_2.l$

Let us clarify these rules:

p1: If the bit is assigned a "0" the value is set to zero.

p2: If the bit is assigned a "1" the value is set to 2 to the power of the scale attribute.

p3: The first two equations are what are called *copy equations*. They copy attributes from one node in the syntax tree to another. Here this applies to the attributes v and s. The last equation sets the length attribute of the list to one.

P4: Here we have four equations: First the value attribute for the list node L_1 is set to the sum of the value attributes of the sons (L_2 and B), the value is propagated upwards in the tree. The next two equations propagate the scale attribute down to B and L_2. $L_2.s$ is set to one more than $L_1.s$ Finally the length attribute for the list L_1 is set one more than the length attribute for L_2.

p5: The value attribute, *v*, is copied to *N* and the scale attribute, *s*, for the list node is set to zero.

P6: The value attribute, *v*, is set to the sum of the value attribute of the two sons (L_1 and L_2) and the scale attribute, *s*, is set to zero in L_1 and to $-L_2.l$ in L_2.

Figure 4.2 shows the syntax tree corresponding to the string 1101·01 in which all attributes are shown with their values. The value attribute, *v*, for the root node contains the decimal value for the binary string ($1101{\cdot}01_2 = 13.25_{10}$). •

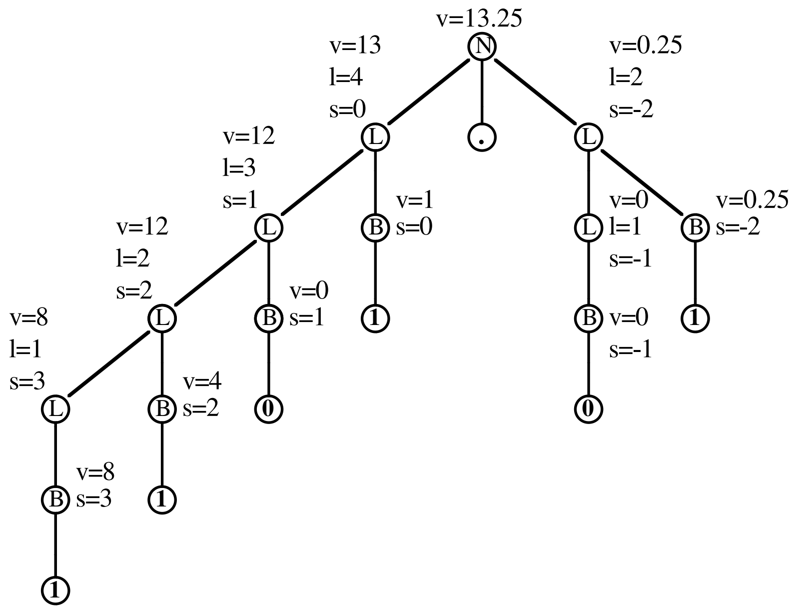


Figure 4.2: Attributed syntax tree for the binary number 1101·01.

Formal Definition of Attribute Grammar

We are now ready for a formal definition of attribute grammar:

Definition Attribute grammar

An attribute grammar is a context-free grammar, G = (V_T, V_N, *S*, *P*) to which we add attributes and semantic rules in the following way: To each symbol $X \in V_T \cup V_N$ we associate a finite set of attributes, *Attr*(*X*). The attributes are partitioned into two disjoint sets, the synthesized attributes, *Syn*(*X*), and the inherited attributes, *Inh*(*X*). We require *Inh*(*S*) to be empty and we require *Syn*(*X*) to be empty if *X* is a terminal.

For each production p: $X_0 \rightarrow X_1 \ldots X_{np}$ we give a set of semantic rules defining the synthesized attributes for the symbol X_0 and the set of inherited attributes for the symbols $X_1 \ldots X_{np}$ in terms of the elements in $Attr(X_i)$ for $0 \le i \le n_p$. •

From the definition we have that the start symbol, i.e., the root node in any syntax tree, cannot have any inherited attributes and that terminals cannot have synthesized attributes. The semantic functions defining the attributes in a production have the form $a_0 = f(a_1, \ldots, a_m)$ where $a_0 \ldots a_m$ are attributes of the symbol X_0 or the sons of X_0. When applied to an instance of a production $a_0 \ldots a_m$ denote attribute instances.

An attribute grammar is said to be *well-formed* if it fulfils the properties that the start symbol does not have any inherited attributes and that for each production the only equations are those defining all synthesized attributes of X_0 and all inherited attributes of sons to X_0.

For each nonterminal node in the syntax tree a certain production is *applied*. Consider figure 4.3 that shows a part of a syntax tree. We have one production, p: A → B C, which is applied twice.

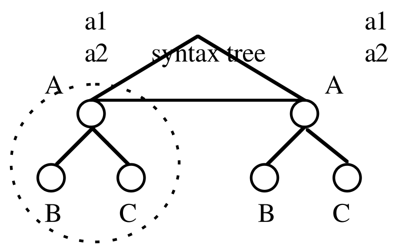


Figure 4.3: Part of syntax tree with two instances of the production $A \rightarrow B\ C$.

The encircled part in figure 4.3 is called an *instance* of the production $A \rightarrow B\ C$. Each nonterminal node (except the root node) belongs to two instances. Once as the right-hand side of a production and once as the left-hand side. For each symbol in the syntax tree there is an attribute instance for each attribute assigned to that symbol. In figure 4.3 we have two attributes for the symbol A ($Attr(A) = \{a_1, a_2\}$) and two instances of each of the attributes a_1 and a_2.

Other terms frequently used for attribute grammars include:

- *available* – attributes that have received values are said to be available.
- *consistent* – an attribute is consistent if all its arguments are available and its defining function holds. A tree is consistent if all its attribute instances are consistent.
- *decorated (attributed) syntax tree* – a syntax tree in which all attributes have a value.

- *solution* – if the syntax tree is consistent its attributes is a solution to the equation system in which all attributes instances are variables.
- *uniquely-defined* – the grammar is uniquely-defined if every possible syntax tree has exactly one solution.
- *well-defined* – the grammar is well-defined if every possible syntax tree has at least one solution.
- *underdetermined* – the grammar is underdetermined if it is well-defined but some syntax tree has more than one solution.

Dependency Graphs

An important concept for attribute grammars is the *dependency graph.* This is a relation showing all the dependencies among the attributes. If we for instance take the last production (*p6*) in example 4.2:

$$N \rightarrow L_1 \cdot L_2 \qquad N.v := L_1.v + L_2.v \text{ , } L_1 .s := 0 \text{ and } L_2.s := -L_2.l.$$

Here the attributes $L_1.v$ and $L_2.v$ are used to calculate $N.v$ thus we say that $N.v$ depends on $L_1.v$ and $L_2.v$. In the same way $L_2.l$ is used to obtain the value for $L_2.s$ so $L_2.s$ depends on $L_2.l$. These dependencies can be represented in the dependency graph for this production. In the graph there is a node for each attribute and a directed edge for each dependency (see figure 4.4).

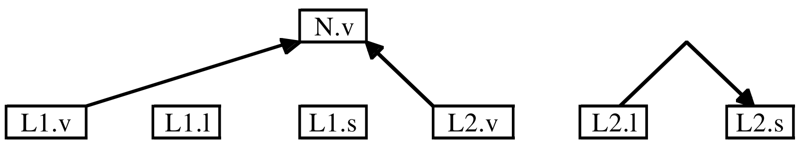


Figure 4.4: Dependency graph for production $N \rightarrow L_1 \cdot L_2$.

We denote this dependency graph *D*(*p6*). If we combine the dependency graphs for all production instances we get a directed graph describing all dependencies of attributes in the syntax tree, this we denote *D*(*T*). Dependency graph can be defined as:[13]

Definition Dependency graph
A directed graph that represents functional dependencies among the attribute instances of a production *p* or a tree *T*, denoted *D*(*p*) and *D*(*T*), respectively, and defined as follows:
a) For each attribute instance *b*, the dependency graph contains a vertex *b*.
b) If attribute instance *b* is an argument of attribute instance *c*, the depen-

[13]Definition from [Rep84] page 18.

dency graph contains a directed edge (b, c).
An edge from b to c has the meaning: b is used to determine the value of c.
•

Dependency graphs play an important role in *attribute evaluation* – the process of assigning the correct value to attribute instances.

Evaluation

Evaluation techniques are well studied, and several different methods have been suggested. Here we explain the basic terminology and some variants of attribute evaluation important to this work.

The goal for evaluation is to assign the correct values to all attribute instances in the syntax tree. First we talk of *exhaustive* and *incremental* evaluation. Exhaustive evaluation evaluates all attributes in a given syntax tree whereas incremental evaluation restores consistency in a tree by re-evaluating attributes that have become inconsistent due to some changes in the tree.

The value for an attribute is defined by the semantic function for this attribute. Depending on the implementation of this function we get two evaluation techniques:

- *Data-driven evaluation* – where each attribute instance is represented with a cell in memory and the evaluation of the attribute stores the correct value in this cell.
- *Demand-driven evaluation* – where each attribute is represented by its semantic function. Each time the attribute is needed the semantic function is called.

Choosing between these two techniques is a matter of choosing between space and time, where data-driven consumes space and demand-driven time. Combining the two techniques we get *lazy attribute evaluation*; this technique uses the semantic function the first time the attribute is needed, stores the resulting value in a cache, and uses the cached value if the attribute is needed again.

If we assume that we only have inherited attributes in our syntax tree then the evaluation can simply be done by visiting all nodes in such a way that we ensure that all parents are visited before their sons. For each visited node we simply assign values for all inherited attributes for all sons. If only synthesized attributes are allowed the evaluation is simple if we visit all sons to a node before a visit to the node. But for attribute grammars to be useful things are not that simple. We must allow a mix of inherited and synthesized attributes and allow these attributes to be dependent on each other. This give rise to the problem of *evaluation order,* that is, in which order the attributes can be evaluated.

The dependency graph contains all dependencies between attributes in the syntax tree and can therefore be used in determining evaluation order. The dependency graph can be constructed either statically when the evaluator is constructed or dynamically during evaluation. This gives us a *static* and a *dynamic* evaluation algorithm.

Many different classes of attribute grammars have been suggested in the literature. As for the grammars in the Chomsky hierarchy the different attribute grammars constitute a grammar hierarchy with different expression power and different complexity for each class. This classification can be done based on different features of the grammar, e.g., based on the order between attributes or based on evaluation features such as static or dynamic evaluation, evaluation in one or several passes. The top of this classification is non-circular attribute grammars (anything allowed except cycles in the dependency graph for some syntax tree), and the bottom is S-attribute grammars (AG using only synthesized attributes) respectively.

We here make a brief presentation of two attribute grammars needed later, namely *ordered attribute grammar* and *object-oriented attribute grammar*.

Ordered AG

Ordered attribute grammars (OAG), introduced in [Kas80], is an important and often mentioned class of attribute grammars. These attribute grammars are characterised by the way the attributes are ordered. As stated in [Kas80]: "For each symbol of the grammar a partial order over the associated attributes can be defined, such that in any context of the symbol the attributes are evaluated in that order." This means that we have to calculate a partial order[14] for all attributes for each symbol ($X \in V_T \cup V_N$). This order must be applicable in all contexts in which the symbol can occur. This means that a static evaluation algorithm can be used.

Visit Sequences

The evaluator for ordered attribute grammars can be based on *visit sequences*. This means that the attribute dependencies are converted into a series of EVAL/VISIT/RETURN statements. Here EVAL(a) means evaluate the attribute instance a; VISIT(i,r) means visit the i'th son for the r'th time; and RETURN(r)

[14]Partial order is a relation $\Re$ on a set A that is reflexive, for all $x \in A$, $x \Re x$ holds; antisymetric, $x \Re y$ and $y \Re x$ implies $x=y$; and transitive, $x \Re y$ and $y \Re z$ implies $x \Re z$. A relation from two sets A and B is any subset of $A \times B$ ([Grim89]). For the order of attributes this means (assuming a, b, c are attributes): (a depends on a); if a depends on b and b depends on a then a=b; if a depends on b and b depends on c then a will depend on c.

means return to the father node for the *r*'th time. These visit sequences are computed for each production in the grammar.

The parameter *r* is included to allow evaluators to use more than one visit to each node. If, during exhaustive evaluation, each node is visited only once the grammar is said to be *1-visit*. This is a subclass of ordered attribute grammars for which it is simple to implement effective evaluators.

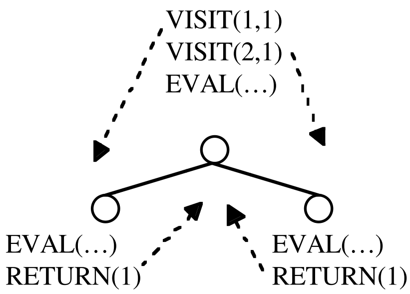


Figure 4.5: Simple example of visit sequences.

In [Kas80] it is shown how these visit sequences can be implemented using recursive procedures, a stack automaton or a finite automaton.

Object Oriented AG

So far the attribute grammars have been based on standard context-free grammars. We now turn to an attribute grammar based on object-oriented context-free grammars (OO CFG). We add attributes and equations to the OO CFG. Both attributes and equations are inherited by all subclasses. A definition of an equation overrides corresponding definitions made in superclasses.

Two different types of classes are used: *alternation*- and *construction*-node classes. Alternation classes are used to model abstract language concepts and construction classes are used for concrete language constructs. Figure 4.6 shows a small class hierarchy containing two types of statements, construction classes WhileStmt and AssignStmt, both subclasses to the alternation class Stmt.

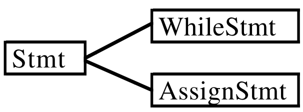


Figure 4.6: Part of class hierarchy for statement.

The alternation classes are never instantiated. There is no such thing as just a "statement", in any program the actual statements are either while- or assign-statements, i.e., instances of WhileStmt or AssignStmt construction classes.

Note that we now have two different forms of inheritance. First we have inheritance in attribute grammars sense, i.e., inherited attributes and second we have inheritance in the object oriented sense, i.e., subclasses inherits from super classes. We will use the term OO-inherit to mean the latter form of inheritance.

OOSL

OOSL[15] is a specification language introduced in [Hed92] and is used as a tool for describing object-oriented attribute grammars, Door attribute grammars (introduced in the next chapter), and incremental attribute evaluation for these grammars. In the later chapters of this document we will make extensive use of OOSL. For a complete description of OOSL see chapters 5, 6, and 8 in [Hed92].

Limitations of AG Specifications

The attribute grammars presented in this chapter have several limitations.[16] The main ones are:

- Attribute values can only be regular values, i.e., cannot be references. This means that to be able to model for instance the concept of a local block in a programming language an environment attribute containing the symbol table has to be distributed to all syntax nodes, instead of having one symbol table and letting each node have a reference attribute pointing to this symbol table.

- Semantic functions always define a whole attribute and the attribute is defined in terms of whole attributes. This makes it impossible to model a declared identifier by one object consisting of type and name information and let some syntax node use only the type information.

- Equations can only use attributes belonging to the nodes of a single production. This makes it impossible to connect a name application with its corresponding name declaration situated far apart in the syntax tree.

In the next chapter we present *Door attribute grammar*, an extended attribute grammar that solves these problems.

---

[15]OOSL stands for Object-Oriented Specification Language.

[16]Discussion from [Hed92].

5 Door Attribute Grammars

Door attribute grammars, introduced in [Hed92], is an extension to standard attribute grammars. It extends the syntax tree by adding *door objects* and *semantic objects*. *Reference attributes* are allowed which can lead to *non local dependencies*. *Collection-valued object* are introduced to handle structured values. All this is briefly introduced below. For a complete coverage of Door attribute grammars the reader is referred to [Hed92], [Hed93], [Hed94], and [KLMM93].

EST an Extended AST

In Door attribute grammars the syntax tree is seen as a tree consisting of instances of classes. Besides the syntax node classes introduced in object oriented AG's we here add two more types of classes. These are:

- *Semantic classes*: instances of semantic classes, semantic objects, are used for representing static semantic structures such as symbol tables and visibility graphs (see chapter 8).

- *Door classes*: instances of door classes, door objects, are used as an interface between syntax nodes and semantic objects.

Both semantic objects and door objects are defined as *part-objects* referenced by its owner by a *static reference*. By static reference we mean a reference that only can denote one object, meaning that once assigned a value the reference cannot change. A part-object is an object pointed to by a static reference, i.e., an object that is a permanent part of the declaring object (see [MMN93]). This is declared in OOSL in the following way:

```
C: class { … };

D: door { … };
```

```
E: class
{  rC: object C;
   rD: doorobject D;
};
```

In class *E* there are two part objects, a *C* (semantic) part object, *rC*, and a *D* (door) part object, *rD*. These object will be automatically created when *E* is created. *E* is said to *own* the part objects. The references *rC* and *rD* can never be changed.

An *extended syntax tree* (*EST*) is a syntax tree with the above extensions. See figure 5.1.

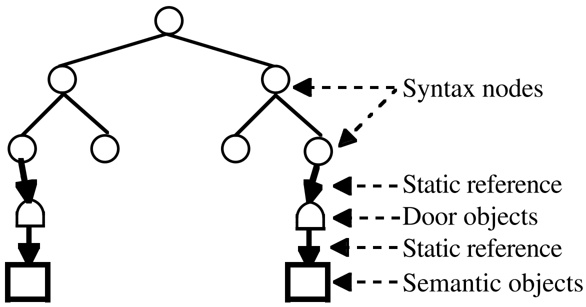


Figure 5.1: Extended syntax tree.

Door objects are owned by syntax nodes. Semantic objects can be owned by door objects, by other semantic objects or they can be defined globally (not owned by any other object).

As in standard attribute grammars the nodes and objects have attributes and equations. The following rules apply: (A local attribute is an attribute that only can be used in the node itself.)

- *Syntax nodes* have inherited, synthesized and local attributes, and equations defining the local and synthesized attributes for the node and all inherited attributes of its son nodes and door objects.
- *Door objects* have, as syntax nodes, inherited, synthesized and local attributes, and equations defining the local and synthesized attributes for the door object and all local attributes of its semantic part objects.
- *Semantic objects* have only local attributes and no equations. Attributes are defined by the owning door object.

Reference Attributes

One of the advantages with Door attribute grammars is that *reference attribute*s are allowed. This is declared in the following way:

```
E: class
{ rC: ref C;
};
```

Here *rC* is a reference to an object of type *C* (or a subclass to *C*). A reference may denote syntax node objects, door objects and semantic objects. Using reference attributes we can access information in any part of the syntax tree. This leads to non-local dependencies among the attributes. See figure 5.2.

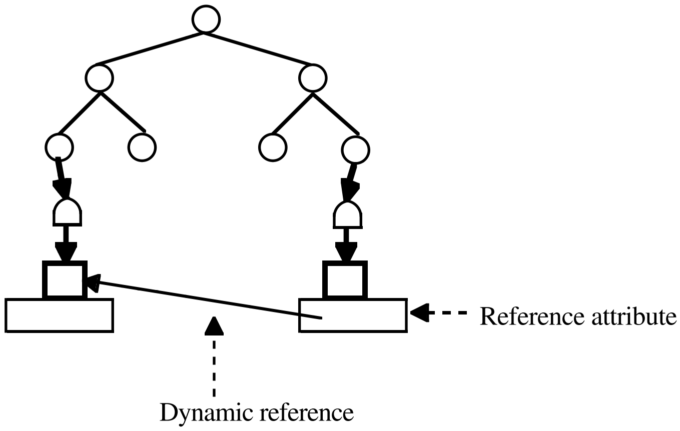


Figure 5.2: Reference attribute.

In principle, non-local access could be used in all kinds of nodes, but to simplify the attribute evaluation we make two restrictions:

1. The use of non local access is only allowed in door objects and
2. only attributes in semantic or door objects may be accessed non-locally.

This means that the doors are used to encapsulate all the non-local dependencies in the tree and that standard methods for attribute evaluation can be used for the syntax nodes.

We have three different kinds of references in the syntax tree. First, all syntax nodes are connected using *son-references*, references created and modified as a result of syntactic changes in the tree. Next, we have the *static references* (or *owner references*). These are references between syntax nodes and door objects, door objects and semantic objects or between semantic objects. They are never changed. Last, we have the *dynamic references* introduced by the reference attributes. These are references between door objects or between door objects and semantic objects and are changed as a result of attribute evaluation.

Collections and Conditions

The last extension made in Door attribute grammars is the addition of *conditions* and *collections*. These are used for handling *collection-valued* attributes. This is an attribute constructed out of several different attributes. A common example of this is a symbol table where many declaration nodes contribute to the contents. In standard attribute grammars, collection-valued attributes lead to complex specifications containing many attributes and equations used for gathering and propagating the attributes. This problem can be simply solved in Door attribute grammars by using a condition that defines an attribute as member of a collection.

A collection is a semantic object containing a number of member elements, e.g., lists, sets or bags. A condition is a boolean expression stating an object as a member in a collection. In a consistently attributed syntax tree all conditions must be true.

Consider the following symbol table example:[17]

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

Entry: class { … };

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

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);
};
```

[17]Example from [Hed92].

List is a general class used for storing objects. The function *contains(x)* returns true if *x* is a member of the list. A class *SymbolTable* is used to model the symbol table. It contains a collection, *entries*, which stores *Entry* objects by use of the *List* class.

The door class *D1* owns a *SymbolTable* object and propagates a reference to the symbol table using the synthesized reference attribute *tbl*. The door class *D2* inherits a reference to the symbol table and defines its part-object, *myEntry*, as member of the collection *entries* in the symbol table by means of a condition, *reg*. Different *D2* objects can define members in the collection. Note that these *D2* objects can be situated far from the *D1* object in the syntax tree.

Terms

Some of the terms used in Door attribute grammars are:[18]

- *Well-formed condition* – a condition *c* in a door *d* is said to be well-formed if and only if:
  - The condition is a boolean expression from which, given an otherwise consistent attribution, it follows that an object *x* is member of each of the collection objects in a (possibly empty) set *S*, and where
    - *x* is either *d* or a (direct or indirect) part object of *d*.
    - None of the collection objects in *S* are (direct or indirect) part objects of *d*.
    - No other membership follows from the expression.
  - There is no other condition *c'* in *d* stating the membership of *x'* as a member of each of the collections in *S'*, where $x' = x$ and $S \cap S' \neq \emptyset$.

This means that if a condition is well-formed then the condition can only state membership for the door itself or for direct or indirect part objects owned by the door, and it cannot state membership in collection objects owned by the door. In addition to this there is no other condition stating membership for the same object ($x = x'$) in a set of collection objects (*S*') that is not disjoint from *S*.

- *Well-formed Door AG* – a Door AG is well-formed if and only if:
  - The root node class has no inherited attributes.
  - Each node class has equations defining the synthesized and local attributes of the node itself and equations defining the inherited attributes of its son nodes and its door objects.

[18]From [ISAh92].

– Each door class has equations defining the synthesized and local attributes of the door itself and equations defining the local attributes of all its semantic part objects (owned directly or indirectly).
– There are no other equations.
– All conditions are well formed.

- *Consistent attribute* – an attribute is said to be consistent if the equation defining the attribute is fulfilled.

- *Consistent condition* – a condition is said to be consistent if the condition expression is true.

- *Consistent tree* – a syntax tree described by a Door AG is said to be consistent if and only if: all attributes and conditions are consistent; no collection contains members that do not follow from any condition.

- *Fix attribute* – an attribute is said to be fix if its value is not affected by any subtree replacement. In OOSL the keyword **fix** is used to declare an attribute as fix. This is used to simplify the implementation of the attribute evaluator.

Evaluation

One of the main goals of Door attribute grammars is that it should be "reasonably simple to develop efficient incremental attribute evaluators for a given specification"[19].

Main Grammar and Door Package

First of all the Door attribute grammar specification is divided into two parts (figure 5.3):

- *Main grammar* – containing syntax node classes and constant semantic objects. The evaluator for this part can be implemented automatically using standard methods.

- *Door package* – containing door classes, semantic classes, and constant semantic objects. This part requires manual implementation of the evaluator.

The main grammar contains reference attributes but it does not contain any non-local access and can thus be handled as a standard attribute grammar. From the

[19][Hed92] page 53.

main grammar point of view the doors are treated as syntax nodes. This means that the evaluator for the main grammar can be automatically implemented using methods for standard attribute grammars.

All the non-local dependencies and all collection objects are isolated in the door package. This makes the implementation of the evaluator for the door package a bit more difficult. In [Hed92] a manual but systematic method for this implementation is developed. The door package can be seen as modelling different semantic concepts used in the language. These concepts are often the same in different languages. This means that one door package can be developed and used for many different languages.

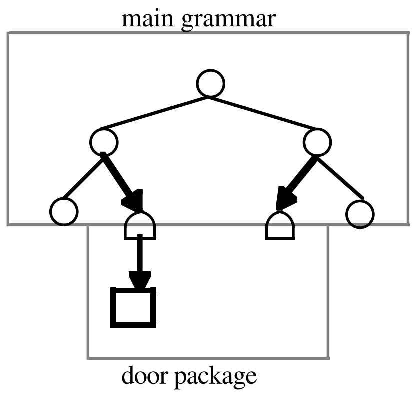


Figure 5.3: A Door AG is divided into a main grammar and a door package.

This gives us the formal definition of Door attribute grammars:[20]

Definition Door attribute grammars
A Door attribute grammar is a quadruple $G = (N, D, S, C)$ where N is a set of syntax node classes, D is a set of door classes, S is a set of semantic classes, and C is a set of constant semantic objects.
The grammar can be split into a main grammar, $G_M = (N, C_M)$, and a door package, $G_D = (D, S, C_D)$, and where $C = C_M \cup C_D$. •

Techniques

The evaluator for Door attribute grammars is based on *visit sequences*. The evaluation starts by calling a visit procedure in a syntax node (at the root node in the case of exhaustive evaluation or in the changed syntax node in the case of incremental evaluation) and spreads to other syntax nodes and door objects by calling

[20]From [Hed92] page 116.

visit procedures in these nodes. There are several different visit procedures, these are:

procedure	in	called from	purpose
exhDeEvalVisit	door	owning syntax node	De-evaluates all conditions.
exhVisit	syntax node	father syntax node	Evaluates all equations.
exhEvalVisit	door	owning syntax node	Evaluates all equations and conditions.
incDoorVisit(*d*)	syntax node	owned door (*d*)	Re-evaluates equations depending on the synthesized attributes of door *d*.
incSonVisit(*s*)	syntax node	son (*s*)	Re-evaluates equations depending on the synthesized attributes of son node *s*.
incFatherVisit	syntax node	father	Re-evaluates equations depending on the inherited attributes in the syntax node.
incOwnerVisit	door	owning syntax node	Re-evaluates equations and conditions depending on the inherited attributes in the door.
deeval$_L$	door (*d*)	A door which *d* is non-locally dependent on.	De-evaluate equations and conditions according to the non-local dependency labelled L.
eval$_L$	door (*d*)	A door which *d* is non-locally dependent on.	Evaluate equations and conditions according to the non-local dependency labelled L.

We will use a *1-visit* evaluator for the main grammar. Because of the extended expressive power of Door attribute grammars a 1-visit evaluator is sufficient for all the language constructs we need (i.e., classes, reference, types and procedures).

Implementation

Implementing the incremental attribute evaluator means constructing and adding visit procedures to all syntax node classes and to all door classes. These visit procedures are constructed out of the dependency graph for the Door attribute grammar. As mentioned above this can be done automatically for the main grammar but for the door package manual intervention is needed. This includes analysing the non-local dependencies and extending the dependency graph for the door package. Using this extended dependency graph the implementation of visit procedures for the door package is straight-forward. This is covered in detail in chapter 9 and 10 in [Hed92].

Figure 5.4 shows the resulting layout of a language specification and the implementation of the corresponding attribute evaluator. The layout is split into two parts: the *specification* and the *implementation*. The specification part contains the specifications of all syntax node classes (in the Main Grammar part) and the specification for all door classes and semantic classes (in the Door Package part). The door package interface is the part of a door specification accessed directly by the main grammar. This interface contains the inherited and the synthesized attributes of all doors.

The implementation part contains all visit procedures. In addition to this it also contains dependency attributes, dependency functions, and dependency procedures added to the door package as a result of the analysis of the non-local dependencies.

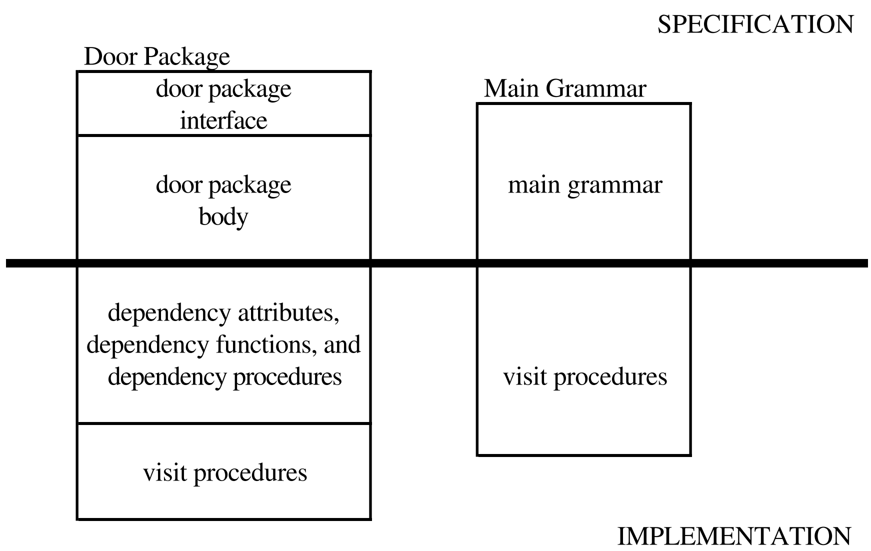


Figure 5.4: Layout of specification and implementation of main grammar and door package.

In the implementation a mix of demand and data attributes are used. The table below summarises the default implementation techniques:

	inherited	**synthesized**	**local**
syntax nodes	demand	demand	demand
door objects	demand	demand	data
semantic objects	–	–	data

Advantages of Door Attribute Grammars

Using the techniques explained above some of the limitations of standard attribute grammars can be relaxed. For instance using door attribute grammars it is possible to specify the declaration and application of names occurring in any order. This can be done without having to pass large environment attributes in the syntax tree, and the resulting grammar can use a simple 1-visit evaluation technique. (See chapter 8 and [Hed92] for more information.)

Another advantage of Door attribute grammars that has been shown is that it is possible to specify grammars that lead to better incremental behaviour for the evaluator of the specified language. The size of the set of attributes changed due to a change in the syntax tree are often significant smaller than it would be for the corresponding specification using standard attribute grammars.

The rest of this document is about different applications using Door attribute grammars. Amongst others it contains numerous examples of specification and implementation of different language constructs.

PART 2 ARCHITECTURE

In this part we will introduce a view on language based tools, such as compilers, editors, and pretty printers, as metaprograms, i.e. programs that manipulate other programs. We will explore the properties and benefits of such a view. We will also describe our own metasystem, *Embla*, which is a metasystem for manipulating extended syntax trees. The goal of Embla is to provide support for integration with existing tools and ease development of new ones. The Embla system is a BETA fragment, i.e. a separately compiled unit. A tool using the Embla system, includes the fragment, and is then compiled into a program.

In order to test and explore the metasystem, we have specified a programming language that contains a number of semantically difficult constructs. We call this language *Ask*.

We have also implemented a syntax tree editor, *WMEdit*. The editor is a tool that makes heavy use of the metasystem and basically allows visualisation and manipulation of the extended syntax tree that Embla upholds. It is not our goal that *WMEdit* should be a complete or even useful editor, it is merely a tool that demonstrates the metasystem. As to the integration of existing tools, research has been done to integrate *Sif*, the syntax directed editor of the Mjølner BETA system, with Embla and has up until this point shown promise.

We have followed the tradition of the Mjølner project in picking names from Nordic mythology. Ask and Embla are the names of the first humans, and will thus symbol the authors first exploration in the area of metaprograms, grammar specification, and incremental compilers. We consider WMEdit not to be a full member of this project, hence the name is fetched from the authors initials, as has been the case in earlier projects.

This part is structured as follows: Chapter 6 describes a way to implement a door attribute grammar. In chapter 7 we introduce the Embla metaprogramming system and in chapter 8 the first small version of the Ask-language in presented.

6 Implementation of a Door AG

In the previous chapter we introduced Door Attribute grammars, what they are and how to use them. In this section we will discuss what we can use them for. The main applications of all grammars are without doubt to specify some language that has a textual representation, such as programming languages. Research has been done in the area of using attribute grammars for other purposes. For example, [Hud87] describes how to use a system of attributes to maintain a user interface. As the author points out, such interface is “self-updating”, since the attributes are updated through evaluation of its dependencies.

As stated in chapter 5 a door attribute grammar specification contains three different kinds of classes. Syntactic node classes that describe the syntax of a language, semantic classes that describe the semantics of a language and door classes to provide an interface between the syntax nodes and the semantic objects. The three kinds of classes can be divided into three *packages*. In the Embla system each package is a BETA fragment. This organisation of the different classes are not the only one imaginable, but one we have found useful.

Figure 6.1: The three packages of a Door AG implementation.

Considering these three packages, one can draw some conclusions on their contents for a wide range of programming languages. The door and semantic packages are more general then the syntactic package since many languages share semantic properties, such as declarations of variables. Developing a language-dependent tool, such as a compiler or parser, is often a tedious and error prone process. From a developers point of view it is then a great benefit to be able to “reuse” previous materials.

The viewpoints presented in this chapter are based upon our experience from building the Embla system.

Since many languages share properties in the different packages, we will divide the three packages into two parts, the *independent* and the *specific*.

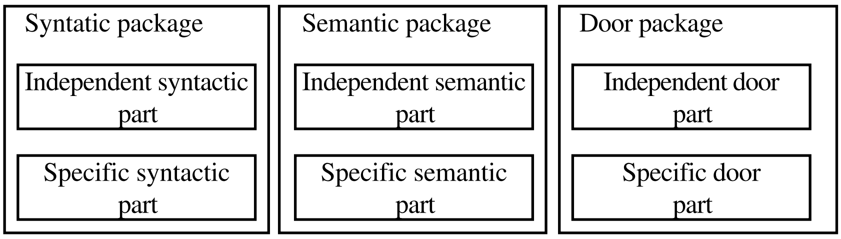


Figure 6.2: The independent and specific parts of the packages.

The independent part of a package contains the essence of some construct found in many programming languages. This part is language independent, while the specific part represents an actual language. The main benefit of this approach is reuse, which means faster development and safer software when applied correctly. But other benefits are possible. Suppose that the independent part of the packages contains the mechanisms for name analysis and scope rules. Then a tool for showing which variables are accessible at a particular point in the program, can be implemented. This tool will be language independent and can be used for any language that uses this mechanism at the specific level.

The specific part of a package can use the classes found in the independent part either as is, by subclassing, or by creating a new class which combines several independent classes. The independent classes that are not useful for a particular language can be left aside.

For the approach of independent and specific parts to work, independent classes that fit many languages must exist. It is our belief that many such classes can be designed and we will look into a few examples below. It would be desirable to find a set of independent classes that could cover most known programming languages. Such research is however outside the scope of this work.

Syntactic Package

The syntactic package contains syntax node classes that correspond to productions in the grammar. In chapter 3 we introduced the nonterminals <DeclType>, <Stmt>, <Op>, etc. these would all be classes in this package. The syntactic package corresponds to the main-grammar part found in [Hed92].

The syntactic package is tightly connected to the language that the grammar represents, but common elements can be found. The BETA metaprogramming environment has four such nodes, *NameDecl*, *NameAppl*, *String* and *Const*. The Embla system is more limited in terms of syntax and only contains one, namely

ID. In BETA, NameDecl and NameAppl model the declaration and usage of an identifier. In Embla, ID is used in a similar way.

Semantic Package

The semantic package contains the classes used for modelling semantic objects. A semantic object is a construct introduced by a tool, like a compiler or analyser, during the process of evaluating the program. A semantic node differs from a syntactic in the sense that it does not occur in the user's source code. An example of a semantic object is a symbol table.

What semantic objects are useful depends on what semantic concepts the language supports. When constructing a language new semantic classes will, most certainly, have to be added. However, one can introduce a set of independent semantic classes that represents orthogonal semantic concepts.

The relationship between the door package and the semantic package is strong, since the door is a mechanism for providing an interface between syntax nodes and semantic objects. So any change to the semantic package will probably call for changes to the door package. Examples of this will be shown in the next part where we describe the additions and modifications we have done to the existing door and semantic package.

By introducing an independent class for a semantic concept like types, we can construct doors that are independent on the actual type of the semantic node. A symbol table entry, for example, might consist of an identifier and a type. The identifier node is found in the syntactic package described above. The entry provides a connection between the identifier and the type, the actual type is irrelevant with regards to the symbol table. If we later on introduce a new type, like for instance the CARDINAL type of MODULA-2, no changes have to be made to the symbol table.

There is a connection between the different tasks that normally falls into the domain of semantic analysis and the semantic classes. Among these tasks the most prominent are name analysis and type checking.

Although name analysis differs among programming languages, the fundamental concept is the same. All identifiers are stored in some kind of symbol table. Applications or usages of identifiers are checked against one or more symbol tables. The scope rules, i.e. the way to find the reachable symbol tables from a certain point in the program, can be modelled by a semantic object that ties symbol tables together.

Door Package

A door encapsulates a semantic concept, such as identifier scopes and type matching. An independent door should then try to find the essence of the problem it was created for, so that it may solve similar problems.

An independent door has knowledge of the contents of the independent parts of the syntactic and semantic packages. From these syntax nodes and semantic objects independent door objects can be constructed. Suppose that the syntactic package contains an *ID* node for identifiers and that the semantic package contains objects for symbol tables and types. Then the *DeclDoor*, found in [Hed92] and in the independent package for the Ask language[21], can be specified.

An example of usages of doors is the *TypeDecl* syntax node found in the specific part of the syntactic package for the Ask language. The *TypeDecl* node successfully assembles both independent and specific doors of the Ask door package. The specification of the *TypeDecl* node can be found in chapter 12, “User defined types”.

[21] See chapter 8, “The Ask Language”.

7 The Embla Metaprogramming System

Overview

When developing a metaprogramming environment, one can choose from different architectures. An overview of grammar based architectures is given in [KLMM93]. For the Embla system we have chosen the so called *common representation architecture*. The common representation is, as in the Mjølner environments, the view on a program as an abstract syntax tree.

The Embla system is a metaprogramming system for manipulation of extended syntax trees. It also provides incremental evaluation of the tree. The metasystem is implemented as a BETA fragment which provides a programmer interface for tool developers. This means that the tool and the Embla system is compiled into a complete program. Other approaches can be considered, such as reading grammar dependent information from file at initialisation. This might, however, lead to slower execution.

To generate source code from the grammar and then compile the tool, will inevitable lead to slow turnaround time during development. The Embla system is compilable, like the Mjølner BETA metaprogramming system, while the Mjølner Orm system has a more sophisticated mechanism, where the grammar development phase is integrated into the environment. The Mjølner Orm environment is a grammar interpreting system where the grammar is read in an intermediate form during initialisation.

The tool communicates with the Embla environment through Embla's built-in class hierarchy, its tree editing methods, and the three packages.

The Embla system is grammar independent in the sense that it can handle syntax trees and packages for any grammar. The Embla has, however, a small grammar dependent part that will be described later. Some tools can be grammar independent if they view the syntax tree at an abstract level, i.e. do not access the packages. Other tools can access the independent parts of the three packages and thus be *package* dependent, but not language dependent. If some sort of class concept is available in the any of the independent parts, then a class browser can be considered package dependent. Other tools may access the specific parts of some particular package and thus be fully language dependent.

The Runtime Model

Since the Embla system is compiled as part of a tool, the tool is the controlling part. The tool uses the Embla system through methods and access to two syntax nodes in the tree, *theRoot* and *theFocus*. The focus node is the current focus of attention, for example the editing point of a syntax directed editor. The Embla system upholds the syntax tree and keeps it evaluated at all times.

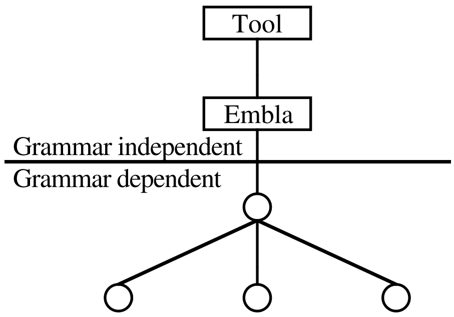


Figure 7.1: The relationship between a tool and the Embla system.

The Implementation Architecture

Figure 7.2 shows the organisation of the source code of the Embla system. The Embla pattern encloses all the different parts for manipulating the syntax tree. The BETA slots for the three packages can hold one or more independent and/or specific part of any package. More information on how to assemble a tool is given in the section "The assembly of a tool". The tree editing methods and the abstract class hierarchy are described in the following sections.

Interface of the Embla System

Tree Editing Methods of Embla

Embla provides five basic tree operations. These operations are used by the tool to manipulate the syntax tree. One operation for replacing ordinary trees, three operations for administration of lists, and one for the creation of new trees.

- *ReplaceSubTree* replaces a subtree with another.
- *ReplaceInList* replaces a subtree in a list, i.e. the son of a LISTNODE.
- *InsertInList* insert a subtree in a list.
- *DeleteInList* deletes a subtree in a list.
- *NewTree* creates a new, empty tree.

All these operations perform incremental evaluation and devaluation of the syntax tree. So the tree is consistent after performing any of these operations. Although it is possible for the tool to manipulate the tree directly, any such operation will probably result in an inconsistent tree.

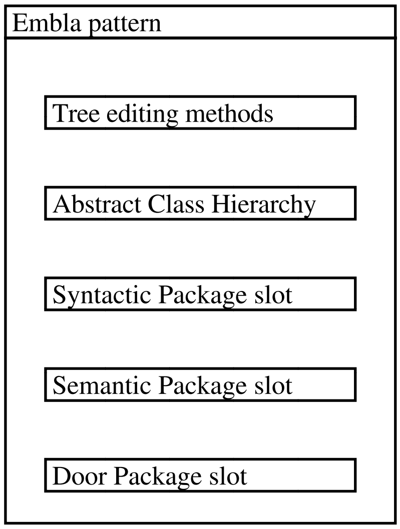


Figure 7.2: The organisation of the Embla system.

Class hierarchy

It is possible for Embla to manipulate the tree for any grammar, since all grammar dependent classes are specialisations of a set of abstract classes. These classes represent a classification of the different nodes that exist in the decorated tree.

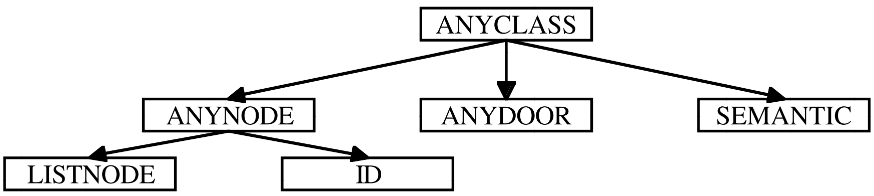


Figure 7.3: The classification and inheritance of the Embla classes.

The classes allow a more abstract view on the syntax tree and are used by the tools and Embla to perform tree operations. Here is an example of tree navigation taken from WMEdit:

```
Up :
 (#
 do  (if theFocus[] <> NONE // TRUE then
       (if theFocus.father[] <> NONE // TRUE then
         theFocus.father[] -> SetFocus
       if);
     if);
 #)
```

The current editing focus is moved around in the tree as response of user actions. The traversal of the tree is grammar independent, since it does not access the packages.

Introduction to ANYCLASS

The abstract classes and their methods are the most important part of the metaprogramming system. Many of the methods are used by Embla or the syntax nodes themselves, but are available for tool developers as well. As shown in figure 7.3, ANYCLASS is the base class of all classes in the Embla system. ANYCLASS contains no methods.

Introduction to ANYDOOR

The abstract class describing a door contains virtual evaluation methods. Through ANYDOOR, Embla can evaluate doors in the independent and specific door package.

- *exhEvalVisit* performs exhaustive evaluation of the door.
- *exhDeEvalVisit* performs exhaustive de-evaluation of the door.
- *incOwnerVisit* performs incremental evaluation of the door, when demanded by the owning syntax node.

Introduction to ANYNODE

ANYNODE is the most complex class in the hierarchy. All syntax nodes will be subclasses of ANYNODE. Even though an ANYNODE object is a syntax node, some methods view the ANYNODE object as the root of a subtree. These views are both correct and thus some methods of the ANYNODE class concern only the syntax node itself, while other concern the subtree rooted by the syntax node. The methods of this class can be divided into three categories; evaluation methods, son handling methods, and door methods.

- *exhVisit* performs exhaustive evaluation of the syntax node.
- *incFatherVisit* performs incremental evaluation, demanded by its father.
- *incSonVisit* performs incremental evaluation, demanded by one of its sons.
- *incDoorVisit* performs incremental evaluation, demanded by one of its doors.

- *copyInheritedAttributesFrom* copies data attributes from old to new subtrees.

- *replaceBy* replaces a son with another.
- *son* returns the son with a specified index.
- *putSon* replaces a son at a specified index.
- *noOfSons* returns the number of sons the node has.
- *Index* returns the index of a given son.
- *GetAllSons* collects all sons of the syntax node.
- *setAllFathers* sets the father of the subtree to a given syntax node.

- *allDoors* collects all doors in the subtree.
- *findDoors* collects all doors of the syntax node that satisfy a given condition.
- *findAllDoors* collects all doors in the subtree that satisfy a given condition.

Introduction to LISTNODE

A special case of ANYNODE is a node that handles a list of sons. LISTNODE is used for lists of syntax nodes, for example parameter list to procedures.

- *incBrotherVisit* performs incremental evaluation, demanded by brother.
- *evalInheritedAttributesOfSon* evaluates the inherited data attributes of a given son.
- *whatSon* returns the index of a given son.

Introduction to ID

ID is a class for manipulating identifiers and it belongs to the syntactic package. Through the ID node, the Embla system and the other packages can perform operations on identifiers. One such operation is name analysis.

Introduction to SEMANTIC

The class SEMANTIC is the super class of all semantic classes. At this stage it contains no methods or variables, since semantic objects differ greatly from each other. The SEMANTIC class can be used to distinguish between a semantic object an object of other kind. All classes in the semantic package is subclasses of SEMANTIC.

Grammar Dependent Parts of Embla

When creating a syntax tree, either from user action or reading a syntax tree from file, new nodes have to be created. The creation of nodes are grammar dependent. The Embla system and tools using it, may only need to access the tree at the abstract level, i.e. ANYNODE, ANYDOOR, etc. But a tree with only instances of these classes is without meaning.

To isolate these grammar dependent parts, the Embla system contains three methods.

- *CreateNode* creates an instance of a given class name.
- *ExpandNode* creates and builds a default subtree rooted in a given class name.
- *BuildInitialTree* builds the initial tree for a particular grammar.

The implementation of these methods could however be generated automatically from the grammar and put in a separate BETA fragment. The consequences of this is that a grammar independent tool such as WMEdit must be recompiled for each language.

The Assembly of a Tool

Figure 7.4 shows the source code structure of a complete tool. The tool assembles the different BETA fragments that makes up a complete system. The boxes representing the three packages symbolises the inclusion of one or more fragment containing independent or specific classes.

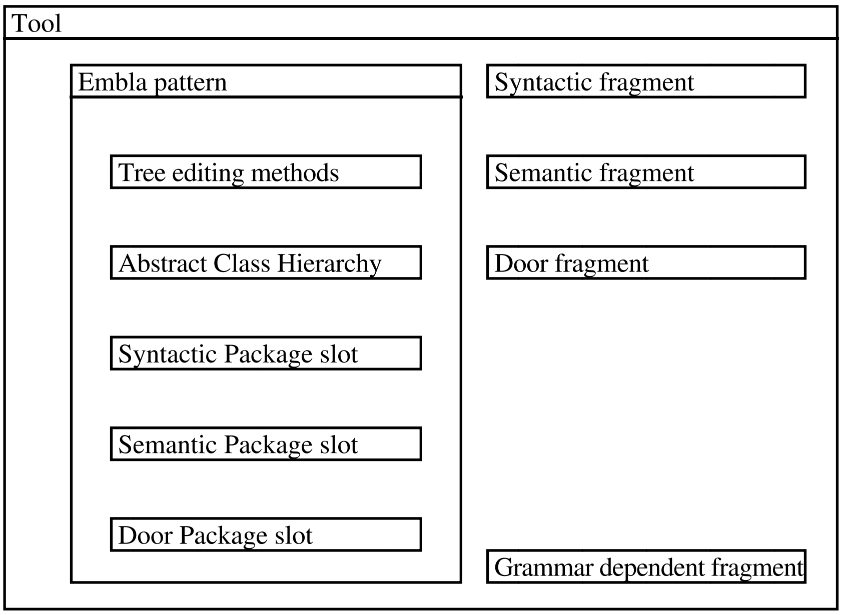


Figure 7.4: The parts of a complete tool.

The Ask Language

In this section we describe the foundations of the Ask-language. All constructs described here originate from a test system for the language introduced in [Hed92]. This test system, implemented in Simula, was rewritten and adapted to the BETA environment. We will use these language constructs in later additions to the Ask-language as described in part 3 of this document. For more details on the topics covered in this chapter see [Hed92] chapters 8 and 11.

Contents of the Syntax Package

This first version of the Ask-language is very small, supporting only a few language elements, these are:

- Two different declarations: variable-declaration and class-declaration.
- Two statements: block-statement and assign-statement.
- Two expressions: a use expression and a remote access expression.

The syntax of the declaration, statements, and expressions is straight-forward, resembling for instance Simula. The variable-declarations use the syntax node *VarDecl*. Three types of simple declarations are supported, integer, boolean, and references. For more information on declaration types see the section “Types”. The additional declaration supported is a class-declaration (using syntax node *ClassDecl*).

Block-statement is, as the name suggests, used to introduce a new block in the program. The syntax node is *BlockStmt*. Assign-statement is used for variable assignments. The two expressions, *Use* and *RemoteAccess*, are used for modelling the use of simple variables and reference variables. The block-statement is also used as the first statement in a program. Consider the small program and the corresponding syntax tree in figure 8.1.

```
BEGIN
  INTEGER x;
  INTEGER y;
  X := Y;
END;
```

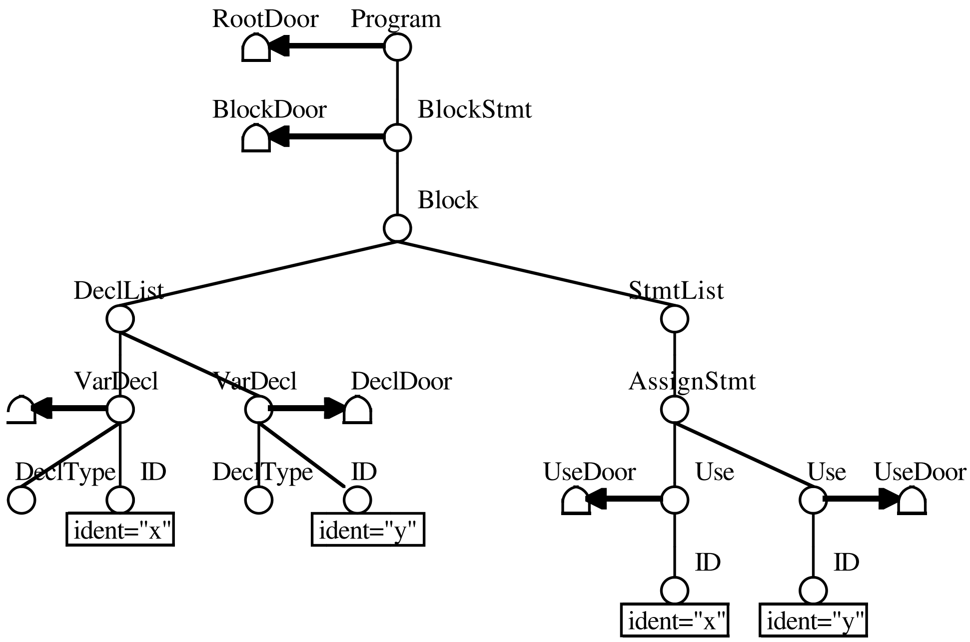


Figure 8.1: Syntax tree for a simple program.

As seen in figure 8.1 there are additional constructs not mentioned above, these are nodes such as *Program*, *Block*, *StmtList*, and *DeclList*. Program is the root of the syntax tree. The syntax class Block denotes a sub block, i.e., a part of the program containing local declarations and statements. The classes DeclList and StmtList are lists of declaration and statements containing zero or more elements.

Some extra alternation classes are used to form the top of the classification hierarchy. In *Descendant* two attributes needed in all construction nodes are defined. These are *table* and *path*. The specifications for *Main*, *Root*, *Descendant*, and *Program* are shown below.

```
Main: alt {};

Root: alt Main {};
```

```
Descendant: alt Main {
  inh path: ref SearchPath fix;
  inh table: ref SymbolTable fix;

  eq son Descendant.path :- path;
  eq son Descendant.table :- table;
};

Program: cons Root ( blkStmt: ref BlockStmt ) {
  rDoor: doorobject RootDoor;

  eq blkStmt.path :- rDoor.rootPath;
};
```

Contents of the Semantic Package

Three semantic concepts are modelled in this first version of the Ask-language: search paths, symbol tables, and a type hierarchy.

Visibility Graphs

Before we go into details of the contents of the semantic package we will introduce *visibility graphs*. These will be used to model the search rules for programming languages in this and later chapters.[22]

A visibility graph is a directed acyclic graph ($G = (V, E)$) where the vertices, V, are divided into three disjoint sets. We have:

- Each local symbol table is represented with a table vertex, t, with no outgoing edges.
- Each connection in the graph is represented with a path vertex, p. The path vertex has one or more outgoing edges. In our examples we will only use path vertices with two edges.
- A *Null* vertex with no outgoing edges is used to terminate searches.

To clarify this let us look at an example. Consider the following program structure where two classes B and C are defined so that C is a subclass of B. In class C we have two anonymous blocks.

```
BEGIN
    CLASS B;
    BEGIN
    END;
```

[22] Definition and example from [Hed92].

```
      B CLASS C
      BEGIN
          BEGIN
          END;

          BEGIN
            aVariable := 17
          END;
      END;
  END;
```

Figure 8.2 shows the corresponding visibility graph. This graph contains five symbol table vertices: one for each block and one for each class. A total of seven path vertices are used to connect the graph. Each of the anonymous blocks are modelled using a path vertex where the first edge points to a table vertex and the second edge points to the path vertex of the enclosing block or to a Null vertex if no enclosing block exists. These path vertices are labeled s_n standing for static path vertex. To model the search rules for the classes we use two path vertices. One for the static path as for anonymous blocks and one additional to model the class hierarchy. In the figure these are labeled p_n which stands for prefix path vertex. Null vertices are indicated using a diagonal line in the outgoing slot of the path vertex.

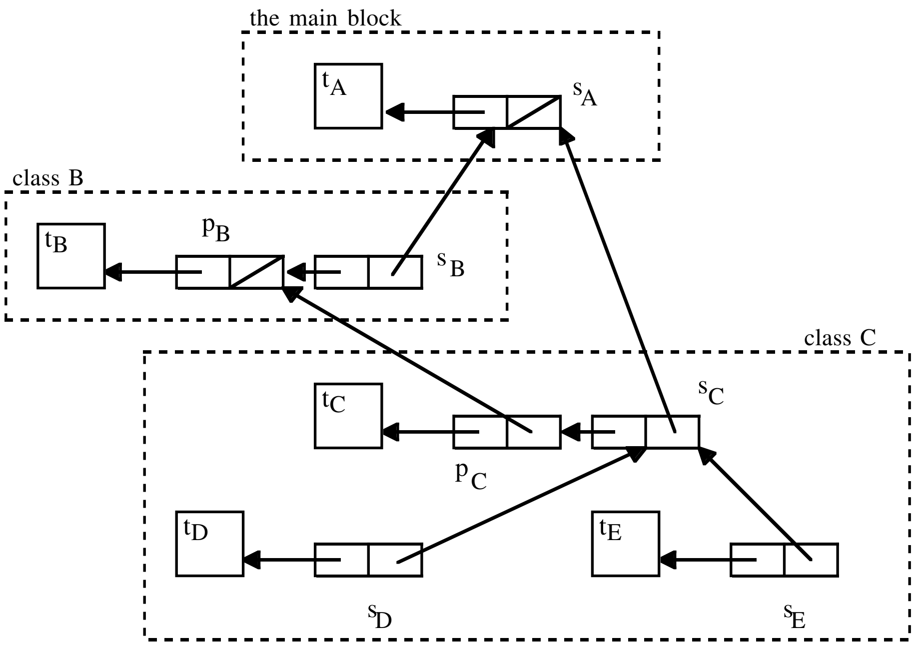


Figure 8.2: Visibility graph example.

A search in the graph is done in the following way. If we start at the assignment in the above program example (block E) we will start the search in path vertex

s_E. The two outgoing edges in a path vertex are ordered so that the left edge are examined first. This means that we will examine the vertices in the following order: s_E, t_E, s_C, p_C, t_C, p_B, t_B, s_A, and t_A. In other words: we will first search the local symbol table, then the symbol tables in all superclasses, and then the symbol tables in all enclosing blocks. If the enclosing block is a class we will again search the symbol tables for all superclasses of this class. Normally a search is terminated with the first match.

Paths

The first part of the semantic package, paths, is used to model the scope rules in the program, i.e., the visibility graph. The class *SearchPath* is used as an abstract superclass for all classes and objects in this package. The virtual function *lookup* returns a SymbolTable entry and is used in finding a specific entry in the symbol table. This function is implemented in all subclasses so that the visibility rules are enforced. To model the path vertices in the visibility graph the classes *TwoPath* and *SymbolTablePath* are used. TwoPath has two references to other path-objects and SymbolTablePath has one reference to a symbol table. *emptyPath* is a constant object used to denote an empty search path. One additional class will later be added to the paths-package, this is the *WatchPath* used in the With-statement introduced in chapter 10.

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

TwoPath: class SearchPath
{ loc first: ref SearchPath;
  loc second: ref SearchPath;
};

SymbolTablePath: class SearchPath
{ loc table: ref SymbolTable;
};

emptyPath: object SearchPath;
```

The scope rules in Ask are the same as in traditional ALGOL-like languages. All variables are visible in the block in which they are declared and in all blocks declared in this block. Variables declared in a class are visible in all subclasses. A search for a variable searches first the current class (if any), then all superclasses (if any), then the current block, and last all enclosing blocks (if any). As a simple example, consider the program and corresponding visibility graph in figure 8.3. Figure 8.4 shows the same visibility graph modelled with the paths-package. For example a search in the class A starts at the point marked with ① in figures 8.3 and 8.4.

```
BEGIN (* main *)

  CLASS A;
  BEGIN
        ①
  END;

END;
```

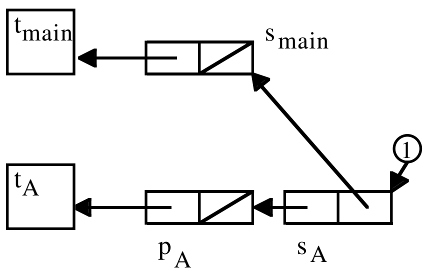


Figure 8.3: Visibility graph for a simple program.

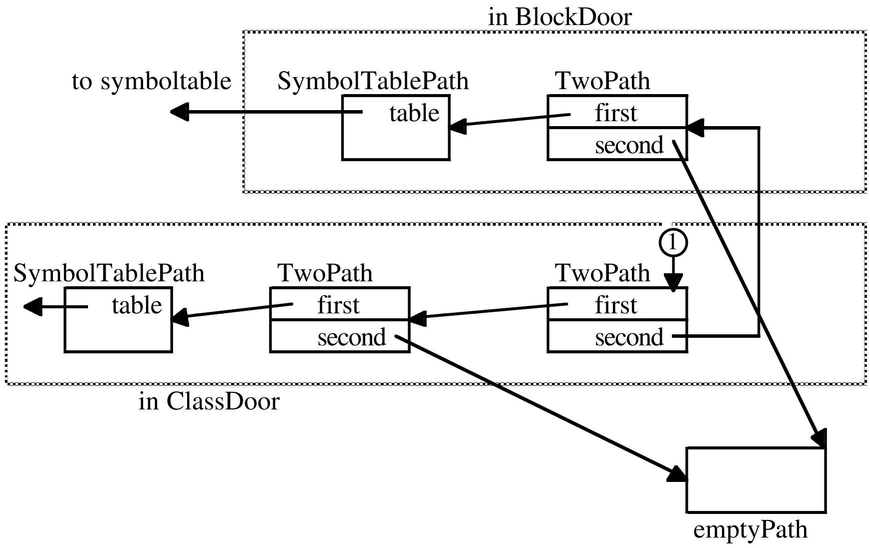


Figure 8.4: Corresponding path structure.

Symbol Tables

An important concept in (almost) all programming languages is to be able to store identifiers. This is done in Ask by the *SymbolTable* class. We use a collection part-object, *entries*, to store a list of *Entry* objects. Each entry has an identifier (a string denoting its name), and a reference attribute pointing to a type object. To find an entry in the symbol table the function *lookup* is used.

```
SymbolTable: class
{ collection entries: object List[Entry];
  lookup: func ref Entry(ident: string)
      (*may return NONE*)
};

Entry: class AbstractEntry
{ loc ident: string;
  loc tp: ref Type;
};
```

```
nullEntry: object AbstractEntry;
```

A reference attribute will be used to represent the binding of an identifier. This attribute will be a reference to an entry object if the identifier is declared. The nullEntry object is used to represent all identifiers not declared.

Types

In the entry object the attribute *tp* points to a type object. This object is used in modelling the type hierarchy. The class *Type* is the common superclass to all classes in the type hierarchy.

```
Type: class;
```

Two simple types exist in the Ask-language, integer (*intType*) and boolean (*boolType*). Besides this the object *unknownType* is used to denote a variable with unknown type.

```
unknownType: object Type;
intType: object Type;
boolType: object Type;
```

To model the class hierarchy in an Ask-program the classes *AbstractClassType*, *ClassType*, *objectClassType*, and *noClassType* are used. For each class defined in the program there exists one ClassType object. ClassType is a subclass of the abstract superclass AbstractClassType in which the functions *subclassOrEqual* , *getPrefixPath*, and *getRefType* are defined. Suppose A and B are two classes defined in Ask, then these functions will return the following:

- A.subclassOrEqual(B) is a boolean function that will return true if the class A is a subclass of class B or if A and B are the same class.
- A.getPrefixPath returns the prefix path (the path to the superclass) of class A.
- A.getRefType is used to get the corresponding reference type to class A.

objectClassType is used to model the most general class, i.e., the superclass of all classes. All references to NONE are modelled by the object noClassType. subclassOrEqual and getPrefixPath return emptyPath for the objectClassType and the noClassType. For technical reasons, noClassType is viewed as modelling a subclass of all Ask program classes, thus turning the hierarchy into a lattice. This simplifies type checking of NONE references as explained in section 3.5 in [Hed92].

```
AbstractClassType: class Type ( abstract *)
{ subclassOrEqual: func boolean (ct: ref
                                AbstractClassType);
  getPrefixPath: func ref SearchPath fix;
  getRefType: func ref AbstractRefType fix;
};
ClassType: class AbstractClassType;
objectClassType: object AbstractClassType;
noClassType: object AbstractClassType;
```

Figure 8.5 shows the class lattice for a simple Ask-program.

```
BEGIN

  CLASS A;
  BEGIN
  END;

  A CLASS B;
  BEGIN
  END;

  A CLASS C;
  BEGIN
  END;

END;
```

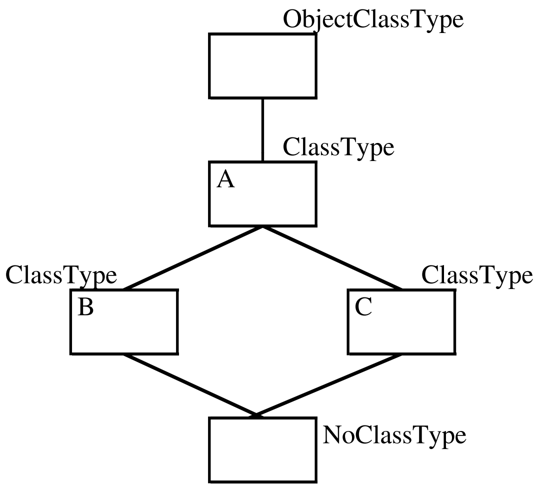


Figure 8.5: Class lattice for a simple Ask-program.

The last types in the Ask-language are reference types. These are handled similar to the class types. Each reference type in an Ask-program has a *RefType* object. *ObjectRefType* is the common “super reference” and *noRefType* models the type of NONE.

```
AbstractRefType: class Type (* abstract *)
{ getClassType: func ref AbstractClassType fix;
};
RefType: class AbstractRefType;
objectRefType: object AbstractRefType;
noRefType: object AbstractRefType;
```

The function getClassType in AbstractRefType is used to get the corresponding class type. All the above taken together results in the type class hierarchy shown in figure 8.6.

Contents of the Door Package

We now turn to the contents of the door-package. The first version of the package contains five doors, these are:

- *RootDoor* for the enclosing scope of the program.
- *BlockDoor* for block structuring.
- *DeclDoor* for name declarations.
- *UseDoor* for name applications.
- *ClassDoor* for class declarations.

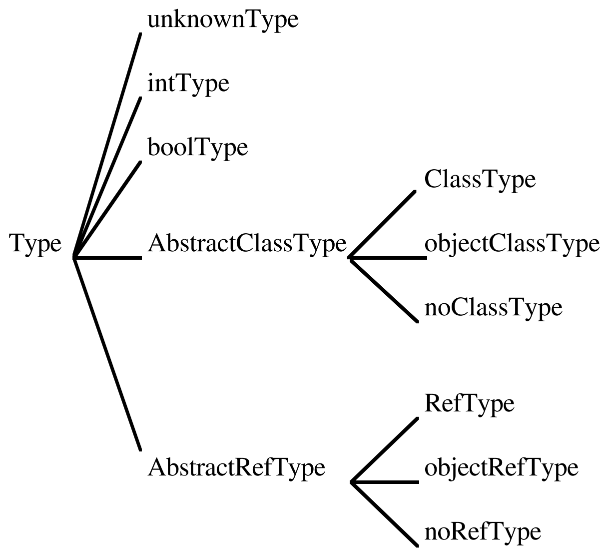


Figure 8.6 : Type class hierarchy.

For all doors we present the OOSL-specification of the interface and a figure showing the door with its attributes and part-objects. By the interface we mean those parts of a door specification accessed directly by the syntax package. The interface contains the inherited and the synthesized attribute of the door.

Figure 8.7 shows the graphical notation used throughout this report.[23]

[23] This is the same notation as used in [Hed92].

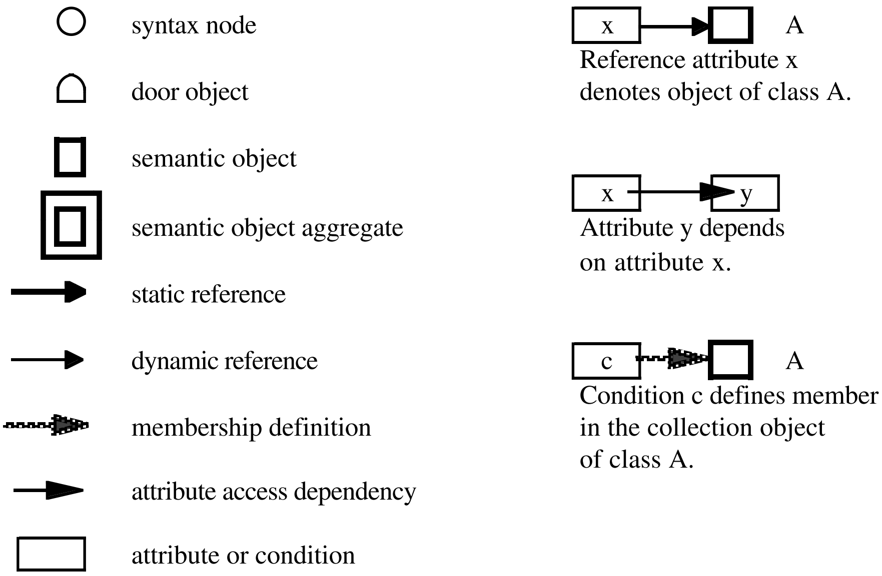


Figure 8.7 : Graphical notation used.

RootDoor

RootDoor represents the top of a program and is used for terminating searches of different kinds. It is a very small door, defining a synthesised attribute, *rootPath*, which denotes the semantic object, *emptyPath*. There is only one RootDoor object in a syntax tree.

```
RootDoor: door
{ syn rootPath: ref SearchPath fix;
};
```

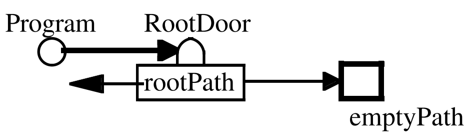


Figure 8.8: RootDoor.

BlockDoor

BlockDoor is used for modelling a local block. It maintains a symbol table for local declarations. A TwoPath object is used to model connection to local and enclosing scopes. There is one BlockDoor object for each block (each block-statement) in the program. The inherited attribute *encPath* is a reference to the enclosing block and the synthesized attribute *locPath* and *table* are references to the local path, i.e., the path that includes the local symbol table, and the symbol table respectively.

```
BlockDoor: door
{ inh encPath: ref SearchPath fix
  syn locPath: ref SearchPath fix;
  syn table: ref SymbolTable fix;
};
```

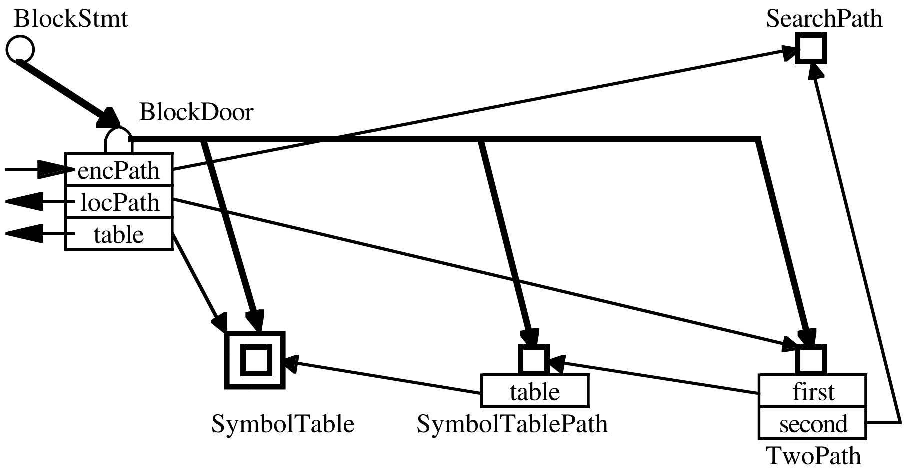


Figure 8.9: BlockDoor.

DeclDoor

A DeclDoor is used for declarations. It contains an Entry part-object, that will be registered in a symbol table. Since symbol tables are semantic objects, the syntax tree is not allowed to manipulate it directly. This manipulation is done through a DeclDoor. The attributes are *table*, *tp*, and *ident*, all three inherited. *table* is a reference to the used symbol table and *tp* and *ident* are the type and the identifier of the entry.

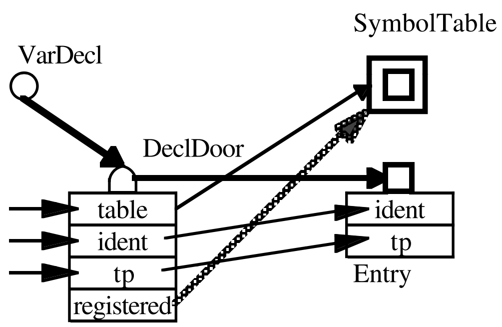


Figure 8.10: DeclDoor.

```
DeclDoor: door
{ inh table: ref SymbolTable fix
  inh tp: ref Type;
  inh ident: string;
};
```

The condition *registered* in figure 8.10 is used to state that the entry owned by this DeclDoor is a member in the symbol table referenced to by the table attribute. The syntax package is not allowed to access *registered* directly, so *registered* is not included in the interface specification for DeclDoor.

UseDoor

All usages of identifiers are modelled through a UseDoor. UseDoor inherits the attributes *path* and *ident*. path points to a SearchPath object in which the search for the identifier starts. Finding the identifier is done by using the function lookup in SearchPath. ident is a string denoting the name of the identifier. The synthesized attribute *tp* is a reference to the type-object of the entry and the synthesized attribute *declared* is true if the identifier is declared, i.e., the resulting entry from the lookup is not a nullEntry object.

```
UseDoor: door
{ inh path: ref SearchPath fix
  inh ident: string;
  syn tp: ref Type;
  syn declared: boolean;
};
```

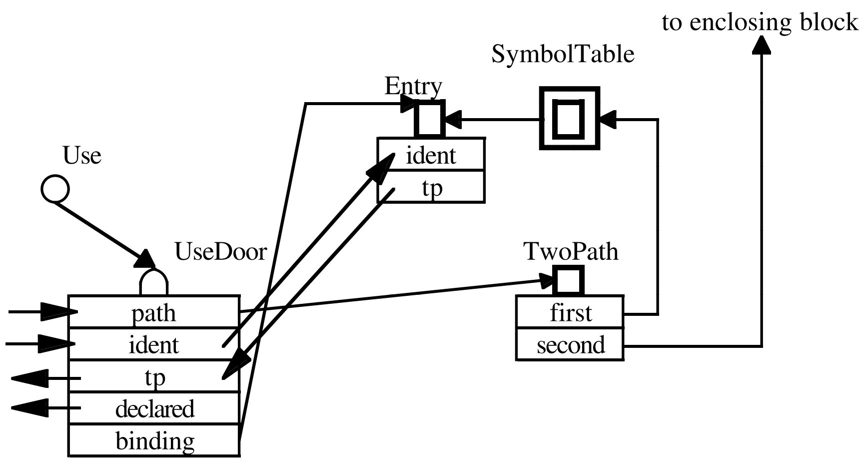


Figure 8.11: UseDoor.

One additional attribute, binding, is shown in figure 8.11. This is a local attribute accessible only from within UseDoor and used for retrieving the type of the identifier.

ClassDoor

ClassDoor is, as the name suggests, used for class declarations. This door is the most complex of the five. The door has to model the visibility rules for classes. This is done by using two SearchPath-objects (TwoPath): one for the prefix path and one for the static path. ClassDoor also contains a symbol table used to collect entries defined locally in the class. The TwoPath-object for the prefix path points to the local symbol table and to the prefix path of the superclass or to an emptyPath-object if there is no superclass. As for the BlockDoor the static path points to the local symbol table but here through the TwoPath of the prefix path, and then to the static path of the enclosing block. The attributes in ClassDoor are:

- *encPath* a reference to the static path of the enclosing block.
- *syntPrefixClass* a reference to the prefix path of the superclass.
- *locPath* pointing to the TwoPath-object of the static path.
- *table* pointing to the local symbol table.
- *classTp* pointing to the ClassType part-object and thus denoting the type of the class.
- *causesCycle* a boolean attribute that is true if this class causes a cycle in the visibility graph.

```
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;
};
```

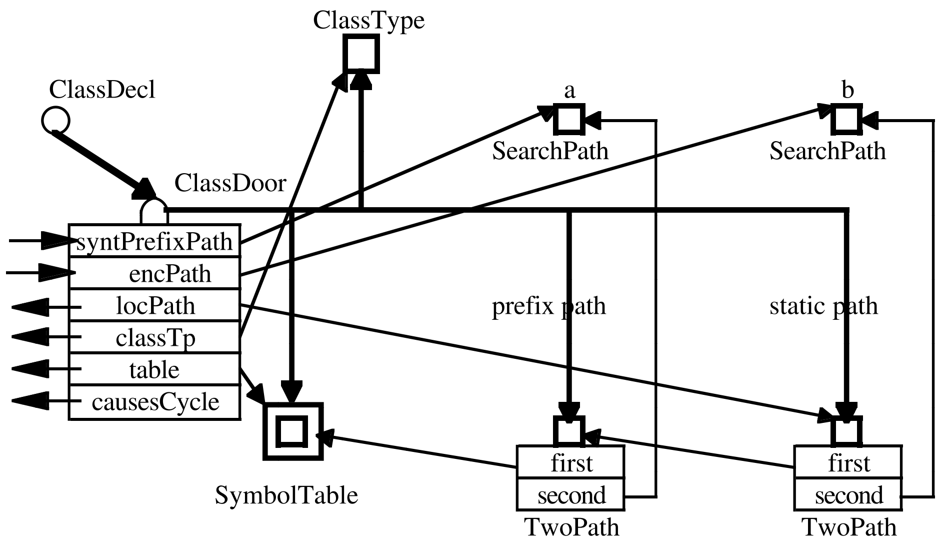


Figure 8.12: ClassDoor.

PART 3 CASE STUDIES

This, the last part of this report, covers our extensions to the Ask-language. The origin for the Ask-language is an adaptation of the language constructs introduced in [Hed92] as described in the chapter 8. The addition made here are if-, while-, and with-statements; procedures and parameters; and user defined types.

This part is structured as follows: In chapter 9 two simple extensions, an if- and a while-statement, are made. Chapter 10 introduces the with-statement and the InspectDoor. The two main chapters in this part, 11 and 12, contains the addition of procedures and of user defined types. The ideas in chapter 10 are from [KLMM93] and from a graduate course in methods for incremental semantic analysis held autumn 1992.

For each of the new constructs we will present a short overview of the semantic problems involved with this construct, a description of the solution including the OOSL-specification and appropriate syntax-trees, followed by a complete analysis including dependency graphs and visit sequences. All the methods and algorithms used in the analysis in the following chapters are from [Hed92]. These include: algorithms 7-5 for dependency graph construction, 7-6 for exhaustive visit sequences construction, and 7-9 for incremental visit sequences construction. All algorithms are used with the modifications done in section 10.8.2 in [Hed92] to include doors. Characteristic sets for doors are described in section 10.4.2 in [Hed92]. In the examples in this part we use capital letters to mean Ask-language and lower-case letter to mean other languages such as Simula, OOSL or Pascal.

9 If- and While-Statements

In this chapter we make two simple additions to the Ask-language, we add if- and while-statements.

Overview

The *if-statement* provides the possibility for a two-way branch. Figure 9.1 shows the syntax tree for an if-statement. If *condition* is true then statements_1 are executed else statements_2 are executed. During semantic analysis we only need to check if the condition is of type boolean.

```
BEGIN
 IF condition
 THEN statements1
 ELSE statements2
 END IF;
END;
```

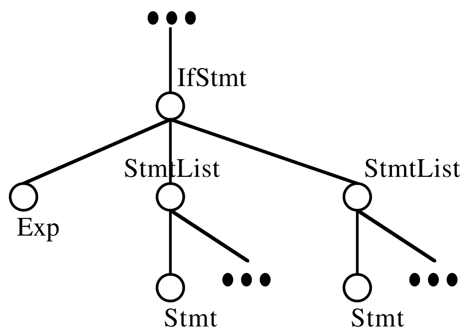


Figure 9.1: Syntax tree for if-statement.

The most common forms of iteration found in programming languages are *for*-, *while*-, and *repeat-statements*. To show how iteration can be added to the Ask-language we add a *while-statement*. In a while-statement the statements are executed until the condition is false. The syntax and the corresponding syntax tree for a while statement is shown in figure 9.2. As for the if-statement, the semantic analysis must check the type of the condition.

```
BEGIN
 WHILE condition DO
 statements
 END WHILE;
END;
```

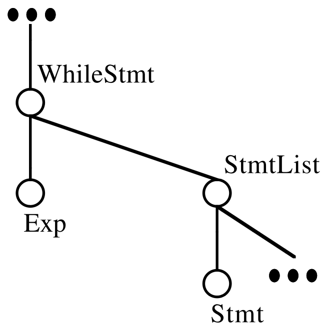


Figure 9.2: Syntax tree for while-statement.

The only new nodes added in this chapter are IfStmt and WhileStmt. All other nodes are from the original Ask-language. This means for example that we use the same exp nodes as for the assign-statement and thus we will get the same incremental update mechanism as for the assign-statement.

Grammar Extensions

To the Stmt production we add the nonterminal IfStmt and WhileStmt.

<Stmt> ::| <NullStmt> | <BlockStmt> | <AssignStmt> | <IfStmt> | <WhileStmt>

<IfStmt> ::= **IF** <Exp> **THEN** <StmtList> **ELSE** <StmtList> **END IF**

<WhileStmt> ::= **WHILE** <Exp> **DO** <StmtList> **END WHILE**

IfStmt

The OOSL specification for the if-statement contains, in addition to the syntactic constructs, one local error attribute, *condError*, and the equation for defining this attribute.

```
IfStmt: cons Stmt ( cond: ref Exp,
  thenPart: ref StmtList, elsePart: ref StmtList ) {
  loc condError: boolean;

  eq condError := not(  cond.tp == unknownType or
                        cond.tp == boolType );
};
```

condError is true if the type of the condition is known and is not boolean.

WhileStmt

The OOSL specification of the while-statement is similar to that of the if-statement.

```
WhileStmt: cons Stmt ( cond: ref Exp,
                       doPart: ref StmtList  {
  loc condError: boolean;

  eq condError := not(  cond.tp == unknownType or
                        cond.tp == boolType );
};
```

Analysis

Analysis of If Statement

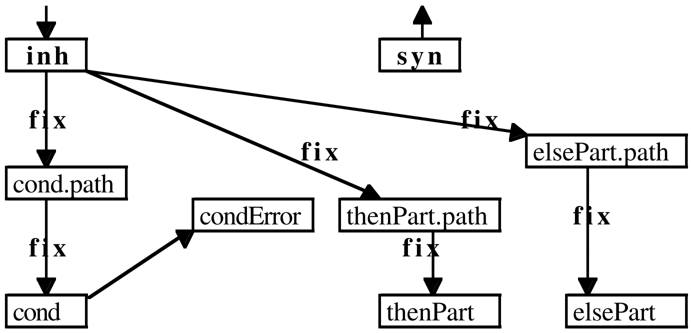


Figure 9.3: Dependency graph for if statement.

From the dependency graph we get the following visit sequences:[24]

```
addto IfStmt {
  impl exhVisit {
    eval cond.path;
    cond.exhVisit;
    eval condError
    eval thenPart.path;
    thenPart.exhVisit;
    eval elsePart.path;
    elsePart.exhVisit;
  };
```

[24] Note that the incremental father visit is empty because the path attribute is fixed (see [Hed92] section 10.8.2.3 for information on handling fix attributes).

```
  impl incSonVisit {
    if S == cond then
      eval condError;
    end if;
  };
};
```

Analysis of While Statement

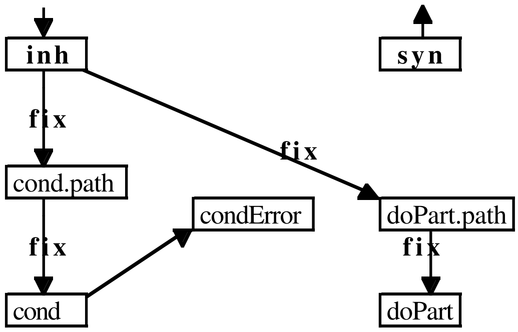


Figure 9.4: Dependency graph for while statement.

Here we get the following visit sequences:

```
addto WhileStmt {
  impl exhVisit {
    eval cond.path;
    cond.exhVisit;
    eval condError
    eval doPart.path;
    doPart.exhVisit;
  };

  impl incSonVisit {
    if S == cond then
      eval condError;
    end if;
  };
};
```

The *with-statement* is used to “open up” a record or an object. As an example, suppose we have a temperature record containing three integers, *max*, *min*, and *current*. Then assigning values to the temperature record can be done in the following two ways:

```
temperature.max := 23;
temperature.min := 10;
temperature.current := 17;

WITH temperature DO
BEGIN
   max := 23;
   min := 10;
   current := 17;
END  WITH;
```

That is, using a with-statement the contents of the temperature record can be accessed directly.

By adding an extra path vertex the lookup rules for the with-statement can be represented. The new vertex, *v*, has two edges, one points to the prefix path vertex for the opened record or class, and one to the path vertex of the surrounding block. The prefix path vertex is examined first. See figure 10.1 for an example.

```
BEGIN (* main *)
  CLASS A BEGIN END;
  REF(A): R;

  WITH R;
  BEGIN
  END WITH;

END;
```

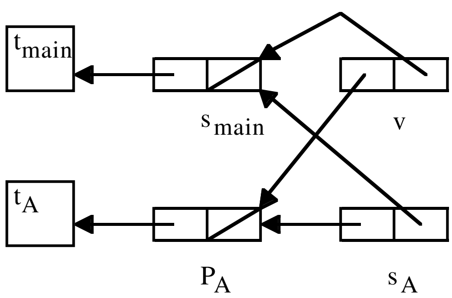


Figure 10.1: Visibility graph for a with statement.

To handle the extra inspect vertex we have developed a new door, *InspectDoor*. Figure 10.2 shows the syntax tree for a simple with statement.

```
BEGIN
 WITH expression DO
  statements
 END WITH;
END;
```

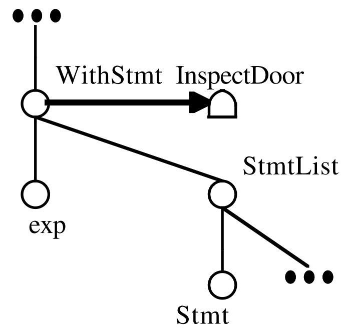


Figure 10.2: Syntax tree for with statement.

The specification of the with-statement contains the son-nodes, *exp* and *StmtList*; an *InspectDoor*; a local error attribute, *inspObjError*; and equations defining the attributes.

```
WithStmt : cons Stmt
( inspObj: ref Exp, doPart: ref StmtList ) {
  iDoor: doorobject InspectDoor;
  loc inspObjError: boolean;

  eq inspObj.path :- path;
  eq iDoor.normalPath :- path;
  eq iDoor.classTp :-
    inspect $x :- inspObj.tp
    when AbstractRefType do getClassType
    otherwise noClassType;
  eq inspObjError := (iDoor.classTp == noClassType);
  eq doPart.path :- iDoor.inspectPath;
};
```

The type attribute of the exp node is examined and the actual class is obtained using the function *getClassType*.

Grammar Extension

<Stmt> ::| <NullStmt> | <BlockStmt> |
<AssignStmt> | <IfStmt> |
<WhileStmt> | <WithStmt>

<WithStmt> ::= **WITH** <Exp> **DO** <StmtList>
END WITH

InspectDoor

The main responsibility of the *InspectDoor* is to add an inspect path vertex to the visibility graph. This is done by adding a *TwoPath* object (*newPath*) to the door. The first edge in *newPath* points to the prefix path vertex of the class of the inspected object and the second edge to the path vertex of the surrounding block. A reference to *newPath* is passed as path attribute to *StmtList*.

By not assigning the first edge of *newPath* directly to the prefix path vertex but through a watch path object (*theWatch*) all bindings via the inspect path vertex can be registered. This information is used for later incremental rebinding of UseDoors bound through this with-statement.

The inherited attribute *classTp* is the type of the inspected expression. This is either a class type or a *noClassType* object. This gives the following specification of InspectDoor:

```
InspectDoor : door {
   inh normalPath : ref SearchPath fix;
   inh classTp : ref AbstractClassType;
   syn inspectPath : ref SearchPath fix;

   theWatch : object WatchPath;
   newPath : object TwoPath;
   eq theWatch.path :- classTp.getPrefixPath;
   eq newPath.first :- theWatch;
   eq newPath.second :- normalPath;
   eq inspectPath :- newPath;
};
```

Note that if *classTp* denotes a *noClassType* object then *getPrefixPath* returns *emptyPath*. Figure 10.3 shows a *InspectDoor* with all attributes and static objects and figure 10.4 shows the information flow in a *WithStmt*.

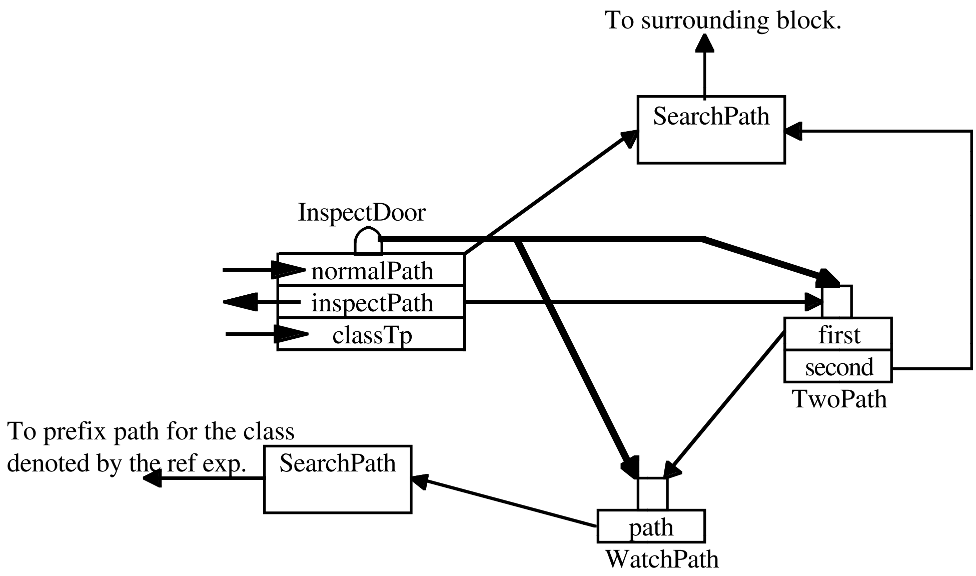


Figure 10.3: InspectDoor with attributes and static objects.

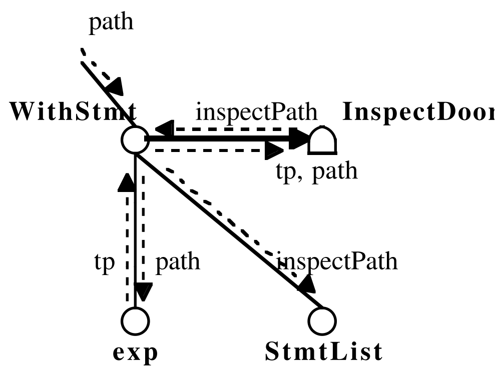


Figure 10.4: Information flow for WithStmt.

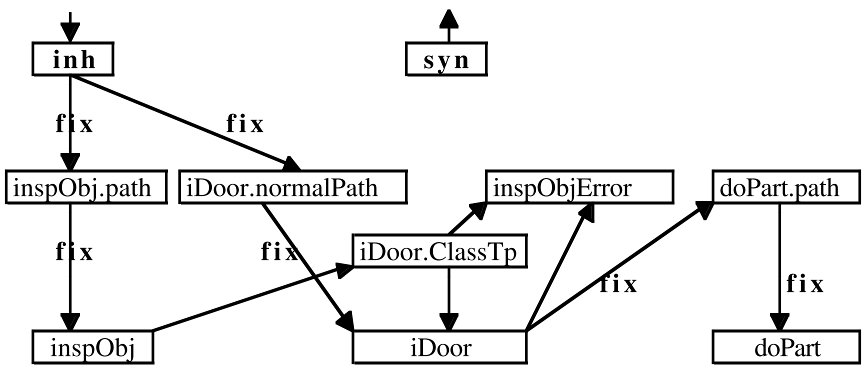


Figure 10.5: Dependency graph for WithStmt.

Analysis

Analysis of With Statement

Figure 10.5 shows the dependency graph for the *WithStmt*.

The corresponding visit-sequences are:

```
addto WithStmt {
  impl exhVisit {
    eval inspObj.path;
    eval iDoor.normalPath;
    inspObj.exhVisit;
    eval iDoor.classTp;
    iDoor.exhEvalVisit;
    eval inspObjError;
    eval doPart.path;
    doPart.exhVisit;
  };

impl incSonVisit {
    if S == inspObj then
      eval iDoor.classTp;
      iDoor.incOwnerVisit;
      eval inspObjError;
    end if;
  };

  impl incDoorVisit {
    eval inspObjError;
  };
};
```

Analysis of InspectDoor

Step I - *Implement conditions*: This step is empty.

Step II - *Local analysis*: Build the dependency graph.

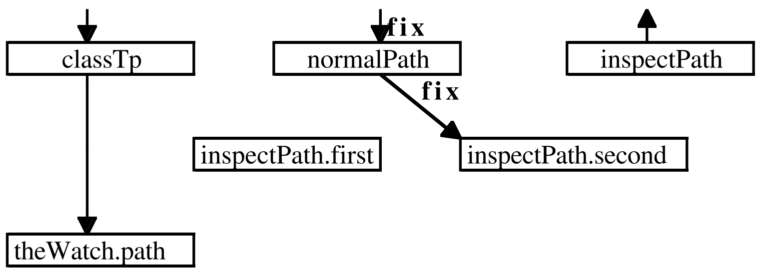


Figure 10.6: Dependency graph for InspectDoor after step II.

Step III - *Add send vertices*: Added send vertex “lookupChanged”.

Step IV - *Simplify send / receive vertices*: Empty.

Step V - *Implement dependency function*: The send vertex of InspectDoor produces:

```
addto InspectDoor {
  fAttemptedEdge: func ref Set[UseDoor]
    :- theWatch.attemptedEdge.contents;
};
```

Step I - *Implement conditions*: This step is empty..

Step II - *Local analysis*: Extend the dependency graph with the send vertex.

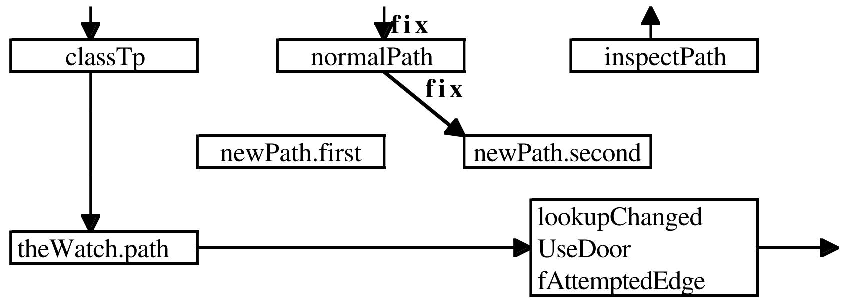


Figure 10.7: Modified dependency graph for InspectDoor after step II.

This results in the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	Ø	Ø	Ø
exhEvalVisit	Ø	{ **v**(theWatch.path, newPath.second, newPath.first) }	{ **v**(lookupChanged, UseDoor, fAttemptedEdge) }
incOwnerVisit	Ø	{ **v**(theWatch.path) }	{ **v**(lookupChanged, UseDoor, fAttemptedEdge) }

exhEvalVisit and *incOwnerVisit* can now be implemented.

```
addto InspectDoor {
  impl exhEvalVisit{
    attemptedSet: ref Set[UseDoor];
    attemptedSet :- fAttemptedEdge;
    for $d :- attemptedSet.each
    do $d.deEvalLookupChanged;
    end for;
    eval theWatch.path;
    eval newPath.second;
    eval newPath.first;
    for $d :- attemptedSet.each
    do $d.evalLookupChanged;
    end for;
    evaluator.worklist.addSet( attemptedSet );
  };

  impl incOwnerVisit {
    attemptedSet: ref Set[UseDoor];
    attemptedSet :- fAttemptedEdge;
    for $d :- attemptedSet.each
    do $d.deEvalLookupChanged;
    end for;
    eval theWatch.path;
    for $d :- attemptedSet.each
    do $d.evalLookupChanged;
    end for;
    evaluator.worklist.addSet( attemptedSet );
  };
};
```

11 Procedures and Parameters

In this chapter we describe how support for procedures and parameters can be added to the Ask-language.

Overview

A form of abstraction found in almost all programming languages is the procedural abstraction. This is a collection of declarations and statements packaged together and stored with a name. The procedure can be called from different parts in the program. Furthermore each procedure has its own local environment in which it executes. To be able to change the execution the procedure can have parameters. These parameters can be sent to the procedure or from the procedure.

There is a wide range of possible procedure constructs ranging from a simple procedure without any parameters to the procedure pattern in BETA, which contains multiple in- and out-parameters. Together with block the procedure is the simplest form of program modularization.

Often a procedure returning a parameter is called a function. We use the term *procedure* to mean both procedure and function.

Semantic Problems

The semantics for a procedure can be broken down into two main parts, the declaration and the usage. Several different semantic problems arise, the most important are:

- The procedure introduces a new block and possibly introducing local variables.

- It is possible to parameterize the procedure, that is, to use the same procedure with different data. The parameters supplied at the usage of the procedure (the *actual parameters*) must be checked against the parameters given in the declaration of the procedure (the *formal parameters*). This involves matching the types of actual and formal parameters and to check to see if the numbers of actual and formal parameters are the same.

- Parameters can be of different sort, i.e., in, out or in/out.

- Parameters can be passed using several different techniques, e.g., by value, by reference or by name.

- The procedure can be used in two ways, as a statement or as an expression.

These items will briefly be covered below. We start by discussing different aspects of procedure declaration and after that, in the section "Calling the Procedure", we discuss procedure usage.

A New Block

As stated above, the procedure executes in an environment of its own. In a language with a block structuring mechanism (such as Ask), this means that declaring a procedure introduces a new block. In conjunction with this, it introduces a new symbol table, which is added to the visibility graph. Adding a new block also means that the procedure can declare local variables. Figure 11.1 shows the syntax tree for a procedure declaration with an added block. A *ProcDecl* node is added for each procedure declaration. The *ID* node in the tree is the name of the procedure and the *block* node is the same node class as used for anonymous blocks.

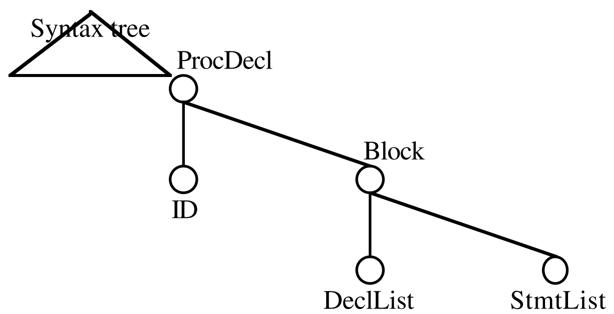


Figure 11.1: Syntax tree with procedure declaration, first attempt.

Variables used in a procedure are bound first in the local block and if not found there, in the most closely nested block that contains the declaration of the variable. The visibility graph for procedures is thus the same as for anonymous blocks. For a description and definition of visibility graphs see chapter 8 "The Ask-language". Figure 11.2 shows the visibility graph for a simple procedure declaration. The scope for variables declared in a procedure is the procedure block and all nested blocks in the procedure. This is also the same as for anonymous blocks.

```
BEGIN (* main *)

  PROCEDURE P;
  BEGIN
  END;

END;
```

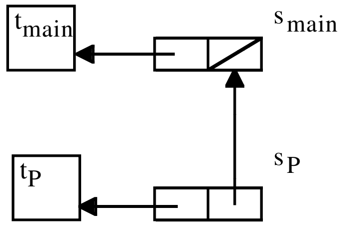


Figure 11.2: Visibility graph for a simple procedure.

Parameters

As a procedure can have parameters, the procedure declaration must contain these parameters (called formal parameters). Each of these can be seen as a declaration of a variable having a name and a type. A procedure declaration can contain variable numbers of parameter declarations so we need a list to handle the formal parameters. The (formal) parameters can be used as ordinary variables in the procedure block so they must be added to the local symbol table. In figure 11.3 the syntax tree for a procedure declaration with formal parameter list is shown. We add the node class *FormalParamList* to handle the formal parameters. This is a list of *VarDecl* nodes.

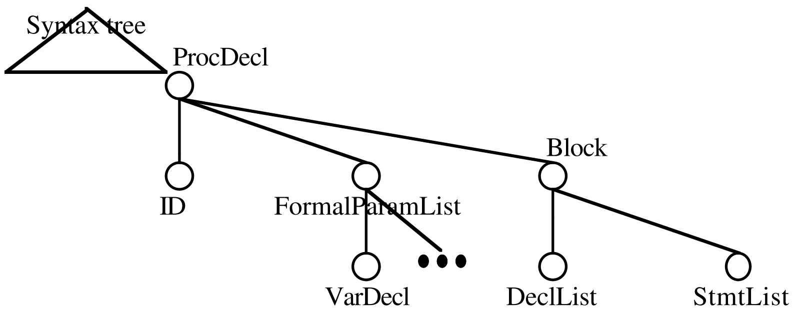


Figure 11.3: Syntax tree with procedure declaration, second attempt with formal parameter list added.

In some languages (such as ADA) the parameters can be declared as in, out, or in/out. An in-parameter is not changed by the procedure, an out-parameter is assigned a value by the procedure and returned to the caller but cannot be used in any other way in the procedure. An in/out-parameter can both be used and assigned a value. Thus the semantics of the parameters is different depending on the declaration in the formal parameter list.

Another aspect to be considered is parameter passing, that is, by which mechanism the parameters are transferred to the procedure. The most common mechanisms are:[25]

- Value – the value of the actual parameter is transferred.

- Reference – a reference (pointer to) the actual parameter is transferred.

- Name – information about the actual parameter is transferred. This information contains type, if the parameter is variable or expression, and how to evaluate the expression.

The parameter passing information is used in code generation and lacks static-semantic implications.

Sort and Type

A way to implement support for different passing mechanisms and for in/out parameters is to extend the type system. This could be done by adding a kind attribute to the DeclDoor and the UseDoor. This kind-attribute would be passed from the declaration of the parameters to the usage in the same way as the tp attribute. This is not done in this first version of procedure implementation.

Return Value

Until now we have only talked about parameters and overlooked the fact that a procedure can return a value. This means that a type has to be added to the procedure declaration and that the type system has to be extended. It also means that we need some construct to assign the procedure a value. For this purpose we add a *DeclType* node class and a *return-statement* (*ReturnStmt* node class). See Figure 11.4 for the resulting syntax tree.

Note that to be able to check the type of the return-statement, the return type of the procedure must be known within the procedure block. The return type must of course also be known in the block in which the procedure is declared.

Calling the Procedure

A procedure can be called in two ways, as a statement or as an expression. This leads to two constructs, *ProcCall* for the usage as a statement, and *ProcUse* for the usage as an expression. The following Simula code illustrates the different usage of a procedure.

---

[25] See [ASU86] for a description of parameter passing.

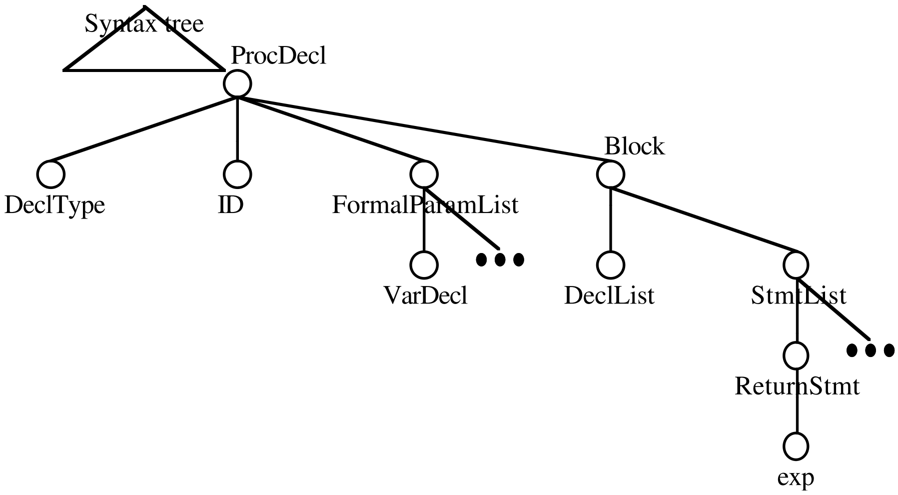


Figure 11.4: Syntax tree with procedure declaration, third attempt with return type and return statement added.

```
begin
   integer a;
   integer procedure P;;

   P;          ! P used as a statement;
   a := P;     ! P used as an expression;
end;
```

The ProcUse node class has a son of type *ActualParamList*. This makes it possible to specify a list of zero or more actual parameters. Each parameter is represented by an *ActualParam* node. The usage as a statement is handled by the node class *ProcCall*, a small class using ProcUse as a son. Figure 11.5 shows a syntax tree for procedure usage.

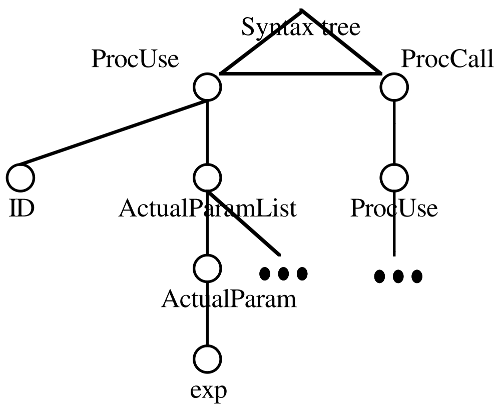


Figure 11.5: Syntax tree with procedure usage.

Recursion, meaning that a procedure can be used in the local block, is possible if the procedure declaration is visible from within the procedure.

And More

A problem arising when using procedures and arrays is how to differentiate between arrays and procedure call. In for instance Pascal the syntax are different, using parenthesis for procedures and brackets for arrays, as in *p*() and *a*[] (supposing *p* is a procedure and *a* an array). This is not so in for instance Simula where parenthesis are used both for procedures and for arrays (*p*(), *a*()). In Ask we assumed different syntax for procedures and arrays.

Grammar extensions

Formalising the above syntax trees we get the following additions to the context-free grammar for procedure declarations and usage.

<Decl>	::|	<NullDecl> | <VarDecl> | <ClassDecl> | <ProcDecl>
<ProcDecl>	::=	<DeclType> **PROCEDURE** <ID> (<FormalParamList>) <Block>
<FormalParamList>	::+	<VarDecl> **,**
<ProcCall>	::=	<ProcUse>
<ReturnStmt>	::=	<Exp>
<ProcUse>	::=	<ID> (<ActualParamList>)
<ActualParamList>	::+	<ActualParam> **,**
<ActualParam>	::=	<Exp>

Procedure Declaration and Parameters

We now turn to the details of the actual implementation of procedures in the Ask-language. Our aim was to make a simple implementation and to try to reuse the doors we already had. We also tried to make the changes to the rest of the system as small as possible. In this first implementation we do not model different parameter sorts, assuming all parameters are in-parameters. Figure 11.6 gives an example of the syntax for procedure declaration, procedure usage, and the return statement.

```
INTEGER PROCEDURE P(INTEGER: A; BOOLEAN: B)
BEGIN
  RETURN 1;
END;
P(n, TRUE);
B := P(n, TRUE);
```

Figure 11.6: Example of procedure declaration and usage.

As seen in the previous sections we have added several different nodes to the syntax tree. Nodes such as *ProcDecl* for procedure declarations, *ProcUse* and *ProcCall* for procedure usage, and *FormalParamList* and *ActualParamList* for parameter checking. Three new doors have also been constructed, these are *ProcDoor*, *ProcUseDoor*, and *ReturnDoor*. In the next few sections we take a closer look at these syntax node classes and doors classes in turn. Figure 11.7 gives a preview of how the syntax tree will look like after addition of the doors.

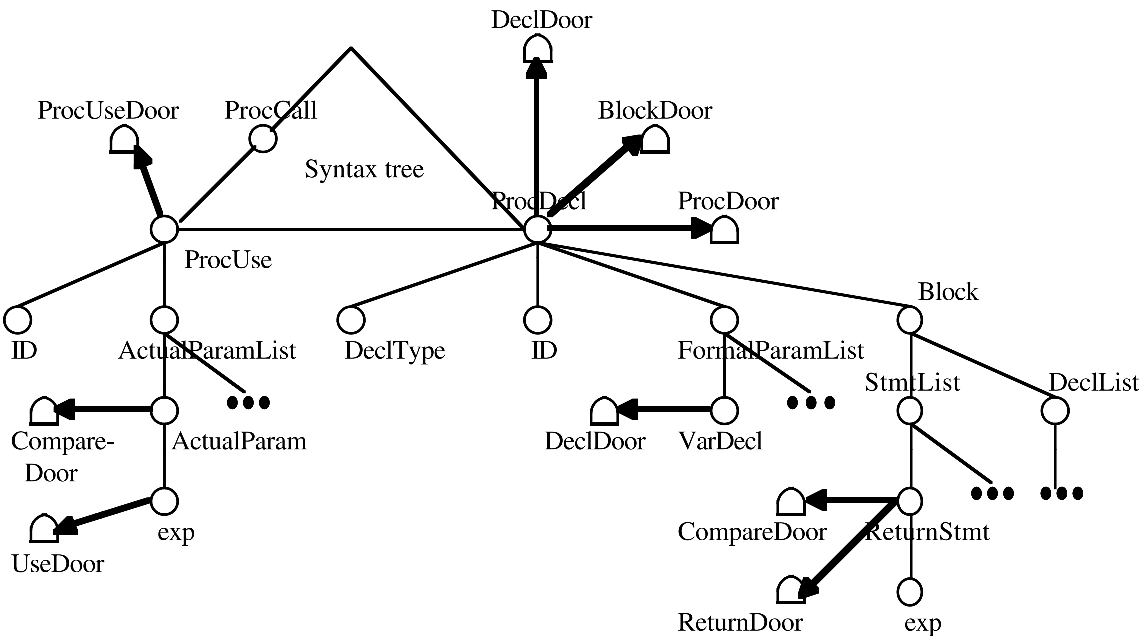


Figure 11.7: Syntax tree with procedure declaration and usage, final attempt with appropriate doors added.

FormalParamList, ActualParamList and Their Contents

The first step is to define the classes needed for formal and actual parameters. The main semantic concern regarding parameters is to find a way to check the type of the actual parameters against the formal parameters. Procedure usage and procedure declaration can be situated far apart in the program and therefore the distance between the corresponding syntax nodes in the syntax tree can be large. To pass information between the actual and the formal parameters we need a door. Depending on where we add the doors this can be done in two ways:

1. Each parameter (formal and actual) has a door of its own or

2. the procedure declaration and the procedure usage own the doors and we use a list construction mechanism to assemble and disassemble the type information.

We use the second approach, using the list constructions introduced in [Hed93]. At each procedure declaration a compound attribute containing all type information is constructed and at procedure usage this attribute is broken up and distributed to the actual parameters.

We need some way to store the type of a parameter according to the position of the parameter in a list (an integer). For this purpose we introduce a generic class *ArrayContainer* with the following interface: *put* stores an element, *Elem*, at position *i*; *get* returns the element at position *i*; and *size* returns the number of stored elements.[26]

```
ArrayContainer: class [E: class ANYCLASS]{
  put: proc(x: integer, Elem: ref E);
  get: func ref E (x: integer);
  size: func integer ;
};
```

Next we define the classes needed to store the formal types. *AbstractFormalTypes* is, as the name suggests, an abstract superclass and as such not instantiated. The functions in this class are: *at* returns the type stored at position *i*; *add* adds a type at position *i*; and *noOfPara* returns the number of stored parameters.

```
AbstractFormalTypes: class  {  (* abstract *)
  at: ref Type func(i: integer);
  add: ref FormalTypes func(i: integer, tp: ref type);
  noOfPara: integer func;
};
```

FormalTypes is a subclass of AbstractFormalTypes in which a static object *theArray* is added and all functions are implemented. theArray is of type *ArrayContainer*[*Type*] and the implementation of the functions in *FormalTypes* uses the get, put, and size functions in ArrayContainer.

```
FormalTypes: class AbstractFormalTypes  {
   loc theArray: object ArrayContainer[Type]
   impl at :- theArray.get(x);
   impl add: { theArray.put(i, tp);
               add :- this FormalTypes;
             };
   impl noOfPara := theArray.size;
};
```

[26]In the actual implementation in BETA, the class *ArrayContainer* in BETA's container library is used.

EmptyFormalTypes is used to denote an empty list of formal parameters. The function *at* returns *unknownType* and noOfPara returns 0.

```
EmptyFormalTypes: object AbstractFormalTypes   {
  impl at :- unknownType;
  impl add :- this EmptyFormalTypes;
  impl noOfPara := 0;
};
```

We are now ready to implement the syntax node classes for the formal and actual parameter lists. Both classes are declared as list and as a subclass of *Descendant*. *FormalParamList* stores nodes of type *VarDecl* and has one synthesized attribute, *formals*. Note the use of attribute construction in the equation defining the attribute *formals*. In attribute construction (AC) the synthesized attribute, in this case formals, is constructed using synthesized attributes of son nodes. AC introduces a temporary variable, $X, that is used in the AC sub-expressions. The first of the two sub-expressions is used for initialization of $X and the second is executed once for each son node (in left to right order). The expressions are separated by "|". As stated in [Hed93] the expressions must be without side-effects. The add procedure in FormalTypes will add the son's position and type and return a reference to the same FormalTypes object. Although this is a side-effect it is local to the FormalTypes object and does not change any other object. This means that the equation defining formals has no side-effects as a whole.

```
FormalParamList: list Descendant(f: ref VarDecl) {
  syn formals: ref FormalTypes;

  (* build formals consisting of type information *)
  (* from all sons *)
  eq formals :- AC $X := (new FormalTypes |
             $X.add(son.pos, son.tp));
};
```

By using *VarDecl* as the node type stored in *FormalParamList* the formal parameters are treated as ordinary variable declarations.

A special node class, *ActualParam*, is added and used as the node type for the list class *ActualParamList*. ActualParam contains the attributes and equations needed to type-check the actual parameter. The error attribute *typeError* is true if there is a mismatch between the actual and formal parameter types. *ActualParam* has an *Exp* node as son. This makes it possible to use arbitrary expressions as actual parameters.

```
ActualParam: cons Descendant(is: ref Exp) {
  inh formalTp: ref Type;  (*should be of this type*)
  loc typeError: boolean;  (* wrong actual type *)
  cmpDoor: doorobject CompareDoor;
```

```
    eq cmpDoor.leftType :- formalTp;   (* inh type *)
    eq cmpDoor.rightType :- is.Tp;     (* type of son *)
    eq typeError := cmpDoor.inComparable; ;
};
```

Matching of types is an important and often used operation so we have constructed a door to be used when matching types. The interface to CompareDoor is presented below. See chapter 13, “Matching of types”, for a complete description of the CompareDoor.

```
CompareDoor: door {
    inh leftType: ref Type;
    inh rightType: ref Type;
    syn inComparable: boolean;
    syn notStaticKnown: boolean;
};
```

As the last class for parameter checking *ActualParamList* is implemented. We make use of list distribution to distribute the formal type of a parameter at position *i* to the actual type at position *i*. The local attribute *parameterError* is true if the numbers of the actual and formal parameters are different.

In this specification we use three built-in features in list nodes and in sons of list nodes. These are:

- **cardinal** - an attribute in list nodes indicating the number of sons in the list.
- **son** - a pseudo variable indicating a current son node. This is used in collective equations to access or define attributes of a son node.
- **pos** - an attribute in all nodes that are sons of a list node. It indicates the position in the list, starting with 1 for the leftmost son.

```
ActualParamList: list Descendant(a: ref ActualParam)
{
    inh formals: ref FormalTypes;
    loc parameterError: boolean;
    eq parameterError := cardinal <> formals.noOfPara;
    eq son.formalTp :- if son.pos <= formals.noOfPara
                        then formals.at(son.pos)
                        else unknownType;
};
```

ProcDecl

We now have all utility classes needed to implement *ProcDecl*. ProcDecl owns three doors, DeclDoor, BlockDoor, and ProcDoor. The use of DeclDoor and BlockDoor is the same as for example in declarations of a class, i.e., DeclDoor is used to declare the procedure in the current block and BlockDoor is used in introducing a new local block for the procedure. The difference between simple declarations and procedure declarations is that in a procedure declaration we need to store information about the formal parameters. This information is, as said above, assembled in the son node *FormalParamList* and transferred to the ProcDecl node. Here we set up equations to transfer the formal parameter information to ProcDoor. Besides this the return type of the procedure is also transferred to ProcDoor.

In ProcDoor the type object of the procedure is assembled and the type is transferred back to ProcDecl and is used as the type in the declaration of the procedure. ProcDecl can be seen as a syntax node used for transferring information between the doors and the son nodes (figure 11.8).

```
ProcDecl: cons Decl (
    returnType: ref DeclType,    (* returned type *)
    declID: ref ID,              (* procedure name *)
    fParam: ref FormalParamList, (* formal parameters *)
    b: ref Block) {              (* procedure block *)

  dDoor: doorobject DeclDoor;
  bDoor: doorobject BlockDoor;
  pDoor: doorobject ProcDoor;

 (* equations for the inherited attributes in DeclDoor *)
  eq dDoor.tp :- pDoor.ProcTp;
  eq dDoor.ident :- declID.ident;
  eq dDoor.table :- table;

 (* equations for the inherited attributes in BlockDoor *)
  eq bDoor.encPath :- path;

 (* equations for the inherited attributes in ProcDoor *)
  eq pDoor.retType :- returnType.tp;
  eq pDoor.formals :- fParam.formals;

 (* equations for the formal parameter list *)
  eq fParam.table :- bDoor.table;

 (* equations for the inherited attributes in block *)
  eq b.path :- bDoor.locPath;
  eq b.retTp :- pDoor.ProcTp;  (* 1 *)
  eq b.table :- bDoor.table;
};
```

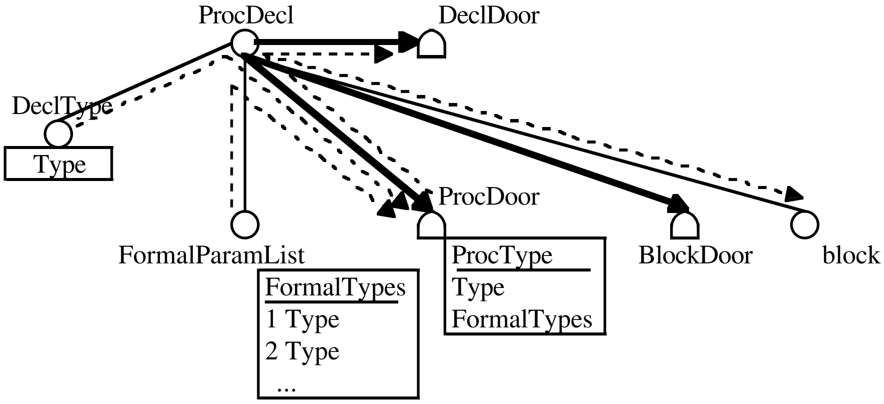


Figure 11.8: Information flow in ProcDecl.

Note the equation 1 in the specification of ProcDecl. Here we propagate the type object to the block node. This is used later on to type-check the return-statement. See the section "Return Statement and ReturnDoor" for more details.

ProcDoor

In *ProcDoor* the type of the procedure is assembled. The type is constructed from the inherited attributes *retType* (the type returned by this procedure) and *formals* (the type of all formal parameters). The synthesized attribute *procTp* is used to transfer the procedure type to the owning node. Note that the attribute *procTp* is fixed meaning that the value of *procTp* (a reference to *theProcType* object), once evaluated, is not changed. However the contents of *theProcType* can change due to changes in the syntax tree.

```
ProcDoor: door{
  (* attributes *)
  inh retType: ref type;           (* return type *)
  syn procTp: ref ProcType fix;    (* the type *)
  inh formals: ref FormalTypes;

  (* part-objects *)
  theProcType: object ProcType; (* declaration type*)

  eq theProcType.formals :- formals;(* formal param*)
  eq theProcType.retType :- retType;(* return type *)

  (* equations for procTp *)
  eq procTp :- theProcType;
};
```

Extensions to the Type System

In ProcDoor we used a new type to declare a procedure. This means that this type has to be added to the type system. The additions are:

- *AbstractProcType* – an abstract superclass for the procedure types.
- *ProcType* – a type used to indicate procedure declaration.
- *noProcType* – a type used to indicate missing procedure declaration.
- *noType* – a type indicating that a procedure does not have a return type, i.e., it is not a function.

```
AbstractProcType: class type {  (* abstract *)
  loc formals: ref AbstractFormalTypes;
  loc retType: ref type;
};

ProcType: class AbstractProcType { };

noProcType: object AbstractProcType {
  eq formals :- emptyFormalTypes;
  eq retType :- noType;
};

noType object type { };
```

The resulting type class hierarchy is shown in figure 11.9.

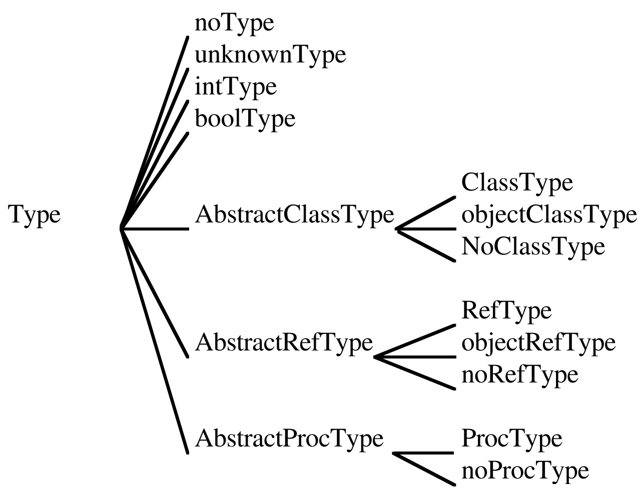


Figure 11.9 : Type class hierarchy after addition of procedure types.

Procedure usage

ProcUse

We now turn to the usage of procedures and explain the classes *ProcUse*, *ProcCall*, and *ProcUseDoor*. If we want to use a procedure as an expression the node class ProcUse is used. ProcUse is an extension of the *Use* node class. The additions are that ProcUse has one additional son, *aParam*, for the actual parameter list; an extended *UseDoor*, *ProcUseDoor*; an additional local error attribute, *noProcedure*; and some additional equations. These equations define the error attribute and the attributes in *ProcUseDoor* and *aParam*.

```
ProcUse: cons exp (
    useId: ref ID,                    (* ident of proc *)
    aParam: ref ActualParamList) {    (*formal parameters*)

  pDoor: doorobject ProcUseDoor;

  loc noProcedure: boolean;  (* this is not a procedure*)
  loc notDeclared: boolean;  (* name not declared *)

  eq pDoor.path :- path;
  eq pDoor.ident := useId.ident;
  eq tp :- pDoor.tp;

  (* equation for actual parameter list *)
  eq aParam.formals :- pDoor.formals;
  eq aParam.path :- path;

  (* equations for error attributes *)
  eq notDeclared := useId.ident <> "" and
                    not pDoor.declared;
  eq noProcedure := useId.ident <> "" and
                    pDoor.tp == noProcType;
};
```

ProcCall

To be able to use procedures as a statement we define a construction node *ProcCall* as a subclass of *Stmt*. ProcCall has a *ProcUse* node as son.

```
ProcCall: cons Stmt (theP: ref ProcUse) { };
```

ProcUseDoor

The extended *UseDoor* used in *ProcUse* has an additional synthesized attribute *formals* for the formal parameter types and an equation defining *formals*. A small change in the equation defining *tp* is made to reflect the changes made to the type system.

```
ProcUseDoor: door UseDoor{
  syn formals: ref FormalTypes;
  eq tp :- inspect $E :- binding.getTp
           when AbstractProcType do $E.retType
           otherwise noProcType;

  eq formals :- inspect $E :- binding.getTp
                when AbstractProcType do $E.formals
                otherwise EmptyFormalTypes;
};
```

From UseDoor we have the following attributes and equations:

```
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 declared := binding =/= nullEntry;
};
```

Summary

Figure 11.10 summarises all of the above. Here we see that each of the formal parameters are registered in the local symbol table (for the procedure). This is done using the same DeclDoor as in all other declarations. Furthermore the procedure declaration is registered in the symbol table for the enclosing block using the same mechanisms. The Types of all formal parameters are assembled in the attribute *formals* in *FormalParamList* and propagated to the semantic object *theProcType* in *ProcDoor*. The Type object in the DeclDoor for the procedure declaration contains a reference to theProcType.

The usage of the procedure is bound to the DeclDoor for the procedure declaration using the attribute *binding*.

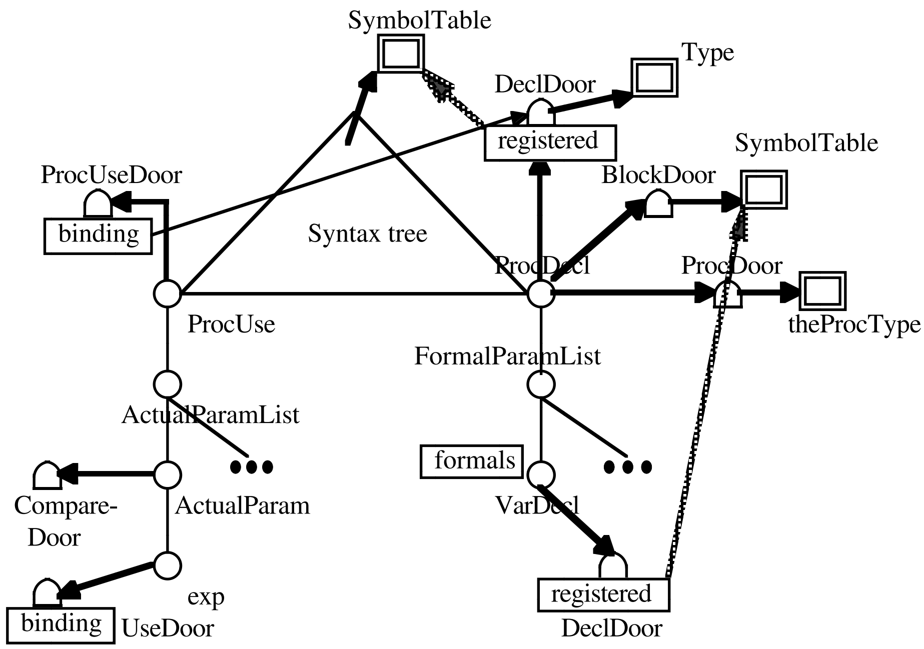


Figure 11.10: Attributed syntax tree for procedure declaration and usage.

Return Statement and ReturnDoor

In the previous sections we introduced procedure declaration and usage. To complete the introduction of procedures we need a mechanism to specify and check the return parameter in a block. There are mainly two ways to return a value in a procedure: as an assignment to the procedure name, as in for example Simula, or by using a return statement, as in for example ADA. We choose to implement a return statement. In this first attempt only a single value is returned, this could later be extended to a list of parameters.

The semantics for the return statement should be that:

- it is valid only in the statement part of a procedure declaration, i.e., a *ProcDecl* where the type is specified and
- the type of expression matches the return type in the procedure declaration.

First of all we need a mechanism to propagate the return type in the syntax tree. For this we add an inherited attribute, *retTp*, to the Descendant node class. Here we also add a collective equation that propagates this attribute to all son nodes of class Descendant or any subclass of Descendant. As Descendant is the super class of all node classes except Program, this means that the retTp attribute will be propagated throughout the syntax tree. In the Program node we set the retTp to *noProcType* to denote an empty return type.

```
addto Descendant {
  inh retTp: ref AbstractProcType fix;

  eq son Descendant.retTp :- retTp;
};

addto Program {
  eq son Descendant.retTp :- noProcType;
};
```

Note that the retTp attribute is a reference to an AbstractProcType, that is, the complete procedure type and not just to the return type. As stated above, the ProcDecl specification contains an equation that sets the retTp attribute to the type object of the procedure, (which of course contains the return type).

The return statement is a construction subclass of *Stmt*. We use a CompareDoor to type check the returned parameter. The local attribute *typeError* is true if the types do not match. If the return statement is used outside a procedure with a return type defined, the local error attribute *noFunction* is true.

```
ReturnStmt: cons Stmt(rt: ref Exp) {
  loc typeError: boolean;
  loc noFunction: boolean;
  rDoor: doorobject ReturnDoor;
  cmpDoor: doorobject CompareDoor;

  eq rDoor.retTpD :- retTp;   (* type down to rDoor *)
  eq rt.path :- path;         (* path down to son *)
  eq cmpDoor.leftType :- rDoor.tp;(*type from rDoor*)
  eq cmpDoor.rightType :- rt.tp;  (*type from son *)
  eq typeError :- cmpDoor.inComparable;
  eq noFunction :- rDoor.tp == noType;
};
```

A new door, *ReturnDoor*, is constructed and used in ReturnStmt. This door is implemented as follows:

```
ReturnDoor: door {
  (* attributes *)
  inh retTpD: ref AbstractProcType;
  syn tp: ref type;        (* the type *)
  eq tp :- retTpD.retType;
};
```

Other solutions

Another solution to the problem of type checking the parameters is to let each parameter (formal and actual) have a door of its own. This has both advantages and disadvantages. The main advantage is that this makes it possible to incrementally evaluate each parameter separately. The disadvantage is that it takes considerably more space to have one door for each parameter.

Analysis

In this section all classes introduced are analysed using the methods introduced in chapter 9 and 10 in [Hed92] and in [Hed93] for list constructions. For all classes the dependency graph, the topological order, and all the visit sequences are presented. The characteristic sets for all doors are also presented. For Proc-UseDoor and ReturnDoor the analysis is described in more detail.

Note that for clarity the attributes *table*, *path*, and *retTp* defined in *Descendant* are not always shown in the analysis.

Analysis of FormalParamList

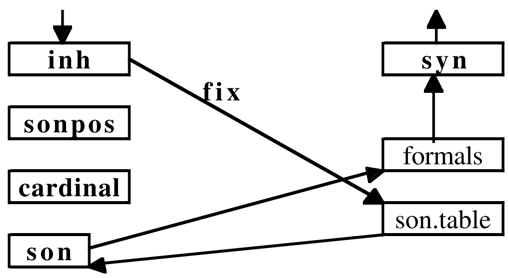


Figure 11.11: Dependency graph for FormalParamList.

Sort	exhVisit	incFatherVisit	incSonVisit	incBrotherVisit
inh		—		
son.table	∀son **eval son**.table			
son	∀son **son**.exhVisit			
formals	**eval** formals		**eval** formals	**eval** formals
syn			father.incSonVisit	father.incSonVisit
cardinal				
sonpos				

Analysis of ActualParamList

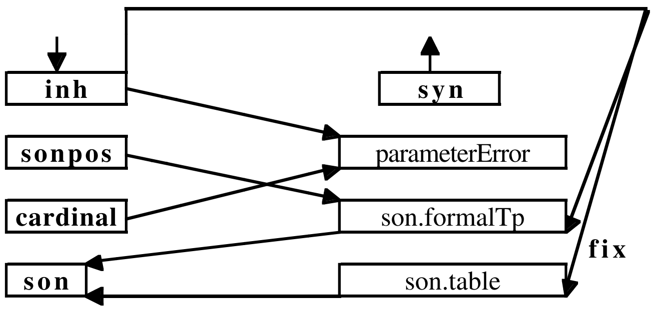


Figure 11.12: Dependency graph for ActualParamList.

Sort	exhVisit	incFatherVisit
inh		
son.table	∀son eval **son**.table	
sonpos		
cardinal		
parameterError	**eval** parameterError	**eval** parameterError
son.formalTp	∀son **eval son**.formalTp	∀son **eval son**.formalTp
son	∀son **son**.exhVisit	∀son **son**.incFatherVisit
syn		

Sort	incSonVisit	incBrotherVisit
inh	—	
son.table		
sonpos		
cardinal		
parameterError		**eval** parameterError
son.formalTp		∀son (from aft) **eval son**.formalTp
son		∀son (from aft) **son**.incFatherVisit
syn		

Analysis of ActualParam

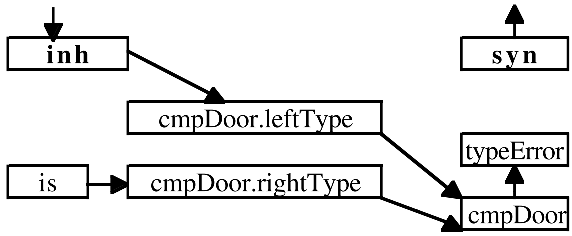


Figure 11.13: Dependency graph for ActualParam.

Sort	exhVisit	incFatherVisit
inh		
is	is.exhVisit	
cmpDoor.leftType	**eval** cmpDoor.leftType	**eval** cmpDoor.leftType
cmpDoor.rightType	**eval** cmpDoor.rightType	
cmpDoor	cmpDoor.exhVisit	cmpDoor.incFatherVisit
typeError	**eval** typeError	**eval** typeError
syn		

Sort	incDoorVisit for cmpDoor	incSonVisit for is
inh		
is		
cmpDoor.leftType		
cmpDoor.rightType		**eval** cmpDoor.rightType
cmpDoor		cmpDoor.incOwnerVisit
typeError	**eval** typeError	**eval** typeError
syn		

Analysis of ProcUse

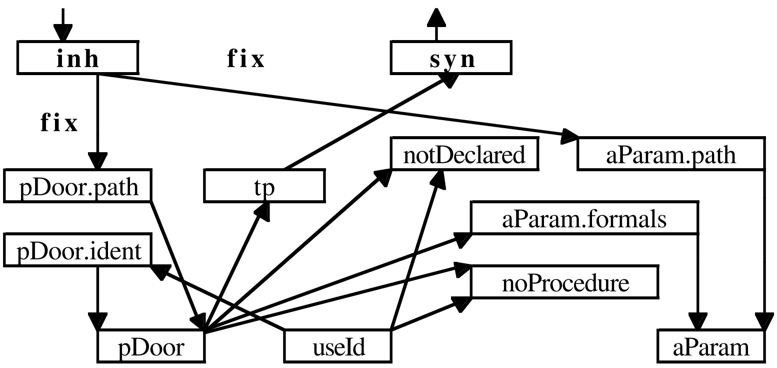


Figure 11.14: Dependency graph for ProcUse.

Sort	exhVisit	incFatherVisit	incSonVisit for useId
inh		—	
pDoor.path	**eval** pDoor.path		
useId	useID.exhVisit		
pDoor.ident	**eval** pDoor.ident		**eval** pDoor.ident
pDoor	pDoor.exhVisit		pDoor.incOwnerVisit
notDeclared	**eval** notDeclared		**eval** notDeclared
noProcedure	**eval** noProcedure		**eval** noProcedure
aParam.formals	**eval** aParam.formals		**eval** aParam.formals
aParam.path	**eval** aParam.path		
aParam	aParam.exhVisit		aParam.incFatherVisit
tp	**eval** tp		**eval** tp
syn			father.incSonVisit

Sort	incDoorVisit for pDoor	incSonVisit for aParam
inh		—
pDoor.path		
useID		
pDoor.ident		
pDoor		
notDeclared	**eval** notDeclared	
noProcedure	**eval** noProcedure	
aParam.formals	**eval** aParam.formals	
aParam.path		
aParam	aParam.incFatherVisit	
tp	**eval** tp	
syn	father.incSonVisit	

Analysis of ProcDecl

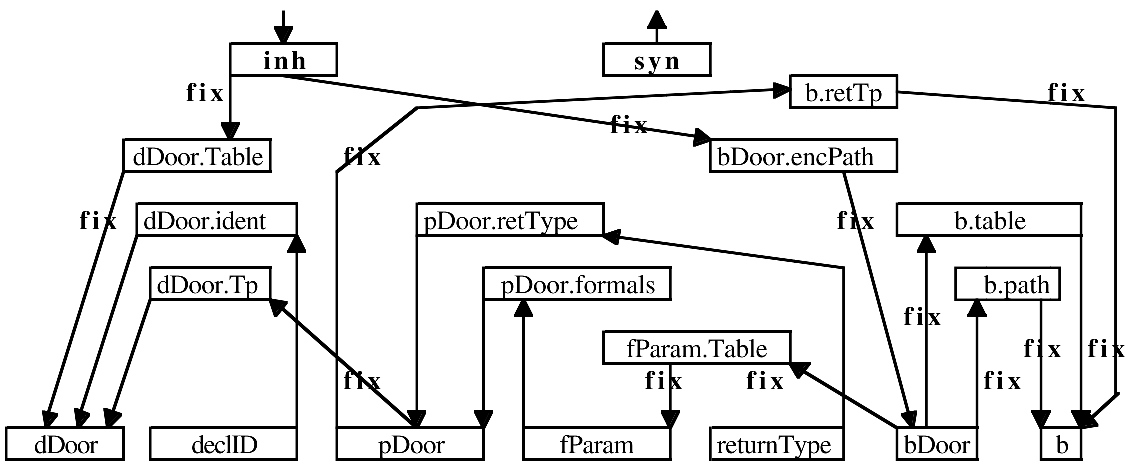


Figure 11.15: Dependency graph for ProcDecl.

Sort	exhVisit	incFatherVisit	incDoorVisit for pDoor
inh		—	—
dDoor.table	**eval** dDoor.table		
bDoor.encPath	**eval** bDoor.encPath		
declId	DeclId.exhVisit		
dDoor.ident	**eval** dDoor.ident		
returnType	returnType.exhVisit		
pDoor.retType	**eval** pDoor.retType		
bDoor	bDoor.exhVisit		
fParam.table	**eval** fParam.table		
fParam	fParam.exhVisit		
b.path	**eval** b.path		
b.table	**eval** b.table		
pDoor.formals	**eval** pDoor.formals		
pDoor	pDoor.exhVisit		
b.retTp	**eval** b.retTp		
b	b.exhVisit		
dDoor.Tp	**eval** dDoor.Tp		
dDoor	dDoor.exhVisit		
syn			

Sort	incSonVisit for fParam	incSonVisit for DeclId	incSonVisit for returnType
inh			
dDoor.table			
bDoor.encPath			
DeclId			
dDoor.ident		**eval** dDoor.ident	
returnType			
pDoor.retType			**eval** pDoor.retType
bDoor			
fParam.table			
fParam			
b.path			
b.table			
pDoor.formals	**eval** pDoor.formals		
pDoor	pDoor.incOwnerVisit		pDoor.incOwnerVisit
b.retTp			
b			
dDoor.Tp			
dDoor		dDoor.incOwnerVisit	
syn			

	incDoorVisit for bDoor	incDoorVisit for dDoor	incSonVisit for b
	—	—	—

Analysis of ProcUseDoor

Step I - *implement conditions*. This step is empty.

Step II - *local analysis*. There are four non-local dependencies:

- The function *lookup* in the equation defining the attribute binding. v(*lookupChanged*) is added to the dependency graph.
- The function *getTp* in the equations defining the attributes tp and formals. v(*getTpChanged*) is added to the dependency graph.
- The reference to *retType* in the equation defining tp adds *v*(*retTypeChanged2*).
- The reference to *formals* in the equation defining formals adds *v*(*formalsChanged*).

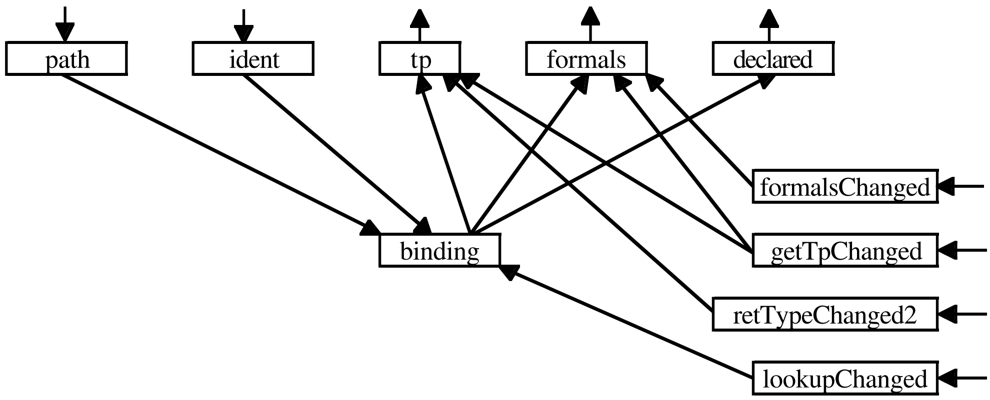


Figure 11.16: Dependency graph for ProcUseDoor after step II.

Step III - *Add send vertices*. Add send vertices for *v*(*RetTypeChanged2*), *v*(*formalsChanged*), *v*(*getTpChanged*), and *v*(*lookupChanged*). The send vertices for *v*(*getTpChanged*) and *v*(*lookupChanged*) are the same as for the simple UseDoor as described in [Hed92] section 9.5.3 and are not covered here.

For *v*(*RetTypeChanged2*), *v*(*formalsChanged*) there are dependencies from binding.getTp when getTp returns an AbstractProcType. This means there are two static dependencies:

(ProcDoor.theProcType.retType, ProcUseDoor.tp) and

(ProcDoor.theProcType.formals, ProcUseDoor.formals).

To find the affected ProcUseDoors we add a dependency function to ProcDoor, this function is implemented in step V.

```
addto ProcDoor {
  fProcUses: func ref Set[ProcUseDoor];
};
```

Two new send vertices, *v*(*RetTypeChanged2*, *ProcUseDoor*, *fProcUses*) and *v*(*formalsChanged*, *ProcUseDoor*, *fProcUses*) are added to ProcDoor.

Step IV - *Simplify send / receive vertices*. The two receive vertices *v*(*RetTypeChanged2*) and *v*(*formalsChanged*) are removed and replaced by *v*(*ProcTypeChanged*).

The two send vertices in ProcDoor, v(*RetTypeChanged2*, *ProcUseDoor*, *fProcUses*) and *v*(*formalsChanged*, *ProcUseDoor*, *fProcUses*) are replaced by *v*(*ProcTypeChanged, ProcUseDoor*, *fProcUses*).

Step V - *Implement dependency functions*. The dependency function fProcUses should return a set containing all ProcUseDoors using the AbstractProcType object. A collection is added to AbstractProcType that keeps track of these ProcUseDoors.

```
addto AbstractProcType {
  collection ProcUses: object
         UnorderedCollection[ProcUseDoor];
};
```

Membership in the ProcUses collection is established by a condition in ProcUseDoor. Local copy attributes are added to ensure that evaluation and de-evaluation of the door uses the same information.

```
addto ProcUseDoor {
  loc localPath: ref SearchPath;
  loc localIdent: string;
  eq localPath :- path;
  eq localIdent :- ident;

  eq binding :-
     if localIdent = "" then nullEntry
     else inspect $E:-localPath.lookup(localIdent)
          when Entry do $E
          otherwise nullEntry;
```

```
  cProcUses : cond inspect $B :- binding.getTp
                  when AbstractProcType do
            $B.ProcUses.contains(this ProcUseDoor)
                  otherwise true;
};
```

Now the dependency functions fProcUses can be implemented:

```
addto ProcDoor {
  impl fProcUses :- theProcType.ProcUses.contents;
};
```

In step V we added a condition to ProcUseDoor so we go back to step I to implement this condition.

Step I - *implement conditions*. The evaluation and de-evaluation procedures for *cProcUses* are implemented by using the add and remove procedures of the collection.

```
addto ProcUseDoor {
  evalCProcUses: proc
  { inspect $B :- binding.getTp
    when AbstractProcType do
      $B.ProcUses.add(this ProcUseDoor)
    end  inspect;
  };

  deEvalCProcUses: proc
  { inspect $B :- binding.getTp
    when AbstractProcType do
      $B.ProcUses.remove(this ProcUseDoor)
    end  inspect;
  };
};
```

Step II - *local analysis*. The condition and the local copy attributes are added to the dependency graph (*cAttempted* and *cUses* are from the analysis of the simple UseDoor):

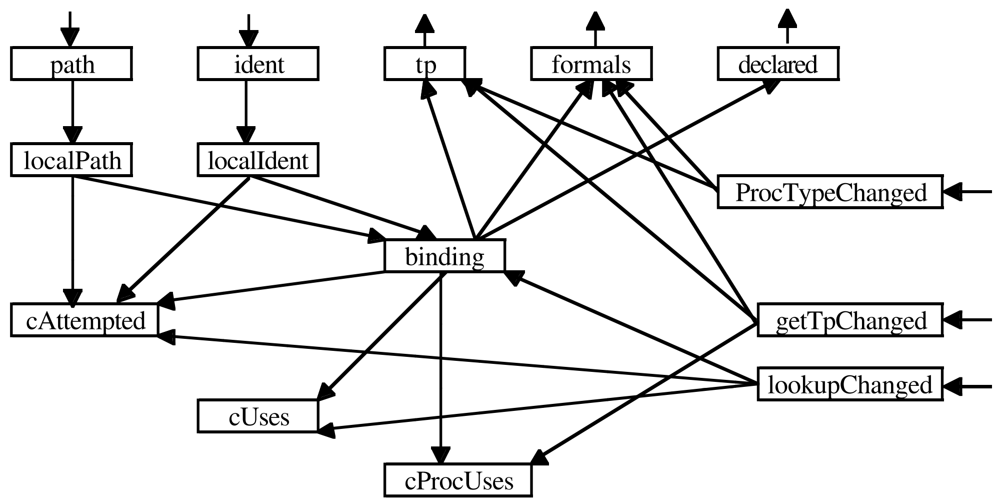


Figure 11.17: Resulting dependency graph for ProcUseDoor.

The dependency graph for ProcUseDoor gives the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	{ v(cUses), v(cAttempted), v(cProcUses) }	Ø	Ø
exhEvalVisit	Ø	{ v(localPath), v(localIdent), v(binding), v(cUses), v(cAttempted), v(cProcUses) }	Ø
incOwnerVisit	{ v(cUses), v(cAttempted), v(cProcUses) }	{ v(localPath), v(localIdent), v(binding), v(cUses), v(cAttempted), v(cProcUses) }	Ø
deEval/Eval getTpChanged	{ v(cProcUses) }	{ v(cProcUses) }	Ø
deEval/Eval lookupChanged	{ v(cUses), v(cAttempted), v(cProcUses) }	{ v(binding), v(cUses), v(cAttempted), v(cProcUses) }	Ø
deEval/Eval ProcTypeChanged	Ø	Ø	Ø

This is implemented as:

```
addto ProcUseDoor {
  impl exhDeEvalVisit {
    (* Step 3. De-evaluate local conditions. *)
    deeval cUses;
    deeval cAttempted;
    deeval cProcUses;
  };

  impl exhEvalVisit {
    (* Step 4. Evaluate local invariants. *)
    eval localPath;
    eval localIdent;
    eval binding;
    eval cUses;
    eval cAttempted;
    eval cProcUses;
  };

  impl incOwnerVisit {
    (* Step 3. De-evaluate local conditions. *)
    deeval cUses;
    deeval cAttempted;
    deeval cProcUses;
    (* Step 4. Evaluate local invariants. *)
    eval localPath;
    eval localIdent;
    eval binding;
    eval cUses;
    eval cAttempted;
    eval cProcUses;
  };

  deEvalGetTpChange proc {
    deeval cProcUses;
  };

  evalGetTpChange proc {
    eval cProcUses;
  };

  deEvalLookupChanged proc {
    (* Step 3. De-evaluate local conditions. *)
    deeval cUses;
    deeval cAttempted;
    deeval cProcUses;
  };
```

```
  evalLookupChanged proc {
    (* Step 4. Evaluate local invariants. *)
    eval binding;
    eval cUses;
    eval cAttempted;
    eval cProcUses;
  };
};
```

Analysis of ProcDoor

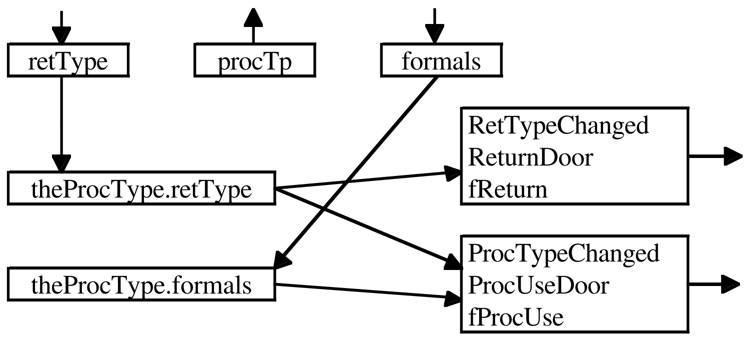


Figure 11.18: Dependency graph for ProcDoor.

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	Ø	Ø	Ø
exhEvalVisit	Ø	{ v(theProcType. formals), v(theProcType. retType) }	{ v(RetTypeChanged,…), v(ProcTypeChanged,…) }
incOwnerVisit	Ø	{ v(theProcType. formals), v(theProcType. retType) }	{ v(RetTypeChanged,…), v(ProcTypeChanged,…) }

Analysis of ReturnStmt

Dependency graph for ReturnStmt:

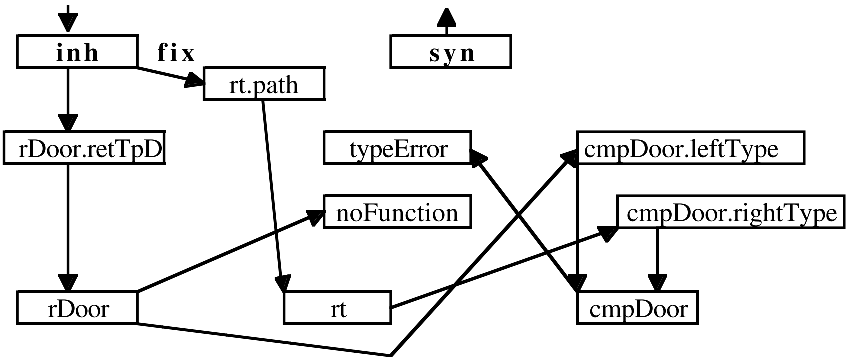


Figure 11.19: Dependency graph for ReturnStmt.

Eval / visit sequences:

Sort	exhVisit	incFatherVisit
inh		
rDoor.retTpD	**eval** rDoor.retTpD	**eval** rDoor.retTpD
rt.path	**eval** rt.path	
rDoor	rDoor.exhVisit	rDoor.incOwnerVisit
rt	rt.exhVisit	
cmpDoor.leftType	**eval** cmpDoor.leftType	**eval** cmpDoor.leftType
cmpDoor.rightType	**eval** cmpDoor.rightType	
cmpDoor	cmpDoor.exhvisit	cmpDoor.incOwnerVisit
typeError	**eval** typeError	**eval** typeError
noFunction	**eval** noFunction	**eval** noFunction
syn		

Sort	incDoorVisit for rDoor	incDoorVisit for cmpDoor	incSonVisit for rt
inh			
rDoor.retTpD			
rt.path			
rDoor			
rt			
cmpDoor.leftType	**eval** cmpDoor.leftType		
cmpDoor.rightType			**eval** cmpDoor.rightType
cmpDoor	cmpDoor.incOwnerVisit		cmpDoor.incOwnerVisit
typeError	**eval** typeError	**eval** typeError	**eval** typeError
noFunction	**eval** noFunction		
syn			

Analysis of ReturnDoor

Step I - *implement conditions*. This step is empty.

Step II - *local analysis*.

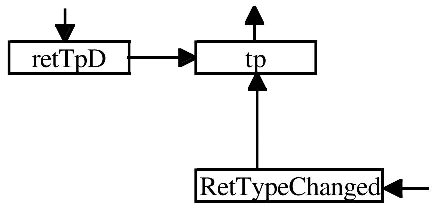


Figure 11.20: Dependency graph for ReturnDoor after step II.

Step III - *Add send vertices*. Add send vertices for *v*(*RetTypeChanged*). Dependency from *retTpD.retType* This means there is a static dependency:

(ProcDoor.theProcType.retType, ReturnDoor.tp).

To find the affected *ReturnDoors* we add a dependency function to ProcDoor, this function is implemented in step V.

```
addto ProcDoor {
  fReturns: func ref Set[ReturnDoor];
};
```

A new send vertex *v*(*RetTypeChanged*, *ReturnDoor*, *fReturns*) is added to ProcDoor.

Step IV - *Simplify send / receive vertices*. Empty step.

Step V - *Implement dependency functions*. The dependency function *fReturns* should return a set containing all ReturnDoors using the AbstractProcType object. A collection is added to AbstractProcType that keeps track of these ReturnDoors.

```
addto AbstractProcType {
  collection RetUses: object
         UnorderedCollection[ReturnDoor];
};
```

Membership in the RetUses collection is established by a condition in ReturnDoor. A local copy attribute, *localRetTpD*, is added to ensure that evaluation and de-evaluation of the door uses the same information.

```
addto ReturnDoor {
  loc localRetTpD: ref AbstractProcType;
  eq localRetTpD :- retTp;
  eq tp :- localRetTpD.retType;
  cReturns: cond localRetTpD.RetUses.contains
                    (this ReturnDoor)
};
```

Now the dependency function *fReturns* can be implemented simply by returning the contents of the *RetUses* collection.

```
addto ProcDoor {
  impl fReturns :- theProcType.retUses.contents;
};
```

In step V we added a condition to *ReturnDoor* so we go back to step I to implement this condition.

Step I - *implement conditions*. The evaluation and de-evaluation procedures for *cReturns* are implemented by using the add and remove procedures of the collection.

```
addto ReturnDoor {
  evalCReturns: proc {
    localRetTpD.retUses.add(this ReturnDoor)
  };

  deEvalCReturns: proc {
    localRetTpD.retUses.remove(this ReturnDoor)
  };
};
```

Step II - *local analysis*. The condition, cReturns, and the local copy attribute, localRetTp, are added to the dependency graph for the *ReturnDoor*.

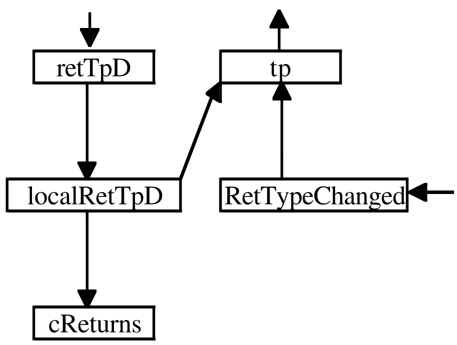


Figure 11.21: Resulting dependency graph for ReturnDoor.

The dependency graph for *ReturnDoor* gives the following characteristic sets:

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	{v(cReturns)}	Ø	Ø
exhEvalVisit	Ø	{v(localRetTpD), v(cReturns)}	Ø
incOwnerVisit	{v(cReturns)}	{v(localRetTpD), v(cReturns)}	Ø
deEval / eval RetTypeChange	Ø	Ø	Ø

This is implemented as:

```
addto ReturnDoor {
  impl exhDeEvalVisit {
    (* Step 3. De-evaluate local conditions. *)
    deeval cReturns;
  };

  impl exhEvalVisit {
    (* Step 4. Evaluate local invariants. *)
    eval localRetTpD;
    eval cReturns;
  };

  impl incOwnerVisit{
    (* Step 3. De-evaluate local conditions. *)
    deeval cReturns;
    (* Step 4. Evaluate local invariants. *)
    eval localRetTpD;
    eval cReturns;
  };

  deEvalRetTypeChange proc {
    (* Empty *)
  };

  evalRetTypeChange proc {
    (* Empty *)
  };
};
```

User Defined Types

This chapter describes a type extension to Ask making it possible to define new types. The type extension we specify is similar to record types found in other languages.

Overview

Ask today supports integer, boolean, class, and procedure types. We define a *complex* type as a named type containing one or more variable declarations. Thus similar to the Pascal record type. Here is an example of such type:

TYPE Point : [**INTEGER** : x, **INTEGER** : y];

This declaration defines a new complex type, *Point*. The variables occurring inside the brackets are called *individuals*, Point contains two individuals, x and y, both having the type integer. The new type can be used in an ordinary variable declaration:

Point : aPoint;

Or encapsulated in another complex type declaration, as follows:

TYPE Line : [Point : p1, Point : p2, **INTEGER** : length];

The Ask language supports operations between complex types, as most languages do:

aPoint := anotherPoint;

This assignment is legal if both operands have the same type, in this case *Point*. In Ask it is also possible to match a complex type with a list of its individuals. Consider the following example:

aPoint := [1, 2];

This is also a legal assignment, the list on the right side is said to have a *anonymous* type. An anonymous type can be legally matched with any complex, or anonymous, type as long as the type of the individuals match.

aPoint := [1, **TRUE**];

This assignment will generate a type mismatch, since the types of the individuals of *Point* is two integers and **TRUE** is considered boolean.

Since an anonymous type can be matched with another anonymous type, the following assignment is legal in Ask:

[aInt, anotherInt, aBool] := [1, 2, anotherBool];

Nested type lists is also possible. Consider the above definition of Line, the following assignment is perfectly legal:

aLine := [[1, 2], aPoint, aInt];

Cyclic Dependencies

The type declaration can, like the class declaration, lead to cyclic dependencies. Consider this erroneous example:

TYPE A : [A : aA];

This example introduce a static semantic error, since the type *A* is defined in terms of itself. In the Ask language the type of the individuals are needed, to fully determine the type of the declared name. When considering the dependencies that this declaration introduces, it is analogous to the following class declaration:

A **CLASS** A;

Cyclic dependencies can occur in several cases with both type and class declarations. These declarations of classes will also yield a cycle:

C **CLASS** A;
A **CLASS** B;
B **CLASS** C;

From the dependency view, the equivalent type declarations are:

TYPE A : [C : aC];
TYPE B : [A : aA];
TYPE C : [B : aB];

C is defined in terms of B that is defined in terms of A that is defined in terms of C. Thus a cyclic dependency has occurred. This latter example of cyclic dependencies can only occur in languages that do *not* require the names to be declared before use. In Pascal the type names used inside a record declaration must be declared earlier. This means that we, in the above example, cannot have a variable of type C, since C is not declared *before* it is used. Note that this not true for the first example, when the type is defined in terms of itself, but this can be considered a special case and therefore much easier to detect.

The Ask classes do not support multiple inheritance, so the dependencies follow a single chain, from the class itself to its super class. This list of ancestors has a cycle, if the lists ends are connected. When dealing with complex types the dependencies do not form a chain. The following declarations demonstrate this:

TYPE A : [**INTEGER** : i];
TYPE B : [**INTEGER** : j];
TYPE C : [A : aA, B : aB];

C is defined in terms of both A and B, i.e. C is dependent on both A and B. So, a type can have several sources that can form a cycle.

Grammar Extensions

The new type construct will lead to a number of new productions. First of all the user has to be able to declare a new type, therefore a new production *TypeDecl* is added to the grammar:

```
<TypeDecl>      ::=  TYPE <ID> : [ <VarDeclList > ]
<VarDeclList>   ::+  <VarDecl> ,

<Decl>          ::|  <NullDecl> | <VarDecl> | <ClassDecl> |
                     <ProcDecl> | <TypeDecl>
```

The *Decl* production is modified to allow type declarations where appropriate.

To allow variable declarations with a complex type, a new production is introduced and the *DeclType* production extended:

```
<UseDeclType>   ::=  <Use>

<DeclType>      ::|  <NullDeclType> | <IntDeclType> |
                         <BoolDeclType> | <RefDeclType> |
                     <UseDeclType>
```

To support the different usages of a complex type, two new productions are added and the *Exp* production modified:

```
<ComplexExp>     ::=  [ <ExpList> ]
<ExpList>        ::+  <Exp> ,

<Exp>            ::|  <NullExp> | <ComplexExp> | <Use> |
                      <RemoteAccess>
```

Understanding the Specification

In this section we will investigate the actual syntax tree for some examples, where the trees are decorated according to the specification given in the later sections. We will try to motivate the specification of the different nodes, in particular the complex specification of the TypeDecl node.

Consider this small program with a rather uncomplicated type declaration:

BEGIN
 TYPE Point : [**INTEGER** : x, **INTEGER** : y];
END;

The type declaration will yield the subtree found in figure 12.1. The complete syntax tree is to large to show even for this small program.

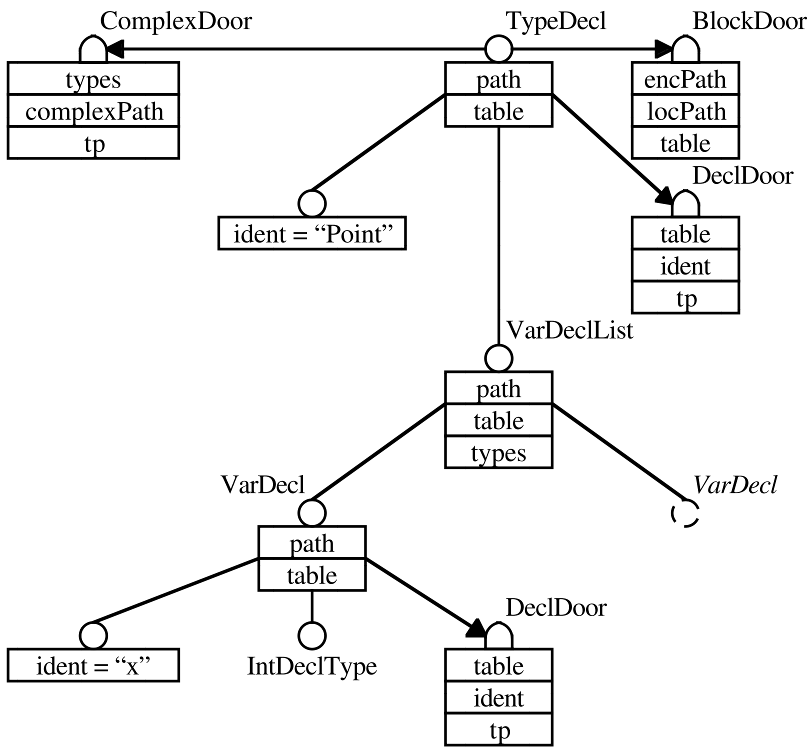


Figure 12.1: Syntax tree example.

The italic VarDecl node on the right in figure 12.1 symbolises an identical sub-tree for *y*, as the VarDecl node for *x*, found on the left. Figure 12.1 reveals that TypeDecl contains three doors. The *BlockDoor* and the *DeclDoor* are known from chapter 8 "The Ask Language" The *ComplexDoor* are specified later in this chapter. TypeDecl, like VarDecl, should introduce a new name, *Point*, into the symbol table of the scope it is declared in. This is achieved through the DeclDoor. The declarations inside a type declaration forms its own scope. The TypeDecl node uses the BlockDoor to register its individuals.

The somewhat complex figure 12.2 shows the information flow, how attributes are propagated in the syntax tree.

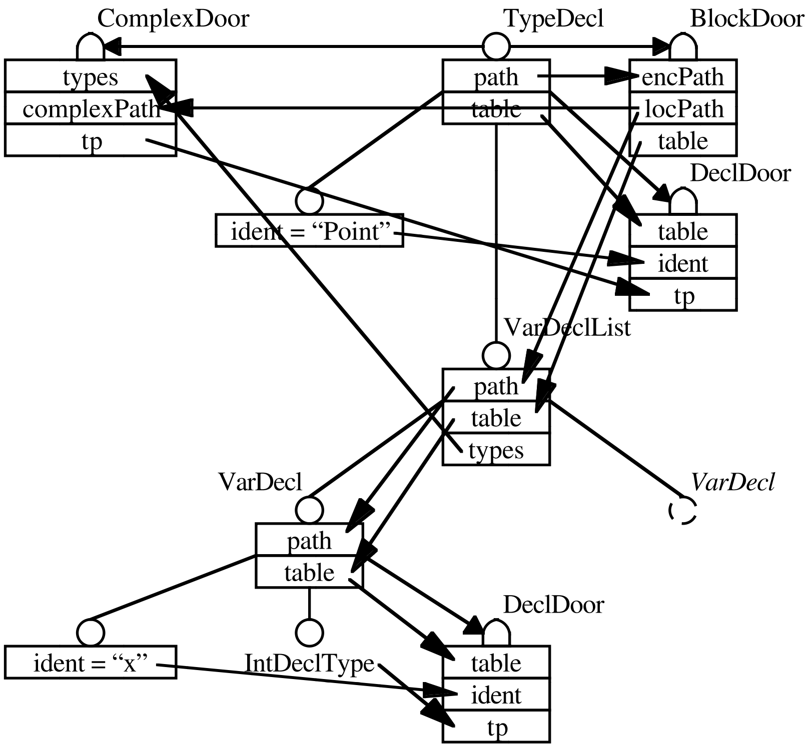


Figure 12.2: Information flow in the example tree.

If we extend our program to include a variable declaration of type Point, we get the following program and abbreviated syntax tree:

> **BEGIN**
> **TYPE** Point : [**INTEGER** : x, **INTEGER** : y];
> Point : aPoint;
> **END**;

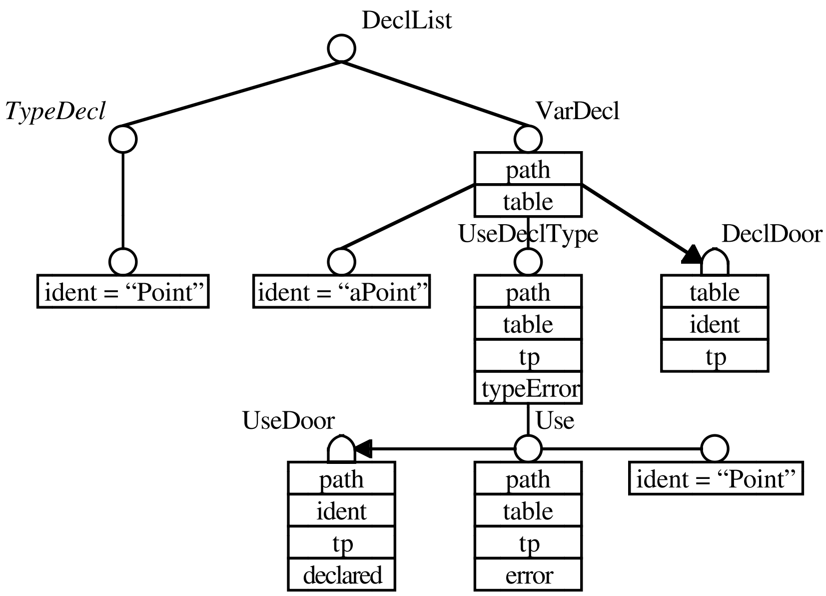


Figure 12.3: Type declaration and usage tree.

The *UseDeclType* node makes it possible to use a type name, but a conflict involving both names and types arises. The identifier *aPoint* is a normal variable that can be used in an assignment. The name *Point* on the contrary should only occur when declaring a variable, as in this example. So there is a difference between the two names and there is a difference in their types. The variable *aPoint* is a complex type, but not of the same type as the type name *Point*. One of the names, Point, is used in declarations and will be of *ComplexDeclType*, while the other is a usage of a type name and will be of *ComplexUseType*.

In the next sections we will specify the new syntactic, semantic, and door classes in OOSL. We will also analyse their dependencies. We start with the extensions of the type hierarchy.

Extending the Type Hierarchy

The extensions to the type system will involve adding a new subtree to the hierarchy. All type classes connected with complex types are gathered under the abstract super class *AbstractComplexType*:

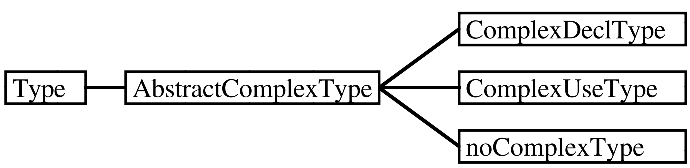


Figure 12.4: Type hierarchy.

Before we go into a detailed specification of these new types, we will examine a utility class *ComplexTypeList*.

Specification of ComplexTypeList

As stated before, Ask needs the types of the individuals to fully determine the type of a declared type name. In practise this means that we maintain a list of the individual types. To maintain an ordered collection of some element, we introduce the interface of class *OrderedCollection*:

```
OrderedCollection: class [T: class ANYCLASS] {
  Append: proc ( e: ref T );
  Head: func ref T;
  SucOf: func ref T ( e: ref Type );
  Empty:  func boolean;
};
```

The type list is modelled through the class, ComplexTypeList. The class contains an ordered collection of types and methods to add and retrieve types from it. A static object to denote the empty list is also needed. This object is called *emptyComplexTypeList* and provide new implementations of the methods found in ComplexTypeList.

```
ComplexTypeList: class {
  itsTypes: OrderedCollection[Type];

  Add: proc ( tp : ref Type ) { itsTypes.Append( tp
); };
  First: func ref Type :- itsTypes.Head;
  Next: func ref Type ( cur : ref Type )
    :- itsTypes.SucOf( cur ) };
};

emptyComplexTypeList: object ComplexTypeList {
  impl Add {};
  impl First :- NONE;
  impl Next :- NONE;
};
```

Specification of AbstractComplexType

AbstractComplexType is a class that define the common behaviour of all complex types. It defines a function for retrieving the types of its individuals, i.e. a type list. The TypeDecl node must define its own local scope where the individuals register. In some cases it is necessary to be able to “open up” this scope, like for instance the with-statement in Pascal. Therefore a function to retrieve the path to that symbol table is included.

```
AbstractComplexType: class Type {
  getTypes: func ref ComplexTypeList;
  getComplexPath: func ref SearchPath fix;
};
```

Specification of ComplexDeclType

As we noted in the section "Understanding the specification" we need two types to fully model the different usages of a complex type object. There is only one occasion where it is semantically legal to access the type name. That is in the declaration of a variable, either as a normal variable or as an individual of a complex type. In order to separate the two cases we have introduced two different kinds of complex types. ComplexDeclType for the declaration of the type name and ComplexUseType for the usage of that type.

The ComplexDeclType and the ComplexUseType are interconnected in a similar way as ClassType and RefType, but we have chosen an alternative specification.

The specification of ComplexDeclType introduces some local attributes and specialises the equations declared in AbstractComplexType. The ComplexDeclType uses a door called ComplexDoor, which we will give a detailed specification of in the section "Introducing ComplexDoor". At this stage we use the ComplexDoor to retrieve the path to the symbol table that encloses the individuals.

```
ComplexDeclType: class AbstractComplexType {
  itsUsage: object ComplexUseType;
  itsTypes: ref ComplexTypeList;
  loc owner: ref ComplexDoor;
  loc anonymous: boolean fix;

  eq itsUsage.owner :- this;

  impl getTypes :- itsTypes;
  impl getComplexPath :- owner.complexPath;
  getComplexType: func ref ComplexUseType fix {
    getComplexType :- itsUsage;
  };
};
```

Specification of ComplexUseType

The ComplexUseType models the usage of a complex type.

```
ComplexUseType: class AbstractComplexType {
  loc owner: ref ComplexDeclType;

  impl getTypes :- owner.getTypes;
  impl getComplexPath:- owner.getComplexPath;
};
```

Specification of noComplexType

The object noComplexType is used instead of a NONE reference. The noComplexType object is propagated to the syntax tree by UseDeclType and ComplexDoor, both described below, when the declaration of a type is incomplete or erroneous. The *CompareDoor* introduced in the chapter "Matching of types" also uses this object to perform type matching between two complex types.

```
noComplexType: object AbstractComplexType {
  impl getTypes :- emptyComplexTypeList;
  impl getComplexPath:- emptyPath;
};
```

Introducing ComplexDoor

The ComplexDoor has the responsibility of assembling a ComplexDeclType from a list of types.

```
ComplexDoor : door {
  inh types: ref ComplexTypeList;
  inh complexPath: ref SearchPath fix;
  inh anonymous: boolean fix;

  syn tp: ref ComplexDeclType;

  theType: object ComplexDeclType;

  eq theType.itsTypes :- types;
  eq theType.anonymous := anonymous;
  eq theType.owner :- this;
  eq tp :- theType;
};
```

Specifications of Syntax Classes

In the extension to the Ask grammar presented earlier in this chapter, five new productions were introduced. In this section we will specify these productions and their attributes and equations.

Specification of TypeDecl

TypeDecl is the node class that models the declaration of a new type. To maintain a list of individuals and to collect their types, it uses a *VarDeclList* that we will present later on.

The sons of TypeDecl is an identifier and a list of variables. The identifier is the new name that will be associated with the type. Earlier in this chapter we made certain conclusions about the behaviour of TypeDecl. We discovered that we needed a DeclDoor to register the new name in a symbol table, a BlockDoor to encapsulate the individuals in a new scope, and a ComplexDoor to maintain the semantic type object.

```
TypeDecl: cons Decl( typeID: ref ID,
                     typeList : ref VarDeclList ) {
  dDoor: doorobject DeclDoor;
  bDoor: doorobject BlockDoor;
  xDoor: doorobject ComplexDoor;

  eq bDoor.encPath :- path;
  eq typeList.table :- bDoor.table;
  eq dDoor.table :- table;
  eq dDoor.ident := typeID.ident;
  eq dDoor.tp :- xDoor.tp;
  eq xDoor.types :- typeList.types;
  eq xDoor.complexPath :- bDoor.locPath;
  eq xDoor.anonymous := false;
};
```

Specification of VarDeclList

As stated before VarDeclList is a list of variable declarations. Except for the list functionality, modelled through the *list* keyword, VarDeclList has a synthesized attribute *types*. ComplexTypeList was presented in the section “Extending the Type Hierarchy”, and maintains a list of type objects. The actual building of the types attribute is specified using attribute construction.

```
VarDeclList : list Descendant( d: ref VarDecl ) {
  syn types: ref ComplexTypeList;

  eq types :- AC $t :=
       ( new ComplexTypeList | $t.add( son.tp ) );
};
```

Specification of UseDeclType

The *UseDeclType* makes it possible to use a type name when declaring a new variable or as a return type for a function. UseDeclType has one son, which is a Use node. The local attribute *typeError* and the equation that define it, are used to assure that the used name is really the name of a type. It is also legal to use a undeclared name, that will be introduced later. The tp attribute is a synthesized attribute inherited from DeclType.

The need for the typeError attribute and the difference between ComplexDeclType and ComplexUseType has been discussed earlier in this chapter. Note that UseDeclType operates on ComplexDeclType, but propagates ComplexUseType to its context, through the call to *GetComplexType*.

```
UseDeclType: cons DeclType( typeUse: ref Use ) {
  loc typeError: Boolean;

  eq typeError := not typeUse.tp in ComplexDeclType
                  and typeUse.tp <> noComplexType;
  eq tp :- if typeUse.tp in ComplexDeclType
             typeUse.tp.GetComplexType;
           else
             noComplexType;
};
```

Specification of ComplexExp

ComplexExp is a simpler version of TypeDecl, used to create an anonymous (nameless) complex type for use as an expression. For example, this class makes it possible to model the following assignment:

aPoint := [1, 2];

The right-hand side of the assignment is clearly an expression. Since the assignment is between two complex types, a new ComplexDeclType and ComplexUseType has to be created. The equations are similar to those of TypeDecl. The son is a list of expressions instead of variable declarations, but since both declarations and expressions have well defined types they are in this sense equal.

```
ComplexExp: cons Exp( expL : ref ExpList ) {
  xDoor: doorobject ComplexDoor;

  eq tp :- xDoor.tp;
  eq xDoor.types :- expL.types;
  eq xDoor.complexPath :- path;
  eq xDoor.anonymous := true;
};
```

Specification of ExpList

ExpList is, as stated above, analogous to VarDeclList. The equation defining types is similar since both declarations and expressions have a tp attribute.

```
ExpList : list Descendant( e: ref Exp ) {
  syn types: ref ComplexTypeList;

  eq types :- AC $t :=
        ( new ComplexTypeList | $t.add( son.tp ) );
};
```

Detecting and Breaking Cyclic Dependencies

To break and detect cyclic dependencies in complex types might at a first glance look simple, but leads to a rather complex solution. As a first step we can consider what happens if we ignore the possibility of cycles. The answer is that in the case of an erroneous Ask program with cyclic types, the evaluator enters an endless loop. This comes as no surprise, but what actually causes the loop?

Well, first we will consider the propagation of the different type objects, *tp* and *types*.

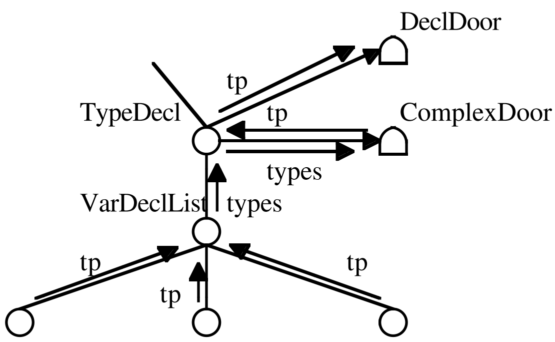


Figure 12.5: Propagation of *tp* and *types* attributes.

VarDeclList collects the different tp attributes into a ComplexTypeList, which is propagated to TypeDecl. TypeDecl passes the types attribute to its ComplexDoor, which composes them into a new tp attribute that is returned to the TypeDecl node. From there the tp attribute is propagated out to the rest of the environment, through the DeclDoor.

If a type declaration contains a cycle, one of the subtrees of VarDeclList must contain a UseDeclNode. One of the propagated tp attributes shown in the figure, must be propagated from that UseDeclNode. The UseDeclNode fetches that tp attribute from another declaration, through a UseDoor. Since we have a cycle, the UseDoor is connected, directly or indirectly, to the TypeDecls DeclDoor. The result is that the tp attribute propagated to the DeclDoor in TypeDecl, is the same tp attribute propagated upwards to the VarDeclList.

The detection of cyclic dependencies is located in the ComplexDoor, by adding a synthesised attribute. The ComplexDoor detects the cycle by stating that a type is defined in terms of all type names that occur among the individuals. The ComplexDeclType maintains a collection of the types that it is defined in terms of.

TYPE T: [T1 : a, T2 : b, T3 : c];

The type T is defined in terms of T1, T2, and T3. By searching the types in the collection for a reference to the defined type, in this case T, we can detect a cycle. Note that if one of the types found in the collection is a usage of a complex type, then the search must continue in that types own collection. In the above example the search would propagate from the search of the collection of T, to the collections of T1, T2, and T3.

We extend the specification of *ComplexDeclType* to include a collection of other *ComplexDeclTypes* that the type is defined in terms of. We also add a collection of objects that causes a cycle in this type declaration. The *causers* collection can be used for error reporting. The interface of the collection can be found in the previous section “Specification of ComplexTypeList”

```
addto ComplexDeclType {
  collection terms: object
      OrderedCollection[ ComplexDeclType ];
  collection causers: object
      OrderedCollection[ ComplexDeclType ];
  definedInTermsOf:
    func Boolean ( tp: ref ComplexDeclType );
};
```

The *ComplexDoor* is extended with a synthesised attribute *causesCycle*. The implementation of the conditions and corresponding evaluation procedures are only sketched below, since they are rather lengthy. The *terms* and *causers* collections are build from the *ComplexDeclTypes* contained in the inherited *types* attribute, which follows the rules in [Hed92], §9.4.3. The ComplexDeclTypes that do not cause a cycle in this type is added to the terms collection, the others are added to the causers collection.

```
addto ComplexDoor {
  syn causesCycle: Boolean;

  eq causesCycle := not theType.causers.Empty;

  cTerms: cond …
  evalCTerms: proc …
  deEvalCTerms: proc …

  cCausers: cond …
  evalCCausers: proc …
  deEvalCCausers: proc …
 };
```

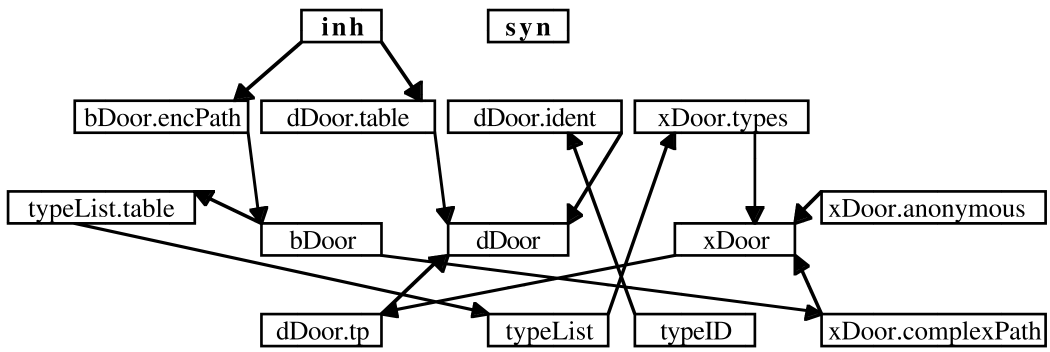
The only missing piece is the *definedInTermsOf* function. This function should check if the passed type, *tp*, is defined in terms of the current declaration, *this*. The declaration is defined in terms of tp, if tp is equal to this or if the declarations collection of types is defined in terms of tp. This is a recursive definition and will cause all affected collections to be scanned.

```
addto ComplexDeclType {
  impl definedInTermsOf :=
    if tp = this then
      true
    else
      loop $t :- terms.first
        if $t = NONE then
          false
        else if $t.definedInTermsOf( tp ) then
          true
        else next $t :- $t.suc;
};
```

Analysis

Analysis for TypeDecl:

Dependency graph for TypeDecl:



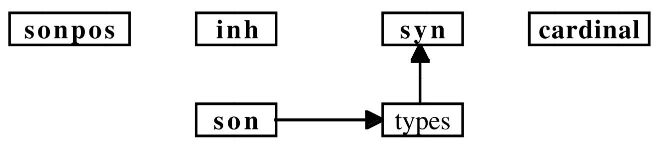
Visit procedures for TypeDecl:

```
addto TypeDecl {
  impl exhVisit {
    eval dDoor.table;
    eval bDoor.encPath;
    typeID.exhVisit;
    eval dDoor.ident;
    bDoor.exhEvalVisit;
    eval xDoor.complexPath;
    eval typeList.table;
    typeList.exhVisit;
    eval xDoor.types;
    eval xDoor.anonymous;
    xDoor.exhEvalVisit;
    eval dDoor.tp;
    dDoor.exhEvalVisit;
  };
```

```
  impl incSonVisit {
    if S == typeID then
      eval dDoor.ident;
      dDoor.exhEvalVisit;
    else if S == typeList then
      eval xDoor.types;
      xDoor.incDoorVisit;
      eval dDoor.tp;
      dDoor.incDoorVisit;
    end if;
  };

  impl incDoorVisit {
    if D == dDoor then
    else if D == bDoor then
      eval xDoor.complexPath;
      eval typeList.table;
      typeList.exhVisit;
      eval xDoor.types;
      xDoor.exhEvalVisit;
      eval dDoor.tp;
      dDoor.exhEvalVisit;
    else if D == xDoor then
      eval dDoor.tp;
      dDoor.incDoorVisit;
    end if;
  };
};
```

Analysis for VarDeclList:

Dependency graph for VarDeclList:



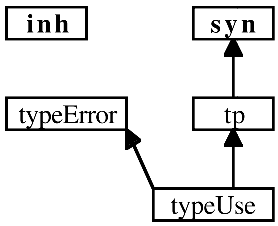
Visit procedures for VarDeclList:

```
addto VarDeclList {
  impl exhVisit {
    ∀ son son.exhVisit;
    eval types;
  };

  impl incSonVisit {
    eval types;
    father.incSonVisit( this );
  };
```

```
    impl incBrotherVisit {
      eval types;
      father.incSonVisit( this );
    };
  };
```

Analysis for UseDeclType:

Dependency graph for UseDeclType:

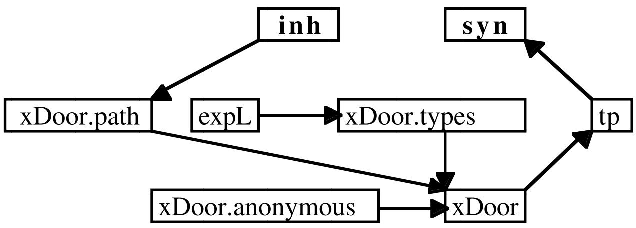


Visit procedure for UseDeclType:

```
  addto UseDeclType {
    impl exhVisit {
      typeUse.exhVisit;
      eval typeError;
      eval tp;
    };

    impl incSonVisit {
      eval typeError;
      eval tp;
      father.incSonVisit( this );
    };
  };
```

Analysis for ComplexExp:

Dependency graph for ComplexExp:

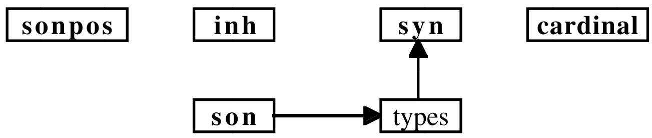


Visit procedures for ComplexExp:

```
addto ComplexExp {
  impl exhVisit {
    expL.exhVisit;
    eval xDoor.path;
    eval xDoor.types;
    eval xDoor.anonymous;
    xDoor.exhEvalVisit;
    eval tp;
  };

  impl incSonVisit {
    eval xDoor.types;
    xDoor.incOwnerVisit;
    eval tp;
    father.incSonVisit( this );
  };

  impl incDoorVisit {
    eval tp;
    father.incSonVisit( this );
  };
};
```

Analysis for ExpList:

Dependency graph for ExpList:

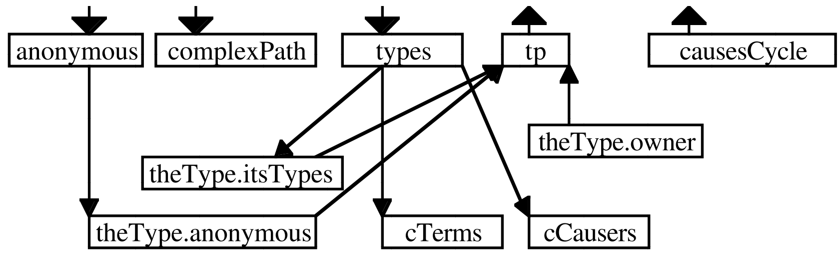


Visit procedures for ExpList:

```
addto ExpList {
  impl exhVisit {
    ∀ son son.exhVisit;
    eval types;
  };

  impl incSonVisit {
    eval types;
    father.incSonVisit( this );
  };

  impl incBrotherVisit {
    eval types;
    father.incSonVisit( this );
  };
};
```

Analysis of ComplexDoor:

Dependency graph for ComplexDoor:



Characteristic set for ComplexDoor:

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	{ **v**(cTerms), **v**(cCausers) }	Ø	Ø
exhEvalVisit	Ø	{ **v**(theType.anonymous), **v**(theType.owner), **v**(theType.itsTypes), **v**(cTerms), **v**(cCausers) }	Ø
incOwnerVisit	{ **v**(cTerms), **v**(cCausers) }	{ **v**(theType.itsTypes), **v**(cTerms), **v**(cCausers) }	Ø

Visit procedures for ComplexDoor:

```
addto ComplexDoor {
  impl exhDeEvalVisit{
    deeval cTerms;          (* 3 *)
    deeval cCausers;        (* 3 *)
  };

  impl exhEvalVisit{
    eval theType.anonymous;  (* 4 *)
    eval theType.owner;      (* 4 *)
    eval theType.itsTypes;   (* 4 *)
    eval cTerms;             (* 4 *)
    eval cCausers;           (* 4 *)
  };
```

```
  impl incOwnerVisit {
    deeval cTerms;           (* 3 *)
    deeval cCausers;         (* 3 *)
    eval theType.itsTypes;   (* 4 *)
    eval cTerms;             (* 4 *)
    eval cCausers;           (* 4 *)
  };
};
```

13 Matching of Types

This chapter addresses the problems involved when two types needs to be match to each other. When matching of two variables is done, the types rather than the actual values of the variables are considered. For example two integers may be matched with each other, while one integer and one boolean is a type mismatch. For scalar types, such as integers and boolean, this is a straightforward task. Complex types and classes call for more elaborate matching.

Overview

Matching of types occurs frequently in computer programs. Before any arithmetics or assignments can be done, the operands need to be matched. In a procedure call the actual parameters need to be matched with the formal parameters. Since matching is so common, a schema to encapsulate the matching for consistency and efficiency is needed. We will also show that matching, in some cases, requires non-local and non-fix information. These requirements points towards a special door object for matching.

As stated above, matching integers and booleans is straightforward. So, let us consider matching of classes. Matching, in almost all cases, is done between two variables. The Ask language supports reference variables to class instances, in the same way as Simula does. When matching classes, we have two reference variables to work with. In the following example, a class *A* and a subclass *B* is assumed:

> **REF**(A) : rA;
> **REF**(B) : rB;
>
> rA := rB;

The variables *rA* and *rB* are obviously of different types, yet is the assignment legal in Ask and so also in many other object oriented languages. This is possible since we regard subclassing as *subtyping*. The class B is a subtype of A. This can also be expressed as B is a specialisation of A. It is common to allow a general type to be assigned a specialised value, since this is always correct. Consider the following declarations and assignment in ADA:

> **subtype** SpecialInteger **is** INTEGER **range** 1..10;
>
> aInt : INTEGER;
> aSpecialInt : SpecialInteger;
>
> aInt := aSpecialInt;

We have declared an interval of integers as a subtype to integer, i.e., a specialisation of integer. The assignment is always legal since a variable of integer type naturally can be assigned any value between 1 and 10.

Reference variables introduce further complications, than the subtypes in ADA. References are allowed to denote an instance that actually is a specialisation of the declared type.

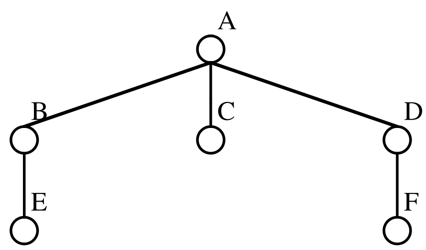


Figure 13.1: Example class hierarchy.

Consider the class hierarchy in figure 13.1 and the Simula example below:

> **ref**(A) rA;
> **ref**(B) rB;
>
> rB :- rA;

The reference *rA* can in fact denote instances of all classes in the hierarchy, while *rB* can only denote instances of class *B* and *E*. As you can see, the assignment is legal in some cases and illegal in others. If *rA* denotes instances of class *B* or *E* the assignment is valid, otherwise it is illegal. The matching of the assignment can not be determined during the static semantic analysis, but must be performed dynamically, i.e., during runtime. Different languages handle this case differently, Simula will cause a runtime error if the assignment is illegal. C++ requires an explicit type conversion by the programmer, i.e., an unchecked guarantee from the programmer that it is really a legal assignment.

Introducing CompareDoor

In order to handle matching of types, we have constructed a door object. The CompareDoor is based upon the CompareClassDoor in [Hed92], page 237. The CompareDoor is a general purpose door which matches any type in the Ask type hierarchy. Any additions to that hierarchy will affect CompareDoor as well. CompareDoor features two inherited type attributes and two synthesized boolean attributes. The boolean attributes represent the cases described above; *inComparable*, the types does not match; *notStaticKnown*, the matching can not be determined at the static semantic analysis.

```
CompareDoor: door {
  inh leftType: ref Type;
  inh rightType: ref Type;
  syn inComparable: boolean;
  syn notStaticKnown: boolean;
};
```

Since this is a general purpose door and there exists many different types in the Ask type hierarchy, we introduce several utility functions:

```
addto ComplexDoor {
  isReference: func boolean ( tp: ref Type ) :=
    inspect $r :- tp
    when AbstractRefType do true;
    otherwise false;

  isComplex: func boolean ( tp: ref Type ) :=
    inspect $r :- tp
    when AbstractComplexType do true;
    otherwise false;

  compareClasses:
    func boolean (gen, spec: ref AbstractClassType):=
      spec.subClassOrEqual( gen );

  compareAnonymous:
    proc boolean (t1, t2: ref AbstractComplexType) {
    ...   This procedure matches the individuals
          of t1 and t2
  };

  compareComplex:
    func boolean (t1, t2: ref AbstractComplexType) :=
    if t1 == noComplexType or t2 == noComplexType
    then true
    else if t1.anonymous or t2.anonymous then
      compareAnonymous( t1, t2 )
    else
      t1 == t2;
```

```
  comparableTypes:
     func boolean ( t1, t2: ref Type ) :=
    if t1 == unknownType or t2 == unknownType then
      true
    else if isReference(t1) and isReference(t2) then
      compareClasses(t1.getClassType, t2.getClassType)
    else if isComplex(t1) and isComplex(t2) then
      compareComplex(t1, t2)
    else
      t1 == t2;
 };
```

With the help of these functions we can easily specify the equations that define the synthesized attributes:

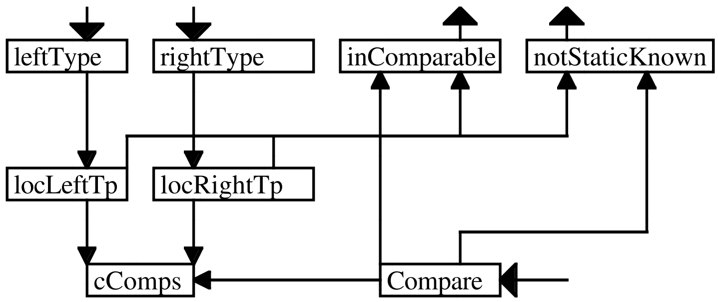
```
addto CompareDoor {
  eq inComparable :=
    not comparableTypes( leftType, rightType );

  eq notStaticKnown :=
    if isReference( leftType ) and
       isReference( rightType ) then
      not compareClasses(  leftType.getClassType,
                           rightType.getClassType )
      and compareClasses(  rightType.getClassType,
                           leftType.getClassType );
    else
      false;
};
```

Analysis

The non-local dependency introduced by calling the *subclassOrEqual* function in *ClassType*, leads to additions to the *CompareDoor*. These additions, which also involve *Type* and *ClassDoor*, are described in [Hed92], page 240. Since *CompareDoor* matches *Type* objects, the collection *comps*, in [Hed92], is added to *Type* instead of *ClassType*.

```
addto CompareDoor {
  loc locLeftTp: ref Type;
  loc locRightTp: ref Type;
  cComps: cond …
 };
```

Dependency graph for CompareDoor:



Characteristic set for CompareDoor:

	S_{cond}	S_{loc}	S_{send}
exhDeEvalVisit	{ cComps }	Ø	Ø
exhEvalVisit	Ø	{ **v**(locLeftTp), **v**(LocRightTp) }	Ø
incOwnerVisit	{ cComps }	{ **v**(locLeftTp), **v**(LocRightTp) }	Ø
deEval / eval Compare	{ cComps }	Ø	Ø

Visit procedures for CompareDoor:

```
addto CompareDoor {
  impl exhDeEvalVisit{
    deeval cComps;       (* 3 *)
  };

  impl exhEvalVisit{
    eval locLeftTp;      (* 4 *)
    eval locRightTp;     (* 4 *)
  };

  impl incOwnerVisit {
    deeval cComps;       (* 3 *)
    eval locLeftTp;      (* 4 *)
    eval locRightTp;     (* 4 *)
  };

  deEvalCompare : proc {
    deeval cComps;       (* 3 *)
  };

  evalCompare : proc {
    eval cComps;         (* 4 *)
  };
};
```

In this chapter we describe some of our experiences of using the BETA language. A complete description of BETA can be found in [MMN93]. [BN93] explains how to translate OOSL constructs to BETA.

Singular Objects

As mentioned in chapter 5, part-objects and static references are widely used in OOSL. These objects can easily be translated to singular objects in BETA. As an example the reference to *InspectDoor*, *iDoor*, in the OOSL-specification of *WithStmt* uses a static reference:

```
WithStmt : cons Stmt
( inspObj: ref Exp, doPart: ref StmtList ) {
  iDoor: doorobject InspectDoor;
  …
}
```

Using a singular object this will in BETA look like:

```
WithStmt: Stmt
(# inspObj: ^Exp; doPart: ^StmtList;
  iDoor: @InspectDoor(# … #);
  …
  enter ( inspObj[], doPart[] )
#);
```

Implementation of Demand Attributes

A very useful feature in BETA is the possibility to add code when a subpattern is declared or when a pattern is used. This feature can be used in the implementation of demand attributes. A demand attribute is normally implemented as a virtual function. When one wants to assign a value to a demand attribute one has to provide a new behaviour for that function. In conventional object-oriented languages this means declaring a subclass, this can lead to a potentially unlimited number of subclasses. BETA however, supports additions of code when declaring a variable. The example below shows the implementation of demand attributes in *CompareDoor*. The door contains two inherited demand attributes:

```
CompareDoor: ANYDOOR
(#
  …
  leftType:<
      (# outType: ^Type; do INNER Exit outType[] #);
  rightType:<
      (# outType: ^Type; do INNER Exit outType[] #);
  …
#);
```

These demand attributes can be specified further when a static reference to a complex door is declared.

```
  cmpDoor: @CompareDoor
    (#  leftType::<(# do lhs.tp -> outType[] #);
        rightType::<(# do rhs.tp -> outType[] #);
  #);
```

This means that the same door class can be used on several places, without having to explicitly declare a subclass.

The qual Pattern

The BETA language is a small language in the sense that it only contains a few built-in mechanisms, these include for-, if-, label-, and jump-imperatives. All other constructs needed can be built using patterns. Some often used constructs are defined in the standard include file *betaenv.bet,* for instance the *ForTo*-, *Cycle*-, and *Loop*-patterns.

One mechanism not included in the language or in betaenv is the qua-operation found in Simula. A simple pattern can be constructed and used where a qua is needed. This pattern will look like:

```
qual: (# T:<Object; r: ^T; enter r[] do inner #);
```

Consider the following use of qua in Simula:

```
class car;;
car class volvo; begin integer model; end;
ref(car) rCar;
…
outInt(rCar qua volvo.model, …);
```

This can be translated to BETA using the qual pattern:

```
car: (##);
volvo: car (# model: @integer; #);
rCar: ^car;
…
rCar[]->qual(# T::<volvo; exit r.model #) -> putint;
```

In the above examples we have a reference to a car (*rCar*) that is assigned an object of type *volvo*. To be able to access the variable *model* found only in the volvo pattern we use the qual mechanism.

The inspect Pattern

Another mechanism not included in BETA is the inspect-statement found in OOSL and Simula. As for the missing qua-operation, in BETA we can define a general inspect pattern and use this where an inspect-statement is needed.

The inspect pattern is defined as:

```
inspect:
  (# T :< Object;
    when:<(# r: ^T; enter r[] do inner #);
    otherwise:<(# do inner #);
    OK:@Boolean; r: ^Object;

  enter r[]
  do (if r[] // NONE then false -> OK;
     else ( r.struc <= T## ) -> OK;
     if);
     (if OK // true then r[] -> when
     else otherwise
     if);
  exit OK
  #);
```

The different parts of the pattern have the following meaning:

- *T* – the virtual class pattern.
- *when* – a virtual pattern that is executed if the inspected object is of type T.
- *otherwise* – a virtual pattern that is executed if the inspected object is not of type T.
- *r* – a reference to the inspected object.
- *OK* – the boolean variable returned by the inspect-pattern, indicating if the inspected pattern is at least of class T.

Using the inspect-pattern:

```
rCar[] ->
inspect(# T::<volvo;
          when::< (# do 'A Volvo!!' -> putText; #);
          otherwise::< (# do 'no!' -> putText; #);
        #);
```

Note that the boolean variable *OK* in the BETA inspect-pattern makes it possible to use the inspect-pattern in checking the actual qualification of a pattern. In the following example the inspect-patterns returns true if *rA* denotes an object that is at least of class *A*.

```
(if rA[] -> inspect(# T::<A #) // true then … if);
```

Data Types

The Mjølner BETA system contains several predefined container classes. We use the list- and set-patterns for all data structures in the Embla system.

Transformations

In BETA each pattern has an enter-part and an exit-part. Combining several patterns so that the exit-part of the first pattern is compatible with the enter-part in the second pattern and so on, gives us what in BETA is called a *transformation*. In the following example the pattern *A* is evaluated its exit-part is assigned to the enter-part in *B*. *B* is evaluated and its exit-part is assigned to the next pattern. This is carried out to the last pattern.

```
A -> B -> … -> Z
```

An example of how this can be used is found in *wmlist.bet*, the interface to the list-pattern in BETA's container library. The involved patterns have the following purposes:

- *at* – returns the position of the entered element.
- *suc* – returns the successor of the entered position.
- *is* – returns the element of the entered position.

```
e[] -> at -> suc -> is -> found[];
```

e is an element and the transformation through *at* gives us the position of *e*, after *suc* we have the next position in the list and after *is* we end up with the element following *e*.

Fragments of a System

In the BETA system the Modularization of programs is accomplished by use of the *fragment language* – a formal way to handle abstract syntax trees and parts of abstract syntax trees located on different files. See [MMN93] for a complete description of the fragment language.

In the Embla system we use the fragment language for normal structuring of code, such as dividing the interface from the implementation. Since the fragment language is more powerful than the modularization mechanisms found in other languages other usages are possible. An example of this can be found in the Embla pattern shown below:

```
ORIGIN '~beta/basiclib/v1.3/betaenv';
BODY 'private/metalevelbody'
[[
--- INCLUDE …
--- lib: attributes ---
Embla:
(#
…
<<SLOT MainLib: attributes >>;
<<SLOT DoorLib: attributes >>;
<<SLOT SemLib: attributes >>;
#);
--]]
```

The three slots will be bound to an actual implementation in the tool using the Embla pattern. *MainLib* will contain the syntactic package, *SemLib* the semantic package, and *DoorLib* the door package.

The file *maingrammar.bet* contains the syntactic package for the Ask language. As seen below all patterns belong to the MainLib attributes slot.

```
ORIGIN '../meta/metalevel';
[[
--- INCLUDE '../doorpackage/doors'
--- INCLUDE '../semantic/types'
--- MainLib: attributes ---
… Pattern declarations for the syntax package
      are made here.
--]]
```

The Embla pattern and the syntactic package for Ask is assembled in the WMEdit tool, by including both files.

```
ORIGIN '~beta/Xt/v1.7/awenv';
[[
--- INCLUDE '../meta/metalevel'
--- INCLUDE '../ask/maingrammar'
--- PROGRAM:  descriptor ----
AwEnv
(# … #)
--]]
```

It should be noted that all patterns in maingrammar.bet are declared in the environment of the MainLib attributes slot in the Embla pattern. This makes it possible to use all patterns and variables declared in the Embla environment.

Pros and Cons

Below is a summary of the particular advantages (+) and disadvantages (-) we found using the BETA language and the Mjølner BETA system for our application:

+ The possibility to add code when declaring or using subpatterns.

+ Good support for X-windows in BETA's X Windows System Libraries.

+ The container classes.

+ The built-in dependency analysis in the BETA compiler.

- In the used implementation (release 2.4, December 1992) of the Mjølner BETA system the attributes fragment only allows declaration of patterns, i.e., variable declaration is not supported. This is a major constraint making good modularization of programs harder to accomplish.

- The fragment system is hard to learn.

15 Conclusions

This Work

The conclusions and experiences gained from this project can be divided into four categories. The first category is founded upon the specification and implementation of the Embla system and the Ask language and thus represent the concrete conclusions. The second category is the experiences drawn from the methods and systems we have been using to achieve our goals. The third category is a reflection on how we approached this project as a whole, how we have disposed our time and resources. The fourth category covers our viewpoints on incremental compilation and incremental semantic analysis.

The Systems

We have, in this document, gone to a great length in stating guidelines for implementing the syntactic, semantic, and door package. These guidelines are based upon the experiences gained from the actual implementation. This means that the current implementation has some weaknesses in regards to the guidelines. There is still work to be done in separating the packages and their independent and specific parts.

The integration of the syntax directed editor Sif, found in the Mjølner BETA metaprogramming system, was one of the original goals that we have not fulfilled. The structure and organization of the editor is at this point poorly documented. The interface of Embla was, however, based upon our own research on Sif. So, a future integration should be straight-forward.

The Methods

The BETA programming language is a simple, yet powerful, language. But the implementation we have worked with, release 2.5, does not cover the whole language and has some annoying limitations. The performance of the compiler is slow. The debugger has a tendency to crash. The limitations in the implementation of the fragment language has caused parts of the project to grow to unreasonable sizes. On the positive side we place the good support for window systems, specially the Athena widgets, which were used for the WMEdit program. On the whole we are positive, but look forward to a faster, smaller, more complete, and more stable implementation. We agree with the following quote from Dr. Dobb's Journal:

> "Keep in mind that these minor complaints are about the implementation, not the language. Beta springs from a unified, object-oriented theoretical foundation. The result is a very clean implementation of virtually all recognised and desired object-oriented capabilities in a simple, easy-to-understand language available in commercial-grade tools. Beta is truly an evolution of object-oriented technology." [Man93]

The incremental evaluation of attributes is a complex process. A change in the extended syntax tree propagates not only upwards and downwards, but also through doors to other branches of the tree. The evaluation spreads through the tree by dependencies. The dependencies can be a father-son relationship in the syntax tree, but also through nodes registered in some collection. Naturally, when developing an incremental environment, errors occur. The debugging of the evaluation phase requires an in-depth knowledge of the specification, i.e. what should really happen during the evaluation, which nodes should be visited. When the attributes are demand-driven yet another dimension is added. We consider these complexities to be the most negative aspect of this area. Traditional compilers are difficult to construct, incremental compilers magnifies this complexity. Therefore it is essential to develop methods to aid in the specification and implementation of incremental compilers.

Door attribute grammars is a step in this direction. By using automatic methods for some parts of the specification and manual, but systematic, methods for others, an incremental evaluator for the language can be developed. We feel that this development cycle is too complex at this stage. More automatic support would be a large improvement. Further research in this area is needed.

The Object-Oriented Specification Language, OOSL, is a language used for Door attribute grammar specifications. While specifying the Ask language we have had problems in obtaining a correct specification. Since no compiler or checker exists for OOSL, the only means of debugging is consulting the chapters defining OOSL in [Hed92].

The Approach

This section covers the fundamental aspects of this project. It touches the crucial decisions made about tools, methods, and management. Note that the viewpoints in this section is in regards to the fulfilment of the original goals.

We argue that using the BETA programming language has both advantages and disadvantages. Although BETA has been a new and exciting experience, choosing a not fully mature environment has its side effects. The nature of the language and the state of the implementation has caused us delays in achieving the goals. Another disadvantage was that the testbench was written in Simula and a large proportion of our work was devoted to move it to BETA.

Another issue to contemplate is the decision to use the testbench instead of building a system from scratch. There exists no easy answer to this issue. We feel that if we had not started with the testbench, we would at an earlier stage been forced to an understanding of how our system should work. This would have been a benefit in later stages in the project. One could also argue the other way around and say that starting from the testbench helped us focus on more complex issues. A software engineering approach, with an analysis, design and implementation phase, would probably have resulted in a better system.

The Subject

Despite the complexity of incremental compilers, they provide a unique tool. The advantages that they bring are without doubt so strong that they will outmatch the traditional compilers in the near future. During the course of this project we have seen some incremental environments reach the commercial market.

The term *incremental* is not crystal clear and the size of an increment can, and probably will, vary among environments. We will most surely see a lot of incremental compilers in the coming years, where the usage of the term incremental could be discussed. By choosing this area for our project, we have gained unique knowledge and experience in a field that we feel will be the dominating subject for compiler development in the nineties.

Future Work

As always when completing a larger work all the things not done spring into mind. This work is no exception. With the knowledge and experience gained by this project, the only right thing would be to start all over. As this is not possible we can give some hints on how to continue and improve the work done.

Improvements in Embla

The Embla metasystem is a large and complex system and several improvements and additions can be made. Below, we list some of the more important:

- The syntactic package should be divided into smaller modules. Today it contains several thousand lines of source code and can be difficult to maintain. This is a concern only as long as the package is hand-coded. A way to solve this problem is thus to develop a generator for the syntactic part.

- Complete the work of integrating the Sif-editor with Embla.

- Introduce caching of demand attributes.

- Last and least it would be nice if someone could find the last bug in Embla (we have spent a year looking for it).

Further Investigations

Besides all the changes in Embla, further investigations in different areas are needed, here are some:

- Investigate and develop a generic door package. Can a generator for the door package or part of it be developed?

- To measure performance of the Embla system.

- To write a generator for the syntactic package. As stated before, the syntactic nodes do not contain any non-local dependencies and it is therefore possible to generate all patterns in this package automatically.

- Investigate if and how Embla can be changed to simultaneously support more than one language. This will be a useful feature when extending the metasystem to a laboratory system for language specifications using Door AG.

OOSL Compiler

During this work we found it very time consuming to have to implement all constructs first in OOSL and then by hand translate them into BETA. This leads to long turn-around time and to an unwillingness to test different solutions. An OOSL compiler or translator is strongly needed. This could either be a traditional compiler generating object code or a translator generating BETA patterns as described in [BN93]. As a first step a syntax checker for the OOSL-language could be developed. This will be a great help when writing specifications in OOSL.

Additions to Ask

Even after our additions, Ask is still a small language. Many interesting constructs could still be added, such as the BETA pattern, parametrized classes, the let-expression in OOSL, the OOSL addto-mechanism, and others.

A Final Comment

During the course of this work we have come to get a good insight in formal specification using Door attribute grammar, the inner secrets of some language constructs, and how to structure and build a metasystem. Besides the subjects directly involved in the work, we have gained much additional knowledge such as object-oriented programming techniques, the BETA programming language, how to draw pictures in Microsoft Excel, using the BETA-debugger, programming in X-Windows, how to proof-read a document, converting documents from IBM-PC to Macintosh, egrep in UNIX, emacs, writing short sentences in English…

Acknowledgements

We would like to thank Görel Hedin for reading and commenting on numerous earlier drafts of this report, and for taking the time to answer all our, sometimes not too intelligent, questions.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[BN93] Werner Burghard and Magnus Nilsson. Translation of OOSL constructs to Simula and BETA. Technical Report LU-CS-IR:93-3, Lund University, Dept. of Computer Science, Lund, Sweden, 1993.

[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.

[Grim89] Ralph P. Grimaldi. Discrete and Combinatorial Mathematics an Applied Introduction (2nd Edition). Addison-Wesley, Reading, Mass., 1989.

[Hed92] G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Lund University, Lund, Sweden, 1992. Tech. Rep. LUTEDX/(TECS-1003)/1-276/(1992).

[Hed93] G. Hedin. Adding list constructions to OOSL. Internal note, Lund University, Lund, Sweden, June 1993.

[Hed94] G. Hedin. An Overview of Door Attribute Grammars. In: P. Fritzon, editor, *Proceedings of CC'94 International Conference on Compiler Construction*, Edinburgh, April 1994, LNCS 786, Springer-Verlag.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass., 1979.

[Hud87] S. E. Hudson. Implementing a User Interface as a System of Attributes. In *ACM SIGSOFT/SIGPLAN*, January 1987. ACM. SIGPLAN Notices vol. 22 no 1.

[ISAh92] Görel Hedin. Lecture notes from a doctoral seminar in incremental semantic analysis, Lund University, Dept. of Computer Science, Lund, Sweden, 1992.

[Kas80] Uwe Kastens. Ordered Attribute Grammars. *Acta Informatica,* 13:229-256, 1980.

[KLMM93] J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson, editors. *Object-Oriented Environments. The Mjølner Approach*. The Object-Oriented Series. Prentice Hall, 1993.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968, with corrections in *Mathematical Systems Theory*, vol. 5, No 1, page 95-96.

[Kos91] Kai 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.

[Man93] Steve Mann. The Beta Programming Language. In *Dr. Dobb's Journal #206, Beyond C++*, October 1993.

[MMN93] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[MPS92] *The Mjølner BETA System Metaprogramming System Reference manual*. Mjølner Informatics Report, MIA 91-14(1.0), November 1992.

[Ray86] V. J. Rayward-Smith. *A First Course in Computability*. Blackwell scientific publications, 1986.

[Rep84] Thomas W. Reps. *Generating Language-Based Environments*. MIT Press, Cambridge, Ma., 1984.

[Web92] *New Webster's Dictionary and Thesaurus of the English Language*. Lexicon Publication, Inc., Danbury, CT, 1992.

[Wir71] N. Wirth. The Programming Language Pascal. In *Acta Informatica* 1:35-63, 1971.

Appendix A: CFG for the Ask Language

<table>
<tr><td><Main></td><td>::|</td><td><Root> | <Descendant></td></tr>
<tr><td><Root></td><td>::|</td><td><Program></td></tr>
<tr><td><Descendant></td><td>::|</td><td><Block> | <Decl> | <DeclType> | <Stmt> |
<Exp> | <ActualParam></td></tr>
<tr><td><Program></td><td>::=</td><td><BlockStmt></td></tr>
<tr><td><Block></td><td>::=</td><td>BEGIN <DeclList> <StmtList> END</td></tr>
<tr><td><DeclList></td><td>::+</td><td><Decl> ;</td></tr>
<tr><td><StmtList></td><td>::+</td><td><Stmt> ;</td></tr>
<tr><td><Decl></td><td>::|</td><td><NullDecl> | <VarDecl> | <ClassDecl> |
<ProcDecl> | <TypeDecl></td></tr>
<tr><td><NullDecl></td><td>::=</td><td></td></tr>
<tr><td><VarDecl></td><td>::=</td><td><DeclType> := <ID></td></tr>
<tr><td><ClassDecl></td><td>::=</td><td><Use> CLASS <ID> <Block></td></tr>
<tr><td><TypeDecl></td><td>::=</td><td>TYPE <ID> : [<VarDeclList>]</td></tr>
<tr><td><VarDeclList></td><td>::+</td><td><VarDecl> ,</td></tr>
<tr><td><DeclType></td><td>::|</td><td><NullDeclType> | <IntDeclType> |
<BoolDeclType> | <RefDeclType> |
<UseDeclType> | <noDeclType></td></tr>
<tr><td><ProcDecl></td><td>::=</td><td><DeclType> PROCEDURE <ID>
(<FormalParamList>) <Block></td></tr>
<tr><td><FormalParamList></td><td>::+</td><td><VarDecl> ,</td></tr>
<tr><td><NullDeclType></td><td>::=</td><td></td></tr>
<tr><td><IntDeclType></td><td>::=</td><td>INTEGER</td></tr>
<tr><td><BoolDeclType></td><td>::=</td><td>BOOLEAN</td></tr>
<tr><td><RefDeclType></td><td>::=</td><td>REF (<Use>)</td></tr>
<tr><td><UseDeclType></td><td>::=</td><td><Use></td></tr>
<tr><td><IntType></td><td>::=</td><td></td></tr>
<tr><td><BoolType></td><td>::=</td><td></td></tr>
</table>

<Stmt> ::| <NullStmt> | <BlockStmt> | <AssignStmt> |
<IfStmt> | <WhileStmt> | <WithStmt> |
<ProcCall> | <ReturnStmt>

<NullStmt> ::=
<BlockStmt> ::= <Block>
<AssignStmt> ::= <Exp> := <Exp>
<IfStmt> ::= **IF** <Exp> **THEN** <StmtList>
ELSE <StmtList> **END IF**
<WhileStmt> ::= **WHILE** <Exp> **DO** <StmtList>
END WHILE
<WithStmt> ::= **WITH** <Exp> **DO** <StmtList> **END WITH**
<ProcCall> ::= <ProcUse>
<ReturnStmt> ::= <Exp>

<Exp> ::| <NullExp> | <ComplexExp> | <Use> |
<RemoteAccess> | <ProcUse>

<NullExp> ::=
<RemoteAccess> ::= <Exp> **.** <Exp>
<Use> ::| <ID>
<ProcUse> ::= <ID> **(** <ActualParamList> **)**

<ActualParamList> ::+ <ActualParam> **,**
<ActualParam> ::= <Exp>

<ComplexExp> ::= **[** <ExpList> **]**

<ExpList> ::+ <Exp> **,**