Brussels Free University
Faculty of Sciences
Academic Year 1992-1993

A Customizable, Ergonomic, Hybrid Structure-Oriented Editor

Thesis submitted in view of obtaining the degree of Master in Computer Science

By Koen De Hondt

Promotor Prof. Theo D'Hondt

CONTENTS

Acknowledgements

Many people, among which my girlfriend, my mother and my sister, thought that I bit off more than I could chew when I decided to study yet one more year, during which I had to do my regular job too. At this time, I think it is fair to say that they were right. Especially the last few months, I had too much on my plate.
But I think it was worth the trouble. This year has been instructive in many ways and for that I would like to thank the following people :

Professor Theo D'Hondt for being the promotor of this thesis.

All members of the Agora group for many interesting discussions that have directly or indirectly contributed to my work: Wim Codenie, Theo D'Hondt, Carine Lucas, Patrick Steyaert, Thanh Son Du, Marc Van Limberghen.

Patrick Steyaert, Carine Lucas and Serge Demeyer for reading drafts of this document. I am particularly gratefull to Patrick for his helpful comments and suggestions concerning the structure of this thesis.

Other people of the Programming Technology Lab, including Professor Viviane Jonckers, who showed an interest in my work and made useful suggestions.

My student colleagues Marleen De Molder and Karel Driesen for discussions on various subjects to improve our work.

Finally, I want to thank my girlfriend Nele for taking care of things while I spent all my time "hitting the keyboard" during the last month in order to get this thesis finished.

Koen De Hondt
August 1993

One
Introduction

"Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed and debugged in an environment that consistently acknowledges and reinforces this viewpoint" [Teitelbaum & Reps 81].

Since the work of Teitelbaum and Reps, research on structure-oriented editors and structure-oriented environments has boomed. In the last decade many software development environments featuring a structure-oriented editor were developed, and today structure-oriented editing still is an active research topic.

Several approaches have been taken. Ad-hoc implementations for existing programming languages have been made, structure-oriented editor generators have been developed, structure-oriented editors have been integrated in generic environments. Several formalisms suitable for structure-oriented editing were developed in order to represent programs internally. Besides for programs, structure-oriented editing has also been used for manipulation of structured documents.

Until now, very few prototypes have left the research laboratories. Due to deficiencies in their design, current structure-oriented editors are hard to use. They have to struggle against the well designed text editors of today, with which programmers have a long experience. As long as ergonomics of structure-oriented editors does not improve, the average programmer will be reluctant to use them.

The work reported in this thesis is intended to be a step in the direction of ergonomic structure-oriented editors.

1.1 Structure-Oriented Editing

A structure-oriented editor, or structure editor, is a tool to write programs in terms of the grammatical constructs of a target programming language. A structure editor manipulates an internal representation of a program that is different from a traditional text representation. Using a structure editor shortens the edit-compile-link cycle, because building the internal representation directly makes the parsing phase redundant. In an integrated programming environment, the internal representation is a representation which can be used by all parts of the environment. Translations from one format to another are therefore not necessary anymore.

Structure-oriented editing, or structure editing, is the way how editing is performed with a structure-oriented editor. Two approaches to structure-oriented editing exist, the *pure* and the *hybrid* approach.

1.1.1 Pure Editing

Pure structure-oriented editing is based on *structure synthesis* and *structural transformation*. *Templates* and *placeholders* are two key concepts in structure synthesis. A template is a grammatical construct of the target programming language. It contains placeholders to be filled in. Structure synthesis is achieved by *expanding* placeholders by templates. Lists of templates are synthesized by means of an *insert* operation. A *remove* operation allows deletion of a template or placeholder from a list or from another template.

Structural transformation is the replacement of a template by another one which is a transformation of that template. Structural transformation can be viewed as a combination of multiple expand, insert and remove operations, and is therefore a very useful operation.

Structure-oriented editing in a pure fashion ensures that only correct programs can be composed, because it restricts the editing commands to those which are applicable at a given time, and because templates are predefined entities that are not typed. Introducing syntactical errors is impossible.

1.1.2 Hybrid Editing

Hybrid editing is pure structure editing enhanced with text editing capabilities. Besides structure synthesis and structural transformation, a hybrid editor also provides textual editing of program fragments. The degree in which the editor is hybrid depends on whether text editing is restricted to predefined program fragments or whether it is not restricted at all.

Structure-oriented editing in a hybrid fashion may introduce syntactical errors in a program. A parser is needed to check whether a typed program fragment is syntactically correct and to generate the internal representation of that fragment. Since a program is parsed incrementally, the overhead of the parsing phase is minimal and therefore the edit-compile-link cycle in hybrid structure editor environments is still shorter than in traditional programming environments.

1.1.3 Existing Systems

Several structure-oriented editors have been designed in the last decade. In this thesis features of some of the existing systems are evaluated and used to compare our design with. An overview of the systems mentioned in this thesis is given here. One older system, the Cornell Program Synthesizer, is taken for its impact on all structure-oriented environments of the last decade. Two recent systems, SbyS and GSE, are taken to contrast the Cornell Program Synthesizer with and to show which aspects of structure-oriented editors have, or have not evolved in the last ten years.

Cornell Program Synthesizer

The CPS, or Cornell Program Synthesizer [Teitelbaum & Reps 81], is one of the first structure-oriented programming environments. It was hand-crafted for PL/CS, a dialect of PL/1. In a sense the CPS had a hybrid nature. It allowed textual editing of *phrases*, i.e. identifiers and small expressions. Structure synthesis was performed in a command driven fashion. After each editing step static semantics was checked and errors were highlighted in the program text. Important in its design was the consistent view of programs as being hierarchical structures. The structure was not only important for the editor, but also for other parts of the environment, such as the interpreter and the debugger.

Structure by Structure

SbyS, or Structure by Structure [Minör 90], is a pure structure-oriented editor integrated in the Mjølner/Orm environment for incremental language and software development [Magnusson & al. 90]. SbyS is based on grammar interpretation, that is, program synthesis is done according to language grammars that are interpreted during editing. Moreover, the user is able to modify the grammars in an interactive way. The interpreted grammars can be changed while an editor using them is executing.

GSE

GSE, or Generic text and Structure Editor [Koorn 92], is a hybrid structure-oriented editor integrated in the Centaur environment [Clément 90]. GSE is language independent. An editor can be generated when a language definition in SDF, or Syntax Definition Formalism [Hendriks 91], is given. It is based on parsing, rather than on pretty printing. There is thus no need for a program presentation formalism and comments in programs are handled in a predictable way. Arbitrary program fragments can be edited textually. As a consequence, GSE smoothly integrates textual and structure-oriented editing.

1.2 Evaluation of Structure Editors

In this section design issues of existing structure-oriented editors are evaluated. Such evaluation is not an easy task, since not all structure-oriented editor implementations are made public. In the literature several systems are reported on, but issues concerning ergonomics, where we are particularly interested in, are seldom discussed. Many documents describe how a structure-oriented editor is designed on the inside, but fail to mention what the *look and feel* is on the outside.

Several systems are evaluated based on one report (see the references), while the three systems mentioned in the previous section are evaluated in detail.

The Cornell Program Synthesizer is evaluated based on the tutorial introduction [Teitelbaum 80] and report [Teitelbaum & Reps 81].

SbyS is evaluated based on [Minör 90], [Minör 91], [Mjølner 91] and the implementation in the Mjølner/Orm environment, version 1.3. Working in Orm, an abstract grammar of a target language was specified and the resulting structure editor for that language was used to write programs with. The abstract grammar was also specified with SbyS.

GSE is evaluated based on [Koorn 92], [Centaur 92], [Hendriks 91] and the implementation in the Centaur environment, version 1.2. Centaur uses GSE for language specifications in the SDF formalism. An SDF specification of a target language was given using GSE. Since more specifications in other formalisms had to be given before Centaur could generate an environment for the target language, GSE was not used to write target language programs with.

1.2.1 Structure Synthesis

Traditional structure synthesis is done by selecting a program fragment and giving an expand command. Expand commands take different forms: the Cornell Program Synthesizer has as many commands as templates. SbyS and GSE display a (hierarchical) menu containing all expansion alternatives.
Expansion of terminals, i.e. identifiers and literals, is rather special in structure synthesis. Some editors allow input of terminals (or arithmetic expressions) directly into the program text, other put up a dialog window after an expand command to input the terminal. Note that the latter violates the principle of direct manipulation (see later).

Structure editors basically provide three operations for program synthesis:

- expand — to replace a placeholder by a template
- insert — to add a template to a list
- remove — to delete a template or a placeholder

Generally, editors also support a clipboard facility by means of cut, copy and paste operations. These operations can be implemented in terms of the basic operations above.

Editors using grammar formalisms containing optionals (e.g. {statement} in EBNF) do not require an insert operation as meant here. These editors automatically generate a new placeholder when the last placeholder in a repetition gets expanded. Editing with such editors is a troublesome task as they do not provide an operation to insert a template before another one.

Traditional structure editors only support expanding placeholders, that is, replacing a placeholder by an appropriate grammatical construct as defined by the language's grammar. As a consequence changing the component of a grammatical construct requires two editing steps : removal of the component and expansion of the placeholder appearing after the removal. It is clear that such editors lack a general replacement operation.

Structure synthesis is done top-down, since a template must be created before its components can be expanded. This constrains the user who usually builds programs in a mixed top-down and bottom-up fashion. This is especially apparent when building program fragments containing little or no surface syntax, such as numerical expressions. The user is expected to think about the desired infix expressions as prefix expressions, since the operation is the first fragment to be generated before the operands can be expanded.

Structure synthesis ensures syntactical correctness of the synthesized program (modulo the unexpanded placeholders). Due to the limited editing operations some changes to the program are hard to make. It takes a sequence of actions to carry out "complex" changes such as, for example, substituting a program fragment by another one containing that fragment. Even "simple" changes take several actions before completion: Changing a Modula-2 [Ford & Wiener 85] procedure to a function is a simple task when using a text editor (adding a colon and a function type), but becomes a complicated one when using a pure structure editor (cutting the formal parameter list, cutting the procedure body, removing the procedure template, inserting a function template, pasting the body, pasting the formal parameter list, expanding the function type).

1.2.2 Structural Transformation

Structural transformation is the replacement of a subtree of the program by another one that is a transformation of the original. The two subtrees belong to the same syntactic category. For example, replacing a statement by a loop statement containing the original statement as the body is a structural transformation.
Structural transformation solves the problems discussed at the end of the previous section.

In [Shani 83] it is stated: "The reason for preferring textual commands to perform major structural changes in a program is the lack of appropriate structural commands in current editors.". To date, still little editors support structural transformations, and if they do, the transformers are usually hard coded.

The Cornell Program Synthesizer offers hard coded template-to-template transformations to "alleviate inconveniences imposed by structural constraints". In the concrete this means that some template insertion commands are applicable on other templates. For example, a '.du' command after selection of a do-while construct transforms the do-while into a do-until enclosed by an if statement and with the condition negated. This transformation is not just a structural transformation, the editor knows the semantics of the templates. He changes a '<=' comparison to a '>' comparison when negating the condition in the example.
The behavior of an insertion command thus varies whether it is applied to a placeholder or to a template. This inconsistent behavior should be avoided [Apple 92].

Transforming is substantially different from inserting, therefore these operations should be invoked by different commands.

In the structure editor of the Gandalf environment commands for "common transformations" are implemented [Notkin & al. 83]. What "commonality" means is left open. To the best of our knowledge the use or non-use of structural transformations is never investigated. Therefore incorporating transformers in a structure editor is mostly done in an ad-hoc manner, based on experience with writing programs. The set of transformers usually is the greatest common divisor of the transformations most frequently used by a set of persons.

Clearly, a structure editor should be open-ended concerning transformers. A fixed set of transformers is unsatisfactory. It limits user freedom. A user should be able to add, remove and change transformers. Hard coding them precludes modification and should therefore be avoided.

It is striking that relatively recent structure editor environment generators, like Centaur [Hendriks 91][Centaur 92], DOSE [Kaiser & al. 88], Pregmatic [van den Brand 92] and generic structure editor environments, like Orm [Mjølner 91], do not provide any formalism to describe structural transformations, as if structural transformation is felt to be redundant. We can only regret this fact. Structural transformation is an indispensable structure-oriented editing tool.

1.2.3 Direct Manipulation

A direct manipulation interface is an interface that models the real or an imagined world without using intermediary structures as command languages or menus [Shneiderman 83]. In [Apple 92] it is stated: "Direct manipulation allows people to feel that they are directly controlling the objects represented by the computer. According to the principle of direct manipulation, an object on the screen remains visible while a user performs physical actions on the object. When the user performs operations on the object, the impact of those operations is immediately visible.".
Direct manipulation has been shown to be very suitable for drawing and for managing files, as illustrated by the Macintosh Finder™ and numerous drawing applications [Apple 92].

In [Minör 91] a direct manipulation interface for structure-oriented editors is suggested (but not implemented). Program pieces are fetched from a palette window to be inserted in a program displayed in the target window. Temporary storage of program

fragments is supported by a multi item clipboard window. Only pieces that fit can be inserted at a placeholder in the target window.
The suggested palette window displays program fragments in a semigraphical way, but investigation has shown that a graphical representation is no more efficient than a more simple implemented text-based representation [Toleman & al. 92]. The latter has the advantage not to be expensive with respect to *screen real estate*.

1.2.4 NAVIGATION

Navigation is the way how a user has to interact with the editor in order to select program fragments.
Early structure-oriented editors, like the Cornell Program Synthesizer, executed on machines with character-based screen displays. Typically, those editors came with a variety of cursor motion keys[1].
Today, most computers are equipped with high resolution monitors and pointing devices. Navigation with the mouse has become easier from a user's point of view, but the implementor has to design new algorithms to make *random access* to program fragments possible.

Navigation can also be performed with keyboard or menu commands. Menu command names like 'zoom in', 'zoom out', 'next child', 'previous child' [Hendriks 91] should be avoided. The meaning is not clear to the user in the context of programming.

Search facilities are usually not provided. Nevertheless, searching should be an integral part of a structure editor. It is an ideal operation to navigate in a program.

1.2.5 FOCUS OF ATTENTION

A structure editor maintains two selections, the current selected subtree and the current selected text. In a pure structure editor the current text selection always corresponds to the current subtree selection, that is, the selected text is the text representation of the selected subtree. In a hybrid structure editor there is not a relation between the text and subtree selection at al times, because arbitrary text selections can be made.

The focus of attention is the current selection. In most systems the focus corresponds to the current selected subtree.

Different editors display the focus of attention in different ways.
The Cornell Program Synthesizer sets the cursor on the first character of a selected template. A phrase is selected when the cursor is positioned on one of its characters.

[1]The Cornell Program Synthesizer has 14 cursor motion commands!

GSE is a hybrid editor. The focus is depicted through a line around the current selected subtree. Text editing can only be performed within the focus. Issuing text editing commands outside the focus enlarges the focus. Because of the tendency to grow, the possibility of introducing multiple focusses in GSE is investigated.
When using GSE one gets distracted by the continuously changing focus outline. Displaying the focus outline adds no information to the interface. The user is not interested in what part of the program tree is selected. A textual selection should be enough. Let the editor worry about subtree selections, not the user.

1.2.6 HYBRID EDITING

Hybrid editing solves many of the interaction problems found in pure structure-oriented editors. In particular, hybrid editing is a solution for the problems in connection with structure synthesis mentioned before.

In [Waters 82] Waters warns against pure structure editing. Some of his remarks are not well-studied, as noted in [Notkin & al. 83], but we agree when he states: "When you come up with a new *better* way to do something, should you create a system which *mandates* its use or should you create a system which *offers* its use in addition to older methods?".

Since then, it is widely accepted that hybrid editing solves some interaction problems of pure structure editing as validated by several existing hybrid editors: GSE [Koorn 92], Magpie [Schwartz & al. 84], Poe [Fisher & al. 84], PSG [Bahlke & Snelting 86], SUPPORT [Zelkowitz 84], Sif [Mjølner 90], Syned [Horgan & al. 84].

However, the hybrid editing approach has been criticized in [Minör 90]:

1. The same program is represented in two different ways and the user has to keep track of both representations.
2. The two representations are manipulated by different command sets.
3. The text editing commands are not applicable on the structure representation and structure editing commands are not applicable on the text representation.
4. Syntactic errors in the textual representation may either cause inconsistencies between the two representations, or the user gets stuck in a mode until the errors are corrected.
5. Different parsing algorithms put restrictions on the languages they are able to parse, while grammars for structure synthesis have to meet other conditions.

The first four arguments are considered interaction issues, but they are in fact all a consequence of Minör's view that the user of a structure editor writes a program by building its tree representation in stead of its textual representation. In our view interacting with a structure editor is not that different from interacting with a text editor: A structure editor has some higher level commands not found in text editors and uses a slightly different selection strategy. With that in mind the first four arguments boil down to technical problems for the designer.
We will try to comment on these arguments one by one.

Argument 1. The program is represented by the computer in two different ways, but the user interacts with only one, the text representation. Editors that assume the user to have knowledge about the underlying abstract grammar tree have a poorly designed interface. The user has knowledge about the concrete syntax of the language, not the abstract one. The editor should present the user a text on which, under some conditions, more commands are applicable than on an ordinary text. This is comparable with a word processor which provides commands for manipulation of paragraphs and sections.

Argument 2. In our view this is not the case. The user interacts with the program text and all commands are applicable thereon. The structure-oriented commands are an extention of the text-oriented commands.

Argument 3. This argument is not completely correct. Cut, copy and paste commands are applicable equally well on text as on structure selections.
The user's view of the program is a structured text. He should be able to text edit the program at any time, at any place. Structure editing should be available whenever possible, that is, when a structure selection is made.

Argument 4. The place of a text edited piece of the program should be marked in the abstract syntax tree with a special placeholder until it can be replaced by a subtree generated by the parser. If the editor supports only one focus of attention the user gets stuck in an "edit this incorrect program piece" mode, as argumented, otherwise the user can make selections/changes outside the edited piece and return later to correct the problem. In the latter case no inconsistencies arise, since the non-parsed text is considered a placeholder with a non-traditional text representation.

Argument 5. This is truly a technical problem, especially for environments that use a parser generator, e.g. YACC [Mason & Brown 90], to generate a parser for the target language. The generated parser not necessarily produces the same derivation trees as the structure synthesis component of the structure editor. [Ballance & al. 88] reports on an approach of maintaining two different grammars and guaranteeing consistency between them by viewing abstract grammars as abstractions of parsing grammars.

1.2.7 ERGONOMICS

Ergonomics is the "study of work and working conditions in order to improve people's efficiency" [Oxford 92].

Ergonomics of structure editors (and that of many other applications) has increased drastically since the generalization of see-and-point interfaces [Apple 92][Goldberg 84b] stemming from the introduction of pointing devices.
But, after evaluation of some pure structure-oriented editors, one comes to the conclusion that the working conditions have deteriorated compared to text editors. Programmer's efficiency certainly has not improved.

WIMP interfaces are employed for structure editors, but clearly little time is spent to implement common human interface guidelines [Apple 92]. As a consequence structure-oriented editors are unattractive and hard to use.

1.2.8 SUMMARY

The evaluation is summarized here for ease of reference:

- Structure synthesis is primitive
  - Replacement and insertion alternatives are fixed
  - Editors lack a general replacement operation
- Structural transformations (if provided) are hard coded
- Direct manipulation interfaces do not exist
- Navigation commands are sometimes awkward
- Search facilities are not provided
- Hybrid editing is indispensable to overcome interaction problems which arise in pure structure-oriented editors
- Displaying textual selections and structural selections in different ways, distracts the user
- Ergonomics is almost always neglected

1.3 ROOM FOR IMPROVEMENT

It is clear from the evaluation in the previous section that the design of present day structure-oriented editors leaves room for improvement. In this section conclusions of the evaluation are drawn and suggestions for improvements are made.

1.3.1 STRUCTURE SYNTHESIS – STRUCTURAL TRANSFORMATION

Current editors provide too few editing operations for structure synthesis. A limited number of operations constrains the user who often has to go through several editing steps to achieve his goal.
Moreover, some operations lack generality. A structure editor should support replacement of any grammatical construct by another one of the same grammatical category, as prescribed by the language's grammar. Note that this assumption includes replacement of a placeholder by another placeholder.

In our view a structure editor should come with the following operations to build programs:

- replace/expand
- unexpand
- insert before
- insert after
- remove
- transform
- cut, copy, paste

The replace operation replaces an arbitrary structural selection by another picked from a set of alternatives. Expansion is replacement *of* a placeholder. Unexpansion (shrinking) is a replacement *by* a placeholder. Two insert operations are necessary to build lists.

Structural transformations should not be hard coded. Instead they should be user-defined. Providing a language to manipulate the internal representation of a program with, probably seems appealing to many (cf. Mentol in [Donzeau-Gouge & al. 84]), but adding such a language to the environment puts a burden on the user who has to learn an additional language and the internal representation used by the structure editor. In this document a pattern matching technique, encapsulated in a well designed interface, is suggested.

1.3.2 Direct Manipulation

Selecting program fragments and dragging them to another place is known as *drag and drop*, moving objects around without cut & paste operation. Drag and drop is a very powerful direct manipulation tool. It can be used to expand, replace and insert program fragments. In particular identifiers can be typed once and dragged and dropped later on.

Drag and drop in combination with cross window dragging makes of all structure editor windows on the desktop one big multi item clipboard.

1.3.3 Navigation – Search

The number of navigation commands provided by a structure editor executing on a graphical workstation with a pointing device, should be minimal. The pointing device should be the major navigation tool. Navigation commands in command menus are awkward to use, they should only be provided for navigation commands that are not frequently used.

A structure-oriented editor should not lack search facilities. Searching can be performed on the program text and on the internal representation of a program.

1.3.4 Focus of Attention – Hybrid Editing

In our view, a hybrid structure-oriented editor should not use different ways for structural and textual selection, as GSE does. A textual selection is appropriate for both selections, because it is consistent.
If a text selection corresponds to a subtree (after a structural selection for example) the structure editing commands should be available to the user, otherwise only text editing operations should be possible. In this view a hybrid structure-oriented editor is a text editor with structure editing capabilities, not a structure editor with text editing facilities.

Multiple focusses, as suggested for GSE, are adopted here. There is only one current selection. If non parsed text exists, it is not displayed in a special way, it is displayed as all other program text. Internally, the editor keeps track of non parsed pieces and merges them when possible or needed.

1.3.5 Customizability

Customizability is the ability of a system to let the user tailor it to his needs. In general customizability is supported by presenting the user a set of parameters that can be altered. In structure-oriented editors this is not different, interactively changing the presentation schemes is an example thereof, but the designer can go beyond the ordinary.

The set of replacement and insertion alternatives for program synthesis and the set of available structural transformations are candidates for customization. There is no reason why these sets should not be extendible. Extendibility increases the efficiency of the user, since he can accomplish several editing operations much faster with his own customized commands.

Changing the set of replacement and insertion alternatives can be supported in two ways. Program fragments built with the editor can be installed with a command in order to be used later, and an interface can be provided to add and remove alternatives and to modify already installed alternatives.
Specifying and modifying structural transformations can be supported by providing an interface to lay down how a source program fragment is transformed into a destination program fragment.

1.3.6 Ergonomics

Ergonomics is very important. A structure editor of which the ergonomic value is not higher than the average text editor today, will not be used.
The ergonomic value can be increased by supplying a well designed, easy to use, user friendly user interface in which direct manipulation plays a major role. Not only the user interface, but also the structure-oriented commands are subject to improvements. The user should be able to give commands easily. It is fundamentally wrong to provide commands that are verbose and non intuitive.

1.3.7 Conclusion

In this chapter an evaluation of some design issues of structure editors was made and recommendations to improve interaction were given. All recommendations boil down to considerations for improving the ergonomic value of present day structure editors.

In our view, a pure structure-oriented editor is of no use. A structure editor should be more ergonomic than a text editor, otherwise programmers are not inclined to use it.
Ergonomics of pure structure editors can be increased drastically by turning them into hybrid editors. A comprehensive set of customizable synthesis operations, including structural transformation, is indispensable to exploit structure editors to their full potential.

1.4 Major Goals and Contributions

The first major goal has been the design of a customizable, ergonomic hybrid structure-oriented editor for Agora. The second major goal was to consider such design in a general way to obtain a generic structure-oriented editor.

Generic means here that the design of the editor is independent of the target language, i.e. the language for which the editor serves. Making the editor generic has the advantage that only a grammar of (and possibly a parser for) a target language must be specified in order to obtain a structure-oriented editor for that language. The editor itself need not be modified.

Customizability is the ability of the editor to let the user tailor it to his needs. Customization includes extendibility, i.e. the ability to let the user extend the command set of the editor.

Ergonomics is an aspect that often gets neglected. Several present day structure-oriented editors suffer of interaction difficulties due to badly designed interfaces and neglected, yet important, issues like ease of use and user friendlyness. The way how editing tools are presented to the user should be carefully considered. In several systems the editing commands are verbose and hard to learn. A structure-oriented editor should be a more powerful tool than the text editors used today. It should increase people's efficiency. Nevertheless many systems fail to raise their ergonomic value above the text editor level. Therefore it is obvious that such structure-oriented editors will never catch on by the average programmer outside universities and research laboraties.

Hybrid editing, as opposed to pure structure-oriented editing, is necessary to solve many interaction problems of pure structure-oriented editing. One goal was to examine how text editing can be mixed with pure structure-oriented editing to obtain a hybrid editor.

Structure-oriented editing is still an active research topic. Many of the topics discussed in this thesis are rarely described in the literature. In particular ergonomic issues are seldom or never dealt with.
The major contributions of this work are considered to be:

- A formal definition of structured grammars
- Use of structured grammars in order to make a structure-oriented editor generic

- A direct manipulation interface for structure-oriented editors
- Structural transformation as an indispensable structure-oriented editing tool
- Hybrid editing that does not introduce modes in a structure-oriented editor
- Extendibility of a structure-oriented editor to increase ergonomics
- An ergonomic interface for structure-oriented editors

1.5 THE AGORA PROJECT

The work reported on in this document is accomplished as part of the Agora project, which originated in 1987 at the Programming Technology Lab of the Computer Science Department at the Free University of Brussels. Agora is an object-oriented programming language under development. One of its hallmarks is its consistent use of mixins for inheritance [Steyaert & al. 93]. Currently, seven people are part of the Agora group. Current research topics are reflection, data flow analysis, typing, concurrency, efficient execution, structure-oriented editing.

Within the Agora project the author occupies himself with the programming environment aspect of the project. This aspect comprises text editors, structure-oriented editors, browsers and inspectors for programming in and experimenting with Agora.

1.6 THE AGORA STRUCTURE EDITOR

The early design of a structure-oriented editor for Agora began in 1990. In the beginning overcoming the technical problems for structured navigation in a program with a pointing device was the main goal, as described in [De Hondt 90]. Since then the editor has matured considerably on many areas. Navigation has been perfected, flexibility became important, hybrid editing is introduced properly, powerful editing tools are provided and in the whole design ergonomics has played a very important role.

Agora is a programming language with little surface syntax. As a consequence, the structure of a program is not always clear. It is our believe that a structure-oriented editor is a necessary aid to make the structure of programs, written in a language with little surface syntax, tangible.

1.7 STRUCTURE OF THIS THESIS

The topics discussed in the following chapters help to meet our design goals, presented in this introduction.

First the inside of a structure-oriented editor is discussed.

Chapter two surveys terminology and properties of standard *context-free grammars* and *attribute grammars*, as found in the literature, and defines *abstract grammars*.

The terminology and properties are needed for the formal definition of *structured context-free grammars*, *structured abstract grammars* and *structured attribute grammars* in chapter three. Due to their structure, these grammars are very suitable for specification of programming languages in a structure-oriented editor. From structured abstract grammars, the internal representation of programs, i.e. the *abstract derivation tree*, is derived.

In chapter four, after a discussion how placeholders are derived from a structured abstract grammar, the abstract derivation tree is extended with placeholders to obtain *abstract syntax trees*, the internal representation of a program used by a structure editor. The *tree path* is introduced for selection purposes in abstract syntax trees and the *template* is presented to embody all properties of a production rule in a structured abstract grammar.

After the first four chapters, attention is turned to structure editor aspects on the outside.

Chapter five introduces presentation schemes, which lay down how programs, represented by abstract syntax trees, are presented textually. How a textual representation is actually generated is described by means of a structured attribute grammar.
At the end of chapter five the *HV polygon* is presented. HV polygons are the basis for the mapping between the internal and the external representation of a program. This mapping is used extensively for selection purposes.

Chapter six focuses on the interaction with the user interface of a structure editor. It is shown how structure-oriented editing commands are given and what interfaces for customization of presentation schemes, expansion/insertion menus and structural transformations may look like.

Finally, in chapter seven the implementation status is briefly described, future work is presented and conclusions are drawn.

Two
STANDARD GRAMMARS

Programming environments consist of functional components that manipulate the internal representation of a program. For all components except a text editor, a text representation of a program is a poor data structure. It has little or no structure, it certainly does not reflect the structure of the program.
An integrated programming environment, that is, a programming environment of which all functional components are integrated in one program, benefits considerably from a uniform internal representation that can be used by all integrated components. A uniform representation eliminates conversions from one format to another.

In the late fifties and early sixties formal language theory originated in the work of Chomsky (see [Harrison 78] for historical surveys on formal language theory). The theory of (context-free) grammars has proven to be very usefull in program language design. Given a grammar and a program written in the language generated by that grammar, a parser can generate the program's internal representation, a so called derivation tree or parse tree.
A parse tree is an internal representation with some overhead, and is therefore not the ideal representation of a program for the functional components of a programming environment.

In the seventies abstract grammars were introduced in the Mentor project [Donzeau-Gouge & al. 80]. Abstract grammar trees reflect the structure of programs without the overhead parse trees suffer of. In the last decade abstract grammars were generally employed in programming environments. A parser and/or a structure editor builds an abstract derivation tree which is used by all other components of a programming environment.

Several components decorate abstract grammar trees with attributes to annotate each piece of a program with information relevant for the programming environment. Attribute grammars [Wait & Goos 84] lay down the theoretical foundation for this annotation.

Programming environments utilize different grammars for different purposes. Environments parse programs by means of concrete grammars. The parser generates an abstract grammar derivation tree to represent the program internally. Abstract grammars are also used by structure-oriented editors for program synthesis. Semantic grammars (attribute grammars) define static semantics and code-generation grammars translate the abstract grammar tree into an intermediate language expressing the dynamic semantics [Magnusson & al. 90].

This chapter gives an overview of properties of *context-free grammars*, *abstract grammars* and *attribute grammars*. Some of these properties are needed for the introduction of *structured grammars* in the next chapter. Several properties are commented on from a programming environment point of view.
The definitions and theorems concerning context-free grammars are taken from [Harrison 78], [Lewis & Papadimitriou 81] and [Aho & al. 86], the definition and the theorem concerning attribute grammars is taken from [Wait & Goos 84] and [Nijholt 88]. Proofs of theorems are omitted here; they can be found in the first four references.

2.1 CONTEXT-FREE GRAMMARS

Concrete grammars are used to describe the syntax of a programming language in a formal way. Usually a programming environment utilizes concrete grammars for parsing purposes. Therefore concrete grammars are also referred to as parsing grammars[2].

Several types of parsing grammars exist (Chomsky hierarchy, see [Harrison 78]) of which the context-free grammars lend themselves well to describe a large set of programming languages. Many programming environments make use of context-free grammars, often with constraints (LL, LR, …) for efficiency reasons and ease of implementation.

---

[2]In [Minör 90] and [Magnusson & al. 90] the term 'concrete grammar' has a different meaning. Minör's concrete grammars describe presentation schemes, rather than language syntax. To avoid confusion, the expression 'parsing grammar' will be used in the rest of this document.

Definition 1 A **context-free grammar** (CFG) is a tupel G = (N, T, P, S) with
N = set of non terminal symbols
T = set of terminal symbols
$P : N \rightarrow (N \cup T)^*$ the set of production rules
$S \in N$ the start symbol.

In what follows, ε (epsilon) denotes the empty string of terminals and non terminals.

Notation Given a CFG G = (N, T, P, S)

$A \rightarrow_G \beta$ if $(A, \beta) \in P$
$\alpha \Rightarrow_G \beta$ if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \omega \alpha_2$, $A \rightarrow_G \omega$
with $\alpha, \beta, \alpha_1, \alpha_2 \in (N \cup T)^*$
$\Rightarrow_G^*$ reflexive and transitive closure of $\Rightarrow_G$
When the grammar is clear from context the subscript G in $\Rightarrow_G$ and $\rightarrow_G$ is omitted.

$\alpha \in (N \cup T)^*$ is said to be generated by G if $S \Rightarrow^* \alpha$.
A series $\omega_0 \Rightarrow \omega_1 \Rightarrow \ldots \Rightarrow \omega_n$ is called a derivation (from ω_0 to ω_n).

$L(G) = \{\omega \in T^* \mid S \Rightarrow^* \omega\}$ is the language generated by G. A language generated by a CFG is called a context-free language.
G and G' are equivalent if they generate the same language.

Definition 2 Let G = (N, T, P, S) be a CFG.
A production $A \rightarrow \varepsilon$ in P is called an ε **rule**.
A production $A \rightarrow B$, with $B \in N$, is called a **chain rule**.

Definition 3 A CFG G = (N, T, P, S) is ε **free** if
$\forall\ A \rightarrow \varepsilon \in P : (A = S \text{ and } \neg \exists\ B \rightarrow \omega S \nu)$.
(only the start symbol may go to ε and, if so, the start symbol may not appear in the right hand side of any production rule)
A CFG G = (N, T, P, S) is called **chain free** if
$\forall\ A \rightarrow \alpha \in P : \alpha \notin N$.

Example The grammar with the following production rules is chain free, but not ε free. It generates lists of one or more identifiers separated by a comma.

IdentifierList $\rightarrow$ Identifier ListTail

ListTail $\rightarrow$, Identifier ListTail

ListTail $\rightarrow \varepsilon$

The production rules can be reworked to get an equivalent ε free grammar with the following production rules.

IdentifierList $\rightarrow$ Identifier

IdentifierList $\rightarrow$ Identifier ListTail

ListTail $\rightarrow$, Identifier ListTail

ListTail $\rightarrow$, Identifier

Theorem 1 For every CFG G = (N, T, P, S) there exists an ε free CFG G' = (N', T, P', S') with L(G) = L(G').

Theorem 2 Let G = (N, T, P, S) be a CFG. There exists an equivalent CFG G' = (N, T, P', S) such that for every production rule $A \rightarrow \alpha \in P'$ with $A \rightarrow \alpha \neq S \rightarrow \varepsilon$, $\alpha \notin N \cup \{\varepsilon\}$ holds.

Any context-free grammar G = (N, T, P, S) containing productions $A \rightarrow B$ with $B \in N$ can thus be transformed in an equivalent one without this kind of useless productions.

Theorem 3 For each CFG G = (N, T, P, S) there exists an equivalent CFG G' = (N, T, P', S) only having productions of the form $S \rightarrow \varepsilon$, $A \rightarrow \alpha$, $A \rightarrow a$, with $\alpha \in NN^+$, $a \in T$.

Each context-free language can thus be generated by an ε free and chain free context-free grammar.

Theorem 4 Let G = (N, T, P, S) be a CFG and $A \rightarrow \alpha B \beta$ with $B \in N$; $\alpha, \beta \in (N \cup T)^*$. Let $B \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k$ be all rules with left hand side B. Define G' = (N, T, P', S) with $P' = P - \{A \rightarrow \alpha B \beta\} \cup \{A \rightarrow \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_k\beta\}$.
Then L(G) = L(G').

Definition 4 Given CFG G = (N, T, P, S).
Symbol $X \in (N \cup T)^*$ is **useless** if there exists no derivation $S \Rightarrow^* wXy \Rightarrow^* wxy$ with $w, x, y \in T^*$.
G is **reduced** if it contains no useless symbols.

Theorem 5 For any CFG G there exists a reduced CFG G' with L(G) = L(G').

Definition 5 A CFG G = (N, T, P, S) is in **Chomsky Normal Form** (CNF) if every production rule in P has one of the following forms :

(1) $A \rightarrow BC$ with A, B, $C \in N$

(2) $A \rightarrow a$ with $a \in T$

and if $\varepsilon \in L(G)$ then $S \rightarrow \varepsilon \in P$ and S does not appear in the right hand side of any production rule.

Theorem 6 For any CFG G = (N, T, P, S) there exists a CFG G' in CNF with L(G) = L(G').

In other words, every context-free grammar has an equivalent binary context-free grammar.

This theorem has direct practical use in the implementation of parse trees. Derivation trees in Chomsky Normal Form contain only binary nodes, that is, nodes of fixed length, for which memory management is implemented more efficiently than memory management for nodes of arbitrary length. In MiniPas, an educational environment for a subset of Pascal [Jadoul & al. 88], this technique is used (in fact only the abstract grammar is made binary). The loss of speed, as the tree contains more links, is compensated by the optimal memory management.

Definition 6 A CFG G = (N, T, P, S) is in **binary standard form** if each rule is of the form $A \rightarrow BC$, with B, $C \in N \cup T$.

Theorem 7 For any CFG G = (N, T, P, S) and $\omega \in T^*$ it is decidable whether $\omega \in L(G)$.

The corollary of this theorem is that for any context-free grammar a parser can be written.

Efficient 'parsing methods' were developed to check whether an arbitrary string belongs to a context-free language, e.g. Earley's algorithm and the algorithm of Cocke-Kasami and Younger [Harrison 78]. For subclasses of context-free grammars, such as LL and LR grammars, methods were developed to parse in linear time with regard to the length of the input string [Wait & Goos 84].

Definition 7 A CFG G = (N, T, P, S) is **ambiguous** if there exists a $\omega \in L(G)$ for which there are at least two derivations.

A context-free language L is **inherent ambiguous** if no non ambiguous CFG G exists for which L = L(G).

Example The grammar with the following production rules

Term $\rightarrow$ 3

Term $\rightarrow$ 4

Term $\rightarrow$ 5

Term $\rightarrow$ Term + Term

is ambiguous, because the sentence '3 + 4 + 5' has more than one derivation, for example

Term $\Rightarrow$ Term + Term $\Rightarrow$ 3 + Term

$\Rightarrow$ 3 + Term + Term $\Rightarrow$ 3 + 4 + Term $\Rightarrow$ 3 + 4 + 5

and

Term $\Rightarrow$ Term + Term $\Rightarrow$ Term + Term + Term

$\Rightarrow$ Term + Term + 5 $\Rightarrow$ 3 + Term + 5 $\Rightarrow$ 3 + 4 + 5

Theorem 8 It is not decidable whether a CFG G = (N, T, P, S) is ambiguous.

In other words, there exists no algorithm that can determine whether an arbitrary context-free grammar is ambiguous.

Definition 8 A CFG G = (N, T, P, S) is **left recursive** if $\exists\, A \in N, \exists\, \beta \in (N \cup T)^* : A \Rightarrow^+ A\beta$.
It is **right recursive** if $\exists\, A \in N, \exists\, \beta \in (N \cup T)^* : A \Rightarrow^+ \beta A$.

Example A grammar with the following production rules is left recursive.

IdentifierList $\rightarrow$ Identifier

IdentifierList $\rightarrow$ IdentifierList , Identifier

Definition 9 A CFG G = (N, T, P, S) is in **Greibach Normal Form** (GNF) if each rule is of one of the forms $A \rightarrow aB_1 \ldots B_n$, $A \rightarrow a$, $S \rightarrow \varepsilon$ with $a \in T$; $B_1, \ldots, B_n \in N$.

Theorem 9 Every CFG G = (N, T, P, S) can be transformed to an equivalent ε free one with production rules $S \rightarrow \varepsilon$ or $A \rightarrow aX$ with $a \in T$, $X \in N^*$.

Every context-free language can thus be generated by a context-free grammar in Greibach Normal Form.

Corollary 10 For any CFG G = (N, T, P, S) there exists a non left recursive CFG G' with L(G) = L(G').

This corollary has practical use in the implementation of table driven parsers using LL(1) grammars, which is the class of context-free grammars suitable for recursive descent parsing. A LL(1) grammar may not contain left recursion. Reworking a left recursive CFG to a non left recursive one typically introduces right recursion and ε rules in the resulting grammar. CFGs containing production rules with common prefixes are also not LL(1), but there exists a grammar transformation algorithm to correct this problem for most programming language grammars. The Pascal grammar, with the if-then-else construct, is a notable exception. Due to the dangling else problem, no LL(1) grammar for Pascal exists [Fisher & LeBlanc 88].

Definition 10 A CFG G = (N, T, P, S) is in **Operator Form** (OF) if each rule in P is of one of the forms
$A \rightarrow B$, with $B \in (N \cup T) - \{\varepsilon\}$
$A \rightarrow \alpha BC\beta$, with $\forall\ \alpha, \beta \in (N \cup T)^*$, $B, C \in (N \cup T)$: either $B \in T$ or $C \in T$.

In other words, context-free grammars in operator form have no production rules with two adjacent nonterminals in the right-hand side.

Theorem 11 Let G = (N, T, P, S) be an ε free CFG. There exists a CFG G = (N', T, P', S) in Operator Form such that L(G) = L(G').

Definition 11 A **derivation tree of a CFG** G =(N, T, P, S) is an ordered and labeled tree such that
- the label of the root node is S
- every leaf node has a label $l \in T$
- every internal node has a label $l \in N$
- if an internal node has label A and the descendents of A are labeled from left to right with $X_1, \ldots, X_k$, $k \geq 1$, then $A \rightarrow X_1 \ldots X_k$ is a production rule in P.

2.2 Abstract Grammars

Concrete grammars often contain productions for resolving ambiguities, precendence and associativity (adding term and factor productions for numerical expressions is a good example thereof). As a consequence they do not reflect the intuitive structure of a language.

Example Consider CFG G = (N, T, P, S) with

N = {RelExp, Exp, Term, Factor, SubExp, RelOp, AddOp, MulOp}

T = {id, (,), +, -, *, /, <, >}

S = RelExp

P =

RelExp	$\rightarrow$	Exp RelOp Exp
Exp	$\rightarrow$	Term
Exp	$\rightarrow$	Term AddOp Term
Term	$\rightarrow$	Factor
Term	$\rightarrow$	Factor MulOp Factor
Factor	$\rightarrow$	id
Factor	$\rightarrow$	Subexp
SubExp	$\rightarrow$	(RelExp)
RelOP	$\rightarrow$	<
RelOp	$\rightarrow$	>
AddOp	$\rightarrow$	+
AddOp	$\rightarrow$	-
MulOP	$\rightarrow$	*
MulOp	$\rightarrow$	/

This grammar generates relational expressions. A relational expression *RelExp* has a relational operator *RelOp* and two operands which are additive expressions *Exp*. An additive expression, in turn, consists of an additive operator *AddOp* and two multiplicative expressions *Term*. Finally, a multiplicative expression has a multiplicative operator *MulOp* and two operands *Factor*, which are identifiers *id* or subexpressions *SubExp*.

The nesting of production rules for relational, additive and multiplicative expressions is mandatory to reflect the precedence rules of these expressions. This causes the necessary introduction of two rules per expression type (*Exp* and *Term*) in order to support generation of expressions with factor operands. One rule generates an expression as described above, the other directly falls through to a rule generating an expression of lower precedence (Exp $\rightarrow$ Term and Term $\rightarrow$ Factor).

The rule for subexpressions, introducing enclosing parentheses for an arbitrary expression, is needed to resolve ambiguities and associativity.

To define grammars in a more convenient way for the environment, abstract grammars were introduced [Donzeau-Gouge & al. 80]. Abstract grammars reflect the structure of a language where concrete grammars reflect the textual representation. There is no one-to-one relation between concrete and abstract grammars. For any concrete grammar several abstract grammars can be defined, each of them more or less abstract than the other. In [Ballance & al. 88] classes of abstractness are defined reflecting the resemblance of an abstract grammar to a concrete grammar.

An abstract grammar is the basic definition of the language structure. Concrete grammars are only used by a programming environment to generate a derivation of the abstract grammar which is used later on by all tools of the environment (often annotated with attributes).

The following definition gives the most general form of an abstract grammar.

Definition 12 An **abstract grammar** is a tupel AG = (A, L, P, S) with
A = alphabet
L = set of labels
$P : L \rightarrow (A \cup L)^*$ the set of production rules
$S \in L$ the start label

The alphabet A of an abstract grammar is the set of terminals, i.e. identifiers and literals. L contains all labels used to identify abstract syntax tree nodes.

Example The abstract syntax of the CFG given before can be described by (A, L, P, S) with
A = {id, <, >, +, -, *, /}
L = {Exp, Op}
S = Exp
P = Exp → Exp Op Exp
Exp → id
Op → <
Op → >
Op → +
Op → -
Op → *
Op → /

Abstract grammar derivation trees (not defined here) are more compact than concrete grammar trees. They contain less nodes and they do not contain surface syntax.

As a consequence, abstract grammar derivation trees lend themselves well to store programs efficiently on secondary storage. Storing such a derivation tree can be described with a simple pre-order recursive tree traversal algorithm.

2.3 ATTRIBUTE GRAMMARS

Attribute grammars were introduced in an attempt to subsume syntax and semantics under one formalism. Each node of a derivation tree is *decorated* with attributes describing properties of that node. The information collected in the attributes is derived from the environment of that node. It is the task of an analyser to compute their values and check their consistency.
Attribute grammars have proven to be a useful aid in representing the attribution of derivation trees, notably for semantic analysis, code generation, and, as presented in chapter five, for generation of textual representations of programs.

The definitions and theorems in this section are taken from [Wait & Goos 84] and [Nijholt 88].

Definition 13 An **attribute grammar** is a tupel AtG = (G, A, R). It is based upon a reduced context-free grammar G = (N, T, P, S).
Consider $p : X_0 \rightarrow w_0\, X_1\, w_1 \ldots w_{n-1}\, X_n\, w_n \in P$.

- For $X \in N$,
  - A(X) is an finite set of attributes
  - $A(X) = S(X) \cup I(X)$, $S(X) \cap I(X) = \varnothing$
  - $S(X) = \{X.a \mid \exists\, p : X \rightarrow \chi \in P\}$, the *synthesized attributes*
  - $I(X) = \{X.a \mid \exists\, q : Y \rightarrow \mu X \nu \in P\}$, the *inherited attributes*
- Terminal symbols do not have attributes
- S has no inherited attributes
- $A = \bigcup_{X \in N} A(X)$
- R(p) is a finite set of attribution rules; these rules define S attributes of X_0 and I attributes of X_j, $1 \le j \le n$:

  $X_k.a \leftarrow f(X_1.a_{i1}, \ldots, X_m.a_{im})$

  f takes on any of a specified set of values
- $R = \bigcup_{p \in P} R(p)$

In the literature definitions of attribute grammars, containing *semantic conditions* are given, but these conditions can equally well be described with Boolean attributes associated with the left-hand side of the production [Wait & Goos 84]. Since conditions are not needed in what follows, they are omitted here for clarity.

As terminal symbols do not have attributes, the fact that attribute grammars are based on context-free grammars is unimportant. Attribute grammars can also be based upon abstract grammars. This is usually the approach taken in programming environments when the internal representation is an abstract, rather than a concrete derivation tree.

Attributes which arise in conjunction with derivation tree construction are called *intrinsic attributes*. For example, a *value* attribute of a tree node representing a constant and a *symbol* attribute of a tree node representing an indentifier are intrinsic attributes.

Notation Given an attribute grammar AtG = (G, A, R) with $p : X_0 \rightarrow w_0\ X_1\ w_1 \ldots w_{n-1}\ X_n\ w_n \in P$.

$IA(p) = \{X_0.a \mid a \in S(X_0)\} \cup \{X_j.a \mid a \in I(X_j),\ 1 \leq j \leq n\}$
$OA(p) = \{X_0.a \mid a \in I(X_0)\} \cup \{X_j.a \mid a \in S(X_j),\ 1 \leq j \leq n\}$
IA(p) is the set of inside attributes of p
OA(p) is the set of outside attributes of p

Attribution rules thus define the inside attributes of a production rule.

Definition 14 An attribute grammar is in **Bochmann Normal Form** (BNF) if all inside attributes are defined in terms of the outside attributes.

Theorem 12 Each (non circular) attribute grammar can be transformed into an equivalent one in Bochmann Normal Form.

The definition does not specify how attributes are computed. Attribute grammars can be classified according to the evaluation order in which the attributes are evaluated. Several pre-specified orders exist, of which three simple ones are mentioned here. The attributes of *only-S* and *only-I* attribute grammars, having only synthesized or only inherited attributes, can be evaluated bottom-up or top-down respectively. *Class L*, i.e. left to right, one pass, attributed derivation trees can be evaluated with a left to right depth first tree traversal algorithm. Class L attribute grammars are generally employed in programming environments because the evaluation order agrees with the program structure represented by the derivation tree.

2.4 SUMMARY

In this chapter basic terminology and properties of context-free grammars, abstract grammars and attribute grammars are were given. *Context-free grammars* are used for the formal specification of context-free languages. *Abstract grammars* define the structure of programs. They contain no surface syntax and no production rules for resolving precedence, associativity and ambiguities, as context-free grammars do. *Attribute grammars* are used to combine syntax and semantics into one formalism.

The presented properties are used in the following chapter when structured grammars are introduced.

Three

STRUCTURED GRAMMARS

Standard grammars are felt to be too general for use in a structure editor environment. Therefore several formalisms, restricting and structuring parsing grammars and abstract grammars, were developed in the eighties. [Minör 90] gives an overview of these formalisms, of which the GRAMPS formalism [Cameron & Ito 84] influenced the structured grammar formalisms used in Mjølner [Madsen & Nørgaard 87][Minör 90] and the Agora Structure Editor.

This chapter starts with the introduction of a *structured context-free grammar*. In the literature no formal definition of structured context-free grammars is given, so we try to give one here. We also proof that a structured context-free grammar can be found for any standard context-free grammar.
From structured context-free grammars *structured abstract grammars* are derived. They lay down the internal representation of a program used in the subsequent chapters of this document. *Structured attribute grammars* are defined next. They are used for describing presentation schemes in the next chapter. Finally, the *internal program representation* used in the following chapters is defined according to the structured abstract grammar formalism.

3.1 STRUCTURED CONTEXT-FREE GRAMMARS

A context-free grammar knows only one kind of terminal symbol, but in practice, e.g. when writing compilers, there are two kinds. There is a clear distinction between *surface syntax*, i.e. the reserved words of the target language, and the *alphabet*, i.e. identifiers and literals, later on referred to as lexemes. A parser receives the former as tokens and the latter as tokens with an associated string or value. These strings or values are stored in a symbol table. Surface syntax tokens may be put in the derivation

tree, but generally they are not, because most parsers build an abstract rather than a concrete derivation tree (on which more later).
A pure structure-oriented editor also considers surface syntax to be substantially different from lexemes. Lexemes can be changed with the editor, but surface syntax cannot.

Context-free grammars do not have a notion of sequence. Sequences must be described by recursive production rules. This affects the style of interaction with the structure editor. Using grammars with sequences results in a more convenient interaction style.

Restricting a grammar in a suitable way results in a grammar that is more practical from an implementor's point of view. The LL, LR, … restrictions to make generation of parsers easier to implement is an example thereof.
Putting constraints on grammars used in structure-oriented editors makes the design of editors more clean and less complex. It makes automatic generation of grammar class definitions in an object-oriented language possible [Madsen & Nørgaard 87].

The grammar formalism defined in the previous section is too rich to be used in a structure editor environment. In [Madsen & Nørgaard 87] a description of a structured context-free grammar is given, but it is not formally defined. A structured context-free grammar imposes a certain structure on the productions of the grammar. It is inspired by the GRAMPS formalism (GRAmmar-based MetaProgramming Scheme)[Cameron & Ito 84], a structured variant of Bachus Naur Form, which is more suitable for structure-oriented environments.

In this section a structured context-free grammar is defined, and a theorem, stating that a structured context-free grammar can be found for each standard context-free grammar, is proven.

Definition 15 A **structured context-free grammar** (SCFG) is a tupel $G = (N, T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$ with

N	set of non terminal symbols
$S \in N$	start symbol
T_a	set of lexemes (alphabet)
T_s	set of tokens including ε (surface syntax)
$P_A : N \rightarrow NN^+$	set of alternation rules
$P_C : N \rightarrow T_s(NT_s)^+$	set of construction rules

$P_R : N \rightarrow T_s N T_s T_s$	set of repetition rules
$P_L : N \rightarrow T_s N T_s T_s$	set of list rules
$P_X : N \rightarrow T_a$	set of lexeme rules

$T_a \cap T_s = \varnothing$
$\forall$ P, Q $\in$ $\{P_A, P_C, P_R, P_L, P_X\} : P \cap Q = \varnothing$
$\forall$ $(A, \alpha), (B, \beta) \in P_A \cup P_C \cup P_R \cup P_L \cup P_X : A = B \Rightarrow \alpha = \beta$
$\forall$ $(A, Y_1 Y_2 \ldots Y_k) \in P_A : Y_i \neq Y_j, \forall\, i \neq j$

Two production rules have different left-hand sides.

An alternation rule $(A, Y_1\ Y_2 \ldots Y_k) \in P_A$ specifies that $Y_1, \ldots, Y_k$ are derivations of A. The non terminals in the right-hand side are mutually different.

A construction rule $(C, c_1\ Y_1\ c_2 \ldots c_k\ Y_k\ c_{k+1}) \in P_C$ specifies a *heterogeneous* aggregation of nonterminal symbols with the corresponding surface syntax.

A repetition rule $(R, b\ Y\ s\ e) \in P_R$ specifies a *homogeneous* aggregation of nonterminal symbols with the corresponding surface syntax.
The rule must be read as follows : R is a repetition of *zero or more* Ys separated by separator *s* and bracketed by begin and end delimiters *b* and *e*.

A list rule $(L, b\ Y\ s\ e) \in P_L$ is a repetition of *one or more* Ys.

A lexeme rule specifies that a syntactic construct has an associated string. The format of that string is considered a lexical issue, not a grammatical issue. Therefore it is not specified in the grammar.

In order to describe structured context-free grammars in a convenient way, the following notation is introduced.

Notation Given a SCFG $G = (N, T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$

$A \rightarrow_G Y_1 \mid Y_2 \mid \ldots \mid Y_k$	if $(A, Y_1\ Y_2 \ldots Y_k) \in P_A$
$C \rightarrow_G c_1\ Y_1\ c_2 \ldots Y_k\ c_{k+1}$	if $(C, c_1\ Y_1\ c_2 \ldots Y_k\ c_{k+1}) \in P_C$
$R \rightarrow_G b\ Y\ s^*\ e$	if $(R, b\ Y\ s\ e) \in P_R$

$L \rightarrow_G b\ Y\ s^+ e$ if $(L, b\ Y\ s\ e) \in P_L$
$X \rightarrow_G$ LEXEME if $X \in dom(P_X)$

When the grammar is clear from context the subscript G in $\rightarrow_G$ is omitted.

If the tupel $(N, T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$ is not explicitly given the start symbol is the left hand side of the first production by convention. The contents of the tupel can easily be deduced from the production rules.

A structured context-free grammar is ε free by definition. It is only chain free if it contains no alternation rules and no construction rules lacking surface syntax and having only one non terminal in the right-hand side.

One could argue that repetition rules are redundant, because repetitions could as well be described by recursive construction productions (cf. Chomsky Normal Form). However, repetition productions express repetitions in a natural fashion and they also result in a more convenient interaction in structure editors.

Definition 16 The **underlying context-free grammar of a SCFG** $G = (N, T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$ is a CFG $G' = (N', T, P, S)$ with

$$T = T_a \cup T_s$$

$$N' = N \cup \{\text{new non terminal } R' \mid R \in dom(P_R)\} \cup \{\text{new non terminal } L' \mid L \in dom(P_L)\}$$

$$\begin{aligned} P = & \bigcup_{A \rightarrow_G Y_1 \mid Y_2 \mid \ldots \mid Y_k} \{(A, Y_1), \ldots, (A, Y_k)\} \\ & \cup P_C \\ & \cup \bigcup_{R \rightarrow_G b\ Y\ s^*\ e} \{(R, b\ e), (R, b\ Y\ e), (R, b\ Y\ R'\ e), (R', s\ Y), (R', s\ Y\ R')\} \\ & \cup \bigcup_{L \rightarrow_G b\ Y\ s^+\ e} \{(L, b\ Y\ e), (L, b\ Y\ L'\ e), (L', s\ Y), (L', s\ Y\ L')\} \\ & \cup P_X \end{aligned}$$

The definition of the underlying context-free grammar specifies how the production rules of a structured context-free grammar are interpreted.

- An alternation rule $A \rightarrow Y_1 \mid Y_2 \mid \ldots \mid Y_k$ is written in standard CFGs as

  $A \rightarrow Y_1$

  …

$A \to Y_k$.

- A repetition rule $R \to b\ Y\ s^{*}\ e$ is a short hand notation for the standard (ε free) CFG production rules

  $R \to b\ e$

  $R \to b\ Y\ e$

  $R \to b\ Y\ R'\ e$

  $R' \to s\ Y$

  $R' \to s\ Y\ R'$.
- A list rule $L \to b\ Y\ s^{+}\ e$ is a short hand notation for the standard (ε free) CFG production rules

  $L \to b\ Y\ e$

  $L \to b\ Y\ L'\ e$

  $L' \to s\ Y$

  $L' \to s\ Y\ L'$.
- Construction and lexeme rules are written in the same way as in standard CFGs.

Definition 17 The **language generated by a structured context-free grammar** is the language generated by its underlying context-free grammar.

Notation Given a SCFG SG and its underlying CFG G.

$\alpha \Rightarrow_{SG} \beta$ if $\alpha \Rightarrow_{G}^{*} \beta$

$\Rightarrow_{SG}^{*}$ reflexive and transitive closure of $\Rightarrow_{SG}$

When the grammar is clear from context the subscript SG in $\Rightarrow_{SG}$ is omitted.

$\alpha \in (N \cup T_a \cup T_s)^*$ is said to be generated by SG if $S \Rightarrow_{SG}^{*} \alpha$.

A series $\omega_0 \Rightarrow \omega_1 \Rightarrow \ldots \Rightarrow \omega_n$ is called a derivation (from ω_0 to ω_n).

$L(G) = \{\omega \in (T_a \cup T_s)^* \mid S \Rightarrow^{*} \omega\}$ is the language generated by G.

G and G' are equivalent if they generate the same language.

Note that derivation of repetition and list productions is not an atomic operation.

Example Consider the part concerning statements of a grammar of a Pascal-like programming language:

statement	$\rightarrow$	ifStat | whileStat | callStat
statSequence	$\rightarrow$	statement ; *
ifStat	$\rightarrow$	IF condition THEN statSequence ELSE statSequence END
whileStat	$\rightarrow$	WHILE condition DO statSequence
callStat	$\rightarrow$	LEXEME

The *statement* rule is an alternation rule. It specifies that *ifStat*, *whileStat* and *callStat* are possible derivations of *statement*.
The *statSequence* rule is a repetition rule with little surface syntax; only the separator (a semicolon) is declared. A compound statement would look similar, the terminals BEGIN and END should only be added as begin and end delimiters.
The *ifStat* rule is a construction. It specifies that an if statement has three components: a condition (not specified in the grammar fragment) and two statement sequences.
The *callStat* rule is a lexeme rule. It specifies that a call is part of the alphabet.

Theorem 13 For any CFG G = (N, T, P, S) with $T = T_a \cup T_s$ and $T_a \cap T_s = \emptyset$ there exists a SCFG SG = (N', T_a, T_s, P_A, P_C, P_R, P_L, P_X, S) with L(G) = L(SG).

Proof follows from the fact that in standard CFGs repetition rules are described with construction rules.
Assume G to be ε free, chain free and reduced (by theorems 3 and 5).

First, the CFG G is transformed into an equivalent one from which the SCFG SG can easily be deduced.

<u>Introducing lexeme rules</u>
For every lexeme a rule is introduced and each appearance of a lexeme in the right-hand sides of the production rules is substituted by the corresponding rule.

$P_{T_a} = \{(N_a, a) \mid a \in T_a\}$ inducing $N_{T_a} = \bigcup_{a \in T_a} N_a$

$P_{LEX} = \{X \rightarrow_G \alpha a \beta \mid a \in T_a;\ \alpha, \beta \in (N \cup T)^*\}$

$P_1 = (P - P_{LEX}) \cup \{(X, \alpha N_a \beta) \mid (X, \alpha a \beta) \in P_{LEX};\ (N_a, a) \in P_{T_a}\}$

Due to theorem 4, $G_1 = (N_1, T, P_1, S)$, with $N_1 = N \cup N_{T_a}$, generates the same language as G.

Absorbing surface syntax rules

All rules with only surface syntax in the right-hand side are removed and the appearances in other rules are substituted by the surface syntax itself.

$P_{T_s} = \{(X, s) \mid s \in T_s^+\}$

$P_{SUR} = \{Y \rightarrow_{G_1} \alpha X\beta \mid (X, s) \in P_{T_s}\}$

$P_2 = (P_1 - P_{SUR}) \cup \{(Y, \alpha s\beta) \mid (Y, \alpha X\beta) \in P_{SUR}; (X, s) \in P_{T_s}\}$

Due to theorem 4, $G_2 = (N_1, T, P_2, S)$ generates the same language as G_1.

Introducing correct construction rules

All construction rules must have the correct form, i.e., each non terminal symbol in the right-hand side may be separated from another by at most one surface syntax symbol. This resembles the Operator Form for CFGs (see definition 10), but instead of no two adjacent non terminals, no two adjacent terminals are allowed.

Introducing the correct constructions can be achieved in an analogous way as theorem 11 is proven (see [Harrison 78]). The set of production rules is altered and extended, so the result of putting the rules in the desired form is a grammar $G_3 = (N_3, T, P_3, S)$.

Introducing alternation non terminals

Rules with the same left-hand side are considered. For each right-hand side a chain rule is introduced to achieve that all equal left-hand sides have a non terminal in the corresponding right-hand side.

Consider $P_P = \{Y \rightarrow_{G_3} \alpha \mid \alpha \in (N \cup T_s)^+\}$, the set of productive productions that reside in P_3.

For all $Y \in \mathrm{dom}(P_P)$ define $R_Y = \{Y \rightarrow_{G_3} \alpha\}$, the set of rules with left-hand side Y.

Number the rules in R_Y from 1 to $|R_Y|$: $Y \rightarrow \alpha_i$.

Define $Q = \{R_Y \mid Y \in \mathrm{dom}(P_P) \text{ and } |R_Y| > 1\}$.

Introduce a new non terminal symbol Y_i for each right-hand side α_i, remove all rules with left-hand side Y and introduce rules $Y \rightarrow Y_i$.

This procedure introduces chain rules :

$P_4 = (P_3 - \bigcup_{R \in Q} R) \cup \bigcup_{R \in Q} \bigcup_{i \leq |R|} \{Y_i \rightarrow \alpha_i, Y \rightarrow Y_i\}$

Due to theorem 4, $G_4 = (N_4, T, P_4, S)$, with $N_4 = dom(P_4)$, generates the same language as G_3.

G_4 is the CFG from which SG can be derived as follows.

Defining the SCFG

Define all parts of SCFG SG according to the definition of a SCFG using G_4 as the underlying context-free grammar.

$P_X = \{(Y, a) \mid Y \rightarrow_{G_4} a\}$

$P_C = \{(Y, \alpha) \mid Y \rightarrow_{G_4} \alpha, \alpha \in (N \cup T_s)^+\}$

$P_A = \{(Y, Y_1Y_2 \ldots Y_k) \mid Y \rightarrow_{G_4} Y_i, Y_i \in N_4\}$

$P_R = \emptyset$

$P_L = \emptyset$

$N' = N_4$

Since, by transitivity, the language generated by G_4 is equal to the language generated by the original grammar G, SG is a structured context-free grammar for G with L(SG) = L(G).

Note that repetition and list rules are completely ignored in the construction, because, as said before, they are in fact redundant : Repetitions can be described by a combination of constructions and alternations.

Consequently, this theorem is not constructive, as is often the case with grammar rearranging theorems. Using this theorem's technique to construct a SCFG out of a CFG does not result in a "good" structured context-free grammar. "Good" means here that CFG production rules constituting repetitions and lists are found and combined in SCFG repetition and list rules. Whether an efficient algorithm for "good" structuring a grammar exists, should be further investigated.

3.2 Structured Abstract Grammars

The abstract grammar defined in the previous chapter is too general to be used in a structure editor environment. It does not reflect the intuitive structure of a language. The generality does not allow clear and clean design of structure-oriented editing operations. By restricting abstract grammars, implementation becomes less complex.

From the definition of a structured context-free grammar a definition of structured abstract grammars can be derived as follows.

Definition 18 A **structured abstract grammar** (SAG) is a tupel $G = (N, T_a, P_A, P_C, P_R, P_L, P_X, S)$ with

N	set of non terminal symbols
$S \in N$	start symbol
T_a	set of lexemes (alphabet)
$P_A : N \rightarrow NN^+$	set of alternation rules
$P_C : N \rightarrow N^+$	set of construction rules
$P_R : N \rightarrow N$	set of repetition rules
$P_L : N \rightarrow N$	set of list rules
$P_X : N \rightarrow T_a$	set of lexeme rules

$\forall\, P, Q \in \{P_A, P_C, P_R, P_L, P_X\} : P \cap Q = \varnothing$

$\forall\, (A, \alpha), (B, \beta) \in P_A \cup P_C \cup P_R \cup P_L \cup P_X : A = B \Rightarrow \alpha = \beta$

$\forall\, (A, Y_1Y_2 \ldots Y_k) \in P_A : Y_i \neq Y_j,\ \forall\, i \neq j$

[Minör 90] gives an overview of alternative formalisms with approximately the same expressive power. Also see [Cameron & Ito 84]. In [Centaur 92] and [Hendriks 91] a formalism combining abstract and concrete syntax is presented.

Notation

$A \rightarrow_G Y_1 \mid Y_2 \mid \ldots \mid Y_k$	if $(A, Y_1 \ldots Y_k) \in P_A$
$C \rightarrow_G Y_1\ Y_2\ \ldots\ Y_k$	if $(C, Y_1 \ldots Y_k) \in P_C$
$R \rightarrow_G Y^*$	if $(R, Y) \in P_R$
$L \rightarrow_G Y^+$	if $(R, Y) \in P_L$
$X \rightarrow_G$ LEXEME	if $X \in dom(P_X)$

When the grammar is clear from context the subscript G in $\rightarrow_G$ is omitted.

When the tupel $(N, T_a, P_A, P_C, P_R, P_L, P_X, S)$ is not explicitly given the start symbol is the left-hand side of the first production by convention.

It is obvious that a structured abstract grammar can be found for each structured context-free grammar. But, as mentioned before, a context-free grammar contains rules for solving precedence and ambiguities. Consequently, only dropping the surface syntax from the SCFG is not sufficient to obtain a well designed SAG. Some hand crafting is still needed. Usually the mapping from concrete to abstract syntax is hard coded in a parser. Employing SCFGs and SAGs for table driven parsing is certainly a topic for further research.

Given a structured context-free grammar the corresponding structured abstract grammar is likely to look substantially different, as the following example shows. The context-free grammar is designed to reflect the precedence of unary, operator and keyword messages. The abstract grammar reflects the basic language structure. Each message consists of a receiver (an expression) and a pattern (unary, operator or keyword).

Example Agora's structured context-free grammar (simplified version)

Message	→ Operation KeywordPattern
Operation	→ Unary OperatorPattern
Unary	→ Factor | UnaryPair
UnaryPair	→ Unary Name
OperatorPattern	→ OperatorPair *
OperatorPair	→ Operator Operation
KeywordPattern	→ KeywordPair *
KeywordPair	→ Keyword : Operation
Factor	→ Block | Name | Submessage | Literal
Block	→ [Message ; *]
Submessage	→ (Message)
Keyword	→ LEXEME
Operator	→ LEXEME
Name	→ LEXEME
Literal	→ StringLiteral | IntegerLiteral
StringLiteral	→ LEXEME
IntegerLiteral	→ LEXEME

Agora's structured abstract grammar (simplified version)

Expression	→ Block | Message | Literal | UnaryPattern
Block	→ Expression *
Message	→ Expression Pattern
Pattern	→ UnaryPattern | OperatorPattern | KeywordPattern
UnaryPattern	→ LEXEME
OperatorPattern	→ Operator Expression
KeywordPattern	→ Pair +
Pair	→ Keyword Expression
Keyword	→ LEXEME
Operator	→ LEXEME
Literal	→ StringLiteral | IntegerLiteral

StringLiteral	$\rightarrow$ LEXEME
IntegerLiteral	$\rightarrow$ LEXEME

Definition 19 Given a SAG $G = (N, T_a, P_A, P_C, P_R, P_L, P_X, S)$.
$X \in N$ is an **alternative** of $Y \in N$, notation $X \downarrow Y$,

if $Y \in \mathrm{dom}(P_A)$
and $Y \rightarrow_G Y_1 \mid \ldots \mid X \mid \ldots \mid Y_k$ or $\exists\, i \in [1, k] : X \downarrow Y_i$

or $Y \notin \mathrm{dom}(P_A)$
and $Y = X$

Definition 20 An **abstract derivation tree (ADT) of a SAG** $G = (N, T_a, P_A, P_C, P_R, P_L, P_X, S)$ is an ordered and labeled tree such that

- the label of the root node is an alternative of S
- every node has a label $l \in N - \mathrm{dom}(P_A)$
- if a node has label Y and the descendents of that node are labeled from left to right with $Y_1, \ldots, Y_k$, then one of the following conditions holds:

$Y \rightarrow_G X_1 \ldots X_k$	and $k \geq 1$, $\forall\, i \in [1, k] : Y_i \downarrow X_i$
$Y \rightarrow_G X^*$	and $k \geq 0$, $\forall\, i \in [1, k] : Y_i \downarrow X$
$Y \rightarrow_G X^+$	and $k \geq 1$, $\forall\, i \in [1, k] : Y_i \downarrow X$
$Y \rightarrow_G$ LEXEME	and $k = 0$

- every leaf node has an attribute named 'text', the lexeme text

A **partial abstract derivation tree** of G is an abstract derivation tree of which the root node is labeled by any non terminal rather than an alternative of S.

Nodes having a construction, repetition, list or lexeme label are called construction nodes, repetition nodes, list nodes and lexeme nodes respectively.
As expected, alternation productions have no counterpart in the derivation tree. They are consulted during program synthesis and the definition of placeholders, on which more later. Lexeme nodes have an attribute (traditionally stored in a symbol table) that represents the lexeme text.
Notice that an abstract derivation tree does not contain nodes for intermediate derivations, that is, chain production rules in the grammar have no counterpart in the derivation tree, only *productive* production rules have corresponding nodes in the tree.

3.3 STRUCTURED ATTRIBUTE GRAMMARS

Since we employ structured abstract grammars rather than standard abstract grammars in the remainder of this document, a way of attributing a SAG is needed. A definition suitable for our needs in subsequent chapters is given here.

Definition 21 A **structured attribute grammar** (SATG) is a tupel AtG = (G, A, R) with G = $(N, T_a, P_A, P_C, P_R, P_L, P_X, S)$ a structured abstract grammar and

$A = \bigcup_{X \in N} A(X)$ with for $X \in N$,

$A(X)$ is an finite set of attributes

$A(X) = S(X) \cup I(X)$, $S(X) \cap I(X) = \emptyset$

$R = \bigcup_{p \in P} R(p)$ with R(p) a finite set of attribution rules on production rule p. These rules define synthesized attributes of left-hand side non terminals and inherited attributes of right-hand side non terminals as follows, f taking on any of a specified set of values :

Alternation rules $A \rightarrow_G Y_1 \mid Y_2 \mid ... \mid Y_k$ have no attribution rules.

Lexeme rules $X \rightarrow_G$ LEXEME have only attribution rules of the form

$X.s \leftarrow f(X.text, X.i)$, with $s \in S(X)$, $i \in I(X)$.

Repetition and list rules $R \rightarrow_G Y^*$ and $R \rightarrow_G Y^+$ have only attribution rules of the form

$R.s \leftarrow f(Y_1.t, \ldots, Y_{R.size}.t, R.size, R.i)$

$Y_k.j \leftarrow f(Y_1.s_1, \ldots, Y_{k-1}.s_{k-1}, k, R.size, R.l)$

with $s \in S(R)$; $i, l \in I(R)$; $t, s_1, \ldots, s_{k-1} \in S(Y)$; $j \in I(Y)$.

Construction rules $C \rightarrow_G Y_1\ Y_2\ ...\ Y_k$ have only attribution rules of the form

$C.s \leftarrow f(Y_1.s_1, Y_2.s_2, \ldots, Y_k.s_k, C.i)$

$Y_n.j_n \leftarrow f(Y_1.s_1, \ldots, Y_{n-1}.s_{n-1}, C.i)$

with $s \in S(C)$, $i \in I(C)$;

$\forall\ m \in [1, k] : s_m \in S(Y_m)$, $j_m \in I(Y_m)$.

S does not have inherited attributes.
Lexeme non terminals have an intrinsic 'text' attribute.

Alternation productions $A \rightarrow_G Y_1 \mid Y_2 \mid ... \mid Y_k$ do not have attribution rules, because they have no counterpart in the abstract derivation tree. Conceptually, the left-hand side A and all right-hand side derivations Y_i have the same synthesized and inherited attributes:

$$A.s \leftarrow Y_j.s \text{ and } A.i \leftarrow Y_j.i$$
$$\text{with } s \in S(A) \cap \bigcap_{j \in [1,k]} S(Y_j),\ i \in I(A) \cap \bigcap_{j \in [1,k]} I(Y_j).$$

For notational convenience $Y.j \leftarrow f(Y_1.s_1, \ldots, Y_{k-1}.s_{k-1}, k, R.size, R.l)$ is used to lay down attribution rules which apply to all descendents of a repetion or list production (see the example below).
For notational convenience $Y.j \leftarrow f(Y_1.s_1, \ldots, Y_{n-1}.s_{n-1}, C.i)$ is used to lay down attribution rules which apply to all descendents of a construction production

When a construction production contains a non terminal more than once, indices may be used in the attribution rules to disambiguate the non terminals.

A SATG is in Bochmann Normal Form by definition. Due to the constraints on the attribution rules, a SATG is a class L attribute grammar. The attributes can be evaluated with a left to right depth first tree traversal algorithm.

Example Consider a structured abstract grammar with production rules

Bit	→ Zero | One
Zero	→ LEXEME
One	→ LEXEME
BitString	→ Bit $^+$

This grammar generates bit strings. Suppose the bitstrings represent positive integer numbers with the least significant bit on the left and suppose one wants to compute their value. The following structured attribute grammar attributes the production rules above to achieve this goal.

Bit	→ Zero | One	
Zero	→ LEXEME	Zero.val ← 0
One	→ LEXEME	Zero.val ← $2^{One.pos}$
BitString	→ Bit $^+$	Bit.pos ← i - 1

$$\text{BitString.val} \leftarrow \sum_{i=1}^{\text{BitString.size}} \text{Bit}_i.\text{val}$$

The non terminals *Zero* and *One* have an inherited attribute *pos*, which denotes the position of a bit in a bitstring. *Zero*, *One* and *BitString* have a synthesized attribute *val*, which contains the value after evaluation of the attribution rules.
In the attribution rule $\text{Bit}_i.\text{pos} \leftarrow i$ the index i is not specified. The rule applies to all *Bits* in a *BitString*.

3.4 Program Representation

The structured abstract grammar for a target language describes the internal representation of programs written in that language. A program is represented by an abstract derivation tree. Each node of the derivation tree is labeled with the name of the production rule it is an instance of. Each node has a number of descendents (possibly none). All nodes have the same structure, i.e. they all are represented by n-ary nodes.

Notation	N	any program tree node
	R	the program tree root
	$N[N_1, \ldots, N_p]$	N with p subnodes $N_1, \ldots, N_p$
	N[]	N having no subnodes
	#N	size of N, i.e. $\#N[N_1, \ldots, N_p] = p$, $\#N[] = 0$
	∇N	label of N
	N[i]	the i th subnode of N
	N[i][j]	the j th subnode of N[i] for $N[N_1, \ldots, N_i[M_1, \ldots, M_j, \ldots, M_q], \ldots, N_p]$, $N[i][j] = M_j$
	$M \in N$	M is a subnode of N for $[N_1, \ldots, N_p]$, $\exists\, i \in [1, p] : M = N_i$
	N.A	attribute A of N
	N.A := (B, C, D)	definition of N.A having 3 components B, C and D
	N.A.B	component B of N.A

3.5 Summary

In this chapter structured grammars were introduced. *Structured context-free grammars* are the structured variant of the standard context-free grammars generally used for the formal specification of programming languages. From structured context-free grammars, *structured abstract grammars* were derived. Structured abstract grammars are the foundation for *abstract derivation trees*, the internal representation of programs used in the remainder of this document. Structured abstract grammars can be decorated with attribution rules by means of *structured attribute grammars*, a structured variant of standard attribute grammars used for mixing syntax and semantics. Structured attribute grammars are used in the chapter five to describe program presentations, i.e. how programs, specified in the structured grammar formalism, are presented textually.

Four

Structured Grammar Extensions For Editing

In this chapter extensions of, and additions to structured abstract grammars are given.

Since synthesizing programs with a structure editor is done top-down, placeholders are needed to mark descendents of syntactical constructs to be elaborated further. Given a target language, the designer of a structure editor needs to determine which placeholders are necessary to support program synthesis. With an editor providing too little placeholders, probably no complete programs can be synthesized. With an editor providing too many placeholders, many redundant editing steps (mainly placeholder expansions) are required for program synthesis.
Deriving placeholders from a target language's grammar is not handled in the literature. Here, derivation of placeholders from a structured abstract grammar is discussed.

To support text editing in structure editors the *text placeholder* is introduced. It represents a piece of non parsed program text in the internal representation of a program.
Instances of production rules are presented to act as arbitrary replacements or insertions for replace and insert commands.

A structure-oriented editor performs structure-oriented commands on structural selections. A structural selection is a selection of a grammatical construct, i.e. a subtree of the abstract grammar tree. In this chapter the *tree path* is introduced to represent structural selections.

A *template* contains all information about a production rule of the target language's grammar. Together with placeholders, they are the basis for program synthesis. What templates consist of and how they are consulted during editing, is presented.

4.1 Deriving Placeholders From a SAG

There are two ways of deriving placeholders from an structured abstract grammar:

- introduce a placeholder for each non terminal
- introduce a placeholder for each alternation and lexeme non terminal

Introducing a placeholder for each non terminal of a structured abstract grammar has some undesirable side effects considering the usage of a structure editor.
If the start production is a construction production, a newly openend editor presents a placeholder that can be expanded to the construction's right-hand side only. For example, an editor for Pascal presents a 'program' placeholder, which must be expanded by the user. This problem can be generalized: Inserting a syntactical construct having construction, repetition or list descendents into a program requires expansion of the placeholders for the descendents, although the expansion is fixed.

Obviously, this is unwanted. Expansion of construction, repetition and list placeholders is redundant, because the placeholders have only one possible expansion. The editor should expand construction, repetition and list descendents automatically to release the user from unnecessary manual placeholder expansions.

These objections do not hold for alternation productions, nor for lexeme productions. An alternation rule specifies that a non terminal has several possible derivations. Clearly, a placeholder for an alternation is needed to let the editor present the user a choice of possible expansions. Lexeme productions specify that an element of the alphabet should be given. Since a lexeme generally has many possible expansions, a placeholder is needed to denote a lexeme within a grammatical construct.

So, placeholders need only be introduced for alternation and lexeme non terminals.

Example From the Agora grammar (simplified version) given in chapter three, following placeholders are derived (given with their category):

Expression	alternation
Pattern	alternation
UnaryPattern	lexeme
Keyword	lexeme
Operator	lexeme
Literal	alternation

StringLiteral	lexeme
IntegerLiteral	lexeme

4.2 PLACEHOLDERS IN ABSTRACT SYNTAX TREES

In order to store placeholders in the internal representation of a program, the definition of an abstract derivation tree must be extended. Instead of modifying the definition given in chapter three, an internal representation including placeholders is defined in such a way that an abstract derivation tree is a particular case thereof.

Definition 22 An **abstract syntax tree (AST) of a SAG** G = (N, T_a, P_A, P_C, P_R, P_L, P_X, S) is an ordered and labeled tree such that

- every node has a label $l \in (N - dom(P_A)) \cup \{\Pi\}$
  Nodes labeled Π are called placeholder nodes
- placeholder nodes have two attributes, one named 'type', a label in $dom(P_A) \cup dom(P_X)$ and one named 'name', a character string
- define the type of a tree node M as
  $\Delta M = \nabla M$ if $\nabla M \in N - dom(P_A)$
  $\Delta M = M.type$ if $\nabla M = \Pi$
- if a node has label $Y \in N - dom(P_A)$ and the descendents of that node are labeled from left to right with $Y_1, \ldots, Y_k$, then one of the following conditions holds:
  $Y \rightarrow_G X_1 \ldots X_k$ and $k \geq 1, \forall i \in [1, k] : \Delta Y_i \downarrow X_i$
  $Y \rightarrow_G X^*$ and $k \geq 0, \forall i \in [1, k] : \Delta Y_i \downarrow X$
  $Y \rightarrow_G X^+$ and $k \geq 1, \forall i \in [1, k] : \Delta Y_i \downarrow X$
  $Y \rightarrow_G$ LEXEME and $k = 0$
- lexeme nodes have an attribute named 'text', the lexeme text
- the label of the root node is an alternative of S

A **partial abstract syntax tree** of G is an abstract syntax tree of which the root node is labeled by any non terminal of G, rather than an alternative of S.

The Δ operator defines the *type of a syntax tree node*. The type of a placeholder node is the non terminal for which the placeholder reserves a place. For a non placeholder node the type is its label. The Δ operator thus determines which production rule the node is an instance of.

The definition lays down that a placeholder of type T may appear where a node with label L is expected, as long as T is an alternative of L. An abstract derivation tree is thus a particular case of an abstract syntax tree. One can state that an ADT is completely synthesized AST.
Abstract syntax trees are often called syntax trees for short.

4.3 Text Fragments in Abstract Syntax Trees

A text fragment is not that different from a placeholder. They both reserve a space for a correct program fragment. Therefore we will introduce the *text placeholder*, marking a place in an abstract syntax tree that is not yet parsed. A text placeholder has an intrinsic text attribute denoting the edited text. This ensures that an AST containing text placeholders can be unparsed.

When a textual editing operation is initiated, the editor has to determine which partial AST corresponds with the current text selection. If the current text selection denotes a structural selection, determination is trivial. If the current text selection, on the other hand, denotes a selection over more than one partial AST, the editor has to determine the smallest partial AST enclosing the subtrees in which the beginning and the end of the text selection is located.

Having determined the subtree containing the current text selection, the editor has to check whether it is appropriate for text editing. It is not appropriate if it cannot be parsed without context. In that case the editor has to look for a more appropriate enclosing program fragment.

There is another reason why the partial AST to be text edited should be carefully chosen. The user can change the contents of a text edited piece of the program in such a way that its grammatical category changes. When this happens it is likely that the user wants to change the meaning of an enclosing grammatical construct, rather than that of the selected one.

An approach exists to solve the two problems. Given a textual selection, the editor determines the smallest enclosing subtree that is a component of a repetiton or list node with a separator and begin and end delimiters in its textual representation.
The textual representation of repetition or list descendents is independent of the textual representation of the enclosing node. Text editing a descendent never causes the enclosing node to change of grammatical category, except when the enclosing repetition or list does not have a separator and begin and end delimiter.

Example The Agora keyword pattern is defined by the production rule

KeywordPattern $\rightarrow$ Pair $^{+}$.

The keyword pattern has no separator and no begin and end delimiter in its textual representation, as seen in the example pattern

at: 1 put: object.

Consider an Agora keyword message having a one-pair keyword pattern

vector at: index

Suppose the pair 'at: index' is changed by text editing into 'at index' (the colon is dropped). If the editor uses a text placeholder for the pair, then the text placeholder stays in the AST, because 'at index' cannot be parsed to a program fragment that is allowed in a KeywordPattern.

Since a KeywordPattern has no associated surface syntax, the textual change should result in cascaded unary messages

vector at index

4.4 INSTANCES OF PRODUCTION RULES

Given the definition of an abstract syntax tree, instances of production rules can be defined. An instance of a production rule is a, possibly incomplete, derivation of that rule. Instances are needed by the editor for use with program synthesis commands and for consultation to check applicability of commands.

As mentioned in chapter four, an editor in our design supports replacement of any grammatical construct by another of the same type. Therefore two kinds of instances can be distinguished: *default instances*, incarnations of the production rules, and *custom instances*, arbitrary derivations of production rules. During interaction with the editor, the user can install custom instances (see chapter six) to be used later on.

A default instance of a production rule is represented by an abstract syntax tree of which the descendents are filled in properly: All alternation and lexeme descendents are placeholders, all other descendents are instances of the corresponding production rules. Instances of production rules having no descendents are placeholders. The following table shows the instances for each production rule type.

Production Rule	**Default Instance**	
$Y \rightarrow X_1 \mid \ldots \mid X_k$	N[]	with $\nabla N = \Pi$, N.type = Y, N.name = Y
$Y \rightarrow$ LEXEME	N[]	with $\nabla N = \Pi$, N.type = Y, N.name = Y
$Y \rightarrow X_1 \ldots X_k$	$N[D_1, \ldots, D_k]$	with $\nabla N = Y$, $\forall$ i : D_i is def. inst. of X_i
$Y \rightarrow X^*$	N[D]	with $\nabla N = Y$, $\Delta D = X$
$Y \rightarrow X^+$	N[D]	with $\nabla N = Y$, $\Delta D = X$

Example Given Agora's structured abstract grammar (simplified version) given in section 3.2, the following are some default and custom instances.

Non Terminal	**Default Instance**	**Custom Instance**
Message	*Expression Pattern*	*Expression* + 5
		vector at: 3 put: y
Pair	*Keyword* : *Expression*	at: 1
		do: *Expression*
Expression	*Expression*	*Expression*
IntegerLiteral	*IntegerLiteral*	12345
Block	[*Expression*]	[]
		[(3 + x) print]

The set of default instances is in fact the editor's internal representation of the production rules of the target language's grammar. It consults this set during editing operations, as discussed later.

4.5 Selection in Abstract Syntax Trees

Structure-oriented commands are only applicable on structural selections. A structural selection is the selection of a subtree of the abstract syntax tree.
An editor manages two "pointers" which denote the selection range. The first points to the beginning and the second points to the end of the structural selection range. If they both point to the same subtree, a *single selection* is made, otherwise a *multiple selection* is made.

In [De Hondt 90] the *tree path* was introduced to act as the "pointers" above. A tree path represents a path in a syntax tree from a node (usually the root node) to a reachable underlying node.
Since then the tree path has matured, as laid down in the definition that follows.

Tree paths are used extensively during editing operations. Tree paths point to the current structural selection, are changed during structure-oriented editing steps, are passed to the unparser to generate partial textual representations of the syntax tree, are modified during search operations, and so on. The tree path is the basic editor component to identify structural selections with, and as such, they are indispensable.

Definition $TP(N) = (I_1, \ldots, I_p)$ is the (**relative**) **tree path of a tree** N to a node M where

$$M = N[I_1][I_2]\ldots[I_p]$$

$TP(N) = ()$ is the empty tree path of a tree N. It points to N.

LEN(TP(N)) is the length of a tree path.

$$TP(N) = () \text{ then } LEN(TP(N)) = 0$$
$$TP(N) = (I_1, \ldots, I_p),\ p \geq 1 \text{ then } LEN(TP(N)) = p$$

$ATP = (N_1, \ldots, N_p)$ is an **absolute tree path of a tree** N_1 to node N_p when

$$p \geq 1 \text{ and } \forall\, i \in [2, p] : N_i \in N_{i-1}.$$

$TP(N) = (I_1, \ldots, I_p)$

ATP(TP(N)) is the **absolute tree path corresponding to a tree path** TP(N), i.e.

$ATP(TP(N)) = (N_0, N_1, \ldots, N_p)$ with

$$N_0 = N$$
$$N_k = N_{k-1}[I_k],\ \forall\, k \in [1, p]$$

$TP(N) = (I_1, \ldots, I_p)$

$TP(N) + I$ is the tree path $TP'(N) = (I_1, \ldots, I_p, I)$

$TP_1(N) = (I_1, \ldots, I_p)$, $TP_2(M) = (J_1, \ldots, J_q)$ then

$$TP_1(N) \supseteq TP_2(M) \text{ iff } q \leq p \text{ and } \forall\, k \in [1, q] : J_k = I_k$$
$$TP_1(N) \supset TP_2(M) \text{ iff } q < p \text{ and } \forall\, k \in [1, q] : J_k = I_k$$
$$TP_1(N) = TP_2(M) \text{ iff } q = p \text{ and } \forall\, k \in [1, q] : J_k = I_k$$

$ATP_1 = (N_1, \ldots, N_p)$, $ATP_2 = (M_1, \ldots, M_q)$ then

$$ATP_1 \supseteq ATP_2 \text{ iff } q \leq p \text{ and } \forall\, i \in [1, q] : M_i = N_i$$
$$ATP_1 \supset ATP_2 \text{ iff } q < p \text{ and } \forall\, i \in [1, q] : M_i = N_i$$
$$ATP_1 = ATP_2 \text{ iff } q = p \text{ and } \forall\, i \in [1, q] : M_i = N_i$$

In an implementation a tree path is nothing more than a traditional stack with added functionality. It contains relative references (indices) to nodes. The absolute path is a cache which is invalidated whenever a relative reference or the path's syntax tree changes.

4.6 TEMPLATES

A template is a description of all information of a production rule needed by the structure-oriented editor. The editor uses templates extensively for display and editing purposes.

A template contains

- the production rule itself, represented by its default instance
- associated surface syntax
- layout information
- the initial selection when the template is inserted in the program
- selection information for clicks on the surface syntax
- set of custom instances of the production rule, installed by the user

Templates of alternation production rules only contain the last item, the other are not relevant. We will assume that an alternation rules' set of custom instances is the union of the sets of custom instances of the alternation's alternative derivations.

The associated surface syntax is an ordered collection of terminals from the language's grammar.
The layout information is a presentation scheme, as defined in chapter five.
The initial selection is a tree path from the root of the production rule tree to an arbitrary node in that tree. When an editor puts a template in the syntax tree, it extends the current selection path, pointing to the template being put, with the initial selection path. The selection displayed after the operation will be the one specified in the template.
The selection information is an ordered collection of tree paths, indicating which structural selection should be made when an indirect hot spot click happens (see chapter six).

Since all left-hand sides of the production rules of a SAG are unique, there is a one-to-one relation between non terminal symbols and templates. Given a syntax tree node, its template can thus be determined by means of its label.

4.6.1 CONSULTING TEMPLATES DURING EDITING

An editor has to check what structure-oriented commands are applicable at a given time. The set of applicable commands depends on the current structural selection. For the editor, not the type of a selected syntax tree node, but its base type is important.

Definition Given a SAG G = (N, T_a, P_A, P_C, P_R, P_L, P_X, S).
The **base type of a syntax tree node** N_i in M[N_1, …, N_i, …, N_k] is the ith non terminal of the production rule with left-hand side ∇M (the label of M).
The base type of the root node of a syntax tree is S.

In other words, given a node N at the ith place of a node M, the base type of N is the label of the ith descendent of the default instance of the production rule having the label of M as left-hand side. Or rephrased, the base type of a node is the non terminal it is originally synthesized from.

Example Consider Agora's structured abstract grammar (simplified version) given in section 3.2(or the full version given in the appendix).

The lexeme '1' in the block '[1]' has base type Expression, since a Block is a repetition of Expressions. Its type is IntegerLiteral.

The keyword/argument pair 'at: 5' in the message 'vector at: 5' has base type Pair, since the pattern of the message is a KeywordPattern, a list of Pairs. Its type is also Pair.

The placeholder '*StringLiteral'* in the message '*StringLiteral* print' has base type Expression, since the receiver of a Message is an Expression. Its type is StringLiteral.

The base type of a structural selection determines how the selection can be manipulated.

The replace command can be applied whenever a structural selection is made. The possible set of replacements is the set of custom instances found in the template corresponding with the base type of the selected syntax tree node. For example, if the receiver of an Agora message 'vector at: 5' is selected, the editor will display all custom instances of Expression productions, since the receiver has base type Expression.

An insert command can be applied only if a descendent of a repetiton or list rule is selected. The base type of the selection determines what the set of possible insertions is.

An unexpand command can be applied to any structural selection. The unexpansion is the default instance of the production rule with the base type of the selection as left-hand side. For example, unexpanding the receiver of the Agora message 'vector at: 5' results in '*Expression* at: 5', while unexpanding the pair results in 'vector *Keyword*: *Expression*'.

Replacing, inserting and unexpanding are two-stage operations. First the base type of the current selected syntax tree node is determined by consulting the template of the father node, then the template of that base type's production rule is consulted in order to complete the command.

4.6.2 Consulting Templates During Unparsing

An unparser (see chapter five) needs the surface syntax and the layout information of a tree node to generate its textual representation correctly. In an implementation the unparser is distributed over the different classes of syntax tree nodes, i.e. lexeme, construction, repetition and list node classes each have their own unparser. The node unparsers have direct access to a node's template through an instance variable of the object representing the tree node.

4.7 Pattern Matching Syntax Trees

Pattern matching of abstract grammar trees is used by several functional components of a structure-oriented editor. The transformation operation, mentioned in chapter one and described in detail in chapter six, uses pattern matching to check whether a structural transformation is applicable to a partial syntax tree. Searching is also implemented by means of pattern matching. If a search string can be parsed, searching can be conducted in a structural fashion rather than a textual one.
Agora is used in the examples of this section. Consult the appendix when not familiar with Agora's concrete and abstract grammar.

4.7.1 Structural Patterns

A structural pattern (pattern for short) is represented by an abstract syntax tree. Placeholders in the tree act as wildcards for the pattern.

Example "*Expression* **Method**: *Expression*" is a pattern. It matches every keyword message only having a keyword 'Method:'.

4.7.2 Structural Pattern Matching

Structural pattern matching is the act of comparing a structural pattern with an abstract syntax tree. The result of the match is a collection of matches <placeholder name, subtree> and an indication whether the pattern match succeeded or not. We will call subtrees of the pattern subpatterns.

A syntax tree T matches a pattern P when the following conditions are met (lexeme() and rule() are predicates indicating whether a syntax tree node is a lexeme node or a construction, repetition or list node, respectively) :

MATCH	T no placeholder	T placeholder
P no placeholder	$\nabla T = \nabla P$ and $\#T = \#P$ and ((lexeme(T) and T.text = P.text) or (rule(T) and $\forall\ i \in [1,\#T]$: match(T_i, P_i))	false
P placeholder	T is instantiation of P	T is an instance of P

Example Consider the pattern P ="*Expression* **Method**: [turtle go]".

"startTurtle **Method**: [turtle go]" matches P
"*Literal* **Method**: [turtle go]" matches P
"startTurtle **Method**: *Expression*" does not match P

Structural pattern matching can be combined with textual pattern matching of lexeme texts to obtain advanced pattern matching. Placeholders denote wildcards for structural matching, metacharacters denote wildcards for textual matching.

In the following example it is assumed that # is the one-character wild card and % is the many-character wildcard.

Example In a program using turtle graphics one finds several names for turtle objects, such as aTurtle, thisTurtle, thatTurtle. To find Agora messages sent to such objects one could enter the following pattern when applying a search operation :

"%turtle% *pattern*"

The following pattern can be used to find all binary one-character messages (so called operators):

"*Expression # Expression*"

In the Smalltalk environment [Goldberg 84b] the metacharacters # and * are used for matching of one, respectively more characters. Smalltalk uses textual pattern matching for search of class or method names, which may not contain # and * characters. Lexical constraints of lexemes depend on the target language used. Thus, to keep a search facility open-ended, the meta-characters should be adjustable. This open-endedness also affects the parser in a hybrid editing environment since it has to support parsing of lexemes containing wildcard characters.

4.8 SUMMARY

In this chapter extensions of, and additions to structured abstract grammars for structure-oriented editors were presented and technical problems were discussed. How *placeholders* are derived from a structured abstract grammar was described. Fitting placeholders in abstract derivation trees was done by introducing the *abstract syntax tree*, which is the internal representation of programs used from now on. *Default* and *custom instances of production rules* and *templates* were introduced and their significance for editing was discussed. Instances of production rules are syntax trees representing derivations of production rules. A template contains all information about a production rule needed by the editor. The introduced *tree path* is used for selection purposes; it points to a structural selection in an abstract syntax tree. Structural pattern matching was briefly handled. It is important for structural transformation introduced in chapter one and discussed in the chapter six.

Five

PRESENTATION OF PROGRAMS

In the following of this document we will suppose that programs are presented as text. A discussion of other kinds of presentation can be found in [Minör 90]. In this chapter textual presentation of programs is handled.

Presentation schemes are used to generate a textual representation of an internal representation of a program (with the *format* command in Smalltalk-80 for example [Goldberg 84b]), as opposed to *pretty printing* used to make a textual representation more readable by rearranging it (with the *format* command in Scheme [Abelson & Sussman 85] environments for example).

In the literature 'pretty printing' is used both for program presentation and for pretty printing as we call it. Several approaches to 'pretty printing' are reported on, e.g. in [Oppen 80], [Rubin 83], [Leavens 84], [Blaschek & Sametinger 89], [Jokinen 89]. Textual representations are generated from an abstract derivation tree by an *unparser*. Generally, an unparser implements a simple depth first tree traversal algorithm, but an algorithm with more passes over the derivation tree is more powerful. Presentation of a program fragment can then depend on the presentation of its subfragments. It is clear that such an algorithm requires more time and more space.

In this chapter a description of a fairly simple formalism for program presentation is given. The formalism is engrafted on structured abstract grammars, defined in chapter three.

In [Minör 90] five characteristics of schemes for textual presentation are given : *flexibility* is the ability for the user to influence the layout of the structure and the ability to modify the syntactic sugar of the language. *Context-sensitiveness* describes to what extent the context may determine the presentation format. Predicates determining the presentation format are used to allow *conditional presentation*, which is more flexible than context-sensitive presentation. *Adaptive presentation* is used to adapt the format to the available window size. *Elision* (or folding) enables the user to suppress details of a program to get a more abstract view. *Multiple views* enable presentation of a program in various ways.

Folding and multiple views are addressed later in this chapter, while the other are commented on now.
It is commonly accepted that the user must be able to control the layout of his programs[3] (see chapter four and six), but it is questionable if allowing changes to the syntactic sugar of a language is desirable for a one-language programming environment. Only generic environments need this feature to support incremental language development. This technique is used in the Mjølner Orm environment [Mjølner 91][Magnusson & al. 90]: even when a structure editor for the target language is open, changes to the grammar can be made (this is a consequence of the grammar interpretive approach used in Orm).
Adaptive presentation is obsolete in modern window systems allowing scrolling and can, in many cases, be handled by conditional presentation.

In our design a structure editor displays a textual representation of a program. Through this textual interface the user interacts with the editor. Every operation issued by the user changes the program tree and, as a consequence, the displayed program text.
To support selection of program fragments (i.e. subtrees of the program's syntax tree) to issue operations on, a mapping between the internal and the external representation of the program must exist. Every operation does not only affect the two representations, but also the mapping between them. A good mapping is one that can be updated efficiently.
In this chapter we will present a mapping that can be updated in O(n) time, n being the nesting level of the edited program tree node.

5.1 A Simple Presentation Scheme

A presentation scheme is the information needed to layout a program fragment. Instead of allowing a presentation scheme per production rule, a presentation scheme per kind of production rule is adopted. Since alternation production rules have no counterpart in the abstract derivation tree, only four kinds of presentation schemes must be considered.

---

[3]Although little programming environments support this feature !

Later on we will attribute the production rules of a structured abstract grammar by means of a structured attribute grammar to specify how instances of those rules (i.e. abstract derivation tree nodes) should be presented textually.

For convenience the surface syntax is also recorded in the presentation schemes, but, in essence, this is unnecessary because the surface syntax is fixed. It is determined by the concrete grammar and is recorded in the templates, presented in the previous chapter.

The layout characters are carriage returns and spaces, collectively called *blanks*. The term *token* is used to represent the textual representation of surface syntax tokens.

5.1.1 REPETITION PRESENTATION

A repetition rule R $\rightarrow$ N * has the following concrete counterpart : R $\rightarrow$ b N s * e, that is, a repetition is bracketed by begin and end delimiters *b* and *e* and all components of a repetition are separated by a separator *s*.

There are several ways to layout a repetition, as the table shows (not all layouts are listed).

b
   Ns
   …
   Ns
   N
e
	b Ns
	   …
	   Ns
	   N e
	b
	e
		b Ns Ns … Ns N e
		
		b NsNs…NsN e
		
		b e
			b
			   Ns Ns … Ns N
			e
			b
			   NsNs…NsN
			e

We suppose that components of a repetition are all placed either after or below each other and that the separator immediately follows the components. Begin and end delimiters can be separated from the components with a space, a carriage return or no blank at all. Components are indented a few spaces. When presented horizontally a space can be used to separate components from one another. Note the empty repetitions in the second and third column.

To present empty repetitions and non empty ones, or small repetitions and big ones in different ways conditional presentation is necessary. The only conditional we will allow here, is the size conditional. It can be broken down into two parts : the number of descendents is smaller than a given number (horizontal) or it is strict greater than that given number (vertical).

The presentation of a repetition can thus be parameterized with the following (the names are somewhat cryptic for notational convenience later on):

<u>layout</u>		
hMax	Integer	maximum number of horizontal descendents
indent	Integer	indentation spaces when vertical
hSep	space/none	separating layout character when horizontal
hBefore	space/none	layout character before descendents when horizontal
hAfter	space/none	layout character after descendents when horizontal
vBefore	cr/space/none	layout character before descendents when vertical
vAfter	cr/space/none	layout character after descendents when vertical
<u>surface syntax</u>		
b	Token	begin delimiter
e	Token	end delimiter
s	Token	separator

Example The Agora grammar given before contains a production for a block:

Block $\rightarrow$ Expression *.

The presentation (2, 1, space, none, none, cr, cr, '[', ']', ';') would present blocks as

```
(1)    []
(2)    [Expression; Expression]
(3)    [
        Expression;
        Expression;
        Expression
       ]
```

The presentation specifies that Agora blocks are presented vertically when they contain more than two descendents. The descendents are indented one space and carriage returns are put between the delimiters and the descendents (3). When a block is presented horizontally, a space is used between a separating semicolon and the succeeding descendent and no blanks are added between the delimiters and the descendents of the block (2).

5.1.2 List Presentation

List presentation is the same as repetition presentation, except that lists cannot be empty.

5.1.3 Construction Presentation

Consider the construction rule $C \rightarrow N_1\ N_2\ ...\ N_k$. Its concrete counterpart $C \rightarrow c_1\ N_1\ c_2\ N_2\ c_3\ ...\ c_k\ N_k\ c_{k+1}$ shows that a construction is bracketed by two delimiters c_1 and c_{k+1} and the components are separated by surface syntax $c_2, \ldots, c_k$.

Constructions can be presented in various ways, as the table shows (not all layouts are listed). In general a construction takes the presentation of the first or third column.

$c_1\ N_1\ c_2\ N_2\ c_3\ ...\ c_k\ N_k\ c_{k+1}$	$c_1\ N_1\ c_2$
	   $N_2\ c_3$
	   $...\ c_k$
	   N_k
	c_{k+1}
		$c_1\ N_1\ c_2\ N_2\ c_3\ ...\ c_k$
		   N_k
		c_{k+1}

We suppose that each component of a construction may be placed arbitrarily, that is, spaces or carriage returns may be placed between each component and its preceding and succeeding surface syntax. When presented vertically, all components starting at the beginning of a line are indented equally.

The presentation of constructions can thus be parameterized with the following :

<u>layout</u>		
indent	Integer	number of indentation spaces
before	array of cr/space/none	layout characters before descendents
after	array of cr/space/none	layout characters after descendents
<u>surface syntax</u>		
c	array of Token	surface syntax tokens

Example The Agora grammar given before contains a production for a pair:

Pair $\rightarrow$ Keyword Expression.

The presentation (1, [none, none], [space, none], ['', ':', '']) would present pairs as follows :

```
Keyword: Expression
```

The presentation (3, [none, cr], [none, none], ['', ':', '']) would present pairs as follows :

```
Keyword:
   Expression
```

5.1.4 Lexeme Presentation

A lexeme in a program is identified solely by its textual representation (number literals, e.g. integers, reals, doubles, …) or by a preceding and/or succeeding delimiter (other literals, e.g. characters, strings, symbols, …). In either case, the textual representation of a lexeme is a lexical issue. Consequently, the presentation scheme of a lexeme contains no information, and is therefore needless.

5.1.5 Precedence

The structure of an abstract derivation tree reflects precedence relationships between program fragments. For example, the abstract derivation tree of the expression '(1 + 2) + (3 + 4)' reflects that the two subexpressions should be treated in another way than the subexpressions of the expression '1 + 2 + 3 + 4'.

Until now, this aspect has been neglected in the presentation schemes, yet it is important because it introduces extra characters in the textual representation of a program. In most common programming languages *parentheses* are used to emphasize precedence relationships. Parentheses are needed when a node's precedence is lower than (or sometimes equal to) the precedence of its father node. We will assume here, for reasons of simplicity and clarity, that parentheses are the *only* characters used to express precedence with.

Example Agora has three message types, unary, operator and keyword messages, which may be nested arbitrarily. Unary messages have highest, keyword messages have lowest precedence.
Using an operator or keyword message as receiver for unary messages or as argument of operator messages, or using a keyword message as receiver or as argument of a keyword, operator or unary message, introduces parentheses in their textual representation, e.g.

(1 + 2) print
1 - (2 + 3)

vector at: 5 put: (vector at: 1)
(vector at: 2) print
(vector at: 2) + (vector at: 7)

To express precedence in the textual representation, the presentation schemes must be extended with precedence information. In general, each descendent of an abstract derivation tree node may have lower precedence than that node, so with each non terminal in the right-hand side of a production rule denoting a repetition, list or construction, a set of parentheses needing non terminals is associated.

Example Consider a SAG for numerical expressions

Addition	$\rightarrow$ Exp_1 Exp_2
Product	$\rightarrow$ Exp_3 Exp_4
Substraction	$\rightarrow$ Exp_5 Exp_6
Exp	$\rightarrow$ Id | Addition | Product | Substraction
Id	$\rightarrow$ LEXEME

The non terminals are indexed for notional convenience below.
The sets of non terminals with lower precedence are

lower(Exp_1) = { }
lower(Exp_2) = { }
lower(Exp_3) = {Addition, Substraction}
lower(Exp_4) = {Addition, Substraction)
lower(Exp_5) = { }
lower(Exp_6) = {Addition, Substraction}

Exp_3, Exp_4 and Exp_6 will thus be enclosed with parentheses when Products and Substractions containing them are unparsed.

So, repetition and list presentation schemes are extended with *one* set of non terminals with lower precedence and construction presentation schemes are extended with an *array* of such sets, one for each non terminal in the right-hand side of the corresponding production rule.

5.2 Generating a Textual Representation

In this section a structured attribute grammar for unparsing is given. It utilizes the presentation schemes from the previous section.

All SAG production rules of the same type share the same attribution rules. Since alternation rules have no attribution rules by definition and lexeme non terminals have an instrinsic 'text' attribute, only attribution rules for construction and repetition production rules must be specified.
It is assumed that the components of the presentation schemes, given in the previous section, are intrinsic attributes of the right-hand side non terminals.

In order to arrange the attribution rules conveniently, the following functions are used. The if-then-else-end construct should be considered a functional if-then-else.

- The binary operator + is the text concatenation operator.
- L(Blank, Integer) is a function that returns the Blank and if it is a carriage return it also appends the number of spaces indicated by the Integer.
- $T(N_i, i, C)$ = **if** $N_i \in C.lower_i$ **then** '(' + N_i.text + ')' **else** N_i.text **end**
- $T(N_i, R)$ = **if** $N_i \in$ R.lower **then** '(' + N_i.text + ')' **else** N_i.text **end**

The production rules are attributed as follows:

$A \rightarrow N_1 \mid \ldots \mid N_k$

(no attribution rules)

$T \rightarrow$ LEXEME

(no attribution rules)

$C \rightarrow N_1 \ldots N_k$

$N.I \leftarrow C.I + C.indent$

$$C.text \leftarrow \sum_{i=1}^{k} (L(C.before_i, C.indent) + T(N_i, i, R) + L(C.after_i, C.indent) + C.c_i) + C.c_{k+1}$$

$R \rightarrow N^*$ and $R \rightarrow N^+$

$N.I \leftarrow R.I + R.indent$

$$R.text \leftarrow R.b + \textbf{if } R.size > R.hmax \textbf{ then } L(R.vBefore, R.I) + T(N_1, R) + \sum_{i=2}^{R.size} (R.s + L(cr, R.I + R.indent) + T(N_i, R)) + L(R.vAfter, R.indent)$$

else

$$R.hBefore + T(N_1, R) + \sum_{i=2}^{R.size} (R.s + R.hSep + T(N_i, R)) + R.hAfter$$

end

$+ \quad R.e$

There are only two proper attributes. The inherited attribute *I* stands for the current indentation level, i.e. the number of spaces generated when a carriage return is generated. After evaluation, the synthesized attribute *text* contains the well formatted text representation.

5.3 FOLDING

Folding or elision is a technique for suppressing program fragments. Elision was introduced in the Cornell Program Synthesizer [Teitelbaum & Reps 81] to fit more of the program on character based screen displays. Comment templates were used to associate a comment with a sequence of statements (not necessarily a compound statement). Folding is also used to get a more abstract view of the program. For instance, by folding the bodies of procedures in a Pascal-like programming language, the user gets an overview of the available procedures.

From an abstractness point of view, a browser on a program does a better job than a structure editor with folding features. Moreover, folding in a hybrid structure editor poses some conceptual and technical problems. Suppose the user starts text editing a piece of the program containing a folded fragment. It is not clear what should be done with the folded fragment. It can be unfolded, what results in undesired visual effects for the user, or it can be left as it is, what results in a piece of edited text that can impossibly be parsed. Restricting text editing to program pieces not containing folded fragments constrains the user, since this approach is inconsistent with editing of arbitrary program pieces.

Because of the problems it poses concerning the style of interaction, folding is felt to be a feature that belongs in pure structure editors of the past, when browing techniques did not yet exist.

5.4 Multiple Views

Multiple views enable the user to see his programs in various ways. Multiple views come in handy during editing. A browser, for example, offers functionality that a structure editor does not and the other way around. Depending on the functionality needed, the user can switch from one view to the other to achieve his goals.
In a programming environment provided with a structure-oriented editor which only supports textual presentation of programs, multiple editor views are useful to see several parts of the program at the same time.

The basis of multiple views is that all views use the same internal representation of the object they are viewing, but that the presentation of the views differ.
Optimally, changes made by editing operations in one view should directly be reflected in other views on the same object, so that no inconsistencies between views arise, while changes made to the presentation in one view should *not* be propagated to other views.

A view needs information in order to be able to display a representation of the object viewed. For instance, a structure editor uses a textual representation for display purposes and it may annotate the viewed abstract syntax tree with folding information. It is clear that the latter technique is a bad one. A key factor for support of multiple views on a program (or any object for that matter) is that view information is kept in the view, not in the program viewed, otherwise a presentation change in one view could mess up the presentation in the other views.

5.5 Mapping Text Onto a Syntax Tree

As described in chapter one, a structure-oriented editor maintains two selections, the textual selection displayed in the program window and the structural selection denoting the partial syntax tree the textual selection corresponds to. Making structural selections is necessary to be able to perform structure-oriented editing operations.
The structural selection is represented by a tree path, introduced earlier. A selection is made by means of the mouse or a selection command. Making selections is done frequently, or better, continuously. Therefore an efficient algorithm is necessary to translate mouse clicks into structural selections.

HV polygons are the means for structure-oriented navigation in a program. An HV polygon is an attribute of a syntax tree node describing the relative position and the size of the node's textual representation within the program's text. By overlaying the program's text with HV-polygons, every character in the text belongs to a syntax tree node. This many-to-one relationship between characters and tree nodes is the basis for picking, selection and the drag&drop technique.

The HV polygon is the mature version of the 'editable rectangle' in earlier work [De Hondt 90]. It is similar to the size-and-shape structures found in GSE [Koorn 92]. Both concepts are incremental and rely on relative information about the position and the size of a program fragment.

5.5.1 HV POLYGONS

An HV polygon of a program tree node describes what part of the textual representation of the program corresponds to that node. HV polygons are generated by the unparser. Since the unparser generates a stream of characters, HV polygons can be represented as pairs of indices in that stream, the first indicating the first character and the second indicating the last character of the substream of the corresponding program tree node. HV polygons of subnodes of a tree node are always nested in the HV polygon of that node.

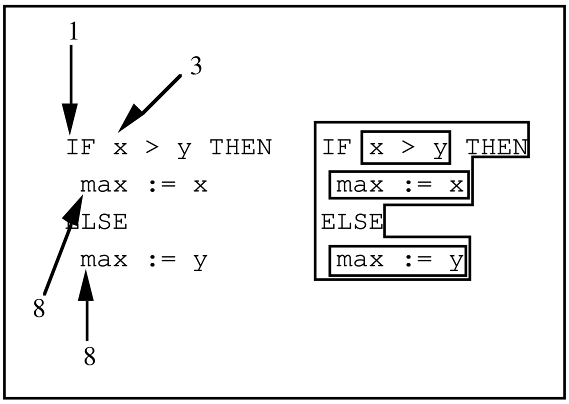
Using absolute indices is not appropriate since every edit operation on a node would require updating the HV polygons of all nodes in the program tree. It is better to take a relative approach.

Definition AHV(N) = (B, E) is the **absolute HV polygon of a program tree node** N starting at absolute position B and ending at absolute position E.
HV(N) = (O, L) is the (relative) **HV polygon of a program tree node** N starting at relative offset O and having length L - 1, thus ending at relative position O + L.

AHV(R).B = 1
HV(R).B = 1
For $N[N_1, \ldots, N_p]$:
$AHV(N_1).B = AHV(N).B + HV(N_1).O$
$AHV(N_i).B = AHV(N_{i-1}).E + HV(N_i).O, \forall\, i \in [2, p]$
AHV(N).E = AHV(N).B + HV(N).L

Absolute HV polygons are calculated during an unparse phase. The equations in the definition determine the relative HV polygons. Note that the relative offset and the absolute offset of the root node are equal.

The offset of the first subnode of a node is relative to the first character corresponding to that node. The offset of any other subnode is relative to the last character of the preceding subnode. The figure makes this clear. On the left a Pascal style if-then-else statement is shown with the values of the HV polygon offsets of the corresponding program tree nodes. Carriage returns and indentation spaces are each counted for one character. On the right the absolute HV polygons of the if statement node and its subnodes are drawn (the figure shows where the term 'HV polygon' comes from).



HV polygons of surface syntax need not be stored since they can be calculated from the HV polygons of the node and its subnodes together with the layout information stored in the templates.

When a program tree is attributed with relative HV polygons the absolute HV polygon of a node pointed to by a tree path is easily computed, as the following algorithm shows.

Algorithm

```
ComputeAbsoluteHVPolygon
in    tree path TP(R) = (I_1, …,I_p)
out   absolute HV polygon AHV

var   abs
begin
   abs := 1
   N := R
   for k in [1, p] do
      N := N[k]
      for j in [1, I_k-1] do
         abs := abs + HV(N[j]).O + HV(N[j]).L
      abs := abs + HV(N[j]).O
   AHV.B := abs
   AHV.E := AHV.B + HV(N).L
end
```

5.5.2 PICKING

We assume here that the editor uses an index in the program's text to denote a (selected) character. This index is called a *character position* further on. The first character has position 1.

The hierarchical composition of the HV polygons gives rise to a simple picking algorithm. The algorithm takes a program tree and an absolute character position and returns a tree path to the node picked, i.e. the node of which the absolute HV polygon contains the given character position.

Algorithm

```
Pick
in    character position P
      program tree R
out   tree path or ⊥

   PickSubnodes
   in    character position P
         tree node N
         character position S
         tree path TP
   out   tree path

   begin
      for i in [1, #(N)] do
         S := S + HV(N[i]).O
         if P in [S, S + HV(N[i]).L] then
            return PickSubnodes (P, N[i], S, TP + i)
         S := S + HV(N[i]).L
      return TP
   end

begin
   if P in [HV(R).O, HV(R).L] then
      return PickSubnodes (P, R, HV(R).O, ())
   else return ⊥
end
```

This algorithm returns a tree path pointing to the most deeply nested syntax tree node containing the input character position. As a result, a click between two components of a repetition picks the repetition. This is not sufficient for a convenient interaction style. The editor must be able to detect clicks between descendents of a repetition or list.

The algorithm can be reworked (not given here) in order to return a tree path and an indication whether the given character position lies in, before or after the program fragment pointed to by the tree path.

5.5.3 UPDATING HV POLYGONS

After every editing operation, some HV poygons need an update, because an editing operation almost always changes the textual representation of a program fragment. Due to the relative nature of the HV polygons, only the HV polygons of the abstract syntax tree nodes on the tree path from the root to the edited node need an update.

The effect of basic program synthesis commands on the HV polygons is discussed and algorithms for updating the polygons are given.

REPLACE

Replacement of a partial AST by another one only affects the HV polygons of the program tree nodes on the absolute tree path pointing to that subtree because of their relative and nested nature.

Suppose a node N pointed to by TP(R) is to be *replaced* by a node M. Two cases can be distinguished:

1. N = R
   No HV polygons must be updated because the whole program tree is replaced.

2. $N = L_k$ and $L[L_1, \ldots, L_k, \ldots, L_p]$
   All nodes enclosing N must be updated with the change in length of the HV polygon.

The following algorithm updates the HV polygons for the two cases. Assume that the length of the HV polygon of the replacement subtree is already calculated.

Algorithm

```
UpdateHVPolygonsAfterReplace
in     tree path to N TP(R) = (I1, …,Ip)
       HV polygon of replaced node OLDHV

var    delta
       ATP
begin
   ATP = (N1, …, Np, N) := ATP(TP(R))
   delta := HV(N).L - OLDHV.L
   if LEN(TP(R)) > 0 then
      for i in [1, p] do
         HV(Ni).L := HV(Ni).L + delta
   HV(N).O := OLDHV.O
end
```

INSERT

Suppose a node M is to be *inserted before* a node N pointed to by TP(R). Two cases can be distinguished:

1. $N = L_1$ and $L[L_1, \ldots, L_p]$
2. $N = L_k$ $(k > 1)$ and $L[L_1, \ldots, L_k, \ldots, L_p]$

Suppose a node M is to be *inserted after* a node N pointed to by TP(R). Two cases can be distinguished:

1. $N = L_p$ and $L[L_1, \ldots, L_p]$
2. $N = L_k$ $(k < p)$ and $L[L_1, \ldots, L_k, \ldots, L_p]$

In both cases the HV polygons of the inserted node M and node N must be updated. All nodes enclosing M must be updated with the change in length of the HV polygon of L. In the first case, the HV polygon of the node following M, and in the second case, the HV polygon of the node preceding M is affected.
The following algorithms update the HV polygons for the two cases. Assume that the length of the HV polygon of the inserted subtree is already calculated.

Algorithm
```
UpdateHVPolygonsAfterInsertBefore
in    tree path to M TP(R) = (I1, …,Ip)

var   NEWHV
      ATP
begin
   ATP = (N1, …, Np, M) := ATP(TP(R))
   NEWHV.O := HV(Np[Ip]).O
   NEWHV.L := HV(M).L
   HV(Np[Ip+1]).O := IthOffsetFromTemplate(ΔNp,Ip+1)
   UpdateHVPolygonsAfterReplace(TP(R), NEWHV)
end
```

In other words, when a subtree M is inserted at index I_p before a subtree N $(= N_p[I_p+1])$, its HV polygon offset gets the value of the offset of N and N's offset must be updated. N's offset is fetched from the template corresponding with the repetition or list N_p. It can be calculated from the layout information and the associated surface syntax. Finally executing `UpdateHVPolygonsAfterReplace` ensures proper update of the enclosing HV polygons.

Algorithm
```
UpdateHVPolygonsAfterInsertAfter
in    tree path to M TP(R) = (I1, …,Ip)

var   NEWHV
      ATP
begin
   ATP = (N1, …, Np, M) := ATP(TP(R))
   NEWHV.O := IthOffsetFromTemplate(ΔNp,Ip)
   NEWHV.L := HV(M).L
   UpdateHVPolygonsAfterReplace(TP(R), NEWHV)
end
```

When a subtree M is inserted at index I_p after a subtree N (= $N_p[I_p$-1]), its HV polygon offset gets a value fetched from the template corresponding with the father node N_p.

REMOVE

Suppose a node N pointed to by TP(R) is to be *removed* from a node L. Two cases can be distinguished:

1. $N = L_1$ and $L[L_1, \ldots, L_p]$
2. $N = L_k$ ($k > 1$) and $L[L_1, \ldots, L_k, \ldots, L_p]$

The HV polygons of the removed node N and node L must be updated. All nodes enclosing L must be updated with the change in length of the HV polygon of L.
The following algorithm updates the HV polygons for the two cases.

Algorithm

```
UpdateHVPolygonsBeforeRemove
in    tree path to M TP(R) = (I1, …,Ip)

var   delta
      ATP
begin
   ATP = (N1, …, Np, M) := ATP(TP(R))
   if Ip = 1 & #(Np) > 1 then
      HV(Np[Ip] + 1).O := HV(M).O
   delta := HV(M).L
   for i in [1, p] do
      HV(Ni).L := HV(Ni).L - delta
end
```

With this algorithm updating HV polygons has to be done before removal otherwise the update information (delta) is unknown. It is possible to change it in order not to have that restriction.

Since only HV polygons of parent nodes and possible one brother node are affected by the editing operations, the HV polygon update is an O(N) algorithm, where N is the nesting depth of the edited subtree in the program's abstract syntax tree.

5.5.4 TEXT FRAGMENTS AND HV POLYGONS

Since text fragments are an integral part of the abstract syntax tree, they have an associated HV polygon.

During structure-oriented editing operations the HV polygon of an AST node does not change, since those operations are atomic. During text editing however, the HV polygon of the edited program piece conceptually changes with every keystroke. Not only the HV polygon of the edited piece, but also all enclosing HV polygons need an

update with every keystroke. As a consequence, a text editing operation may seriously tax the ability of the structure editor to handle keystrokes.
The solution is a delayed HV polygon update when a program piece is being text edited. The update is delayed until a command requiring correct HV polygons is given, e.g. making selections outside the edited text.

5.5.5 SUPPORTING MULTIPLE VIEWS

The HV polygon is introduced earlier as attribute of a syntax tree node, but this approach causes the problem of not being able to have multiple views on one abstract syntax tree.
As indicated earlier in this chapter, view information, such as HV polygons, must be kept separate from the information to be viewed. View information should be stored in the view on the object, not in the object itself.
Despite the introduction as attribute, HV polygons have not been written down as proper attributes. In the algorithms in the previous section, the HV polygon of a syntax tree node N is written as HV(N), as the definition prescribes, not N.HV. The HV polygon of a node is thus considered to be information that can be determined when *given* a node, rather than information directly fetched *from* a node.

In an implementation environment supporting multiple views on objects, such as Smalltalk [Goldberg 84b], where objects are models and windows are views on those models, the abstract syntax tree is the object being manipulated by the structure-oriented editor and the program window is a view on the syntax tree. Every change in the syntax tree made by the editor is propagated to all its views. Since the views manage the HV polygons of the AST they are viewing, they update the HV polygons and reflect the changes in the textual program representation they display.

5.6 SUMMARY

In this chapter simple *presentation schemes* have been introduced. They lay down how the internal representation of a program should be represented textually. A structured attribute grammar has been used to describe how such textual representation is to be generated.

Two topics concerning program presentation have been discussed. *Folding*, an operation to get a more abstract view of a program, is felt to be obsolete. Its functionality is incorporated in present day browsers. *Multiple views* on a program are considered useful.

HV polygons have been introduced to create a mapping between an abstract syntax tree and its textual representation. Algorithms for picking program fragments and for updating the mapping after editing operations have been presented. Updating HV polygons of text fragments during text editing has been discussed.

Six

INTERACTING WITH STRUCTURE EDITORS

This chapter presents an interface for the structure-oriented commands described in the previous chapters. All editing operations will be demonstrated with small examples taken from the Agora Structure Editor.

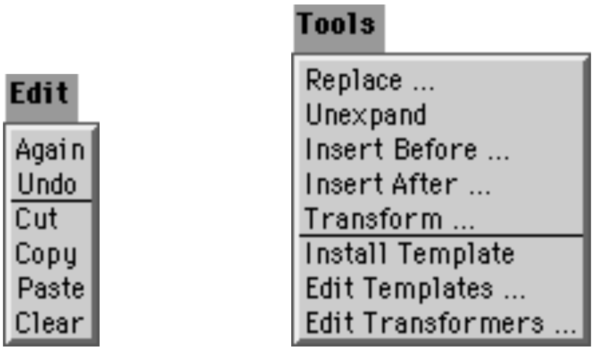


Figure 1

All structure-oriented commands of the Agora Structure Editor are grouped in the 'Edit' and the 'Tools' menus, as shown in figure 1. The ellipsis sign behind a command in the top half of the 'Tools' menu is a visual cue which indicates that a pop-up menu will appear after selection to complete the command.

In the bottom half of the 'Tools' menu one finds commands concerning templates. In this context, i.e. the user interface, 'template' has a different, yet related, meaning than the one given in chapter four. For the user, a template is a custom instance. In this context template thus means *arbitrary program fragment*.

The commands in the edit menu have structure-oriented functionality. They are special purpose insert, replace and remove commands.

Every editing operation can be undone by another editing operation. Before a replace or clear command is executed, the current selection is copied for undo purposes. Undoing a replacement is thus replacing the replacement selection with the saved copy. A clear command is undone with an insert command and undoing insert operations is done with a clear operation. Since paste, unexpand and transform operations are in fact replace operations, they can be undone as described.

6.1 PROGRAM SYNTHESIS COMMANDS

The synthesis commands Replace/Expand, Insert Before and Insert After all display a menu of possible replacements/expansions or insertions. The editor performs the command when a selection in that menu is made. The contents of the displayed menu depends on the grammatical category of the current fragment selection. In the figure below, the displayed menu is associated with an Agora expression, which is, according to Agora's grammar, an umbrella term for aggregates, messages, literals and patterns, all listed in the menu.

Figure 2

Since it is possible to install an arbitrary instantiation of a grammatical category in a menu (see later: Manipulating Templates), replacing and inserting is not restricted to expansion and insertion of placeholders. Complex syntactic constructs can be replaced and inserted by a single command. This contrasts strongly with the replace and insert capabilities of present day structure editors.

The remove command, previously defined as a synthesis command, is not listed in the Tools menu, because its behavior is equal to the that of the Clear command found in the Edit menu. Removing program fragments is thus carried out with the Clear command.

The menu put up after a 'Replace…' command lists all installed custom instances that can be substituted for the current selected program fragment. Assume the replace command was selected in figure 2. Selecting 'Method:' from the submenu as depicted puts the editor window in a state shown in the figure 3.

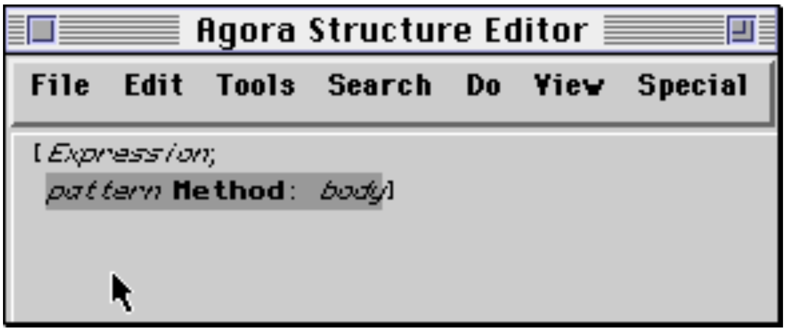


Figure 3

The menu put up after an 'Insert Before…'/'Insert After…' command lists all installed templates that can be inserted before/after an element of a repetition or list, of which the selected fragment must be a component. Inserting the first component of a repetition or list must be done with a structural transformation, although a structure editor may easily provide an 'Insert First…' command that is in fact a structural transformation of an empty repetition into one having a single component. Editors providing the suggested replacement feature discourage installation of empty repetitions and lists, so an 'Insert First…' command is felt to be somewhat redundant.

The clear command removes a selected element of repetition and list constructs. Stroking the delete (or backspace) key has the same effect. The only component of a single component list cannot be removed.

6.2 DRAG & DROP

Drag & drop is the direct manipulation variant of the replace and insert commands. When a program fragment is selected, it can be dragged across the program text. While dragging a fragment, all dragged over fragments that can be replaced by that fragment are highlighted.
Releasing the mouse button while a fragment is highlighted results in the replacement of that fragment by the dragged fragment. In particular placeholders are expanded when dragged fragments are released above them.
Releasing the mouse button while the mouse pointer is located between components of a repetition results in insertion of the dragged fragment.

Normally drag & drop has *copy behavior*, that is, the replacement fragment or the inserted fragment is a copy of the dragged fragment. Sometimes *move behavior* is more convenient: The dragged fragment is removed from its original position in the program and put elsewhere. An editor should support both styles, since move behavior is more ergonomic than simulating it by copying a fragment and removing the original.

From experience with earlier prototypes of the Agora Structure Editor it is become clear that drag & drop is a very powerful tool. It supports programming by example: if the user needs some fragment, he drags & drops a similar one and edits it until it meets his needs. Identifiers are typed once. From then on, they are dragged & dropped.
To the best of our knowledge, the Agora Structure Editor is currently the only structure-oriented editor featuring drag & drop.

Drag & drop is in fact also an implementation of cut/copy and paste (cut for move behavior, copy for copy behavior), using the whole program as a random access multi item clipboard. If cross window dragging is supported not only the target window, but all structure editor windows on the desktop act as one big clipboard.

6.3 Transforming Program Fragments

A structural transformer is an association between two structural patterns, the source and destination patterns. The former specifies what program fragments the transformer is applicable on. The latter specifies what the first looks like after transformation.
Transformation is performed by:

- pattern matching a target tree to the source pattern, associating with each placeholder in the source the matching subtree of the target
- replacing the target tree by a copy of the destination pattern with each placeholder substituted by its associated subtree originating from the pattern match

The 'Transform…' command in the Tools menu puts up a menu containing all structural transformations applicable on the current selection. The editor determines applicability of the transformations by pattern matching the current selection with each transformer source pattern.

For example, applying a 'Transform…' command to the selection of figure 3, after replacement of the 'body' placeholder by a block with two expressions, displays the transform menu shown in figure 4.

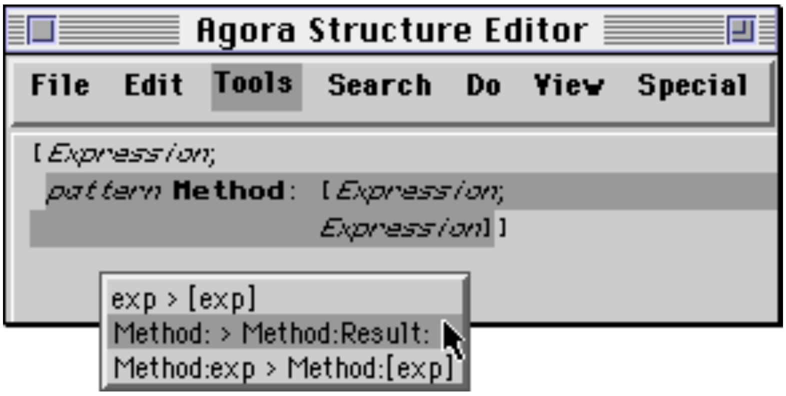


Figure 4

When the menu item under the mouse pointer is selected, the current selection is transformed according to the transformer having the name 'Method: > Method:Result:'. This transformer changes an Agora imperative method declaration into a functional method declaration, as shown in figure 5.

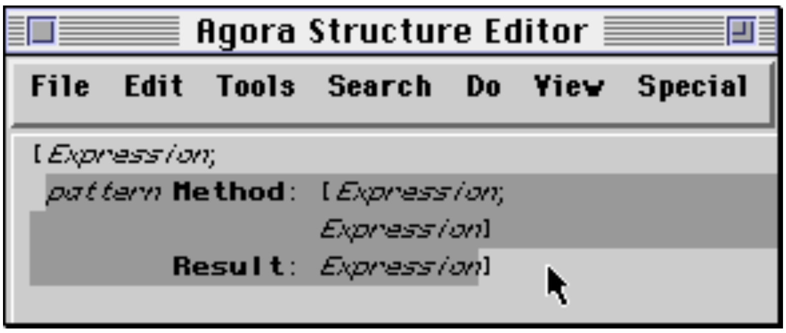


Figure 5

This approach to structural transformation seems very promising. Nevertheless, investigation is still to be done to make the pattern matching technique more powerful. The current matching technique only supports matching a tree to a placeholder, but, in practice, matching a repetition of trees to a placeholder is necessary to construct more powerful transformations. For example, one wants to be able to transform a repetition containing some component into one without that compoment.

This technique is new. It is not found in other structure editor environments. This technique is simple, yet powerful. Therefore it is surprising that other environments do not provide a similar transformation operation.

6.4 EDITING TRANSFORMERS

Structural transformers can be added, removed and edited. The suggested interface shown in figure 6 supports manipulation of transformers. The list on the left contains the names of all currently installed transformers. The two structure editor views, 'Original' and 'Transformed', contain the source and destination patterns.

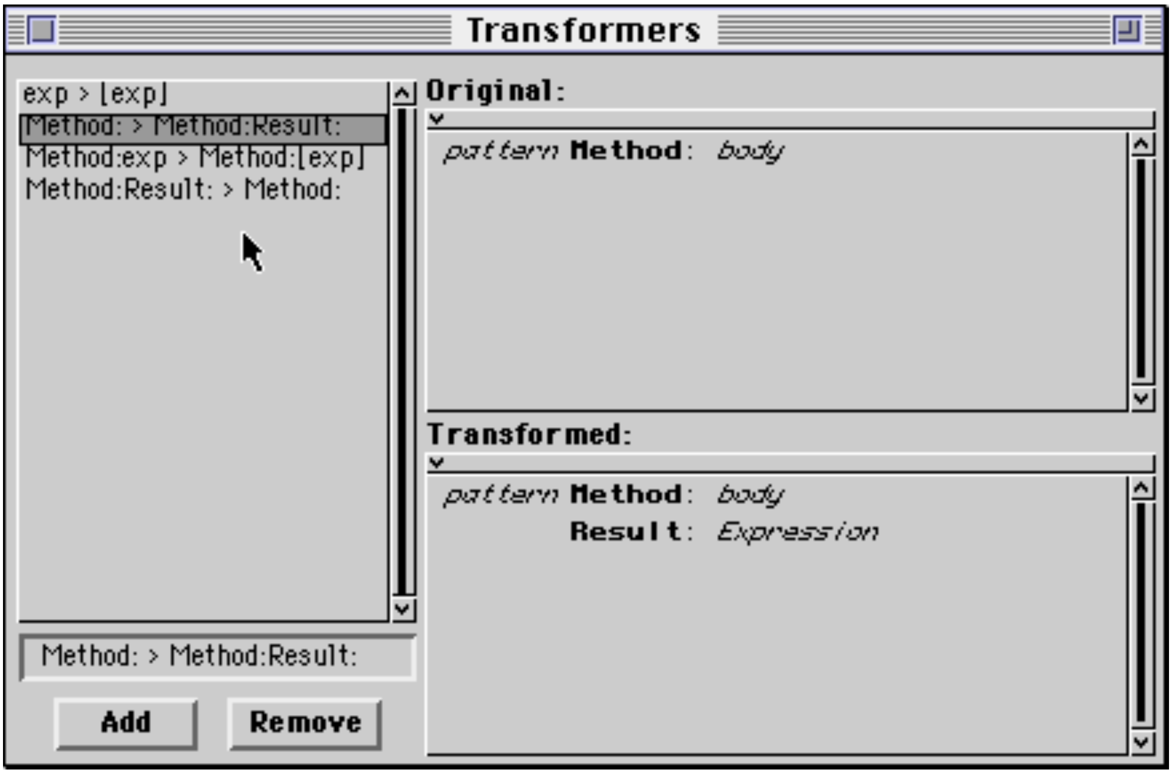


Figure 6

The transformer being defined in figure 6 is the one applied in the previous section. Note that named placeholders are a key factor here. Custom placeholder names resolve ambiguities when a pattern contains more than one placeholder of the same syntactical category. In the figure 'body' and 'Expression' have the same syntactical category. 'body' is a custom name for an Expression placeholder.

6.5 NAVIGATION

Program fragments must be selected before one can issue commands on them. The editor provides several methods to select pieces of a program.
As the editor allows text editing of programs the usual ways of making selections must be overloaded to support structural selections of program fragments. *Selection* will be used to refer to structural selection and *text selection* will be used to denote selection on character basis.

In what follows we assume that the structure-oriented editor executes on a graphical workstation equipped with a pointing device having three buttons, the red, yellow and blue buttons (as e.g. in [Goldberg 84b]). Since the yellow button is used for popping up a command menu in windows and the blue button is the window system command button, only the red button remains for selection purposes.

6.5.1 HOT SPOTS

A hot spot of a program fragment is a place to click in order to select that fragment or one of its subfragments. Two kinds of hot spots can be distinguished: *Direct hot spots* serve as hot spots for the fragment, *indirect hot spots* serve as hot spots for the subfragments. Hot spot without qualification is used for direct hot spots. Hot spots are defined by the HV polygons, introduced in the previous chapter.

The hot spot of lexemes and placeholders is the lexeme or placeholder itself. Lexemes and placeholders have no indirect hot spots.
Repetitions and lists have two hot spots, the begin and end delimiters. The separators and blanks separating the components are indirect hot spots for the component they succeed.
A construction has several hot spots, that is, all surface syntax and blanks contained in the construction, but these direct hot spots could serve equally well as indirect hot spots, as the following examples shows.

```
REPEAT
  DoThis;
  DoThat
UNTIL Done;
```

Consider the repeat-until statement (a construction) in many Pascal-like programming languages. The body of the construct (a repetition representing a statement sequence) has no begin and end delimiters as all surface syntax is part of the repeat construction. Only having direct hot spots introduces the problem of not being able to select the whole body of the repeat-until construct. Only having indirect hot spots has the disadvantage that the whole repeat-until cannot be selected because clicking the repeat keyword selects the body and clicking the until keyword selects the end condition. Having mixed direct and indirect hot spots is inconsistent and, as a consequence, very confusing for the user of the structure editor.

A construction needs direct as well as indirect hot spots otherwise a syntactical subconstruct lacking associated surface syntax cannot be selected. The direct and indirect hot spots of a construction are the same, therefore a structure editor will have to overload the mouse click to differentiate between direct and indirect hot spot clicks.

6.6.2 Selection With the Mouse

The red mouse button is used to make selections. Several current text editors [Apple 92][Goldberg 84b] use the following conventions for text selection:

- single click positions the caret between two characters
- single click while the shift key is pressed expands or shrinks the current text selection range
- double click selects a string of characters (usually a word)
- triple click selects a whole line of characters

Sometimes quadruple clicking results in selection of the whole text. Note that the clicking conventions are engrafted on the structure of a simple text: character–word–line–text.

The red mouse button used for text selection is overloaded in a natural and logical fashion. For use in a hybrid structure editor some of its functionality is altered :

- single click positions the caret between two characters
- single click while the shift key is pressed expands or shrinks the current text selection range
- double click on a direct hot spot selects the corresponding program fragment
- double click on an indirect hot spot while the command key is pressed selects the program fragment corresponding to the spot
- multiple click expands the current fragment selection to its enclosing fragment

Any character in a program's text belongs to a hot spot, so every double click results in fragment selection.
The structural double click is a logical extension of the textual double and triple click: double clicking a hot spot of a fragment results in selection of the textual representation of that fragment. For lexemes and placeholders this is like textual double click, for other fragments this is analogous to textual triple click. There is one downside: Textual double click on hot spots is not possible any more.

Multiple clicking to select enclosing fragments proves to be very handy. It is even necessary in order to solve the awkward problem of not being able to select a repetition lacking surface syntax. Moreover, a user cannot see the difference between the selection of the single component of such a repetition and the selection of the repetition itself. Multiple clicking permits making the correct selection. The desired selection can also be made using an indirect hot spot if the enclosing fragment is a construction and if that construction has indirect hot spots.

6.6.3 Selection With the Keyboard

Text selection can also be achieved by means of the keyboard. Current text editors provide the following ways:

- pressing an arrow key moves the caret around one character or line at a time
- pressing an arrow key together with a command key moves the caret across strings of characters (usually words) or paragraphs (lines not containing a line break)
- holding down the shift key while pressing the keys mentioned above enlarges or shrinks the current text selection according to the move the caret makes without pressing the shift key

For use in a hybrid editor some of the functionality is altered:

- pressing an arrow key moves the caret around one character or line at a time
- pressing a horizontal arrow key together with a command key moves the caret across strings of characters
- pressing a vertical arrow key together with a command key moves the caret across preceding/succeeding fragments or, if none, across enclosing fragments
- holding down the shift key while pressing the keys mentioned above enlarges or shrinks the current text selection according to the move the caret makes without pressing the shift key

Since formatting is done automatically by the unparser, the tab key is obsolete when text editing program fragments. The tab key can thus be used for structure editing purposes. It comes in handy for selection of placeholders, especially after inserting a template of which components (placeholders) must be filled in.
More general stroking the tab key selects the first placeholder after the current selection and stroking shift-tab selects the first placeholder before the current selection. Pressing the command key while stroking the tab key selects an arbitrary fragment rather than a placeholder.

6.6 Picking By Command

In the previous section the functionality of the mouse for selection was discussed. We now turn to making selections via commands.

The 'Pick Enclosing…' command is a general picking command to select an enclosing program fragment of some syntactical type.
The 'Find Again' command is a general search command to select program fragments that conform to a given search pattern.
The figure on the right shows the Search menu of the Agora Structure Editor. Besides several commands for finding program fragments, it contains commands for replacing a found fragment by another one specified in the Find&Replace dialog put up by the 'Find…' command.

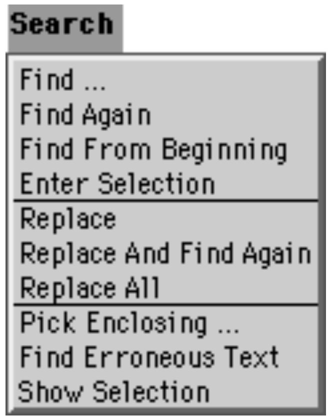


Figure 7

The 'Show Selection' command is useful for scrolling the selection into view. 'Find Erroneous Text' picks the first non parsed text following the current selection. Since in our design an editor can have multiple focusses, such a command is indispensable.

6.6.1 Picking Surrounding Program Fragments

Usually picking is performed with the mouse, but in some cases it is simpler to have a command perform the needed pick.
An example thereof is picking a surrounding syntactical construct. The 'Pick Enclosing…' command puts up a menu which lists all possible syntactical constructs. By selecting one, the first construct of that category enclosing the current selection will be selected. If no enclosing fragment of the desired category is found, the editor gives an audible signal. The figure shows the pick menu in the Agora Structure Editor. The contents is generated automatically from Agora's abstract grammar.

Figure 8

Picking enclosing fragments this way is especially usefull for selection of enclosing fragments that do not fit entirely in the editor window. Notice that the menu also contains lexemes. It is possible to select an enclosing lexeme when the caret or the selection range is located within one.

6.6.2 Picking Through Search

Searching can be done at character or at fragment level. The editor will search at fragment level if the search parameters allow it. For instance, searching for identifiers is very fast, because the program text need not be consulted for this kind of search.

When the search string can be parsed by the language's parser, a pattern match algorithm can be used to find subtrees that comply with the search string. If the parser accepts program text containing placeholders, placeholders in the search string act as wildcards.
When the search string cannot be parsed a textual search is performed. Note that a textual search in an abstract grammar tree is not an easy job (nor a fast one). Keeping a textual representation of the program besides the tree representation could speed up the search process drastically.

6.7 Manipulating Templates

When the command 'Edit Templates …' is selected from the 'Tools' menu (see figure 1), a window is displayed by which the replacement and insertion menus can be changed. The 'Install Template' command install a copy of the current structural selection in the appropriate menu. The figure depicting the interface for manipulation of templates is too big to show here, therefore it is moved to the appendix.

With every production rule a pop-up menu of alternatives is associated. These pop-up menus are nested according to the possible derivations, as laid down by the target language's grammar. The nesting can be determined by the user. Although more than two levels are discouraged from an ergonomic point of view, the user is free to nest as he likes, as long as the language's grammar is not violated.

For example, in the figure shown in the appendix, the contents of the expression alternatives menu is listed alphabetically in the list on the left. It contains four items, namely 'Literal', 'Message', 'Name' and '[]'. 'Literal' and 'Name' denote nested menus, 'Mesage' and '[]' (an Agora block) are real custom instances of an Agora expression.

When an item is selected in the list on the left, its expansion is displayed in the structure editor view on the right. This view implements a complete structure editor to manipulate the expansion with. The editor commands are not visible in the picture, they are located in the menubar elsewhere on the screen.

6.8 MODIFYING PRESENTATION SCHEMES

When the command 'Presentation …' from the 'View' menu (not shown in this document) is selected, a dialog like the one depicted in figure 9 is displayed. The look and feel of the window differs from the other figures, because the screendump is taken from another prototype. Figure 9 shows the presentation of an Agora block as it is given in the example of section 5.1.1.

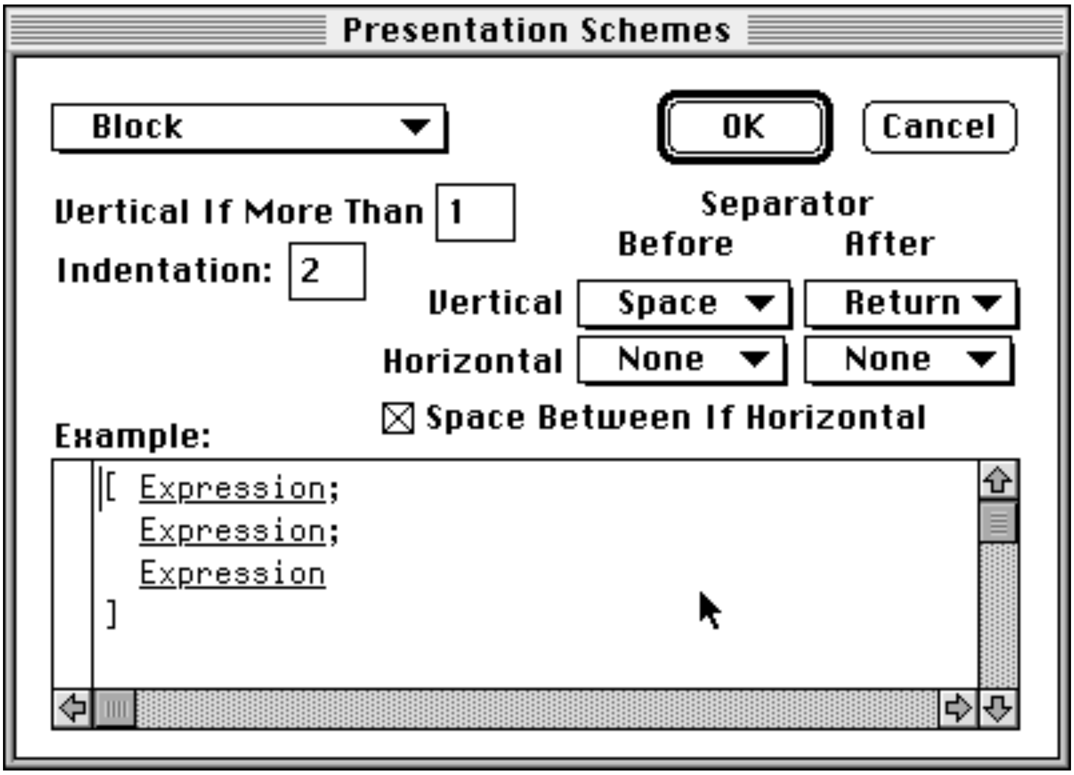


Figure 9

The pop-up menu in the top left corner lists the names of all production rules. For each production rule type the interface elements below the pop-up menu differ. The interface adapts itself when an item of the pop-up menu is selected. The dialog shown in figure 9 is the layout for all repetition and list productions. It contains interface elements for all presentation scheme items given in section 5.1.1.

6.9 TEXTUAL EDITING

In our view a hybrid structure-oriented editor should support text editing of arbitrary program pieces. Text editing should be possible at all times and it should not put the editor in a mode. The user should be able to switch from text editing to structure editing and back at any time.

These design options pose some technical difficulties. When the attention is moved away from a text edited program piece, a parser must be invoked to generate the corresponding partial AST. To support hybrid editing without modes, the editor should not complain about erroneous program text. From a technical standpoint this means that more than one erroneous program piece can exist in the AST. Furthermore, these incorrect program pieces should be subject to structure-oriented editing operations as all other program fragments. To prevent that the AST turns into a tree only containing erroneous program pieces, the editor should merge pieces into pieces that can be parsed.

In this section switching from text to structure and the other way around is briefly discussed. Since extending and merging erroneous program fragments are topics still under investigation at the time of writing, it will not be handled here.

6.9.1 Switching From Structure to Text

When a text editing operation is initiated the edited program fragment is replaced by a text placeholder with the same HV polygon, since the fragment and the text placeholder have the same textual representation.
Due to this fact, no screen updates are necessary at the beginning of textual change. The user thus does not sense the internal transition from program fragment to text placeholder. He feels as if he can start text editing at any time, without delay.

6.9.2 Switching From Text to Structure

When a switch from text editing to structure-oriented editing is made, the edited text needs to be parsed and the generated partial AST is to be unparsed to update the view on the program. If the edited text cannot be parsed the unparse phase is skipped.

Parsing and unparsing takes some time, consequently the user senses the transition from edited text to correct program fragment. For instance, when the user clicks with the mouse outside the edited text, he will have to wait until the edited piece is parsed and unparsed before his picking command is executed. If the edited pieces are rather big, the user interface may come over as sluggish. A suggested solution to this problem could be that parsing and unparsing are performed in background when the editor is in a idle state.

6.10 Summary

In this chapter an interface for structure-oriented editors has been presented. Editing commands and their effect have been shown. Navigation with the mouse, with the keyboard and by command has been discussed. Suggested interfaces for customization of replacement and insertion menus, presentation schemes and structural transformations were shown. Switching from text to structure and back has been discussed briefly.

Seven

Conclusion and Future Work

In this chapter, after a brief discussion on the implementation of the Agora Structure Editor and a presentation of work that still needs to be done, and investigations that are still to be conducted, conclusions are drawn.

7.1 Implementation

Several protoypes of the Agora Structure Editor exist. The first prototype (1992), implemented in Pascal, supports construction of complete Agora programs. It provides all editing operations presented in previous chapters, except structural transformation and general structural search & replace. It features drag & drop and textual editing of arbitrary program pieces. Presentation schemes can be modified and replacement and insertion menus can be adapted to the user's needs.

The current prototype is implemented in VisualWorks\Smalltalk. It is integrated in the programming environment for Agora. Now, the editor is completely independent of Agora. An editor is available for a target language if (1) a grammar class hierarchy for some target language is constructed, (2) those classes are connected with the structured grammar class hierarchy used by the editor classes, (3) a parser for the target language is provided.

The editor currently provides all structure-oriented editing operations, including structural transformation. Transformers can be added, removed or changed through the interface presented in chapter six. Due to minor technical problems involving dragging, drag & drop is not yet supported, but will be in the near future. The implementation of the hybrid aspect and the search facilities is currently ongoing.

The use of VisualWorks has drastically reduced the time spent on building user interface components. More time can now be spent to evaluate them and make them more ergonomic.

The use of VisualWorks\Smalltalk as implementation environment ensures cross platform portability of the Agora programming environment, since VisualWorks images can be moved to another hardware platform without a single modification.

7.2 FUTURE WORK

Few people have evaluated the prototypes mentioned above. Clearly, an evaluation by a large group of persons is necessary in order to improve the robustness and the interface of the editor. The practical usability should be evaluated by means of empirical investigations measuring the effort required to accomplish a certain task.

Multiple structural selection is mentioned in chapter four, but it is not implemented in any of the existing prototypes. Multiple selection introduces some technical difficulties, since pieces of a program, not representing a valid grammatical construct, cannot be cut, copied, pasted, replaced and inserted. The editor will have to provide facilities to manipulate such special selections. In particular a user must be able to manipulate parts of repetition and list constructs, because manipulating a sublist one component at a time is not desirable from an ergonomic standpoint.

Structural transformation and searching, as presented in chapter one and six, cannot be exploited to the full extent, because of the rather primitive, yet powerful, structural pattern matching technique. Matching placeholders to sublists can make structural pattern matching even more powerful. The problem of associating a placeholder with a sublist is similar to the problem of manipulating arbitrary sublists for multiple selection purposes mentioned above.

The structure editor is designed to be independent of the target language, but a parser for a target language still needs to be specified. Whether a table driven parser can be derived automatically from a structured abstract grammar together with the associated surface syntax and associativity and precedence constraints (as done in the SDF formalism [Hendriks 91][Centaur 92]), should be further investigated.

The presentation schemes presented in chapter three are too simple, although they are satisfactory in most cases. A general method to describe conditional presentation with, would enhance the presentation schemes considerably. With the current presentation schemes, the presentation of a grammatical construct cannot depend on the construct it is a component of. This is particularly problematic for the presentation of Agora programs, where the font style of an identifier depends on the expression it is part of (see the structured context-free grammar for Agora in the appendix). More elaborate presentation schemes should be further examined.

Since more than one non parsed text fragment can appear in the abstract syntax tree, a way for merging non parsed text fragments should be searched for, otherwise the syntax tree could end up in a completely degenerate state.

7.3 CONCLUSION

In this thesis, design issues and techniques for a customizable, ergonomic hybrid structure editor have been presented.

Structured grammars have been defined formally to lay down how programs are represented internally, as well as externally. Abstract syntax trees were derived from abstract derivation trees in order to represent incomplete programs containing placeholders.

Techniques to support our design have been highlighted. The HV polygon concept has been introduced to create a mapping between the internal and the external representation of programs. Algorithms for picking program fragments and for updating the HV polygons after editing operations have been presented.

A complete set of structure-oriented editing commands has been presented. What really stands out against other work is the customizability of the editor. Presentation schemes can be modified, replacement and insertion menus can be tailored to the user's needs and structural transformations can be created interactively.

Since we embrace ergonomics as a principle, and other systems seem to neglect it, the editor differs strongly from other systems in that respect. The direct manipluation technique drag & drop is but one of the features that raise ergonomics to a higher level.

The editor has a hybrid nature. Text editing can be initiated at any moment, at any place in the program. No modes are introduced by the editor. The user can switch from text to structure editing and back at any time.

Due to the fact that the design is independent of the target language, the editor can also be considered generic. The independence relies on the application of structured grammars for internal representation of programs. The production rules of the target language's grammar are represented internally by instances of themselves, which are interpreted by the editor during editing operations.

We hope that the design and the techniques presented in this thesis are useful contributions in order to render structure-oriented editors into powerful program editing tools for general use outside universities and research laboratories.

APPENDIX A

A.1 AGORA'S GRAMMAR

A.1.1 AGORA'S STRUCTURED CONTEXT-FREE GRAMMAR

Reifier	$\longrightarrow$	ReifierMessage | ReifierPattern
ReifierMessage	$\longrightarrow$	ReifierOperation ReifierKeywordPattern
ReifierPattern	$\longrightarrow$	ReifierKeywordPattern | ReifierOperatorPattern | ReifierName
ReifierOperation	$\longrightarrow$	ReifierUnary ReifierOperatorPattern
ReifierUnary	$\longrightarrow$	Message | ReifierUnaryPair
ReifierUnaryPair	$\longrightarrow$	ReifierUnary ReifierName
ReifierName	$\longrightarrow$	LEXEME
ReifierOperatorPattern	$\longrightarrow$	ReifierOperatorPair *
ReifierOperatorPair	$\longrightarrow$	ReifierOperator ReifierOperation
ReifierOperator	$\longrightarrow$	LEXEME
ReifierKeywordPattern	$\longrightarrow$	ReifierKeywordPair *
ReifierKeywordPair	$\longrightarrow$	ReifierKeyword ReifierOperation
ReifierKeyword	$\longrightarrow$	LEXEME

Message	$\longrightarrow$	Operation KeywordPattern
Pattern	$\longrightarrow$	KeywordPattern | OperatorPattern | Name
Operation	$\longrightarrow$	Unary Name
Unary	$\longrightarrow$	Factor | UnaryPair
UnaryPair	$\longrightarrow$	Unary Name
Name	$\longrightarrow$	LEXEME
OperatorPattern	$\longrightarrow$	OperatorPair *

OperatorPair	→	Operator Operation
Operator	→	LEXEME
KeywordPattern	→	KeywordPair *
KeywordPair	→	Keyword Operation
Keyword	→	LEXEME

Factor	→	Aggregate | Name | Submessage | Literal
Aggregate	→	[Reifier ; *]
Submessage	→	(Reifier)
Literal	→	StringLiteral | IntegerLiteral | CharacterLiteral | RealLiteral
StringLiteral	→	LEXEME
IntegerLiteral	→	LEXEME
CharacterLiteral	→	LEXEME
RealLiteral	→	LEXEME

It catches the eye that Agora contains little surface syntax, only brackets, parentheses and semicolons. Other symbols are used to differentiate between the lexemes :

StringLiteral	→	" Characters "
CharacterLiteral	→	' Character '
IntegerLiteral	→	PositiveInteger | NegativeInteger
PositiveInteger	→	Digit +
NegativeInteger	→	- PositiveInteger
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
RealLiteral	→	Real | RealWithExponent
Real	→	IntegerLiteral . PositiveInteger
RealWithExponent	→	Real e IntegerLiteral

Keyword	→	Name :
ReifierKeyword	→	µ Keyword
Operator	→	Symbol +
ReifierOperator	→	µ Operator
Name	→	Letter LetterOrDigits
ReifierName	→	µ Name
Letter	→	a | … | z | A | … | Z
LetterOrDigit	→	Letter | Digit
LetterOrDigits	→	LetterOrDigit *

Symbol	→	! | … | ~
Character	→	Letter | Digit | Symbol
Characters	→	Character *

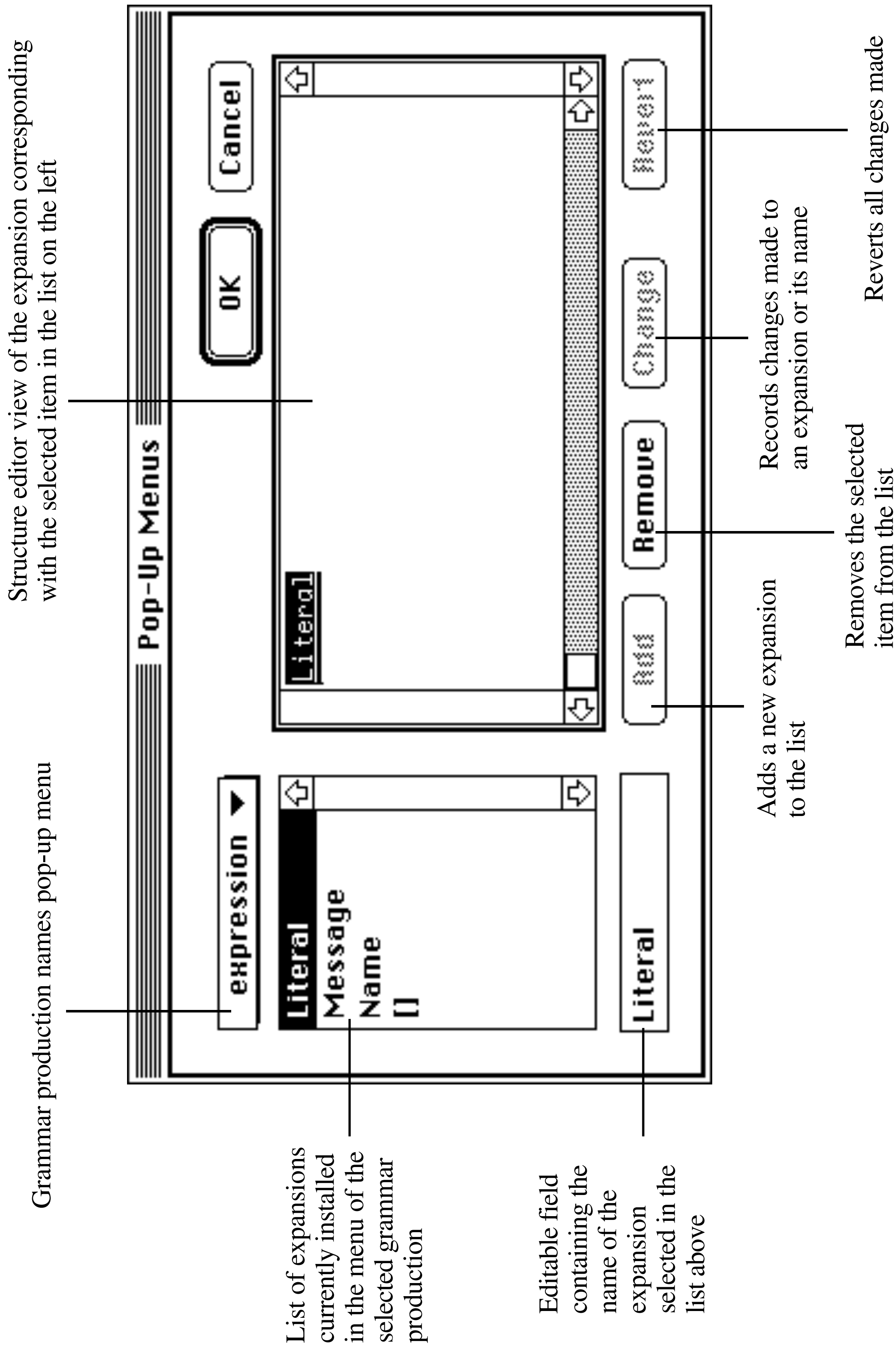
The μ symbols are used to differentiate between normal messages and reifier messages [Codenie & al. 91]. Identifiers with μ prefix in the grammar are never presented to the user as such. In the user interface they appear boldfaced without μ prefix.

A.1.2 AGORA'S STRUCTURED ABSTRACT GRAMMAR

Expression	→	Block | Message | Pattern | Literal
Block	→	Expression *
Message	→	Expression Pattern
Pattern	→	UnaryPattern | OperatorPattern | KeywordPattern | ReifierUnaryPattern | ReifierOperatorPattern | ReifierKeywordPattern
Literal	→	StringLiteral | IntegerLiteral | CharacterLiteral | RealLiteral
UnaryPattern	→	LEXEME
ReifierUnaryPattern	→	LEXEME
OperatorPattern	→	Operator Expression
ReifierOperatorPattern	→	Operator Expression
KeywordPattern	→	Pair +
ReifierKeywordPattern	→	Pair +
Pair	→	Keyword Expression
Keyword	→	LEXEME
Operator	→	LEXEME
StringLiteral	→	LEXEME
IntegerLiteral	→	LEXEME
CharacterLiteral	→	LEXEME
RealLiteral	→	LEXEME

A.2 MODIFYING THE POP-UP MENUS

The look and feel of the figure below is different of the other figures because it is taken from a previous prototype.



REFERENCES

[Abelson & Sussman 85] Abelson, H., Sussman, G.J., "Structure and Interpretation of Computer Programs", The MIT Press, 1985

[Aho & al. 86] Aho, A.V., Sethi, R., Ullman, J.D., "Compilers. Principles, Techniques and Tools", Addison-Wesley, 1986

[Allison 83] Allison, L., "Syntax Directed Program Editing", Software-Practice and Experience, Vol. 13, 453-465, 1983

[Apple 92] Apple Computer, Inc., "Macintosh Human Interface Guidelines", Addison-Wesley, November 1992

[Arefi & al. 90] Arefi, F., Hughes, C.E., Workman, D.A., "Automatically Generating Visual Syntax-Directed Editors", Communications of the ACM, Vol. 33, No.3, March 1990

[Bahlke & Snelting 86] Bahlke, R., Snelting, G., "The PSG System: From Formal Language Definitions To Interactive Programming Environments", ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, 547-576, October 1986

[Ballance & al. 88] Ballance, R.A., Butcher, J., Graham S.L., "Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor", SIGPLAN Notices, Vol. 23, No. 7, July 1988

[Blaschek & Sametinger 89] Blaschek, G., Sametinger, J., "User-adaptable Prettyprinting", Software—Practice and Experience, Vol. 19(7), 687-702, July 1989

[Brooks 87] Brooks, F.P., Jr., "No Silver Bullet. Essence and Accidents of Software Engineering", IEEE Computer, April 1987

[Cameron & Ito 84] Cameron, R.D., Ito, M.R., "Grammar-Based Definition Of Metaprogramming Systems", ACM Transactions on programming Languages and Systems, Vol. 6, No. 1, January 1984

[Centaur 92] Centaur 1.2 Manual, September 1992

[Clément 90] Clément, D., "GIPE : Generation of Interactive Programming Environments", Technique et Science Informatique, Vol. 9, No. 2, 1990

[Codenie & al. 91] Codenie, W., Steyaert, P., Van Limberghen, M., "Agora, a Short Introduction", Internal Report Agora 91-04, Programming Technology Lab, Brussels Free University, 1991

[Dart & al. 87] Dart, S.A., Ellison, R.J., Feiler, P.H., Habermann, A.N., "Software Development Environments", IEEE Computer, November 1987

[De Hondt 90] De Hondt, K., "Structuureditie", Licentiate Thesis, Brussels Free University, 1990 (Dutch)

[Delisle & al. 84] Delisle, N.M., Menicosy, D.E., 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, SIGPLAN Notices, Vol. 19, No 5, May 1984

[D'Hondt & al. 87] D'Hondt T., Tilman M, Steyaert P., "A Formalism For A Fully Reflexive Object Oriented Language With Embedded Parallelism", Free University Brussels, 1987

[Donzeau-Gouge & al. 80] Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., "Programming Environments Based on Structure Editors: The MENTOR Experience", Barstow et al., Interactive Programming Environments, McGraw-Hill, 1984

[Donzeau-Gouge & al. 84] Donzeau-Gouge, V., Kahn, G., Lang, B., Mélèse, B., "Document Structure and Modularity in Mentor", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 9, No. 3, May 1984

[Dykes & Cameron 90] Dykes, L. R., Cameron, R. D., "Towards High-Level Editing in Syntax-Based Editors", Software Engineering Journal, July 1990

[Engels & al. 86] Engels, G., Nagi, M., Schaefer, W., "On the Structure of Structure-oriented Editors for Different Applications", Communications of the ACM, 1986

[Fisher & al. 84] Fisher, C.N., Johnson, G.F., Mauney, J., Pal, A., Stock, D.L., "The Poe Language-Based Editor Project", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 9, No. 3, May 1984

[Fisher & LeBlanc 88] Fisher, C.N., LeBlanc, R.L.Jr., "Crafting a Compiler", The Benjamin/Cummings Publishing Company

[Ford & Wiener 85] Ford, G., Wiener, R., "Modula-2, A Software Development Approach", John Wiley & Sons, 1985

[Garlan & al. 86] Garlan, D., Krueger, C.W., Staudt, B.J., "A Structural Approach to the Maintenance of Structure-Oriented Environments", Communications of the ACM, 1986

[Goguen & Moriconi 87] Goguen, J., Moriconi, M., "Formalization in Programming Environments", IEEE Computer, November 1987

[Goldberg 84a] Goldberg, A., "The Influence of an Object-Oriented Language on the Programming Environment", Proceedings of the 1983 ACM Computer Science Conference, February 1983, also in the book Interactive Programming Environments, Barstow et al., 1984

[Goldberg 84b] Goldberg, A., "Smalltalk-80, The Interactive Programming Environment", Addison-Wesley Publishing Company, 1984

[Goldberg & Robson 89] Goldberg, A., Robson, D., "Smalltalk-80, The Language", Addison-Wesley Publishing Company, 1989

[Grune & Jacobs 88] Grune, D., Jacobs, C.J.H., "A Programmer-friendly LL(1) Parser Generator, Software—Practice and Experience, Vol. 18(1), 29-38, January 1988

[Halewood & Woodward 88] Halewood, K., Woodward, M.R., "NSEDIT: A Syntax-directed Editor and Testing Tool based on Nassi-Shneiderman Charts", IEEE Software — Practice and Experience, Vol. 18, No. 10, October 1988

[Halme & Heinanen 88] Halme, H., Heinanen, J., "GNU Emacs as a Dynamically Extensible Programming Environments", IEEE Software — Practice and Experience, Vol. 18, No. 10, October 1988

[Harrison 78] Harrison, M.A., "Introduction to Formal Language Theory", Addison-Wesley, 1978

[Hauge & al. 89] Hauge, T., Nordgard, I., Oscarsson, D., Ræder, G., "Event-Driven User Interfaces Based on Quase-Parallelism", Technical Report LU-CS-TR:89-51, Department of Computer Science, Lund University, Sweden, 1989

[Hedin & Magnusson 88a] Hedin, G., Magnusson, B., "Supporting Exploratory Programming in Simula", Technical Report LU-CS-TR:88-31, Department of Computer Science, Lund University, Sweden, 1988

[Hedin & Magnusson 88b] Hedin, G., Magnusson, B., "The Mjølner Environment : Direct Interaction with Abstractions", Proceedings of the European Conference on Object-Oriented Programming, August 1988

[Hedin 88] Hedin, G., "Incremental Attribute Evaluation with Side-effects", Proceedings of the Workshop on Compiler Compiler and High Speed Compilation, Berlin GDR, October 1988

[Hedin 89] Hedin, G., "An Object-Oriented Notation for Attribute Grammars, Technical report LU-CS-TR:89-42, Department of Computer Science, Lund University, Sweden, 1989

[Hedin 91] Hedin, G., "Incremental Static Semantic Analysis for Object-Oriented Languages using Door Attribute Grammars", Technical Report LU-CS-TR:91-79, Department of Computer Science, Lund University, Sweden, 1991

[Hedin & al. 88] Hedin, G., Magnusson, B., Taube, M., "An Object-Oriented View of Programs and Program Execution", Technical Report LU-CS-TR:88-33, Department of Computer Science, Lund University, Sweden, 1988

[Henderson & Notkin 87] Henderson, P.B., Notkin, D., "Integrated Design and Programming Environments", IEEE Computer, November 1987

[Hendriks 91] Hendriks, P.R.H., "Implementation of Modular Algebraic Specifications", PhD thesis, University of Amsterdam, 1991

[Horgan & Moore 84] Horgan,m J.R., Moore, D.J., "Techniques for Improving Language-Based Editors", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 9, No. 3, May 1984

[Jadoul & al. 88] Jadoul, M., Vanbaelen, M., D'Hondt, T., "The MiniPas Educational Environment", Programming Technology Lab, Brussels Free University, 1988

[Jokinen 89] Jokinen, M.O., "A Language-independent Prettyprinter", Software—Practice and Experience, Vol. 19(9), 839-856, September 1989

[Kaiser & al. 88] Kaiser, G.E., Feiler, P.H., Jalili, F., Schlichter, J.H., "A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation", Software—Practice and Experience, Vol. 18(8), 733-748, August 1988

[Kiong & Welsh 92] Kiong, D., Welsh, J., "Incremental Semantic Evaluation in Language-based Editors", Software-Practice and Experience, Vol. 22(2), 111-135, February 1992

[Klint 90] Klint, P., "A Meta-Environment for Generating Programming Environments", Report CS-R9064, CWI Department of Software Technology, Novermber 1990

[Koorn 92] Koorn, J.W.C., "GSE : A Generic Text and Structure Editor", Report P9202, University of Amsterdam, Programming Research Group, January 1992

[Leavens 84] Leavens, G.T., "Prettyprinting Styles for Various Languages", SIGPLAN Notices, Vol. 19, No. 2, February 1984

[Lewis & Papadimitriou 81] Lewis, H.R., Papadimitriou, C.H., "Elements of the Theory of Computation", Prentice-Hall, 1981

[Madsen & Nørgaard 87] Madsen Ole Lehrmann, Nørgaard Claus, "An Object-Oriented Metaprogramming System", Computer Science Department, Aarhus University, November 1987

[Magnusson & al. 90] Magnusson, B., Bengtsson, M., Dahlin, L., Fries, G., Gustavsson, A., Hedin, G., Minör, S., Oscarsson, D., Taube, M., "An Overview of The Mjølner/ORM Environment: Incremental Language and Software Development, Technical Report LU-CS-TR:90-57, Department of Computer Science, Lund University, Sweden, 1990

[Marti 91] Marti, R., "Design And Implementation Of A Token-Editor", TOOLS 4, 1991

[Mason & Brown 90] Mason, T., Brown, D., "Lex and Yacc", ed. by Dougherty, D., O'Reilly, 1990

[Minör 87] Minör, S., "A generic Synthesizer and its Implementation", Proceedings of the Fifteenth SIMULA Conference, September 1987

[Minör 88] Minör, S., "Structured Command Interaction based on a Grammar Interpreting Synthesizer", Technical report LU-CS-TR:88-34, Department of Computer Science, Lund University, Sweden, 1988

[Minör 90] Minör S., "On Structure-Oriented Editing", PhD. Thesis, Department of Computer Science, Lund University, Sweden, 1990

[Minör 91] Minör, S., "Interacting with Structure-Oriented Editors", Technical report LU-CS-TR:91-74, Department of Computer Science, Lund University, Sweden, 1991

[Mjølner 90] Mjølner Informatics ApS, "Sif - A Hyper Structure Editor, Users Guide", Mjølner Informatics Report MIA-90-11(0.1), November 1990

[Mjølner 91] The Mjølner Group, "Mjølner/Orm User's Guide (Version 1.3)", Department of Computer Science, Lund University, Sweden, 1991

[Motet & Billaut 91] Motet, G., Billaut J.C., "Hierarchical Object-Oriented Design Of A Syntactic Editor", TOOLS 4, 1991

[Nijholt 88] Nijholt, A., "Theoretical Computer Science Course II", Brussels Free University, 1988

[Notkin & al. 83] Notkin, D., Habermann, N., Ellison, R., Kaiser, G., Garlan, D., "Respons to Waters' article on structure-oriented editors", SIGPLAN Notices, Vol. 18, No. 4, April 1983

[Oppen 80] Oppen, D.C., "Prettyprinting", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980

[Oxford 92] "Oxford Advanced Learner's Dictionary", Seventh Impression, Oxford University Press, 1992

[Rekers & Koorn 91] Rekers, J., Koorn, W., "Substring Parsing for Arbitrary Context-Free Grammars", SIGPLAN Notices, Vol. 26, No. 5, May 1991

[Reppy & Ganser 86] Reppy, J.H., Ganser, E.R., "A Foundation for Programming Environments", Communications of the ACM, 1986

[Reps & Teitelbaum 87] Reps, T., Teitelbaum, T., "Language Processing in Program Editors", IEEE Computer, November 1987

[Rubin 83] Rubin, L.F., "Syntax-Directed Pretty printing - A First Step Towards a Syntax-Directed Editor", IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983

[Sandvad 89] Sandvad, E., "Hypertext in an Object-Oriented Programming Environment", Report No. DAIMI PB-280, Computer Science Department, Aarhus University, May 1989

[Schwartz & al. 84] Schwartz, M.D., Delisle, N.M., Begwani, V.S., "Incremental Compilation in Magpie", Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, June 1984

[Shani 83] Shani, U., "Should Program Editors not Abandon Text Oriented Commands ?", SIGPLAN Notices, Vol. 18, No. 1, January 1983

[Shneiderman 83] Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages", IEEE Computer, August 1983

[Steyaert & al. 93] Steyaert, P., Codenie, W., D'Hondt, T., De Hondt, K., Lucas, C., Van Limberghen, M., "Nested Mixin-Methods in Agora", Proceedings of the ECOOP 93 Conference, July 1993

[Teitelbaum 80] Teitelbaum, T., "The Cornell Program Syntheziser: A Tutorial Introduction", Technical Report TR 79-381, Revised June 1980, Department of Computer Science, Cornell University, Ithaca, New York, 1980

[Teitelbaum & Chapman 90] Teitelbaum, T., Chapman, R., "Higher-order Attribute Grammars and Editing Environments", Proceedings of the ACM SIGPLAN '90 Conference on programming Language Design and Implementation, 1990

[Teitelbaum & Reps 81] Teitelbaum, T., Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM, Vol. 24, No. 9, September 1981

[Toleman &al. 92] Toleman, M.A., Welsh, J., Chapman, A.J., "An Empirical Investigation of Menu Design in Language-Based Editors", Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Software Engineering Notes, Vol. 17, No. 5, December 1992

[van den Brand 92] van den Brand, M.G.J., "Pregmatic, A Generator for Incremental Programming Environments", PhD. Thesis, Catholic University of Nijmegen, the Netherlands, 1992

[Wait & Goos 84] Wait, W.M., Goos, G., "Compiler Construction", Text and Monographs in Computer Science, Springer Verlag, 1984

[Waters 82] Waters, R.C., "Program Editors Should Not Abandon Text Oriented Commands", SIGPLAN Notices, Vol. 17, No. 7, July 1982

[Yeh & Kastens 88] Yeh, D., Kastens, U., "Automatic Construction of Incremental LR(1)-Parsers", SIGPLAN Notices, Vol. 23, No. 3, March 1988

[Zelkowitz 84] 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, SIGPLAN Notices, Vol. 9, No. 3, May 1984