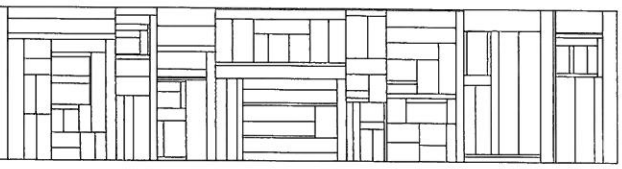
ISSN 0105-8517

Transformations and Abstract Presentations in a Language Development Environment

Kurt Nørmark

DAIMI PB – 222
February 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 83 55 *Telex: 64767 aausci dk*



Contents

Abstract

Muir is a syntax-directed editing environment whose primary purpose it is to support the development of artificial languages. The focus of this thesis is twofold. First, it is demonstrated how simple, pattern-based transformations can be used to improve the functionality of the environment. Second, and somehow orthogonal to this, it is shown how abstract and systematically produced screen presentations can improve user interface of the environment.

For the syntactic description of artificial languages, a new formalism for definition of context free syntax has been developed. The formalism is a variant of the grammar definition formalisms based on operators and phyla. To model the specialization/generalization hierarchy of syntactic domains, the phyla are organized in a directed acyclic graph. The phylum hierarchies of multiple languages are connected into a single-rooted graph structure.

A simple tool that automatically solves the majority of a task seems to be an interesting alternative to a complicated and fully automatic tool. A semi-automatic facility that helps keep documents updated when the underlying grammar is modified has been designed and implemented in Muir. As part of a grammar modification the tool creates a transformation template, which typically must be refined manually. The resulting transformation can be used to update instances of newly modified grammatical constructs in the documents that depend on the grammar.

A semi-automatic tool has also been created to ease the conversion of documents from one formalism to documents in another formalism. A relation among the syntactic domains in the source language and the target language is the fundamental basis of this tool. Some manual work should be expected during the creation of the transformations, and during the application of the transformations on the source documents. However—depending on the conceptual difference between the languages—the majority of the work can be done automatically. The multi-formalism transformation facility has been tried out on a pair of programming languages. The transformation framework has finally been used for the implementation and interpretation of nearly all structure-oriented edit operations in the environment.

The effectiveness of the interactive manipulation of grammars, programs, and other documents in an environment like Muir depends crit-

ically on the way these documents are presented on the screen. In the same way as textual presentations can be generated systematically via grammar-related presentation rules it is demonstrated that more abstract presentations can be defined in similar ways. Two particular presentation formalisms are introduced, one for showing the overall compositional structure of a document, and one for showing selected, transverse relationships on graph form. The two presentation formalisms have both been implemented. Because the setting is a structure-oriented editing environment, it is also discussed how documents can be edited and otherwise manipulated through such abstract presentations.

Dansk Resumé

Emnet for denne licentiatafhandling er transformationer og abstrakte præsentationer i et sprogudviklingssystem. Et sprogudviklingssystem er et specialiseret editeringssystem, der understøtter udviklingen af nye formelle sprog, for eksempel programmeringssprog. I afhandlingen beskriver vi et sådant system, som hedder Muir. Hovedvægten er centreret omkring anvendelsen af transformationer og generering af overordnede, abstrakte skærmpræsentationer. De fleste af emnerne, som beskrives i afhandlingen, er ligeledes relevante for syntaks-dirigerede editeringsomgivelser, d.v.s. editorer der opbygger et dokument af grammatiske byggeklodser i modsætning til enkelt-tegn.

Indledningsvis introduceres en ny formalisme til definition af kontekstfrie grammatikker. I denne nye formalisme, som kaldes hierarkiske grammatikker, beskrives hierarkiet af syntaktiske domæner som et generaliserings/specialiserings hierarki i stil med klassehierarkier, som bl.a. kendes fra programmeringssprogene Simula og Smalltalk. I Muir behandles der typisk flere sprog ad gangen. Begrebsmæssigt modelleres dette ved at hierarkierne, der syntaktisk set definerer disse sprog, er sammenkoblet i ét hierarki. Ligeledes tillades tværgående relationer mellem hierarkierne, hvilket betyder, at det er muligt at håndtere multi-formalisme dokumenter.

Internt i Muir repræsenteres grammatikker og dokumenter som abstrakte syntakstræer. Til manipulation af sådanne abstrakte syntakstræer anvendes der transformationer. En transformation lokaliserer en række instanser af et givet mønster i et abstrakt syntakstræ, og disse instanser udskiftes med kopier af et andet træ, hvori delstrukturer af de fundne instanser kan indgå. Det demonstreres, hvordan de fleste editeringsoperationer i en syntaks-dirigeret editor kan forstås og implementeres som transformationer. Ligeledes vises det, hvordan sådanne editeringsoperationer kan tilknyttes "punkter" i grammatikhierarkiet, og derved defineres på givne syntaktiske konstruktioner.

I et sprogudviklingssystem må det forudses, at der relativt ofte ændres på de sprog, der defineres. Efter sådanne ændringer kan der opstå uoverensstemmelser mellem allerede eksisterende dokumenter og grammatikken, der definerer dokumenterne. I afhandlingen beskrives, hvorledes transformationer kan anvendes til at opdatere de allerede eksisterende

dokumenter. Grundideen er at generere en eller flere transformationer for hver enkelt grammatikændring. Systemet kan kun i de færreste tilfælde generere disse helt automatisk. Typisk må brugeren, der ændrer en grammatik, også "afpudse" den transformationsskabelon, som systemet har skabt. Når dette er sket, kan systemet principielt opdatere dokumenterne, så de stemmer overens med den nuværende grammatik.

Konvertering mellem forskellige, men beslægtede sprog er et andet typisk anvendelsesområde for transformationer. Vi beskriver en facilitet, som tillader direkte transformation af abstrakte syntakstræer i én formalisme til abstrakte syntakstræer i en anden formalisme. Systemet er bl.a. interessant derved, at en meget væsentlig del af transformationerne kan genereres automatisk af Muir. Grundlaget for dette er en relation mellem de syntaktiske elementer i kilde- og målsprogene. Denne relation, såvel som de hierarkiske grammatikker for de to sprog, skal naturligvis defineres manuelt. En given mængde af transformationer kan anvendes på et kildedokument, hvorved der typisk produceres et dokument, i hvilket der stadig findes nogle uoversatte konstruktioner. Disse uoversatte konstruktioner kan elimineres manuelt ved hjælp af normale editeringsoperationer.

Som det sidste emne i afhandlingen beskrives der teknikker til definition af såkaldte abstrakte præsentationer. I et system, hvor grammatikker og dokumenter er repræsenteret som abstrakte syntakstræer, må disse træer på en eller anden måde afbildes på skærmen. Traditionelt producerer sådanne afbildninger et detaljeret, tekstuelt billede. En abstrakt præsentation resulterer derimod i et mere overordnet billede af dokumentet, som ikke nødvendigvis er tekstuelt. Vi beskriver to generelle præsentationsteknikker, en som er velegnet til at producere et træ, der beskriver den overordnede opbygning af et dokument, og en anden som tillader præsentation af mere tværgående sammenhænge i et dokument. Fælles for disse to præsentationsteknikker er, at de beskrives af regler, der tilknyttes de grammatiske elementer af sproget. Abstrakte præsentationer kan bl.a. anvendes til at vise en hierarkisk grammatik som en graf, hvis knuder repræsenterer syntaktiske domæner. Det diskuteres også i afhandlingen, hvordan det underliggende dokument, som er repræsenteret som et abstrakt syntakstræ, kan modificeres gennem en sådan grafpræsentation.

Acknowledgements

I am grateful to Terry Winograd for hosting me for two years in the *System Development Language Group* (SDLG) at *the Center for the Study of Language and Information* (CSLI) at Stanford University. Terry's competent guidance and support have been extremely valuable for this work, and beyond.

I am also grateful to Ole Lehrmann Madsen for his help before and in the course of the work on the thesis. Without his support, this thesis would never have been written.

Furthermore, I want to thank the people who are working, and who have been working in the SDLG group at Stanford. They are Raul Duran, Bradley Hartfield, Olaf Henjum, Mary Holstege, Birgit Landgrebe, Greg Nuyens, Liam Peyton, and Kaizhi Yue.

Also thanks to Brian Mayoh, Peter Mosses, and Jørgen Lindskov Knudsen, all at the *Computer Science Department* at Aarhus University, for comments on the thesis.

This research has received support from a grant by the System Development Foundation to CSLI, which provided the computing facilities for this research. Without this support from CSLI it would not have been possible for me to carry through this work.

Chapter 1

Introduction

The present thesis is the result of a two and a half years Ph.D. (licentiat) study at Aarhus University, Denmark. The study started in July 1984, and it was completed in January 1987. Two of the years, however, were spent at Stanford University in California, and nearly all the research reported in the thesis has been carried out at the *Center for the Study of Language and Information* (CSLI) at Stanford University.

The general topic that has been studied is aspects of structure-oriented editing environments. An *editor* is a tool in a computer system that allows some kind of information to be created and modified. The most common kind of information is text, but editors for information such as technical drawings, VLSI designs, Petri nets, etc. are also viable. An editor may be a separate tool, or it may be applied in connection with other tools. An *editing environment* includes the functionality of an editing tool, but it also allows people to carry out a wider application entirely inside the system. In this thesis, we will especially focus on an editing environment for language development.

We are only interested in editing of documents that belong to *artificial languages* such as programming languages, specification languages, grammar definition languages, and other kinds of formal notation. Traditionally, such documents have been considered as text strings that satisfy a given set of constraints. Consequently, they have been—and are still to a large extent—edited by the same kind of text editors that are used for editing of natural language documents. However, we prefer to look at documents such as programs, specifications, and grammars as structures, whose surface form very well may be textual, but whose primary representation reflect their inherent structure, first of all their basic composition. In a *structure-oriented editor*, the primary document represen-

tation therefore models this structure, and documents can be created, modified and otherwise manipulated in terms of their structure. If, in addition, the documents are constrained by a formal grammar, some of the editing primitives may be derived from the grammar, and grammatical constructs may be referred to from the documents. In this case, we say that the editor is *syntax-directed.* Many different structures can be, and have been used to represent documents in structure-oriented editors. Because we mainly are interested in the compositional structure of the document, as opposed to structures that make explicit more complicated and transverse relationships, we will restrict ourselves to study only tree-structured representations.

Although structure-oriented editing has been known for a couple of decades, the use of structure-oriented editors is very limited outside universities and research labs. The main reasons are undoubtedly that (1) it is hard to break old editing habits, (2) the performance of existing tools is too bad and their capacity is too small, and finally (3) the potential of the technology has not yet been fully explored. Seen from our perspective, time must resolve the first two points. The general goal of this thesis work has been to contribute to the third point. I.e., we hope that the results of this thesis will help convince people that a more structure-oriented approach to editing of formal language documents is superior to purely textual approaches. More specifically, we will demonstrate how more "power" can be added to a structure-oriented editing environment by incorporating a pattern-replacement based transformation framework into the environment. We will focus on three major applications of the transformation framework: Implementation of structure-oriented editing operations, a technique to keep documents updated w.r.t. to a grammar that is under development, and a transformation facility among different—although similar—formalisms. Besides this, we will show how abstract and high level screen presentation techniques can improve a structure-oriented editing environment. In an abstract presentation, selected aspects of a document are projected onto the screen while other aspects are left out of consideration in the presentation.

The thoughts and ideas in this thesis have been implemented and make up the kernel of a system we call Muir. Muir is a *language development environment*, i.e., an editing environment that supports development of new artificial languages and documents belonging to these languages. The language development aspects of Muir are stressed in [70].

It is also reasonable to consider Muir as a contemporary *syntax-directed editing environment* that in particular emphasizes creation of grammars, and which in various ways supports the relationships between grammars and dependent documents. All the proposed facilities are grammatically general. I.e., they are not directed towards specific languages or formalisms.

Our basic orientation is to provide tools and support facilities, not to solve the user's problems by automatic means. Moreover, our work is primarily directed towards support of experienced and skilled users. When possible, we seek to automate frequently occurring routine tasks. More specialized and intellectually demanding tasks should in our opinion be carried out by the user in interaction with the system. To support both automatic facilities and manual routines it seems to be attractive to integrate the automatic facilities into an interactive editing environment. The borderline between the routine tasks and more intellectual tasks is of course fuzzy, and it will undoubtedly move over time. Our orientation is different from so-called automatic programming [6], where also more intellectually demanding tasks are attempted to be automated.

The contents of this thesis is affected by the author's previous work in the field. After these words, the author's background w.r.t. syntax-directed editors will therefore briefly be described. Next, a couple of methodological remarks are given. Following this, we outline the contents of the rest of the thesis. Finally, the main results are summarized.

1.1 The Author's Background

My interest in syntax-directed editing started back in 1980 at Aarhus University, Denmark, where I as a student—together with three other people—designed and implemented a syntax-directed editor called Eagle [67]. Eagle was inspired by a similar locally developed system called Treed [13], which in turn was inspired by Hansen's Emily editor [39,40]. The purpose of Eagle was to make a syntax-directed editor available for experiments on our Dec-10 system via ordinary terminals. (Treed was—as Emily—designed for a vector-oriented graphic screen.) In that respect we succeeded, but Eagle never reached a state where it was realistic to use it for practical program editing. One of the reasons was poor performance characteristics on the time-shared Dec-10 system. So when the

computer science department in Aarhus received its first powerful personal workstation in 1982, an ICL Perq, it was evident that it should be possible to design and implement a better system based on the new equipment. At the same time I was looking for a masters thesis area together with Karen Borup and Elmer Sandvad. It was not hard to decide that we would design a program development system that took advantage of the facilities on a modern graphical workstation. The result of these efforts was EKKO [10,69], a design of a program development system for a "Pascal-like" language. As a part time programmer in the department, I implemented the syntax-directed editor-part of EKKO on the Perq. EKKO was clearly better than Eagle, but it was also apparent that it needed additional design/performance improvements to be a realistic tool, most noticeably related to screen updating when working with big programs. Besides demonstrating that syntax-directed editing is feasible when implemented on proper hardware, EKKO was used to carry out a minor experiment, in which four people used the system one week each to solve small programming tasks. By August 1984 I entered a Ph.D. study at Aarhus University, and I was lucky enough to be able to spend two years at Stanford University in California, where I worked in Terry Winograd's *System Development Language Group* at CSLI. At the outset, the plan was for me to take part in the development of an environment, including a structure-oriented editor, for Winograd's specification language called Aleph [99]. Aleph was at that time, and is still (as of January 1987), in active development. As time went by, the goal became to construct a more general language development environment, which we call Muir in honor of John Muir, the great Californian environmentalist and explorer of the Sierra Nevada. This thesis documents my activities in the Muir project.

1.2 Methodological Remarks

If a single word should be used to characterize the method used in this thesis work it must be the word *experimental*. In the area of computer systems and "environments", the ultimate test of an idea is achieved through an implementation of the idea on a suitable computer. An implementation can be used to judge if the idea is feasible, and equally important, users can through the implementation try the idea out in

practice, and user-reactions can be collected.

At the beginning of the study, a collection of problems with syntax-directed editors was apparent. Due to our background in the area (see section 1.1), we also had some weak ideas of how to solve these problems. During the course of the thesis work, the problems were clarified through an interaction between analysis and experiments. The analysis identified areas, in which practical work was necessary and realistic in the scope of the already existing system. The experiments quite often provoked re-thinking of the topics, or they identified the need for work in other areas.

The analysis resulted in *working papers*, most of which are edited into this thesis. The result of the experiments is the system—or more correctly—the *prototype system*, which we call Muir. The prototype has been used to convince ourselves and others that our approach is feasible. We have not used the system to gather empirical results outside the research group. The purpose of Muir is to support a language development process. That is a new and an interesting topic in itself. However, from the point of view of this thesis, the prime importance of Muir is its role as a system, in which certain ideas have been tried out. The ideas are reflected by the main chapters of this thesis.

1.3 Outline of this Thesis

The thesis is organized in eight chapters. Following the introduction in this chapter we give an overview of the area, and we describe the grammar definition formalism that has been developed for Muir. Chapter four through seven are the main chapters of the thesis. They are on applications of pattern-based transformations and on abstract screen presentations. In chapter eight we summarize the conclusions. In the rest of this section we describe the contents in more details.

Chapter two gives an overview of structure-oriented editing. We describe some of the main developments in the areas of structure-oriented text editors, Lisp editors, and syntax-directed editors.

In *chapter three* we describe the fundamental grammatical framework used in Muir. This chapter is the key to a deep understanding of the rest of the thesis. Muir is based on so-called hierarchical grammars, a formalism that has been developed in the Muir project. Hierarchi-

cal grammars originate from grammars based on operators and phyla. An operator describes the abstract syntax of a primitive construct in a language, for example that an assignment has two constituents, a left-hand side variable and a right-hand side expression. Phyla describe syntactic domains, possibly in terms of more primitive syntactic domains. *Variable* and *expression* are examples of phyla. The reason we call our grammar formalism hierarchical is that the phyla are organized in a generalization/specialization hierarchy, more specifically in a directed acyclic graph. Typically, however, the hierarchy happens to be a tree. Interior nodes in the phylum graph are called categorical phyla, and leaf nodes are called terminal phyla. There is a one-to-one correspondence between the terminal phyla and the operators. A hierarchical grammar can cross language boundaries by referring to phyla in other grammars. This facilitates creation of multi-formalism documents. The phylum hierarchies of the different languages supported by Muir are connected into a single phylum hierarchy, whose root is the most general phylum in the system. In chapter three we also compare hierarchical grammars with BNF grammars and with grammars based on operators and phyla. Finally, we describe the most important principles and mechanisms in the Muir environment.

The following three chapters describe various applications of a pattern-replacement based transformation framework. In *chapter four*, we first introduce the transformation framework. A general transformation-step applies a pattern on an abstract syntax tree, and the resulting matches are substituted by replacements, in which substructures of the matches may occur. A pattern is a multi-formalism abstract syntax tree in the transformation language and some object language. Next in chapter four, we demonstrate how structure-oriented edit operations can be understood and/or implemented as transformations. Finally, we describe how transformations can be used to assist in carrying out major systematic modifications to a program.

Chapter five describes another application of the transformation framework. The problem we deal with is how to keep a set of documents updated with respect to a grammar that is under development. This has proven to be a problem in most environments where documents are represented and stored as abstract syntax trees. In particular, it is a problem that needs attention in a language development environment. Our solution is based on a notion of operator-versions. The basic idea is

to create a transformation template for each new version of an operator. In the most simple cases the transformation template is complete, but typically the user must refine the transformation manually. The environment supports identification of outdated constructs. Application of the created transformations will bring the outdated constructs up-to-date. We also discuss how modification of the phylum hierarchy can affect dependent documents, and how the system can support the updating of the documents after such grammar modifications.

Chapter six demonstrates how multi-formalism transformations can be incorporated into an editing environment. If the environment supports two languages, such as two programming languages, we demonstrate how transformation capabilities between these two languages can be made part of the environment. A relation among phyla in the source and the target languages is the only extension to the hierarchical grammar framework. This relation, called the source-target relation, allows us to consider transformation from one language to another as multi-formalism structure editing. The most trivial (identity) transformations can be created automatically by the system, and templates for the rest can be supplied automatically too. All in all it becomes a reasonable task to create a relatively complete set of transformations between (not too distant) languages. Application of the set of transformations on a source document typically produces a mixed-formalism result. During normal structure editing, plus a few additional supporting mechanisms, the user must eliminate the remaining source constructs. We illustrate the approach with transformation from Modula-2 to Pascal and vice versa.

Whereas the previous chapters all have treated manipulation of the internal representation, *chapter seven* is about the projection of the internal representation into a window on the screen. Traditionally this process has been called unparsing or pretty printing; we find it more correct and neutral to call it a presentation process. This thesis will only treat what we call abstract presentations, i.e., presentation where certain objects and relationships are emphasized, and where the remaining are left out. We will furthermore apply graphical (non-textual) means in abstract presentations, partly to challenge the widespread understanding that documents such as programs *are* textual. (However, many abstract presentations can be made with textual means as well.) We develop an abstract presentation, by which the overall composition of a document can be presented as a tree. More generally, we also develop a graph pre-

sentation style by which more "crossing" relationships in a document can be illustrated. Both the tree presentations and the more general graph presentations are defined by presentation rules, which are associated with the operators in the grammar. As a topic that links this chapter together with the remaining part of the thesis, we discuss what constitute "natural" edit operations on tree and graph presentations. Finally we discuss so-called overlapping presentations, i.e., two or more presentations that illustrate aspects of the same abstract syntax tree. It turns out that overlapping presentations are likely to occur in a system that applies abstract presentations together with more "concrete" presentations.

1.4 Summary of Results

In this section we will briefly point out what we find are the most interesting achievements of this thesis work.

In many application areas, the so-called generalization/specialization hierarchies have been used successfully to model a part of "the real world." We apply generalization/specialization hierarchies, called phylum hierarchies, to model the syntactic domains in a grammar. Very general as well as very specific syntactic domains are made explicit in this framework. Various qualities can in a natural way be made subject to inheritance in the phylum hierarchy. We find the proposed grammar model particularly interesting and useful in a language development environment.

One of the most important results is probably our proposal for keeping documents updated when the underlying grammar is modified. The power of the proposal is not based on advanced and automatic facilities. Rather, the novelty of the approach lies in the interaction between the user and the system. In this interaction the system creates a template of a transformation that is intended to update the documents, and the user refines this transformation based on his or her knowledge about the modification of the grammar.

Also with regard to translation between similar languages, the use of semi-automatic facilities has been successful. Most interesting, perhaps, is the system's capability to create a set of transformation templates from a relation among the syntactic domains in the grammars. Theoretically, it is of interest how the relation among the syntactic domains is used to

define a multi-formalism relation among the syntactic domains in both the source grammar and the target grammar. The semi-automatic nature of the system is repeated in the actual translation process. The application of the transformations is expected to create a mixed-formalism document, in which both source constructs and target constructs are represented. It is demonstrated how, through syntax-directed editing, this mixed-formalism document can be converted to a "pure document" in the target formalism.

Our work in the screen presentation area is a contribution to making better, overall presentations of big documents that are represented as abstract syntax trees. We find it important that abstract and non-textual presentations can be defined through grammar-related presentation rules. Our work can be seen as a generalization of the well-known techniques for generation of detailed, textual presentations of abstract syntax trees. As an interesting application of the general presentation framework, it is demonstrated how it can be used to present and manipulate our grammar definition formalism as a generalization/specialization hierarchy.

We also feel that we have been successful in comprehending and realizing a wide variety of edit operations as pattern-based transformations. Inheritance in the phylum hierarchy is used as a means for defining the constructs on which an edit operation can be applied. In a realistic syntax-directed editor, the necessity of a flexible repertoire of editing operations cannot be satisfied by grammar-induced edit operations alone. A general, pattern-based transformation framework seems to be a good vehicle for extension of the set of edit operations.

Chapter 2

Overview of Structure-Oriented Editing

This chapter is meant to provide an overview of the area of structure-oriented editing. It is not our intention to give a comprehensive survey of the area.[1] The primary aim is here to trace the "roots" of the tools and the environments that have been influential on the work described in this thesis. Secondary, this chapter could be a reasonable starting point for readers with little or no background in the field. We cover the whole area of structure-oriented editors, from the general text-oriented structuring tools, via Lisp editors to syntax-directed editors. The main emphasis, though, will be directed towards editors for artificial languages.

2.1 Editors for Structuring Text

In most traditional text editors there are very few, if any mechanisms that support the overall and logical structuring of documents. Despite that nearly all documents are divided into sectional units, which again are divided into smaller units, it has not been common for text editors to support such hierarchical structures. As far as most text editors are concerned, all what there is to text editing is characters that form words, and words that can be composed into lines and sentences. It is up to the user to follow a discipline that makes it possible to manage the overall structuring problems in one way or another. In this section we will describe some existing systems that directly support a hierarchical document structuring.

[1] We are not aware of any specific survey of structure-oriented editors. However, a survey covering the whole area of interactive editing systems can be found in [63].

One of the first systems in the field was NLS[2], a system developed at SRI International in the mid-sixties [26]. NLS was envisioned as a system for augmenting the human intellect, and its software as well as its hardware concepts were pioneering in the field. (The pointing device now broadly known as "a mouse" was invented in the NLS project.) One of the key facilities in NLS is a writing tool based on an explicit hierarchical structure of the documents. A textual unit is in NLS called a *statement*, which may or may not contain sub-statements. This hierarchical structure makes it flexible and fast to navigate around in the documents, and it becomes possible to show an outline presentation of the documents on the screen. (The importance of this aspect of NLS is elaborated further in chapter 7.5 on page 162 of this thesis.) In addition to the hierarchical structure, NLS also provides for arbitrary cross linking between statements, both inside and outside the actual document. In that respect NLS is an early *hypertext* system [66].

In 1978 the company Tymshare took over the development of NLS, and they also renamed the system to *Augment*. Reference [26] by Engelbart and English contains a description of NLS, written in 1968. A decade later the *Seybold Report on Word Processing* devoted an entire issue to a description of Augment and its history [83]. NLS/Augment is also briefly described in the survey paper by Meyrowitz and van Dam [63].

The basic ideas of NLS have also been tried out in other systems. ED3 [88,89] is a structure-oriented text editor from Linköping, Sweden, which has been used extensively for several years. XS-1 from ETH in Zürich is another example of such a tool [12]. The structuring of documents in ED3 is accomplished by two kinds of nodes, text nodes and tree nodes. Text nodes contain a piece of text, and they terminate the tree. Tree nodes also contain a piece of text, and in addition they refer to one or more subnodes. Both ED3 and XS-1 are intended for more general applications than text editing. Information in a very broad sense (pictures, diagrams, etc.) is found to have a hierarchical structure, and editors like ED3 and XS-1 can be used for the overall structuring and organization of such information. The idea of structure-oriented editors for general data structures is also described by Fraser in [32,33]. In recent years some of the concepts originally proposed in NLS have found their way

[2]NLS is an acronym for "oNLine System."

to commercial products, such as Thinktank [41,95]. In the commercial area the editors and environments are known as, for example, "outlining tools" and "idea processors."

Artificial language documents, for example programs belonging to a programming language, constitute another example of information that has a hierarchical structure. Therefore editors like NLS, ED3, and XS-1 are well-suited for editing of such documents as well. In [88] it is for example described how ED3 has been used to impose a structure on large programs, and how this structure is encoded into the program text as comments. The regular composition of artificial language documents makes it possible to apply structure-oriented editing techniques uniformly, both at the overall level and at the detailed level. To some degree, a similar regularity exists at the detailed level in natural language documents, but for most people, this regularity is of secondary importance when natural language documents are worked out. For artificial language documents, on the other hand, the language user has to be very conscious about the structure of the detailed units.

In the following two sections we will look at a variety of systems whose primary focus it is to support structure-oriented editing at the detailed as well as at the overall level. As we shall see, there is no general agreement on whether structure-oriented editing at the relatively fine-grained level is good or bad. It has been argued that text-oriented techniques are equally well-suited, or perhaps even preferable to structure-oriented techniques. We are first going to discuss structure-oriented editors for Lisp, a programming language for which structure-oriented editors have been used for many years.

2.2 Lisp Editors

One might ask why the discussion of Lisp editors and editors for "more conventional languages" is separated. The fundamental reason is the basic differences between the syntax of Lisp and the syntax of languages like Fortran, Cobol, Algol, and their descendants. The basic data structure and program structure in Lisp is called S-expressions. For our purpose it can be thought of as nested parenthesized list structures. Lisp gains much of its power, simplicity, and elegance by *not* distinguishing between various kinds of syntactic categories of lists. Editors for Lisp should con-

sequently be well-suited to manipulate the *structure* of such lists. The exact *form* of the list and the *syntactic domain* to which the list belongs are of secondary importance.

One of the primary purposes of a Lisp editor is to encourage or enforce a discipline that makes the parentheses balanced at any time. Without any support of that it turns out to be syntactically very error-prone to program in Lisp. So Lisp programmers have, to a much higher degree than programmers in "more conventional languages", felt an urgent need to adapt the editors to the peculiarities of the programming language. Fortunately for them, it turned out to be a reasonable job to do so *within* Lisp, because Lisp programs in a natural way can handle other Lisp programs as data.

A Lisp environment can roughly be characterized as either a *residential environment* or a *source file environment* [82]. In a residential environment, one and only one copy of a Lisp program exists in the environment, and this structure is manipulated directly by the editor. (Outside the environment, however, Lisp expressions are represented as text on files.) In a source file environment, a copy of the program is edited by a text editor, typically more or less outside the Lisp environment, and it is re-loaded into the Lisp environment when it has been changed. Interlisp is the most dominant residential environment, whereas MacLisp systems are prototypical examples of source file environments. We will first discuss several generations of editors in the Interlisp environment, and thereafter we briefly touch on editing of Lisp programs in a source file environment.

2.2.1 Editors in Residential Environments

In principle, it is possible to use the basic Lisp functions (CONS, CAR, CDR, RPLACA, RPLACD, etc.) to edit Lisp structures. But in order to speed up the editing process, a special "edit mode", in short known as *Edit* in Interlisp, was introduced [48]. In this "mode" a variety of terse commands are provided to modify Lisp structures, and to navigate around in Lisp structures. These edit commands implicitly operate on an expression called the *current expression.* Moreover, pattern-based search and replace capabilities add a considerable amount of power to the editor. Because the editor facilities were designed to work on TTY terminals, an explicit "pretty-print" command is necessary to present the resulting Lisp

expression on the output medium. Via the pretty-print command, the effect of the changes is typically presented every time a few modifications have been made to the current expression.

With the emergence of CRT screens it became possible to improve the TTY-based editor. Instead of asking for explicit pretty printing of the Lisp expression after a number of modifications, it is possible for the environment "continuously" to show an up-to-date picture of the Lisp expression. DED is an example of such an editor [8]. DED divides the screen into three regions: A prettyprint region in which the Lisp expression is presented, an interaction region in which the same edit commands as mentioned above can be issued, and a menu region, via which some frequently used commands can be initiated (by typing a number on the keyboard.) An automatic zoom facility is central to DED. If a Lisp list is too long or too deep, some details are left out, and only the expressions around the current focus are fully presented. The zoom facility can be seen as a compensation for the lack of screen space on a character and line-oriented terminal.

Following the next major improvement of output and input devices, namely with the emergence of the graphical workstation [94], it became possible again to improve the Interlisp editor. The display-oriented Lisp editor on the Interlisp-D workstations from Xerox is known as *Dedit* [48]. Compared with DED, the major advantage of Dedit is that it no longer is necessary to master the "complicated" TTY-oriented editing commands. The navigation in Dedit is done by selecting structures via the mouse, and the modification of the Lisp expression is done via a few (typically less than 10) different and intuitively simple commands. The commands are initiated from a menu, again via the mouse. The structure of the Lisp expression being edited can be modified via the main editing window. The contents of existing expressions, and entirely new expressions, must be entered in a buffer window as text. Only when the parentheses have been balanced, it is possible to exit the edit buffer. The new expression can then be inserted in various ways in the main editing window. Zooming is not used in Dedit. The Lisp expression is shown in full detail, and if it is too large to fit into the Dedit window, the window can be scrolled. Most people feel that Dedit is much easier to use than the TTY-oriented Edit. However, it is also the case that Dedit, as explained above, is less powerful than Edit. Therefore the whole functionality of Edit (for example the pattern matching and replacement) is made available in

Dedit as well. This is possible because Dedit actually is implemented on top of the existing TTY-oriented editing tool.

At the time of this writing, Dedit is the editor in current use on the Interlisp machines from Xerox. However, a new Lisp structure editor called *Sedit* [22,101] is currently being developed at Xerox Parc, and it will probably be available in a subsequent release of the Interlisp-D system. It is attempted to make the user interface of Sedit resemble that of Tedit [46], the text editing and formatting tool in Interlisp-D. It is also a goal to facilitate editing of other languages by the new tool, for example Prolog and Loops. Compared with Dedit, the most notable difference is that Lisp expressions in Sedit can be modified in the main editing window. I.e., the edit buffer is not needed any more. One of the purposes of having the separate buffer window in Dedit is to enforce the Lisp expression to have balanced parentheses. (As already noticed, it is simply not possible to exit the Dedit buffer without satisfying this constraint.) One of the major concerns in Sedit is therefore to maintain the balanced parenthesis structure via other means—by always inserting and removing pairs of parentheses. In Dedit, it is quite annoying to be forced to correct details, such as the spelling of an atom, in the edit buffer. In Sedit, such details can be corrected "at location", and in a more natural way. Finally, meta-commands on the keyboard constitute the primary way to initiate editing commands in Sedit. However, it is possible to ask for a menu of editing commands as in Dedit.

2.2.2 Editors in Source File Environments

Although our topic is structure-oriented editors, i.e., editors whose primary internal representation reflects the compositional structure of the programs, it is also interesting and relevant to study how text-oriented editing techniques can be adapted to work well for Lisp. One of the most refined systems in that respect is Emacs [85,86] and its descendants.

Text-oriented editors for Lisp programs do usually not enforce balanced parentheses, but they typically provide mechanisms that help the user convince himself or herself that the parenthesis structure is correct. When typing a closing parenthesis in Emacs, the corresponding opening parenthesis is highlighted for a second, or so. This makes it quite easy during the editing to maintain the structural understanding of the Lisp expression. This can be extended to also include manipulation of struc-

tural units as opposed to textual units. Furthermore, some editors, for example *Zwei* of the Lisp Machine from MIT [37], provide for formatting ("pretty-printing") of the Lisp expression while it is entered, and some amount of cross referencing support.

If a source file system is combined with (automatic) incremental loading of changed functions (as it is the case on the MIT Lisp Machine) the real difference in functionality and flexibility between structure-oriented and text-oriented editors for Lisp is probably very minimal. It turns out that it is a matter of background, taste, and (one could be tempted to say) "religion", which kind of editor the individual programmer prefers. (For a discussion or pros and cons, see Sandewall's survey paper on Lisp [82], and the subsequent discussion with Stallman—the implementor of Emacs. Both Sandewall's paper and Stallman's comments are reprinted in [7].)

2.3 Syntax-Directed Editors

The basic idea in syntax-directed editing is to create and modify documents in terms of *syntactic templates* instead of single characters. The templates can be understood as building blocks or toy bricks, which only can be assembled in certain ways. I.e., each kind of brick only goes well together with certain kinds of other bricks. Typically, a syntax-directed editor presents the range of possible templates that can be inserted at a given location in the document, and the editing is carried out by repeated *selection* of such templates from a menu. The most basic kinds of templates together with their composition rules can be defined by a context free grammar for the language.

The following list summarizes what we find are the most important advantages of syntax-directed editors:

- Violations of the context free syntactical rules can be prevented.
- It is possible to create and modify a document without being familiar with the syntax of the language, to which the document belongs.
- Fewer key strokes (and other input actions) are needed to enter a document.
- Structural well-defined constructs as opposed to arbitrary text intervals are manipulated during the editing of the document.

- By representing a document in tree form at any time during its development there is a potential of doing more powerful operations on the documents than if the document is represented as text. Furthermore, no time consuming parsing is ever needed.

As with structure-oriented Lisp editors, there is no general agreement on what to prefer: A syntax-directed editor or a good text-oriented editor (perhaps augmented with special language-oriented features.) The following points of view reflect commonly heard arguments for text-oriented editing, or a hybrid approach:

- Some modifications can be done very easily via text-oriented editing, whereas the corresponding structure-oriented modifications are complicated and tedious to carry out. This is especially the case for expressions and similarly detailed structures, but there are also some higher level constructs, for which this is true.

- In general, most people in the field feel comfortable with text editors, but some feel "claustrophobia" in a syntax-directed editor, because the "freedom" can be, and typically is, rather limited in such an editor.

- Unless a parser exists that can produce the internal representation of the syntax-directed editor, the new tool is incompatible with existing tools.

In [96] it has been argued that a hybrid approach, i.e., editors that support both text editing and syntax-directed editing should be preferred.

In the rest of this section we will first describe some of the major, and well-documented contributions to the field: They are Emily, Mentor, Gandalf, and the Cornell Program Synthesizer. The ordering of the description of these systems reflects the chronological ordering of the start of the respective research projects. In the last section, we take a broader look at the most interesting qualities of other syntax-directed editors that we are aware of.

2.3.1 Emily

Emily is the first syntax-directed editor to be described in the literature [39,40], and it is dated back to 1971. It was primarily designed to support creation of PL/I programs. Emily is for syntax-directed editing what NLS

is for structure-oriented text editing (see section 2.1.) Many of the ideas and the principles that have been put into syntax-directed editors in the seventies and the eighties can somehow be traced back to Emily. There are several "cornerstones" in the Emily efforts:

- *The basic idea*: Most important is the basic idea that a context free BNF grammar can be used constructively in an editor.
- *The hierarchical hypothesis*: This hypothesis says that "people think in terms of hierarchies and systems that manipulate hierarchies are better suited to creative work than systems that treat information as unstructured text."
- *The hardware basis*: The hardware, for which Emily was designed, is a vector-oriented graphic display (IBM 2250), a light pen, a program function keyboard, and an ordinary keyboard. The IBM 2250 was attached to a time-shared IBM 360 computer model 75.
- *The user engineering principles*: The design of the Emily editor was driven by four basic user engineering principles: (1) Know the user, (2) minimize memorization, (3) optimize operations, and (4) engineer for errors.

The basic idea and the hierarchical hypothesis have not been changed significantly since they were formulated. The hardware basis for Emily and "modern" syntax-directed editors is different. Vector technology screens with light pens have by and large been replaced with bit-mapped, raster-graphical screens and "mice." Some of the problems with Emily (such as the awkward interaction via the lightpen, and slow display of the legal set of productions in the menu) are clearly caused by qualitative or quantitative difficulties with the hardware. The Emily user engineering principles reflect "common sense", and they have not changed significantly either. However, since 1971 there has clearly been a movement towards some standard and proven interaction techniques (e.g., windows and popup menus), from which interactive tools like syntax-directed editors have been able to benefit.

Besides the four cornerstones that were mentioned above, a wide range of ideas were touched on in the Emily work. Let us briefly summarize what we find are the most important of them:

- Emily is not an editor for a single fixed language. Rather it is a *generator* that accepts a context free grammar in a special format. The abstract syntax and the concrete syntax are defined together in a BNF-like formalism, but it is concluded that a separation of the abstract syntax specification, the concrete syntax specification(s), and the syntax for the menu items is desirable. Emily supports its own grammar definition language.

- The notion of so-called *holophrasts* was introduced in Emily. A holophrast is a cover name of a substructure of a program which is presented instead of the structure itself on the screen. Emily, and *not* the user, generates the holophrast names, but the user is requested to select the structures to be elided. However, it is possible to change the *expansion depth*, which specifies the level in the tree where to "cut off and make holophrasts." In addition to the holophrast facility, Emily also supports a "safe and restore" facility for screen images. It makes it possible to switch between places in the document rather quickly.

- A facility that supports named fragments is supported by Emily. Fragments can be edited, and they can be inserted into other fragments. The "main text" that is edited in a session is just a fragment called *MAIN TEXT*.

- The idea of initiating a special routine when certain actions occur is also proposed. This has later been called *action routines* [59]. As an application, it is proposed during the creation of a procedure call to insert templates of the procedure arguments automatically, based on the declaration of the procedure.

- The observation that syntax-directed editors seem to be better suited for novices than for expert users has already been made during the use of Emily. The special problems with expressions is also noticed, and it is proposed to integrate a parser into the system. Of special interest for this thesis it is noticed that syntax-directed editors are well-suited for the language designer, because programs can be created immediately, even without knowledge about the syntactic details of the language.

- As a future direction it is observed that "it would be possible for an Emily-like system to be the primary interface between the user

and the system." This is, for example, a major point in Gandalf (see section 2.3.3.)

2.3.2 Mentor

The Mentor project was initiated at INRIA (France) in 1974 [23,24]. With Mentor, a new formalism for the definition of abstract syntax was introduced. Instead of nonterminals and productions the formalism is based on so-called *operators* and *sorts*. A sort is a name of a set of terms. A term is represented as an abstract syntax tree, but theoretically, a term is thought of as an element of a carrier of a sorted algebra. An operator is an n-ary function ($n \geq 0$) that maps n sorted terms into a new sorted term. Nullary operators represent constant terms. The set of operators with ranges of sort S characterize (and essentially define) the sort S. With regard to the grammar definition formalism, Gandalf, the Synthesizer Generator (both described below) as well as Muir (see chapter 3) are all inspired by Mentor.

One of the most interesting aspects of Mentor is its special-purpose tree manipulation language called *Mentol* [23]. Mentol serves as an interactive editing language as well as a programming language, which especially is well-suited for tree manipulation. To be well-suited as a command language for the editor, the language is terse and concise. One of the most important concepts in Mentol is the tree pattern matching and instantiation. Also at this point, Mentor has had a profound influence on the work described in this thesis. More recently, a meta language called *Metal* has been designed for Mentor (see [24].)

In some sense, editing in Mentor and editing in the Interlisp environment using the most primitive *Edit* tool (see 2.2.1) has many similarities: Both systems are based on special purpose command languages and powerful pattern matching facilities. Furthermore, none of the editors are screen oriented. It is up to the user to ask for a fresh, "pretty-printed" picture of the current editing focus. As argued in [23], this approach makes transportation to other systems much easier, and the consistency problems between the screen and the internal program representation vanish. However, the non-screen oriented approach has not been adapted by "younger systems." It seems as though the direct manipulation and the visuality is "a must" in more recent syntax-directed editors.

The so-called *gates* and *annotations* are also central to Mentor [24].

A gate is a mechanism that allows a document in one formalism to be a substructure of a document in another formalism. Files in a directory structure, and assembly language code in a high level program are given as examples of the use of gates. In many situations, the inner document may be considered as "atomic", when seen from the outer document. The gate mechanism is a static structuring mechanism. I.e., at grammar definition time, the possible places for gates must be determined. Annotations, on the other hand, are dynamic. Typical annotations are comments, larger pieces of documentations, and assertions about the document. In Mentor, an *annotation frame* has associated a formalism, to which the annotation must belong. So typically, an abstract syntax tree in one formalism can annotate an abstract syntax tree in another formalism. There are many similarities between annotations in Mentor and properties in Lisp [47].

A programming environment for Pascal has been developed in Mentor. I.e., the syntax of Pascal has been defined in the meta language, and a large amount of Mentol procedures have been made to facilitate creation, refinement, and transformation of Pascal programs. The development of Mentor itself has been done in this environment.

2.3.3 Gandalf

The Gandalf project [68] was initiated at Carnegie-Mellon University in 1976. The overall emphasis of the project is to construct and generate *software development environments*, i.e., environments that support the whole software development process. As examples of tools that have been created and integrated into a Gandalf prototype can be mentioned a project management tool (briefly described in [68]), a version control tool [51], and a tool for incremental programming [29]. A syntax-directed editor called *Aloe*[3] (A Language Oriented Editor) [25,59,60] is the primary interface to the tools in Gandalf software development environments. In this section it will be described what we find are the most interesting features of Aloe.

As Emily and Mentor, Aloe is not a fixed, syntax-directed editor, but an editor generator. As input it accepts a context free grammar

[3]A note on the chronology: The work on Aloe has been done in the beginning of the eighties, i.e., after and concurrent with the work on the Cornell Program Synthesizer, which is described in the following section.

in a particular formalism, and a set of so-called *action routines*, which are activated before and after certain events in the editor. There are some similarities between the grammar definition of Gandalf and that of Mentor (see 2.3.2.) In Gandalf a *class* represents a set of operators, and it roughly corresponds to a sort in Mentor. Gandalf also deploys the notion of operators, not as functions, but more as right-hand sides of productions. Two kinds of operators are distinguished: Terminal operators, which have no offspring, and non-terminal operators, which either have a fixed number of offspring, or a variable number of offspring (list of elements.) Action routines were originally formulated in an imperative programming language, namely C, but recently a special *action routine language* (ARL) has been created, together with a more systematic technique for activation of such routines [3].

One of the key capabilities of Aloe-based editors is the separation of abstract and concrete syntax, and the possibility to formulate several concrete syntaxes for a language, so-called *unparse schemes*. Certain kinds of constructs can furthermore be designated as *scenes*. When a scene construct is entered, a new window is opened, and the construct is presented in this window.

As opposed to, for example, the Cornell Program Synthesizer (see below), an Aloe editor instance is intended to be used to generate the entire program structure, including the expression and variable structures. As mentioned in section 2.3.1, the experience of many people seems to indicate that this is problematic. One of the problems is that expressions must be entered according to a prefix style (+ a b), whereas most people like to think of expressions in infix style (a + b). To enhance the structure-oriented editing of expressions and similar constructs, a technique has been developed that allows expressions to be entered in infix style. When the "+" is entered in the expression "a + b", the already existing operand "a" is nested into a template for the plus-operator. This technique is described in detail in [52].

In Medina-Mora's thesis [59] a tree partitioning mechanism called *file nodes* is proposed. The substructures of a node that is designated as a file node are not necessarily present in the memory of the computer. Rather, the "file node" refers to a file, on which its substructures are stored. There are some similarities between file nodes and the virtual memory mechanism based on paging, as described, for example, by Tanenbaum in [90]. Aloe is also one of the first editors to have *nest* and *transform*

edit operations, in addition to the more simple expansion and reduction operations.

Originally, the Gandalf editors and environments were designed for ordinary, line-oriented terminals. I.e., no advanced window and mouse support was provided. Recently, however, an Aloe-like editor called Gnome [15] (or MacGnome) has been implemented on the Apple MacIntosh.

Large volumes of literature are available on Gandalf. One of the best ways to approach the work is through the collection of Gandalf papers in *The Journal of Systems and Software*, vol. 5, number 2, May 1985. The last paper in this collection contains a bibliography of the Gandalf literature.

2.3.4 The Cornell Program Synthesizer

The Cornell Program Synthesizer [91,92] is a syntax-directed programming environment for a particular programming language, namely PL/CS (a "instructional dialect" of PL/I.) The work on the Synthesizer was initiated in 1978. Based on the experience with this work, an editor generator called the *Synthesizer Generator* [78,80] is being developed at Cornell University. As the name indicates, the Synthesizer Generator is not specific to any language. Rather, it accepts an attribute grammar in a particular formalism, and it generates an editor from this description. In this section we will first describe the most important lessons learned from the original synthesizer. Next, a description of the Synthesizer Generator is given, together with a brief survey of the work on attribute grammars.

In the Cornell Program Synthesizer, a new terminology was introduced: *templates, placeholders*, and *phrases*. Templates correspond to the right-hand sides of productions and placeholders correspond to nonterminals. Phrases are short textual units, which are modified via simple text editing instead of using template manipulation. Phrases are parsed after they have been created or modified, and in this way, errors in phrases are immediately revealed. The phrases reflect a choice of what to edit textually, and what to edit structurally. Expressions, assignments, and lists of names are examples of syntactic constructs that could be edited as phrases.

Comments in the Cornell Program Synthesizer are syntactic (and not lexical) units, and they are coupled to a facility for *syntactic elision* (which corresponds to holophrasting, see section 2.3.1.) A comment

may be associated with certain kinds of templates, and the user can switch between seeing the comment, and the template together with the comment.

The Cornell Program Synthesizer is designed for ordinary CRT terminals. As in the Mentor and the Aloe editors (see 2.3.2 and 2.3.3), the cursor must consequently be moved via special cursor control commands (typically bound to special keys on the keyboard.) A great variety of such cursor control commands are available in the Synthesizer. Some special cursor moving commands cause templates to be inserted in the middle of a list, and some make templates for optional constructs visible.

Not only the context free syntax of the programs is enforced. Also the more context sensitive rules of the language are checked, and whenever an error occurs, the faulty constructs are marked on the screen. The Cornell Program Synthesizer also supports execution and debugging of PL/CS programs. Selected variables can continuously be monitored, the point of control can be followed in the editing window, single stepping is possible, and a simulated reverse execution facility is also provided for.

As already described above, the *Synthesizer Generator* is meant to be a generalization of the Cornell Program Synthesizer. Editors to be generated by the Synthesizer Generator are specified in a language called SSL (the Synthesizer Specification Language.) Conceptually, the abstract grammar is defined in a similar way as in Mentor (see section 2.3.2.) A set of sorted terms is called a *phylum.* The phylum structure is flat, i.e., there is no notion of named phyla that contain named subphyla.

A *phylum declaration* states that a set of named operators—or more correctly, terms that are rooted by these operators—belong to the phylum. The operators themselves are defined as part of the phylum declaration. In essence, a phylum declaration does not differ very much from a set of productions in a context free grammar, all of which have identical left-hand side nonterminals. A variety of other declarations can be given. Most important, perhaps, there are declarations that define attributes of the phyla, and attribute equations of the operators. It is also possible to declare unparse rules, parse rules that define the concrete input syntax, and pattern-based transformations.

SSL does not enforce a particular strong ordering of the declarations. It implies that declarations in an SSL specification can be grouped together such that various aspects (for example abstract syntax, unparse rules, and attributes) are declared together, as opposed to being mixed

with the definition of other aspects.

SSL owns much of the expressiveness of a "general purpose programming language." The phyla play the role of data types in SSL, and the primitive data types, such as boolean and integer, are considered as predefined phyla. It is possible to define term-valued functions in SSL, there are variables, and there is a specialized conditional expression construct. All in all, an SSL editor specification can be quite complicated.

As indicated above, the use of attribute grammars plays a key role in the Synthesizer Generator. Attribute grammars can be used to check that the static semantic rules are fulfilled (e.g., that a variable is declared in the surrounding of its use) and to detect certain anomalies in programs (e.g., that a variable has to be initialized before it can be used). To allow incremental updating of attributes during interactive editing, considerable efforts have been put into the development of an incremental attribute updating algorithm [77,79]. The overall goal is to minimize the time needed to establish a fully attributed tree following one or more changes to the abstract syntax tree. There is still active research going on in this area [81,44] that elaborates Reps' results from [79].

2.3.5 Other Editors

The editors mentioned in the previous sections probably represent some of the more well-known contributions to the field. However, there exists many other structure-oriented editors for artificial (programming) languages: SED [2], PSG [4], EKKO [10], Magpie [20,21], Poe [31], R^n [42], Syned[43], Eliot [50], GLSE [55], MUPE-2 [57], Cépage [61], PECAN [75,76], and SUPPORT [103]. Of these editors and environments, only few are limited to the pure syntax-directed editing technique, in which every construct is inserted via selection of templates in a menu. In this section we will discuss the diversity of textual, structure-oriented editing techniques found in various systems.

Several editors adhere to a hybrid approach. I.e., they support syntax-directed as well as textual editing. PSG [4], Syned [43], PECAN [75,76], and SUPPORT [103], are examples of hybrid editors or hybrid editor generator systems. Also the Synthesizer Generator (see 2.3.4) is able to generate hybrid editors. Typically, a hybrid editor allows text entered via the keyboard to replace the currently selected template (placeholder), of which the syntactic category is known. When the entering of text is com-

pleted, the text is parsed using the already known syntactic category as a goal symbol, and the resulting tree substitutes the original template. If the entered text cannot be parsed, the user is requested to correct the input, or some kind of automatic error correction can be attempted (as in Poe [31].) Following the insertion of the syntax tree resulting from the parse process, portions of the screen image is usually regenerated from the syntax tree. This makes formatting and fonting correct and homogeneous, and it also makes it easier to deal with the mapping between the screen and the internal tree structure. If an already existing construct is edited as text, it is typically too time consuming to reparse the whole construct. If one or more of its subconstructs are unaffected by the modification, a *partial* re-parsing can be done, and the unaffected subtrees can be grafted into the resulting tree directly. A more limited, and somewhat strange hybrid approach is found in SED [2], where programs are created as text, but modified in terms of their structure.

After having discussed the hybrid approach, we will now look at some systems, for which textual editing is the only means. In Poe [31], a construct is inserted by typing one or more tokens that characterize the construct. It is not always necessary to enter the full textual representation of the construct, because automatic completion is provided. Moreover, automatic error-repair is attempted if the user-specified tokens do not immediately make sense. This framework can be considered as advanced parsing. It can also be considered as an alternative way to select the desired templates in the purely syntax-directed approach. Anyway, the net effect is that context free errors are prevented.

In Magpie [20,21], which is a Pascal programming environment running on an experimental Tektronix workstation, incremental parsing is used to construct the syntax tree while the program text is entered via the keyboard. A dedicated processor in the workstation continuously performs syntax analysis and compiling while another processor supports activities closer to the user interface of the system. The basic structure of a Pascal program is reflected by the *code browsers* in Magpie. Code browsers are alternatives to templates with placeholders. A code browser enumerates the basic constituents in one pane of an editing window, and in another pane the selected constituent (called a *fragment*) can be edited. The browsing technique is inspired by the Smalltalk-80 environment [35]. Incremental and immediate parsing becomes more manageable when it is known that only well-defined and minor fragments are affected, as op-

posed to large portions of a program.

Finally, Eliot [50] is a text editor for Pascal, in which the overall structuring of the program (procedures, functions, and statements) is represented structurally, whereas other program structures are represented as text. Although the context free syntax of programs is checked, the more detailed program constructs are never converted to a syntax-tree representation. In addition to the textual editing commands it is also possible to create the overall program structure in a syntax-directed style, i.e., via pre-defined templates.

Chapter 3

Muir Background

In this chapter we will describe the grammar model that we have developed in the Muir project, and after that, principles and mechanisms in the Muir environment will be highlighted. The work described in this chapter is joint work, to which contributions have been made by several people in the SDLG group. However, the elaboration and the formalization of the grammar model, as reflected in this chapter, is entirely the responsibility of the author of this thesis.

3.1 Hierarchical Grammars

The abstract grammar model and the document representation that we are about to describe in this section are results of a "refinement" of the grammar framework based on so-called operators and phyla. The operator phylum model was introduced by the Mentor group [23], and we briefly introduced the formalism in section 2.3.2 where the Mentor system was described.

The main emphasis in the hierarchical grammar model has been directed towards a description of general syntactic domains in terms of more specific syntactic domains. In, for example, a programming language the syntactic domain *statement* can be described as being either a *simple statement* or a *structured statement*, and each of these can be described as more specialized syntactic domains, such as *assignment statement* and *repetitive statement* respectively. We also use this simple principle to classify all constructs of a language into a single syntactic domain, and in turn, to classify all constructs of all supported formalisms into a universal "root domain" of the environment. The primitive constructs of the languages are associated with the most specialized syntactic domains in

the hierarchy.

The hierarchical organization of the syntactic domains is similar to the organization of Simula and Smalltalk classes in generalization/specialization hierarchies ([18] and [34] resp.) As in these languages, it is also tempting in the hierarchical grammar framework to make use of *inheritance* as a means to define the qualities of the elements in the hierarchy. In section 4.2.3 it is described how edit operations can be made subject to inheritance in the phylum hierarchy. In addition, one could imagine that other qualities, for example presentation rules and attribute equations, could be defined via inheritance.

We feel that the hierarchical grammar model reflects a natural and an intuitive simple way to think of the grammatical elements of a language. The model has a clear conceptual separation between structures that describe syntactic domains (or-structures) and structures that describe constructs of the language (and-structures.) This is not the case in a pure BNF grammar, in which the production concept is used for both purposes. As we will discuss in more detail in section 3.1.6, this weakness has been alleviated in some variants of BNF. A hierarchical grammar should be considered as an operator/phylum based grammar (as found, for example, in Mentor [23] and in the Synthesizer Generator [80]), in which the main emphasis is to model the hierarchy of syntactic domains in an explicit way. Every "classical" operator/phylum-based grammar can be formulated directly in the hierarchical grammar framework.

In section 3.1.1 we describe the most important concepts of the hierarchical grammar framework. In the following section, 3.1.2, this is made completely formal. Next, in section 3.1.3, some special phyla are introduced, among others the universal phyla, and in section 3.1.4 we define what we mean by multi-formalism grammars. Our notion of abstract syntax trees is defined in section 3.1.5. In particular it is emphasized how nodes in abstract syntax trees are associated with phyla in hierarchical grammars. Finally, in section 3.1.6, the hierarchical grammar model is compared with BNF and BNF-like formalisms, and with the "classical" operator phylum formalism.

3.1.1 Phyla and Phylum Hierarchy

According to the *The Shorter Oxford English Dictionary*, a phylum is a word from biology that means "a tribe or race of organisms, related by

descent from a common ancestral form". *Webster's New World Dictionary* states that a phylum in addition can signify a "language family". In computer science, the word "phylum" has been used to designate a set of terms [80], for example a set of constructs from an artificial language, and this is the meaning of the word that we adopt in this thesis. A phylum name is often called a sort, and one frequently talk about "sorted terms." When working with artificial languages, terms are typically represented as abstract syntax trees.

We organize the phyla in a hierarchy. This, we feel, is in the spirit of the biological meaning of the word "phylum." Concretely, a phylum hierarchy is a directed acyclic graph whose nodes represent phyla. We distinguish two kinds of phyla in a phylum hierarchy: Categorical phyla and terminal phyla. A categorical phylum corresponds to an interior node in the graph, and a terminal phylum corresponds to a leaf. The edges in the graph represent phylum/subphylum relations among the phyla. I.e., if P is a descendant of Q in the graph, the phylum represented by P is a subphylum of Q. We prefer the interpretation that a phylum hierarchy represents generalization/specialization relations among the various constructs in a language. The terminal phyla correspond to constructs, for which no specializations are described in the syntax of the language. The categorical phyla, i.e., the super phyla of the terminal phyla, correspond to various classifications of the constructs in a language.

Before we go further, let us look at the example in figure 3.1, which shows a phylum hierarchy for the statements in Pascal [49]. *For-down-to*, *for-to*, *repeat*, etc. represent terminal phyla, and *statement*, *labelled*, *unlabelled*, etc. represent categorical phyla. The hierarchy shown in figure 3.1 happens to be a tree. In general, however, there may exist nodes in the hierarchy that have more than one ancestor.

An operator is a function that maps n terms into a new term. For example, the *if-then-else* operator from Pascal is the function:

if-then-else: expression $\times$ statement $\times$ statement $\rightarrow$ statement.

Each of the operands of an operator belongs to a sorted phylum from the phylum hierarchy. In the hierarchical grammar framework there is a one-to-one correspondence between operators and terminal phyla in the phylum hierarchy. Thus, the *if-then-else* operator mentioned above is associated with the identically named phylum in figure 3.1.

In the framework described above there is an obvious inspiration from

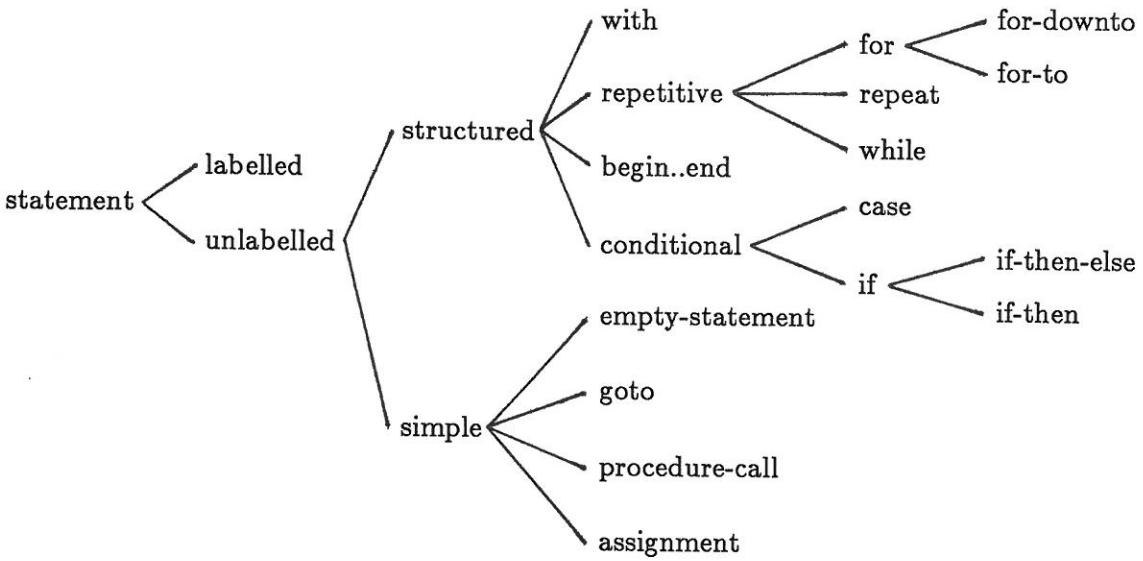


Figure 3.1: Phylum hierarchy for statements in Pascal.

many-sorted algebras (see, for example, [36].) A slightly different terminology has been adopted when talking about abstract grammars, but basically, we are working with algebras. The phylum hierarchy that we introduced above is a concrete manifestation of a partial ordering among the sorts of the algebra. In the following section the concepts that have been introduced in this section will be further formalized. We also introduce some convenient notation that will be used throughout the rest of the thesis.

3.1.2 Formal Hierarchical Grammars

In this section we give a formal account of hierarchical grammars. The meaning of "phyla" and "operators" from above will not be changed. However, we start by defining a formal structure with different, although related concepts, and it is then shown in a rigorous way how these concepts define the phyla and the operators. It should be noticed that the formalization described in this section is one out of many possible. We do not claim that it is "the best" or "the most natural" one. It has been chosen because it is quite similar to the well-known formal definition of context free grammars (see, for example, [1]), and because it reflects the actual objects in the implementation of the framework (to be described

in section 3.2.1.)

Definition. *Hierarchical grammar*

A *hierarchical grammar* is a tuple $G = (\mathcal{P},\mathcal{C},\mathcal{T},\mathcal{D})$ where

1. $\mathcal{P}$ is a finite set of *phylum symbols.*
2. $\mathcal{C}$ is a subset of $\mathcal{P} \times \mathcal{P}^+$. An element in $\mathcal{C}$ is called a *categorical phylum declaration.*
3. $\mathcal{T}$ is a subset of $\mathcal{P} \times \mathcal{P}^*$. An element in $\mathcal{T}$ is called a *terminal phylum declaration.*
4. $\mathcal{D}$ is a three-tuple of distinguished phylum symbols, each of which belongs to $\mathcal{P}$.

The distinguished phylum symbols will be discussed in section 3.1.3 and in section 3.1.5.

We require that the first component of a phylum declaration identifies it, and furthermore that the set of "terminal phylum symbols" and the set of "categorical phylum symbols" are disjoint. I.e.:

1. If both $(P,(P_1 \ldots P_n))$ and $(Q,(Q_1 \ldots Q_m))$ belong to $\mathcal{C} \cup \mathcal{T}$ and if $P = Q$ then $n = m$ and $P_i = Q_i$ for $i \in [1..n]$.
2. $\{P|\ (P,(...)) \in \mathcal{T}\}$ and $\{P|\ (P,(...)) \in \mathcal{C}\}$ are disjoint.

Because of these restrictions we can talk about *the* phylum declaration of a phylum symbol.

The phylum declarations in $\mathcal{C}$ are production-like structures that define the phylum hierarchy of a hierarchical grammar:

Definition. *Phylum hierarchy*

Let $G = (\mathcal{P},\mathcal{C},\mathcal{T},\mathcal{D})$ be a hierarchical grammar. The phylum declarations in $\mathcal{C}$ define a directed graph $\mathcal{G} = (\mathcal{P},E)$. If $(P,(P_1 \ldots P_n))$ is an element in $\mathcal{C}$, the edges (P,P_1), ..., (P,P_n) belong to E. Nothing else is in E. $\mathcal{G}$ is called the *phylum hierarchy of G.*

We see that a categorical phylum declaration defines a node together with its immediate descendants in the phylum hierarchy.

Having defined formally what a phylum hierarchy is, we are able to formulate two additional restrictions on hierarchical grammars. Besides

restriction 1 and 2 from above we require that

3. The phylum hierarchy of a hierarchical grammar must be acyclic.

4. If L is a leaf in the phylum hierarchy then there must exist exactly one phylum declaration (P,(P_1 ... P_n)) in $\mathcal{T}$ such that L = P.

Restriction 3 prohibits recursively defined phyla, and restriction 4 makes sure that for each leaf in the phylum hierarchy there exists one and only one terminal phylum declaration.

The categorical phylum declarations define a couple a very useful relations among the phyla in $\mathcal{P}$:

Definition. *Subphyla and superphyla relations*

Let (P,(P_1 ... P_n)) be a categorical phylum declaration in a hierarchical grammar G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$).

1. P_i is said to be an *immediate subphylum* of P for $i \in [1..n]$, and we use the notation $P_i \subset P$.

2. Symmetrically, P is said to be an *immediate superphylum* of P_i, and we write $P \supset P_i$.

The *subphylum* and *superphylum* relations are defined as the reflexive transitive closures of $\subset$ and $\supset$ respectively, and we write $P \subset^* Q$ if P is a subphylum of Q, and $P \supset^* Q$ if P is a superphylum of Q.

The subphylum relation is similar to the subset relation from set theory. The notation that we use for the subphylum and superphylum relations is therefore inspired by the corresponding notation from set theory.

It can be confusing to use the same notation for categorical phylum declarations and for terminal phylum declarations. We therefore introduce the following notation, which will be used in the rest of the thesis:

Notation. *Phylum declarations*

If (P,(P_1 ... P_n)) is a categorical phylum declaration and (Q,(Q_1 ... Q_n)) is a terminal phylum declaration in a hierarchical grammar then

1. (P,(P_1 ... P_n)) will be denoted P = {P_1 ... P_n}, and

2. (Q,(Q_1 ... Q_n)) will be denoted Q: Q_1 ... Q_n.

If $n = 0$ in a terminal phylum declaration, we call it a *nullary phylum declaration.* Notice that n cannot be 0 in a categorical phylum declaration. We now relate a hierarchical grammar to the definition of phyla and operators as given in section 3.1.1:

Observation

Let G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$) be a hierarchical grammar.

1. A *phylum* is a set of terms. The set of phylum symbols $\mathcal{P}$ is the sort set (phylum names) of the phyla. If a phylum symbol is a leaf in the phylum hierarchy of G, its corresponding phylum is called a *terminal phylum*, else it is called a *categorical phylum.*

2. An *operator* is a function that maps n sorted terms into a new sorted term, $n \geq 0$. If P: P_1 ... P_n is a terminal phylum declaration in $\mathcal{T}$ then there exists an operator that we here will call P′, and which is defined as P′: $P_1 \times ... \times P_n \rightarrow P$.

A given term t belongs to one and only one terminal phylum, say of sort T. Non-composite terms come from nullary operators, and they can be considered as constants. If P′: $P_1 \times ... \times P_n \rightarrow P$ is an operator derived from the terminal phylum declaration P: P_1 ... P_n then the i'th argument of P′ can be the term t if T is a subphylum of P_i.

Whenever we have a hierarchical grammar, we will talk about the phyla, the operators, and the phylum hierarchy of the grammar. In doing this we formally refer to the sets, functions, and graphs defined above.

3.1.3 Special Phyla

In this section we will describe two of the distinguished phylum symbols of a hierarchical grammar. If G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$) is a hierarchical grammar, we will in this section assume that

$$\mathcal{D} = (StartPhylum, Anything, Always).$$

Anything is the most general phylum in a grammar and *Always* defines general applicable constructs. The role of *StartPhylum* will be discussed in section 3.1.5.

A phylum hierarchy of a hierarchical grammar is seldom strongly connected. However, we now force the phylum hierarchy to be strongly con-

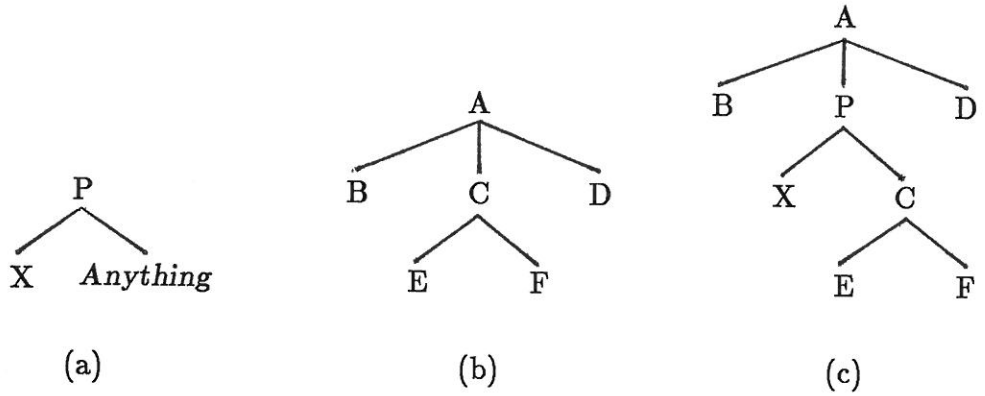


Figure 3.2: Syntax trees that illustrate the use of *Anything* and *Always*.

nected by defining a phylum of which any other phylum is a subphylum. We call this phylum *Anything* (or sometimes *G.Anything* if the grammar we are working with is G.) *Anything* can be used in operators to define places where every possible fragment of the document is legal. *Anything* is an implicitly defined phylum. I.e., the immediate subphyla of *Anything* need not to be defined explicitly.

The other special phylum that we define in this section is called *Always* (or *G.Always* where G again is the name of a hierarchical grammar.) The grammar definer should explicitly declare the subphyla of *Always*. *Always* is considered to be a subphylum of any other phylum in the grammar. Thus, the effect of defining P as a subphylum of *Always* is that P-constructs[1] can substitute any other construct defined by the grammar. Because *Always* is a special phylum, the terminal status of the terminal phyla will not be affected by the fact that *Always* is a subphylum of the terminal phyla. Symmetrically to the phylum *Anything*, the immediate superphyla of *Always* are implicitly defined.[2]

Let us look at an example where *Anything* and *Always* are used (see figure 3.2). In a given grammar we want to define a P-construct, in which it is possible to nest any construct into its second constituent.

[1] A "P-construct", where P is a phylum symbol, is a formalized notion. An if-then statement can be called an "if-then-construct" or a "statement-construct", because *if-then* is a subphylum of the phylum *statement*. See section 3.1.5 for the formal definition of a P-construct.

[2] If the subphyla of *Always* participate in the phylum hierarchy, cycles among the phylum symbols cannot be avoided. If, for example, P is a terminal phylum symbol for which P $\subset^*$ *Always*, the definition of *Always* implies that *Always* $\subset^*$ P. One could require that the subphyla of *Always* do not participate in the phylum graph. However, because *Always* (as well as *Anything*) are special phyla, the cycles in the phylum "hierarchy" do not harm, and they will not cause difficulties. We therefore accept cycles caused by *Always*.

Such a P-construct is shown in figure 3.2(a). If we nest the C-construct in figure 3.2(b) into the second constituent of the P-construct, we get the tree shown in figure 3.2(c). Independent of the explicitly defined phylum-subphylum relations, this operation is guaranteed to be syntactically legal if P is a subphylum of *Always*, and if "the phylum of the second constituent" of the P-construct is *Anything*. A practical example that illustrates the same aspects will be encountered in section 4.1.2.

3.1.4 Multi-formalism Grammars

Using the definitions from the previous sections it is possible to define a set of independent hierarchical grammars. We will now introduce the notion of so-called multi-formalism grammars in which certain interdependencies are allowed among grammars.

Definition. *Gate phyla and multiformalism grammars*
Let $G_1 = (\mathcal{P}_1,\mathcal{C}_1,\mathcal{T}_1,\mathcal{D}_1)$ and $G_2 = (\mathcal{P}_2,\mathcal{C}_2,\mathcal{T}_2,\mathcal{D}_2)$ be two hierarchical grammars, and let $P = \{P_1 \ldots P_n\}$ be a categorical phylum declaration from $\mathcal{C}_1$.

1. If $P_i \in \mathcal{P}_2$ (as well as $P_i \in \mathcal{P}_1$) then P_i is called a *gate phylum* from G_1 to G_2 in P ($i \in [1..n]$.)
2. A *multi-formalism grammar* is a grammar that contains one or more gate phyla.

Intuitively, a gate phylum in a categorical phylum declaration links the phylum hierarchies of the two hierarchical grammars together. Besides the gate phyla in categorical phylum declarations, it is also convenient to allow gate phyla in terminal phylum declarations. If P: P_1 ... P_n is a terminal phylum declaration from $\mathcal{T}_1$, and if P_i is a phylum that belongs to $\mathcal{P}_2$, then P_i is called a gate phylum from G_1 to G_2 in P. [3]

Multi-formalism grammars can be used to separate a grammar into several sub-grammars that are linked together with gate phyla. In other words, gate phyla can be used as a means for making a grammar more

[3]Gate phyla in terminal phylum declarations are not strictly necessary, because we can always declare a "categorical helping phylum" $H = \{P_i\}$ in $\mathcal{C}_1$, in which P_i is a gate phylum from G_1 to G_2. In that case, the terminal phylum declaration P: P_1 ...P_i... P_n should be replaced by P: P_1 ...H... P_n.

modular. When working on a big grammar, the advantages of splitting a grammar into several grammar modules can be compared with the well-known advantages of modularizing a program. In the Muir environment we have used this facility to factor separate issues of a grammar into sub-grammars. We have sub-grammars for an attribute grammar extension and for the presentation formalism.

Until now we have developed a grammatical framework where the syntax for each formalism is described by a single-rooted phylum hierarchy, and we allow the phylum hierarchies to be inter-connected via gate phyla. We finally enforce a single root of all phylum hierarchies. This absolute root of the phylum hierarchy we consider as a universal phylum in a special *system grammar*. Formally, if an environment supports the grammars G_1, G_2 through G_n, we declare *System.Anything* in the following way:

$$\text{System.Anything} = \{G_1.\text{Anything } G_2.\text{Anything } \ldots \ G_n.\text{Anything}\}.$$

Figure 3.3(a) sketches the top level of the phylum hierarchy for an environment that supports Pascal, Modula-2 and a grammar definition formalism. Dotted links in figure 3.3 are implicitly defined phylum-subphylum relations, i.e., they are not defined in any categorical phylum declaration.

In some situations it turns out to be convenient to attach qualities to the phylum *System.Anything* and have them inherited to any phylum in the system, independent of the language to which the phylum belongs. Also, it is sometimes necessary to use the phylum *System.Anything* as a gate phylum in an operator to describe a place where any possible construct in the system is allowed. We will see an example of that in section 4.1.2.

In section 3.1.1 we described that the phylum *Always* is considered to be a subphylum of any other phylum in a grammar. The system grammar also contains a phylum *System.Always* that is considered to be a subphylum of *G.Always* for any other hierarchical grammar G supported by the environment. It means that a P-construct, for which P is a subphylum of *System.Always*, can substitute any construct, independent of the language. Figure 3.3(b) illustrates the role of the *Always*-phyla in the phylum hierarchy of a grammar G.

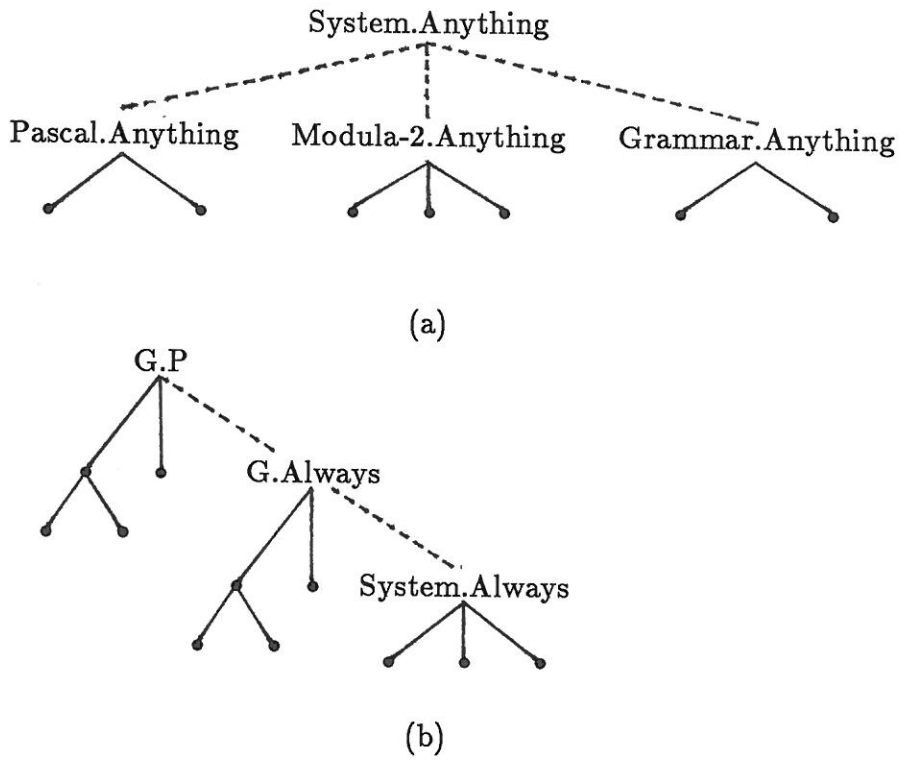


Figure 3.3: Top level of phylum hierarchy (a), and the role of *Always* (b).

3.1.5 Abstract Syntax Trees

In section 3.1.1 the notion of terms was introduced. Loosely, an abstract syntax tree is a term in which so-called unexpanded nodes are allowed. To be more precise, we give the following definition.

Definition. *Abstract syntax trees*
Given a hierarchical grammar G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$). An *abstract syntax tree* T is an ordered tree (see, for example, [1]) whose nodes are labelled with symbols in $\mathcal{P} \cup \{\text{NIL}\}$. In addition, the following conditions must be fulfilled:

1. If N is a leaf node in T, either
   (a) N is labelled NIL, and it is called an *unexpanded node*, or
   (b) N is labelled P where P is the phylum symbol of a nullary terminal phylum declaration in $\mathcal{T}$.
2. If N is an interior node, N is labelled P, where P: P_1 ... P_n is a terminal phylum declaration in $\mathcal{T}$, and $n > 0$. In this situation, the node N is required to have n sons.

In the following, "abstract syntax tree" will be abbreviated to "AST." We sometimes call an AST, all of whose nodes are phylum symbols in a grammar G, *an AST w.r.t. G*. When we have developed some more machinery, syntactically valid ASTs will be defined.

According to the definition given above an AST is an ordered tree. One could alternatively define an AST as an un-ordered tree, whose nodes are labelled (*tag*,P), where *tag* is an operator defined tag-name of the phylum symbol P. Figure 3.4 shows a fragment of a Pascal program together with its AST.

If B is a subtree (see [1]) of an AST A, B will be called a *constituent* of A. B is an *immediate constituent* of A if the root of B is a son of A. In figure 3.4 the if-then-else statement is an immediate constituent of the while statement, and the assignment statement is a constituent of the while statement. Notice that each constituent of an AST also is an AST. We introduce the following notation for ASTs:

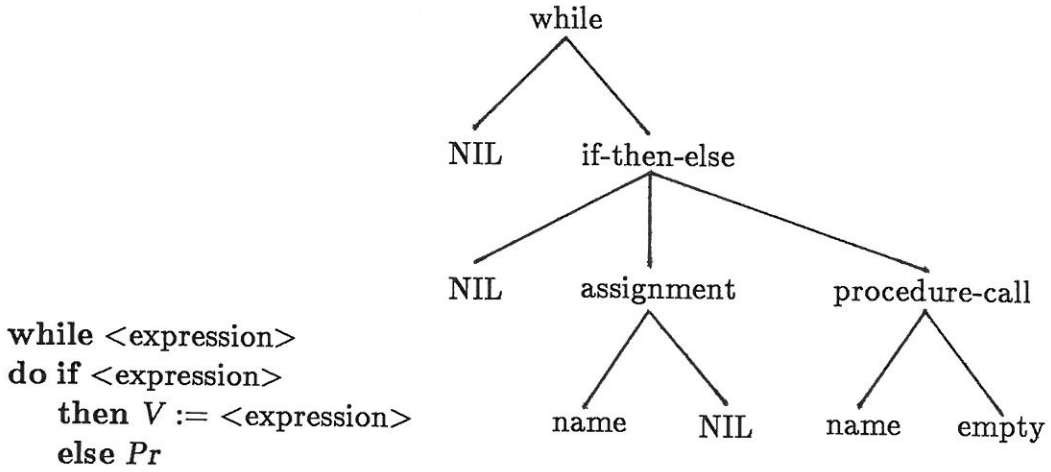


Figure 3.4: A Pascal statement (a) together with its AST (b).

Notation

If A is an AST whose root is labelled T, T $\neq$ NIL, and if C_1, ..., C_n are the immediate constituents of A (complete, from left to right), then we will use the notation T(C_1 ... C_n) for A.

We are now in a position where we can define two important phylum symbols that characterize an AST and its context.

Definition *Identification phylum*

Given an abstract syntax tree A w.r.t. a hierarchical grammar G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$). The *identification phylum* of A is defined in the following way:

1. If A = T(C_1 ... C_n), the identification phylum of A is the phylum symbol T.
2. If A is an unexpanded node, the identification phylum of A is undefined.

The identification phylum of A will be denoted φ(A).

Definition *Choice phylum*

Let A = T(C_1 ... C_n) be an abstract syntax tree w.r.t. the hierarchical grammar G. Let the terminal phylum declaration of the phylum symbol T be T: P_1 ... P_n.

1. For $i \in [1..n]$ the *choice phylum* of the abstract syntax tree C_i is the phylum symbol P_i.

2. The choice phylum of the root of an AST is undefined.

The choice phylum of an AST A will be denoted ω(A).

The identification phylum is the most specific phylum that identifies a construct, and consequently it is always a terminal phylum symbol. In figure 3.4(b) the identification phyla of the interior nodes are shown directly, and the identification phyla of the three unexpanded nodes are undefined. The choice phylum of a construct describes the set of possible alternative constructs that can fill it. In figure 3.4, the choice phyla of the if-then-else statement, the assignment, and the procedure-call are all *statement*. The choice phylum of the while statement is undefined, because it is out of context. In Muir, we do not represent the choice phyla explicitly in the ASTs. Rather, as the definition of the concept suggests, we look the choice phylum of a node N up in the terminal phylum declaration of the "supertree" of N. For example, the choice phylum of the assignment in figure 3.4 is the second phylum symbol in the declaration of the if-then-else terminal phylum.

Analogous to classes and instances of classes in object-oriented programming languages, an AST may be interpreted as nested instances of phyla from the phylum hierarchy. Using this analogy, an unexpanded node corresponds to an instance of a categorical phylum. An expanded AST-constituent corresponds to an instance of a terminal phylum. Such an instance is characterized by a reference to its context, by a given number of immediate constituents, and of course by the class of which it is an instance. Using this interpretation, the identification phylum of an AST-node corresponds to a class, of which the choice phylum of the node corresponds to a superclass, the qualification of the node.

Having the concepts of identification phyla and choice phyla, we can now easily define what we mean by a syntactically valid AST:

Definition. *Syntactically valid AST*

An AST A is *syntactically valid* if, for any constituent C of A, $\varphi(C) \subset^* \omega(C)$, whenever both $\varphi(C)$ and $\omega(C)$ are defined.

For notational convenience, we will in the rest of the thesis use the term "AST" to mean a syntactically valid AST.

Earlier in this chapter, we have several times in an informal way used the notation "a P-construct", where P is a phylum symbol. This is now made precise:

Definition. *P-construct*

A P-*construct* is an AST A for which $\varphi(A) \subset^* P$.

A context free grammar generates a language, which is a set of strings over some alphabet. The corresponding concept in a hierarchical grammar is that of a formalism:[4]

Definition. *Formalism*

Given a hierarchical grammar $G = (\mathcal{P},\mathcal{C},\mathcal{T},\mathcal{D})$, and let $\mathcal{D} = (S,Anything,Always)$. The *formalism* defined by G is the set of syntactically valid S-*constructs*.

Notice that this definition defines the role of the distinguished start phylum symbol. If every fragment is considered to belong to the formalism, S = *Anything* can be used. Also notice that a tree in a formalism is allowed to have unexpanded nodes as constituents.

In section 3.1.4 we defined multi-formalism hierarchical grammars and gate phyla. In this section we finally define what we understand by gates and multi-formalism ASTs.

Definition. *Gate*

Let $G_1 = (\mathcal{P}_1,\mathcal{C}_1,\mathcal{T}_1,\mathcal{D}_1)$ and $G_2 = (\mathcal{P}_2,\mathcal{C}_2,\mathcal{T}_2,\mathcal{D}_2)$ be two hierarchical grammars, and let Q be a gate phylum from G_1 to G_2 in some phylum declaration from $\mathcal{C}_1$. If A is an AST and if $Q \subset^* \omega(A)$, then A is called a *gate* (from G_1 to G_2.)

Thus, a gate is (a constituent of) an AST, for example an unexpanded node, that can be, but not necessarily is refined to an AST in another

[4]Our formalism notion is inspired by a similar notion in Mentor, see [24].

formalism. A *multi-formalism AST* in an AST that actually contains nodes belonging to two or more different formalisms.

3.1.6 Comparison with Other Grammar Models

In this section we will compare the hierarchical grammar model that has been developed above with other formalisms for definition of abstract syntax. For this comparison we choose the following formalisms:

- The BNF formalism [65].
- GRAMPS [14], which is a BNF-like formalism in which the productions are classified as *construction rules*, *alternation rules*, *repetition rules*, and *lexical rules*.
- The grammar definition formalism based on operators and sorted phyla, as defined in Mentor (see section 2.3.2.)
- The hierarchical grammar definition formalism, as defined in this chapter.

We are only concerned with the definition of abstract syntax. The BNF and GRAMPS grammars that we talk about in this section should therefore only contain nonterminal symbols. In other words, these grammars are stripped for terminal symbols. All the four grammar definition formalisms are variants of context free grammars [1]. Given a grammar in one of the formalisms mentioned above, it is straightforward to construct a similar grammar in one of the other formalisms. "Similar" means here that the two grammars generates the same (abstract) language, but not necessarily the same formalism. This is the difference between *weak equivalence* and *strong equivalence*, as drawn, for example, by Winograd in [97]. We have already in section 3.1.2 described how a hierarchical grammar is related to a pure operator/phylum based grammar. Let us in a similar way relate our grammar definition formalism to BNF[5]:

[5]In the original formulation of the BNF formalism [65] a production was called a metalinguistic formulae. A nonterminal was called a metalinguistic variable.

Observation

Let G = ($\mathcal{P}$,$\mathcal{C}$,$\mathcal{T}$,$\mathcal{D}$) be a hierarchical grammar.

1. To each phylum symbol P corresponds a nonterminal symbol. Let us here call it P′.

2. If P: P_1 ... P_n is a terminal phylum declaration in $\mathcal{T}$ then P′ ::= P'_1 ... P'_n is a production in the BNF version of the grammar.

3. A categorical phylum declaration P = {P_1 ... P_n} in $\mathcal{C}$ gives n productions P′ ::= P'_1 through P′ ::= P'_n in the BNF grammar.

It should be noticed that both categorical phylum declarations and terminal phylum declarations are "converted" to productions in the BNF grammar. In a similar conversion to the GRAMPS formalism, a terminal phylum declaration would derive a construction rule, and a categorical phylum declaration would give an alternation rule.

The structure of the syntax trees is important for the comparison in this section, because it affects the syntax-directed editors, which are based on the grammar. Unnecessary and redundant levels should be removed from the document structure, as presented for the editor user. A natural way to do this is to make the syntax trees "as flat as possible." This is difficult to achieve in a pure BNF grammar, if also intermediate syntactic domains (as shown, for example, in figure 3.1 on page 31) have to be represented as ordinary nonterminals in the grammar and as nodes in the ASTs. Similarly, it is awkward to share a syntactic construct between two or more nonterminals without introducing intermediate nonterminals that make the syntax trees deeper. (The only way to accomplish the sharing without introducing intermediate nonterminals is to duplicate some productions, but this is clearly awkward with regard to grammar maintenance.)

What is needed to solve the problems mentioned above is a separation between alternation rules and construction rules, exactly as in GRAMPS. With this separation, only the construction rules should affect the syntax trees. The alternation rules define the syntactic domains of the languages. In the hierarchical grammar definition formalism we have the same separation, namely as categorical phyla and terminal phyla. The main difference between the GRAMPS formalism and the hierarchical grammar formalism is that GRAMPS follows the BNF traditions, whereas hierar-

chical grammars originate from formalisms based on operators and phyla. Moreover, the "main structure" of a hierarchical grammar is the phylum hierarchy, which captures the whole alternation structure of a GRAMPS grammar.

Compared with a pure operator phylum grammar, a hierarchical grammar emphasizes the classification of phyla in a generalization/specialization hierarchy. As far as it can be figured out from [23], this is also possible in the operator phylum formalism in Mentor (via ordered sorts), but it is by no means a central mechanism. In the Synthesizer generator and in Gandalf, the phylum hierarchies are "flat." In addition, the hierarchical grammar model introduces a phylum for each operator. In other words, each operator has its own "type." Hereby it becomes meaningful to consider the hierarchy of phyla, without even referring to the operators.

3.2 The Muir Environment

Muir is an environment for development of and experimentation with artificial languages such as programming languages, specification languages, grammar formalisms, and logical notations. The kernel of the system is a structure-oriented editing environment that supports uniform manipulation of hierarchical grammars and documents derived in these grammars. The central facility in Muir is its capability to support instances of a structure-oriented, syntax-directed editor, which we call *Sedit*[6] (for *S*tructure *edit*or.) As mentioned earlier, an editor is *structure-oriented* if the primary representation of the documents reflects their structure as opposed to their surface form. Typically, documents are represented as abstract syntax trees. If, in addition, the structural building blocks provided by the editor are defined by a grammar, we call the editor *syntax-directed.* Syntax-directed editing seems to be especially attractive in a language development environment. The reason is that in a language development situation, the documents we work with belong to formalisms that are unfamiliar to most people. I.e., the support and guidance offered by a syntax-directed editor can really make a difference, not least if it is compared with textual editing techniques.

[6]It should be pointed out that Sedit in Muir has nothing to do with the Lisp editor of the same name (see 2.2.1.) Our use of the name "Sedit" can be traced back to an early phase of the Muir project [58].

As mentioned in the introduction of the thesis, Muir is being developed in the SDLG group at CSLI on the Xerox 1100-series Interlisp machines. Earlier in the SDLG project, there has been developed more specialized environments for the specification languages Aleph [99] and Dao [102]. The description given in this section reflects version 1 of Muir. After the author left Stanford, work on version 2 of the environment was initiated. In this section we will review the fundamental principles and mechanisms in the environment. In appendix C a "guided tour in Muir Woods" is given. Many of the aspects that are discussed in this section are illustrated in the appendix.

3.2.1 Realization of Hierarchical Grammars

In section 3.1 we formally defined the concepts of hierarchical grammars, abstract syntax trees, and their multi-formalism variants. The aim of this section is to describe how these concepts actually have been realized in the Muir environment.

When making a hierarchical grammar in version 1 of Muir, the grammar definer creates objects that are closely related to the categorical and the terminal phylum declarations of a formal hierarchical grammar.[7] The terminal phylum declarations contain the presentation rules. These are the rules that determine how an AST is presented on the screen. The constituents of a terminal phylum declaration are named (using so-called *tag names*), and these names are used in the presentation rules to refer to the "abstract constituents."

In version 1 or Muir, a hierarchical grammar is roughly a long, linear sequence of phylum declarations. This is by no means ideal. Much of the research in the Muir project has therefore been aiming at a more natural interface to a hierarchical grammar. Our vision is that it should be possible to create and modify a hierarchical grammar via its phylum hierarchy. The phylum hierarchy imposes an overall structure on the grammar, and from this presentation of the grammar more detailed aspects should be accessed. However, in practice, this goal has not quite been reached. Let us briefly mention the way we intend to realize the vision. As we shall

[7] Actually, a terminal phylum declaration consists of two declarations, a "dummy terminal phylum declaration" and a so-called operator declaration. However, in most grammars that have been made, there is a one-to-one correspondence between the terminal phylum declarations and the operator-declarations. Therefore in the thesis we consider these as a single declaration.

see in the following section, a grammar is just an AST, which is defined and constrained by a so-called meta grammar. The user interface to the grammar strongly depends on the way the grammar-AST is presented on the screen. At present, the list of phylum declarations is presented in a rather straightforward manner. Our goal has been to provide presentation mechanisms that make it possible to present the grammar-AST as a graph similar to the phylum hierarchy. There is of course no reason why this presentation mechanism should be limited to grammar-ASTs. As we shall see in chapter 7 of this thesis, there are many other interesting applications of such a powerful graph presentation framework. Currently, we are able to generate a phylum hierarchy from a grammar-AST via general graph presentation rules, but we have not implemented natural graph edit operations. In section 7.3 a general solution to this problem is proposed.

Nodes in an AST can be decorated with properties, exactly as it as possible to associate properties with atoms in Lisp [47]. I.e., an arbitrary number of data-structures can be put onto an AST-node as named tree properties. The tree properties of ASTs have been extremely useful in the Muir environment. They have, for example, been used for attributes (in an attribute grammar extension), for comments, to condense the information in grammar trees such that they can be used to drive the syntax-directed editors (see chapter 5), and to store relations among pairs of grammars (see chapter 6.) Furthermore, literal constants such as names, numbers, and strings can also be considered as properties of certain leaf nodes in an AST. The leaf nodes that can associate these properties are determined by a special *leaf phylum*.[8]

The representation of linear list structures in ASTs is an important issue that we also will touch on in this section. The most natural way to represent linear list structures in operator/phylum based grammars is probably through homogeneous, variable arity operators. A variable arity

[8]We have not included the leaf phylum as a distinguished phylum symbol in the formal description of the grammar framework, because this aspect is not central to the issues that we discuss in this thesis. Let us, however, here describe the mechanism that we use. The leaf phylum *LeafPhylum* is a distinguished phylum symbol in the same way as, for example, the start phylum. Let us assume that a phylum symbol T is a subphylum of *LeafPhylum*, and that T represents a terminal phylum. If that is the case, it is assumed that the phylum declaration associated with T is a nullary phylum declaration, and furthermore that it is possible to associate leaf properties to AST-nodes, whose identification phylum is T. Muir recognizes four different phylum symbols *nameApl* (applied occurrences of names), *nameDcl* (defining occurrences of names), *string*, and *number*, each of which may be a subphylum of the leaf phylum of a grammar.

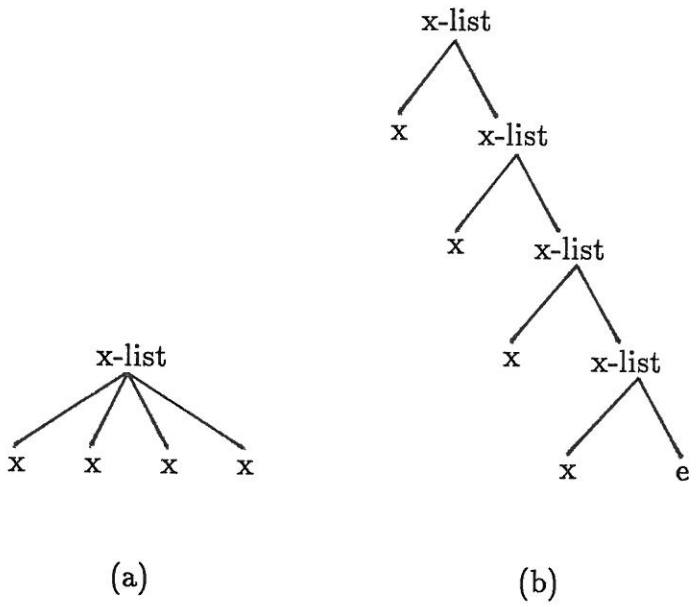


Figure 3.5: A flat and a binary list representation.

operator is an operator that can have an arbitrary number of immedidate constituents, and it is homogeneous if all constituents belong to the same phylum. Figure 3.5(a) shows such a list representation. In Muir, however, we have chosen to represent lists as binary nested trees of heads and tails, as shown in figure 3.5(b). It is the representation that can be derived by the following right-recursive production and its "completing production:"

x-list→x x-list
x-list→e

Using this representation, list structures do not require any special attention at the AST-level of the system, because they are aggregated via fixed arity operators, as are all other constructs. At the user interface level, however, it is not natural to manipulate lists directly in this representation. Therefore, we have defined special edit operations for creating lists, adding elements before and after existing elements, and deleting elements from a list. They are basically nest and unnest AST transformations, and they will be described in section 4.2.5.

The environment supports so-called list-operators, which describe the characteristics of a list (mainly presentation elements such as separators between elements, the prefix string in front of the first element, etc.) A list-operator is a description that allows the necessary operators and

phylum declarations to be generated by the system. In that way, the language definer does not need to do this manually. As an alternative to our approach, it seems to be more flexible to supply the information now contained in the special list-operators at the places in the operators where a list can be filled in.

Based on the experience with Muir, we are not convinced that the binary list representation should be preferred to the more flat representation. In some sense, the binary representation is not abstract enough. During the work on Muir we have experienced several peculiarities that can be blamed on the binary list representation. If, for example, we add an element in front of the first element in the list, we get a new root of the list. A tree property of the former root of the list is therefore not located at the root of the extended list. Selection and manipulation of sublists are not natural either. So, although flat lists require extra attention in several parts of the editor (for example in the presenter, in linear file representations of ASTs, etc.), the flat representation might in the long run be the best choice.

Finally, to deal with multi-formalism ASTs and versions of grammatical elements, an AST-node N contains the following three pieces of information:

(grammar-identification version terminal-phylum-name).

Grammar-identification is an abbreviated name of the grammar (such as PA for Pascal and MO for Modula.) *Terminal-phylum-name* is the identification phylum of the node, and it belongs to the grammar that is designated by *grammar-identification*. *Terminal-phylum-name* is NIL for unexpanded AST-nodes. *Version* refers to a specific version of a terminal phylum. We will defer the discussion of grammar versions, and hereby the *version* field, to chapter 5.

3.2.2 The Uniform Representation Approach

Every document that can be manipulated by Muir is represented as an abstract syntax tree, and it is constrained by a hierarchical grammar. The most important document in the system is the so-called *meta grammar*, i.e., the grammar for grammars. This grammar defines and constrains existing and future grammars supported by the environment, including itself. It also defines the basic functionality of the syntax-directed editor,

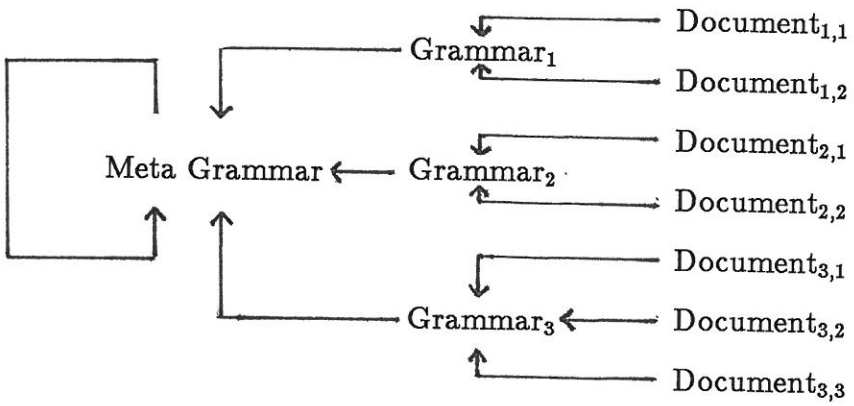


Figure 3.6: Schematic diagram of documents and their grammars.

by which it is possible to create new hierarchical grammars. The semantics of the meta grammar—which includes the account given in section 3.1—is deeply reflected in the implementation of the environment. For any other grammar supported by Muir, the environment only has syntactic knowledge. The first version of the meta grammar, or more precisely, the AST representation of the first version of the meta grammar, was partially hand crafted using Interlisp tools and the Bobs parser generator [27]. Succeeding versions of the meta grammar have been, and will be created via the current meta grammar in bootstrapping processes. In order to adapt the environment to a new meta grammar some changes in the Interlisp code, which realizes the system, must be anticipated. The meta grammar is therefore expected to be stable for a relatively long period of time.

The representation of grammars is conceptually identical to the representation of the meta grammar. As an example, we could create a hierarchical grammar for a programming language, say Pascal. Similarly, the hierarchical Pascal grammar in turn defines the basic functionality of a Pascal editor. The representation of Pascal programs, created via the Pascal editor, is conceptually identical the representation of the Pascal grammar. The three levels of grammars and documents are illustrated in figure 3.6. An arch from x to y means that y is a grammar, and that the document x is constrained by, and is dependent on the grammar y.

The uniform document representation is a cornerstone in the environment, because it allows any possible document to be stored, retrieved, edited, presented, etc. by the same means. As an example, it is possible

to apply transformations (see chapter 4) both on grammars and on the documents derived from the grammars. It is even possible to transform a transformation, because a transformation is represented in the same way as other documents (see section 4.1.2.)

The idea of having a meta language is not new. Already in Emily this idea was deployed [39]. In Muir, however, the only realistic way to create a new grammar, and hereby a supporting environment for a new formalism, is through the meta grammar. In other words, there exists no alternative (textual) input format, from which a functional editing environment can be generated. During the course of this thesis work we have created hierarchical grammars for the programming languages Pascal and Modula-2, together with a small set of sample programs in these languages. Besides these, considerable parts and subgrammars of the meta grammar have been created in the environment.

3.2.3 Separation Between Presentation and Representation

In the Muir environment a screen image of a document is called a *presentation* of the document. To a large extent, document representation and screen presentation of documents are separate issues. From the user's point of view, the document is identified with its presentation, and every action on the document must be understood through the presentation. It means that the user should not care about representational details. Similarly, when discussing the AST representation of documents, it should not be necessary to care about presentation issues.

A presentation is defined through a so-called presentation scheme. A presentation scheme belongs to a certain presentation style, and it consists of a set of presentation rules that are associated with the operators of the grammar. In version 1 of Muir we support text style presentations and graph style presentations. For a given grammar and a given presentation style, an arbitrary number of presentation schemes can be defined. Some of these can be used to produce traditional textual presentations of ASTs, whereas others can produce more detailed or more abstract presentations. As a key feature of the environment, the user can at any time select which presentation style and which presentation scheme to apply on a given document. The fundamental operation that makes this possible is called *Re-present*. The Muir presentation framework is described

in detail in [73].

There are two important motivations for providing the above mentioned presentation framework:

- It should be possible to emphasize certain objects and relationships among the objects in a presentation, and suppress the presentation of "irrelevant" details. Such presentations, which we call *abstract presentations*, are discussed thoroughly in chapter 7 of this thesis.
- We consider it as a goal at any time to be able to work with small, well-defined, and relatively self-contained presentations as opposed to "arbitrary slices" of large and detailed document images.

The two goals are not independent because many abstract presentations tend to be much smaller and clearer than their monolithic and textual counterparts. A typical way of working with Muir is initially to present a document at a relatively high abstraction level, and later when necessary, to open more detailed presentations of selected sub-documents in a browsing-like manner.

Several presentations of the same internal AST structure can exist on the screen. We say that two presentations are *overlapping* if some modification of the shared AST structure affects both presentations. Overlapping presentations are essential in the Muir environment because an overall, abstract presentation in general presents some of the same aspects that will be shown in more detailed presentations. Muir attempts to keep overlapping presentations consistent in the sense that modifications carried out through one of the presentations, say P, immediately will be reflected in the remaining presentations that overlap with P. More details about our approach to overlapping presentations will be given in section 7.4.

3.2.4 Co-existence with the Interlisp Environment

As already mentioned, the Muir environment is implemented in the Interlisp-D environment. Following the tradition in the Interlisp community, Muir is not a subsystem of the Lisp environment, but a co-existing component and an augmentation of the system. In this section, we will take a closer look at how Muir fits into the Interlisp-D environment.

A standard Interlisp-D environment contains two important editors: *Tedit* [46] and *Dedit* [48]. Tedit is a "What You See Is What You Get"

(WYSIWYG) text editing and formatting system. Dedit is non-syntax-directed structure-editor for Lisp expressions (see section 2.2.) Sedit, on the other hand, is a syntax-directed structure editor for formalisms, which have been defined by hierarchical grammars. While using Muir, it is possible to activate both Tedit instances and Dedit instances. As we shall see in the following chapter (section 4.2.6), the Muir environment has been tied together with its "host environment" in a simple way (it is possible to activate Dedit from Sedit.)

There is an even closer, although a more implementation-oriented connection between Sedit and Tedit. Text style Seditors are namely implemented entirely via the so-called *hooks* and the *functional interface* provided by Tedit [46]. This is based on the observation that a text style syntax-directed editor at the user interface level can be simulated by disciplined use of a text editor. This discipline is programmed into the system, and thereby enforced by Muir. The hooks in Tedit provide for execution of user supplied functions upon certain interactions. The functional interface makes it possible in a programmatic way to carry out any sequence of editing actions that can be done interactively. The functions that are supplied as hooks typically change the state of Tedit via calls to the functions that comprise the functional interface. Besides being an interesting implementation technique, at least for people with a static Pascal background, it implies that presentation-oriented editing (text editing) in Sedit would be relatively easy to integrate, and trivial to make consistent with text editing in Tedit. However, in version 1 of Muir, presentation-oriented editing is not available, because no parser has been properly integrated into the environment.

3.2.5 Initiation of Actions

Finally in this chapter, we will describe how various kinds of actions can be initiated in Sedit instances. Basically, two kinds of actions exists: Selection of constructs, and execution of edit operations. As it is nearly standard in so-called modeless systems, the command syntax is postfix. I.e., first an argument is selected, and next an operation is executed on the argument. We will now describe the various options for making sub-document selection, and for initiating edit operations.

The currently selected sub-document in an Sedit instance is called the *focus of the Seditor* (or the *focus of the presentation.*) The focus of the

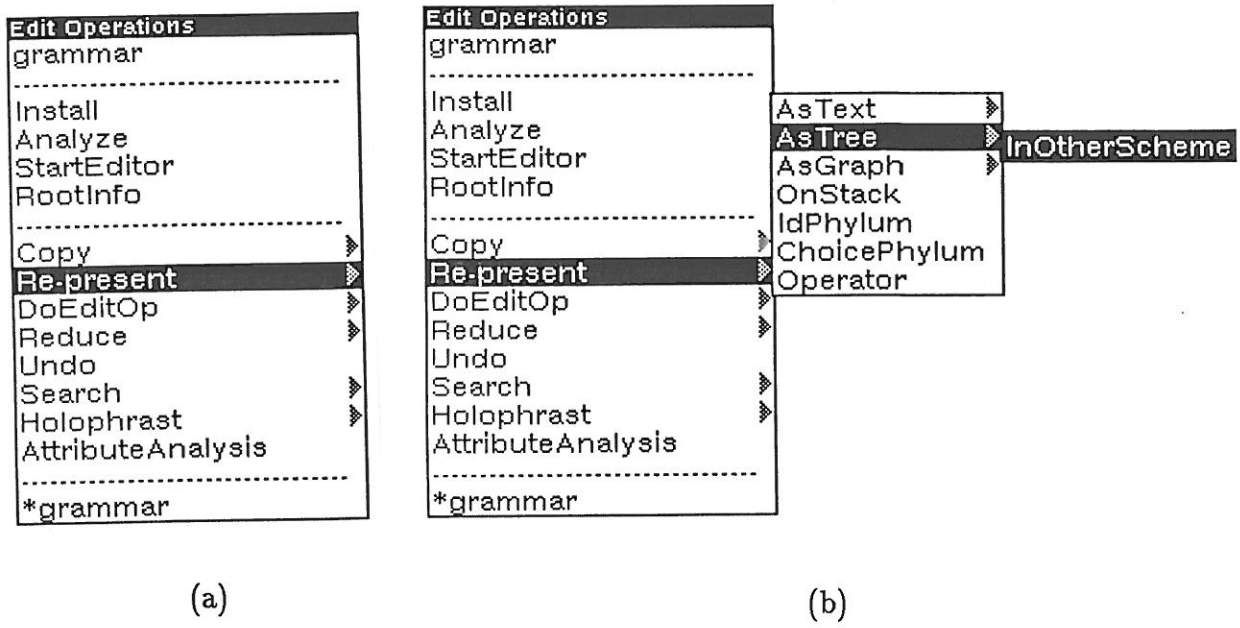


Figure 3.7: Examples of Muir popup menus.

current Seditor (i.e., the Seditor instance to which the latest interaction has been directed) is called the *current focus.* The primary way to change the current focus is via the mouse. The selection technique is basically independent of the already existing selection. However, if the same position is selected more than once, the current focus will be the minimal enlargement of the already existing focus. (In terms of the internal AST representation, the focus is moved one level up in the tree.) To a limited degree, it is also possible to move the current focus via the keyboard (using a so-called static command, see below.) In version 1, this is limited to moving the cursor "forward" to the next placeholder.

The primary way to initiate edit operations is through a popup menu, where the operations are grouped into a fixed number of (possibly empty) segments. Figure 3.7 shows examples of popup menus for the top-construct of a grammar. The upper section in figure 3.7(a) contains edit operations derived from the operators of the grammar, and they are also called primitive edit operations. (In figure 3.7 there is only one edit operation in this segment.) Section two contains transformational and programmed edit operations specific to the current focus. Section three contains edit operations that always make sense, independently of the current focus. (It is the edit operations that are associated with the

phylum *System.Anything*, see section 3.1.4 and section 4.2.3.) Section four makes available so-called composite templates, and also these depend on the current focus. Finally, a fifth section may contain operations on lists. In figure 3.7 the list-section is empty. All the different kinds of edit operations will be discussed in more detail in chapter 4.

The menu items that contain small arrows have sub-menu items associated with them, and these can be reached by "rolling" the menu out one or more levels. This is a standard facility in the Interlisp-D environment. Figure 3.7(b) shows the pop-up menu rolled two levels out. The operation that is just about to be initiated in figure 3.7(b) will be denoted *Re-present*▷*AsTree*▷*InOtherScheme*. We primarily use sub-menus for specialization of operations. *Re-present*▷*AsTree*▷*InOtherScheme*, for example, re-presents the current focus in a new Seditor window using a tree style presentation scheme chosen by the user.[9] Secondary, we use sub-menus as a grouping facility in order to improve the clarity and in order to minimize the size of the popup menus.

In addition to activation of edit operations via popup menus, Muir also supports what is called static and dynamic keyboard commands. A *static keyboard command* is activated via a single control character, and therefore only a few, but essential functions are made available in this way. For example, control-Z (↑Z) moves the current focus to the next placeholder, and control-E extends a list with an additional element. A *dynamic keyboard command* is interpreted w.r.t. the current focus, and it is activated as the static command control-X followed by an operation name. The operation name should be one of the names in the popup menu for the current focus, but only a prefix that determines the operation uniquely is necessary. If, for example, the current focus gives the popup menu in figure 3.7(a), the keystrokes '↑X c CR' will activate a copy operation. As a limitation in version 1 of the environment, only edit operations at the outer level of the popup menu can be activated via dynamic keyboard commands.

There is also an edit operation called *DoEditOp*, which allows the user to activate an edit operation that somehow is presented on the screen. Finally, it is of course possible to activate any edit operation via a Lisp

[9]It would have been more natural to let the third level determine the tree presentation scheme directly. However, to make the creation of popup menus more efficient, *Re-present*▷*AsTree*▷*InOtherScheme* prompts the user for the desired presentation scheme in yet another menu.

function call, but normally this possibility is only used programmatically, and not interactively from a Lisp listener window.

3.3 Summary

The hierarchical grammar framework is based on the operator/phylum model. Hierarchical grammars encourage the understanding of a grammar as a generalization/specialization hierarchy of syntactic domains, the phylum hierarchy. In the formal description of hierarchical grammars the so-called phylum declarations define the operators as well as the phylum hierarchy. We consider the phylum hierarchy as the primary interface to a hierarchical grammar. Multi-formalism grammars are introduced via the gate phylum concept, and gate phyla induce gates in the corresponding multi-formalism ASTs. It is straightforward to interpret a hierarchical grammar as an abstract BNF grammar where two kinds of productions exist: Construction rule and alternation rules.

A great deal of flexibility is gained from the use of the special phyla *Anything* and *Always* and their multi-formalism variants. Without the conventions built into *Anything* and *Always* it is difficult to create very general syntactic domains, and it is awkward to allow a specific construct to be used anywhere in a document.

The uniform representation of the meta grammar, other grammars, and their derived documents is of prime importance for the environment. The ability to have an arbitrary number of editor instances is another key characteristic of the environment. Each of the editor instances presents a document in a given style, and by using a given presentation scheme. Both the presentation style and the presentation scheme are selected by the editor user.

Proper environmental support is crucial for the success of the hierarchical grammar model. An environment that allows creation and modification of a hierarchical grammar by direct manipulation of its phylum hierarchy seems to provide a good basis for a language development environment.

Chapter 4

Transformations and Edit Operations

In this chapter we will first introduce a pattern-based transformation framework on abstract syntax trees. Following that it is demonstrated how this transformation framework can be used to implement a variety of structure-oriented edit operations in a syntax-directed editor. Association of edit operations with the constructs, on which they can be applied, is an important aspect of the implementation of edit operations. It is shown how inheritance in the phylum hierarchy can be used for this purpose. In chapter 5 and 6 we will study other applications of the transformation framework.

4.1 The Transformation Framework

The transformation framework in Muir is based on tree-pattern matching and replacement. Basically, a pattern can be applied on a list of ASTs to produce a list of matches, and all the matches will be replaced by a replacement-AST, in which (copies of) substructures of the matches may occur. Many other systems, for example, Mentor [23], PDS [16], and TAMPR [11], use similar transformation mechanisms.

4.1.1 How it Works

We will now describe the functionality of the Muir transformation framework. To avoid confusion, we will call the abstraction that we are about to describe a *transformation*. The process of applying a transformation on a document, which also could be called a transformation, will in this thesis be called a *transformation step*.

A transformation obeys the following syntax:

Transformation *name* (*parameter-list*)
Explanation *explanation*
Pre-condition *pre-condition*
Transformation-class *transformation-class*
Focus-modification *focus-modifier*
Pattern *pattern*
Replacement *replacement*

The first argument passed to a transformation must be a list of ASTs that determines on which parts of a document the transformation is applied. We call this argument the *transformation focus.* An arbitrary number of additional arguments may follow the transformation focus. The *explanation* is intended to contain a helping text that describes the effect of applying the transformation on a document. The environment can use the explanation field to guide the user in various ways. The *transformation class* allows us to vary the effect of the pattern matching and the replacement. This is explained below. The *focus modifier* makes it possible to redefine the transformation focus before the pattern/replacement mechanism is activated. Hereby a transformation can be carried out on an enclosing construct or on a sub-construct of the original transformation focus. This is sometimes useful when a transformation is applied as an edit operation (see section 4.2.5.)

A *pattern* is an AST that matches instances of that AST in the transformation focus. An unexpanded node in the pattern matches any construct C, for which the identification phylum of C is a subphylum of the phylum indicated in the unexpanded node. Textual patterns with wild cards can be associated with the leaves in the pattern-AST. In addition, special patterns that match any transformation focus (ANY), any P-construct (ANY of SyntaxCategory P), and any unexpanded construct (ANY Unexpanded) are supported. Most of the mechanisms can be combined such that quite versatile patterns result. Any constituent of a pattern may be named such that the corresponding constituents of the matches can be referred to in the replacement and in the pre-condition.[1] A given name should only be applied once in a pattern. Alternatively, we

[1] In the current version of the system, named constituents of matches cannot be referred to in the pre-condition.

could have allowed identically named pattern constituents and hereby require that these constituents should be identical (in some sense.) This is the interpretation that has been adopted in, for example, the Synthesizer Generator [80].

The *pre-condition* is an assertion, which may refer to the transformation focus and to the matches found by the pattern matcher. In the pre-condition, a match is referred through the name $Match. A pattern only matches a constituent of an AST if the pre-condition is fulfilled. The *replacement* is substituted for all the matches. As described above, named constituents of the pattern can be related to constituents of the replacement. If the pattern constituent C_P in this way is related to the replacement constituent C_R then the construct matched by C_P is copied and substituted for C_R. C_R is usually an unexpanded node in the replacement. A special replacement called "MATCH" indicates that the replacement is equal to the match; i.e., no "real" replacement takes place, and the transformation is therefore trivial. This feature makes it possible to implement structural search operations in terms of transformations (see section 4.2.4.) Finally, the list of replacements is returned as the result of the transformation. If the replacement is "MATCH", the list of matches is consequently returned.

There are two basic transformation classes: *General* and *primitive*. If the transformation is of class general, the transformation is recursively applied on the replacements. This is not the case for primitive transformations. These two transformation classes also have variants, *general-by-reference* and *primitive-by-reference* that do not copy the substructures of the matches when they are transferred to the replacement. Rather they are inserted directly[2]. Finally, the transformation class *primitive-expand* provides an efficient way to insert templates (represented in the replacement part of the transformation) into a document. A *primitive-expand* transformation can, however, be implemented as a less efficient *primitive* transformation as well.

When a transformation is applied interactively, as, for example, in an edit operation, the presentations on the screen must be updated following the completion of the transformation. Unless explicitly decoupled, the screen updating is handled automatically by Muir. When, during the realization of a transformation step, it is identified that a tree T is going

[2]This, of course, precludes use of overlapping sub-matches and duplication of sub-matches in the replacement.

to substitute a tree S, a request for screen updating is issued. In order to collect details about the position and the extent of the presentation of S, and hereby to facilitate incremental screen updating, it is important that the request is issued before T actually substitutes S. All the screen updating requests are queued, and after the transformation step is completed, they are realized. The realization naturally depends on the presentation style in the affected windows. This subject is discussed in more details in section 7.4.

4.1.2 Representation of Transformations

Transformations in the Muir environment are represented as multi-formalism abstract syntax trees (see section 3.1.5.) The outer level of a transformation, the name, the explanation, the pre-condition, the transformation class, and the focus modifier belong to the meta grammar of Muir. The pattern and the replacement constituents are gates that allow any possible construct from any supported formalism to be pattern and replacement respectively. I.e., the choice phyla of the pattern and the replacement are *System.Anything*.

In order to give a substructure S of the pattern (or the replacement) a name we nest it into a special naming construct that belongs to the meta grammar. As an example of that, let us assume that the pattern is a Pascal *if-then-else* statement, and that the then-statement and the else-statement must be named "first" and "second" respectively. The standard presentation of this pattern together with a tree presentation that reflects its AST representation are shown in figure 4.1. The boxed nodes in the AST representation belong to the meta formalism, the others are Pascal nodes.[3] In order to make the subPatName-constructs syntactically valid, the terminal phylum symbol *subPatName* is a subphylum of *System.Always* (see section 3.1.4.)

The representation of a transformation as a multi-formalism AST allows us to manipulate (edit, present, store, and even transform) a transformation like any other document supported by Muir.[4] It also saves us

[3]Both figure 4.1(a) and figure 4.1(b) can be generated as presentations of ASTs in Muir. The boxes in 4.1(b), however, are added by hand.

[4]A meta problem may occur if we want to transform a transformation. Let us assume that we, as part of a transformation, want to match all *subPatName*-constructs (see figure 4.1) in a collection of transformations. The system will consider the *subPatName*-construct in the pattern as a naming construct, and process it in a special way. It will not be considered as a construct

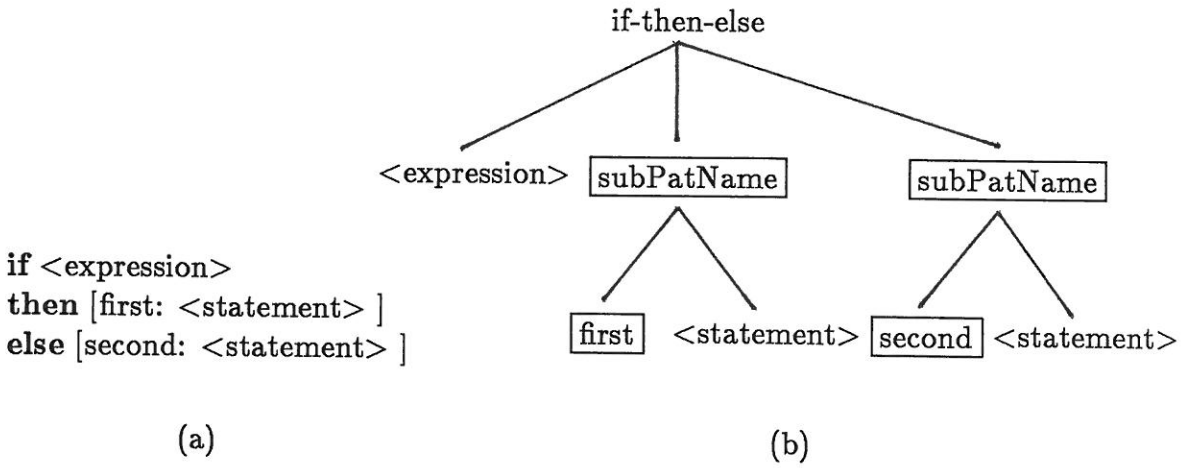


Figure 4.1: A textual and a tree presentation of a pattern.

the effort to come up with a special denotation for program fragments, such as "if-then-else(<expression>, *first*, *second*)" for a reference to an if-then-else statement where the first statement is named 'first' and the second statement is named 'second' (as used in the Synthesizer Generator [80].) Moreover, the distinction between abstract and concrete program fragments, as drawn in [54], becomes a question of presentation (and not *re*presentation.)

4.1.3 Limitation of Pattern-Based Transformations

The pattern matching capabilities of the transformation framework, as described above, clearly suffers from some limitations. It is, for example, not possible to define a pattern as a union of more primitive patterns, or as a negative pattern. Consequently, it is difficult to specify that any construct C that matches a pattern P_1 *or* a pattern P_2 should be replaced by a construct C', and that any construct C that does *not* match a pattern P should be replaced by a construct C'. It would be relatively straightforward to extend the pattern matching framework to include unions of patterns, negative patterns, and other options, but there remain translations that are hard and awkward to realize via pattern-based transformations. We will now take a closer look at that.

Basically, a pattern-based transformation causes one or more constituents of a document, which have been captured by the pattern and the

for which to search. To alleviate this problem, a construct like a *subPatName*-construct can be marked in a special way—using a tree property—in order to override its special meta meaning in the processing of the pattern and the replacement.

pre-condition, to be replaced by a given replacement. The constituents of the matches that explicitly have been enumerated through the pattern can be inserted into the replacement. This model for changing a document makes it difficult to move a construct from one place in a document to another. In order to do that in a single transformation, the "from-place" and the "to-place" must be captured by the same pattern. I.e., the pattern must match a structure of the document, of which both the "from-place" and the "to-place" are sub-structures. Furthermore, both of the places must be enumerated explicitly, and the gap in between these places must somehow be matched too. If the distance between the "from-place" and the "to-place" is great, typically if they belong to different structural levels, it may be difficult to formulate a sufficiently broad pattern.

Because it sometimes is difficult and awkward to move a construct from one place in a document to another, sorting and other kinds of re-organizations are also difficult to carry out via pattern-based transformations. As an example of a hard reorganization, we will later on in the thesis study the translation of a Modula-2 declaration part (which is a flat list of mixed declarations) to a Pascal declaration part (in which the declarations are grouped into constants, types, variables, and procedures.) (See section 6.4.4.)

It should be noticed that we are *not* claiming that certain translations are *impossible* to implement via pattern-based transformations. More powerful pattern-matching capabilities, and the use of multi-pass transformation processes with intermediate documents could alleviate many of the shortcomings. However, in order to solve the moving problem, the sorting problem, and the re-organization problem, we find that it is more natural and understandable to apply already well-known programming-oriented techniques. I.e., we will not insist on applying pure pattern-based transformations in situations where it is easier to write a procedure in a "general-purpose" programming language that solves the problem. This approach is also advocated in the work from Irvine [87] and in GRAMPS [14]. In GRAMPS a set of specialized procedures and functions (generated from a grammar) are used for so-called meta programming. What is called meta programming in GRAMPS includes the applications that we are discussing in this thesis. Notice though that the primitive actions in a meta program very well may activate the pattern-matcher several time to solve sub-problems.

4.1.4 Applications of Transformations

Pattern-based transformations along the lines described above have numerous applications. Many of these are described in Partsch and Steinbrüggen's survey "Program Transformation Systems" [72]. In an overall classification of these applications, various kinds of optimizations are far the most dominant in the literature. This is made very clear by Balzer and Cheatham in the introduction to a special issue on program transformations in *IEEE Transaction on Software Engineering*:

> "... the transformation field has focused on altering the performance characteristics of programs while preserving their "semantics" [...]. Thus, transformations are simply a methodology for improving program efficiency." [5].

Let us take a closer look at some typical sub-classes of optimizations, first optimization of a program that results in a semantically equivalent program in the same language [19,87]. Among such optimizations, removal of recursion, substitution of (non-recursive) procedure calls by their bodies, and removal of redundant code and redundant variables are common. To do these kinds of program manipulations, certain pre-conditions must be fulfilled. Assertions like "the function F is commutative", "the value of variable V is independent of the value of W", and "the fragment F is side-effect free" are typical examples of enabling pre-conditions of transformations. Such pre-conditions require knowledge about the objects in the program, a knowledge that goes beyond the simple syntactic knowledge of Muir. In many cases the necessary knowledge can be figured out from some kind of analysis of the program, for example, from data flow analysis.

Transformations can also be used to convert a program in an inefficient programming language to an equivalent program in a more efficient language. A good example of that is translation of Lisp programs to Fortran in TAMPR [11]. A similar application is transformation of special and unimplemented notation to already implemented constructs in the programming language. PDS [16] is an example of a system that deploys that technique. Finally, transformations are frequently used as an aid in realizing high level specifications in terms of programs in implemented programming languages. Transformational implementation of the specification language Gist is an example of that [28]. It is a major

point in this work that it isn't possible with today's technology to compile high level specifications to machine executable representations. In other words, high level specifications cannot be translated automatically to an efficient program. A semi-automatic translation, though, in which a transformation facility is used as a tool, is attractive.

With appropriate extensions along the lines described in section 4.1.3, and with the availability of semantic information, it would be possible to use the Muir transformation framework for most of the optimization tasks described above. However, the transformational applications described in this thesis have nothing to do with optimization. What we are going to describe in this and the following two chapters could be characterized as use of transformations in software engineering, or more specifically, use of transformations in construction and maintenance of grammars, programs, and other formal documents.

4.2 Structure-Oriented Edit Operations

In the rest of this chapter we will discuss an application of the Muir transformation framework, namely *structure-oriented edit operations*. A structure-oriented edit operation modifies the internal document representation. Immediately following a structure-oriented edit operation the environment must update the external screen representations in such a way that they are consistent with the modified document representation. If the screen presentation is textual, the screen updating process is similar to so-called pretty printing [71].

As an alternative to the structure-oriented model we will briefly touch on another possible model, the *presentation-oriented edit operations*. Edit operations following this model manipulate the elements in the screen presentation. Subsequently the environment must modify the internal document representation such that the presentation and the representation are consistent. For textual presentations, a presentation-oriented edit operation typically inserts or deletes a text string at a given position in the document, and the internal representation is kept up-to-date via a process known as parsing [1].

In this thesis we will only discuss structure-oriented edit operations on documents whose internal representations are abstract syntax trees. More specifically, it will be discussed how the transformation framework

that was introduced in the previous section can be used to implement a variety of structure-oriented edit operations. In an environment based on structure-oriented edit operations, the tendency seems to be that a large amount of specialized edit operations are defined and accumulated, and that most edit operations only make sense on certain types of constructs. Consequently, it is important to relate the edit operations to the constructs, on which they can be applied. In this section we will therefore also discuss how, through inheritance in the phylum hierarchy, edit operations can be associated with the constructs on which they make sense.

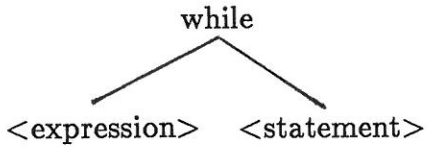
We begin by considering edit operations that can be derived directly from a grammar. Following that, the definition and application of more composite templates, edit operations of transformational nature, structure-oriented search operations, and edit operations for manipulation of lists will be treated. It will also be discussed how to deal with edit operations that cannot in a natural way be implemented as transformations. Finally, it will be exemplified how the transformation framework together with a few other means can be used to carry out major systematic modifications to a document.

4.2.1 Primitive Edit Operations

We define a primitive edit operation as one that inserts a template defined by an operator (or a BNF production) into a document. Recall that in the hierarchical grammar framework an operator is defined by a terminal phylum declaration. As an example, the AST representation of the template defined by the terminal phylum declaration

while: expression statement

from Pascal is shown in figure 4.2(a). Such a template is always a "two layer template." The operation that inserts it into a program could be implemented by the transformation shown in figure 4.2(b). The important part of this transformation is the replacement, in which the *while*-template is contained. The pattern matches any transformation focus. The way we associate primitive edit operations with phyla in the grammar assures that the edit operation only can be applied when it syntactically makes sense. (We will elaborate that below.) No explicit definition of the primitive edit operations exists in Muir. The primitive edit opera-



(a)

Transformation PrimitiveWhileEditOp(Opfocus)
Explanation "Replaces the current focus by a while template."
Pre-condition "T"
Transformation-class Primitive-expand
Focus-modification NoChange
Pattern ANY
Replacement while <expression> do <statement>

(b)

Figure 4.2: A template and a primitive edit operation.

tions are implicitly defined by the terminal phylum declarations of the grammars.

Given the notion of primitive edit operations, we now define on which constructs these edit operation can be applied:

> A primitive edit operation E induced by a terminal phylum declaration T: P_1 ... P_n can be applied on any construct N, whose choice phylum C is a superphylum of T.

We say that the grammar *associates* the primitive edit operation E with the terminal phylum symbol T, and that E is associated with every superphylum of T. We also say that E is *applicable* on the construct N. The primitive *while* edit operation, for example, is applicable on the constructs whose choice phylum is *statement*, *unlabelled*, *structured*, *repetitive*, or *while* due to the phylum hierarchy shown in figure 3.1 on page 31. Figure 4.3 illustrates the association of a primitive edit operation with a terminal phylum and all its super phyla.

In terms of classes in an object-oriented programming system, each of the most specialized classes induces a primitive edit operation that inserts an instance of itself into an AST. Such operations are applied on unexpanded nodes, which correspond to instances of more general classes. Somehow the primitive edit operations must be made applicable on instances of all the super classes of the classes from which they are induced. This can either be achieved by a general "expand operation" that takes a parameter, or via bottom-up propagation of operations to super classes in the class hierarchy.

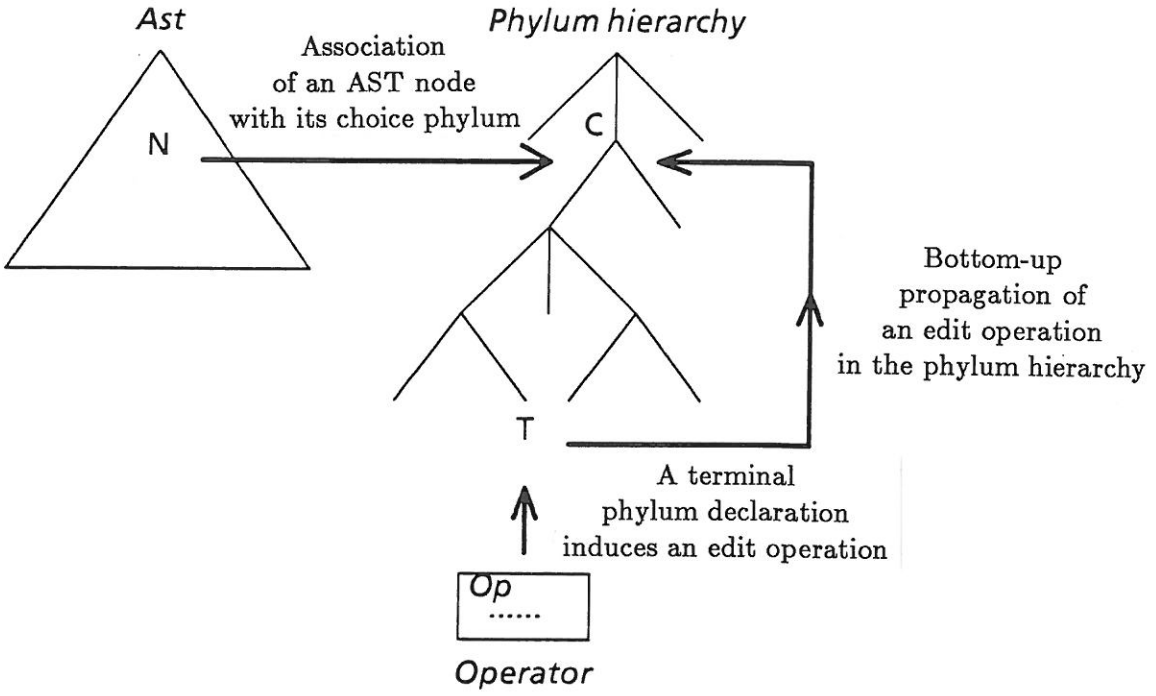


Figure 4.3: Association of a primitive edit operation.

Every document can be constructed exclusively by using primitive edit operations. However, creation and modification using only the set of primitive edit operations is clearly too restrictive for practical editing environments. We even claim that satisfactory and realistic syntax-directed editors cannot be generated automatically from a grammar of a language. Typically, certain combinations of constructs are used so often that it is waste of time and tedious for the user to create them via primitive edit operations every time they are needed. Furthermore, some frequently occurring changes require several primitive edit operations to be carried out. In such situations a single transformational edit operation, which does the same manipulation as the sequence of primitive edit operations, greatly improves the editor. Thus, the need for more elaborate and "advanced" edit operations is pressing, but it seems nearly impossible to define a sufficient set of non-primitive edit operations once and for all. Practical use of the editor should reveal the demands, and it should be easy to extend the set of edit operations while the editor is in use.

In the following sections we will discuss how a flexible repertoire of edit operations can be implemented in terms of the Muir transformation framework, and how such edit operations can be associated with the constructs on which they make sense. We start with a discussion of how

(a) **Procedure** <nameDcl>(<parameter-list>);
<block>

(b) **Procedure** <nameDcl>(<nameDcl>: <type-name>);
<constant-declarations>
<type-declarations>
Var <nameDcl>,<nameDcl>: <type>;
<nameDcl>,<nameDcl>: <type>
<routine-declaration-list>
Begin
<statement>;
<statement>;
<statement>
End

Figure 4.4: Primitive and composite template for a procedure definition.

to handle more composite templates than the two level templates defined by the terminal phylum declarations.

4.2.2 Insertion of Composite Templates

Typically, some *composite templates* are needed more frequently than others during creation of a document. As an example, compare the template in figure 4.4(b) with the template in figure 4.4(a), which is induced by the operator *Procedure* in the Pascal grammar. It can probably be proved empirically that most procedure declarations contain at least one parameter, some local variables, and a few statements. Instead of creating such frequently used templates by combining primitive edit operations every time they are needed, composite templates should be made directly available as a supplement to the primitive edit operations. If some of the constituents of the composite template are superfluous, they are easy to remove or eliminate. In fact we believe that most people find it easier to remove something existing, than to insert something that does not exist yet. Moreover, many unexpanded placeholders can be eliminated automatically if they remain unrefined. (In the Muir environment we have an edit operation called *Reduce▷UnexpandedPlaceholders* that does this. In the Synthesizer Generator [80], the notion of "completing terms" may help eliminate unexpanded placeholders.)

In Muir, an arbitrary document or substructure of a document can be declared as a composite template. In doing that the system defines a

Transformation *procedure(Opfocus)
Explanation "Replaces the current focus by a template"
Pre-condition "T"
Transformation-class Primitive-expand
Focus-modification NoChange
Pattern ANY
Replacement **Procedure** <nameDcl>(<nameDcl>: <type-name>);
<constant-declarations>
<type-declarations>
Var <nameDcl>,<nameDcl>: <type>;
<nameDcl>,<nameDcl>: <type>
<routine-declaration-list>
Begin
<statement>;
<statement>;
<statement>
End

Figure 4.5: A Transformation that inserts a composite template.

transformation as the one shown in figure 4.5, in which the replacement is identical to the selected substructure of the document. This transformation is in principle similar to the one shown in figure 4.2(b). The user is asked to give the template a name. A composite template is legal whenever the primitive edit operation induced by the identification phylum of the root of the template is applicable. If a procedure is a legal alternative during editing of a Pascal program, the system will also make the transformation **procedure* available in the menu. (For an example of a menu that contains composite templates, see section 3.2.5.) If applied, the replacement of the transformation is simply substituted for the current focus.

We have found the composite template facility very useful in the Muir environment, and at present, approximately 50 such templates have been defined for the formalisms that we support. As a modest extension, it would be useful to allow the user to augment the system's template list with a personal set of templates. Also, one could imagine application-specific templates that could be brought into the environment whenever a document belonging to a given application area is edited.

(a) **Transformation** NestIn-if-then(Opfocus)
Explanation "Nests a statement into an if-then construct"
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern ANY of SyntaxCategory statement
Replacement if <expression>
then MATCH

(b) **Transformation** Unnest-if-then(Opfocus)
Explanation "Unnests an if-then construct"
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern if <expression>
then [S: <statement>]
Replacement [S: <Anything>]

(c) **Transformation** procedure→function(Opfocus)
Explanation "Converts a procedure to a function"
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern Procedure [N: <nameDcl>]([PL: <parameter-list>]);
[B: <block>]
Replacement Function [N: <nameDcl>]([PL: <parameter-list>]): <type-name>;
[B: <block>]

Figure 4.6: Examples of transformational edit operations.

4.2.3 Transformational Edit Operations

In the previous sections we have described edit operations that *create* new constructs. In this section we will discuss edit operations that *modify* existing constructs. Such edit operations can loosely be classified as nest operations, extract operations, and proper transformations, all of which can be implemented in terms of the transformation framework described in section 4.1. Figure 4.6 shows examples of transformations that belong to these three classes. The nest operation in figure 4.6(a) makes a statement in a Pascal program conditional. The pattern matches a statement, and the "MATCH" indication in the replacement causes the match to be copied into the then-part of the if-then statement. The transformation in figure 4.6(b) is the inverse operation of *NestIn-if-then*. If applied on

an if-then statement, the statement called "S" will be substituted for the if-then construct. The phylum *Anything* in the replacement reflects that the choice phylum of the replacement is *System.Anything* (see section 3.1.4.) Finally, in the procedure-to-function conversion in figure 4.6(c) the pattern enumerates the three constituents of a procedure such that they can be copied into the function template.

Edit operations for modification—like those in figure 4.6—should also be associated with the phyla of a grammar, and it should be possible to tell on which constructs the transformational edit operations are applicable. The definer of an edit operation must associate the operation with one or more phylum symbols from the phylum hierarchy, and the system will then use the following propagation rule:

> If an edit operation E is associated with a phylum symbol P, E will in addition be associated with all the subphyla of P in the phylum hierarchy.

The edit operation E is applicable on a node N if E is associated with the identification phylum of N, i.e., if the identification phylum of N is a subphylum of the phylum symbol, to which E was associated directly. Compared with object-oriented programming systems, the transformational edit operations are inherited down in the class hierarchy, which corresponds to the phylum hierarchy defined by a hierarchical grammar. Figure 4.7 illustrates the association of an edit operation to a phylum and the top-down inheritance.

The nest operation in figure 4.6 is associated with the phylum *statement*. The unnest operation is only meaningful on if-then statements, so it is associated with the terminal phylum *if-then*. The procedure-to-function conversion only makes sense on procedures, and therefore it is associated with the terminal phylum *procedure*. Notice that in all three cases the pattern-expressions suggest the association of the edit operations. (The patterns are essential in all but the nest transformation in figure 4.6(a). The pattern in the nest transformation could be "ANY", as long as the transformation only is applied on a statement. The pattern "ANY of SyntaxCategory statement" matches any statement in the focus.)

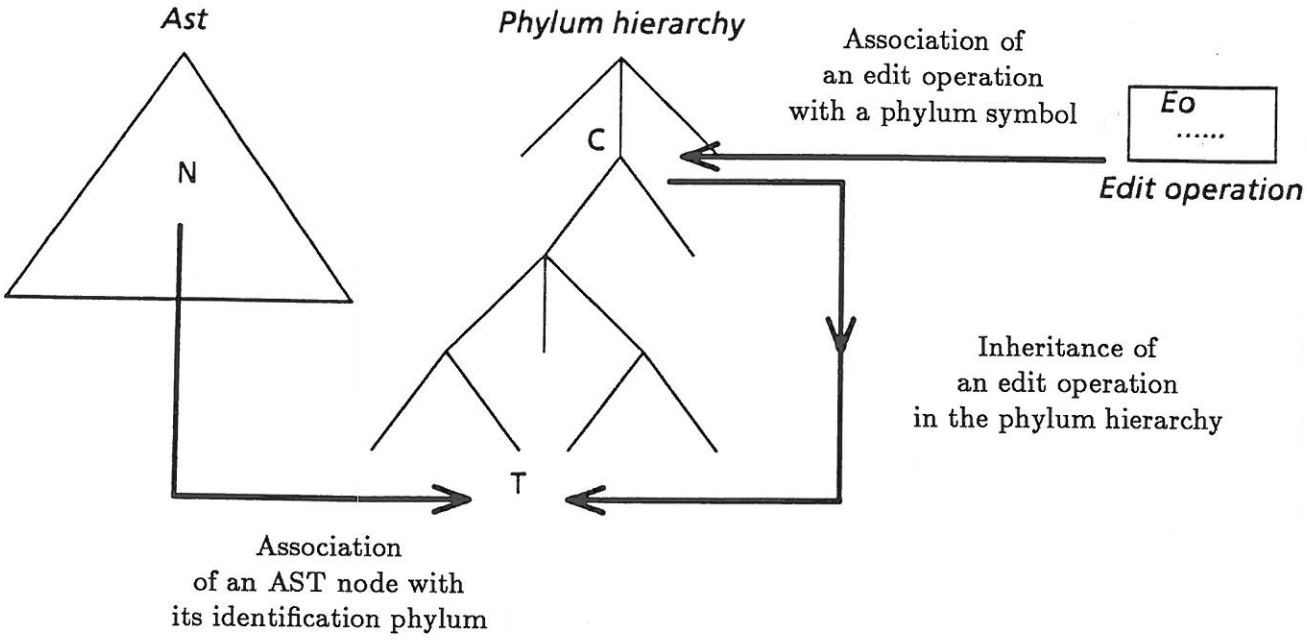


Figure 4.7: Top-down inheritance of an edit operation in a phylum hierarchy.

4.2.4 Structure-Oriented Search Operations

It is well-known that some search operations can be made more powerful on an abstract document representation than on a textual document representation (see, for example, [23].) In an abstract representation it is possible to distinguish some structures that cannot easily be distinguished in a textual representation. In terms of the Muir transformation framework a structure-oriented search operation can be understood as a transformation, in which the pattern and the pre-condition specify the objects for which to search, and in which the replacement equals the matches (see section 4.1.1.)

As an example of how to use the structure-oriented search facilities in Muir, let us assume that we in a Pascal program want to locate all activations of the standard procedure "Write" with two or more arguments, of which the first is "F". Thus, we want to match the procedure calls "Write(F,a)", "Write(F,a,b)", but not "Write(F)". Furthermore, we do not want to match the string "Write(F,a)" in a context in which it isn't a procedure call. The following steps represent a possible way to solve this problem:

1. Select an example of a construct in a Pascal program that should

(a) **Transformation** StructureSearch(Opfocus)
Explanation "Searches for the pattern described in the pattern part."
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern Write(F,a)
Replacement MATCH

(b) **Transformation** StructureSearch(Opfocus)
Explanation "Searches for the pattern described in the pattern part."
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern Write(F,<expression>,<expression-list>)
Replacement MATCH

Figure 4.8: Transformations that implement search operations.

be matched by the structure-oriented search operation. We could, for example, select the procedure call "Write(F,a)".

2. Activate the operation *Search* on the selected construct. This operation opens a new Sedit instance with a structure-oriented search operation, which matches the construct selected in 1. The structure-oriented search operation is shown in figure 4.8(a).

3. Refine the pattern of the search operation, if necessary. In the example we refine the pattern, and we get the search operation shown in figure 4.8(b).

4. Finally, activate the search operation on the desired program fragment. (This can be done with *DoEditOp*, see section 3.2.5.)

If, at a later time, the matches located by the search operation in figure 4.8(b) need to be modified, it is easy to turn the search operation into a "real" transformation.

Showing the actual matches located by a structure-oriented search operation is not always straightforward in Muir, because a document can be presented in such a way that some details are left out. If we initiate a search operation from an abstract presentation where the matches are not presented, either the nearest enclosing constructs in the abstract

presentation can be highlighted one at a time, or a set of new and more detailed Seditor instances can be set up around each of the matches. If the matches happen to be presented in the presentation from which the search was initiated, they can naturally be shown one at a time.

The structure-oriented search operations shown in figure 4.8 are both of class *primitive*. It means that matches inside a match are not located. It should be possible to locate all matches by using the transformation class *general*. However, in version 1 of Muir, it will cause the pattern matcher to loop, because it immediately will find the match again, activate itself on this match, find it again, etc. Notice that this functionality sometimes is useful in transformations if the replacement is different from the matches. It might be worthwhile to support "in depth" structure-oriented searching as a special case.

4.2.5 Edit Operations on Lists

As described in section 3.2.1, linear list structures are in Muir represented as nested binary trees of heads and tails. If only primitive edit operations are used, this representation is awkward for editing. In this section we will describe how a set of generic and more adequate list manipulation operations have been implemented in terms of the transformation framework. These include an operation that creates a list, operations that insert list elements before and after an existing element, and an operation that deletes a list element. To be concrete we will take a closer look at the operation that adds elements in front of an existing list element.

Adding an element E' in front of an existing element, say in front of E_2 in figure 4.9, can be done by nesting the father of E_2 into the second constituent of the tree rooted R (see the figure.) A transformation that implements this nest operation for a given list type, say for a list of names, is straightforward to create. Such an operation is quite similar to the nest operation shown in figure 4.6(a). However, we want to implement a *generic operation*, i.e., an operation that can do the job independent of the syntactic type of the list. Thus, the operation we are looking for should be able to extend a list of names, a list of statements, etc. To obtain this generality the replacement constituent of the transformation must necessarily depend on the construct, on which the transformation is activated. Figure 4.10 shows such a transformation.

The transformation in figure 4.10 is intended to be activated on an

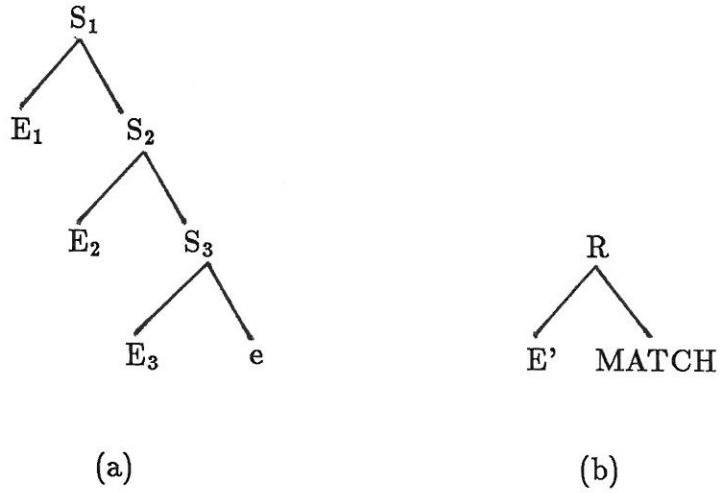


Figure 4.9: Illustration of ASTs for adding E' in front of E_2.

```
Transformation AddElementBefore(Opfocus)
Explanation "Adds a list element in front of the current focus"
Pre-condition "(LISTELEMENT? Opfocus)"
Transformation-class General
Focus-modification "(SUPERTREE Opfocus)"
Pattern ANY
Replacement
  "(PROG (R)
     (SETQ R (OPERATORAST (OPERATORDEFINITION Opfocus)))
     (ASTREPLACE
        (SUBTREE R 2)
        (OPERATORAST (GRAMMARREFERENCE 'match 'MT 1) )
        'NOCHECK )
     (RETURN R) ) "
```

Figure 4.10: A generic list-adding transformation.

existing list element, for example on E_2 in figure 4.9. First, it moves the transformation focus to the father of the list element (from E_2 to S_2 in figure 4.9.) The pattern and the pre-condition matches any list element. The replacement is a Lisp expression. In general, the pattern and the replacement of a transformation are allowed to be AST-returning Lisp expressions. Such a Lisp expression is evaluated in the context of the parameter bindings of the transformation. In the example the Lisp expression returns the template shown in figure 4.9(b). The Lisp code has access to the transformation focus *Opfocus*, and through this, to the syntactic type of the list. The already described functionality of the transformation framework, especially the meaning of "MATCH" (see section 4.1.1) implies that E' is inserted in front of E_2.

Edit operations like *AddElementBefore* must be applicable on all list elements in a document. However, list elements are typically not distinguished syntactically from non-list elements. For example, a Pascal statement is in some contexts a list element, but in others it is not. So it is the context of the construct rather than its identification phylum or its choice phylum that determines whether the list operation makes sense. It would be possible to elaborate the abstract grammar in such a way that list elements syntactically could be distinguished from non-list elements. The phylum *statement* in Pascal, for example, should be split into *single-statement* and *statement-in-list*. Edit operations on list elements could then be associated with a common superphylum of the *x-in-list* phyla, where x is a phylum, for which there exists list structures. However, we think that a clean abstract grammar is more important than one that facilitates an elegant and a simple association of list-oriented edit operations to phyla. So in Muir we have chosen to accept the association of list edit operations to phyla as a special case. In other words we use ad hoc association of list edit operations to phyla.

4.2.6 Programmed Edit Operations

Although the majority of the structure-oriented edit operations can be implemented as pattern-replacement based transformations, there is a "rest group" of operations that cannot in a natural way be realized as instances of Muir transformations. Some of these are of "transformational nature", but because of limitations in the expressiveness of the transformation framework (see section 4.1.3), they cannot be implemented as

transformations. Other edit operations, such as copy operations and re-present[5] operations are not of transformational nature. Finally, it is often useful to define generalized "edit operations" that support other purposes than editing.

In order to deal with the "rest group" of transformations, we allow certain Lisp functions to be activated as edit operations. A Lisp function can be used as an edit operations if (1) its first parameter accepts a list of ASTs (the operation focus), (2) it returns a list of ASTs, and (3) the name of the function is of the form *grammar.editOpName*, where *grammar* is the name of the grammar, to which the edit operation belongs. In addition, the function must explicitly request screen updating, such that the screen can be properly updated after the edit operation has been completed. (See section 4.1.1 and section 7.4 for a discussion of our approach to screen updating.) Edit operations that obey these rules can be activated in exactly the same way as a transformation. As with transformations, we declare functional edit operations in the grammar, and it is also possible to associate them with phyla in the grammar. Like transformational edit operations, programmed edit operations are inherited in the phylum hierarchy, and hereby associated with all subphyla of the phyla, to which the operation explicitly has been associated.

The declaration of a functional edit operation merely consists of the name of the edit operation and an explanation. However, "meta edit operations" on such a declaration make it possible to create and present the Lisp function that corresponds to it. In a simple way this ties together the Muir environment and Dedit [48] in the Interlisp environment.

4.2.7 Major Systematic Modifications

Our approach to definition of edit operations is based on creation of many specialized transformations or Lisp functions, and association of these operations with the phyla, on whose elements the operations make sense. This is in contrast to other structure-oriented editors that provide a few but general structure-oriented operations. Dedit from the Interlisp-D environment [48] is an example of such an editor.[6] There is clearly a limit

[5] Re-present presents the selected fragment of a document in another window, using the same or another presentation scheme and/or style.

[6] In Dedit, however, it is also possible to apply a wide variety of specialized and powerful commands. These commands were originally defined in *Edit*, the teletype-oriented predecessor of Dedit.

for how far our approach can be, and should be pushed. At some level, the user *must* "combine" the elementary operations and the elementary mechanisms that the system provides. Combination of elementary operations is in Muir straightforward syntax-directed editing. Combination of elementary mechanisms, however, is more like an idiom that has to be learned. In this section we will exemplify how the transformation mechanism, the structure-oriented search mechanism, and the so-called AST-stack (explained below) in combination can be used to relieve the burden of carrying some major systematic modifications to a document.

The example that we will consider is adding an extra parameter to a set of procedure declarations and their activations. This involves at least the following manual decisions: (1) Selection of the affected procedures, (2) selection of the position of the new parameter, (3) choice of the formal name and the type of the parameter, and (4) choice of the actual parameter expressions in the procedure activations. If only a few declarations and activations are involved, it is probably easiest to carry out the modification by hand using "standard edit operations." If, on the other hand, tens of procedures and hundreds of procedure activations are involved it is clearly time-saving and less error prone to apply a more systematic modification technique. In Muir we would do the following to carry out the task:

1. Collect the affected procedure declarations. Concretely we select the procedures one after the other, and in order to "hold on to them", we push references to them onto the AST-stack.

2. Apply a transformation on the collection of procedures. The transformation should add the formal parameter.

3. Collect the affected procedure activations. The procedure activations can be collected manually (by selecting them one at a time and pushing references onto the AST stack), or they can be collected by repeated application of structure-oriented search operations (see 4.2.4.)

4. Apply a transformation that adds an argument template (or, if possible, the argument itself) to the collected set of procedure calls.

5. If necessary, visit each of the modifications and adapt them appropriately. This is straightforward because references to all modifications are left on the stack by step 4.

As can be seen, the *AST-stack* is central to the described technique. An element on the AST-stack is a list of (references to) ASTs. The ASTs can be referred to from other contexts as well; for example, they can be sub-structures of documents, which currently are manipulated in the environment. Push, pop, copy, and re-present operations (among others) are available on the stack. The result of a transformation or a structure-oriented search operation can also be pushed onto the AST-stack. Furthermore, if the top of the stack is a transformation, this transformation can be executed on the ASTs referred to by the stack frame next to the top. Hereby the AST-stack is popped twice, and the result of the transformation is pushed onto the stack.

We believe that a wide variety of systematic modifications can be carried out interactively using transformations, structure-oriented search operations, and the AST-stack, in ways similar to the example described above. In appendix C, we describe and illustrate in great detail another example of a systematic modification. Once a task has been completed it is easy to carry out a similar task in the future if the involved transformations and search operation templates are filed.

4.3 Summary

In the first section of this chapter we described the capabilities and the limitations of the pattern-based transformation framework. One of the things that distinguish our work from other's is the way we represent transformations as multi-formalism ASTs. This contributes to the uniformity of the system, because transformations can be handled in the same way as any other piece of information by the environment.

As a major application of the transformation framework, we have described how most structure-oriented edit operations can be understood and/or implemented in terms of pattern-based transformations. We have argued that besides the primitive two-layer templates defined by the operators, more composite templates should be made available. We find that Muir provides a particularly elegant solution to the definition and use of composite templates. Transformational edit operations, such as nestings and extractions, are obviously straightforward to implement as pattern-based transformations.

Edit operations that insert primitive templates are implicitly defined

by the terminal phylum declarations, and they are propagated up in the phylum hierarchy to all their superphyla. Such edit operations are made available through the choice phylum of a construct. Edit operations that modify already existing constructs are also associated with phyla, but they are inherited top-down in the phylum hierarchy, and they are made available through the identification phylum of a construct.

We have also seen that structure-oriented search operations can be implemented in the transformation framework, namely by considering the replacement as equal to the matches. More dynamic edit operations, in which the pattern or the replacement depend on the transformation focus have been exemplified through generic list edit operations. Finally, we have discussed the concept of 'major systematic modifications', a technique to carry out some "tedious" modifications by systematic, interactive, and transformational means.

Chapter 5

Keeping ASTs Consistent with the Grammar

In chapter four we described an application of transformations that is relevant for every editing environment, namely how to implement edit operations in terms of transformations. In this chapter we will focus on an application of particular importance for a language development environment. The problem we will study is how to keep a set of documents consistent with a grammar that is under development. We are only concerned with modification of the abstract grammar. Modification of the concrete grammar, which in Muir is a matter of presentation, can be done at any time without affecting any existing documents inside or outside the environment (see section 3.2.3 and chapter 7.)

As an example of a problem we will deal with, let us assume that an extra argument is added to an operator of a grammar (or equivalently that an extra constituent is added to a terminal phylum declaration.) After this grammar modification, all existing documents that apply the operator are inconsistent, and the documents need to be updated in order to reestablish their syntactic integrity. During a language development process we frequently want to carry out such grammar modifications. In DOSE[1] [30], an environment for language design by Feiler et al., only detection of inconsistencies is supported. DOSE does not support updating of existing documents. I.e., ASTs are discarded if they are found to be inconsistent with the grammar. We find it crucial that the document updating process is supported by the environment in a systematic way. A manual updating process of the documents via some kind of editing is typically overwhelming.

[1]DOSE and Muir have been developed independently of each other.

Although the problem is of special importance in a language development environment, there seems to be widespread agreement among people who work with syntax-directed editing that similar problems exist in any syntax-directed editor. Very few grammars, even grammars for well-known and stable programming languages, have been implemented "just right" at the first try in a syntax-directed editor generator. We will in this chapter discuss a transformational approach to the consistency problem.

5.1 A Priori Limitations

Before we dive into the problems, we will describe a few limitations that reflect our ambition level, and which therefore have a profound effect on the rest of this chapter. The facilities for keeping a hierarchical grammar and its dependent documents consistent are intended to deal with *grammar perturbations* rather than major changes. If extensive grammar modifications have taken place, the resulting grammar should probably be considered as describing a new language, and other means should be used to transform existing documents. The multi-formalism transformation facility, which will be described in chapter 6 of this thesis, is relevant in dealing with more extensive grammar modifications.

We will assume that there exists a relatively straightforward, legal, and context free substitution for constructs in documents that become outdated because of grammar modifications. If that is not the case, it is difficult or impossible to formulate a transformation that remedies the inconsistencies between the grammar and the documents. It should also be noticed that we are not about to propose a facility that always preserves the semantic correctness of documents that depend on the grammar. Our approach only deals with syntactic aspects.

The grain-size of the *grammar modification steps* is of importance for the following discussion. We will restrict a grammar modification step to either (1) add a new phylum to the phylum hierarchy, (2) modify an existing terminal phylum declaration (operator), or (3) delete an existing phylum from the phylum hierarchy. Every grammar modification can be carried out as a sequence of such primitive modification steps. For example, a replacement of a phylum symbol with another phylum symbol can be accomplished by a deletion followed by an addition.

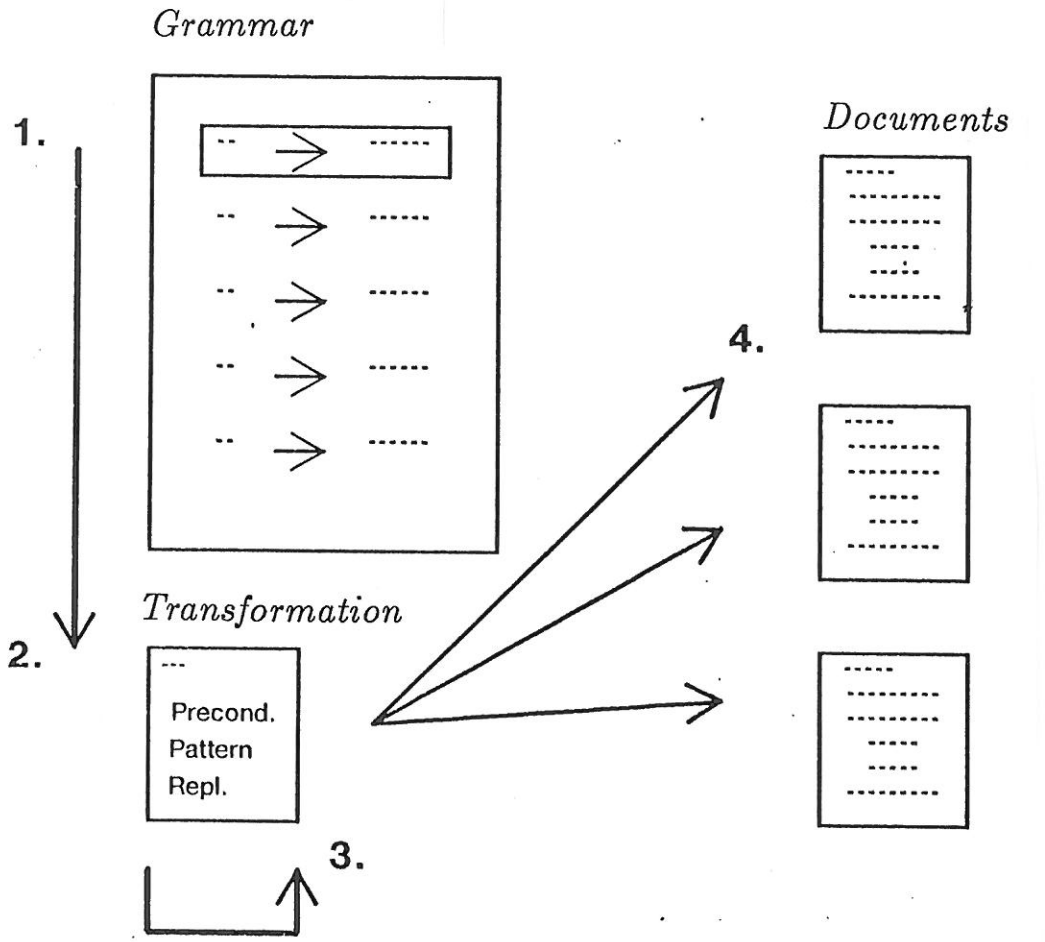
We will *not* assume that it is possible at a given time to process all documents that depend on a grammar. In other words, there are no "pointers" from a grammar to the set of documents that depend on the grammar (but the opposite links are of course present.) It means that the effect of the grammar modifications (basically the transformations) must be stored until we are confident that all dependent documents have been processed.

5.2 The Basic Approach

We mentioned in the introduction that our basic approach in dealing with the consistency problem between grammar and documents is transformational. Let us now describe in more detail what we mean by that.

After the grammar definer has performed a grammar modification step on a grammar G, the modification must be brought into effect in the editing environment. The grammar modifications must, for example, be reflected in the functionality of the syntax-directed editor for G-documents. We will assume that it is well-defined when this happens, for example that the grammar definer explicitly tells the environment when the grammar modification should be effective. As a side effect of this process, the system will create one or more transformation templates for bringing dependent documents up to date. The transformations will be presented for the grammar definer, who then can refine the transformations if necessary. Whenever, in the future, a document with outdated constructs is encountered, the environment can automatically locate the relevant updating transformations, and it can attempt to apply them on the obsolete constructs, hereby bringing them up to date. This grammar modification and document updating model is shown schematically in figure 5.1.

It is instructive to compare the basic grammar modification and document updating technique, as described above, with commonly found ad hoc solutions. A 100% manual identification and updating process is of course the most primitive ad hoc solution one can think of. If that is too big a task, it can be attempted to write a specialized piece of program that searches for outdated constructs, and which updates the document representation appropriately. This solution is very tedious if the documents are represented as text. Even if the documents are represented as



1. The *user* modifies an element in a grammar.
2. The *system* creates a transformation template.
3. The *user* refines the transformation template.
4. The *system* applies the transformation to update the documents, which depend on the grammar.

Figure 5.1: Grammar modification and document updating.

ASTs the AST "patching code" must be written on a case-by-case basis. I.e., a future modification of the grammar will probably demand a new pice of document-updating program.

Techniques for identification of outdated constructs in documents play a key role for the proposed solution of the problem. We will in the next section discuss how a fine-grained version concept of grammatical elements can be used to identify outdated constructs.

5.3 Versions of Grammatical Elements

Because each construct in a document refers to a phylum symbol in a hierarchical grammar, a version concept for grammatical units seems to provide an efficient way to identify outdated constructs. The basic idea is to register a version number for each terminal phylum declaration (operator)[2] of a grammar. When a construct is created in a document, the version number of its identification phylum should be stored in the construct. Each AST-node contains a field in which the version number of the construct can be stored (see section 3.2.1.) If the grammar at some time in the future is modified in such a way that the operator is affected, the version number of the operator will be incremented. If the operator is eliminated, it will be marked in a special way. A comparison of the stored version number of the construct with the version number of the operator will immediately unveil the inconsistency.

For our purpose, thus, a fine grained version concept for phyla is a key to the solution of the consistency problem. A more total version concept for grammars would not be very helpful, because we want to deal with grammar perturbations like those described in section 5.1, and not more coarse grained grammar modification steps. It is not immediately clear, however, how much overhead, in terms of information about older versions of phylum declaration, the version administration requires. Furthermore, handling of the terminal phyla, as discussed above, seems to be relatively unproblematic, whereas a similar notion of versions for categorical phyla is harder to deal with. We will now in turn discuss each of these subjects.

[2]Because there is a one-to-one correspondence between terminal phylum declarations, leaf phylum symbols in the phylum hierarchy, and operators, we will also talk about "versions of operators" and "versions of phylum symbols."

5.3.1 Version Overhead

Let us assume that we keep track the current version of a terminal phylum declaration, and that we in the AST nodes register the version of the phylum symbol that was used during the creation of the construct. This is the idea already sketched above. If the terminal phylum T is a subphylum of P, all the future versions of T will be considered as subphyla of P. I.e., the version of a terminal phylum T does not affect T's placement in the phylum hierarchy. The version mechanism allows us to identify outdated constructs in documents. If, however, we for some reason want to manipulate a document before it is updated w.r.t. its grammar, it is a reasonable requirement that the environment should be able to handle the outdated constructs properly. Most important, the system should be capable of presenting outdated constructs in a satisfactory way until they have been eliminated. If the presentation rules are associated with the operators, and if some operators are modified, the environment will not be able to present instances of older-version operators. In order to do that, at least the presentation rules of the older-version operators must be kept around in the environment.

If we furthermore want to support editing, in which obsolete constructs can be inserted into a document, we also have to store information about the abstract structure of outdated operators. However, we consider the grammar modifications to reflect development steps of a single language, as opposed to a whole sequence of language versions. It means that only the most up-to-date constructs are relevant during editing of "normal documents." The documents that are exceptions in that respect are the transformations that keep "normal documents" up to date. The pattern-parts of such transformations are normally instances of obsolete constructs.

The preceding argumentation apparently indicates that the version administration of the terminal phylum declarations imposes a considerable storage overhead on the grammars. However, by relaxing the requirements a little bit, it is possible to get rid of nearly all the overhead. We will now take a closer look at how this can be done.

An accurate presentation of outdated constructs in documents is clearly desirable, but it is possible to present any AST structure in a standard way, without using stored presentation schemes at all. As an example, figure 5.2(a) shows the standard presentation of a Pascal program frag-

```
if Pointer < StackMaxSize
   then Begin
           Pointer := Pointer +
           1;
           Store[Pointer] := El
        End
else StackError
```

(a)

```
if-then-else
   Pointer < StackMaxSize
   begin..end
      statement-list
         assignment
            Pointer
            plus
               Pointer
               1

      statement-list
         assignment
            Store[Pointer]
            El

procedure-call
   StackError
```

(b)

Figure 5.2: Standard and "skeletal" presentation of a fragment.

ment, and figure 5.2(b) shows the presentation of the same fragment in the alternative "skeletal" way. The latter presentation has been created without use of any stored presentation schemes. The skeletal presentation scheme can be used for outdated constructs, and the normal presentation schemes should only be used to present constructs that are up to date. "Normal" presentation and skeletal presentation will typically be used side by side in the presentation of a document. We think that the skeletal presentation scheme is sufficiently readable, and in addition, it is sufficiently separable from other presentation schemes, so that it highlights outdated constructs as a contrast to the constructs that are up to date. (The environment, or more specifically, the presentation framework, could of course support this highlighting in another way.)

Instead of supporting insertion of obsolete constructs, such constructs can be copied from already existing documents to, for example, the transformations, where they are required. The problem is furthermore reduced because the system in most cases is able to construct templates of transformations, and hereby to insert old-version constructs in the pattern-part of the transformations, just before the grammar modification is carried out.

In summary, the proposed solution does not cause much storage overhead on the grammar. It is of course necessary to store the transformations that result from the grammar modification steps. On the other hand, each node in an AST must store the version number of the construct, and it means that each document, both in internal and external representation, requires more storage space. The increased size of the ASTs seems to be impossible to avoid in our approach.

5.3.2 Versions of Categorical Phyla

The categorical phylum declarations describe the interior of the phylum hierarchy. If the interior of the phylum hierarchy is modified, the syntactic validity of constructs in dependent documents may be affected. The question is whether it is relevant, meaningful, or helpful to deal with versions of categorical phylum declarations, i.e., versions of the syntactic rules that determine if constructs are valid in their contexts.

Let us first notice that versions of categorical phylum declarations are more fuzzy than versions of terminal phylum declarations. If the categorical phylum declaration of P is modified, say an immediate subphylum of it is deleted, it affects all the superphyla of P in the phylum hierarchy. The reason is that the equation $\varphi(\mathrm{C}) \subset^* \omega(\mathrm{C})$, which describes the syntactic constraints imposed on the construct C, involves the transitive closure of the immediate subphylum relation. So it is hardly meaningful to talk about the version of a categorical phylum declaration. We should rather talk about versions of a sub-hierarchy of the phylum hierarchy. This, we feel, will be too circumstantial, and we therefore try to avoid it all together. To do this, let us now discuss how modifications of the interior of the phylum hierarchy can be reflected at the terminal level of the hierarchy.

If a categorical phylum declaration of P is modified, i.e., if the immediate subphylum relation between P and other phyla is changed, the so-called operator-set of P may be (and typically is) affected. *Operator-set*(P) is defined as the set of operator names that can be reached from P via the phylum-subphylum relation. More formally, if P is a categorical phylum symbol then

$$\textit{Operator-set}(\mathrm{P}) = \bigcup_{p \subset P} \textit{Operator-set}(p).$$

If P is a terminal phylum symbol, *Operator-set*(P) = {P}. If, after a

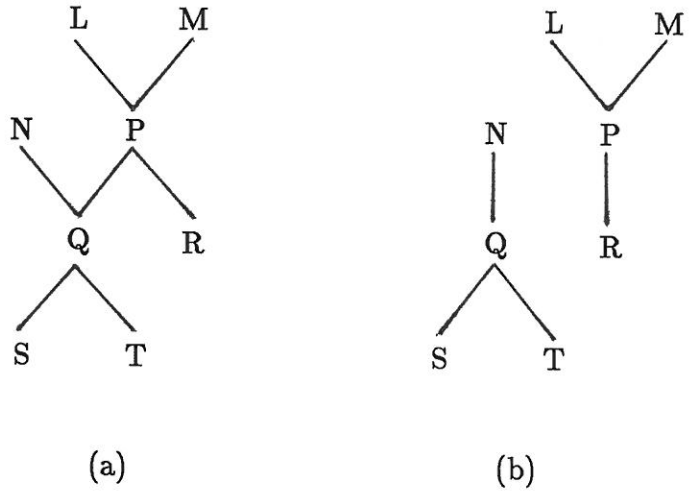


Figure 5.3: Pruning a phylum hierarchy (a) to that in (b).

modification of the categorical phylum declaration of P, *Operator-set*(P) is restricted compared with its value before the phylum modification, documents that refer to P may be affected. If, on the other hand, the operator set of P is extended, the grammar modification will not affect any document. As an example, let C be a construct in a document, and let us assume that the phylum hierarchy rooted by ω(C) is modified such that φ(C) no longer belongs to *Operator-set*(ω(C)). In that case, the construct C is invalid *in the given context*, but in other contexts C may be perfectly valid.

Let us consider a more concrete example to illustrate the effect of a phylum restriction. We will assume that the phylum hierarchy in figure 5.3(a) is modified to the phylum hierarchy in figure 5.3(b). The categorical phylum P has been pruned such that Q no longer is a subphylum of P. If A is an AST, and if ω(A) is a superphylum of P (i.e., it is either L, M, or P), then A might be syntactically invalid. More specifically, A is invalid if it is either a T-construct or an S-construct. T-constructs and S-constructs are nonetheless valid if applied from a context whose choice phylum is either N, Q, T, or S. So T-constructs and S-constructs are not globally invalid, as it would have been the case if they had been eliminated at the terminal level of the phylum hierarchy. It means that we cannot mark them as deleted. Only T-constructs and S-constructs whose choice phyla are superphyla of P should be transformed to something else.

It is interesting to notice that the phylum hierarchy can be re-organized

without affecting any dependent document, as long as the operator-set of each categorical phylum isn't restricted. This is because the choice phylum of a construct C is defined via the context of C. There are strong bindings between an AST and the identification phyla of the nodes in the AST, all of which are terminal phylum symbols. Between the AST nodes and the categorical phyla, on the other hand, there are no strong bindings.

5.4 Structure vs. Text-Oriented Solution

We have until now solely discussed the inconsistency problem for structure-oriented environments. Before we continue with a more detailed account on the elaboration of the facilities in Muir, we will briefly compare structure-oriented and text-oriented environments w.r.t. the subject.

Let us assume that we work with a grammar-based tool in a text-oriented environment, and that we in a similar way as discussed above modify the grammar while there exist some textual-represented documents that depend on the grammar. The tool could be a compiler, a pretty-printer, or a syntax-directed editor. Because the environment is text-oriented, the documents are represented as text and parsed before they are manipulated by the tool. In such a setting, the grammar can be modified arbitrarily as long as it describes the same language, (i.e., as long the modified grammar is weakly equivalent with the old grammar.) A proper extension of the language does not harm either. The parse process will always create the most up-to-date parse tree. Even in a structure-oriented environment that stores documents as ASTs it would be possible to cope with such grammar modifications by doing pretty printing of documents, grammar modifications, and finally re-parsing of the documents. If, on the other hand, the grammar modification changes the language (beyond proper extension), we have the same kinds of problems in text-oriented environments as described above for environments that are based on AST-representation of documents. In fact, the problems are even worse, because the textual document representation in addition is sensitive to changes of the concrete syntax of the language. As already mentioned, this is not the case for the AST representation of documents. So all in all, we find that AST-based environments are superior to text-based environments in dealing with the grammar/document

consistency problem.

5.5 The Solution in Muir

Based on the analysis in the previous sections we have implemented a prototype facility, which is based on versions of terminal phylum declarations. As explained in section 3.2.2, a hierarchical grammar is in Muir represented as an AST in a meta formalism. Before a grammar G can be used to support editing of G-documents, the grammar-AST of G must be installed. During the installation, some of the information in the grammar-AST is condensed such that the editing environment is able to use it efficiently. No grammar tables are generated as the result of the installation. In a terminal phylum declaration, for example, the presentation rules are compiled and stored as tree properties of the declaration. Also the AST-template defined by the operator is constructed and stored as a tree property of the corresponding terminal phylum declaration. In version 1 of Muir, terminal phylum declarations can be installed incrementally, but changes to the categorical phylum declarations must be brought into effect via a total installation. This corresponds roughly to incremental and total compilation of a program.

In order to deal with versions of operators, we have introduced two new operations on terminal phylum declarations: *InstallNewVersion* and *MarkAsDeleted*. Application of these operations signals that grammar modifications should be brought into effect. *InstallNewVersion* increments the version number of the phylum declaration, and as a side effect, a transformation-template is created and stored as part of the grammar. *MarkAsDeleted* marks the terminal phylum as deleted, and it also produces a transformation-template in which the elimination can be programmed. In addition to these two operations, we have found it useful to have an operation *MakeCategorical* that supports the process of turning a terminal phylum into a categorical phylum. Finally, we have an operation *PhylumRestriction* that can report on "phylum restrictions" after the interior of the phylum hierarchy has been modified.

To add more concreteness to the discussion we will now look at some examples. All the examples in this chapter are somewhat artificial and imaginary modifications of a Pascal grammar [49]. We do not claim that these modifications are typical, but we feel that they are representative

Terminal phylum case:
caseExpression : expression
caseListElement : caseElement-list
otherwiseClause : statement

presentation-rules
PA.TextPs:

"Case " caseExpression " of" CR
" " caseListElement CR
" " "Otherwise " otherwiseClause CR
"end"

Explanation: "Case statement"

(a)

Transformation case.version.1.to.2(Opfocus)
Explanation "Transforms version 1 of a case statement to version 2."
{Class 1 transformation.}
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern case
[C1: <expression>]
[C2: <caseElement-list>]
Replacement Case [C1: <expression>] of
[C2: <caseElement-list>]
Otherwise <statement>
end

(b)

Figure 5.4: The phylum declaration of *case* and a transformation.

of modifications to a language in a development process, and that they illustrate our points in terms that are familiar to most people.

5.5.1 Operator Modifications

We will first look at the simple, but nevertheless frequently occurring case where an extra argument is added to an operator. (As mentioned before, this corresponds to adding an extra constituent to a terminal phylum declaration.) Let us assume that we are about to augment the Pascal *case* statement with an otherwise clause. We modify the abstract syntax, and the resulting terminal phylum declaration of *case*, which defines the *case* operator, is shown in figure 5.4(a). Next we apply *InstallNewVersion* on the terminal phylum declaration whereby the version number of the case statement is incremented (say, from 1 to 2), and the system defines the transformation template shown in figure 5.4(b). Because we *append* an argument to the operator, the system-created transformation template is a complete transformation, i.e., it needs no manual modifications to be executable. In other situations, the user must establish the right correspondence between the constituents of the pattern and the constituents of the replacement. Executable transformations are called *class 1* transformations, whereas *class 2* transformations—and higher classes—

are non-executable transformation templates. The user confirms that the transformation is OK by upgrading it from a class 2 transformation to a class 1 transformation. Notice that the presentation of the pattern in figure 5.4(b) is inaccurate. As discussed in section 5.3.1, we do not store the presentation rules of outdated operators, but instead we use a so-called skeletal presentation scheme.

Now let us assume that we start to edit a program in which *case* constructs of version 1 are used. The system can easily discover that outdated constructs are referred to by comparing the version numbers in the AST with those in the operators. Furthermore, it can apply the confirmed class 1 transformations on the outdated constructs. In the current setup, we have an explicit check function *VCheck* (*Version check*) that searches for constructs that belong to old versions, and a function *VUpdate*, which applies the confirmed transformations on outdated constructs. Each transformation updates an operator instance from version v to version $v + 1$. *VUpdate* attempts through repeated application of transformations to obtain as high a version number as possible. The functionality of *VCheck* and *VUpdate* could with relative ease be integrated better into the environment, such that outdated constructs would be identified and brought up to date as soon as they somehow were processed.[3]

In the actual example, the transformation *case.version.1.to.2* appends an unexpanded statement placeholder to each *case* construct in the program. The transformation could of course have been refined such that the otherwise clause was elaborated, for example to an empty statement.[4]

As new aspects have been added to the meta grammar in Muir, grammar modifications like the one described above have often occurred. We have earlier dealt with the problem by writing special Lisp code to "patch" the affected ASTs appropriately. The transformational approach, as exemplified above, seems both to be more convenient and less error prone.

[3]In automatic updating of obsolete documents, the handling of the transformations that update documents again constitute a special case. A transformation is, like the other documents supported by the system, represented as an AST. We should of course be careful not to update the pattern-part of these transformations!

[4]In Muir it is legal for a document to contain unexpanded placeholders. No component of the system will try to eliminate unexpanded placeholders against the will of the user.

5.5.2 Operator Modification and Deletion

We will now look at a more complex grammar modification, in which an operator is eliminated. Let us assume that a grammar for standard Pascal is modified in the following way:

1. The *begin-end* statement is eliminated.

2. Structured statements in which a *begin-end* construct can appear (*if-then*, *if-then-else*, *while*, *for*, *with*, and *case*) are modified such that a list of statements can occur directly, instead of convoluting the list into a *begin-end* construct.

In other words, Pascal is modified towards a Modula-2 style [100]. This is an example of a relatively radical modification of the Pascal grammar, which we successfully have carried out in Muir. We mark the *begin-end* operator as deleted, and we modify the other operators to the new conventions. As a side effect, Muir creates a set of transformation templates. The transformation template for the "deleted" *begin-end* construct will, if applied as it is, tell the user that an application of a deleted operator has been encountered. However, we refine this as well as the other transformations manually. Two of the refined transformations, the transformation for *begin..end* and that for *if-then*, are shown in figure 5.5(a)-(b).

The function *VUpdate* updates the document bottom up. Let us describe how the simple program fragment in figure 5.6 is updated. First the innermost if-then statement is updated by the transformation in figure 5.5(b). The pre-condition of this transformation ensures that the statement part of the if-then statement isn't a statement list (i.e., no begin..end statement has been eliminated in the statement part of the if-then statement.) Although it is not particularly clear from the presentation of the replacement of the transformation in figure 5.5(b), it nests the statement in the then-part into a statement list.[5] Next, the transformation in figure 5.5(a) is applied to eliminate the *begin..end* statement. The transformation unnests the statement list of the *begin..end* construct.

The *begin..end* eliminating transformation leaves the outer *if-then* construct in an inconsistent state. The problem is that the statement

[5] When working with the transformation in the environment, hidden layers, such as the statement-list, are easier to detect, because the identification phylum, the choice phylum, the formalism, and the version number of the current focus can be displayed.

(a) **Transformation** begin..end.version.1.eliminate(Opfocus)
Explanation "The operator begin..end has been marked as deleted."
{Class 1 transformation.}
Pre-condition "T"
Transformation-class Primitive
Focus-modification NoChange
Pattern begin..end
[C1: <statement-list>]
Replacement [C1: <Anything>]

(b) **Transformation** if-then.version.1.to.2(Opfocus)
Explanation "Transforms operator if-then from version 1 to version 2."
{Class 1 transformation.}
Pre-condition "(NOT (EQUAL 'statement-list
(NAMEOFTREE (SUBTREE $Match 2))))"
Transformation-class Primitive
Focus-modification NoChange
Pattern if-then
[C1: <expression>]
[C2: <statement>]
Replacement if [C1: <expression>]
then [C2: <statement>] fi

(c) **Transformation** if-then.version.1.to.2.1(Opfocus)
Explanation "Transforms operator if-then from version 1 to version 2."
{Class 1 transformation.}
Pre-condition "(EQUAL 'statement-list
(NAMEOFTREE (SUBTREE $Match 2)))"
Transformation-class Primitive
Focus-modification NoChange
Pattern if-then
[C1: <expression>]
[C2: <statement>]
Replacement if [C1: <expression>]
then [C2: <statement-list>] fi

Figure 5.5: Begin..end eliminating transformations.

```
if OK
then Begin
        I := 1;
        if Alright
        then P(I)
     End
```

Figure 5.6: Sample program fragment.

part of the outer *if-then* construct is transformed to a list, namely the list consisting of the *assignment* and the inner *if-then* statement. In order to repair the inconsistency, the outer *if-then* statement needs to be transformed too. The transformation in figure 5.5(c), which structurally is trivial, does that. The pre-condition assures that the transformation only is applied if the second constituent is a statement list. I.e., it is only applied if the begin..end eliminating transformation in figure 5.5(a) has been applied on the then-part of the statement.

The example illustrates that in general, more than one transformation is needed to update instances of an operator. In addition to applying *if-then.version.1.to.2* the system also looks for transformations named *if-then.version.1.to.2.i*, for i = 1, 2, 3 etc. The first of these transformations, whose pattern and pre-condition matches the outdated construct will be applied. Notice that because the pattern refers to a specific version of the outdated construct, at most one version i to $i+1$ updating transformation can actually be applied on any given operator instance in a document.

5.5.3 Terminal Phylum to Categorical Phylum

Our next example illustrates an extension of a grammar, by which a terminal phylum is turned into a categorical phylum in the phylum hierarchy. In terms of the grammar modification steps that we described in section 5.1, the grammar modification consists of deletion of the terminal phylum, addition of the categorical phylum, and re-introduction of the terminal phylum as a subphylum of the categorical phylum. This pattern is typical because it reflects the situation where an existing "terminal concept" is extended, and therefore we have chosen to provide special support for it in the environment.

Let us again look at a concrete example from an imaginary extension of Pascal. The existing assignment statement is described by an operator called *assignment*. We extend the assignment concept with a "multi-assignment", i.e., an assignment of the form V1,V2, ...,Vn := E1,E2, ...,En (see, for example, [93].) We choose to keep the normal assignment operator unchanged, and we include a new terminal phylum *multi-assignment* in the grammar. In terms of the phylum hierarchy, the grammar can be modified in two ways, both of which are illustrated in figure 5.7.

Categorical phylum symbols and terminal phylum symbols exist in the same name-space, and therefore the new categorical phylum and the

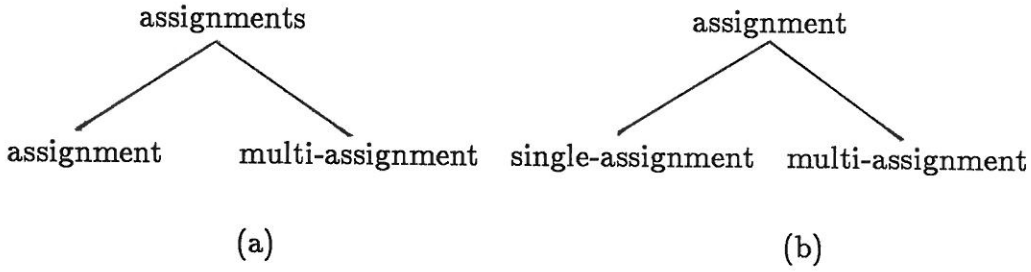


Figure 5.7: Two extensions of the assignment concept.

existing terminal phylum cannot have the same name. Figure 5.7(a) reflects a case where the categorical phylum symbol is a brand new name, *assignments* (plural.) Because the operator-set of all existing phyla are not restricted, this grammar modification does not affect existing ASTs. However, phylum declarations that before the modification referred to *assignment* must now refer to *assignments*. Notice that this in general is a non-local grammar modification, because the existing phylum *assignment* may be referred to from several other phylum declarations. In figure 5.7(b), on the other hand, the categorical phylum inherits the name of the original terminal phylum, and therefore the terminal phylum (operator) needs to be renamed. So if this technique is used, the system must generate a renaming transformation (in this case the name *assignment* must be changed to *single-assignment*.) Such a transformation is trivial, and it can be created automatically by Muir. In return, the grammar modifications are of local nature.

Muir supports an operation named *MakeCategorical* that turns a terminal phylum into a categorical phylum in the same way as in figure 5.7(b). The system prompts the user for another name of the terminal phylum, and in addition it pops up an editor for the new categorical phylum declaration such that the user can elaborate it appropriately. As explained above, it also defines a renaming transformation.

Returning to the example, the user should call *MakeCategorical* on the terminal phylum declaration of *assignment* if he or she wants the modification as illustrated in figure 5.7(b). The categorical phylum declaration, which the system presents for the user as a side effect of this operation, should be refined to have *single-assignment* and *multi-assignment* as subphyla. Furthermore, a new terminal phylum declaration for *multi-*

```
Transformation for-to.version.2.eliminate(Opfocus)
Explanation "*Eliminate an instance of for-to in the document."
{Class 1 transformation.}
Pre-condition "(NOT (MEMBER (IDPHYLUMNAME $Match)
                (SubPhyla (GetDef (CHOICEPHYLUMNAME $Match)) T) ) ) "
Transformation-class Primitive
Focus-modification NoChange
Pattern for [C1: <nameApl>] := [C2: <expression>]
          to [C3: <expression>]
          do [C4: <statement-list>] od
Replacement MATCH
```

Figure 5.8: The transformation template for elimination of *for-to*.

assignment should of course be defined.

5.5.4 Modifications of the Phylum Hierarchy

In the previous sections we have discussed modifications of operators and modifications of the phylum hierarchy at the terminal level. We have also argued that the phylum hierarchy can be restructured safely as long as the operator-sets of the phyla are not restricted. In this section we will discuss how we in Muir deal with changes to the phylum hierarchy that cause restrictions of the operator-set of some phyla.

Muir provides a query operation *PhylumRestriction*, which can be applied on a categorical phylum P, and which returns the names of the operators that have been taken out of the operator-set of P, since the categorical phylum declaration of P last was installed. As an option, the system can create a transformation template for each operator in the "phylum restriction." As a concrete example, let us assume that we eliminate the categorical phylum *for* from the phylum *repetitive* in figure 3.1 on page 31. It means that a repetitive statement now only can be a *repeat* or a *while* statement. If *PhylumRestriction* is applied on *repetitive* after this modification of the phylum hierarchy, it will report that *for-down-to* and *for-to* no longer are valid repetitive statements. Moreover, it can produce a transformation template for each of these. The transformation template for *for-to* is shown in figure 5.8. "MATCH" in the replacement indicates that the replacement is identical to the match, and consequently, if the transformation template is applied as it is shown in

figure 5.8, it will merely print out the text in its explanation.[6] However, the user can refine the replacement to another iterative construct, for example to some while construct that semantically is equivalent to the *for*-statement. The pre-condition ensures that the transformation only is applied if the *for-to*-statement actually should be eliminated in the given context, i.e., if the *for-to*-statement is not a subphylum of its choice phylum. The transformation is, for example, applied if the choice phylum of the *for-to*-statement is *statement*, because *for-to* is not a subphylum of *statement* in the modified grammar. *VUpdate* will apply the transformation in figure 5.8 when a *for-to*-construct (of version 2) is encountered by *VCheck*.

5.6 Summary

We have described a technique to keep documents updated with respect to their grammars in an AST-based environment. The prototype implementation in Muir keeps track of operator-versions, and it supports semi-automatic creation and application of syntactic transformations that bring the documents up to date. If an operator is modified or eliminated, the system creates transformation templates, which, after manual refinement, can update/eliminate instances of the operator from the documents. The system provides special support for turning a terminal phylum into a categorical phylum. If the operator-sets of the categorical phyla are not restricted, the phylum hierarchy can be modified without invalidating existing documents. If the phylum hierarchy is pruned, the system can figure out how the operator-set of a phylum is restricted, and it can create context-sensitive transformation templates, which later must be refined by the user.

The hierarchical grammar model that we use in Muir seems to be well-suited in dealing with the AST consistency problem, because categorical phyla only are referred to indirectly from ASTs. The concrete implementation only causes a minimal storage overhead on the AST representation of grammars. The AST representation of the documents, on the other hand, requires more space to hold the version numbers.

[6] If the first character of the explanation in a transformation is '*', the explanation will be displayed for each match encountered.

Chapter 6

Multi-Formalism Transformations

A set of transformations define a *translation*, i.e., a relation among source documents and target documents [1]. In this chapter we will examine how pattern-based transformations can be used to implement some translations. We primarily aim at a facility that supports *conversions* from one formalism to another. Such a facility is useful in a variety of situations. In a programming environment, for example, it would be valuable to have a tool that helps convert a program from one programming language to another. In a language development environment, the facility could be used to convert documents from a newly developed formalism to "similar" documents in an already existing formalism.

The conceptual distance between the source and the target formalisms is clearly of importance when discussing multi-formalism transformations. We will loosely distinguish the following cases:

1. *Language dialects.* The target language is derived from the source language, and there is a great overlap of *identical* constructs and concepts.

2. *Similar languages.* The source and the target languages are different, but the purpose of the languages are the same, and there is a significant *conceptual* and *structural* overlap between the two languages.

3. *Different languages.* Covers all other cases where the conceptual and structural differences between the two languages are great.

As examples, we consider UCSD Pascal [17] as a dialect of Standard Pascal [49], and we consider Modula-2 [100] and Pascal as similar languages. Pascal and Snobol [38] are in our interpretation examples of

different languages. It should be noticed that it does not influence the translation process if the target language is "an extension" of the source language, whereas the opposite situation may cause problems.

In this chapter we are primarily interested in supporting multi-formalism transformations between languages whose conceptual difference is little. I.e., the tools and techniques that we will describe work best for *language dialects* and *similar languages*. Throughout the chapter our points will be exemplified with transformations from Pascal to Modula-2 and vice versa. We have made hierarchical grammars for both Pascal (as defined by Jensen and Wirth in [49]) and Modula-2 (as defined by Wirth in [100]), and except for the grammatical structures of expressions, the grammars are complete. All the transformations that we are going to discuss in this chapter have been created and tested in the environment.

Pascal and Modula-2 seem to be well-suited to illustrate our points in terms of concepts that are well-known to many people. However, Pascal and Modula-2 are probably too closely related to be representative of "typical translation needs." Therefore it would be desirable to carry out additional experiments on other pairs of languages. This has not been done yet.

We will attack several problems in this chapter. First, we will show how to provide a theoretical basis for multi-formalism transformations in a setting where hierarchical grammars are used. Next, we will demonstrate how to ease the process of creating a set of transformations, and finally, we will discuss how to support and integrate the translation process itself in a structure-oriented editing environment. Before we go into details with these issues, we will more carefully motivate why a multi-formalism transformation facility is desirable in an environment like Muir.

6.1 Motivation

To motivate our approach, let us look at two extremes in the set of possible multi-formalism translation tools:

1. An automatic, complete, and semantic preserving tool.

2. A Semi-automatic tool that cannot carry out the entire translation, and which not necessarily preserves the meaning of the source document.

A transformation tool in the first of these categories tends to be specialized and complicated, and therefore such a tool is rarely constructed and used. If the tool is used on language dialects or similar languages, the majority of the complexity typically stems from a few incompatibilities. It could, for example, be the case that 90% of the complexity handles only 10% of the translations. We find it interesting to investigate much simpler tools in which the difficult cases are left for "manual completion."

As an example of a tool in the second category we will look at translation accomplished by variation in the presentation of the document. Translation tools based on this principle has been proposed by Hansen [39] and by Medina-Mora [59] in his thesis on Aloe. A tool that carries out a translation by applying an alternative presentation scheme is simple to realize in a syntax-directed and AST-based environment, because it only involves creation of a presentation scheme that mimics the concrete syntax of the target language. From a theoretical point of view, we talk about a translation that can be accomplished by a syntax-directed translation schema [1]. The weakness of the approach is that the result is represented as text, and moreover, the translation probably has to be completed via textual manipulations. Essentially, a subset of the abstract syntax of the source language is identified as being equal to the abstract syntax of the target language. The corresponding part of the language can be translated by formulating target-like presentation rules. Despite the weaknesses, the method has, for example, been used successfully to speed up the conversion of a large Pascal program to Ada [89].

We believe that a simple tool in between these two extremes is a viable alternative in an environment like Muir. The tool we have in mind converts a source AST to an AST in the target formalism. Thus, the tool operates solely on the abstract representation of the documents. Consequently, we consider a *translation* to be a relation among ASTs belonging to the source formalism and the target formalism respectively (and *not* a relation among textually represented documents.) Because we base our tool on the pattern-based transformation framework described in section 4.1, we are able to handle some context dependent translations, which cannot be handled in the approach described above. However, it must expected that some aspects cannot be translated (due to the argumentation in 4.1.3), and these aspects are left for manual completion in a syntax-directed editor. Finally, because AST constructs in different formalisms can be distinguished, we are able to identify which constructs

have been translated, and which have not. This is clearly an advantage during the succeeding manually performed translation process.

Even if we had an entirely automatic translation tool, some human "post transformation processing" is often desirable. This is at least the case if the target document is required to be readable and "natural" for humans. Automatic, pattern-based transformations frequently produce badly structured target documents because the transformations fail to identify relations that immediately would have been apparent for a "human transformer." In some sense, the pattern matching capabilities of a programmer are superior to even the most advanced computerized pattern matching we can imagine. This observation indicates that a manual trimming of the resulting document is desirable anyway.

6.2 Overview

Our starting point is two existing formalisms, a *source formalism* and a *target formalism*. We will assume that the environment supports both of the formalisms, i.e., syntactically they are described via hierarchical grammars, and the environment provides Sedit facilities for both of them.

We connect the two grammars by defining a relation among the phyla in the source grammar and in the target grammar. It means that we can consider the two grammars as a single hierarchical grammar during the translation process. The relation among the source and the target phyla is a constraint that limits the "freedom" during the process, and consequently it prevents some kinds of errors. In some situations, however, it is desirable or necessary to violate the rules induced by the relation among the phyla.

The relation between the source phyla and the target phyla also eases the creation of a set of transformations that translates from the source formalism to the target formalism. Muir is able to produce the most trivial transformations automatically, and templates can be created for the vast majority of the remaining transformations. The user is expected to refine the transformation templates, and if necessary, the already created transformations. Depending on the similarities between the two formalisms, it may not be possible to create a set of transformations that defines a 100% translation from the source formalism to the target formalism.

In a metamorphosis-like process, the source document is gradually turned into a target document. First, the transformer applies the set of transformations on the source document, and it typically returns a mixed-formalism document, i.e., a document in which both source constructs and target constructs are present. In order to obtain a pure target document, the user must thereafter complete the translation process via syntax-directed editing in the target language. Besides the already described functionality of Sedit (see section 3.2), we support a search operation that locates source constructs in the mixed-formalism document, and a special edit operation that makes it easy to eliminate source constructs that contain constituents from the target formalism.

6.3 Grammatical Foundation

In this section the grammatical foundation for multi-formalism transformations will be described. We start by defining the so-called source-target relation among the phyla in the source and the target grammars. Next, this relation is used to define a multi-formalism subphylum relation, which is the counterpart to the subphylum relation $\subset^*$ defined in 3.1.2. Finally, the transformation framework is formalized a little bit in order to be able to formulate a sufficient condition that ensures the correctness of the multi-formalism transformations w.r.t. the multi-formalism subphylum relation.

6.3.1 The Source-Target Relation

Given two grammars $G_S = (\mathcal{P}_S, \mathcal{C}_S, \mathcal{T}_S, \mathcal{D}_S)$ and $G_T = (\mathcal{P}_T, \mathcal{C}_T, \mathcal{T}_T, \mathcal{D}_T)$, which are called the *source grammar* and the *target grammar* respectively. When working with either G_S-documents or G_T-documents, the syntactic constraints, as formulated in the respective grammars, limit the set of legal edit operations. We want something similar to be the case for the transformations. I.e., we seek some rules that define what constitute valid transformation steps during the conversion of the source document to a target document. Gate phyla from the source grammar to the target grammar could be used to achieve this, but it seems wrong to incorporate these relationships *statically* into the source grammar. We prefer a somewhat *looser association* between the constraints and the source grammar. Therefore a separate relation among the phyla in the two grammars is

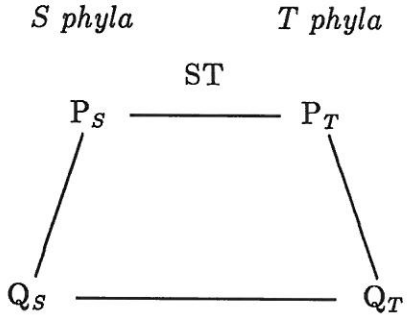


Figure 6.1: Diagram that illustrates a regular phylum P_S.

now introduced, and in the following section this relation will be used to define a multi-formalism subphylum relation.

The *source-target relation* ST is a relation among the phylum symbols in the source grammar and the phylum symbols in the target grammar. I.e., ST $\subseteq (\mathcal{P}_S,\mathcal{P}_T)$. Notationally, $P_S \overset{ST}{\sim} P_T$ means that $(P_S,P_T) \in$ ST. The ST relation is restricted to be a partial function, and if $P_S \overset{ST}{\sim} P_T$ then P_T is functionally referred to as ST(P_S). $P_S \overset{ST}{\sim} P_T$ is intended to mean that a P_S-construct corresponds to a P_T-construct. If both P_S and P_T are terminal phylum symbols, $P_S \overset{ST}{\sim} P_T$ in addition means that a P_S-construct can be transformed to a P_T-construct. In other words, if the terminal phylum symbol P_S is related with the terminal phylum symbol P_T then it must be possible to devise a transformation of the P_S-construct, as defined by the operator of P_S, to the P_T-construct, as defined by its operator.

If a source phylum P fulfills the condition

$$\forall\, Q \in \mathcal{P}_S\text{: } Q \subset^* P \Rightarrow ST(Q) \subset^* ST(P)$$

P is said to be *regular w.r.t. the ST relation.* Figure 6.1 illustrates what it means for a phylum to be regular. In the figure, P_T = ST(P_S) and Q_T = ST(Q_S). The horizontal links reflect the ST relation, i.e., $P_S \overset{ST}{\sim} P_T$ and $Q_S \overset{ST}{\sim} Q_T$. The vertical links are part of the subphylum relations in the two languages. It means that $Q_S \subset^* P_S$ in the S-grammar and that $Q_T \subset^* P_T$ in the T-grammar.

It would be too restrictive to assume that every phylum of the source grammar is regular. It is possible, and even likely that there exist some irregular phyla in the source hierarchy. However, for dialects and similar languages it will typically be the case that a significant subset of the source phyla fulfill the condition. If a given categorical phylum P_S is irregular w.r.t. the ST-relation, and if $P_S \overset{ST}{\sim} P_T$ then there exists a P_S-

construct for which the similar target construct is not a P_T-construct. It might, for example, be the case that the source language and the target language are structured in different ways w.r.t. P_S-constructs and P_T-constructs.

Returning for a moment to a more practical aspect, the ST-relation is defined interactively by selecting pairs of phylum declarations in Sedit instances for the source grammar and the target grammar. While creating the ST-relation the system will report if a source phylum symbol becomes irregular.

6.3.2 The Multi-Formalism Subphylum Relation

The ST relation allows us to define an immediate multi-formalism subphylum relation among the union of phyla from the source grammar and the target grammar. The new relation, which we denote $\sqsubset$, is a counterpart to the immediate subphylum relation $\subset$ (see 3.1.2.) If P and Q are phyla that belong to $\mathcal{P}_S \cup \mathcal{P}_T$ then $P \sqsubset Q$ if and only if (case 3 and 4 are easier to understand by looking at figure 6.2)

1. $P \in \mathcal{P}_S, Q \in \mathcal{P}_S \Rightarrow P \subset Q$
2. $P \in \mathcal{P}_T, Q \in \mathcal{P}_T \Rightarrow P \subset Q$
3. $P \in \mathcal{P}_S, Q \in \mathcal{P}_T \Rightarrow$
   $\exists\, P' \in \mathcal{P}_S\colon P \subset P', P' \overset{ST}{\sim} Q \ \vee$
   $\exists\, Q' \in \mathcal{P}_T\colon P \overset{ST}{\sim} Q' , Q' \subset Q$
4. $P \in \mathcal{P}_T, Q \in \mathcal{P}_S \Rightarrow$
   $\exists\, Q' \in \mathcal{P}_S\colon Q' \subset Q, Q' \overset{ST}{\sim} P \ \vee$
   $\exists\, P' \in \mathcal{P}_T\colon Q \overset{ST}{\sim} P' , P \subset P'$

The multi-formalism subphylum relation $\sqsubset^*$ is defined as the reflexive, transitive closure of $\sqsubset$. Here the reflexive case should be understood such that $A \overset{ST}{\sim} B$ implies that $A \sqsubset^* B$ and $B \sqsubset^* A$. Intuitively, $A_1 \sqsubset^* A_n$ if A_1 equals A_n, $A_1 \overset{ST}{\sim} A_n$, or if one can come from A_1 to A_n by following the edges up in the phylum hierarchies and by following the crossing relations defined by the ST relation (see figure 6.3.)

The starting point of a translation process is a source document, which is represented by a syntactically valid AST entirely in the source formalism. The ultimate goal is to create a target document, which in turn

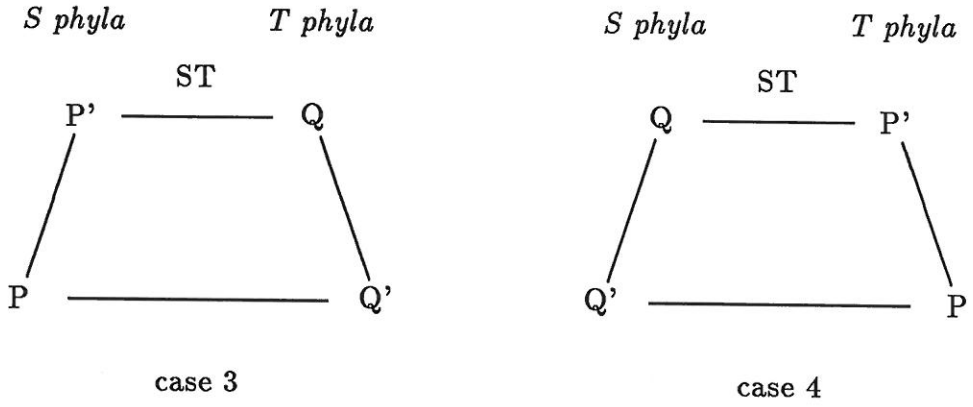


Figure 6.2: Two situations where $P \sqsubset Q$.

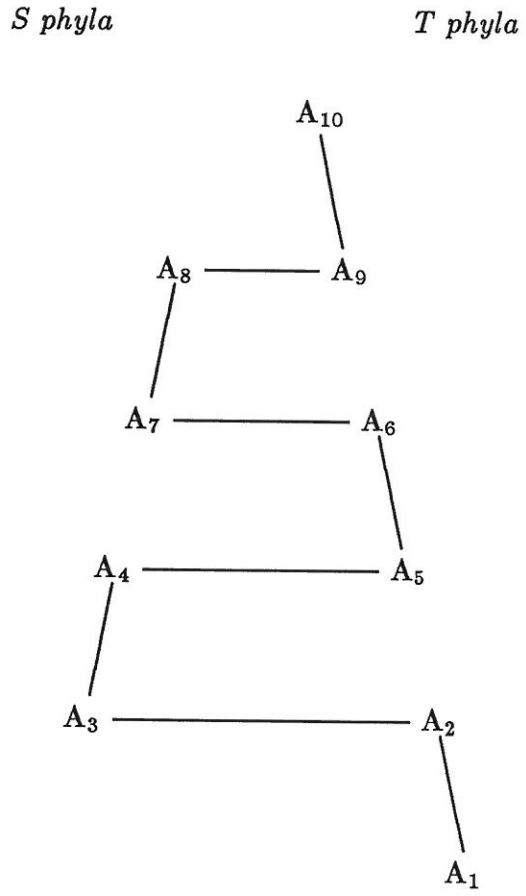


Figure 6.3: $A_1 \sqsubset^* A_n$ for $n \in [1..10]$.

is represented by a syntactically valid AST whose nodes all refer to the target formalism. During the translation process the document is represented as a multi-formalism AST, i.e. an AST in which both source constructs and target constructs are present. An AST A is a *syntactically valid multi-formalism AST* if for every constituent C of A $\varphi(C) \sqsubset^* \omega(C)$, whenever both $\varphi(C)$ and $\omega(C)$ are defined. Notice the similarities between this criterion and the criterion for a syntactically valid AST in the single formalism case (see section 3.1.5.)

Every transformation step (see 4.1.1) performed on a valid multi-formalism AST should produce a new and valid multi-formalism AST. This is at least the ideal, and we want the environment to maintain this constraint, and to warn the user if it is violated. As we will discuss in more details in section 6.3.4, it is sometimes useful to go through invalid multi-formalism ASTs. In that case it is entirely the responsibility of the definer of the transformations to make sure that the succeeding transformation steps remedy the violations.

Let us assume that C_T is a target construct, and that the immediate context of C_T also is a target construct. It means that both $\varphi(C_T)$ and $\omega(C_T)$ are phyla in the target formalism. If the constraint $\varphi(C) \sqsubset^* \omega(C)$ is kept as an invariant for every construct C during the entire translation process then we can be sure that it also holds for C_T. However, we cannot be sure that $\varphi(C_T) \subset^* \omega(C_T)$ in the target grammar. To see why, assume that $A_1 = \varphi(C_T)$ and that $A_{10} = \omega(C_T)$ in figure 6.3. In order to ensure that $\varphi(C_T) \subset^* \omega(C_T)$ in figure 6.3 we must require that the phyla A_4 and A_8 are regular (see section 6.3.1.) Regularity of A_4 and A_8 namely implies that $A_6 \subset^* A_9$ and $A_2 \subset^* A_5$. So unless all the involved source phyla are regular, we must check explicitly if the resulting target document is syntactically correct w.r.t. the target grammar.

In the next section we will look at some practical cases of irregular phyla w.r.t. the Pascal–Modula-2 relation and the Modula-2–Pascal relation, and we will discuss the consequences for the multi-formalism subphylum relations.

6.3.3 Pascal Modula-2 Cases

Difficulties in relating source phyla to target phyla, and irregular source phyla reflect quite well how close the source and the target formalisms are related. To add more concreteness to the discussion, we will in this

section describe some situations encountered in relating Pascal phyla to Modula-2 phyla and vice versa.

6.3.3.1 Source Concept does not Exist in the Target Language

If a source phylum represents a concept for which no counterparts exist in the target language then the source-target relation is undefined on the source phylum. As an example from the Modula-2 to Pascal case, it is not possible to relate the *ModuleExport* and *ModuleImport* phyla to any Pascal construct, because there is no module concept, and consequently no notion of export and import in Pascal.

In other situations, a source concept does not have a direct counterpart in the target language, but nevertheless, the target language supports the source concept in another way. The *by-clause* of the Modula-2 *ForStatement*[1] provides and example of this (again from Modula-2 to Pascal.) The by-clause cannot be related to a single, similar Pascal construct. In two special cases ("BY 1" and "BY -1"), however, the by-clause can be dealt with by its immediate context in the target language (via the *for-to* statement and the *for-downto* statement respectively.) In the opposite transformations, from Pascal to Modula-2, the goto statement[2] causes similar problems. Also the *begin-end* construct[3] in Pascal does not have a separate counterpart in Modula-2. In this case, the immediate target context of the statement-list takes care of the "bracketing."

Let us finally consider a similar example, in which the problems cause a source phylum to be irregular. In Modula-2, a *CaseLabel* in a case statement can be a constant expression, or it can be a *range* whose limits are defined by two constant expressions. In Pascal, a case label must be a constant. The involved phyla are illustrated in figure 6.4. Dotted lines represent the source-target relation, and ordinary lines represent the immediate subphylum relation. Modula-2 is more general than Pas-

[1] In Modula-2, it is possible in the *by-clause* to specify how much to increment the control variable in each iteration of a *ForStatement*. Pascal, on the other hand, has two different *for* statements: A *for-to* and a *for-downto* with fixed increment/decrement.

[2] There is no explicit and general goto statement in Modula-2, as there is in Pascal. However, Modula-2 provides explicit mechanisms to jump out of a *loop-statement* and to jump out of a *procedure*.

[3] In Pascal, *begin-end* constructs are used in structured statements, e.g., in *if-then* statements, to assemble several statements into a single compound statement. This is not necessary in Modula-2, because the structured statements themselves are allowed to contain a list of statements.

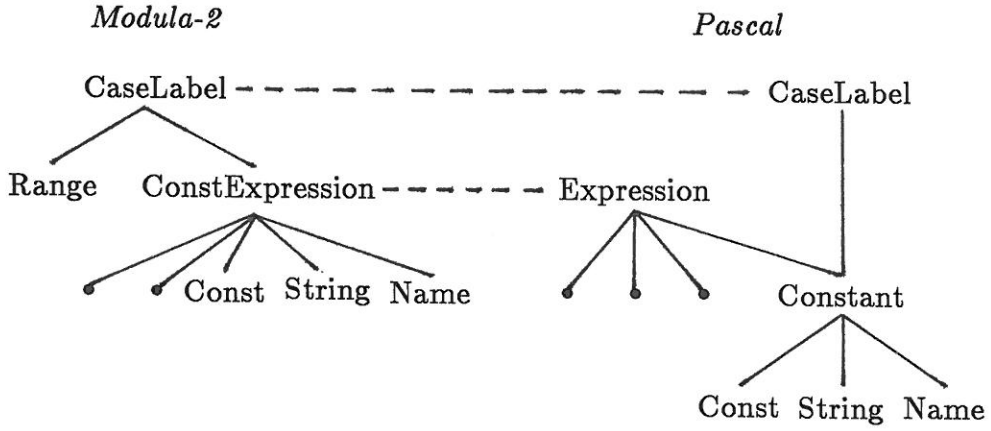


Figure 6.4: Phylum hierarchies for case labels.

cal w.r.t. case labels, and we have no transformational solution to this problem. Here we are only interested in how this affects the source-target relation. Like import and export clauses, a *range* does not have a counterpart in Pascal, and the programming of a case statement, in which the more general case labels are used, must be re-thought and re-arranged in the Pascal version. A constant expression does have a counterpart in Pascal, represented by the more general phylum *Expression*. But the phylum *Expression* is too general to be a subphylum of *CaseLabel* in Pascal, so the Modula-2 phylum *CaseLabel* becomes irregular (see figure 6.4.)

6.3.3.2 Different Structuring of the same Concept

If identical concepts are structured differently in the source and the target grammar, it typically implies that one or more phyla in the source grammar become irregular. In the Pascal to Modula-2 case, the phyla *constant-declarations*, *type-declarations*, and *variable-declarations* become irregular because each of these can be empty in the source language (Pascal), but this is not the case in the target language. Figure 6.5 illustrates that for constant-declarations. In Modula-2, all the declarations (constants, types, variables, etc.) are organized in a single list, and therefore empty declarations are not necessary. The *empty* phyla in the Modula-2 grammar and in the Pascal grammar are "generic" and shared, and they represent the empty alternative in a variety of situations. The empty phyla of the two grammars are therefore related via the source-target

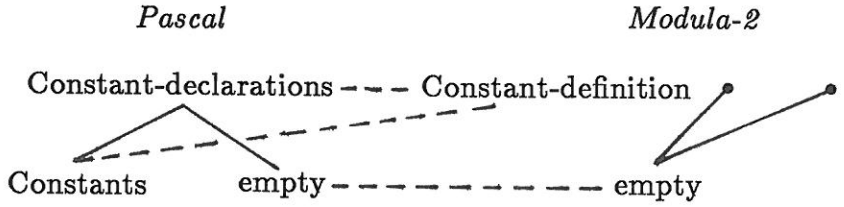


Figure 6.5: Phylum hierarchies for constant declarations.

relation. In the situation shown in figure 6.5 it implies that the Modula-2 phylum *empty* is a multi-formalism subphylum of a Pascal *constant-declarations*. I.e., an empty Pascal declaration can legally be transformed to an empty Modula-2 declaration. So the relation between the *empty* phyla makes the multi-formalism subphylum relation too broad, and it makes the Pascal phylum *Constant-declarations* irregular.

A slightly more complicated example of the same phenomena occurs in the translation of records in the two languages. Especially w.r.t. variant records, the two languages adhere to a different structuring.

6.3.3.3 Inadequate Phylum Hierarchies

If the target grammar fails to define an adequate phylum hierarchy, some source constructs must be related to a more general target phylum than necessary. This makes the multi-formalism subphylum relation less useful than if a better categorization of the target phylum hierarchy existed. As an example, the Modula-2 *ForStatement* covers both the Pascal *for-to* and *for-downto* statements. I.e., *ForStatement* (in Modula-2) cannot be related to only a single one of these. The natural thing to do in the Modula-2 to Pascal relation is to relate the Modula-2 phylum to a categorical phylum in Pascal, of which both *for-to* and *for-downto* (and nothing else) are subphyla. If such a categorical phylum did not exist in Pascal, the Modula-2 phylum had to be related to a more general Pascal phylum, for example, *structured-statement* or *statement* (see figure 3.1 on page 31.)

6.3.4 Transformation Correctness

In this section we will formulate a condition on transformations that guarantees that the transformation, when applied, produces a syntactically valid multi-formalism document. In order to ease the formulation of this

condition, we will first formalize the transformation framework a little bit.

A transformation (see section 4.1) will be considered as a triple τ = (P,R,PR) where P is the pattern, R is the replacement, and PR is the relation among the constituents of P and R (see 4.1.1.) For simplicity it will be assumed that PR is a one-to-one function[4], and if a pattern constituent C_P is related to a replacement constituent C_R we write C_R = PR(C_P). If A is an AST then τ(A) denotes the application of the transformation τ on A.

Given a transformation τ and a syntactically valid multi-formalism AST A, it is *not* possible to formulate a necessary condition such that τ(A) is syntactically valid. In order to do that we need to know the actual matches (as opposed to the pattern), or more precisely, we need to know the identification phyla of those constituents of the matches that correspond to unexpanded nodes in the pattern. However, a sufficient condition can be formulated, which especially is useful for multi-formalism transformations:

Theorem. *Correctness of a transformation*

Given a syntactically valid multi-formalism AST A and a transformation τ = (P,R,PR). Let C_P^1, ..., C_P^n denote the AST-constituents of P that are related to AST-constituents of the replacement R. If

1. $\varphi(R) \sqsubset^* \varphi(P)$ and
2. $\omega(C_P^i) \sqsubset^* \omega(PR(C_P^i))$ for $i \in [1..n]$

then τ(A) is a syntactically valid multi-formalism AST.

The proof of the theorem is given in appendix A. If a transformation τ satisfies the correctness theorem then τ can be applied on a syntactically correct document without violating the syntactic constraints. If an "incorrect" transformation is applied on a syntactically correct document the result may or may not be syntactically correct. It should be noticed that the theorem guarantees multi-formalism syntactic correctness if the pre-conditions are fulfilled, but as already discussed in section 6.3.2 it does not necessarily mean that the target document is syntactically valid w.r.t. the subphylum relation defined by the target grammar. In order for

[4]The results in this thesis are valid for PR being a one-to-many *relation*.

that to be true, "critical" phyla in the source grammar must be regular.

A translation process is carried out as a sequence of transformation steps. The ideal situation is that each transformation step results in a syntactically correct document. However, as argued in section 6.3.2, this is not always the case. So we must be able to deal with situations where certain transformation steps result in documents that are syntactically invalid. We have in several situations found it useful to apply a sequence of transformations, say τ_1, τ_2, and τ_3 where τ_1 introduces syntactically invalid constructs, and where τ_2 and τ_3 "makes it good" again. Seen in isolation, τ_1, τ_2, and τ_3 are "bad", but if it can be proved that they as a whole preserve the syntactic correctness of documents it is OK to apply them in the given sequence. We can specify that a transformation should be applied in unchecked mode. If such a transformation introduces syntactic violations, it is the responsibility of the user to convince himself or herself that the syntactic violations only are temporary. In section 6.4.3 we will encounter a practical example of that.

6.4 The Transformations

It can be a tedious task to create a set of transformations from one language to another. Especially if the languages are similar to each other it is a non-challenging routine task to create the majority of the transformations. Due to our basic philosophy ("simple routine tasks should be done by the system, and more intellectually demanding tasks should be under control of the user", see section 1) the system should therefore create the most trivial transformations automatically, and it should assist the user in creating the remaining ones. In section 6.4.1 we will demonstrate how the information contained in the source-target relation can be used for that purpose. We will illustrate our approach by showing some of the transformations from Pascal to Modula-2 and vice versa. The system-created transformations are all context-free, but often more context dependent transformations are necessary. In section 6.4.3 we will discuss and exemplify context dependent elements in the transformations. Even with some context-dependent transformations, there are some transformation tasks that are hard to carry out via the pattern-replacement based technique. In section 6.4.4 we give an example of that, and we describe how such a transformation can be programmed as an Interlisp function, in which

the most significant functionality still comes from pattern-based search operations.

6.4.1 Semi-automatic Creation of Transformations

By relating a terminal phylum symbol P_S from the source grammar to a terminal phylum symbol P_T in the target grammar we have stated that a P_S-construct can be transformed to a P_T-construct (see 6.3.1.) Thus, the system can create a transformation template in which the pattern is the construct defined by P_S, and the replacement is the construct defined by P_T. In order to refine such a template to a complete and an operational transformation, the pattern, the replacement, and possibly the pre-condition must be elaborated appropriately. In general, this process is hard to automate. But in the cases where neither the pattern nor the replacement need real refinement, and where a trivially true pre-condition is sufficient, the system can attempt to complete the transformation-creation process by relating constituents of the pattern to constituents of the replacement. Let us now describe how this can be done.

Let P_S: P_1^S ... P_n^S and P_T: P_1^T ... P_m^T be terminal phylum declarations in the source and in the target grammars respectively, and like above, we assume that $P_S \overset{ST}{\sim} P_T$. The system can then generate a transformation in which the pattern is a P_S-construct, and the replacement is a P_T-construct. The immediate constituents of both the pattern and the replacement are unexpanded nodes. The unexpanded pattern constituent that corresponds to P_i^S will be related to the unexpanded target constituent corresponding to P_j^T if[5]

$$\exists! j \in [1..m] : P_i^S \sqsubset^* P_j^T.$$

In addition we require that no other pattern constituent has been related to P_j^T. If all the pattern constituents in this way can be related to a constituent of the replacement, the transformation is complete. Notice that the technique used to automatically relate pattern constituents to replacement constituents is of *heuristic* nature. I.e., it cannot be guaranteed to do what an intelligent agent (a programmer) would have done. However, in the vast majority of the cases we have been through, the

[5]Actually, in the current version of the system a slightly less general relation than $\sqsubset^*$ is used.

automatically completed transformation templates turned out to be adequate.

As already described in section 5.5.1, executable transformations are called *class 1* transformations, and non-executable transformation templates have assigned higher class numbers. If a terminal phylum symbol P_S is related to a terminal phylum symbol P_T, the transformation that contains a P_S-construct as the pattern and a P_T-construct as the replacement is called a *class 2* transformation. If, in addition, all the relevant pattern constituents can be related to constituents of the replacement then the transformation is upgraded to a *class 1* transformation. The relation among pattern constituents and replacement constituents can be done manually, or it can be done entirely or in part by the system through the heuristic technique described above. If instead the terminal phylum symbol P_S is related to a categorical phylum symbol in the target grammar then a *class 3* transformation template is created, in which the replacement part is unspecified.

In Muir we can activate an Interlisp function *ConstructTransformations* on the source and the target grammars together with their source-target relation and have it deliver a list of class 1, class 2, and class 3 transformations. The user should manually look through the list of transformations in order to make sure that the class 1 transformations actually are OK. Furthermore, the class 2 and class 3 operations can be upgraded to class 1 transformations once their replacement, pattern, and pre-condition have been elaborated. 100% manually created transformations can also be added to the list of transformations. The organization of the list of transformations depends on the transformation strategy, and this is discussed further in section 6.5.

6.4.2 Examples of Transformations

The purpose of this section is to clarify the previous section through concrete examples of transformations from Pascal to Modula-2 and vice versa. The table in figure 6.6 shows how many transformations and transformation templates there have been created to carry out the transformation tasks. The upper section of the table gives information about the automatically produced transformations before any manual refinement. The lower section shows the actual number of executable transformations, automatically produced and manually refined all together. It should be

		PA→MO	MO→PA
Automatically produced	class 1	40	32
	class 2	22	18
	class 3	8	32
Refined	class 1	63	60

Figure 6.6: The number of created transformations.

noticed that the class 1 transformations in the table—both the automatically produced and the refined transformations in both directions—include approximately 20 transformations for translation of linear list structures. Recall that we represent linear lists as binary trees of list-heads and list-tails (see section 3.2.1.) To translate such list structures we need a specific transformation for each list-element type.[6] These transformations are, of course, trivial, and they can be generated automatically by the system.

In the rest of this section we will give examples of both class 1, class 2, and class 3 transformations. Let us start by considering a couple of class 3 transformations. Figure 6.7 shows two (automatically produced) transformation templates that belong to class 3, one from Pascal to Modula-2, and one in the other direction. In general, the transformations are named $S.C_S \rightarrow T.C_T$, where S and T are abbreviations for the source formalism and the target formalism names respectively. C_S is a name of the source construct, which is translated to the target construct named C_T. In figure 6.7 the target constructs are indeterminable, and they are therefore denoted by question marks in the transformation names. As described above, class 3 transformation templates originate from terminal phyla in the source grammar that are related to categorical phyla in the target grammar, and class 3 transformations are not executable without further elaboration. The terminal phylum *begin..end* in Pascal is related to the Modula-2 phylum *Statement* because there is no separate begin-end construct in Modula-2. The user could refine the transformation template in figure 6.7(a), but in this case it turned out to be more attractive to

[6]The chosen list representation in fact requires two transformations for each list type: One that translates the list structure, and another that translates the empty list indication, which is located at the end of the list. In our list-representation, each kind of list has its own nullary "empty list" operator. It means that we distinguish between, for example, empty statement lists and empty expression lists. None of these entirely trivial transformations are included in the table shown in figure 6.6.

(a) **Transformation** PA.begin..end→MO.?(Opfocus)
Explanation "Explanation"
{"Class 3 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern Begin
[C1: <statement-list>]
End
Replacement MATCH

(b) **Transformation** MO.ForStatement→PA.?(Opfocus)
Explanation "Explanation"
{"Class 3 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern FOR [C1: <nameApl>] := [C2: <expression>]
TO [C3: <expression>] [C4: <ByClause>]
DO [C5: <statement-list>] END
Replacement MATCH

Figure 6.7: Class 3 transformations.

(a) **Transformation** MO.IfStatement→PA.if-then-else(Opfocus)
Explanation "Explanation"
{"Class 2 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern IF [C1: <expression>] THEN [C2: <statement-list>]
[C3: <elsif-list>]
[C4: <ElseStatement>] END
Replacement if [C1: <expression>]
then <statement>
else <statement>

(b) **Transformation** PA.for-to→MO.ForStatement(Opfocus)
Explanation "Explanation"
{"Class 2 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern for [C1: <nameApl>] := [C2: <expression>] to [C3: <expression>]
do [C4: <statement>]
Replacement FOR [C1: <nameApl>] := <expression> TO <expression>
<ByClause> DO <statement-list> END

Figure 6.8: Class 2 transformations.

deal with the transformation of begin-end constructs from their context. The transformation in figure 6.7(b) is a class 3 transformation because the terminal phylum *ForStatement* in Modula-2 is related to the categorical phylum *for*-statement in the Pascal grammar (of which *for-to* and *for-downto* are subphyla.) In this case, we actually refined the transformation to three separate class 1 transformations that cover typical "and easy cases" (namely "BY 1", "BY -1", and no BY-clause at all.) These transformations are not shown here.

The class 2 transformation templates in figure 6.8 are also generated automatically by Muir, but they could not be upgraded automatically to class 1 transformations. In figure 6.8(a) the pattern constituents <statement-list>, <elsif-list>, and <ElseStatement> cannot immediately be related to any of the constituents of the replacement via the heuristic technique described in 6.4.1. The Modula-2 if-statement is more general than the counterpart in Pascal, and this complicates the Modula-2 to Pascal conversion. We will take a more detailed look at these prob-

(a) **Transformation** PA.for-to-begin→MO.ForStatement(Opfocus)
Explanation "Explanation"
{"Class 1 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern for [C1: <nameApl>] := [C2: <expression>]
to [C3: <expression>]
do Begin
[C4: <statement-list>]
End
Replacement FOR [C1: <nameApl>] := [C2: <expression>]
TO [C3: <expression>] BY 1
DO [C4: <statement-list>] END

(b) **Transformation** MO.ArrayType→PA.array-type(Opfocus)
Explanation "Explanation"
{"Class 1 Transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern ARRAY [C1: <SimpleType-list>] OF [C2: <type>]
Replacement array [[C1: <index-type-list>]] of [C2: <type>]

Figure 6.9: Class 1 transformations.

lems in section 6.4.3. The class 2 transformation template for the Pascal for-to statement in figure 6.8(b) is easier to upgrade to class 1. Notice that Muir does not relate any of the two expressions in the pattern to the expressions in the replacement, because there exist more than one constituent in the replacement to which the source expressions could be related.

Figure 6.9(a) shows the manually produced refinement of the class 2 transformation template from figure 6.8(b). By comparing figure 6.8(b) with figure 6.9(a) it can be seen that the statement in the pattern has been refined to a begin-end block, the by-clause of the replacement has been elaborated, and the relation among the pattern constituents and the replacement constituents has been completed. Also the name of the transformation has been changed to reflect the modifications. All of the modifications have been done manually. After these refinements the transformation is executable, and it can be upgraded to class 1. The transformation in figure 6.9(b) is an example of a class 1 transformation

that has been created without manual interference. (The outer level of the double brackets in the replacement of figure 6.9(b) stems from the Pascal syntax of arrays.)

6.4.3 Context Dependencies

In its most simple form, a transformation specifies that (1) a primitive (two-layer) replacement construct must substitute all occurrences of a primitive pattern-construct, and (2) that certain sub-structures of the matches should be "re-used" as substructures of the replacement. Apart from permutation, duplication, and deletion of constituents, such transformations preserve the abstract structure of the documents on which they are applied. Frequently, however, more intricate transformations are required, and the greater the structural and conceptual difference between the source and the target language, the more we need context dependent transformations. We will in this section discuss and exemplify how, and to which degree context dependencies are supported in the Muir transformation framework.

Our points are illustrated quite well by the translation of Pascal if-then statements to similar statements in Modula-2 (and vice versa.) We will therefore start by summarizing the differences between the two languages w.r.t. these constructs. Modula-2 differs from Pascal in two significant ways w.r.t. if-then-else statements. First, the Modula-2 if-statement is designed to handle an if-then-else chain rather than only one condition and two alternative statements. In Pascal, if-then-else chains must be modelled by nested if-then-else statements. Secondly, Modula-2 allows a sequence of statements to occur in the then-parts and in the else-part, whereas Pascal only allows a single statement at these places. Figure 6.10 shows similar if-then-else chains in the two languages.[7] The Pascal construct consists of three nested if-then-else statements, whereas the Modula-2 construct is a single statement with two elsif-clauses and an else-clause. If we assume that every "then-clause" and every "else-clause" in the Pascal construct are enclosed in begin-end constructs, it is trivial to create a transformation that defines a correct translation

[7]The use of if-then-else chains in Pascal is often signalled by a "flat", un-indented pretty-printing instead of the nested pretty printing used in figure 6.10. However, in a structure-oriented environment the pretty-printing is not controlled by the user on a case-by-case basis, and therefore a general set of rules determines the screen presentation of the if-then-else chains.

```
(a) if <expression1>
    then begin
            <statement-list1>
         end
    else if <expression2>
         then begin
                 <statement-list2>
              end
         else if <expression3>
              then begin
                      <statement-list3>
                   end
              else begin
                      <statement-list4>
                   end
```

```
(b) IF <expression1> THEN <statement-list1>
    ELSIF <expression2> THEN <statement-list2>
    ELSIF <expression3> THEN <statement-list3>
    ELSE <statement-list4> END
```

Figure 6.10: Equivalent Pascal (a) and Modula-2 (b) if-then-else chains.

```
Transformation PA.if-THEN-ELSE-begin→MO.IfStatement(Opfocus)
Explanation "Explanation"
{"Class 1 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern if [C1: <expression>]
        then Begin
                [C2: <statement-list>]
             End
        else Begin
                [C3: <statement-list>]
             End
Replacement IF [C1: <expression>] THEN [C2: <statement-list>]
            ELSE [C3: <statement-list>]
            END
```

Figure 6.11: A context free transformation of an if-then-else structure.

of Pascal if-then-else chains to nested if-statements in Modula-2. Figure 6.11 shows such a transformation. The assumption about begin-end enclosement can be fulfilled through a preceding normalization of the source document (see section 6.5.) However, the transformation in figure 6.11 fails to take advantage of the special facilities for the definition of if-then-else chains in Modula-2. Application of the transformations creates a nested if-then-else structure quite similar to the structure of the if-then-else chain in Pascal. In order to produce a better translation more context dependency must be built into the transformation.

The easiest way to add context dependency is to broaden the pattern. In the concrete situation, a sequence of transformations like the one in figure 6.12 could be defined to capture all the relevant source constructs and their desired replacements. The transformation shown in figure 6.12 only handles if-then-else chains of length two, and similar transformations must be defined to handle chains of length 3, 4, 5, etc. This solution clearly suffers from lack of generality because an infinite number of transformations are needed to cover all possible cases.

Figure 6.13 shows two alternative transformations that do a somewhat better, although not a perfect job. The transformation in figure 6.13(a) converts an if-then-else statement, which is in the context of an else-part of another if-then-else statement, to a Modula-2 elsif-list. The context requirement is formulated in the pre-condition of the transformation. In

Transformation PA.if-then-else-chain2→MO.IfStatement(Opfocus)
Explanation "Explanation"
{"Class 1 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern if [C1: <expression>]
then Begin
[S1: <statement-list>]
End
else Begin
if [C2: <expression>]
then Begin
[S2: <statement-list>]
End
else Begin
[S3: <statement-list>]
End
End
Replacement IF [C1: <expression>] THEN [S1: <statement-list>]
ELSIF [C2: <expression>] THEN [S2: <statement-list>]
ELSE [S3: <statement-list>]
END

Figure 6.12: A transformation of an if-then-else chain of "length" two.

(a) **Transformation** PA.if-then-else-chain-1→MO(Opfocus)
Explanation "Explanation"
{Translation of the inner if-then-else statement. The whole pattern is a Pascal statement list, of which the head is an if-then-else statement. The replacement is a Modula-2 elsif-list. The pre-condition says that this statement-list must be in an environment of the else-part (3) of an if-then-else construct. Take at most 2 levels above $Match into consideration. Or (because the transformation is done outside-in), the statement-list must be in the context of an already produced Modula-2 elsif-list. }
Pre-condition "(OR (InEnvironment? $Match 'PA.if-then-else 2 3)
(InEnvironment? $Match 'MO.elsif-list 2 2))"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern if [E: <expression>]
then Begin
[TS: <statement-list>]
End
else Begin
[ES: <statement-list>]
End;
<statement-list>
Replacement ELSIF [E: <expression>] THEN [TS: <statement-list>]
[ES: <elsif-list>]

(b) **Transformation** PA.if-then-else-chain-2→MO(Opfocus)
Explanation "Explanation"
{Translation of the outer if-then-else statements.}
Pre-condition "T"
Transformation-class Primitive-by-reference
Focus-modification NoChange
Pattern if [E: <expression>]
then Begin
[S1: <statement-list>]
End
else Begin
[EI: <elsif>
<elsif-list>]
End
Replacement IF [E: <expression>] THEN [S1: <statement-list>]
[EI: <elsif-list>]
<ElseStatement>
END

Figure 6.13: Transformation of inner and outer if-then-else constructs.

terms of the example in figure 6.10, the "if <expression2>" statement will first be transformed, and next the "if <expression3>" statement will be treated. The transformation in figure 6.13(b) takes care of the outer level of the if-then-else chain. Notice that the pattern is a multi-formalism construct, and furthermore that the pattern is not a valid multi-formalism AST. The Modula-2 elsif-construct in the pattern assures that the transformation in figure 6.13(a) has been applied on the else-part of the if-then-else construct before the application of this transformation.

The only part of the if-then-else chain that is not dealt with properly by the two transformations in figure 6.13 is the trailing else statements, <statement-list4> in figure 6.10. In Pascal, the trailing else-statement is the deepest construct in the structure, whereas in the equivalent Modula-2 statement, the trailing else statement is part of the outer level of the if-statement. In order to extract the trailing else statement it must be possible to formulate a pattern that matches the whole if-then-else chain and that contains the trailing else statement as an explicit constituent. This is not possible without an extension of the transformation framework in Muir. What is needed is a *structural wild card* that matches the maximum number of a given kind of constructs in between two other constructs. We have not designed such a wild card for Muir, so in the concrete situation the translation of the trailing else-statement will be placed in the tail of the Modula-2 elsif-list. It must then manually be moved to the outer level of the resulting Modula-2 if-statement.

The transformation in figure 6.13(a) illustrates how the function *In-Environment?* makes it possible for super-constructs of the match candidates to affect the transformation. *InEnvironment?* returns whether a given construct (for example the match) is in the context of a given type of construct (identified by an identification phylum symbol.) The condition can be narrowed by specifying how far up in the AST to look, and from which constituent the identification phylum must be approached. For example the call

(InEnvironment? '$Match 'PA.if-then-else 2 'else-part)

returns whether the match is two levels (or less) below the else-part[8] of a Pascal if-then-else statement. If it is, a reference to the if-then-else

[8]In the actual implementation, the fourth argument to *InEnvironment?* should be a number—and not an atom such as *else-part*. This number is a constituent number in the construct determined by the second argument. In the concrete example, the *else-part* is the third constituent of the Pascal *if-then-else* construct.

construct is returned. Symmetrically, sub-constructs of the match candidates may affect the transformation through activation of the pattern matcher in the pre-condition.

One of the main observations in this section has been that it is difficult to define general enough patterns in the Muir transformation framework. This is in accordance with the remarks about the inherent limitations of the pattern-based transformation framework, as discussed in section 4.1.3. Also the replacement mechanism seems to be too weak. This can be illustrated if we consider the transformations that implement the translation of Modula-2 if-statements to if-then-else chains in Pascal (i.e, the opposite of the transformation that we studied thoroughly above.) They are shown in figure 6.14, and they work in roughly the same way as the two opposite transformations shown in figure 6.13. The problem here is how to embed the trailing else statement, i.e., the statement named "E" in the pattern of the transformation in figure 6.14(a), deeply into the target construct. The replacement in this transformation does not contain the proper place for the trailing else statement, and it would be tricky to elaborate the transformations in such a way that the desired translation would be defined.

6.4.4 Procedural Transformations

As already discussed in section 4.1.3 it is possible to extend the pattern-based transformation framework such that more and more cases can be handled in a satisfactory way. As another direction, we can chose to escape to procedural solutions when pattern-based transformations are too awkward or perhaps even impossible for a given task. It is sometimes tempting to store a given construct C, do something else, and then insert the translation of the construct C at another place. In general, this is not possible in the pattern-based paradigm, but it is clearly easy to do in a solution where the transformation is programmed in a "general programming language." In this section we will show how we have dealt with a difficult transformation problem by writing an Interlisp transformation function.

We choose to illustrate procedural transformations with the translation of Modula-2 declarations to similar Pascal declarations. In Modula-2, different kinds of declarations can be mixed freely in a declaration list. In Pascal, on the other hand, declarations are grouped into label declara-

(a) **Transformation** MO.IfStatementELSE→PA.if-then-else-chain(Opfocus)
Explanation "*You must move the else clause of an IfStatement to the innermost Pascal if-then-else statement."
{"Class 1 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern IF [C: <expression>] THEN [T: <statement-list>]
[EI: <elsif>
<elsif-list>]
ELSE [E: <statement-list>]
END
Replacement if [C: <expression>]
then Begin
[T: <statement-list>]
End
else [EI: <statement>]

(b) **Transformation** MO.elsif-list→PA.if-then-else(Opfocus)
Explanation "Explanation"
{"Class 1 transformation."}
Pre-condition "T"
Transformation-class General-by-reference
Focus-modification NoChange
Pattern ELSIF [E: <expression>] THEN [SL: <statement-list>]
[RL: <elsif-list>]
Replacement if [E: <expression>]
then Begin
[SL: <statement-list>]
End
else [RL: <statement>]

Figure 6.14: Modula-2 to Pascal transformations of if-statements.

```
(a) VAR i, j: INTEGER;
    CONST Lower = 7;
          Upper = 14;
    TYPE T = [Lower .. Upper]
    VAR TV: T;
    CONST C = 55;

(b) Const Lower = 7;
          Upper = 14;
          C = 55
    Type T = Lower .. Upper
    Var i, j: Integer;
        TV: T
```

Figure 6.15: Equivalent Modula-2 (a) and a Pascal (b) declarations.

tions, constant declarations, type declarations, variable declarations, and declaration of procedures and functions. Figure 6.15 shows an example of a source fragment and the desired target fragment. In essence, the transformation must collect and merge constants, types, variables, and procedures. Furthermore, knowledge about scope rules is necessary to avoid collection of local declarations from procedures and local modules. Figure 6.16 shows a simplified Interlisp function in which the procedural steps are programmed. First, all the Modula-2 blocks are located, and each of them are then transformed (in the outer for-statement.) During the transformation of a single block the various kinds of declarations are collected. (CALL 'ConstsInBlock Block) activates a pre-defined structure-oriented search operation *ConstsInBlock* (not shown here), which returns a list of constant declarations. A pre-condition in the search operation ensures that there does not exist any blocks in between the Block passed as an argument to *ConstsInBlock* and the matches, i.e., the matches do not belong to local blocks. The other kinds of declarations are collected in similar ways. The lists of declarations are then joined into groups of declarations that can be inserted into a template of a Pascal block. Notice that the appropriate places to insert the declarations in the Pascal block also are located via pattern matching. The actual translation of the various Modula-2 declarations is done by subsequent transformations. The aggregated Pascal block substitutes the Modula-2 block, and finally the procedure is via a recursive call repeated on local blocks, for example, belonging to local procedures.

```
(LAMBDA (Focus)
   (* * Transforms Modula-2 declarations to Pascal declarations)

   (* * First locate the Modula-2 blocks: )
   (SETQ Blocks (CALL 'BlockSearch Focus))

   (for Block in Blocks do
      (PROG (Consts PaBlock ...)

         (* * Collect all constant declarations in the block:)
         (SETQ Consts (CALL 'ConstsInBlock Block))

         (* * Collect types, variables, procedures, modules, and the statements
         in the same way it was done for constants. )

         (* * Join all the constant declarations just collected:)
         (SETQ JoinedConsts empty-list)
         (for Dcl in Consts do (ADDSUBLIST JoinedConsts Dcl 'After))

         (* * Handle the other kinds of declarations similarly.)

         (* * Create a template of a Pascal block:)
         (SETQ PaBlock (Template 'ABlock 'PA))

         (* * Insert the Modula-2 declarations into the Pascal block:)
         (ASTSUBSTITUTE (PatternMatch PaBlock 'constantDecl-list)
            (if JoinedConsts then JoinedConsts else empty))

         (* * Insert the other declarations too, and insert the statements.)

         (* * Replace the Modula-block with the Pascal block:)
         (ASTSUBSTITUTE Block PaBlock)

         (* * Apply the transformation recursively on the the local blocks:)
         (MO.block.to.PA.block PaBlock)))

   (* * Return the list of Pascal blocks ))
```

Figure 6.16: Sketch of Interlisp transformation function.

6.5 The Translation Process

The purpose of the translation process is to create a target document similar to the source document, which is supplied as input to the *transformer*. The actual translation process consists of two phases: (1) application of the executable (class 1) transformations, and (2) manual completion of the translation task. In both phases, each transformation step changes one or more source constructs to target constructs in a metamorphosis-like manner. I.e., the source document will disappear during the process (unless the transformer is applied on a copy of the source document.) We will now in turn look at each of the two translation phases.

6.5.1 Application of the Transformations

The class 1 transformations that have been created partly automatically from the source-target relation and partly manually by the user, can be applied on a source document in a "batch-like" process. This is the process carried out by the transformer. The result delivered by the transformer is in most cases a *mixed-formalism document*, which later must be edited to a "clean target document." Apart from various transformation-messages, there is no interaction with the user during the automatic translation process.

The transformations are organized in a linear list, from which those transformations that belong to class 1 are executed (and the others are skipped.) The transformations are applied one after another in the sequence they are defined in the list. The ordering of the transformations is therefore significant in the cases where more than one is applicable. Thus, the most specific transformations (those with the most elaborate patterns and the strongest pre-conditions) should precede the more general transformations.

The environment will refuse to execute a manipulation on a document that violates the multi-formalism subphylum relation. If such a manipulation is attempted, it will be logged and reported to the user. It is, however, possible to specify that a transformation should be executed in "non-checked" mode in order to suppress any kind of syntax-check. This makes it possible to violate the multi-formalism subphylum relation, as already discussed in section 6.3.4.

In the current version of Muir, the algorithm for application of the

transformations is rough and time consuming. Each transformation is applied on the *whole* document. I.e., for each transformation the pattern matcher is activated on the entire source document in order to locate the relevant constructs to transform. If the transformer has to be used for real translation—and not only on toy programs—a more efficient transformer must be implemented. An improved transformer should only attempt to traverse the source document once, and for each construct visited during the traversal, it should only apply those transformations that have a chance to succeed.[9] As a more technical issue, unnecessary and time consuming copying should also be avoided during the translation process. The prototype transformer, which we have used for our experiments, *copies* each of the match constituents that have to be inserted into the replacement construct defined by the transformation (see section 4.1.1.) When possible, it would clearly be more efficient to *move* the match constituents into the replacement construct. In the general case it is necessary to copy, because a match-constituent may be inserted at more than one place in the replacement. However, in a more efficient transformer, we should only pay for this generality when it actually is used.

In some situations it is useful to *normalize the source document* before the multi-formalism transformations are applied. During the discussion of the translation of if-then-else chains in section 6.4.3 we made the assumption that both the then-part and the else-part of Pascal if-then-else statements are begin-end blocks. This is of course not always the case. But rather than have transformation variants for all the combinations of begin-end and "no begin-end", it is an advantage to normalize the source document such that all statements inside structured statements are embedded into begin-end structures. In general, a normalization of this kind may reduce the number of required multi-formalism transformations considerably. Symmetrically, some *post-normalization* of the target document may also be worthwhile. In the Modula-2 to Pascal case, elimination of begin-end blocks around single statements is an example of such a post normalization.

Some readers might at this place miss some quantitative results about the completeness of the Pascal Modula-2 transformations that we de-

[9]The alternative transformation strategy works in most, but not in all situations. Special care is necessary if a pattern refers to a construct that has been inserted by an already performed transformation.

scribed in section 6.4. We have two reasons for not including such results in the thesis. First, we have only translated a few programs with our transformer. Quantitative results based on these translations would not be interesting and representative. A more carefully prepared set of source documents would be needed to make quantitative conclusions. Secondly, it has *not* been a primary goal for us to produce high quality transformers between Pascal and Modula-2. Rather, we wanted with the two example languages to illustrate a semi-automatic translation approach. In a succeeding project, however, it would of course be interesting to test the practical value of our proposal through a series of quantitative measurements of the resulting transformers.

6.5.2 Manual Completion

The result of the automatic phase of the translation, the mixed-formalism document, is left in an Sedit instance in order for the user to complete the translation process. To visualize the mixed-formalism document, it can be presented with the source constructs and the target constructs shown in different fonts. Figure 6.17(a) shows a fragment of a Modula-2 program, and figure 6.17(b) contains the result delivered by the Modula-2 to Pascal transformer (before any post normalization.) Recall that we only have implemented translations of Modula-2 for statements with increment 1 and -1 (see section 6.4.2.) The constructs in normal font are source constructs (Modula-2), and the italic constructs are target constructs (Pascal.)

To eliminate the remaining source constructs, the user is supposed to use ordinary syntax-directed edit operations for the target formalism. However, to make the elimination process easier, a couple of special facilities will be introduced. The first of these is an edit operation that identifies and collects the remaining source constructs in the mixed-formalism document. The collection of source constructs produced by this operation can be used to visit each source construct in the document. During the traversal, arbitrary editing operations can be carried out on the visited constructs. In order not to invalidate the collection of source constructs produced by the search operation, the source constructs are visited in a postorder manner (bottom up in the AST.)

Let us assume that we have identified a source construct in the document, and that we want to eliminate it. Figure 6.18 sketches three cases

```
(a) IF ok
    THEN FOR i := 1 TO 100 BY 5
         DO IF even(i)
            THEN sum1 := sum1 + i
            ELSE sum2 := sum2 + i
            END END
    END

(b) if ok
    then begin
         FOR i := 1 TO 100 BY 5
         DO if even(i)
            then begin
                 sum1 := sum1 + i
            end
            else begin
                 sum2 := sum2 + i
            end END
    end
```

Figure 6.17: A Modula-2 fragment (a), and its partial translation (b).

that must be considered. The hatched areas in figure 6.18 symbolize target constructs, and the white areas are source constructs. Figure 6.18(a) contains two source constructs that each are located inside a target construct. It means that the choice phyla of the source constructs are target phyla, and therefore primitive target edit operations as well as composite templates belonging to the target formalism can be used directly to overwrite the source constructs with target constructs.

Figure 6.18(b) depicts a more typical situation, namely a situation where the two remaining source constructs in turn contain one or more target constituents. To deal with this situation, we introduce the other special facility, the edit operation that we call *split*. If the split operation is applied on one of the source constructs in figure 6.18(b), a copy of the source construct will be presented in a new Sedit window on the screen, and in addition a menu with meaningful target edit operations will be set up. (As in the earlier case, this is possible because the context of the source constructs are target constructs.) A target construct can hereby easily overwrite the entire source construct, and the already existing target constituents can be copied back into the document from the window produced by the split operation. Figure 6.17(b) shows a practical example of the situation in figure 6.18(b). In this case the split

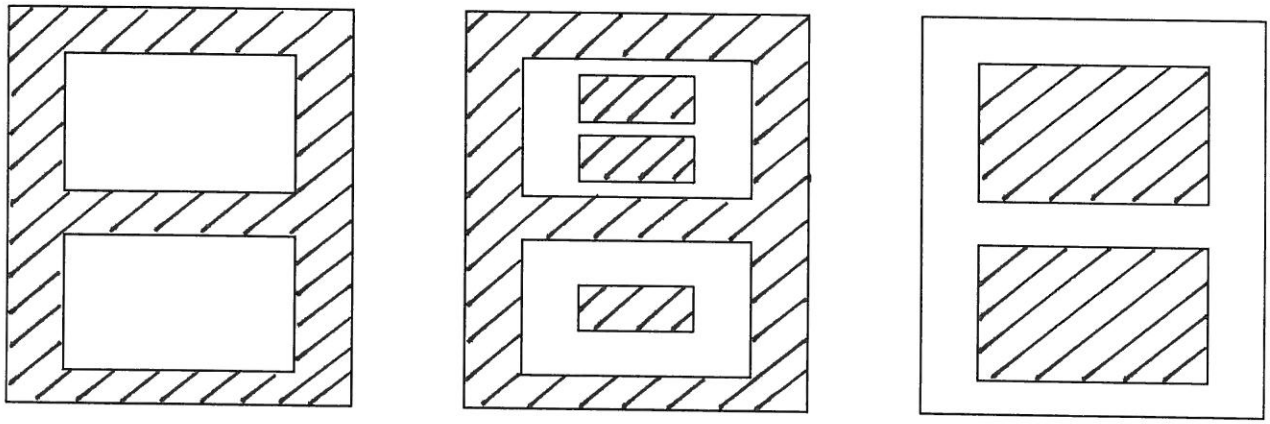


Figure 6.18: Three situations in mixed-formalism documents.

operation should be executed on the Modula-2 for-statement, which remains in the mixed-formalism program, and which therefore should be eliminated. The already translated constituent of the for-statement—the Pascal if-then-else statement—can in this way easily be put into the manually created translation of the for-statement.

Finally, figure 6.18(c) represents a case where the source construct is at the outer level of the document. One way to deal with this situation is to start a new Sedit instance for the target language, and then to transfer the already existing target constituents from figure 6.18(c) to the appropriate places in the new target document.

6.6 Summary

Implementing a multi-formalism translation tool can be a very work-intensive process. This is probably unavoidable if the differences among the source and the target languages are great. In this chapter we have developed techniques that reduce the workload when translating between more closely related languages. The user is required to define a relation among the phyla in the hierarchical grammars of the two languages. This can be done interactively by selecting pairs of phyla on the screen. The relation contains information that makes it possible to automate a major

part of the transformation creation process. The most straightforward transformations can be completed by the system, whereas others require some attention and manual refinement done by the user.

Depending on the efforts put into the completion of the set of transformations, the actual translation process can be carried out more or less automatically. In any case, however, it is possible or even likely that a few target constructs cannot be handled by the transformations, and it is therefore up to the user to complete this task too. We have described how the translation process can be finished via syntax-directed editing.

The relation among the phyla in the source grammar and the target grammar is also used to constrain the AST manipulations in a transformation step. We have demonstrated how the phylum-subphylum relations in the two grammars together with the source-target relation define a multi-formalism subphylum relation. The ideal is that every intermediate AST during the translation process should conform with the multi-formalism subphylum relation, and that the target document finally should conform with the pure target constraints. In practice, however, things are a little bit more messy. Some times it is hard not to violate the multi-formalism subphylum requirement, at least temporarily. And for some phyla, the source-target relation is not regular enough to ensure that multi-formalism correctness implies correctness w.r.t. the target grammar.

The techniques that we have described in this chapter have been implemented in a prototype tool, which is part of Muir. In order to try out the prototype tool we have experimented with translation of Pascal programs to Modula-2, and vice versa. To gain additional insight in the translation problems, it would be interesting to carry out experiments on pairs of other languages. Pascal and Modula-2 are probably to closely related to put forward definitive conclusions about our approach. Some additional work is also needed to make the automatic part of the translation process more efficient.

Chapter 7

Abstract Presentations

The separation of the internal document representation and the external document presentation is a fundamental principle in the work described in this thesis. The most dominant concern in the preceding chapters has been how to manipulate the internal document representation in order to obtain a given goal. With only a few exceptions it has been taken for granted how to present the documents on a screen.

There are good reasons for also considering aspects of document presentation in this thesis. First, in an environment like Muir and in most other contemporary editing environments, the user views, manipulates, and understands the documents *through* the presentation. In that respect the environment is very visually oriented. An environment with excellent functionality, but with poor presentation facilities seems for us to be in vain. Second, the characteristics of the presentations can improve the efficiency of the whole environment considerably. For example, the ability to present an overall diagram of a large document may make some kinds of decisions and modifications easier than if based on traditional and detailed presentations. Third, document representation and document presentations are separate issues, but they are not in all situations independent. Presentation-oriented editing operations clearly depend on the presentation style, and, as we shall see later in this chapter, also structure-oriented editing operations should in general be considered as presentation dependent.

The range of realistic screen presentations depends on the chosen internal document representation, especially if we are working in an interactive setting. The reason is that the consistency between the external presentation and the internal representation must be maintained within narrow time limits. If, for example, a document internally is represented

as an abstract syntax tree there might exist some relationships that cannot in a realistic way be presented in an interactive environment. The same relationships might be much more explicitly represented in, say, a relational data base representation (as in [56].) Until now, the thesis has been based on the assumption that the internal document representation is an abstract syntax tree, and this assumption will be maintained in this chapter as well.

The primary objective of this chapter is to introduce what we call *abstract presentations* and to show that it is possible to define such presentations in a general, language-independent way. As a natural extension to that, we are also concerned about how to edit a document through an abstract presentation. In section 7.1 we first discuss the traditional approach to presentation, and concurrently with that we motivate why better presentation techniques are needed. Next, our abstract presentation techniques will be described. We explain and exemplify in rather great detail two general presentation techniques that have been developed in the Muir project. In section 7.3 we examine how structure-oriented editing can be done through abstract presentations. Again, special attention is given to the two abstract presentation techniques that have been implemented. Following that, in section 7.4, we discuss how to integrate presentations at various abstraction levels into an environment like Muir. Finally, in section 7.5, we take a broad look at alternative presentation techniques, as reflected by the available literature.

7.1 Traditional Presentation Techniques

In an editor where the internal document representation is textual, the screen presentation and the internal representation are closely related. Normally, the entire document is presented in a fixed way. No variation nor adaptation to actual needs is possible. If the internal representation differs structurally from the screen presentation, efficiency concerns, for example during insertion of text in the middle of the existing text, typically explain the differences. Conceptually, however, the internal representation and the screen presentation can be considered as being identical.

There are several factors that explain the strong textual bindings of the presentations and the internal representations. First and foremost,

the dominant form of non-verbal communication in our culture is indeed textual. Second, the definition of artificial languages, for example programming languages, has traditionally been closely associated with textual description techniques. I.e., what constitutes a legal document is defined by a text-generating grammar, and the language definition only allows lexical variations such as spacing and commenting in the textual presentation of a given program. Third, the computer technology, as reflected by CRT screens, printers, and storage devices, is primarily well-suited to processing of text. All in all we witness a strong orientation towards the use of textual description, representation, and presentation techniques.

In an environment where the internal representation is a tree structure, such as an abstract syntax tree, the internal representation must via some interpretation be projected onto the screen. Even though a wide variety of presentation options are open in principle, the vast majority of the current systems imitate the traditional textual presentation. I.e., the entire document is projected onto the screen as linear, formated text, but for a big document, only a fraction of it is visible at any given time. In such presentations it is a serious problem to maintain the general overview of a large document. Flexible and secure manipulation of the document, decision making, and navigation in the document, can be greatly improved if the important, overall relationships between the major components in the document are made readily available on the screen.

In the same way as grammar descriptions and document representations have been raised to higher abstraction levels (abstract grammars, hierarchical grammars, abstract syntax trees), we find that this tendency also should be extended to the user interface of the systems. I.e., we find it highly relevant also to do research in more abstract presentation techniques that are independent of the particular internal document representation. In the following section we will introduce our notion and model of abstract presentations, and we will describe two particular language-independent, abstract presentation formalisms that we have developed for Muir.

7.2 Abstract Presentation Techniques

An *abstract presentation* is a picture of a document, in which certain objects and relationships are emphasized and displayed, and where the remaining objects are left out of consideration. There are no a priori bindings to any particular graphical means in abstract presentations. In many situations it is probably hard to find better means than text. In some situations, however, it is commonly accepted that alternative, graphical presentations are superior to textual presentations. Who is not familiar with the cliché that "a picture is worth a thousand words"? In this thesis we will exemplify this with pictures of well-understood mathematical structures, namely graphs, but other diagrams, nets, and formalisms would probably be valuable for some application areas as well. We do not find the use of so-called "graphics" as a goal in itself. Non-textual graphic means are only interesting if they convey some information that is hard to express clearly enough by textual means.

It would be an ideal situation to have available a variety of presentation options ranging from the well-known concrete ones to specialized, abstract presentations. At any time, it ought to be possible to view a document through the presentation that most satisfactorily meets the instant needs. When it makes sense, it would clearly also be profitable to be able to edit the document through concrete as well as through abstract presentations. As a basis for making abstract presentations and for editing a document through abstract presentations, we are now going to discuss a model in which a document is a set of objects and relations.

7.2.1 Objects and Relations

A document that belongs to an artificial language defines a set of objects together with a collection of relations among these objects. In this and the following section we will discuss abstract presentation techniques, via which a subset of objects and a subset of their relationships can be shown on the screen.

To get a concrete feeling for the objects and the relations we are talking about, we will start by considering some examples from various programming languages. In programming languages, objects like variables, constants, classes, modules, types, and procedures are of prime importance. All of these are named objects. In general, the phyla of a

grammar define these kinds of objects together with a set of more anonymous object types. A great variety of relations exit among these objects:

1. *Constituent*: a relation between constructs and their constituents.
2. *Successor*: A relation among elements in a list and their successor in the list.
3. *Local procedure*: A relation between procedure declarations and their local procedure declarations.
4. *Type of variable*: A relation between variables and type definitions.
5. *Import*: A relation between modules and the modules, from which they import facilities.
6. *Superclass*: A relation between classes and their superclasses.
7. *Calls*: A relation between procedure declarations. If P_1 and P_2 are related, it means that the procedure P_1 contains a call to P_2.
8. *Uninitialized variable*: A relation between procedures and variables of the procedures that are known never to be initialized.
9. *Free variable*: A relation between procedures and the free variable names of the procedures.

These relations, and many others, can be elaborated in such a way that they all are binary. The three relations mentioned first are *compositional*. I.e., they directly reflect the composition of a program. Relations 4 through 7 are *transverse relations* between more distant objects of a program. As opposed to the first three of the relations, the transverse relations do not reflect the actual composition of the program, but they are quite explicitly represented via bindings between applied and defining occurrences of names. Finally, the last two of the listed relations require a more thorough analysis to be figured out from an AST representation. We therefore call them *computed relations*.

A document can in principle be edited by adding or deleting elements to or from one or more of the relations. This we call *constructive use* of the relations. Constructive use of the relations *Constituent* and *Successor* is indeed possible, because the primitive edit operations and the list operations (see section 4.2.1 and section 4.2.5) can be understood

directly in terms of these relations. I.e., addition or removal of an element to/from one of the compositional relations correspond directly to one of the mentioned edit operations. For some of the other relations, there cannot in a reasonable way be established a one-to-one mapping between a given relation and a given document. It would, for example, be nonsense to modify a program by adding or deleting the tuple

"variable V is used freely in procedure P"

to or from the relation *Free variable*. The problem is of course that the variable V can be introduced freely in a countless number of ways in the procedure P. In the same way, adding the tuple

"module M_1 imports from module M_2"

to the relation *Import* may in a given language mean that M_1 imports directly from the module M_2, or that M_1 imports from M_2 via a module M_3, etc. If one and only one of these possibilities can be chosen as "the natural one", the relation can be used constructively. Besides the already mentioned constructive applications of *Constituent*, *Successor*, and possibly *Import*, it would probably be possible to use the relations *Local Procedure*, *Type of variable*, and *Superclass* constructively. It seems unrealistic to use the relation *Calls* constructively, and as with *Free variable*, it would be out of question to use the relation *Uninitialized variable* constructively.

The perception of a document as a collection of relations among various kinds of objects is a useful mental model, but we are still faced with the problem of actually presenting the relations on the screen. In the following section we will discuss various presentation options, and after that we will study two abstract presentation techniques that we have developed in the Muir project.

7.2.2 Presentation of Relations

A *presentation style* is a set of presentations using particular graphical means and structures. Besides the well-known textual presentation style, we will in this thesis primarily look at a graph presentation style, but many other styles could be imagined as well. With respect to a given presentation style, a *presentation formalism* defines a framework in which to formulate presentation rules for internal structures. A set of rules in a presentation formalism that defines the presentation of grammatical

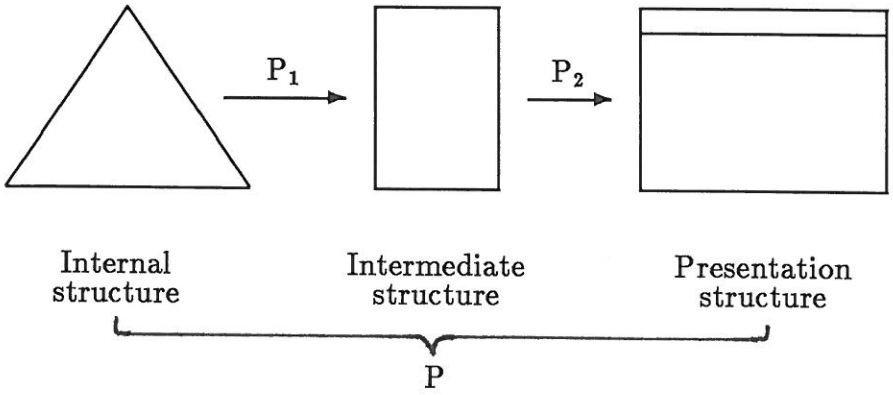


Figure 7.1: The presentation function P is $P_2 \circ P_1$

constructs is called a *presentation scheme*. A presentation scheme is identified by a name that is associated with every presentation rule in the scheme.

The component that effectuates the presentation process will be called a *presenter*. The presenter can be thought of as implementing a *presentation function* whose domain is the internal document representation, and whose range is the actual presentation structure on the screen. The presentation function can be decomposed into two or more functions, whereby one or several intermediate presentation structures are defined. Figure 7.1 outlines this situation. For abstract presentations, one of the intermediate presentation structures is a binary relation among the objects that we choose to emphasize in the presentation. The splitting of the presentation function is useful because it allows us to characterize the qualities of the presentation process more carefully.

Detailed presentations that belong to the textual presentation style are well-suited to illustrate compositional relations in minor objects. In an environment like Muir, the textual presentation formalism allows presentation rules to be associated with the operators in the grammar. A presentation rule determines which of the abstract constituents of an operator to present, the ordering of the constituents in the presentations, the concrete syntax (keywords, punctuations, and fonting), and the layout (CR's and indentation.) The compositional relations are quite naturally visualized by the textual nesting of constructs, and by textual sequencing.

Frequently it is illustrative to present binary relations as a directed graph structure. There is a straightforward translation of a binary relation to a graph structure, namely one where the pairs of the relation are interpreted as edges in the graph. (If several relations have to be pre-

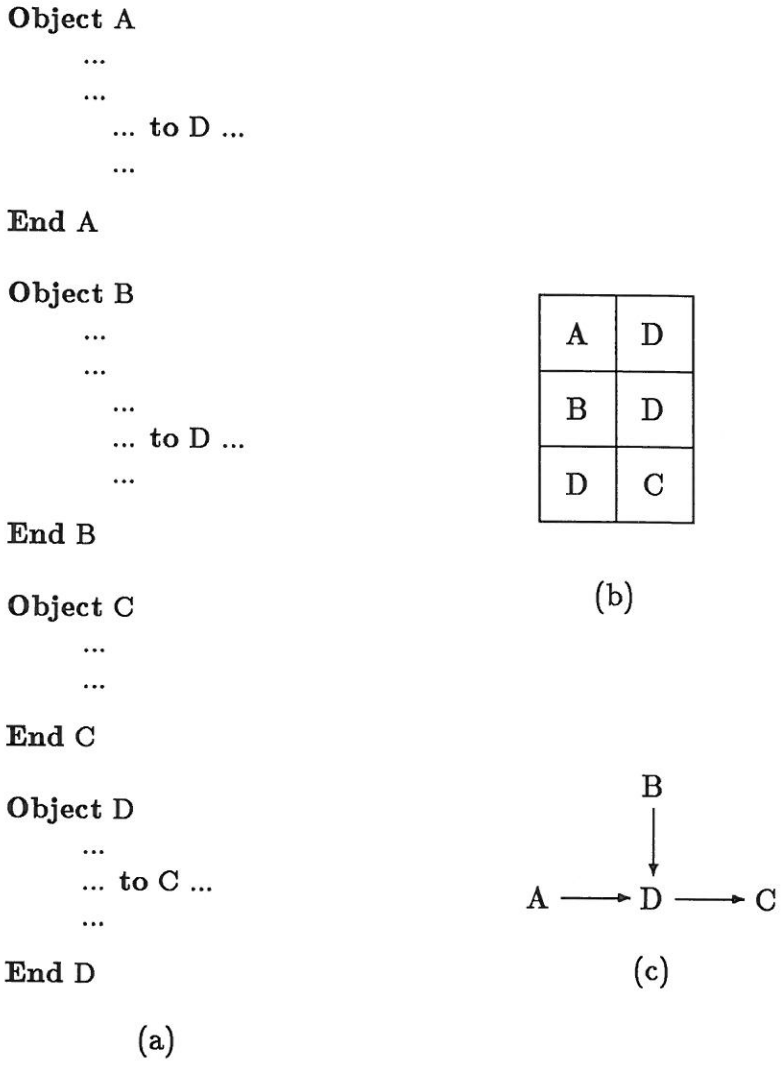


Figure 7.2: A textual, a relational, and a graph presentation.

sented in the same graph, different relations can be distinguished either by different looking edges, or via distinctions in the "from nodes" and/or the "to-nodes.")

We are primarily interested in using the graph presentation style for the illustration of overall, transverse relations among objects. As already indicated in 7.1, we find that proper illustration of the overall relationships is very important, because these relations have a tendency to "drown" in less important details when illustrated in textual presentations. Moreover, graph style presentations seem in many situations to be particularly well-suited to present these relationships. Consider the examples shown in figure 7.2 where we are interested in capturing the relations among the objects A, B, C, and D. In the traditional textual

presentation in figure 7.2(a) the relation is defined by the "to-clauses." The relation is scattered around in the text, and especially if the elisions (marked by "...") represent large structures, it may be difficult mentally to grasp the relation. In figure 7.2(b), the relation is shown in table form, and in figure 7.2(c) it is shown as a graph. We are convinced that the graph provides the clearest[1] presentation of the relation. The reason is that in the graph presentation, each object is mentioned only once, and that a relationship between two objects is illustrated in the simplest way we can imagine, namely via a straight line.

In the following two sections we will describe two abstract presentation formalisms that have been implemented in Muir. We start by describing a rather simple technique that allows us to present compositional structures. Following that, we describe how to generate presentations of transverse relations among objects.

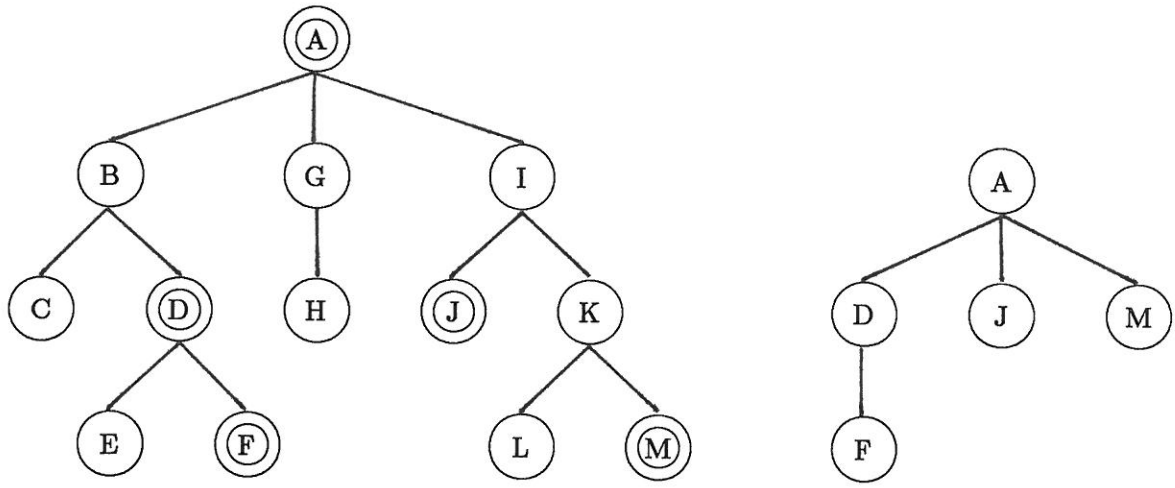
7.2.3 Compositional Tree Presentations

It is hard to illustrate both the overall and the fine-grained compositional relations in a single detailed textual presentation. It is, for example, difficult to capture the procedure-structure of a big program if all the more detailed aspects have to be presented as well. As already noticed in the previous section the overall relationships tend to "drown" in all the detailed relations. However, it is not hard to write a textual presentation scheme that filters out the disturbing details [29]. In this section we will demonstrate that it is possible to design a special presentation formalism that makes it even easier to present overall compositional relations. We have chosen to show the overall compositional relations as tree structures, but an indented textual presentation would be equally useful in this situation.

We decompose the presentation function into two functions whose intermediate presentation structure is a binary relation that defines a tree structure. I.e., the presentation function is split into:

Select: AST $\rightarrow$ *BinaryRelation*
Layout: BinaryRelation $\rightarrow$ *ScreenTree.*

[1]It is of course dangerous to claim that a presentation style is "better" and "clearer" than others without any empirical evidence. However, by looking at a collection of blackboards at places where a group of people illustrates relationships among various objects, it will probably be clear for most sceptics that graph-like presentation styles are very popular and widespread.



(a) (b)

Figure 7.3: An AST (a) and its presentation tree (b).

The "critical part" is the function *Select* that selects the contributing objects in the AST structure. The *Layout* function is accomplished in Muir by the Interlisp Grapher [46], which is a package that maps graph structures in a particular list representation onto the screen. We will now take a closer look at a presentation formalism that allows us to define the *Select* function.

The presentation technique is based on the straightforward observation that an arbitrary marking of a set of nodes in a tree defines a forest of trees by following the original tree links between the marked nodes. Figure 7.3(a) shows an example of an AST with marked nodes (double circled), and figure 7.3(b) shows the derived forest, in this case only consisting of a single tree. A tree presentation formalism needs to define the marking of nodes and the text label for each node in the presentation. Both of these are defined in so-called *tree presentation rules*, which belong to the operators—or equivalently, to the terminal phylum declarations—of the grammar. A tree presentation rule is a list of presentation elements, one for each of the operator constituents. A presentation element is either a name of a textual presentation scheme, or one of the keywords CONTINUE and STOP. Let us assume that the presentation rule

$$(\mathrm{PE}_1\ \mathrm{PE}_2\ \ldots\ \mathrm{PE}_n)$$

is associated with the terminal phylum declaration

P: P_1 P_2 ... P_n.

If PE_i is a name of a textual presentation scheme, the P_i constituents of the P-constructs in the document become objects in the presentation, and the node labels that present these object are produced by applying the textual presentation scheme on the corresponding constructs in the AST. If PE_i is the keyword CONTINUE, the P_i constituents of the P-constructs do not contribute to the presentation, but the search continues in these branches of the AST. The keyword STOP works in the same way, but the searching will be terminated in this branch of the AST. Proper use of the STOP keyword in presentation rules may reduce the time spent by the presenter to create the tree presentation.

The primitives of the compositional tree presentation formalism, as described above, can play together with a variety of the general presentation mechanisms in Muir, most importantly the conditional presentation facility. See [73] and [74] for additional details.

The primary advantage of the compositional tree presentation formalism is that it only needs to associate presentation rules with operators that affect the presentations directly. So in most situations a tree presentation only requires a few presentation rules in order to be fully defined. In traditional frameworks, textual presentation rules require presentation rules on all operators "in between real contributions."

Figure 7.4 shows an example of a compositional tree presentation of a Modula-2 program.[2] The purpose of the presentation is to illustrate the compositional relations among procedures and modules in a compilation unit. In appendix B we have listed an excerpt of the Modula-2 grammar to show how the presentation schemes used in figure 7.4 are defined. Only three presentation rules are required to define the presentation. Among these, one is a tree presentation rule, and the two others are text presentation rules that define the node labels. The *declaration-list* operator has a tree presentation rule called *MO.TreePs*. In Muir, the names of the presentation rules consist of two components: A short name of the language, and the proper name. Essentially the *Mo.TreePs* presentation rule reads "(MO.LabelPs CONTINUE)", where Mo.LabelPs is the presentation element that corresponds to the head of the list, and

[2]The presentation in figure 7.4 together with the presentations in figure 7.5 on page 149, and figure 7.8 on page 159 illustrate aspects of a Modula-2 implementation of the Muir environment. However, no such implementation exists. The document which is presented in the figure 7.4, 7.5, and 7.8 is only refined to such a degree that these presentations can be generated.

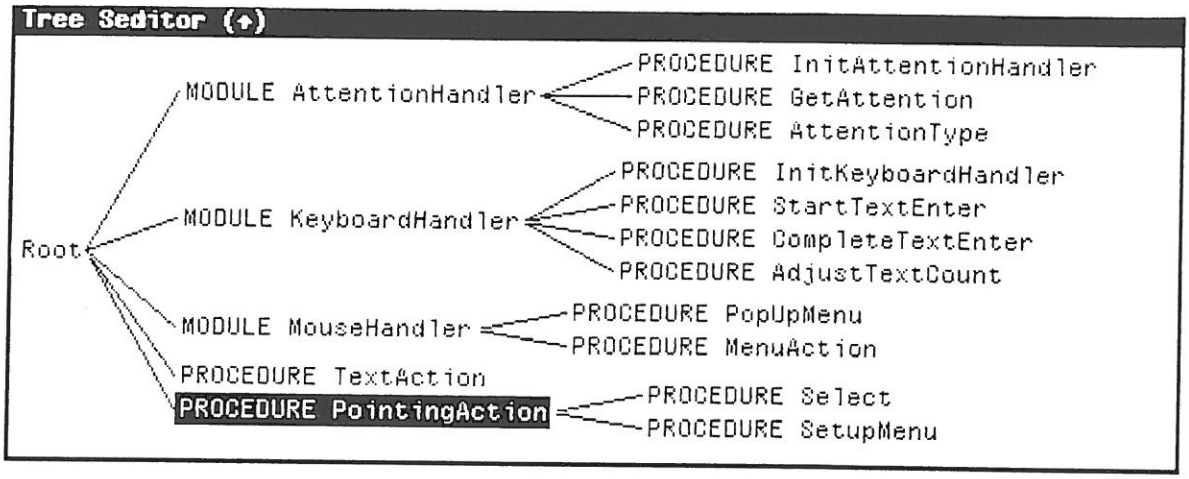


Figure 7.4: A compositional tree presentation of a Modula-2 program.

CONTINUE is the presentation element that corresponds to the tail of the list.[3] The "choice-construct" in the presentation rule (see the appendix) ensures that the declaration only is presented if it is a procedure or a module. The operators *module-declaration* and *procedure-declaration* show how the label scheme *MO.LabelPs* is defined.

7.2.4 Transverse Graph Presentations

In this section we will describe a presentation formalism that allows us to define more transverse binary relations among selected objects, and to display these relations in graph structures on the screen. Transverse graph presentations can be seen as generalizations of the compositional tree presentations that we described in the previous section. As in section 7.2.3, the presentation function will be split into two sub-functions:

Select-and-relate: AST $\rightarrow$ *BinaryRelation*
Layout: BinaryRelation $\rightarrow$ *ScreenTree.*

Also in this case the Interlisp Grapher will be able to carry out the *Layout* function, although some aesthetic graph layout questions become more urgent when "real" graphs (as opposed to trees) are displayed.

In addition to selecting the objects in the document that contribute to the graph presentation, the relations among these objects have to be defined too. The selection of objects, and the production of the node

[3] As described in section 3.2.1, lists are in Muir represented as nested binary trees of heads and tails.

labels can be done in the same way as in the compositional tree presentations. The relations among the objects, on the contrary, have to be defined explicitly here. (In the previous section, the relations among the objects were inherited from the composition of the document.)

The concept in the graph presentation formalism that allows us to define transverse relations among objects is called *object identifications*. Each object that contributes to the graph presentation must have associated an object identification. In order to define the edges of the graph, a list of object identifications is generated for each of the objects. If an object has associated an object identification Id_{obj}, and if it generates a list of objects identifications Id_1, Id_2, ..., Id_k, then there exist edges from Id_{obj} to Id_j for $j \in [1..k]$. This information defines the relation among the objects, and the *Layout* function can project it onto the screen. Let us now take a closer look at the mechanisms in the graph presentation rules that allows us to define the object identifications.

A graph presentation rule for the terminal phylum declaration

$$\mathrm{P}: \mathrm{P}_1\ \mathrm{P}_2\ \ldots\ \mathrm{P}_n$$

is a list of presentation elements

$$(\mathrm{PE}_1\ \mathrm{PE}_2\ \ldots\ \mathrm{PE}_n).$$

In the same way as in the compositional tree presentations, a presentation element can be one of the keywords STOP and CONTINUE. A presentation element can in addition be a triple

[*Node-label-scheme Identification-scheme Edge-scheme*].

If C is an immediate constituent of a P-construct in a document, for which the corresponding presentation element is such a triple, then C is designated as an object that contributes to the presentation. Furthermore, *Node-label-scheme* defines the node label of C (in the same way as in the previous section), and *Identification-scheme* and *Edge-scheme* define its object identifications and its list of "to-object" identifications respectively.

An object identification is a symbol which is required to identify the object in a unique way. The object identification is generated by applying the identification scheme on the object, in a similar way as a textual presentation scheme is applied on an AST-constituent to produce a node label. Also the edge scheme is applied on the object, and it is expected to generate a string of the form

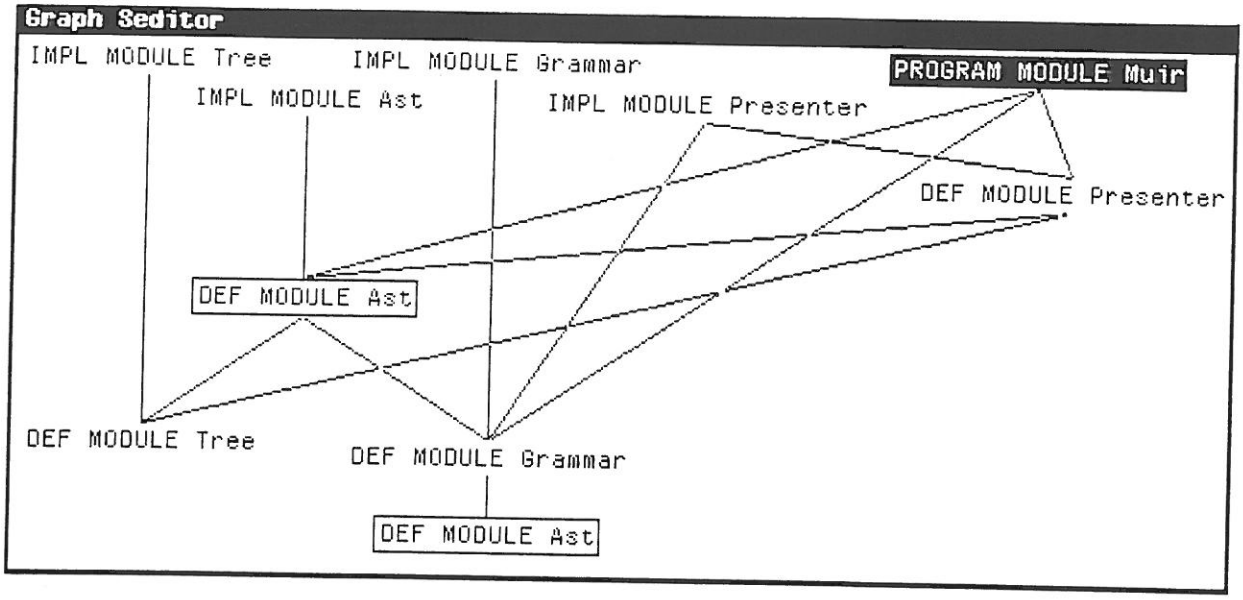


Figure 7.5: A transverse graph presentation of Modula-2 compilation units.

"Identification-1 SEP Identification-2 SEP ... Identification-n SEP"

which is interpreted as a list of n object identification symbols. SEP is a special keyword that separates the object identification symbols in the string.

We have written several graph presentation schemes for the grammars that are supported by Muir. Let us first take a detailed look at the definition of an "import-export presentation" for Modula-2. Figure 7.5 shows an example of such a presentation.[4] The edges in the graph are implicitly oriented from the top towards the bottom of the paper. Boxed nodes represent nodes that participate in cycles in the graph. An arch from a module M_1 (down) to a module M_2 means that M_1 imports facilities from M_2. For the sake of simplifying the example we have modified the meaning of the Modula-2 import clauses slightly. We assume that if a module M_1 imports facilities from a module M_2 then an explicit import clause with a "FROM M_2" indication is present in M_1. The FROM clause in Modula-2 actually implies an unqualification of the names in an import list. I.e., "FROM M_2" is not necessary in order for M_1 to import from M_2. Rather, a remote entity X from M_2 can be referred to as "M_2.X" in M_1.

[4]The layout of the presentation in figure 7.5 has been done manually.

The presentation rules used to generate the presentation in figure 7.5 are shown in appendix B. We organize all Modula-2 compilation units in a linear list. The operator *CompilationUnit-list*, which defines this list, has a graph presentation rule

```
([MO.LabelPs MO.ImportExportIdScheme
MO.ImportExportEdgeScheme] CONTINUE)
```

which belongs to the presentation scheme *MO.ImportExportPs*. This presentation rule states that there must exist a presentation node for every compilation unit in the list. The *ImportExportPs* presentation rules for the operators *MainModule*, *ImplementationModule*, and *DefinitionModule* are all "stop rules", i.e., the interior of the modules are not searched for contributing objects. This makes it relatively fast to create the presentation. Local modules of compilation units are consequently not included in the presentations defined by the presentation scheme *ImportExportPs*. The label scheme and the identification scheme of the graph presentation are defined in the operators *MainModule*, *ImplementationModule*, and *DefinitionModule*. The object identifications are of the form *MainModule.ModuleName*, *ImplModule.ModuleName*, and *DefModule.ModuleName* respectively. The edges are created by the *MO.ImportExportEdgeScheme* rules in the three module operators plus those in the *import*, *import-list*, and *FromModule* operators. The edge scheme essentially locates the constructs "FROM module IMPORT <name-list>" and translates them to a list of identifications. In addition, the edge scheme for *ImplementationModule* makes explicit in the presentation that an implementation module always imports the facilities from its corresponding definition module.

It would be more difficult, although not impossible, to generate a similar import-export presentation for a "real Modula-2" program in the current graph presentation formalism. The reason is that a module M_1 may import a quality Q from M_2 solely by referring to "M_2.Q". In other words, the fact that M_1 imports facilities from other modules is not defined at a particular and fixed place in M_1, but rather, it can be scattered around in the whole module. It might be worthwhile to relax the way edges are defined by the edge scheme. Instead of putting edge scheme rules on all the operators in between the object and the construct that define a contribution (in between the module-declaration and the con-

struct "M_2.Q" in the example), it might be easier to define the edges by searching for the constructs in the objects that contribute with edges (along the same line as contributing objects are located in the graph and the tree presentation formalisms.)

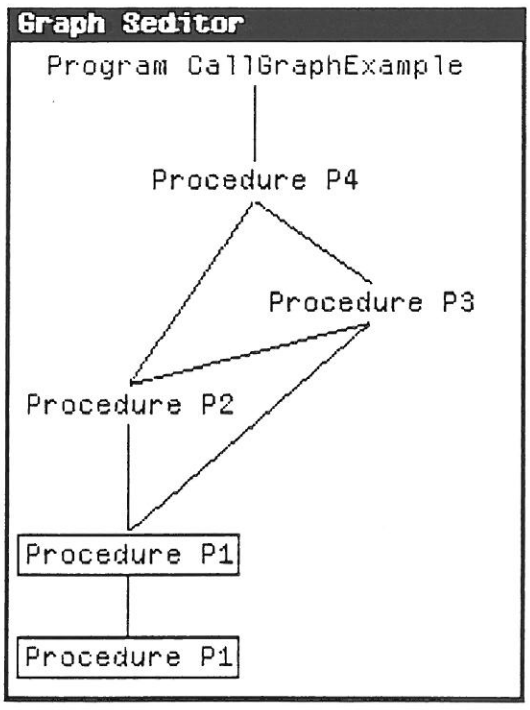
As another example of a transverse graph presentation, we have implemented a graph presentation scheme that, when applied on a Pascal program, presents the static procedure call graph of the program. Figure 7.6 shows a sample Pascal program and the generated procedure call graph. In the same way as in figure 7.5, the edges in the graph are oriented from the top to the bottom of the paper, and boxed procedures participate in cycles, i.e., they are recursive. The graph presentation in figure 7.6(b) is not well-suited for constructive use. Too many details have been left out, and it would be strange to modify the Pascal program directly via this presentation. However, it conveys useful cross reference information, and the presentation can be used as the basis for initiation of other presentations, from which the underlying program can be modified.

As the last example of transverse graph presentations, we will consider the presentation of a collection of categorical phylum declarations as a phylum hierarchy. In section 3.1.2 it was explained that a categorical phylum declaration is an object that enumerates its immediate subphyla, but not its more distant subphyla. A phylum hierarchy was also formally defined in section 3.1.2. In the AST representation of a hierarchical grammar in Muir the phylum declarations are organized in a linear list. I.e., the AST representation of the phylum declarations does not reflect the phylum hierarchy directly. Figure 7.7(a) sketches the list of phylum declarations that define the type hierarchy in Pascal, and figure 7.7(b) shows the graph presentation of the phylum *type*. Figure 3.1 on page 31 shows a similar presentation, namely a presentation of the phylum *statement* in Pascal. It clearly makes sense to use the phylum hierarchy presentations constructively. I.e., instead of modifying the phylum hierarchy indirectly in a presentation similar to the one shown in figure 7.7(a), specialized edit operations should make it possible to modify the phylum hierarchy in terms of nodes and edges in the graph presentation. In the following section we will more thoroughly discuss editing of abstract presentations, in particular editing based on transverse graph presentations.

```
Program CallGraphExample;
Var Count: Integer
Procedure P1(N: Integer );
Var I: Integer
Begin
   for I := 1 to N
   do Begin
         Count := Count + 1;
         P1(N - 1)
      End
End;
Procedure P2(P: Integer );
Var J: Integer
Begin
   J := P;
   while J >= 0
   do Begin
         P1(J);
         J := J - 1
      End
End;
Procedure P3(P: Integer );
Begin
   Case P div 2 of
      0: P1(P);
      1: P2(P)
   End
End;
Procedure P4(Q: Boolean ; R: Integer );
Begin
   if Q
   then P2(R)
   else P3(R)
End;

Begin
   Count := 0;
   P4(TRUE,5)
End.
```

(a)



(b)

Figure 7.6: A traditional and a graph presentation of a Pascal program.

Categorical Phylum type:
Subphyla: simple-type structured-type pointer-type

Terminal Phylum pointer-type:
Operators: pointer-type

Categorical Phylum structured-type:
Subphyla: array-type record-type set-type file-type packed-type

Categorical Phylum simple-type:
Subphyla: scalar-type subrange-type type-name

Terminal Phylum packed-type:
Operators: packed-type

Terminal Phylum file-type:
Operators: file-type

Terminal Phylum set-type:
Operators: set-type

Terminal Phylum record-type:
Operators: record-type

Terminal Phylum array-type:
Operators: array-type

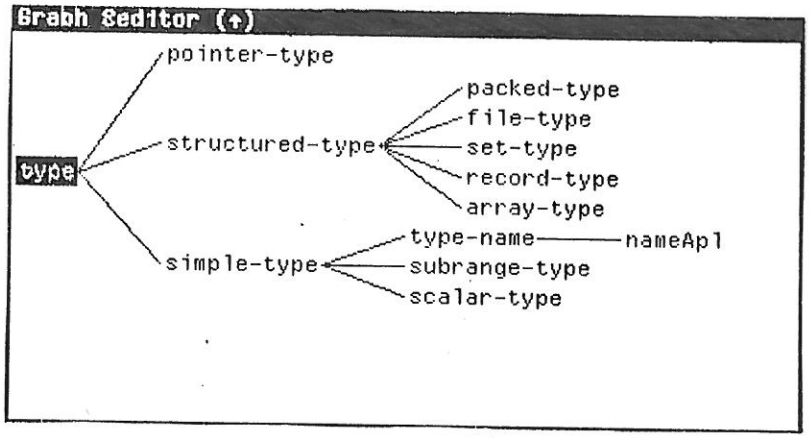
Categorical Phylum type-name:
Subphyla: nameApl

Terminal Phylum subrange-type:
Operators: subrange-type

Terminal Phylum scalar-type:
Operators: scalar-type

Terminal Phylum nameApl:
Operators: nameApl

(a)



(b)

Figure 7.7: A list of phyla, and its graph presentation.

7.3 Editing an Abstract Presentation

In section 7.2 we noticed that in some situations it makes sense to modify a document by modifying one of the relations that are defined among the objects of the document. We called that constructive use of the relations. In other situations, the information contained in the relation is so sparse that a modification of the relation cannot be projected back into the AST representation of the document. In this section we will assume that the relations are of such a nature that constructive use of them makes sense.

In an abstract presentation, one or a few kinds of objects are emphasized, and certain relationships among these objects are displayed. These restrictions affect the set of edit operations that meaningfully can be applied on the document through a given abstract presentation. In general, the edit operations should manipulate the objects and the relations at such an abstraction level that the edit operation makes sense w.r.t. the presentation. As an absolute minimum, the effect of each edit operation should be observable through the presentation. In this section we will discuss how these "guidelines" affect the elaboration of the edit operations on compositional tree presentations and on transverse graph presentations.

As already observed in chapter 4, there are two fundamentally different ways to edit a document in a structure-oriented environment: Via presentation-oriented edit operations or via structure-oriented edit operations. Presentation-oriented edit operations manipulate the elements of the presentations, for example nodes and edges in a graph presentation, and the modifications at this level must be projected back into the AST representation of the document. Structure-oriented edit operations manipulate the internal AST structure, preferably in such a way that the modifications can be understood in terms of the aspects of the document that are emphasized in the presentation. We are not going to discuss presentation-oriented editing techniques in this chapter. As in chapter 4, we will also here stick to the structure-oriented editing approach.

7.3.1 Compositional Tree Presentations

The structure-oriented edit operations that we discussed in chapter 4 were primarily well-suited to manipulate the *detailed* compositional relations of a document. If more *overall* compositional relations are presented, we

must be careful to manipulate constructs at the proper abstraction level. Let us exemplify this by taking a closer look at the edit operations on the tree presentation in figure 7.4 (see page 147.)

When editing a document through an overall, compositional presentation, edit operations that insert composite templates (see section 4.2.2) should in general be preferred to primitive edit operations. If we, for example, use a primitive edit operation to expand one of the procedure declarations in figure 7.4, it is difficult to insert a local procedure or a local module in that procedure because it does not contain the declaration placeholder for local declarations. An appropriate composite template, on the other hand, could be refined in such a way that continued editing at the selected level of abstraction is possible.

In the current implementation of Muir, a compositional tree presentation is re-generated from the AST after every edit operation. This is clearly too slow. In the same way as we have implemented incremental screen updating techniques for text style presentations, this should also be done for the compositional tree presentations. It would be straightforward to make the *Select* presentation function (see 7.2.3) update the binary relation incrementally after each edit operation, and to redo the *Layout* presentation function in its entirety. Hereby the rather time consuming search in the AST for marked nodes is eliminated. Experiments indicate that following this strategy, the screen update time will in most situations be reasonable.

7.3.2 Transverse Graph Presentations

The structure-oriented edit operations that manipulate the compositional structures cannot in general be expected to do very well if applied on a graph presentation that illustrates transverse relationships among objects. The problem is that the edit operations that we introduced in section 4.2, and which primarily were intended to manipulate the compositional relations, contain too much freedom, and they consequently are too primitive to be adequate for editing of transverse graph presentations. In this section we will discuss a general edit operation framework, which is well-suited for editing of a document through graph presentations. The editing technique that we propose has not been implemented and tried out in Muir.

Only a small number of different structure-oriented edit operations

are needed to edit a document via a graph presentation. It should be possible to

1. add a new unrelated object to the document,
2. delete an existing unrelated object from the document,
3. relate two existing objects,
4. un-relate two already related objects in the document, and
5. modify the aspects of the document that are presented in the node label.

One could of course imagine other edit operations and other combinations, but these five groups span the range of possible modifications in a particularly simple way.

The five kinds of edit operations are generally defined for any transverse graph presentation, but they need to be specialized for each particular graph presentation scheme. The basic pattern of action is, for example, available on both the import-export presentation (as in figure 7.5), and on phylum hierarchies (as in figure 7.7). But for each of the presentations, it must be defined in terms of the AST representation of the documents what it means to add and delete new objects and relations. We will now propose how this specialization of the edit operations could be done. To be concrete, we choose to discuss the edit operation that relates two existing objects in a document.

The general edit operation that relates two objects needs two arguments, namely the two objects that must be related. It is not crucial for this discussion whether the command syntax is prefix, infix, or postfix, but let us just assume that the infix command syntax is chosen. I.e., the source node is selected before the edit operation is initiated, and as the initial action of the general edit operation, the user is asked to point at the destination node. The general edit operation then activates a transformation, which is specific for each particular graph presentation scheme. In Lisp terminology, the general edit operation provides a hook [82,85], which must be filled in by the definer of the presentation scheme. For the edit operation that relates two objects, the source node and the target node should be passed as arguments to the transformation, and the transformation is expected to update the AST appropriately.

In many cases we imagine that the underlying AST can be modified by a single pattern-based transformation (see 4.1). In others, several modifications of the AST are needed in order to maintain the consistency between the AST and the graph presentation. If, for example, we want to support a new kind of operation that both adds and relates an object O_1 to another object O_2, both the ASTs that represent O_1 and O_2 are affected. As another example, the edit operation that modifies the label of a graph node N may require changes to N and to all objects that are related to N (if all references to N should be preserved.) Thus in general, the transformation may be implemented by a Lisp function, which in turn may activate several pattern-based transformations.

When the AST is modified, the presentation needs to be updated too. It is out of question to regenerate the whole graph presentation from scratch after each edit operation. It can be quite time consuming to generate the presentation, and even worse, it is hard automatically to solve the layout problem, i.e, the mutual placement of nodes in the window. Some kind of incremental screen updating is necessary. It is trivial to update the screen following the operation that adds a new relation between two nodes, but for the operations that add new objects to the document, we find it most realistic that the user during interaction places the node in the window. It would clearly be desirable if the graph layout could be preserved between sessions. To accomplish this, some presentation information needs to be stored in association with the document.

Even though we claim that the five general edit operations are sufficient for creation and modification of any document through a graph presentation, some of the general edit operations may need two or more specializations. For example, there should be three variants of the "add object" for the import-export presentation, one for each of the three kinds of modules (compilation units) in Modula-2. The operation that adds a new phylum to the phylum hierarchy should also have two variants, one for adding a categorical phylum and one for adding a terminal phylum.

7.4 Environmental Support

Abstract presentations are useful for maintaining the overall relations of a document. But frequently it is necessary during an editing session to

present particular objects at a lower abstraction level, in order to examine or change some of the more detailed relations of the document. In this section we will study how the environment can support this flexibility, and how the presentations at the various abstraction levels interact, when several of them are present on the screen at the same time.

The operation that chains the various presentations together in the Muir environment is called *Re-present*. *Re-present* initiates a new presentation of the current focus in another window. In its basic form, *Re-present* attempts to use a presentation scheme at a lower abstraction level. This reflects the experience that an overall abstract presentation frequently is used as a platform for initiation of more detailed presentations. Figure 7.8 illustrates this for the Modula-2 presentations that we have studied earlier in this chapter (see figure 7.4 and figure 7.5.) The upper graph presentation shows a number of modules and a particular kind of relations among them (the import-export relations.) The focus in this window has been re-presented in the middle window, which shows the program module *Muir* in some detail. The selected procedure *PointingAction* has finally been re-presented in the lower window. This presentation shows all the details of the procedure that are not already apparent in other windows. It implies that the two local procedures of *PointingAction* are not presented at the most detailed abstraction level. We find textual nesting at the detailed level inappropriate to visualize relations among textually large objects. This approach helps to keep all the presentations at a size that make them fit into a single window on the screen. The three abstraction levels shown in figure 7.8 were proposed in EKKO [10,69], which is a design of a programming environment for a Pascal-like language.

There exist various specializations of *Re-present* which make it possible to request the usage of a particular presentation style and a particular presentation scheme. It is, for example, possible to go from a rather detailed presentation to a presentation of the same structure at a higher abstraction level.

As illustrated above, it must be anticipated that some constructs in a document are presented more than once on the screen at the same time. For example, the procedure name *PointingAction* is presented both in the middle window and in the lower window in figure 7.8. We say that two presentations P_1 and P_2 are *overlapping* if changes in the underlying internal structure through P_1 can affect P_2, or vice versa. Several issues

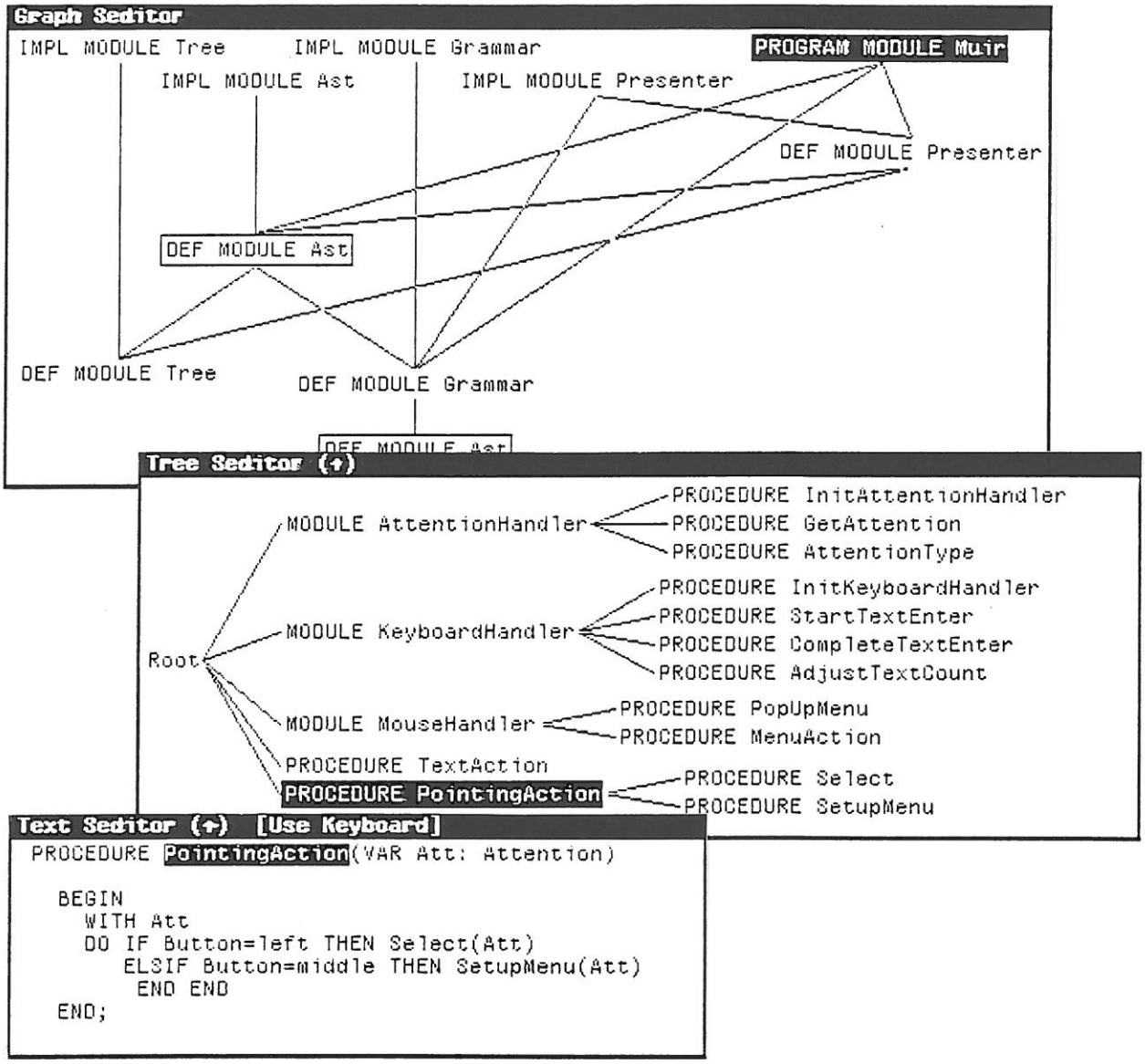


Figure 7.8: Related presentations at three different abstraction levels .

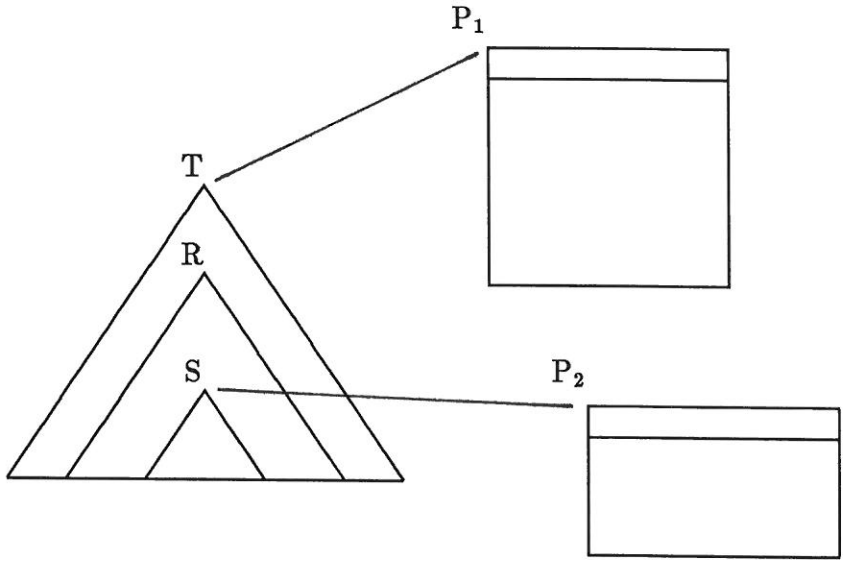


Figure 7.9: Overlapping presentations of a tree T.

need clarification when we work with overlapping presentations. First, it is possible to affect the existence and the identity of one presentation through operations on another presentation. We need a general "policy" to handle that problem. Secondly, we need to decide to which degree overlapping presentations should be consistent with the underlying AST structure. We will now in turn discuss each of these issues.

Let us assume that we have a situation as outlined in figure 7.9. The AST T is presented in P_1, and T's constituent S is presented in P_2. If the construct R is replaced by another construct R' through the presentation P_1, the presentation P_2 could be affected. There are at least three different interpretation of the replacement:

1. P_2 is lost, because the internal structure S, which it presents, no longer is part of the tree T.

2. P_2 is not affected, and S will continue to exist.

3. P_2 will after the tree substitution present some substructure of R'.

We prefer the second solution. We consider a presentation as a "handle" to the underlying document, and as long as such a handle exists we cannot lose it. The part of the document "in between" R and S, on the

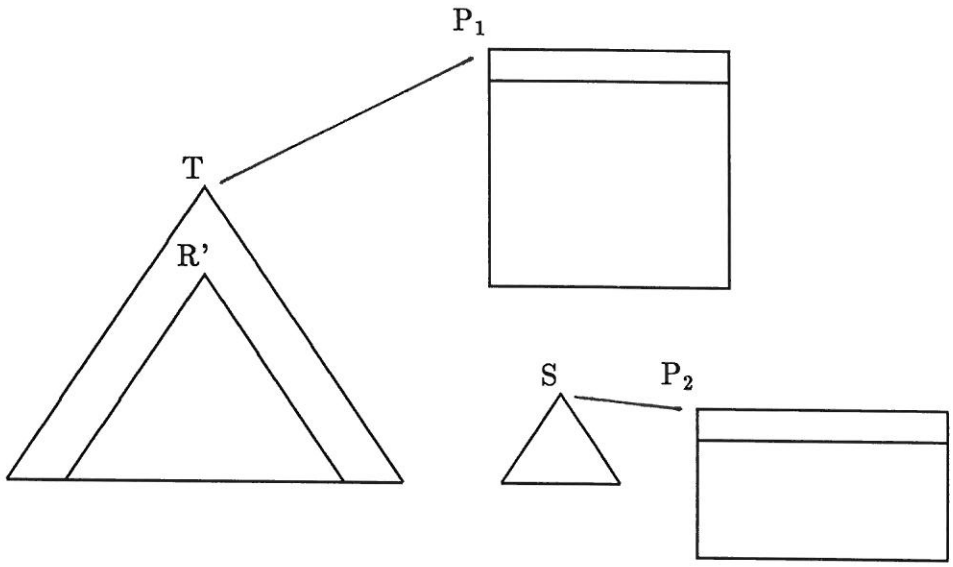


Figure 7.10: Situation after the replacement of R by R'.

other hand, will be lost. Using this interpretation, the situation after the substitution is as shown in figure 7.10.

In the ideal environment, every presentation on the screen should be consistent with its underlying AST structure. In case of overlapping presentations, a modification in one of the windows may affect a number of other windows as well. In some situations, or for some presentations, it might be too time consuming to maintain total consistency between the AST structures and their presentations. We feel that it is too primitive, and perhaps even too dangerous to make it the responsibility of the user to ask for regeneration of a possibly inconsistent presentation. If the system cannot afford "the total consistency approach", it should at least mark the inconsistent presentations as obsolete. The default in Muir is to keep all the presentations consistent at any time. As an option, the "mark as obsolete approach" can be turned on via a global flag. If it is time consuming to update some presentations, this option may make the environment more pleasant to use, because the delay following an edit operation is decreased.

Let us finally sketch how we in the implementation of Muir manage to keep all affected presentations up to date. Let us consider what happens internally when a construct, say R in figure 7.9, is replaced by another construct R'. We maintain a mapping

NodeWindowMap: *TreeNode* → *List of presentations.*

This mapping makes it possible to determine efficiently whether a given node in an AST is a root in a presentation, and moreover, it gives us the presentations rooted by the node. Every presentation affected by the tree modification has its origin on the path from R to the root of the AST.[5] For each presentation found in this way, we check to see if R actually is presented. For each presentation found to be affected, we issue a request for screen updating (passing information about R', R, and its position in the presentation). Such a request is stored as a so-called *screen update record* in a screen update list. An edit operation, which may involve several tree substitutions, can in this way request many screen updatings. Following the edit operation, a screen update procedure effectuates the screen updatings from the screen update list.

7.5 Related Work

In this section we will briefly survey various alternative presentation techniques that have been described in the literature. It is appropriate to start by looking at some early work in the area of text processing. Already in the mid-sixties the presentation problem was recognized by Engelbart and his group at SRI where they developed the NLS system [26,83]. We have already mentioned NLS in chapter 2 (page 10.) A hierarchical structuring of text together with the possibility of controlling how this hierarchy is presented on the screen is central to NLS. To specify the presentation, NLS uses three independent *view-specification conditions*: level, truncation, and content. Level describes the maximum depth of the presentation, and truncation describes the maximum number of lines that will be displayed for each "statement" in the hierarchy. Content is a more powerful mechanism, which allows the user to specify that only statements whose content satisfies the content condition should be presented.

In structure-oriented editors, a simple "holophrasting" technique is frequently used to allow larger portions of a document to be displayed on the screen. A holophrast is a cover name for a substructure of a

[5]This might not always be true for transverse graph presentations. If such a presentation addresses objects outside the tree rooted by the origin of the presentation, the graph presentation may be affected in other ways as well.

document, and it was introduced by Hansen in the Emily editor in 1971 [39,40]. (Emily is described in some detail in section 2.3.1 of this thesis.) In a presentation, the holophrast is shown instead of the substructure itself. Holophrasting and similar concepts have been used in numerous systems [10,31,61,91]. There has also been done some work on *automatic holophrasting* [64,62,84], by which the system, and not the user, determines which construct to holophrast in order to obtain a given presentation goal. In R^n [42]—a programming environment for Fortran—a window used for program presentation is divided into two panes, one besides the other. In the right-hand side pane the entire Fortran program is presented, holophrasted to such a degree that it fits into the pane. In the left-hand side pane, an indicated part of the contents of the holophrasted pane is shown in full detail.

Several contemporary programming systems provide facilities for making abstract presentations. Let us first take at look at Pecan [75,76], a prototype system developed at Brown University. In Pecan it is possible to define so-called *views*. A "syntax-directed editor view," which uses traditional textual presentation of programs, constitute the primary way to present, create, and modify programs. A Nassi-Schneiderman flow chart view is also supported. Besides these, several so-called semantic views of the symbol table, the data type of variables, and the tree representation of expressions are provided. Thus, most of the views in Pecan are quite detailed images of static and dynamic aspects of a program and its execution. More overall program presentations, such as a "module-level abstraction view", are proposed in [76].

Abstract presentations have also been used in Gandalf (see section 2.3.3.) Textual presentation at various abstraction levels is described and exemplified in both [59] (Aloe) and in [29] (Loipe), which builds on top of Aloe. The Aloe syntax description separates the abstract grammar and the so-called unparse schemes (concrete grammars), and it therefore makes sense to define several unparse schemes. Unparse schemes can be used to define several useful abstract presentations, such as procedure structure presentations and procedure call cross references. In Gnome [15], which is a more recent Gandalf environment, a so-called *outline view* is briefly described.

In [9] it is described how abstract presentation techniques have been applied on a specific programming language, namely Beta [53]. A variety of relations among structures in Beta programs and Beta program exe-

cutions are defined. In this work it has been a major concern to define presentations that make use of special graphical symbols instead of textual means. Even the detailed aspects of a Beta program are presented by these alternative means.

If more elaborate document representations than tree structures are used, it may be realistic to come up with more elaborate presentations as well. Linton proposes in [56] to represent programs as relations in a relational database, and he sketches how to define alternative views of programs in such a framework. Horwitz describes in [44,45] a "symbiosis" of attribute grammars and relational database techniques, which builds on top of the work done by Reps and Teitelbaum [78,79]. In this work, a view is defined as a query to a relational database. The query is automatically re-executed for each change in the underlying document. Documents are represented as attributed abstract syntax trees that can be augmented with relations. These relations may contain information that is loosely associated with the document, or they may, in a more explicit way, represent information that in principle also could be derived from the attributed ASTs. This extension of the representational basis of the system makes it possible to present program anomalies and static semantic errors in a rather concise way, because the information explicitly is available in the data base at any time. For example, all unused definitions, and all procedure calls whose number of parameters do not match their corresponding procedure declarations can be presented. (This particular example is discussed in some detail in [45].) The price of this functionality is a considerably higher complexity, because the system must deal with both attributed ASTs and relational representations.

7.6 Summary

We have in this chapter developed two abstract presentation techniques: One by which the compositional structure w.r.t. selected object types can be presented, and another that allows more transverse relationships among selected object types to be presented. Both of the techniques we find most interesting if the presented objects play a role in the overall structuring of the document. We look at our work as a contribution to solving the urgent need for better overall presentations of large documents.

The abstract presentation techniques have been embedded into simple presentation formalisms (or languages.) It means that the abstract presentation techniques are not bound to one or a few "applications" or languages. In the same way as it is possible to make textual presentation schemes (or pretty printers) for every language, it is also possible to make graph style presentations in our framework whenever there exist objects and relationships in the languages that are adequate for this kind of presentation.

We have observed that edit operations in general depend on the presentations through which they are applied. It must be required that the edit operations manipulate the internal AST structure in such a way that the net effect makes sense in the presentation. For the transverse graph presentation this sometimes means that two or more (possibly distant) places in the AST have to be modified in order for the operation to "act naturally" on the presentation. The problems caused by overlapping presentations were finally touched upon. We described how the identity of a presentation can be affected by operations on other presentations. To ensure a more satisfactory interaction we also introduced various degrees of consistency between the internal representation and the screen presentations.

Chapter 8

Conclusions

The conclusions will be organized with respect to the following key areas that have been treated in this thesis: Hierarchical grammars, language development facilities, syntax-directed editing, and semi-automatic facilities. The status of the work together with possible future research topics in the area will also be touched on.

Hierarchical Grammars

A hierarchical grammar represents a new formalism for the definition of the context free syntax of artificial languages. From a conceptual point of view, a hierarchical grammar is a specialization/generalization hierarchy of syntactic domains. We are not aware of other works in which this point of view has been used on grammars. We deal with multiple grammars by linking them all together in a single hierarchy, and by allowing subphylum relations that cross the boundaries between the phylum hierarchies. The treatment of the most general syntactic domain in a single language and in the whole environment (the phylum *Anything*) is particularly nice in our model. The idea of having "general applicable constructs" (the phylum *Always*) also turned out to be very useful.

From a pragmatic point of view there are only minor differences between the new formalism and the already well-known formalisms. We have seen that it is straightforward to interpret the hierarchical grammar concepts as either nonterminals/productions or operators/phyla. The special phyla in the hierarchical grammar framework—*Anything*, *Always*, gate phyla, identification phyla, and choice phyla—can therefore, more or less elegantly, be provided for in the already well-known grammar definition formalisms.

Language Development Facilities

The purpose of Muir is to support the development of new artificial languages. In other words, Muir is a language development environment. This is a new and indeed a very specialized area, and as such it is difficult to put forward definitive conclusions. However, we firmly believe that syntax-directed editing facilities are indispensable in a language development environment. The uniform treatment of grammars and documents, together with the possibility of shifting between grammar editors and document editors at any time, has clearly promoted a good deal of flexibility in Muir. The vision of editing a grammar through its phylum hierarchy has not yet been tried out in practice. However, we have in chapter 7 devised a general presentation technique that could be the basis for a realization of this vision.

Experimentation is a keyword in a language development environment. We envision that it typically will be necessary to modify a grammar, based on some experience gained by creating some sample documents in the new language. In order to reestablish the consistency between the grammar and its dependent documents, we find it important that the existing documents can be updated in a systematic way. The semi-automatic approach devised in chapter 5, which is based on transformation templates created by the system upon grammar modifications, is novel, and we are not aware of similar facilities in other systems. In DOSE [30], a "prototyping environment for language design", this issue has been ignored.

Language analysis tools are clearly relevant in a language development environment. This is an issue that nearly has been ignored in Muir. Also (static) semantic support would be interesting, but we have not gone into that area either. The work on static semantic issues in language-based editors is a good starting point for this. The most natural "next step", though, is to gain some experience with Muir in a real language development situation. Work on that is currently (as of January 1987) going on at Stanford's CSLI, where the Aleph specification language [99] is being "implemented" in Muir.

Syntax-Directed Editing

Most, if not the entire functionality of Muir, is relevant for AST-based, syntax-directed editing environments. Of special interest is the abstract presentation capabilities and the implementation of edit operations as pattern-based transformations.

It can be concluded that abstract presentations in the same way as more concrete presentations can be defined in grammar related presentation rules. In order to gain some experience with the definition of abstract presentations we have defined two simple presentation formalisms; one for showing the overall compositional structure of a document, and another for showing transverse relationships in graph presentations. An AST-based environment is well-suited as a basis for making abstract presentations. It would be very difficult, if not impossible, to support similar capabilities in environments based on a textual representation of documents. Abstract presentations might be one of the best arguments in favor of AST-based environments. We have proposed how general, structure-oriented edit operations can be defined on abstract presentations. However, we feel that more research is needed in that area.

Extensibility and customization are also relevant topics in syntax-directed editors. We find it important to provide a framework that makes it easy to implement new structure-oriented editing operations. The Muir transformation framework, defined in chapter 4, has proven to be well-suited in that respect. The use of inheritance in the phylum hierarchy is an elegant technique that allows us to define on which kinds of constructs a given edit operation is applicable. Finally, because grammars in a syntax-directed editing environment rarely are static, the facility for keeping documents consistent with their grammar is envisioned to be a useful tool in most syntax-directed editing environments.

Semi-Automatic Facilities

One of the major design decisions was to automate the simple routine tasks, and to solve the more intellectually demanding tasks in interaction with the user. The multi-formalism transformation facility in chapter 6 is probably the best example of a facility that has been based on this principle.

One of the most interesting aspects of the multi-formalism transformation facility is the semi-automatic generation of the transformations. Based on our experience with the translation from Pascal to Modula-2 and vice versa it is evident that the system-created transformation templates provide an excellent starting point for making a set of transformations. The price for this functionality is paid during the creation (and the debugging) of the relation among the source and the target phyla. However, it is possible to provide a convenient user interface to that part of the system, namely by letting the user select pairs of phylum symbols in windows that present the source grammar and the target grammar respectively.

The actual translation process is also semi-automatic. The set of transformations can be applied on a document, and the resulting mixed-formalism document can manually be converted to a "pure" target document. The manual translation process is carried out in a normal instance of Sedit, the syntax-directed editor in Muir. It is interesting to notice that only a couple of new structure-oriented edit operations are required for that task.

Finally, the facility that helps keep documents updated w.r.t. their grammar is also an example of a successful semi-automatic component of Muir. It is indeed the interaction between the system's and the user's contribution that makes our approach interesting.

All in all it can be concluded that an editing environment in general, and a syntax-directed editing environment in particular, is a good platform for the realization of many semi-automatic activities.

Appendix A

Proof of Correctness Theorem

In this appendix we prove the correctness theorem for multi-formalism transformations from section 6.3.4 on page 112. Let us rephrase the theorem here:

Theorem. *Correctness of a transformation*
Given a syntactically valid multi-formalism AST A and a transformation τ = (P,R,PR). Let C_P^1, ..., C_P^n denote the AST-constituents of P that are related to AST-constituents of the replacement R. If

1. $\varphi(R) \sqsubset^* \varphi(P)$ and
2. $\omega(C_P^i) \sqsubset^* \omega(PR(C_P^i))$ for $i \in [1..n]$

then $\tau(A)$ is a syntactically valid multi-formalism AST.

Proof.

Let us assume that the application of the transformation τ on the AST A locates m matches M_1, M_2, ..., M_m where $m \geq 0$. Depending of the transformation class of τ, new matches may be located in the replacements substituted for these matches. Each of the matches will be replaced by the replacement R, and in order for this substitution to be legal it must be the case that

(1) $\varphi(R) \sqsubset^* \omega(M_i)$

for $i \in [1..m]$.
Let us now consider one of the matches M_1, M_2, ..., M_m, and for notational simplicity, let us call it M. For each of the constituents C_P^1, ..., C_P^n in the

pattern there exit corresponding constituents in M, namely C^1_M, ..., C^n_M. The relation PR among the constituents of P and R induces a similar relation MR among the constituents of the match M and the replacement R. The match constituents C^1_M, ..., C^n_M are substituted for $MR(C^1_M)$, ..., $MR(C^n_M)$ respectively, and in order for that to be valid we must prove that

(2) $\varphi(C^i_M) \sqsubset^* \omega(MR(C^i_M))$

for $i \in [1..n]$.

If we can prove (1) and (2) for an arbitrary match M (and if the transformation step defined by τ terminates) then the theorem can easily be proved by induction. The basis for the induction is that the AST A, on which the transformation is applied, is a valid multi-formalism AST, and it follows directly from the pre-conditions of the theorem.

Ad (1):
Consider the following chain of assertions:

$$\varphi(R) \sqsubset^* \varphi(P)=\varphi(M) \sqsubset^* \omega(M).$$

$\varphi(R) \sqsubset^* \varphi(P)$ is a pre-condition of the theorem. $\varphi(P)=\varphi(M)$ stems from the fact that the pattern P matches M. And finally, $\varphi(M) \sqsubset^* \omega(M)$ is true because the AST, on which the transformation is applied, is syntactically valid.

Ad (2):
The following chain of assertions proves (2):

$$\varphi(C^i_M) \sqsubset^* \omega(C^i_M)=\omega(C^i_P) \sqsubset^* \omega(PR(C^i_P))=\omega(MR(C^i_M)).$$

The first link, $\varphi(C^i_M) \sqsubset^* \omega(C^i_M)$, is true because we assume that the match-constituent C^i_M is syntactically valid before the transformation is applied. $\omega(C^i_M) = \omega(C^i_P)$ holds due to the fact that the pattern constituent C^i_P matches the constituent C^i_M of the match. Third, the assertion $\omega(C^i_P) \sqsubset^* \omega(PR(C^i_P))$ is a pre-condition of the theorem, and finally, because the relation MR is induced by the relation PR, $\omega(PR(C^i_P)) = \omega(MR(C^i_M))$.

We have hereby proved the induction step, and this concludes the proof the correctness theorem.

Appendix B

Modula-2 Presentation Rules

In this appendix the phylum declarations that define the presentations in figure 7.4 (see page 147) and figure 7.5 (see page 149) are listed.

Compared with the actual declarations in the hierarchical Modula-2 grammar there have been made a couple cosmetic changes to the phylum declarations. First, in order to ease the understanding, only relevant presentation rules are included. Secondly, abstract constituents are referred to by name in the presentation rules, and not via a "constituent number."

```
Terminal Phylum declaration-list:
head : declaration
tail : declaration-list
presentation-rules
   MO.TreePs:
   Choice
         "(EQUAL (NAMEOFTREE (SUBTREE ROOT 1)) 'module-declaration)"
            (Constituent head control: Label= MO.LabelPs);
         "(EQUAL (NAMEOFTREE (SUBTREE ROOT 1)) 'procedure-declaration)"
            (Constituent head control: Label= MO.LabelPs)

   CONTINUE
```

Explanation:"A list of declarations"

```
Terminal Phylum module-declaration:
ModuleIdentifier : nameDcl
ModulePriority : OptionalPriority
ModuleImport : import-list
ModuleExport : OptionalExport
ModuleBlock : block
presentation-rules
   MO.LabelPs: "MODULE " ModuleIdentifier
```

Explanation: "Declaration of local module"

Terminal Phylum **procedure-declaration**:
ProcedureIdentifier : nameDcl
formalParameters : FormalParameters
ProcedureBlock : block
presentation-rules
  MO.LabelPs: "PROCEDURE " ProcedureIdentifier

Explanation: "A full declaration of a procedure"

Terminal Phylum **CompilationUnit-list**:
head : CompilationUnit
tail : CompilationUnit-list
presentation-rules
MO.ImportExportPs:
  [MO.LabelPs MO.ImportExportIdScheme MO.ImportExportEdgeScheme] CONTINUE
Explanation: "A list of compilation units (main module, implementation modules
  and definition modules.)"

Terminal Phylum **MainModule**:
ModuleName : nameDcl
Priority : OptionalPriority
Import : import-list
ModuleBlock : block
presentation-rules
  MO.LabelPs:
    "PROGRAM MODULE " ModuleName
  MO.ImportExportIdScheme:
    "MainModule." ModuleName
  MO.ImportExportEdgeScheme:
    Import
  MO.ImportExportPs:
    STOP STOP STOP STOP
Explanation: "A main program"

Terminal Phylum **ImplementationModule**:
ModuleName : nameDcl
Priority : OptionalPriority
Import : import-list
ModuleBlock : block
presentation-rules
  MO.LabelPs:
    "IMPL MODULE " ModuleName
  MO.ImportExportIdScheme:
    "ImplModule." ModuleName
  MO.ImportExportEdgeScheme:
    "DefModule." ModuleName SEP Import
  MO.ImportExportPs:
    STOP STOP STOP STOP

Explanation: "An implementation module."

Terminal Phylum **DefinitionModule**:
ModuleName : nameDcl
Import : import-list
Definitions : definition-list
presentation-rules
MO.LabelPs:
"DEF MODULE " ModuleName
MO.ImportExportIdScheme:
"DefModule." ModuleName
MO.ImportExportEdgeScheme:
Import
MO.ImportExportPs:
STOP STOP STOP

Explanation: "A definition module."

Terminal Phylum **import-list**:
head : import
tail : import-list
presentation-rules
MO.ImportExportEdgeScheme: head tail

Explanation: "A list of import sections, in each of which it is possible to import a list of named definitions"

Terminal Phylum **import**:
FromClause : FromClause
ImportNameList : nameApl-list
presentation-rules
MO.ImportExportEdgeScheme: FromClause

Explanation: "Specify a list of imported names. If the optional module name is specified, all names are assumed to be imported from this module. In the module, they can be used as if they had been exported in non-qualified mode."

Terminal Phylum **FromModule**:
ImportModule : nameApl
presentation-rules
MO.ImportExportEdgeScheme: "DefModule." ImportModule SEP

Explanation: "Qualifying import list with module name"

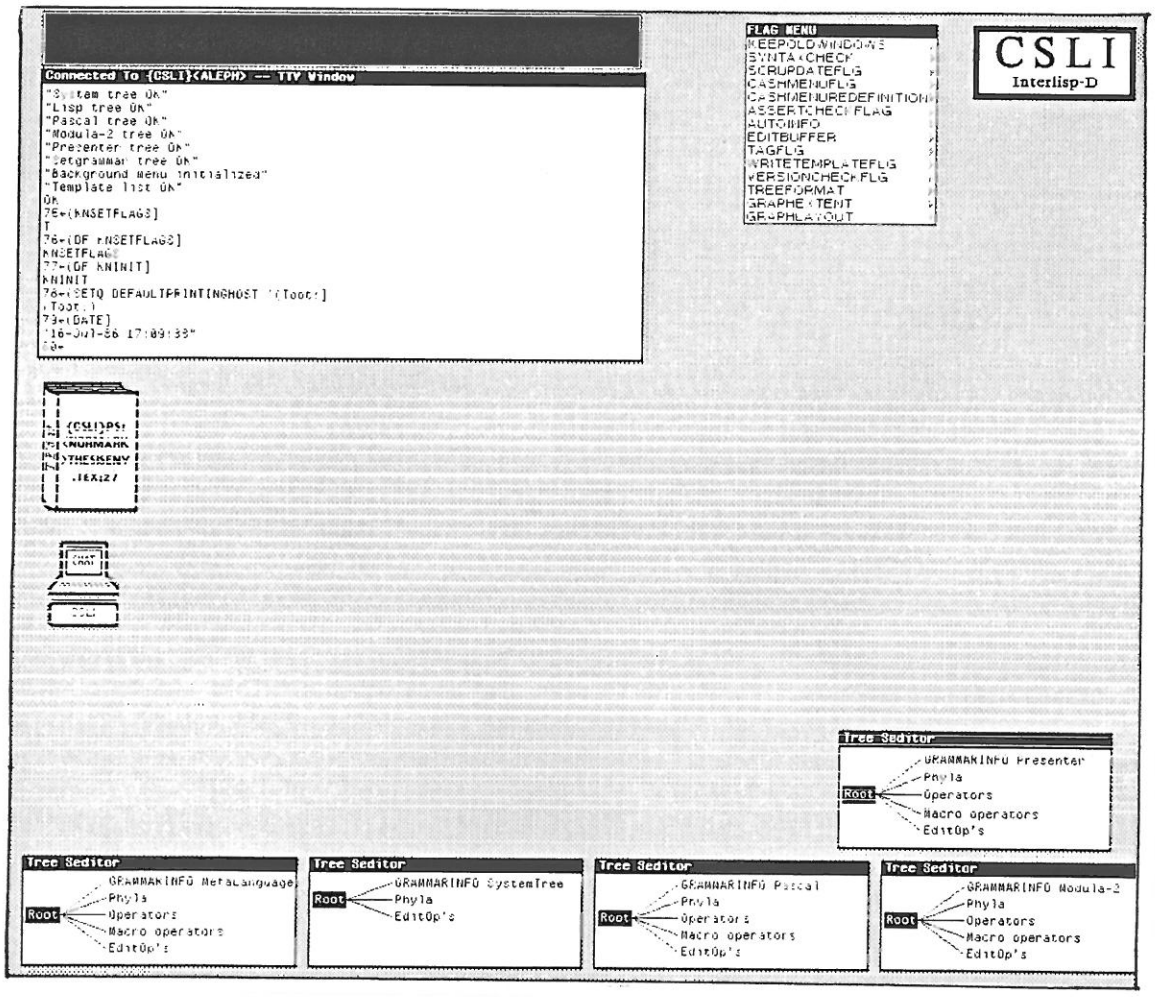
Appendix C

Guided Tour in Muir Woods

This appendix takes the reader on a guided tour through the Muir environment. The appendix is especially well-suited to deepen the understanding of section 3.2 about the Muir environment.

C.1 Start Situation

The first snapshot shows the initial Muir screen.



Snapshot 1. *Start situation. After having loaded the system this picture is shown on the screen.*

Muir is loaded on top of a standard Interlisp-D sysout. The Muir boot function sets up a display in which the most relevant grammars are presented in outline form. In snapshot 1, five such presentations are located at the bottom of the screen. Each of the editor windows are full-fledged syntax-directed editors that contain abstract presentations of hierarchical grammars (see chapter 3.)

We call the structure-oriented editor in Muir "Sedit". Seditors support two different presentation styles, a *text style* and a *graph style*. The graph style presentations that we will meet on this tour all show the composition of programs, and they are therefore constrained to be tree structures. We will refer to such presentations as *tree style* presentations.

Because of the tiny amount of information in the five outline presentations in snapshot 1, we mainly use them to initiate actions on grammars, and to initiate more detailed presentations of various aspects of a grammar. How this is done will be clear in a moment. Besides the outline presentation of the grammars, also the so-called prompt window (the black window at the upper left-hand side corner of the screen), the standard TTY window (below the prompt window), and a couple of icons are shown. The TTY window is the window from which arbitrary Lisp expressions can be evaluated. Occasionally it is useful to initiate an operation related to Muir via the TTY window. The prompt window is mainly used for messages from the system. Finally, a flag menu is present (placed to the left of the CSLI icon.) Through this menu it is possible to change various flags (variables) on which Muir depends.

C.2 How to Initiate Actions

A construct in an Sedit window can be selected by buttoning it with the left mouse button. The currently selected construct in a presentation is called the *current focus*, and it is shown in reverse. The primary way to start an action in Muir is through popup menus. A popup menu is related to the current focus,

and it reflects the various actions that can be executed on the focus. The popup menu appears when the middle mouse button is activated in the window that contains the focus.

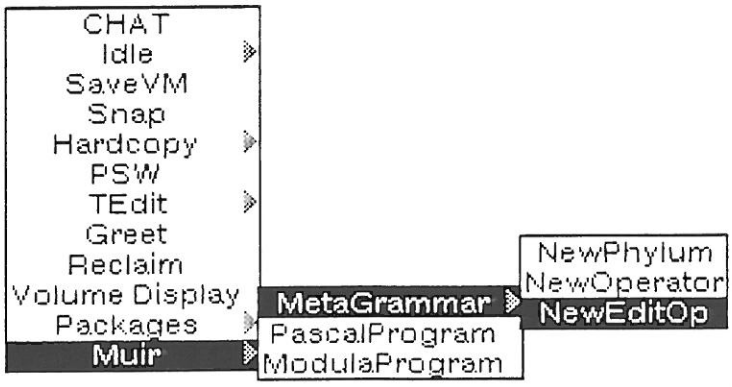
Given the situation in snapshot 1, we want to carry out an action on the Pascal grammar. By middle buttoning the root in the Pascal grammar (the third window from the left at the bottom of the screen) we get the menu shown below.

Snapshot 2. *A popup menu related to the root of the Pascal grammar. The selected operation will start a Pascal Seditor, if initiated. The small arrows in the menu fields mark those fields that have submenus. We use submenus for specializations of operations. (Submenus are provided as a standard Interlisp-D facility.)*

In general, a popup menu contains several segments. Operations in the upper section are the primitive edit operations, which are well-known from most syntax-directed editors. Operations in the second menu section are "generalized edit operations", which are relevant on the current focus. The operations in the third section are focus-independent edit operations. These are always meaningful, and consequently this section is always in the popup menu. Section four contains composite templates, i.e., non-primitive constructs which it makes sense to substitute for the current focus. A fifth section may contain list manipulation operations, but no such operations are relevant on the root of the Pascal grammar. One or more of the menu segments may by empty. If that is the case, the segment numbering just described is different.

A few actions can also be initiated via the background menu (a popup menu which appears when the gray background is buttoned with the right button.) The next illustration shows a snapshot of the background menu when it is

rolled two levels out.



Snapshot 3. *The background menu with sub-menus. While pressing the mouse button the cursor is moved across the two gray right arrows to produce this situation. If the mouse button is released in this state, the NewEditOp operation will be executed. The remaining items in the menu (at the outer level) are not relevant to Muir.*

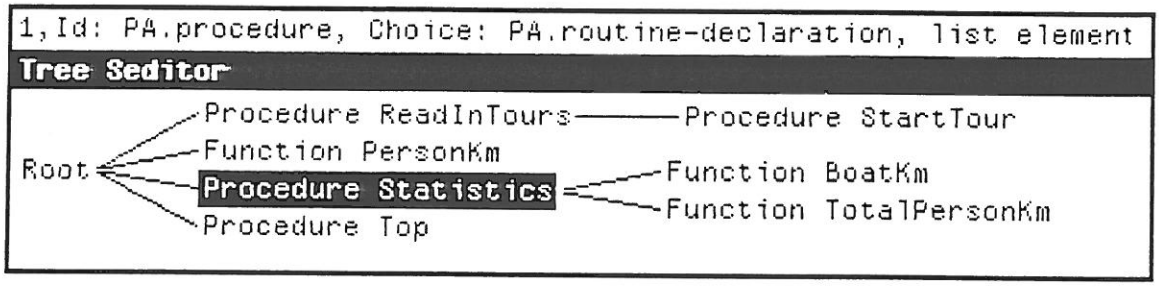
In the following we will denote the menu choice from above with *Muir▷MetaGrammar▷NewEditOp*. Whenever an operation can be activated via a popup menu it can also be activated via the keyboard by typing "Control x" followed by as many characters of the operation that makes it unique[1]. In addition it is, of course, possible to do anything from the TTY window, but it is usually the last resort if, for some reason, the other techniques fail.

C.3 Pascal editing

Most of the remaining discussion of the Muir environment will be centered around a Pascal example, because most people are familiar with Pascal as opposed to our grammar-related formalisms. Using the operation *Muir▷PascalProgram* from the background menu we start a tree style Pascal Seditor, and using the popup menu, we load an already existing Pascal

[1]Operations in sub-menus, however, cannot be activated via the keyboard yet, and in general we think it would be tedious to do so because also all the super-operation names must be specified.

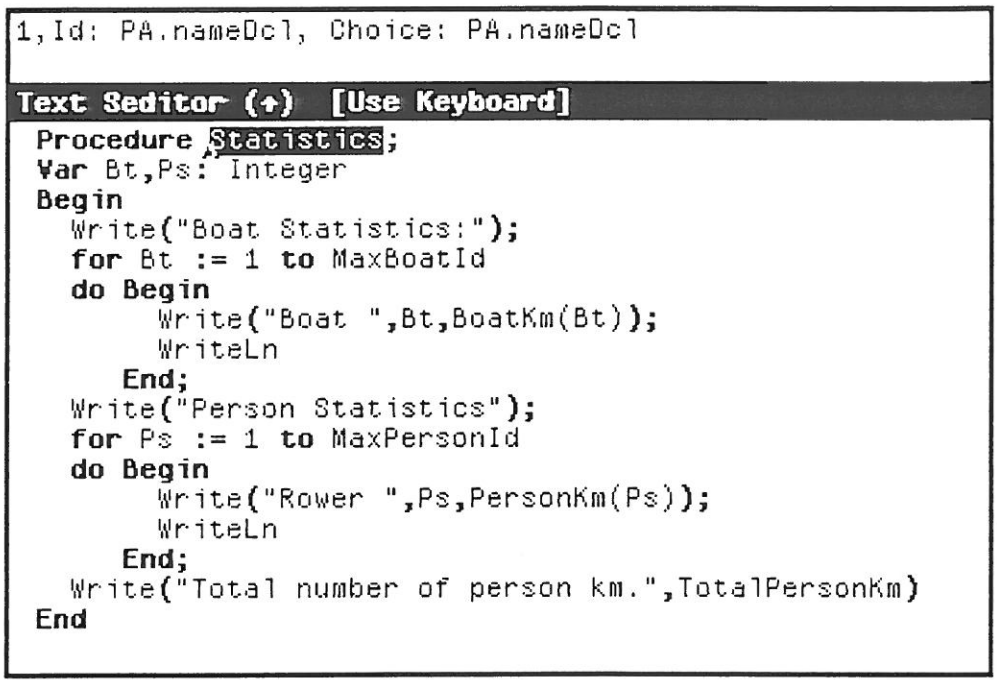
program by *Copy ▷FromFile*. The resulting display is shown below.



Snapshot 4. *A tree Seditor for a Pascal Program. Information about the current focus is shown in the local prompt window of the presentation (just above the title bar of the window.) It can be seen that the identification phylum of the focus is procedure in Pascal (PA), and that the choice phylum is routine-declaration.*

The presentation in this Seditor is an example of what we call *abstract presentation*. The underlying internal structure is an AST for the whole Pascal program, but only the relationship between procedures and their local procedures is shown in the presentation. Abstract presentation is discussed in detail in chapter 7 of this thesis.

Having selected the procedure *Statistics* we can re-present it in the usual textual way, just by using *Re-present* in the popup menu. (When *Re-present* is applied on a tree presentation it will attempt to re-present the internal structure as text, using the so-called default textual presentation scheme.) However, we wish to apply an alternative textual presentation scheme that filters out the (textual) presentation of local procedures. The fact that *Statistics* has local procedures is already apparent from the tree presentation, so why waste space by presenting the local procedures at the detailed level too? To enforce such a presentation we carry out the operation *Re-present ▷AsText ▷InOtherScheme*, which will present a new menu of alternative textual presentation schemes. We select "PA.TextPs1", a somewhat cryptic name for the desired scheme, and the following snapshot shows the resulting textual presentation of "Statistics."

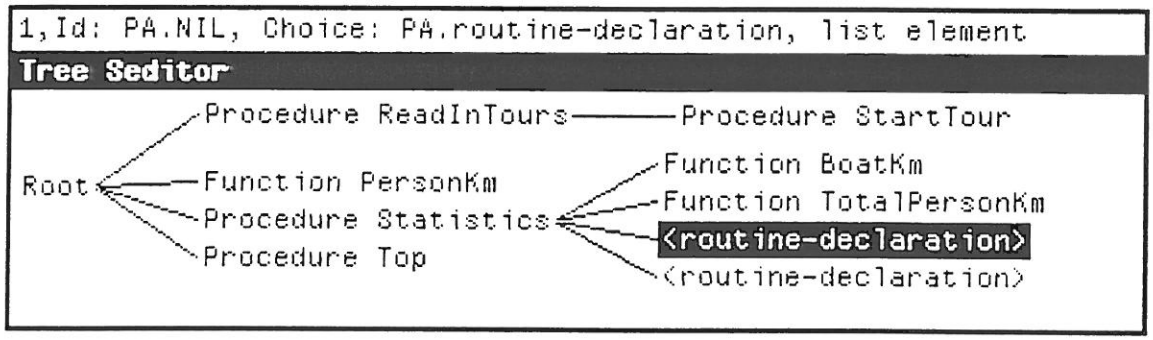


Snapshot 5. *Textual presentation of the procedure Statistics. The two local functions of Statistics are not presented in this window. The whole procedure structure of the program, of which Statistics is part, is shown in snapshot 4. The text* [Use Keyboard] *in the title bar of the window tells that the editor allows input from the keyboard to be substituted for the current focus. The symbol* (↑) *indicates that the document shown in this window is part of another document (in this case the document shown in snapshot 4.)*

The underlying internal representation of the procedure in snapshot 5 is a substructure of the internal representation of the tree presentation in snapshot 4. Thus, if the procedure name is modified through the textual presentation in snapshot 5 it will also be reflected in the tree presentation. Structure-oriented editing can be applied at both the textual level and at the tree level, and they work exactly in the same way[2]. Textual structure editing is relatively well-known, so let us illustrate the editing capabilities at the tree level. If we want to add a third and a fourth local procedure to Statistics, we select the function *TotalPersonKm* in snapshot 4, and we ap-

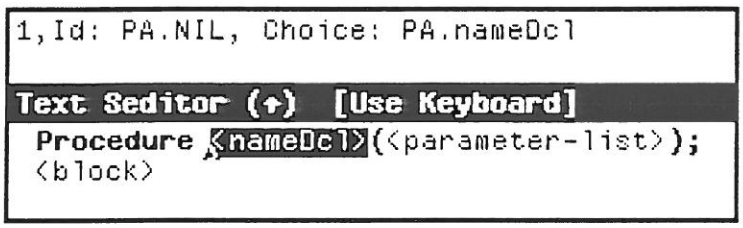
[2]In some sense, the edit operations should depend upon the presentation style. Some edit operations are meaningful in a textual presentation, whereas in a tree presentation, the effect of the operation cannot be observed.

ply "*AddElementAfter▷2*" on this focus. This results in the following tree presentation of the program:



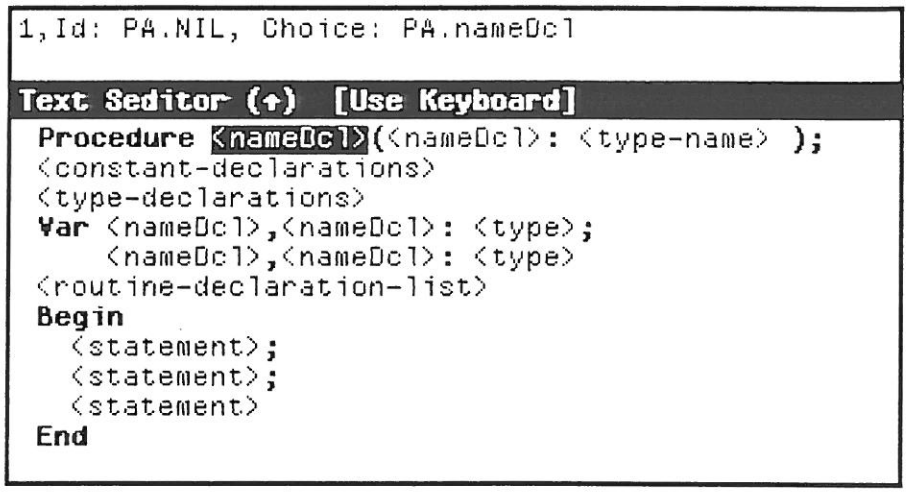
Snapshot 6. *Situation after the operation "AddElementAfter▷2" has been executed on TotalPersonKm in snapshot 4.*

Each of the unexpanded routine placeholders can now be expanded to procedures or functions. Let us assume that the user expands the first <routine-declaration> with the operator *procedure*. To further elaborate this procedure, the user must open a more detailed (textual) presentation, using *Represent*:



Snapshot 7. *A textual presentation of the selection in snapshot 6, after it has been expanded with the operator procedure.*

The parameter list and the block is now refined via structure-oriented editing, and the following template may be obtained:



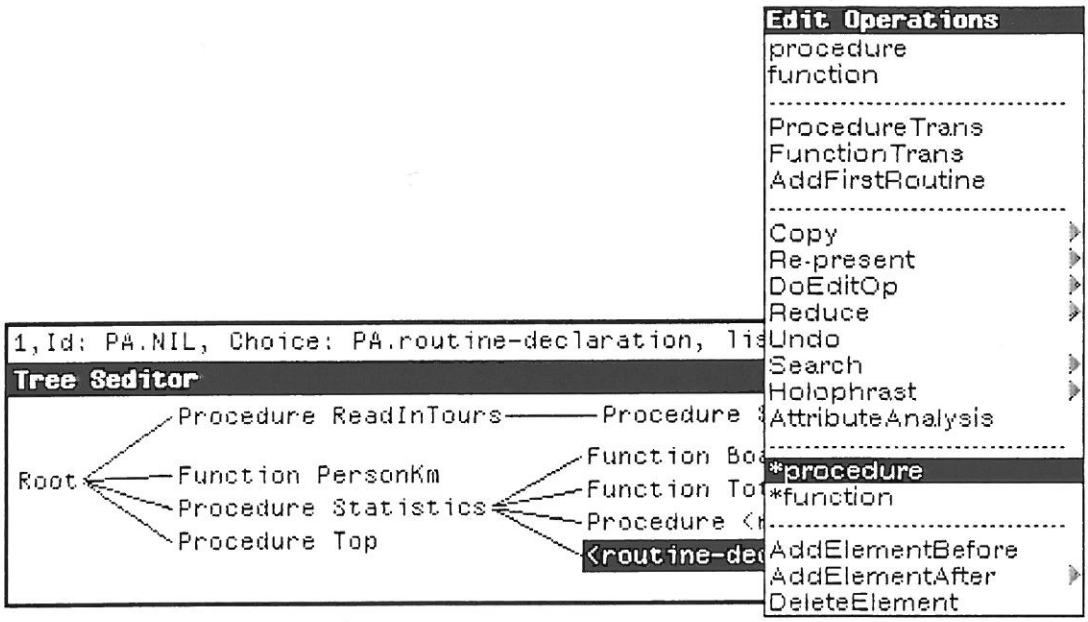
Snapshot 8. *Composite procedure template. The parameter list and the block in snapshot 7 have been refined.*

C.4 Definition of Composite Templates

Every time a procedure is defined, roughly the same refinements of the parameter list and the block are carried out. For procedure definitions, the refinements carried out on snapshot 7 (and shown in snapshot 8) are typical. In this, and similar situations, it is time-saving to define the resulting template once and for all, and make it available for subsequent use. Let us show how this can be done in Muir. We select the whole procedure in snapshot 8. To declare it as a composite template we execute the operation *Copy ▷ToTemplateStore* on it. This operation will prompt the user for a name of the template. We christen it **procedure*. In general, "stared" operator names hide such composite templates in Muir. For example in snapshot 2, **grammar* is a template for a new grammar. Internally, when *Copy ▷ToTemplateStore* is executed, the template is nested into a rather trivial transformation, which is appended to a special list of such transformations. Furthermore, this transformation is associated with the appropriate phyla, such that the edit operation implemented by

the transformation appears in the menu when it is applicable (see section 4.2.2 for more details on that.) Finally, the modified list of templates is written back onto a file, such that the new template is available in the future.

In the following, **procedure* is a valid alternative in the menu whenever a routine declaration is selected. If we, for example, button the unexpanded routine declaration in snapshot 6, the menu now reflects the new option **procedure* (together with a similar option for functions):

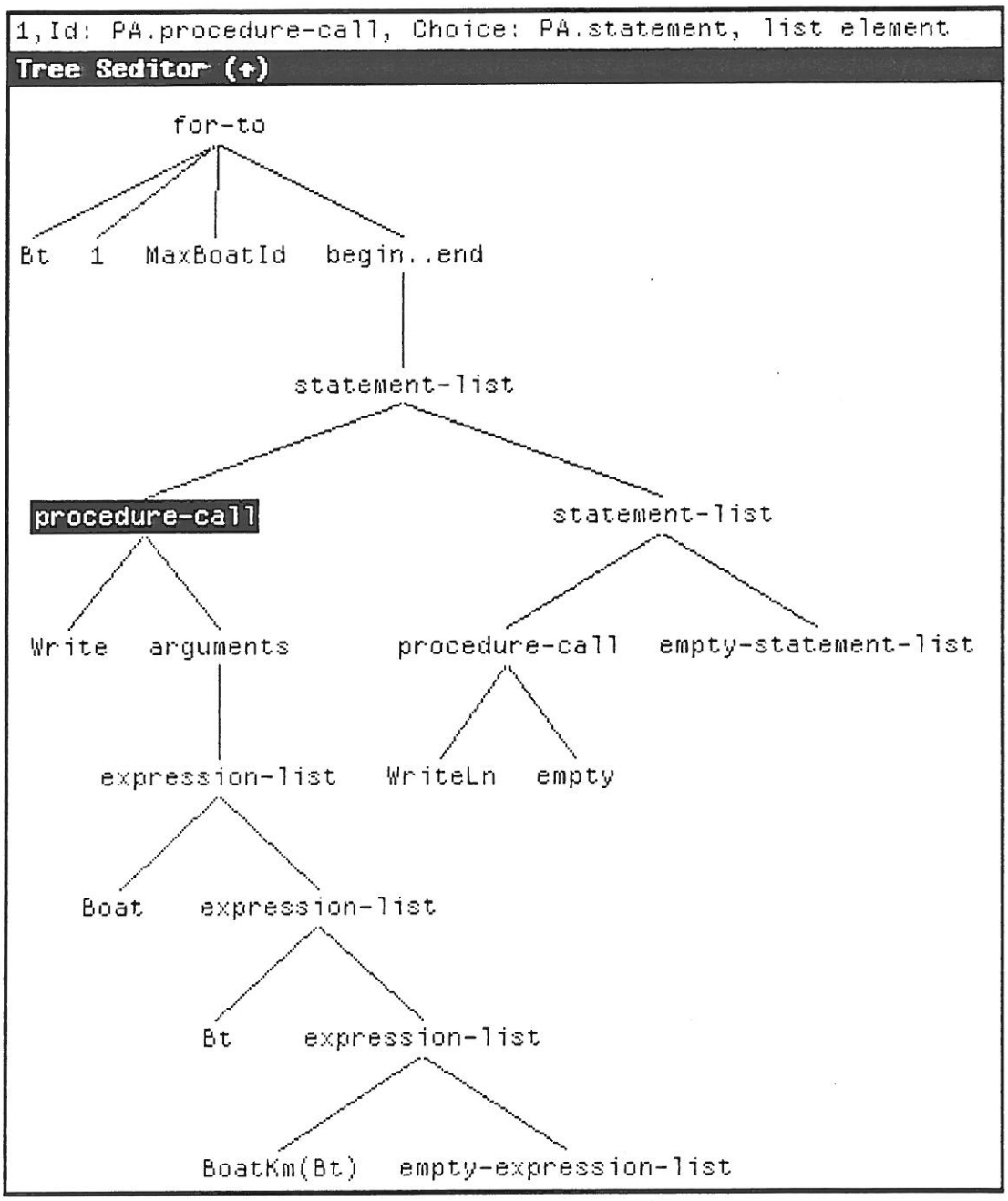


Snapshot 9. *Popup menu on a routine declaration in the Pascal program. Composite templates (*procedure and *function) have been defined for procedures and functions.*

C.5 Skeletal Presentation

Before we continue the discussion of the editing capabilities in Muir, we will show another presentation "feature." Snapshot 4, 6, and 9 showed high level presentations of a Pascal program. Sometimes it can be useful, or at least interesting, to be able present the same information at a very low level. The low level we have in mind is a presentation that resembles the internal AST representation. To illustrate that, we select

the first for-statement in *Statistics* (in snapshot 5). Then we do *Re-present* ▷*AsTree* ▷*InOtherScheme* and we select "tree-skeletal". This creates the following presentation:



Snapshot 10. *Low level tree presentation of a for-statement. This presentation reveals the internal AST representation that we use for Muir. It can be seen that lists internally are represented as binary nested trees. Furthermore, the grammar is not refined to handle the structure of expressions. "BoatKm(Bt)", for example, is not presented structurally. Notice that this presentation can be used for editing exactly as all the other presentations we have encountered.*

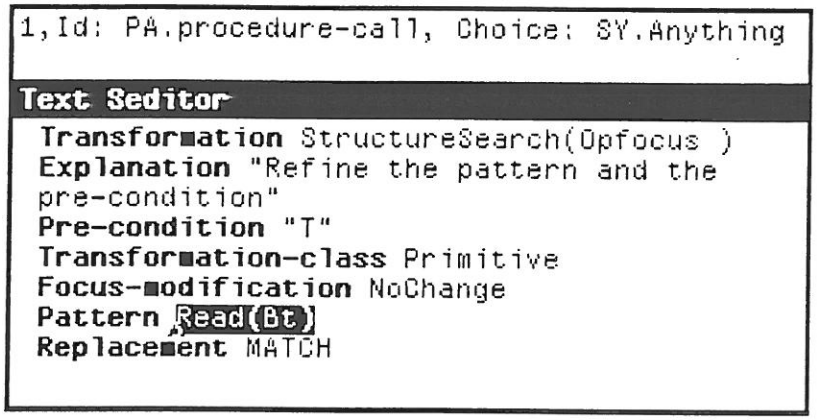
We support a similar textual skeletal presentation scheme where the tree structure is shown by indentation. These very

detailed presentations are rarely useful for editing, but they are pretty good for debugging and demonstration purposes. In addition, the skeletal presentations do not depend on explicitly defined presentation rules in the grammar. So in situations where no presentation rules are present for a document, we are still able to present it by using the skeletal presentation schemes. For a useful application of that, see chapter 5.

C.6 Systematic Modifications

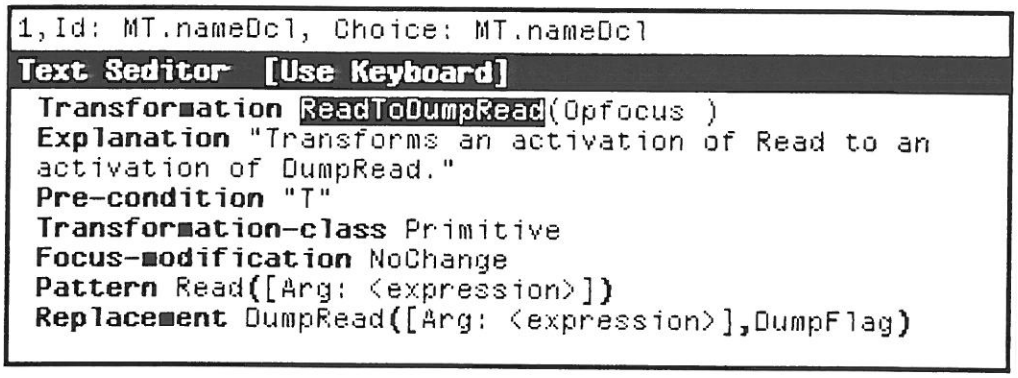
We will now demonstrate how various systematic modifications of a program can be carried out in Muir. The modifications we have in mind are so complicated that no single edit operation can be expected to do the job. On the other hand, the modifications are simple enough to be described in a procedure. This procedure may be totally automatic, or—as we shall see—it may involve a few manual steps. It may be a tedious task to carry out all the steps of the procedure manually, only using the edit operations provided by the system.

Let us assume that all the *Read* and *Write* statements in our Pascal program must dump their output on a file in addition to their "normal" behavior. We create two new procedures *DumpRead* and *DumpWrite*, and we want to substitute all activations of *Read* with *DumpRead*, and all activations of *Write* with *DumpWrite*. Let us first illustrate how we handle the Read case. We select an example of a *Read* statement in the program, and we activate the *Search* command on it. It will set up an Sedit window with a structure-oriented search operation for the given *Read* statement.



Snapshot 11. *Structure-oriented search operation. In the structure-oriented search operation, the replacement is the special "MATCH" indication. It tells that the replacement is the match itself. I.e., the document is not affected by this transformation.*

We now broaden the pattern by reducing the name *Bt* to the unexpanded placeholder <expression>, and we refine the replacement to a call of the procedure DumpRead:



Snapshot 12. *The structure-oriented search operation from above has been refined to a real transformation. The name Arg of the expressions in the pattern and replacement serves to copy the first argument of Read to the first argument of DumpRead.*

We now carry out the transformation on the program. Technically, this is done by selecting the whole program in the tree presentation (snapshot 4), and then activating the edit operation *DoEditOp* in the popup menu. *DoEditOp* will ask us to select a transformation somewhere on the screen. We

point at the newly created transformation, and after a few seconds Muir reports that a specific number of read statements have been transformed to DumpRead. If we happen to have some textual presentations of affected program fragments on the screen, they will all be updated, because Muir always attempts to keep the internal AST structure and their presentations consistent (see section 7.4.)

In a textual environment, the modifications described above could be done with a good text editor. However, global replacement operations are more error prone in a textual environment, because in such an environment it is difficult to separate different constructs that happen to be presented in the same way.

Write is handled in the same way, but we encounter an "unexpected" difficulty because we use *Write* with a variable number of arguments, and it is not possible to have user-created procedures with similar capabilities in Pascal. We decide to activate *DumpWrite* several times instead. In order to find out how many variable length write statements we have around in the program, we create the following search operation (again by selecting an example of a write, and by some manual structure editing on the pattern):

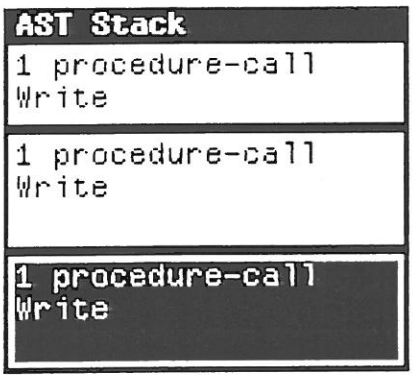
```
1,Id: MT.nameDcl, Choice: MT.nameDcl

Text Seditor  [Use Keyboard]
 Transformation StructureSearch(Opfocus )
 Explanation "Refine the pattern and the pre-condition"
 Pre-condition "T"
 Transformation-class Primitive
 Focus-modification NoChange
 Pattern Write(<expression>,<expression>,<expression-list>)
 Replacement MATCH
```

Snapshot 13. *Structure-oriented search operation. The two expression templates match any expression, and the expression-list template matches any expression list with zero, one or more elements.*

The constructs we are looking for happen to be so regular that we can capture them with the search operation above. For systematic modifications in general, this is not always

the case. The alternative is that we select the *Write* statements in the program manually, and that we put aside references to each of them while we go along. To do this, we call *Re-present* ▷*OnStack* on each of the write statements. Each activation of *Re-present* ▷*OnStack* pushes a reference to the Write-statement onto the so-called AST stack, which is located at the upper right-hand side corner of the screen. After having done that three times, the stack looks like this:



Snapshot 14. *The AST stack. Each stack frame shows the identification phylum of the construct, together with a name that describes it (in this case the procedure name.) In general, a stack frame contains a list of ASTs, and the number of ASTs in each frame is displayed. In this example, all frames are lists of only one element, because we have selected them one at a time in a program, and pushed references onto the stack. The top of the AST stack is selected. The AST stack is a "random access stack." It means that every stack frame can be selected and operated upon.*

We now want to apply a transformation on the stack that transforms a Write statement with two arguments to two activations of DumpWrite. This transformation looks like this:

```
1,Id: MT.nameDcl, Choice: MT.nameDcl

Text Seditor (↑)  [Use Keyboard]
 Transformation Write2(Opfocus )
 Explanation "Transforms a write statement with two
 arguments to a block with two DumpWrite activations."
 Pre-condition "T"
 Transformation-class Primitive
 Focus-modification NoChange
 Pattern Write([E1: <expression>],[E2: <expression>])
 Replacement Begin
               DumpWrite([E1: <expression>],DumpFlag);
               DumpWrite([E2: <expression>],DumpFlag)
             End
```

Snapshot 15. *A transformation that transforms a Write statements with two arguments to two calls of DumpWrite.*

To apply this transformation on the three Write-statements on the stack, we contract the stack to only one stack frame containing a list of the three statements:

```
AST Stack
3 procedure-call
```

Snapshot 16. *The contracted AST stack. The number '3' tells that the stack frame is a list of three constructs (namely the three selected Write-statements.)*

Next we push a copy of the transformation *Write2* from snapshot 15 onto the stack, by using *Copy ▷ToStack* on the whole transformation. This leaves the stack in the following state:

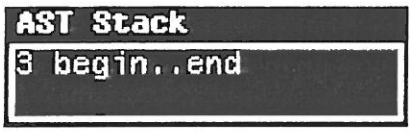
```
AST Stack
3 procedure-call

1 editOp
Write2
```

Snapshot 17. *The AST stack now has two frames. The first is the list of references to the Write statements. The top is a copy of the transformation from snapshot 15.*

A number of operations are available on the stack. They can be activated via a popup menu, which appears when the

stack is middle buttoned. We apply the stack operation *ApplyEditOp*, which requires that the top of the stack is a transformation, and the frame next to the top is the transformation focus. *ApplyEditOp* pops the stack twice, and it pushes the result of applying *Write2* onto the stack. This is a list of three *begin..end* constructs:



Snapshot 18. *The AST stack after the transformation Write2 has been executed on the three write statements. In the same way as the transformation focus was a list of references into the Pascal program, also the current stack top is a list of references into the program.*

We can apply a similar transformation for write statements with three arguments, but we will not show these steps here. It is easy to modify the transformation in snapshot 15 to deal with three argument, but using the current transformation framework, it is harder to formulate a general solution for n arguments ($n \geq 1$.) If we anticipate that similar systematic modifications are needed in the future, we can store the involved transformations from snapshot 12 and snapshot 15 on a file.

C.7 From Program to Grammar

It is easy in Muir to look at a grammar element (a phylum declaration or an operator declaration), which defines a given construct in a document. Let us, for example, assume that we want to look at the Pascal operator declaration that defines the *for*-construct in snapshot 5. We select an example of a *for*-construct and do *Re-present* ▷*Operator*. This sets up a new Sedit window on the *for* operator declaration in the Pascal grammar:

```
1,Id: MT.nameDcl, Choice: MT.nameDcl
Text Seditor (+)  [Use Keyboard]
 Operator for-to:
 for-variable : <nameApl>
 init-expression : <expression>
 terminal-expression : <expression>
 repeatedStatement : <statement>


 presentation-rules
 PA.TextPs:
   KeyWord("for "), for-variable, KeyWord(" := "),
 init-expression, KeyWord(" to "), terminal-expression, CR,
 KeyWord("do "), repeatedStatement
 PA.CallTreeEdgeScheme:
   4

 Explanation: "For to"
 Equations: environment = father.environment
            Defaultequation = <equation-expression>

            check = ( ( for-variable.used
                      U ( init-expression.used
                          U terminal-expression.used ) )
                    \ myself.environment )
            Defaultequation = <equation-expression>
```

Snapshot 19. *Textual presentation of an operator in a grammar. The first five lines of the operator describe the abstract syntax: The phylum of each constituent (such as <nameApl>), and their "internal name" (such as for-variable.) The internal constituent names are used to refer to abstract constituents in the presentation rules. Two presentation rules are given, namely PA.TextPs and PA.CallTreeEdgeScheme. The first of these describes the "standard" textual presentation of a for-statement. The explanation is a helping text that will appear when the user is about to use a for-construct during structure-oriented editing. The last few lines are attribute equations.*

The identification phylum and the choice phylum of the construct can be re-presented in the same way. Because the hierarchical grammar for Pascal is supported by Muir, we can modify the grammar in the same way as we do structure-oriented editing on a Pascal program. Moreover, grammar modifications can be incrementally compiled (installed), and changes are reflected immediately by the system. As an example of this, we modify the textual presentation scheme of the *for*-operator to

Keyword("Do "), repeatedStatement,
Keyword(" for "), for-variable, Keyword(" from "), init-expression,
Keyword(" to "), terminal-expression.

If we now re-present the Pascal procedure Statistics we get the following result (compare with snapshot 5):

```
1,Id: PA.for-to, Choice: PA.statement, list element

Text Seditor (↑)
Procedure Statistics;
Var Bt,Ps: Integer
Begin
  Write("Boat Statistics:");
  Do Begin
       Write("Boat ",Bt,BoatKm(Bt));
       WriteLn
     End for Bt from 1 to MaxBoatId;
  Write("Person Statistics:");
  Do Begin
       Write("Rower ",Ps,PersonKm(Ps));
       WriteLn
     End for Ps from 1 to MaxPersonId;
  Begin
    DumpWrite("Total number of person km.",DumpFlag);
    DumpWrite(TotalPersonKm,DumpFlag)
  End
End
```

Snapshot 20. *The procedure Statistics presented with the new presentation rules for for-statements. The current focus is a single for-statement.*

All Pascal programs we present in the future (using the presentation scheme PA.TextPs[3]) will show *for*-statements in this way (so we prefer to change the presentation rule back to its original definition.) We could also modify the abstract syntax of the *for*-construct, but unless we update all existing Pascal programs to conform with the new abstract syntax, we will get difficulties. The problem is that existing documents

[3]The presentation rule PA.TextPs is the standard textual presentation scheme for Pascal. We have earlier used the alternative textual presentation scheme PA.TextPs1, which doesn't present local procedure and function declarations. However, this presentation scheme is only defined on the relevant constructs (blocks), and one common textual presentation rules (PA.TextPs) is used to present other constructs in the language, such as *for*-statements.

become inconsistent w.r.t. the current version of the grammar. We have dealt with that problem in chapter 5 in the thesis.

Let us go a step further. The Pascal grammar is a document, which is defined by a grammar for grammars. We often call this grammar for the meta grammar. Therefore we can ask the system to re-present the operator declaration of the *for* operator. We select the operator shown in snapshot 19, and we do *Re-present* ▷*Operator*. This sets up the following Sedit window:

```
1,Id: MT.nameDcl, Choice: MT.nameDcl
Text Seditor (+)  [Use Keyboard]
 Operator operator:
 comment : <comment>
 operator-name : <nameDcl>
 abstract-syntax : <symbol-list>
 command-character : <command-xtr>
 std-properties : <property-list>
 presentations : <presentation-list>
 explanation : <explanation>
 equations : SetGrammar.<equation-list>


 presentation-rules
 LabelScheme:
   operator-name
 TreeScheme:
   STOP, STOP, STOP, STOP, STOP, STOP, STOP
 PP1:
   Text "Operator " ((BOLD REGULAR REGULAR )), operator-name,
 ":", CR, abstract-syntax, CR, std-properties, CR,
   Text "presentation-rules" ((BOLD REGULAR REGULAR )), CR,
 presentations, CR,
   Text "Explanation: " ((BOLD REGULAR REGULAR )),
 explanation, CR,
   Text "Equations: " ((BOLD REGULAR REGULAR )), equations
 PSHORT:
   2, ":", 3

 Explanation: "This is an explanation of operator"
```

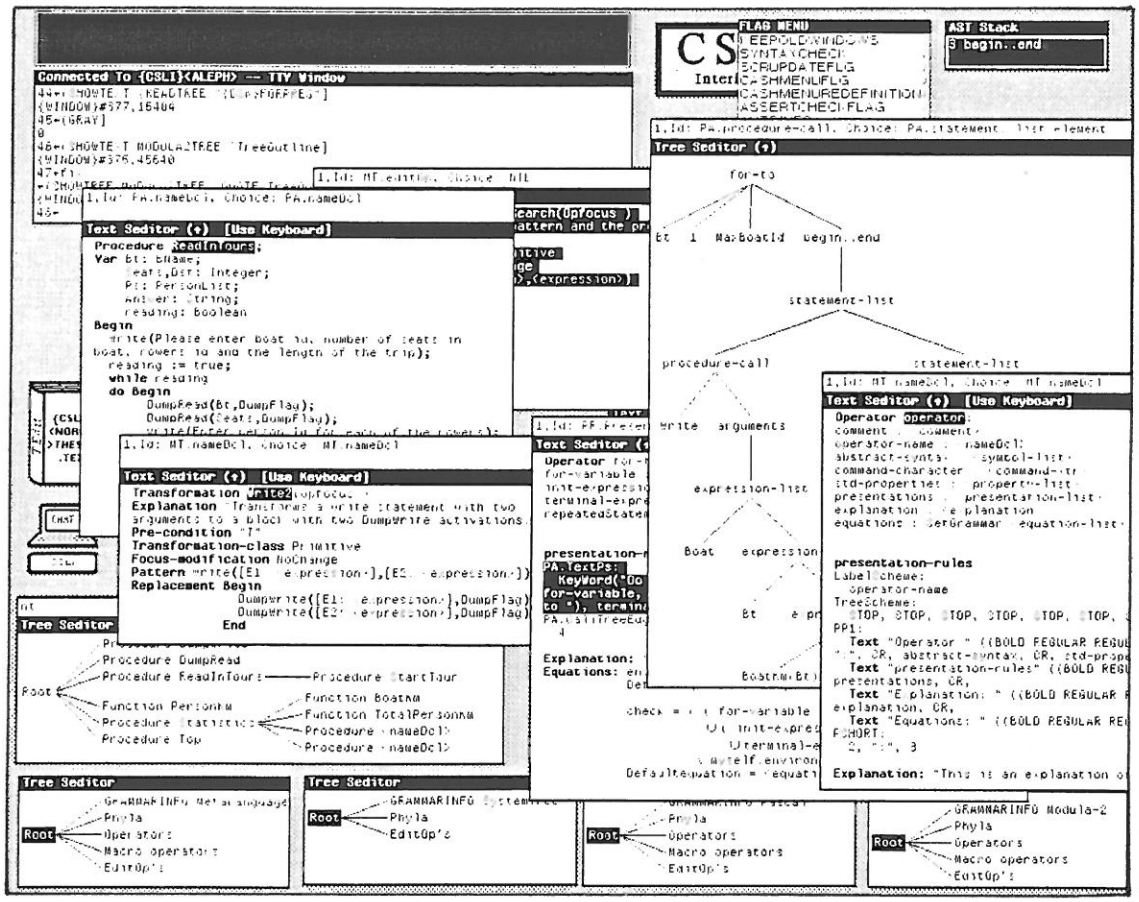
Snapshot 21. *The operator declaration for operators in the meta grammar. The first section lists the abstract constituents of an operator. Below that, the presentation rules LabelScheme, TreeScheme, PP1, and PSHORT are defined. As can be seen, the presentation of the presentation rules is pretty ugly.*

Again we could redefine one of the presentation rules (or the abstract grammar) if we wanted too. The operator dec-

laration for operators defines its own syntax, and the presentation scheme *PP1* in operator *operator* defines its own presentation.[4] So if we repeat *Re-present* ▷*Operator* on this operator, we get an identical presentation to that in snapshot 21.

C.8 The Final Situation.

This completes the short travel through the wood of Muir trees, grammars, and presentations. Let us finally show a picture of the whole screen after these operations.



Snapshot 22. *The final situation. As snapshot 1, this snapshots shows the whole Muir screen. Some of the presentations that have been explained during the tour still remain on the screen.*

We have naturally closed some of the windows that we

[4]The equation-part is not included in the visible part of the window.

have worked with, but quite a few are still around. The picture above is a typical picture of a Muir screen. Instead of working on one large document belonging to a single language we have (partly overlapping) presentations of several documents from several languages. Typically, only small subdocuments are presented in full detail.

Bibliography

[1] Aho, A.V. and Ullman, J.D., *The Theory of Parsing, Translation, and Compiling*, volume 1: Parsing, Prentice-Hall 1972.

[2] Allison, L., "Syntax Directed Program Editing", *Software—Practice and Experience*, vol. 13, 1983, pp. 453-465.

[3] Ambriola, V. and Staudt, B.J., *The ALOE Action Routine Language Manual* (draft), Carnegie-Mellon University, November 1985.

[4] Bahlke, R. and Snelting, G., "The PSG — Programming System Generator", Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, *Sigplan Notices*, vol. 20, no. 7, July 1985, pp. 28-33.

[5] Balzer, R. and Cheatham, T.E. Jr., Editorial: Program Transformations, *IEEE Transactions on Software Engineering*, vol. SE-7, no. 1, January 1981, pp. 1-2 (Introduction to a special section on program transformations.)

[6] Barr, A., Cohen, P.R., and Feigenbaum, E.A., *The Handbook of Artificial Intelligence*, vol. 1-3, William Kaufmann, Inc., 1981.

[7] Barstow, D.R., Shrobe H.E., and Sandewall E. (editors) *Interactive Programming Environments*, McGraw-Hill, 1984.

[8] Barstow, D.R., "A Display-Oriented Editor for INTERLISP" in Barstow, D.R., Shrobe, H.E., and Sandewall, E. (editors) *Interactive Programming Environments*, McGraw-Hill, 1984, pp. 288-299.

[9] Berthelsen, S., Hvidbjerg, S., and Sørensen, P., *Graphical Programming Environments Applied to Beta*, DAIMI IR-64, Department of Computer Science, Aarhus University, Denmark, September 1986.

[10] Borup, K., Nørmark, K., and Sandvad, E., *EKKO—An Integrated Program Development System*, DAIMI IR-51, Department of Computer Science, Aarhus University, Denmark, November 1983.

[11] Boyle, J.M., "Lisp to Fortran—Program Transformation Applied", *Program Transformation and Programming Environments*, Nato ASI Series, Series F: Computer and System Sciences, Vol. 8, 1984, pp. 291-298.

[12] Burkhart, H. and Nievergelt, J., *Structure-Oriented Editors*, ETH Zürich, May 1980.

[13] Bødker, S. and Knudsen, J.L., *TREED, En Interaktiv Syntax-styret Editor*, Department of Computer Science, Aarhus University, 1980 (not published.)

[14] Cameron, R.D. and Ito, M.R., "Grammar-Based Definition of Metaprogramming Systems", *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, January 1984, pp. 20-54.

[15] Chandhok, R., Garlan, D., Goldenson, D., Miller P., and Tucker M., "Programming environments based on structure editing: the GNOME approach", *Proc. AFIPS National Computer Conference*, vol. 54, 1985, pp. 359-369.

[16] Cheatham, T. Jr., Holloway, G.H., and Townley, J.A., *Program Refinement by Transformation*, Center for Research in Computing Technology, Aiken Computation Laboratory, Harward University, TR-10-80, June 1980.

[17] Clark, R. and Koehler, S., *The UCSD Pascal Handbook, A Reference and Guidebook for Programmers*, Englewood Cliffs, N.J., Prentice-Hall, Inc., 1982.

[18] Dahl, O.-J., Myhrhaug, B., and Nygaard, K., *Common Base Language*, Norwegian Computing Center, October 1970.

[19] Darlington, J. and Burstall, R.M., "A System which Automatically Improves Programs", *Acta Informatica*, vol. 6, 1976, pp. 41-60.

[20] Delisle, N.M., Menicosy, D.E., and Schwartz, M.D., "Viewing a Programming Environment as a Single Tool", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 49-56.

[21] Delisle, N.M., Menicosy, D.E., and Schwartz, M.D., "Magpie—An Interactive Programming Environment for Pascal", *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, 1985, vol. II, pp. 588-595.

[22] Dixon, M., *An Extensible Structured Data Editor for Interlisp-D*, (not published.)

[23] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., *Programming Environments based on Structured Editors: The Mentor Experience*, Inria, Rapport de Recherche, no. 26, July 1980.

[24] Donzeau-Gouge, V., Kahn, G., Lang, B., and Mélèse, B., "Document Structure and Modularity in Mentor", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 141-148.

[25] Ellison, R.J. and Staudt, B.J., "The Evolution of the Gandalf System", *The Journal of Systems and Software*, vol. 5, no. 2, May 1985, pp. 107-119.

[26] Engelbart, D.C. and English, W.K., "A research center for augmenting human intellect", *AFIPS Conference Proceedings*, Volume 33, Part one, 1968, Fall Joint Computer Conference, December 1968, San Francisco, pp. 395-410.

[27] Eriksen, S.H., Jensen, B.B., Kristensen, B.B., and Madsen, O.L., *The BOBS-System*, DAIMI PB-71, Computer Science Department, Aarhus University, Denmark, February 1982.

[28] Feather, M.S., "Specification and Transformation: Automated Implementation", *Program Transformation and Programming Environments*, Nato ASI Series, Series F: Computer and System Sciences, Vol. 8, 1984, pp. 223-230.

[29] Feiler, P.H., *A Language-Oriented Interactive Programming Environment Based on Compilation Technology*, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May 1982.

[30] Feiler, P.H., Fahimeh, J., and Schlichter, J.H., "An Interactive Prototyping Environment for Language Design", *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*, 1986, pp. 106-116.

[31] Fischer, C.N., Johnson, G.F., Mauney, J., Pal, A., and Stock, D.L., "The Poe Language-Based Editor Project", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 21-29.

[32] Fraser, C.W., "A Generalized Text Editor", *Communication of the ACM*, vol. 23, no. 3, March 1980, pp. 154-158.

[33] Fraser, C.W., "Syntax-Directed Editing of General Data Structures", Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation, *Sigplan Notices*, vol. 16, no. 6, June 1981, pp. 17-21.

[34] Goldberg, A. and Robson, D., *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Publishing Company, 1983.

[35] Goldberg, A., *Smalltalk-80 The Interactive Programming Environment*, Addison-Wesley Publishing Company, 1984.

[36] Gouguen, J.A., Thatcher, J.W., and Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in R.T. Yeh (editor) *Current Trends in Programming Methodology*, Vol. IV, Data Structuring, Prentice Hall, 1978, pp. 80-149.

[37] Greenblatt, R.D., Knight, T.F Jr., Holloway J., Moon, D.A., and Weinreb, D.L., "The LISP Machine" in Barstow, D.R., Shrobe, H.E., and Sandewall, E. (editors) *Interactive Programming Environments*, McGraw-Hill, 1984, pp. 326-352.

[38] Griswold, R.E., Poage, J.F., and Polonsky, I.P., *The Snobol4 Programming Language*, second edition, Prentice Hall, Inc., 1971.

[39] Hansen, W.J., *Creation of Hierarchic Text with a Computer Display*, Ph.D. dissertation, Stanford University, May 1971.

[40] Hansen, W.J., "User Engineering Principles for Interactive Systems", *Fall Joint Computer Conference*, 1971, pp. 523-532.

[41] Hershey, W.R., "Thinktank, an Outlining and Organizing Tool", *BYTE*, May 1984, pp. 189-193.

[42] Hood, R.T. and Kennedy, K., "A Programming Environment for Fortran", *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, 1985, vol. II, pp. 625-637.

[43] Horgan, J.R. and Moore, D.J., "Techniques for Improving Language-Based Editors", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 7-14.

[44] Horwitz, S.B. and Teitelbaum, T., "Relations and Attributes: A Symbiotic Basis for Editing Environments", Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, *Sigplan Notices*, vol. 20, no. 7, July 1985, pp. 93-106.

[45] Horwitz, S.B., *Generating Language-Based Editors: A Relationally-Attributed Approach*, Ph.D. Thesis, Department of Computer Science, Cornell University, TR 85-696, August 1985.

[46] *Lisp Library Packages*, Xerox Artificial Intelligence Systems, January 1985.

[47] *Interlisp-D Reference Manual*, vol. 1: Language, Xerox Artificial Intelligence Systems, October 85.

[48] *Interlisp-D Reference Manual*, vol. 2: Environment, Xerox Artificial Intelligence Systems, October 85.

[49] Jensen, K. and Wirth, N., *PASCAL User Manual and Report*, Second Edition, Springer-Verlag 1975.

[50] Jesshope, C.R., Crawley, M.J., and Lovegrove, G.L., "An Intelligent Pascal Editor for a Graphical Oriented Workstation" *Software—Practice and Experience*, vol. 15, 1985, pp. 1103-1119.

[51] Kaiser, G.E. and Habermann, A.N., "An Environment for System Version Control", *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, May 1982.

[52] Kaiser, G.E. and Kant, E., "Incremental Parsing without a Parser", *The Journal of Systems and Software*, vol. 5, no. 2, May 1985, pp. 121-144.

[53] Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., and Nygaard, K., "Abstraction Mechanisms in the Beta Programming Language", *Proceedings of Tenth ACM Symposium on Principles of Programming Languages*, 1983, pp. 285-298.

[54] Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B., and Nygaard, K., "An Algebra for Program Fragments", Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, *Sigplan Notices*, vol. 20, no. 7, July 1985, pp. 161-170.

[55] Leblang, D.B., "Abstract Syntax Based Programming Environments", *Proceedings of the AdaTec Conference on Ada*, Arlington, Virginia, October 1982, pp. 187-200.

[56] Linton, M.A., "Implementing Relational Views of Programs", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 132-140.

[57] Madhavji, N.H., Leoutsarakos, N., and Voulioouris, D., "Software Construction Using Typed Fragments", *Lecture Notes in Computer Science*, no. 186, Springer-Verlag, 1985, pp. 163-178.

[58] Madsen, O.L., Note about extension of BNF grammars with alternation and construction-like productions, personal correspondence, 1984.

[59] Medina-Mora, R., *Syntax-Directed Editing: Towards Integrated Programming Environments*, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, March 1982.

[60] Medina-Mora, R. et al., *ALOE Users' and Implementors' Guide*, Department of Computer Science, Carnegie-Mellon University, November 1983.

[61] Meyer, B. and Nerson, J-M., *A Visual and Strutural Editor*, TRCS84-03, Department of Computer Science, University of California Santa Barbara, January 1985.

[62] Meyer, B., Nerson, J-M., and Ko, S.H., *Showing Programs on a Screen*, TRCS84-04, Department of Computer Science, University of California Santa Barbara, January 1985.

[63] Meyrowitz, N. and van Dam, A., "Interactive Editing Systems: Part I and II", *ACM Computing Surveys*, vol. 14, no. 3, September 1982, pp. 321-415.

[64] Mikelsons, M., "Prettyprinting in an Interactive Programming Environment", Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation, *Sigplan Notices*, vol. 16, no. 6, June 1981, pp. 108-116.

[65] Naur, P., (editor), "Report on the Algorithmic Language ALGOL 60", *Communication of the ACM*, vol. 3, no. 5, May 1960, pp. 299-314.

[66] Nelson, T.H., "Replacing the Printed Word: A Complete Literary System" in *Information Processing 80*, Proceedings of IFIP Congress 80, Lavington, S. (editor), North-Holland Publishing Company, 1980, pp. 1013-1023.

[67] Nielsen J., Nielsen, H.W., Nørmark, K., and Sørensen, J., *EAGLE - a Syntax-directed editor*, Users' guide, (In Danish) DAIMI MD-45, Department of Computer Science, Aarhus University, Denmark, January 1982.

[68] Notkin, D., "The Gandalf Project", *The Journal of Systems and Software*, vol. 5, no. 2, May 1985, pp. 91-105.

[69] Nørmark, K., "Program Development on Graphical Workstations", *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, 1985, vol. II, pp. 672-682.

[70] Nørmark, K., "An Overview of Muir—A Language Development Environment", Programming Environments—Programming Paradigms, *Proceedings of a Workshop at Roskilde University Centre*, October 1986, pp. 187-196.

[71] Oppen, D.C., "Prettyprinting", *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 4, October 1980, pp. 465-483.

[72] Partsch, H. and Steinbrüggen, R., "Program Transformation Systems", *ACM Computing Surveys*, vol. 15, no. 3, September 1983, pp. 199-236.

[73] Peyton, L., *Presenter—Presentation in a Language Design Environment*, Programming Project (not published), Department of Computer Science, Stanford University, July 1986.

[74] Peyton, L., *Presenter User's Guide*, Programming Project (not published), Department of Computer Science, Stanford University, July 1986.

[75] Reiss, S.P., "Graphical Program Development with PECAN Program Development Systems", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 30-37.

[76] Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 276-285.

[77] Reps, T.R., Teitelbaum, T., and Demers, A., "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, July 1983, pp. 449-477.

[78] Reps, T. and Teitelbaum, T., "The Synthesizer Generator", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 42-48.

[79] Reps, T., *Generating Language-Based Environments*, The MIT Press, Cambridge Mass., 1984.

[80] Reps, T. and Teitelbaum, T., *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, August 1985.

[81] Reps, T., Marceau, C., and Teitelbaum, T., "Remote Attribute Updating for Language-Based Editors", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, January 1986, pp. 1-13.

[82] Sandewall, E., "Programming in an Interactive Environment: the "Lisp" Experience", *ACM Computing Surveys*, vol. 10, no. 1, March 1978, pp. 35-71, also in Barstow, D.R., Shrobe, H.E., and Sandewall, E. (editors) *Interactive Programming Environments*, McGraw-Hill, 1984, pp. 31-80.

[83] *The Seybold Report on Word Processing*, vol. 1, no. 9, October 1978.

[84] Smith, S.R., Barnard, D.T., and Macleod, I.A., "Holophrasted Displays in an Interactive Environment", *International Journal of Man-Machine Studies*, vol. 20, no. 4, April 1984, pp. 343-355.

[85] Stallman, R.M., "EMACS: The Extensible, Customizable, Self-Documenting Display Editor" in Barstow, D.R., Shrobe, H.E., and Sandewall, E. (editors) *Interactive Programming Environments*, McGraw-Hill, 1984, pp. 300-325.

[86] Stallman, R.M., *GNU Emacs Manual*, First Edition, Emacs Version 16, June 1985.

[87] Standish, T.A., Harriman, D.C., Kibler, D.F., and Neighbors, J.M., *Improving and Refining Programs by Program Manipulation*, Department of Information and Computer Science, University of California at Irvine, Irvine, California, February, 1976.

[88] Strömfors, O. and Jonesjö, L., "The Implementation and Experiences of a Structure-Oriented Text Editor", Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation, *Sigplan Notices*, vol. 16, no. 6, June 1981, pp. 22-27.

[89] Strömfors, O., "A Structure Editor as a Template for Programming Environment Functions", *Proceedings of a Workshop at Roskilde University Centre*, October 22-24, 1986, pp. 197-202.

[90] Tanenbaum, A.S., *Structured Computer Organization*, Prentice-Hall Inc., 1976.

[91] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communication of the ACM*, vol. 24, no. 9, September 1981, pp. 563-573.

[92] Teitelbaum, T., Reps, T., and Horwitz, S., "The Why and Wherefore of the Cornell Program Synthesizer", Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation, *Sigplan Notices*, vol. 16, no. 6, June 1981, pp. 8-16.

[93] Tennent, R.D., *Principles of Programming Languages*, Prentice-Hall 1981.

[94] Thacker, C., McCreight, E., Lampson, B., Sproull, R., and Boggs, D., *Alto: A Personal Computer*, Xerox Palo Alto Research Center Technical Report, CSL-79-11, August 1979.

[95] "Outline Buyers Find New Uses", *Infoworld*, July, 1985, p. 22.

[96] Waters, R.C., "Program Editors should not Abandon Text Oriented Commands", *Sigplan Notices*, vol. 17, no. 7, July 1982, pp. 39-46.

[97] Winograd, T., *Language as a Cognitive Process*, Addison-Wesley Publishing Company, 1983.

[98] Winograd, T., *Muir: A Language Development Environment*, Stanford CSLI report, forthcoming.

[99] Winograd, T., *Aleph: A System Specification Language*, Stanford CSLI report, forthcoming.

[100] Wirth, N., *Programming in Modula-2*, Third, Corrected Edition, Springer-Verlag, 1985.

[101] Wozencraft, J., *Sedit, A Wonderful New Structure Editor*, Xerox Parc, (draft of manual, not published yet.)

[102] Yue, K., *Constructing and Analyzing Specification of Real World Systems*, STAN-CS-86-1090, Ph.D. Thesis, Department of Computer Science, Stanford University, September 1986.

[103] Zelkowitz, M.V., "A Small Contribution to Editing with a Syntax Directed Editor", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984, pp. 1-6.