APPLAB User's Guide
Version 1.2

Elizabeth Bjarnason

LU-CS-IR:96-01

Department of Computer Science

Lund Institute of Technology
Lund University

Box 118, S-221 00 Lund, Sweden

APPLAB
User’ s Guide

(version 1.2 April 1996)

The Software Development Environments Group
Department of Computer Science
Lund University , Sweden

The Softw are Development Environments Group
Department of Computer Science
Lund University
Box118
S-221 00 Lund, Sweden
Email: orm@dna.lth.se

APPLAB - Application Language Laboratory - is the result of continued work on the Orm system whic h w as first developed as part of the Mjølner project.

Contents

1.0 Introduction

APPLAB (*Application Language Laboratory*) is a system that supports language design in an integrated, interactive way making it especially suitable for prototyping (small) domain-specific languages; application languages. The system allows the language designer to work on the language definitions and, simultaneously, experiment with the resulting language. Changes made to the grammars of the language are immediately effective in any program written in that language. APPLAB includes the following functionality:

- *Hybrid grammar-driven structure-oriented editing.* The APPLAB editor is structure-oriented and based on a technique for interpreting grammars. Text editing of structures is supplied by a grammar-interpreting parser component. The interpreting of grammars gives great flexibility and allows the syntax for a language and a program in that language to be edited at the same time. The effects of changing the language are immediately made visible in the program and new language constructs can be tried out immediately.
- *Static semantics.* The support for static semantics is supplied by a demand-attribute evaluator for Door Attribute Grammars. The attributes defined for a language structure of a document can be listed and evaluated by a menu command.
- *Grammar tools.* A set of grammar tools are available to support the user in defining new grammars and editing programs.

1.1 System Requirements

Machines	Sun SPARC workstations
Operating system	Solaris version 2.3 or later or SunOS release 4.1 or later
Window-system	OpenWindows 3.0 or later or X-11 release 4 or later

It is recommended (but not necessary) to use “mouse-moved” focus rather than “click-to-type” focus in the setup of OpenWindows. The cut/copy/paste functionality will then work better. Consult the installation guide for details.

1.2 Scope and Usage

APPLAB is an evolving research system rather than a supported product. Although it contains several bugs it is a working system and can be used for designing smaller languages.

We do not guarantee any correction of bugs or compatibility with future versions of APPLAB. We are, however, very interested in comments on the system and bug reports (mail to <Elizabeth.Bjarnason@dna.lth.se>).

1.3 How to Use This Guide

Go through the guided tour of chapter 2.0, and then create a new grammar from scratch, following the instructions in section 4.3.

This guide also contains chapters of reference manual character. Chapter 3.0 describes basic interaction techniques used, i.e. how to interact with windows, menus, etc. in the system. Chapter 4.0 describes the grammar editor. Chapter 5.0 describes the grammar formalisms. Chapter 6.0 describes the available grammar tools. Chapter 7.0 describes the version handling system which applies to grammars. Chapter 8.0 describes how grammar documents are stored as files on the Unix level, and variations on how to start the APPLAB system.

Finally, there is a chapter 9.0 on troubleshooting.

1.4 More Literature

APPLAB is the result of further development of the Orm Programming Environment that was initially developed as part of the Nordic research project Mjølner. The main research results from the Mjølner project are summarized in a book:

Object-Oriented Environments: The Mjølner Approach.
Editors: Jørgen Lindskov Knudsen, Mats Löfgren, Ole Lehrmann Madsen, Boris Magnusson.
PRENTICE HALL, the object-oriented series, 1994.
ISBN: 0-13-009291-6

More detailed account of the techniques used in developing the Orm system are available in Ph.D. theses and research reports from Dept. of Computer Science, Lund University. Many of the reports are available electronically via anonymous ftp:

```
mjolner.dna.lth.se
```

or via world wide web:

```
http://www.dna.lth.se/Research/ProgEnv/ProgEnv.html
```

1.5 Important Notice on Copying, Removing, and Renaming Documents

APPLAB document filenames have the extension ".gram". However, for each grammar document basefile "lang.gram" there are a number of additional revision files with names ".lang.gram_n".

In copying, removing, or renaming grammar documents all these files must be treated consistently. It is important that the basefile and the revision files are kept together in the same directory and that they are consistently named. Therefore, you should not move, copy, or remove any of these files using normal UNIX commands. Instead, use the following scripts:

`> ormmv lang.gram x.gram` to change the name of a grammar

```
> ormmv lang.gram dir        to move a grammar to another directory
> ormcp lang.gram x.gram     to copy a grammar
> ormcp lang.gram dir        to copy a grammar to another directory
> ormrm lang.gram            to remove a grammar
```

These scripts work similarly to the standard UNIX `mv`, `cp`, and `rm`, but they do not take options or multiple arguments.

1.6 Acknowledgments

I would like to thank Görel Hedin for introducing me to the concept of application languages, and for guiding me through the implementation work with APPLAB. I would also like to thank Klas Nilsson and Mats Nyberg for useful comments and questions on APPLAB.

The following people were involved in the implementation of the Mjølner Orm system from which APPLAB evolved: Boris Magnusson, Görel Hedin, Sten Minör, Mats Bengtsson, Magnus Taube, Lars-Ove Dahlin, Dan Oscarsson, Göran Fries, Anders Gustavsson, Pär-Anders Aronsson, Roger Henriksson, and Torsten Olsson.

2.0 Grammar Editing. A Guided Tour.[1]

The Application Language Laboratory, APPLAB, is a structure-oriented environment used for editing grammars and programs based on those grammars. It allows experimenting with grammars in an explorative fashion. This guided tour will give an introduction to APPLAB.

Please refer to chapter 4.0[2] for more information on the commands used. For trouble shooting see chapter 2.11.

2.1 Enter the Application Language Lab and a Demo Grammar

- Log on to a SUN sparc and start Open Windows or X windows.
- Copy the APPLAB demo grammar "toy.gram" to a directory of your own, using the `ormcp` script:
  `> ormcp` *`ApplabDir`*`/demo/toy.gram .`
  where *`ApplabDir`* is the location of the APPLAB installation directory
- Start APPLAB on the grammar document "toy.gram".
  `> applab toy.gram -latest`
  A *grammar window* for the latest revision of "toy.gram" appears on the screen.
- Resize the window to fill most of the screen by dragging the lower right window corner with the left or middle button on the mouse.

The different icons contain grammars describing different aspects of the target language. "ABSTRACT" contains the abstract grammar defining the syntactic structure of the target language. "CONCRETE" contains the concrete grammar, which defines the syntactic sugar and screen layout of different constructs in the language. "PARSE" contains the parser grammar which defines the priorities of the different language constructs. "OOSL" contains the semantic grammar, which defines the static semantics of the language. Finally, "TARGET" is a target structure which can be edited in accordance with the abstract, concrete and parse grammars.

(To leave APPLAB, see section 2.12.)

2.2 The Target Window

The target window is used for editing the program according to the abstract and concrete grammars in a structure-oriented fashion. For a more thorough presentation of the functionality of the structure-oriented editor, please refer to Section 4.4.

- Double-click on the "TARGET" icon. The target window is opened showing a program fragment.

1. Users acquainted with the grammar editor of the Mjølner/Orm system can skip sections 2.2-2.6.

2. A more technical description can be found in Minör, S., "On Structure-Oriented Editing", Ph.D. thesis, Dept. of Computer Science, Lund University

- Make the window a little larger. Pin up the editor menu by pressing the right mouse button inside the window and select the “pin” at the top of the menu. Add scrollbars to the window by selecting the ***Scrollbars on/off*** entry under the ***Misc*** entry in the menu.
- Make some minor modifications of the program. A new statement, for instance, is created by selecting one statement by repeatedly clicking at it until the whole statement is selected, and then choosing the ***Expand after*** entry. A new placeholder is inserted which may be expanded to a new statement chosen from the hierarchical menu under the ***Expand*** entry. If an “if statement” is selected a template for that statement is inserted containing placeholders for the predicate and the statements of the then part and the else part respectively. These placeholders may be further expanded by choosing the desired entries in the menu under the Expand menu.

2.3 The Abstract Grammar Window

The abstract grammar defines the structure of the language of the program in the target window. The abstract grammar itself is also represented and edited as a structure. It is edited by means of the same editor as in the target window, just using different (meta-) grammars.

An abstract grammar is similar to a conventional BNF grammar. It contains four different production types; construction, list, alternation, and lexeme productions. They have the following form:

```
Stmt::* Stmt                                   List
Stmt::!Block!IfStmt!WhileStmtAssignStmt        Alternative
IfStmt ::= Exp&Stmt&Stmt                       Construction
Exp::!IdUse!Constant!Add!Mult!Greater!....     Alternative
IdUse::= ID                                    Construction
ID ::LEXEM                                     Lexeme
Add::= Exp&Exp                                 Construction
Block::=Decls&Stmts                            Construction
```

Notice that the productions do not contain any concrete syntax such as keywords and delimiters and that different types (such as construction and list) cannot be nested in one production.

- Open the “ABSTRACT” window by double-clicking.
- Resize the window and add scrollbars as described in the previous section.
- Pin up the editor menu.
- Scroll down to the productions for statements (IfStmt, AssignStmt, etc.). Scrolling a page is done by clicking at the small box at the bottom of the scroll “elevator”.

2.4 The Concrete Grammar Window

The concrete grammar specifies the surface syntax of a language, i.e. the keywords, delimiters and formatting information. It is edited in the concrete grammar window, also by means of the structure-oriented editor.

A concrete grammar consists of productions of different types, one type for each production type in the abstract grammar; constructions, lists, and lexemes. An exception is the alternation productions, which are not specified in the concrete grammar. A concrete grammar corresponding to the above abstract grammar may have the following appearance:

```
Stmts::=                                    List
   before:_
   in:";"<nl>
   after:_

IfStmt::=                                   Construction
   "if "
   @1" then "
   @2" else "
   @3

IdUse::=                                    Construction
   _
   @1_

ID::=                                       Lexeme
   before:_
   after:_

Add::=                                      Construction
   _
   @1" +"
   @2_

Block::=                                    Construction
   "begin">>><nl>
   @1<nl>
   @2<<<<nl>"end"
```

A list production contains specification of the concrete syntax tokens before the first list element, between the elements, and after the last list element. A construction production contains first the leading concrete syntax, and then a reference to each subelement (@n) together with the concrete syntax after that subelement. A lexeme production contains the concrete syntax before and after the lexeme contents.

The concrete syntax is expressed using text (“a text”), indentation (>>>), end indentation (<<<), and newline (<nl>). An empty sequence of concrete syntax is represented with an underscore (_).

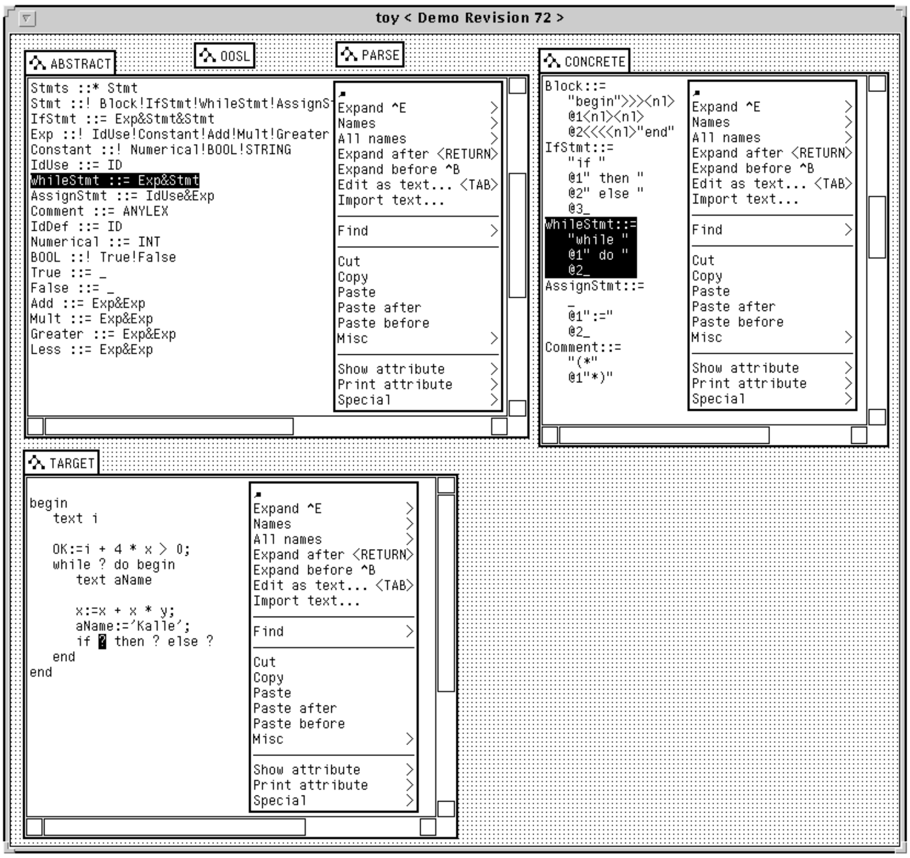
- Open the “CONCRETE” window by double-clicking on it.
- Make the window larger and add scrollbars. Pin up the editor menu. Scroll down to the productions for statements.

After this step the screen will look something like figure 1.

2.5 Make a Small Modification of the Concrete Grammar

The program in the target window is edited in accordance with the grammars. A modification of the grammars will affect the target program. A simple modification is to change the keywords of a statement. To start with, let us change the keywords of a while statement from “while -do” to, for example, “when-do”.

FIGURE 1



Do the following in the CONCRETE window:

- Find the while-stmt production in the concrete grammar by scrolling the window, or by selecting ‘WhileStmt’ from the menu ***Find->Find names->***.
- Select the “while “ token at the second line of the production by clicking at it.
- Choose ***Expand -> Edit lexeme*** from the editor menu. A one-line text editor appears. Change the text from “while “ to “when “ and press return.

Move to the TARGET window.

- Click anywhere in the TARGET window and the program display will change according to the modified concrete grammar.
- Select a statement by clicking on it. Choose ***Expand after*** to get a new placeholder for a statement. Check that the hierarchical menu under ***Expand*** has changed to the new syntax and create a new “when-do” statement by selecting it.

2.6 Extend the Grammar With a New Statement

In the previous section only the surface syntax of the language was modified. We will now extend both the abstract and concrete grammars with a new statement. Since there is a while-statement but no repeat-statement in the language, let us make one. It shall have the following abstract syntax:

```
RepeatStmt ::= Stmt & Exp
```

Do the following in the ABSTRACT window:

- Select a (whole) production and choose ***Expand after***. A placeholder for a production appears. The left-hand side is automatically selected.
- Choose ***Expand -> Edit lexeme*** and enter the name of the production (e.g. “RepeatStmt”) followed by Return. The right-hand side of the production is now automatically selected.
- Choose the **=...&...** entry under ***Expand***, which states the production type is a construction. A placeholder appears. Do ***Expand -> Edit lexeme*** and fill in “Stmt”.
- Do ***Expand after*** and ***Expand -> Edit lexeme*** and fill in “Exp”. We are now ready with the abstract production for the repeat-statement, but we also have to state that the repeat-statement is a statement.
- Scroll the ABSTRACT window upwards until the “Stmt” production is found. It is a fairly long alternation production stating the names of all statements.
- Select one name and do ***Expand after***. Fill in “RepeatStmt” using ***Expand -> Edit lexeme***. Be careful to spell the name right. You could also use the sub menu ***All names->*** and select ***RepeatStmt*** from it. The text ‘RepeatStmt’ is then inserted into the current lexeme.

The abstract grammar is now complete and we have to specify the concrete syntax of the repeat-statement. We suggest the following concrete production:

```
RepeatStmt::=
  "repeat "
  @1 " until "
  @2 _
```

Do the following in the CONCRETE window:

- Select a whole production and do ***Expand after***. A Template for a concrete production appears.
- Fill in the production name “RepeatStmt” using ***Expand -> Edit lexeme***. Be careful to spell it in the same way as in the abstract grammar
- Expand the right-hand side of the production to “? ?” representing a construction production. A template is inserted in which the first placeholder automatically is selected.
- Expand the placeholder to “""” and then choose ***Expand -> Edit lexeme*** to fill in “repeat ”. The second line of the production is now complete.
- Select the “@@” [3]placeholder and do ***Edit lexeme***. Fill in “1”. Expand the following placeholder to “""”, do ***Expand -> Edit lexeme***, and fill in “ until ”. The third line of the production is now complete.

- Select the whole third line by repeatedly clicking at some part of it. Do ***Expand after*** and a new line appears.
- Select the “@@” placeholder and ***Expand -> Edit lexeme*** to “2”. Select the following “?” placeholder and choose ***Cut*** from the menu.
- Save the grammar on file, just to be on the safe side. Select ***Save*** in the popup menu of the grammar document window.

The concrete production is now ready and should look like the production above. We are now ready to use the new language construct in the target program.

Move to the TARGET window.

- Select a statement in the program and do ***Expand after*** .
- Check that the hierarchical menu under ***Expand*** is extended with the new statement.
- Select it and a template is inserted in the program. If you are not fully satisfied with it, modify the concrete production, click in the TARGET window and watch the result of the modification.

You can now use the new statement as any other statement. The expression and statement in the repeat statement may be further expanded. If you are not satisfied with a single statement inside the repeat statement (which can be expanded to “begin ...;... end” to get a list of statements) you can change that in the abstract grammar. Do the following:

- Select the “Stmt” part of the “RepeatStmt” production in the abstract grammar. Do ***Expand -> Edit lexeme*** to change it to “Block”.
- Cut the statement parts of the repeat-statements in the program. You can now insert several statements in the repeat-statement using ***Expand after*** .

2.7 The Parse Grammar Window

The parse grammar contains specifications needed to correctly perform text editing of the language. It is edited in the parse grammar window, using the same editor as the previously described grammar windows.

A parse grammar contains a list of productions of the abstract grammar. An associativity and precedence is defined for each line of productions. A parse grammar for the above abstract and concrete grammars may have the following appearance:

```
Priorities:
   nonassoc: Greater, Less              (1)
<  left: Add                            (2)
<  left: Mult                           (3)

Configuration:
   String Quote: "'"                    (4)
   Comment: "(*" ... "*)"               (5)
```

3. The first @ is part of the unparsing scheme for concrete grammars, while the second @ is a placeholder for an unexpanded lexeme.

(1) The production Greater and Less are non associative and have the same precedence, which is lower than that for the productions Add (2) and Mult (3) which are left associative. Left, right, nonassociative and no associativity can be defined for the productions of the abstract grammar.

The configuration part of the parse grammar is used to configure the text editing for programs based on the current grammar. In the example, (4) a string is specified as being enclosed by the quote character "'", (5) a comment is specified as being recognized by beginning with the token "(*", and ending with "*)".

- Open the "PARSE" window by double-clicking on it.
- Make the window large enough to show its entire contents. Pin up the editor menu.

2.8 Changing the Precedence of Operators

Any structure of the program in the TARGET window can be selected and edited as text. The system translates the edited text into a structure which is then inserted into the selected focus of the TARGET window. When text editing expressions the precedence of the different productions determine which structure the edited text will be translated to.

Do the following in the TARGET window:

- Click at the expression 'i+4*x>0' of the assignment statement until the whole expression is chosen.
- Choose ***Edit as text...*** from the editor menu. A one-line text editor appears containing the text of the current editor focus. Press return. The string is now translated to a structure and inserted into the program.
- Clicking at the + and * symbols of the expression reveals that the resulting structure is equivalent to (i+(4*x))>0.

Now, do the following change in the PARSE window:

- Give Add higher precedence than Mult. Select Add and do ***Cut***. Now select Mult and do ***Paste after***. Select Mult, do ***Cut***. Select the empty marker on the line above. Do ***Expand list***, and then ***Paste***.

Return to the TARGET window:

- Repeat the text edit of the 'i+4*x>0' expression. The text presented for editing is now 'i+(4*x)>0', representing the underlying structure of the expression. Remove the parenthesis and press Return. Investigate the resulting structure. It is now equivalent to ((i+4)*x)>0 since Add has been given higher priority than Mult.

2.9 The OOSL Grammar Window

The static semantics of a language can be defined as an attribute grammar in the OOSL window. It is edited in the OOSL grammar window, by the means of the same structure-oriented editor as the previously described grammar windows.

An OOSL grammar consists of node classes of different types, one type for each production type in the abstract grammar; constructions, alternation, lists and lexemes. An OOSL grammar corresponding to the abstract grammar above may have the following appearance.

```
ANYNODE: node ::!                                   Alternation
{   inh env: ref Set;
    syn undeclaredEnv: ref Set;
    eq son ANYNODE.env := env;
    eq undeclaredEnv := new Set;
    eq son Exp.names := env
};

Exp: node  ANYNODE::!                               Alternation
 {  inh names: ref Set
 };


Mult: node  Exp::=                                  Construction
    (a_Exp1: ref Exp,
    a_Exp2: ref Exp)
 { eq undeclaredEnv :=
       a_Exp1.undeclaredEnv.union( a_Exp2.undeclaredEnv )
 };


Stmts: node  ANYNODE::*Stmt                         List
 { eq undeclaredEnv :=

         AC $X := (new Set | $X.union( son.undeclaredEnv ))
 };


ID: node ::(val: string);                           Lexeme
```

Attributes, and equations for those attributes, can be defined in the different node classes. OOSL is an object-oriented specification language so each node class may inherited attributes and equations from an alternation class. This corresponds to the alternative productions in the abstract grammar. E.g. the `Exp`-production of the abstract grammar above is an alternative production listing all the different kinds of expressions for the language. In the OOSL aspect this corresponds to an alternation class containing all the common attributes and equations for `Exp`, and each specific expression (`Add`, `Mult`, etc.) is represented by a construction class with `Exp` as its superclass.

The OOSL aspect of the demo grammar defines that variables need to be declared in the local or in an outer block to be used. The attribute `env` contains the names of the declared variables at the current node class while the attribute `undeclaredEnv` contains the names of all the used but undeclared variables.

The OOSL attributes and equations need to be compiled to internal structures to be available in the TARGET window. When they are, the attributes are accessible through a submenu.

Do the following in the OOSL window:

- First we need to update the OOSL aspect with a node class for the `RepeatStmt` we added to the grammar earlier. Select a whole node class by repeatedly clicking at it and choose ***Expand after*** . A placeholder for a OOSL declaration appears.

- Choose the ***?: node ????*** entry under ***Expand***, which inserts a placeholder for a node class. The node class-name placeholder is automatically selected.
- Choose ***Expand->Edit lexeme*** and enter the name of the node class (i.e. `RepeatStmt`) followed by return. The prefix placeholder is now automatically selected. Choose the ***?*** entry under ***Expand***, and then ***Expand->Edit lexeme*** and fill in "Stmt". The `RepeatStmt` appears on the right-hand side of the `Stmt` production in the abstract grammar.
- Now choose the ***::= ?*** entry under ***Expand***, which states the node class type to be a construction. A placeholder for a son now appears. The `RepeatStmt` has two sons, `Stmt` and `Exp`. Define the first son by doing ***Expand->Edit lexeme*** and enter "a_Stmt", the name of the son-variable. Define its type by doing ***Expand->Edit lexeme*** at the placeholder after `ref`, and enter "Stmt".
- To insert another declaration of a son select the `Stmt`-son and do ***Expand After***. Let the second son be called `a_Exp` and be of type `Exp`. Specify it in the same manner as the `Stmt`-son.
- If the OOSL window is a bit garbled by now, do a ***Misc->Reunparse*** to refresh it.
- The comment placeholder should now be (automatically) selected. ***Expand*** it to ***NO_comment***.
- The next placeholder contains the body of the node class. Choose the ***{ ? }*** entry under ***Expand***. A declaration body of the node class appears with a selected placeholder.
- Enter an equation by choosing the ***eq ? := ?*** entry under ***Expand***. Define the left hand side of the equation to be the attribute `undeclaredEnv`. Choose ***?*** from the ***Expand***-menu, then ***Expand->Edit lexeme*** and enter "undeclaredEnv".
- Enter the right-hand side of the equation by text editing. Choose ***Edit as text***. A one-line text editor now appears. Enter the text "a_Stmt.undeclaredEnv.union(a_Exp.undeclaredEnv)" and press Return. (You can enlarge the text window by pulling its corners with the left mouse button.) The text will now be processed, and if you have entered it correctly the corresponding structure should have been inserted into the OOSL window. The resulting node class should look like this:

```
RepeatStmt: node  Stmt::=
    (  a_Stmt: ref Stmt,
       a_Exp: ref Exp)
 {  eq undeclaredEnv :=
       a_Stmt.undeclaredEnv.union( a_Exp.undeclaredEnv )
 };
```

- Choose ***Compile OOSL***. The (changed) OOSL grammar is now compiled. If any errors where detected they are marked in the OOSL window with a dotted marker and a message will appear stating the number of errors found. If this happens, press OK to acknowledge that there are compilation errors and look at the error messages by choosing ***Explain next error*** from the menu. If you have entered the node class as described the only possible error should be that the new node class appears before the node classes `Stmt` and/or `Exp` have been defined. If this is the case then move the node class (with ***Cut*** and ***Paste after***) to appear after the declarations of `Stmt` and `Exp`. Redo ***Compile oosl***.

Switch to the TARGET window:

- Select the expression of the `while`-loop. Do ***Show Attributes->*** . A list of all available attributes at that point in the program is shown.
- Choose ***Show Attributes->env*** . The value of the attribute is calculated and displayed in a window. It should contain all the declared variables at this point in the program. Remove the window by doing ***Kill*** .
- Change the declaration of `i` to declare a variable `x`. Re-evaluate the `env` attribute of the expression as before. Note that the variable `x` is now contained in the `env`-attribute instead of `i`.

2.10 The *Names*-menu

In the menu of the TARGET window there is a submenu called ***Names->*** . The contents of this submenu is defined by the OOSL aspect of the base-grammar. The OOSL attribute `names` is used to define the contents of this submenu. In the demo grammar the node class `Exp` has a `names` attribute that defines all (semantically) visible names at the current `Exp`-node in the TARGET window.

- Select the expression of the `while`-loop and choose ***Names->*** from the menu. A submenu containing all visible names will then be displayed. (If the OOSL grammar needs to be recompiled a menu option ***Recompile OOSL grammar*** is given. If so, perform the needed compilation by choosing the given menu option, and then choose ***Names->*** again.) Choosing one of the presented names, e.g. ***OK*** , inserts it into the current selection.

In a similar fashion, variables that are used but not declared, in the demo, can be accessed in a declaration structure.

- Click twice on the OK in the declaration of OK. Choose ***Names->*** and a list of all used, but undeclared variables is presented.

Note! The demo only has support for limited static semantics. For example, the declared types of the variables are not taken into account. This will be possible in future releases of APPLAB.

See sections 4.4.7 and 5.4.3 for further details on the *Names-menu* facility.

2.11 Things That May Go Wrong

One mistake resulting in the changes of a grammar not coming into effect in the target window is misspelling of the production name. Be careful not to add extra blanks and to use the same upper/lower case letters. (Use the ***All names*** facility to avoid this.)

Grammar changes of the abstract, concrete or parse grammars will not come into effect until an operation (click, ***Expand*** ,...) is performed in the TARGET window. Changes of the OOSL grammar need to be compiled to come into effect. When OOSL attributes are accessed (***Names*** , ***Show attribute***) the system checks if the OOSL grammar needs to be recompiled. If so, a menu command for recompiling the OOSL is presented. Recompilation of the current OOSL grammar is only performed if ***Recompile OOSL grammar*** is chosen.

Unexpanded placeholders in the grammars can sometimes give unexpected results. Check that all placeholders are expanded in the grammar.

Inconsistencies may occur between abstract grammars and programs. If an abstract production is modified and the program contains constructs generated from the old production the program is inconsistent with its grammar. The system does not support program transformations from old to new abstract grammars. You have to cut the inconsistent parts manually. However, the system is fairly tolerant to inconsistencies, and does not crash for this reason in general.

Inconsistencies may also occur between abstract and concrete grammar aspects. This may cause the system to crash. E.g. if the abstract aspect contains a production WhileStmt::= Exp&Stmt, and the concrete aspect contains a production

```
WhileStmt ::=
   "while "
   @1 " do "
   @3 _
```

trying to edit a While-statement will cause the unparser to crash.

If the display of any grammar window gets garbled do a ***Misc->Reunparse*** in that window.

If the system should crash while you are working on a grammar revision that revision will be "locked". Unlock it by restarting APPLAB (without the option `-latest`) and do ***Misc->Restore all keys*** in the Evolution Graph Window.

If the system 'freezes' for no apparent reason (no working message, or garbage collection) it may be that the Caps Lock or Num Lock have been engaged. Unengage them for APPLAB to correctly respond to mouse actions.

2.12 Leave the Application Language Lab

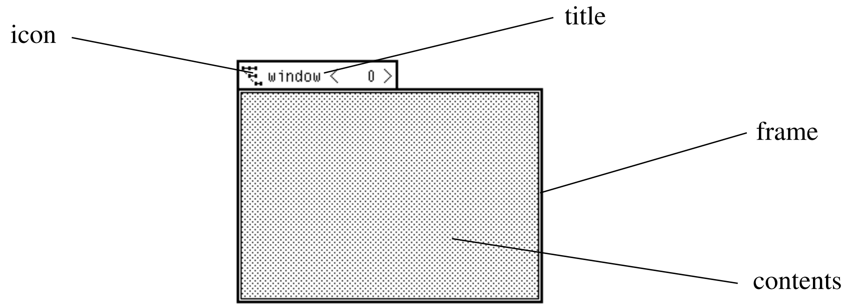
- Do ***Quit*** in the grammar window. If any changes have been made since the grammar document was last changed a prompt box with the alternatives "Save", "Quit", and "Cancel" will appear asking you if you wish to save the changes. Choosing "Save" will save the changes and then quit the program, while "Quit" will quit without saving. The "Cancel" option will neither save nor quit the current grammar document.

3.0 Basic Interaction

This chapter explains how to interact with windows, menus and promptboxes, and how to use the mouse

3.1 Windows

A window is either *opened* or *closed*. An open window has the following parts: an *icon*, a *title*, a *frame*, and a *contents* part. The icon and title parts are optional. When the window is closed, only the icon and title parts are shown.



3.2 The Mouse

The mouse has three buttons: left (LB), middle (MB), and right (RB).
General usage:

LB	Moving windows and selecting items of the window contents.
MB	Resizing windows, initiating text editing.
RB	Bringing up popup menu

In addition to this general usage, there may be specialized behaviour in different windows. The mouse cursor changes shape depending on what it points to. This helps when pointing to small or narrow things, e.g. the window frame.

Note! Do *not* try to “click-ahead”. If the response of the system is not immediate it is better to wait a little while. Otherwise strange things may happen.

3.3 Popup Menus

Different parts of a window may be associated with different popup menus. In particular there is

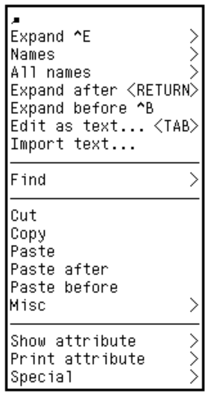
icon menu	on the icon of a closed window
title menu	on the title of a closed window
frame menu	on the icon and frame of an open window

contents menu on the contents part of an open window

The icon and frame menus contain general window commands whereas the title and contents menus are application menus are application specific.

Some menus are “stay-up”-able. These menus have a “pin” at the top. Choosing the pin makes the menu stay up. A pinned up menu disappears when clicking on the pin (RB). Some menu alternatives are “pull-right”-able, indicated by an arrow to the right. Drag the mouse to the right and a new menu appears.

“stay-up”-pin



“pull-right”-arrow

To refer to menu commands we will use the following notation:

choice	select “choice” in a popup menu.
window -> choice	select “choice” in a popup menu of “window”
choice -> subchoice	pull right at “choice” and select “subchoice” in the appearing submenu.
choice -> “txt”	select “choice” in a menu and answer “txt” in a textprompt.

3.4 Basic Commands on Windows

Open	Open window by double clicking (LB) on the icon or title of a closed window, or choose ***open*** in the icon menu.
Close	Close window by click on the icon of an opened window, or choose ***close*** in the frame menu.
Move	Move window by dragging title or frame (LB). (Dragging the icon works only for closed windows.)
Resize	Resize window by holding on to the window frame (MB) and drag. The cursor turns into a lower right corner shape. You may also use the LB for resizing if you start dragging close to one of the corners. Resizing an interior window will automatically resize outer windows as well if this is needed to make all of the window visible.
Front	Bring window to the front by clicking on title or frame (LB). The window is placed in front of sibling windows. Instead of clicking, the ***front*** command in the frame or icon menu can be used.

Back	Bring a window to the back of all its sibling windows by the ***back*** command in the frame or icon menu.

3.5 Additional Menu Commands on Windows

Visible	Resizes outer windows to make sure all of the items of the window contents are applicable
Scrollbars	Adds or removes scrollbars. Only applicable to windows with text contents.
Refresh	Redraws the window or icon.

3.6 Text

Text based interaction is only possible in textprompts.

3.6.1 Positioning and Selection

LB	click	position the text cursor
MB	click	extend the selection

Keyboard arrow keys (up, down, left, right) change the position of the text cursor.

3.6.2 Cut, Copy and Paste

Use the cut, copy, and paste keys on the keyboard or the menu commands in the popup menu.

3.6.3 Scrolling

Add scrollbars to the window by the command in the popup menu or the frame menu.

3.7 Textprompts

Textprompts usually contain an OK and a CANCEL button. Pressing the RETURN key on the keyboard is a short-cut for clicking on the OK button.

Some hints:

- If the text is too large to be completely visible in the prompt you need to resize the promptbox. (Scrolling in promptboxes is currently not implemented.)
- The textprompt can be resized by dragging one of its corners.
- The textprompt can be moved by holding on to its frame and dragging it.

3.8 The “broom”

The APPLAB implementation depends on garbage collection which sometimes takes a few seconds and slightly disturbs user interaction. To give the user feedback on when this happens, the mouse cursor changes shape to a “broom” each time the garbage collector is started, as shown below. If the “broom” appears, simply wait until it disappears again. In case the broom shows up very frequently, you probably need to run APPLAB with a larger heap. This may be the case if you are working with larger grammars. See section 8.1 on how to start APPLAB with a larger heap.

4.0 The Grammar Editor

4.1 Window Structure

The window hierarchy in the grammar editor is the following:

PBO window	Represents all the revisions of the grammar
Evolution Graph window	
	Contains the Evolution graph for the grammar document
Grammar window	Contains the parts of a revision of the grammar, so called *aspect*
Grammar aspects	Contains one part of a grammar, e.g. abstract grammar, concrete grammar, etc.
Target window	Contains a target structure which is edited according to the abstract, concrete and parse grammar aspects.
Text Link window	Similar to a target window but its contents is linked to a text file.

4.2 Entering, Exiting, and Saving a Grammar

Entering — Enter a grammar document "lang.gram" by typing

```
applab lang.gram
```

in a shelltool in the OpenWindows system (or in X windows). The PBO window containing the Evolution Graph of the grammar document now appears. Each "box" represents a revision of the grammar. Double clicking on one of them opens a grammar window for that revision.[4]

Exiting — Exit a grammar revision by ***grammar window -> Quit***. If any changes have been made since the grammar document was last changed a prompt box with the alternatives "Save", "Quit", and "Cancel" will appear asking if you wish to save the changes. Choosing "Save" will save the changes and then quit the program, while "Quit" will quit without saving. The "Cancel" option will neither save nor quit the current grammar document. To finish an APPLAB session each opened grammar window has to be exited and then do ***PBO window->Quit***.

Saving — Save the grammar by ***grammar window -> Save***. It is recommended to save the grammar rather often since the system crashes occasionally. Each save creates a new revision of the program. A revision cannot be changed.

4.3 Creating a New Grammar

A new grammar document is created by starting APPLAB with a new grammar document name with the extension ".gram", e.g. "`applab newlang.gram`". A PBO win-

4. See Section 7.0 for further details on the version control system.

dow with an Evolution Graph containing one (empty) revision node appears. Opening it reveals an empty grammar window. Add an abstract, a concrete, and a target window as described in section 4.6.1. Do not forget to set the startproduction in the target window before editing in the target window (see section 4.4.8). If no startproduction is set the first production of the abstract window is chosen.

4.4 Editing In the Target and Grammar Aspect Windows

Grammars and programs edited in APPLAB are represented internally as abstract syntax trees, *ASTs*. The basic editing technique used is *structure editing* where constructs are inserted into the grammar aspects by selecting them in a menu. The program can contain *placeholders*, which represent incomplete parts of the aspect. The placeholders can be replaced by syntactically correct language constructs by choosing from a menu. This means that it is impossible to construct documents which are syntactically incorrect. APPLAB also supports text editing of structures.

4.4.1 Placeholders

There are three kinds of placeholders.

? The questionmark denotes a placeholder for a language construct, e.g. a statement or an expression. E.g.

```
while ? do
    ?
```

Here the first questionmark is a placeholder for any expression. The second questionmark is a placeholder for any statement.

_ The underscore denotes an empty list. E.g.

```
while ? do
begin
    _
end
```

Here the underscore denotes an empty list of statements

@ The at-sign denotes a placeholder for a *lexeme* in the program. A lexeme placeholder must be replaced by a sequence of characters.

4.4.2 Selection

Language constructs are selected by clicking with the left mouse button. Repeated clicks at the same position extends the selection to the enclosing construct. If the whole AST is selected, a repeated click starts from the “beginning” and selects the smallest construct at the click position.

Unfortunately, there is currently no mechanism for selecting a part of a list, e.g. two statements out of three in a list.

4.4.3 Expansion

A placeholder can be replaced, or *expanded*, by the menu command

Expand ->...

What appears in the submenu depends on the current selection.

Non-placeholder	cannot be expanded
? placeholder	the submenu contains syntactically correct language constructs
_ placeholder	***Expand -> Expand list*** will insert an element into the list.
@ placeholder	***Expand -> Edit lexeme*** will produce a text prompt box where you can type in the new lexeme.

A list can be expanded by selecting one of its elements and selecting ***Expand after*** or ***Expand before***.

4.4.4 Cut, Copy, Paste

The system has an internal "paste buffer" which can hold any language construct. This paste buffer can be used to move language constructs within a grammar aspect. Unfortunately, the paste buffer does not work across different grammar aspects.

Cut	Removes the current selection and replaces it with a placeholder. The removed language construct is placed in the internal paste buffer.
Copy	Copies the current selection into the paste buffer.
Paste	Replaces the currently selected placeholder with a copy of the paste buffer.
Paste after	Inserts a copy of the paste buffer after the currently selected list element.
Paste before	Inserts a copy of the paste buffer before the currently selected list element.

4.4.5 Some Hints

One source of confusion is that some language constructs have the same extent on the screen. This means that one cannot tell which one of them is selected. This is e.g. the case for some lists with one element. One cannot see the difference between selecting the only element and selecting the whole list with the only element. This can be rather confusing. E.g. if you have selected a list with one element where the element is a placeholder, you may think that the placeholder is selected. But the expand command will not work as if you actually have selected the list. In these situations you can find out by repeated selections which construct you have actually selected.

4.4.6 Text Editing

The "expand" command is usually rather tedious to use for constructing and changing, e.g. expressions. Therefore, text editing facilities are also provided for expressions and all other language construct. To edit a structure as text, select it and choose ***Edit as text...*** from the menu. A promptbox appears in which you can edit the selected structure as text. When you are ready, press the RETURN key on the keyboard. The text is then processed, and if it is syntactically correct, the corresponding AST will replace the current selection. If any syntax errors where encountered an error window will appear containing a report on the detected errors, and the edited text is presented to the user for further text editing. Sometimes a window containing grammar errors and warnings appears. E.g. when text editing in the CONCRETE window such a window appears. To avoid it popping up every time a text edit is performed iconize it and it will only appear again if any further grammar errors are detected.

There is a convenient short-cut to text editing. Click on the middle button of the mouse instead of choosing the ***Edit as text*** menu command.

See section 5.3 on how to configure the text editing facilities for a specific language.

4.4.7 Semantic Editing

As a complement to structure-oriented editing and text editing it is also possible to do *semantic editing*. Semantic editing can utilize static semantic information such as scope rules, properties of declared identifiers, and context of the current selection to support high-level editing. At the moment such semantic support is limited to the ***Names*** menu, but in future releases of APPLAB will contain more support for configuring and performing semantic editing of grammars and programs.

The ***Names*** menu is designed to give a submenu of all names (semantically) visible at the current selection (cmp the Names menu in the Mjølner/Orm system). The actual contents of the ***Names*** menu is defined by the OOSL aspect of the base-grammar and can be configured by the language designer. If the contents of the ***Names*** menu has not been defined in the OOSL aspect the current selection it is said to have no semantics defined for it.

See section 5.4.3 for further details on how to define the contents of the ***Names*** menu.

When the user selects a name from the ***Names*** menu the system tries to insert it into the current selection. If the text can not be correctly matched to a structure, error messages are given and the user is presented with a text editing prompt with the chosen text.

The submenu ***All names*** works in the same way as ***Names*** except that it contains a list of all current lexemes in the grammar aspect window. I.e. when a lexeme text is chosen from the ***All names*** -list the system tries to insert it at the current selection.

4.4.8 Other Editor Commands in Grammar Aspect Windows

Edit as text......	Edit the current focus as text.
Import text......	Import the contents of a text file into the current focus.
Find->	Tries to locate a lexeme matching a given search string.
Find...	Asks the user for a search string.
Find pastebuffer	Uses the current contents of the pastebuffer.
Find next	Continues a previous successful search.

Find names->	Presents a list of all lexeme-texts of the current window.
Misc->	
Scrollbars on/off	Add or remove scrollbars to the window
Redisplay	Redisplay the text in the window.
Reunparse	Reunparses the contents of the window according to the concrete grammar and displays it. Used in the target window when the display has been garbled.
Rebuild menus	Recreates the contents of the expand menu.
Save As Text	Saves a grammar aspect or target structure on a text file. You will be prompted for the filename.
Save Interface	Not documented.
Save As Tree	Not documented.
Delete Fork	Deletes this TARGET or grammars aspect window, after confirmation.
Set Title->	Allows the title if a target or grammar aspect window to be changed.
Set Startprod->	Allows a production to be chosen as the start production of the abstract syntax tree in the window. This is typically used when a target window just has been created. The root node of the tree is set to the desired start production of the abstract grammar. Use the "Edit..." alternative to enter an arbitrary production name
Binding Rule->	Not documented.
Show attribute->	Contains the available OOSL attributes at the current focus. Choosing an attribute from the menu evaluates that attribute and displays the result.
Print attribute->	The same as ***Show attribute*** but the value of the attribute is displayed to a textfile. You will be prompted for the filename.
Special->	Contains a few utilities. It varies slightly depending on which aspect it appears in. It basically has the following entries:
Debug->	Undocumented.
Check Grammar	Undocumented.

4.4.9 Short-cuts

There are a number of short-cuts when editing grammar aspect windows.

Mouse short-cuts

- Middle mouse button works as ***Edit as text...***

Keyboard short-cuts

- The arrow keys move the selection.

UP	moves the selection one level up in the AST
DOWN	moves the selection to the first son node in the AST
RIGHT	moves the selection to the next node in a preorder traversal
LEFT	moves the selection to the previous node in a preorder traversal

- Control keys (hold down the CTRL key while typing another character)

^E	Expand list or Expand lexeme
^M (or RETURN)	Expand after
^B	Expand before

- Function keys

  Copy

  Paste

  Cut

  Find
- The TAB key works as ***Edit as text...***

4.4.10 Short-cuts to the Expand Menu

Instead of selecting syntactic constructs from the Expand menu with the mouse, the keyboard can be used. Type the keywords of an entry and press <TAB>. The keyboard entry will be matched with the expand menu entries and the matching construct will be inserted in the program. Each word of the menu entry may be abbreviated. For instance, "while <TAB>", "while do<TAB>", "wh d <TAB>", and "wh<TAB>" will all result in that a while-do template is inserted in the program. Notice that the placeholder symbols ("?" and "...") in the menu entries should not be typed.

These short-cuts are still at an experimental level in the system. They are thus not yet fully supported. The keyboard input is not echoed on the screen, for instance, and the input can not be edited using <DELETE>.

4.5 Editing In the Text Link Window

Working in a text link window is basically the same as working in a target or grammar aspect window. All the editing operations described in section 4.4 are also available in text link windows. The only difference is that a text link window is of a more temporary nature. Each text link window is edited according to a base grammar and its actual contents is linked to a text file. The menu of a text link window contains the additional options

Save & Quit	Saves the contents of the window on the text file and quits APPLAB
Save	Saves the contents of the window on the text file
Quit	Quits APPLAB without saving any changes on the text file.

Text link windows appear when APPLAB is used for socket communication, and when the system is started with the `-text` option. The name of a text file and the base grammar to be used is given. The text is then imported and translated into its corresponding structure according to the given base grammar. The user can then edit the contents of the text file in the structure-oriented editor and save the result, as text, on the original text file. This option is useful for integrating APPLAB in an environment with other (text - based) tools. See section 8.0 for further details on how to use APPLAB in this way.

4.6 Editing In the Grammar Window

4.6.1 Inserting and Deleting Grammar Aspects and Targets

New target and grammar aspect windows can be inserted by menu commands in the grammar window.

Add Abstract	Inserts a new abstract grammar aspect
Add Concrete	Inserts a new concrete grammar aspect
Add Parse	Inserts a new parse grammar aspect
Add OOSL	Inserts a new OOSL grammar aspect
Add T arget	Inserts a new target structure
EDIT	Insert an unnamed grammar aspect

After having selected one of these entries, the system prompts for the name and revision of the meta grammar. Usually you can use the meta grammar name suggested in the prompt box by just pressing return. The revision is selected in a revision graph. To get the latest revision, double-click in the right most box of the graph. The meta-grammar “<SELF>” is suggested when a target window is opened. It means that the target structure is edited using the abstract and concrete grammar aspects in the grammar of which the target window itself is part (this is usually what you want).

A target or grammar aspect window is removed using:

Delete fork	Delete a target or grammar aspect window. The name of the window to be deleted is specified in a prompt box. N.B. Be careful to spell the name exactly as it appears in the title of the window.

4.6.2 Other Commands

The grammar window menu also contains the following entries. We recommend not using the undocumented ones.

Change Meta Grammars	
	Not documented.
Revision	Not documented.
Miscellaneous	Not documented.
Import text...	Imports a program from a textfile to a new grammar aspect.

Import text as...	Same as ***Import text***, but a new metagrammar can be specified
Pretty Print	Generates a pretty print list of the current grammar.
Debug	Not documented.

4.7 Restoring a Crashed Grammar

If APPLAB crashes while you have an opened grammar, you will normally have lost only the changes you made since the last Save command. However, a complication occurs if the system crashes *during* opening or saving the grammar. In this case, the stored grammar may be left in an inconsistent state. Such crashes may occur if you run out of disk quota, heap space, or swap space during these operations.

If the program is in an inconsistent state, APPLAB will crash or give an error message when you start it on the grammar again. Usually, the following error message is given in the Unix shell window:

```
Warning: Wrong version of bytestring
```

In this case, press <CTRL>-C, or press <RETURN> a few times until APPLAB crashes.

To be able to open the grammar again, consistency must be restored by removing the latest revision of the grammar. To do this, proceed as follows.

Save a backup copy of the grammar (in case this cure does not work) by

```
ormcp lang.gram lang-backup.gram
```

(If you are out of disk quota this might not be possible).

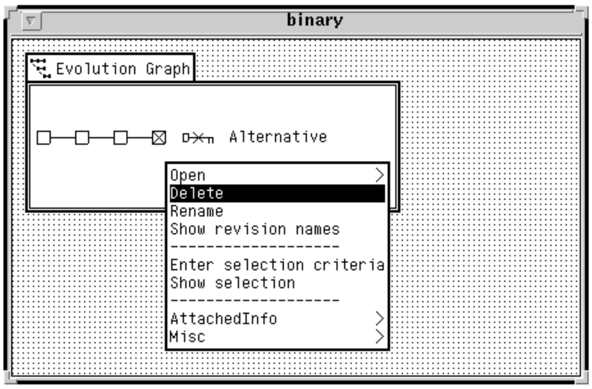
Open the grammar with revision handling by the following command in a unix shell window:

```
applab lang.gram
```

A revision handling window for the grammar appears. The revision handling window contains an “Evolution Graph” window showing small boxes which represent all the existing revisions of the grammar.

1. The revisions are organized in a tree-structured graph of “alternative lines” with older revisions to the left and newer to the right. Often, there is only one alternative line. Each alternative has a “key” which is also displayed in the Evolution Graph window. Since the system crashed the previous time the grammar was edited, the key is probably crossed over, indicating that the key has not been returned properly. To remedy this, do the menu command ***Evolution Graph Window -> Misc -> Restore all Keys***. A promptbox asks if you want to restore all keys. Click on the Yes button.The cross over the key now disappears.
2. The rightmost small box in the Evolution Graph window represents the latest revision. Click on it and it becomes marked. Then delete it by ***Evolution Graph Win-***

dow -> Delete .



A promptbox asks if you want to delete the revision. Click on the Yes button.

3. Check if the cure worked by opening the latest remaining revision of the grammar by double clicking on the rightmost small box of the Evolution Graph Window. If the grammar opens normally you may remove the backup copy. If not, refer to the trouble shooting section in the chapter on version handling, or contact the implementors.

5.0 The Grammar Formalisms

The grammar aspects in the different windows are expressed in different formalisms. The underlying language of the structure in the target window is typically defined by the abstract and concrete grammar aspects in the same grammar. Text editing of the structures in the target window is defined by the abstract, concrete and parse grammar aspects. The abstract, concrete, parse, and oosl aspects are edited according to a fixed metagrammar for each aspect respectively.

5.1 The Structure of Abstract Grammars

An abstract grammar (or more accurately, the abstract aspect of a grammar) only defines the structure of a language. It is similar to a conventional BNF grammar, but since the concrete syntax is omitted it only contains the nonterminals of a conventional grammar. A production can be of four different types: construction, alternation, list, and lexeme. The different types cannot be nested in one production. Some examples are given below:

```
(1) StatementList ::* Statement              List
(2) Statement ::! If ! While ! Assign        Alternation
(3) If ::= Expression & Statement & Statement Construction
(4) Expression ::! ID ! Add ! Sub ! Mult ! DivAlternation
(5) Plus::= Expression & Expression          Construction
(6) ID::= LEXEME                             Lexeme
```

(1) A StatementList consists of a number of statements, specified in a list production ("... ::* ... ").(2) A statement can either be an If, a While, or an Assign statement, specified in an alternation production ("... ::! ... ! ..."); (3) An If statement is an aggregation of an Expression, a Statement, and a Statement, specified in a construction production ("... ::= ... & ..."). (4,5) The productions state that an Expression is an alternation if ID, Add, Sub, Mult, and Div, and that a Plus is a construction of two Expressions. (6) Finally, an ID(identifier) has a textual contents filled in by the user. This is stated in a lexeme production ("... ::= LEXEME").

In the ABSTRACT window, the abstract grammar is edited by means of the structure-oriented editor. It is edited according to the language of abstract grammars. The structure of abstract grammars is defined by an abstract grammar for abstract grammars. Below the structure of abstract grammars is shown. The grammar is written in its own formalism[5]:

```
(1) AbstractGrammar::* Production
(2) Production::= Id & Definition
(3) Definition ::! Alternation ! Construction ! List ! Lexeme
(4) Alternation::* Id
(5) Construction::* Id
(6) List ::= LEXEME
(7) Id ::= LEXEME
(8) Lexeme::=_
```

5. This is actually the grammar interpreted by the structure-oriented editor while editing an abstract grammar in the ABSTRACT window

(1) An abstract grammar consists of a number of productions. (2) A production consists of an Id containing the production name at the left hand side and a definition at the right hand side. (3) The definition states the production type, which can be chosen from Alternation, Construction, List, and Lexeme. (4) An alternation is specifying a number of alternatives. Its structure is a list of Identifiers, each containing the production name of an alternative. (5) A construction consists of a number of descendants. The structure of the construction definition is thus a list of identifiers, each specifying the production name of a descendant. (6) A list specifies that a construct can have an arbitrary number of descendants, all of the same type. The structure of the list definition is a lexeme containing the production name of the construct to be repeated. (7) An Id contains a production name and is thus a lexeme. (8) A lexeme definition is a terminal node in a grammar.

5.2 The Structure of Concrete Grammars

A concrete grammar (or the concrete aspect of a grammar) adds syntactic sugar, delimiters, and formatting information to the abstract grammar. This is used by the structure-oriented editor for presentation of the edited program or grammar. Each production in the abstract grammar, except for alternation productions, has a production in the concrete grammar. The production name in the concrete grammar production has to be the same as in the abstract grammar. An example of a concrete grammar is given below. The grammar describes the concrete aspect of the same language as the example of the abstract aspect in the previous section.

```
(1) StatementList ::=                List
      before:"begin" >>> <nl>
      in:";" <nl>
      after:<<< <nl> "end"

(2) If ::=                           Construction
      "if "
      @1 "then" >>> <nl>
      @2 <<< <nl> "else" >>> <nl>
      @3 <<<

(3) Plus::=                          Construction
      _
      @1 "+"
      @2 _

(4) ID::=                            Lexeme
      before:_
      after:_
```

(1) The production in the abstract grammar describing a StatementList is a list production. In the concrete grammar, the presentation is also specified in a list production with the same name. A list production has three parts, the before, in, and after part. The before part specifies the text and formatting information before the first descendant. It is expressed in a list of templates chosen from text ("..."), indentation (>>>), end indentation (<<<), and newlines (<nl>). In this case the text "begin" is presented followed by increasing the indentation level (>>>) and a new line (<nl>). The in part specifies the concrete syntax separating the descendants of the StatementList, in the example a semicolon followed by a new line. The after part specifies the concrete syntax after the last descendant. In the example, decreasing the indentation level, a new line, and the text "end".

(2) An If statement is a construction production. The concrete production consists of the concrete syntax before the first descendant, “if “, and a number of references to the presentation of the descendants (@...), followed by the concrete syntax after the descendant. In the example “@1” results in presentation of descendant one, the condition expression of the if statement, followed by the text “then”, increasing the indentation level, and a new line. The second descendant, the then-statement, is presented followed by decreasing the indentation level, a new line, the text “else”, increasing the indentation level, and a new line. (3) The Plus production is another example of a construction production. It has no concrete syntax before the first descendant represented by an empty template list (_). The first descendant is followed by the text “+” and the presentation of the second descendant. No concrete syntax follows the second descendant.

(4) Finally, the ID is a lexeme production. It has two parts, “before” and “after”. The before part specifies the concrete syntax presented before the lexeme contents. The lexeme contents is then presented followed by the concrete syntax of the after part. In the example no concrete syntax is preceding or following the identifier.

The structure-oriented editor has knowledge of the structure of concrete grammars. It can be expressed in an abstract grammar for concrete grammars[6]:

```
(1) ConcreteGrammar::* Production
(2) Production ::= Id & Definition
(3) ID ::= LEXEME
(4) Definition ::! Construction ! List ! Lexeme
(5) Construction ::= TemplateList & SonList
(6) TemplateList::* Template
(7) Template ::! Text ! NewLine ! Indent ! EndIndent
(8) Text ::= LEXEME
(9) NewLine ::= _
(10)Indent ::= _
(11)EndIndent ::= _
(12)SonList ::* Son
(13)Son::= SonNumber & TemplateList
(14)SonNumber ::= LEXEME
(15)List ::= TemplateList & TemplateList & TemplateList
(16)Lexeme ::= TemplateList & TemplateList
```

(1) A concrete grammar is a list of productions. (2) A Production consists of an identifier, Id, and a definition. (3) An ID contains a production name and is thus a lexeme. (4) A definition is either a construction, a list, or a lexeme. (5) A construction definition consists of a TemplateList describing the concrete syntax presented before the descendants. The concrete syntax of the descendants is described in the SonList production. (6) A TemplateList consists of a number of templates containing keywords and formatting information. (7) A Template either is a text containing a keyword or delimiter, or formatting information Newline, Indent, or Endindent. NewLine means that a new line is to be inserted, Indent that the indentation level is to be increased, and EndIndent that the indentation level is to be decreased. (8) A Text contains the text to be presented in a lexeme. (9,10,11) The Newline, Indent, and Endindent production do not have any descendants or text contents. They are thus terminals in the grammar. (12) The SonList (in the construction definition) specifies the concrete syntax for the descendants (sons) of a construction. (13) A Son consists of a SonNumber and a TemplateList containing

6. This is the grammar used by the structure-oriented editor when editing concrete grammars in the CONCRETE window

the concrete syntax to be presented after the son. (14) A SonNumber contains the number of the son that is part of a construction. It is a lexeme and its contents should be 1 for the first son, 2 for the second, etc. (15) A List definition has three parts. One TemplateList that specifies the concrete syntax to be presented before the first element of the list, one TemplateList that specifies the concrete syntax separating list elements, and one TemplateList for the concrete syntax after the last list element. (16) A lexeme definition contains a TemplateList containing the concrete syntax before the lexeme contents, and one TemplateList for the concrete syntax after it.

The “bread and butter” of the concrete grammars are the Text, Indent, EndIndent, and NewLine templates. A Text template will cause the lexeme contents of the Text to be presented. The Indent and Endindent will increase and decrease the indentation level respectively. The Indent and Endindent will not come into effect until a NewLine is performed. A NewLine results in a new line in the presentation and an indentation to the current indentation level.

5.3 The Structure of Parse Grammars

A parse grammar (or the parse aspect of a grammar) specifies additional information needed to correctly perform textual editing of a grammar aspect. A *Grammar-Interpreting Parser*[7]-component is used to allow textual editing of any selected structure of a grammar aspect within the structure-oriented editor. The same (abstract and concrete) syntax as is used in the structure-oriented editor is recognized by the parse. Any ambiguities of the abstract grammar are resolved in a parse grammar by specifying the precedence and associativity of ambiguant productions (operators). The parse grammar also partly specifies the lexical syntax of the defined language and configures certain aspects of the parsing component. An example of a parse grammar is given below.

```
Priorities:
left: Add, Sub
< left: Mult, Div

Configuration:
   Comment: "(*" ... "*)"
   String Quote: "'"
```

Each line of the “priority” part contains a list of productions of the same precedence. They are either right, left, or non-associative, or have no associativity. The productions are listed from the lowest to the highest level of precedence. I.e. in the given example Add and Sub, which are left associative, have lower precedence than Mult and Div.

The “configuration” part of the parse grammar is used to specify certain aspects of the lexical syntax, i.e. comments and strings, and to configure the parser component.

APPLAB has knowledge of the structure of the parse grammar. It can be expressed in an abstract grammar for the parse grammar[8].

7. See “A Grammar-Interpreting Parser in a Language Design Laboratory” by E. Bjarnason and G. Hedin, To be presented at the poster session of CC’96

8. This is the grammar used by the structure-oriented editor when editing parse grammars in the PARSE window.

```
(1) ParseGrammar::= priorities&configuration
(2) priorities ::* priority_level
(3) priority_level ::= assoc_kind&prod_list
(4) assoc_kind ::! left!right!nonassoc!NO_assoc
(5) left ::= _
(6) right ::= _
(7) nonassoc ::= _
(8) NO_assoc ::= _
(9) prod_list ::* production
(10)production ::LEXEM
(11)configuration ::* C_config_option
(12)C_config_option::!StringQuote!Comment!ExtendedFocus!NOStri
          ngs!NO_left_factor!NO_priorities
(13)StringQuote ::LEXEM
(14)Comment ::= Keyword&EndComment
(15)Keyword ::LEXEM
(16)EndComment ::! Keyword ! EOLToken
(17)EOLToken ::= _
(18)ExtendedFocus ::= _
(19)NOStrings ::= _
(20)NO_left_factor ::= _
(21)NO_priorities ::= _
```

(1) A parse grammar consists of two parts: one for defining priorities and one for configuring the parser component. (2) The priorities are defined as a list of priority_level:s. (3) Each priority_level has an assoc_kind and a prod_list. (4) An assoc_kind is either left, right, nonassoc or NO_assoc. (5, 6, 7, 8) the left, right, nonassoc and NO_assoc productions do not have any descendants. (9) prod_list is a list of production:s. (10) production contains the name of a production. (11) The configuration is a list of C_config_option:s. (12) C_config_option is either StringQuote, Comment, ExtendedFocus, NOStrings, NO_left_factor or NO_priorities. (13) StringQuote contains a character used to encapsulate strings. (14) Comment contains a Keyword and an EndComment. (15) Keyword is the text of a keyword. (17, 18, 19, 20, 21) the EOLToken, ExtendedFocus, NOStrings, NO_left_factor and NO_priorities productions do not have any descendants.

5.3.1 The Lexical Syntax

In the current version of APPLAB the definition of lexical items are hand-coded, including identifiers, keywords, numerical constants, strings and comments. Start- and end-tokens of strings and comments can be specified in the configure part of the parse grammar.

When doing textual editing in APPLAB special note is taken of the lexeme productions ID, NUM, INT, REAL, and ANYLEX of the abstract grammar. The ID production matches an identifier, the NUM, INT and REAL productions are expected to be used for numerical constants, while ANYLEX represents any combination of characters and should be used for strings and comments. Thus, when defining a grammar for a language which is to be editable as text, the ID, NUM, INT, REAL and ANYLEX productions should be defined as lexeme-productions and used instead of user-defined lexeme-productions. Internally, the predefined lexeme productions, comments, strings, and keywords, are implemented as follows:

```
(1) NUM -> INT | REAL
(2) INT -> Integer
(3) REAL -> RealNumber [Exponent]
(4) ID -> Letter {Letter|Digit|'_'}*
(5) ANYLEX -> (Chars)*
```

```
(6) Comment-> CommentBeginToken ANYLEX CommentEndToken
(7) String -> QuoteChar ANYLEX QuoteChar
(8) Keyword -> (Non-blank)+
(9) RealNumber -> Integer.Integer
(10)Exponent -> (E | e)[+|-]Integer
(11)Integer -> Digit*
(12)Digit -> {0..9}
(13)CommentBeginToken -> (Specified in PARSE)
(14)CommentEndToken ->(Specified in PARSE)
(15)QuoteChar -> (Default: ‘"’, specified in PARSE)
(16)Chars -> {All printable ASCII characters}
(17)Non-blank -> {All printable non-blank ASCII characters}
(18)Letter -> {‘A’..’Z’, ‘a’..’z’}
```

5.3.2 Configure Text Editing

The text editing facilities of APPLAB can be configured for each language described by a grammar in APPLAB. The parser component that performs the text editing works on an internal representation of the current grammar. In order to correctly parse as large a set of grammars as possible this internal grammar representation is restructured to allow the parser to perform correctly. Sometimes, it may be desirable to limit the amount of restructuring done to the grammars but that means that the original grammar must be suitable for parsing without those grammar transformations. The configuring options include:

No left factoring	Common prefixes are not factored out; the grammar should be LL(1)
Do not use priorities	The precedence and associativity of productions specified in the parse grammar should not be used; the grammar must be unambiguous
Extended focus	When selecting a structure for textual editing match to the most general language construct with the same extent on the screen. E.g. for the grammar Exp ::! Add ! ... ! NUM, NUM::LEXEM. Selecting the lexeme-structure of a NUM production is equal to text editing an Exp-structure.

5.4 The Structure of OOSL Grammars

The OOSL grammar (or the oosl aspect of a grammar) is used to define static semantics defined as Door Attribute Grammars[9]. Each production in the abstract grammar has a node class in the OOSL grammar. There are different types of node classes, Alternation(::!), Construction(::=), List(::*) and Lexeme(::LEXEM). They correspond to the alternation, sequence, list and lexeme productions of the abstract grammar. The node class name in the OOSL grammar production has to be the same as in the abstract grammar. An example of an OOSL grammar is given below.

```
Expr: node  ANYNODE::!                          Alternation
   {  inh expectedType : integer;
      loc myType : integer;
      loc sonError : boolean;
```

9. See Hedin, G., “Incremental Semantic Analysis”, Ph. D. thesis, Dept. of Computer Science, Lund University, for a more detailed description of the grammar formalism.

```
        loc locTypeError : boolean;
        eq locTypeError :=
              if expectedType = anyType or
                    myType = anyType then
              false
           else
              expectedType = myType;
        eq error := locTypeError or sonError;
     };

     WhileStmt: node  Stmt::=                    Construction
        (a_Expr: ref Expr,
        a_Stmt: ref Stmt)
      {  eq error :=
           a_Expr.error or
              a_Stmt.error;
         eq a_Expr.expectedType :=
           boolType
      };

      Less: node  Expr::=                        Construction
        (a_Expr1: ref Expr,
        a_Expr2: ref Expr)
      {  eq sonError :=
           a_Expr1.error or
              a_Expr2.error;
         eq son Expr.expectedType :=
           numType;
         eq myType :=
           boolType
      };
```

For each node class attributes and equations defining the value of those attribute can be specified. APPLAB has knowledge of the structure of the OOSL grammar. It can be expressed in an abstract grammar for the OOSL grammar.

5.4.1 Compiling an OOSL Grammar

In order to benefit from the entered OOSL grammar it must be compiled. This is done by choosing the menu alternative ***Compile oosl*** in the menu of the OOSL window, or, the menu alternative ***Recompile OOSL grammar***, which appears when the system discovers the need to recompile. During compilation the OOSL aspect is read, checked for semantic errors and compiled to internal data structures used to perform demand-attribute evaluation. A message is given upon completion of compilation to indicate the outcome of the compilation.

When compiling by issuing the menu alternative ***Compile oosl*** in the menu of the OOSL window an error marker is set at the corresponding language construct in the OOSL window whenever an error is detected. Focusing on this marking and choosing the menu alternative ***Explain error*** gives a short explanation of the error. It is also possible to scroll through the errors with the menu alternative ***Explain next error***. The system then focuses on the next error and presents the error message associated with it. The error markers remain in the window until the next compilation. I.e. even if the error is corrected the error marking will remain in the window (unless the whole construct is deleted) until the grammar construct is correctly compiled.

5.4.2 Demand-Attribute Evaluation

As the user edits a program in the TARGET window it is possible to evaluate the OOSL attributes defined for the current language construct. A list of the accessible attributes is presented when the user chooses ***Show Attributes*** in the menu of the TARGET window. By choosing one of the attributes of this menu it will be evaluated and the result presented in a window on the screen. If something goes wrong during the evaluation, e.g. an attribute does not have an equation defined for it or the target grammar is not complete, then the attribute is said to have an erroneous value. Before presenting the list of accessible attributes the system checks if the OOSL grammar needs recompiling. If so, the menu option ***Recompile OOSL grammar*** is given.

5.4.3 Defining the *Names* Menu

If the current selection has an OOSL attribute with the name `names` the contents of this attribute is presented in the ***Names*** menu. This facility is intended to be used to present a list of all names (semantically) visible at the current selection of the program. It is up to the language designer to define the `names` attribute defining the current names to be presented in the ***Names*** menu. Such an attribute can be defined for any node class of the grammar. If no such attribute is defined for the current selection it is said to have no semantics defined for it. See sections 2.10 and 4.4.7 for a description of the ***Names*** menu facility.

When requesting the ***Names*** menu the system checks if any `names` attribute is defined in the OOSL aspect for the node class of the current selection and if so it is evaluated and its contents presented in a ***Names*** menu in the TARGET window. The system also checks if the OOSL grammar needs to be recompiled. If so, the menu option ***Recompile OOSL grammar*** is given.

5.4.4 Predefined Lexeme Classes

The predefined lexeme-productions, ID, NUM, INT, REAL and ANYLEX, described in Section 5.3.1, are implemented as predefined lexeme classes in OOSL through which the contents of the actual lexeme text can be accessed in the OOSL grammar. The predefined lexeme classes are defined as follows (also found in the LIB aspect of oosl.gram):

```
ID: node  ::!
{  syn val: string;
   syn lex: string
};

NUM: node  ::!
{  syn val: real;
   syn lex: string
};

REAL: node  ::!
{  syn val: real;
   syn lex: string
};

INT: node  ::!
{  syn val: integer;
   syn lex: string
};
```

```
ANYLEX: node  ::!
{  syn lex: string };
```

5.4.5 Predefined Abstract Data Types

In order to more efficiently express static semantics of a language using the OOSL grammar a number of abstract data types have been predefined. The only data type implemented at the moment is a set of strings, but in the future lists, dictionaries and symbol tables are to be predefined. The set is defined as follows (also found in the LIB aspect of oosl.gram):

```
Set: class
    { (* Returns true if the set is empty *)
       empty: func boolean;

       (* Returns true if the set contains item *)
       contains: func boolean
          (item: string);

       (* Adds item to the set *)
       add: func ref Set
          (item: string);

       (* Returns the union of this set and s *)
       union: func ref Set
          (s: ref Set );

       (* Returns true if this set contains exactly the same
          strings as s *)
       equal: func boolean
          (s: ref Set )
    }
```

5.4.6 The Implemented Subset of Door AG

The OOSL compiler and demand attribute evaluator of APPLAB are presently limited to handling only a subset of Door AG constructs. This subset includes

- alt- and cons-classes with (single) inheritance,
- list classes extended with the predefined attributes cardinal and son.pos, and an attribute construction expression. See **List Nodes in OOSL** on page 37 for a description of the added features.
- local, inherited and synthesized attributes,
- virtual functions with parameters,
- classes without parameters,
- types: integer, real, boolean, string, object references
- logical operations: not, or, and, ==, <>, if-then-else,
- arithmetic operations: +, -, *, **, function calls,
- string operations: concat, blanks
- attribute equations,
- collective equations.

Doors and semantic objects have been left out, as well as iterators, collections, conditions, and fix attributes.

5.4.7 List Nodes in OOSL

The list node-construct of OOSL has been extended with the predefined attributes **cardinal** and **son.pos**, and an attribute construction expression. The predefined attribute **cardinal** can be described as follows:

cardinal	attribute of list nodes denoting the current number of sons

In order to construct attribute values based on attributes of the sons of a list node an attribute construction expression has been added. It has the following syntax:

```
AC $<Id> := ( <Start-Exp> | <Loop_Exp>)
```

The **AC**-expression iterates over all the sons of the list node, evaluating <Loop_Exp> for each one of them. The temporary variable $<Id> is used to store the intermediate results and can be accessed in <Loop_Exp>. $<Id> is initiated by evaluating <Start-Exp>. Attributes of the current son are accessed through the predefined reference-variable **son** which can only be used in a <Loop_Exp> of an **AC**-expression. The left-to-right position of the current son can be accessed by **son.pos**. The result of evaluating the **AC**-expression is the resulting value found in $<Id>.

5.5 Editing Metagrammars

All grammars in APPLAB are represented as abstract syntax trees. They are edited by means of the structure-oriented editor according to grammars describing grammars, i.e. metagrammars. All abstract grammars are defined by a grammar defining abstract grammars, containing one abstract and one concrete aspect specifying their structure and presentation respectively. All concrete grammars are defined by a grammar defining concrete grammars, containing an abstract and a concrete aspect. This is also the case for the parse and oosl grammars. The metagrammars, i.e. the grammars specifying grammars, are in turn defined in terms of themselves.

Since the metagrammars are ordinary grammars they can be edited using APPLAB. The concrete aspects of the metagrammars can be edited freely changing the presentation of grammars, e.g. the way an abstract grammar is presented with “... ::= ... & ... & ...”. The abstract aspects of the metagrammars may *not* be changed, since they define the structure of the grammars which is the language APPLAB interprets while editing. Changing the abstract aspects of the metagrammars will cause unpredictable results.

The metagrammars are named abs.gram, con.gram, par.gram, and oosl. gram. They are located in the “grammars” directory. The grammars may be inspected using APPLAB, but we strongly recommend you to *not* edit them.

6.0 Grammar Tools

When defining new languages in APPLAB it is useful to have a few tools that help the language designer in editing and debugging the grammar specifications. Two such simple, rudimentary, tools are included in this release of APPLAB; a *Pretty Printer* and an *OOSL generator*. The *Pretty Printer* compiles the abstract, concrete and parse aspects of a grammar into one list. The OOSL generator is used to generate the outlines of an OOSL grammar aspect corresponding to the abstract aspect.

6.1 The Pretty Printer

The grammar for a language is entered production by production, and information concerning each production is spread over a number of different aspects of the grammar. The *Pretty Printer* collects information about each production from the abstract, concrete and parse aspect, and produces a tree-listing of the grammar. Any undefined productions are marked in the produced list. Defined, but unused, productions are found at the root of a separate tree in the listing. This helps to identify misspelled and/or undefined productions. It also gives a better overview of the defined language since information from the abstract, concrete and parse aspect is compiled into one list. The list is written to a text file specified by the user.

Invoke the pretty printer by the menu command ***Pretty Print*** in the pop-up menu of the grammar window.

6.2 The OOSL Generator

When entering a grammar for a language one usually begins by defining the abstract and concrete aspects. Later on an OOSL aspect may be added. The structure of the node classes has then already been defined in the abstract grammar aspect and entering the corresponding structures once again in OOSL syntax is both tedious and error prone. Instead the OOSL Generator can be used to automatically generate and/or update these structures from the current abstract aspect. The OOSL Generator is invoked by the menu command ***Special->Generate from ABSTRACT*** in the OOSL window. The user is asked to give the name of a general prefix class, i.e. an alternation node class that will be the superclass of all the node classes in the grammar. It is often useful to declare a common node class containing attributes and equations which are general to all the node classes. If no prefix is given, non will be used in the generated structures.

When using the OOSL Generator to update an OOSL aspect messages will be produced in a message-window for any node classes that already exist. If they have a different structure, prefix, or number/type of sons, than is derived from the abstract grammar aspect this is also reported and the node class of the OOSL window is changed to match its abstract aspect. The bodies of existing node classes, containing attributes and equations, are always left intact. The sons of a construction node are named with the prefix ”a_” added to the qualification of the son.

7.0 The Version Handling System

This chapter introduces how to use the finer details of the version handling system of APPLAB and it explains how the version handling system affects the basic operations for entering, exiting and saving grammars. It is not necessary to read this chapter for normal use of APPLAB.

7.1 Introduction

All grammars within APPLAB are uniformly version controlled using the same basic mechanisms. The version handling system supports the user in organizing versions of grammars and maintains relations between those stored objects. Two users may simultaneously work on the same revisions of the same grammar.

The operations for entering, exiting, and saving grammar documents use the version handling system for storing the information. These operations use the version handling system in a straightforward way similar to the traditional way in which programs are stored in a file system. This chapter describes how the version handling system may be used in order to extend the function of the entering, exiting and saving operations.

7.2 Terminology

A grammar document, including all its revisions is called a *Program Base Object* or PBO for short. The *PBOId* is used for identification of a PBO. The revisions of a PBO are organized in an *evolution graph* which describes the development order of the revisions. A series of consecutive revisions in this graph is called an *alternative.* New alternatives may be created by making a branch from a revision in an existing alternative. A *RevisionId* is used for identification of a revision and consists of three names; the *alternative name,* the *revision name* and the *revision number.* Two PBO:s can be connected by a *relation* which describes a dependency between the PBO:s or between two revisions of the PBO:s. A typical example is a program that depends on a grammar. It is possible to store information in *attributes* connected to the revisions and to the PBO. An attribute has an *attribute name* and an *attribute content*. Automatic purge is an example of a facility controlled by an attribute.

7.3 Relations

A relation is used in order to describe a relationship between two PBO:s or even between specific revisions within the PBO:s. The information connected to a relation consists of three parts, the *revision description* which describes the current value of the relation, the *binding type* which is used to describe how the relation should be rebound and the *binding rule* which describes how a revision should be selected in the evolution graph.

- Revision description

  The revision description consists of the PBOId and an Internal number which uniquely identifies a revision within the selected PBO.

PBOId	File name of the PBO or ‘<SELF>’. ‘<SELF>’ is used in order to refer to an open revision despite it not yet being inserted in the evolution graph.
Internal no	Undocumented description.

- Binding type

  The binding type defines how the relation will be rebound.

Explicit	Rebind the relation by letting the user interactively choose PBO and revision.
Fixed	Do not rebind the relation. Uses the revision identification in order to bind the relation. If the revision can not be found the user is prompted to bind the relation interactively.
Dynamic	Rebind the relation according to the binding rule. If the rule can not point out exactly one revision the user is prompted to bind the relation interactively.

- Binding rule

  A binding rule is a wild card specification for selecting revisions in an evolution graph. The wild card specification is a pattern corresponding to the parts of the RevisionId.

```
BindingRule ::= AlternativeName RevisionName RevisionNo
AlternativeName ::= WildCardSpecification
RevisionName ::= WildCardSpecification
RevisionNo ::= WildCardSpecification | NewestRevision | Integer
WildCardSpecification ::= ( Char+ | '*' )*
   Where '*' is a wild card character matching any sequence of
characters.
NewestRevision ::= '>'
```

‘>’ matches the newest revision among the revisions selected by the previous two parts of the binding rule.

Example:

Alternative name: MyAlt*

Revision name: *SELECT*

Revision number: >

This means select the latest created revision with an alternative name starting with ‘MyAlt’ and containing ‘SELECT’ in the Revision name.

7.3.1 Look-up of PBOId

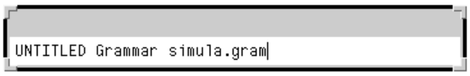
At binding time the PBOId is looked up in the file system in the following order:

1. Current directory.
2. According to the environment variable corresponding to the PBO type.

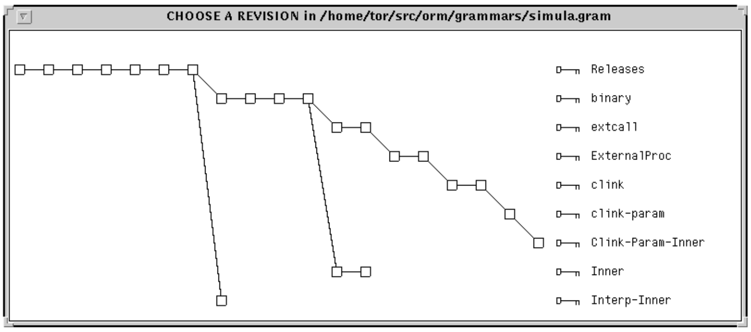
If no PBO is found an explicit binding is initiated.

7.3.2 Explicit Binding of Relations

When the relation is rebound explicitly the user is prompted for the PBOId.



After selecting the PBOId, the full file name of the PBO is looked up and a revision selection prompt is displayed for the selected PBO.



There are two ways of selecting a revision in the revision selection prompt.

- Double click on the revision
- Click on a revision and the use the pop-up menu command

  -> Choose selected

7.3.3 Evaluation of Relations at Binding time

The evaluation of relations at binding time follow two main directions depending on the value of the PBOId.

- <SELF> bound relations

  A relation with PBOId = '<SELF>' is bound to the virtual copy of the current revision. This means that it is possible to refer to an opened revision despite it not being inserted in the evolution graph. The binding type and binding rule is overruled for <SELF> relations.
- File name bound relations

  For relations where the PBOId equals a filename the binding type is interpreted in order to select a revision.

1. Dynamic binding

   The full file name of the PBOId is looked up and the binding rule of the relations is used in order to select a revision. If no revision could be bound an explicit binding is initiated.
2. Explicit binding

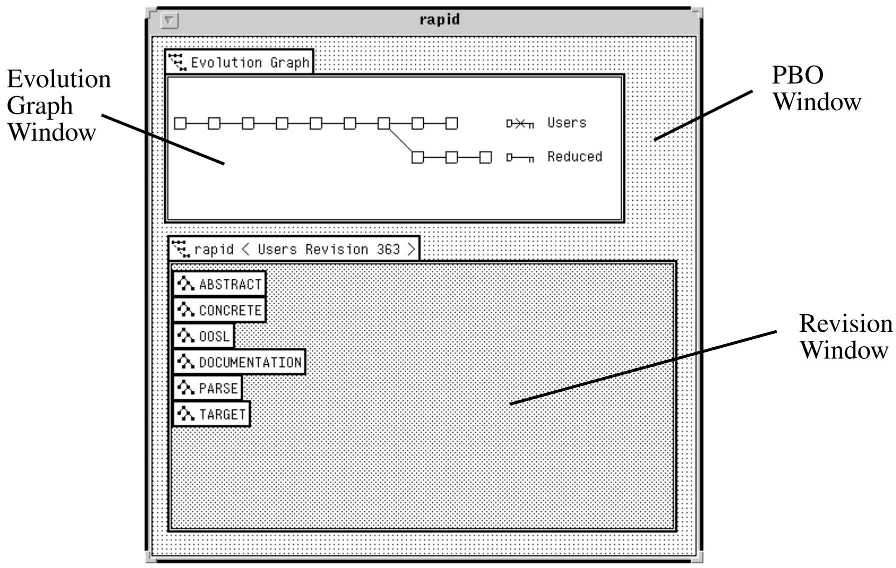
   An explicit binding is initiated. The binding rule is overruled.
3. Fixed binding

The full file name of the PBOId is looked up and the revision description of the relation is used for selecting a revision. If no revision could be bound an explicit binding is initiated. The binding rule is overruled.

7.4 Window Structure

The window corresponding to a grammar represents a single revision of the grammar, and is here called the *revision window.* It is labelled with the PBO-name and the RevisionId. When starting APPLAB an outer window called the *PBO-window* is created which represents *all* of the revisions of the grammar. From the PBO window it is possible to open one or more revision windows of the document. These revision windows are placed inside the PBO window.

Within the PBO-window there is functionality available for viewing and affecting the revisions of the document. The PBO-window is labelled only with the PBOId. Within the PBO-window an *evolution graph window* is displayed. This window shows the evolution graph of the PBO and is used for manipulating the graph and the revisions of the PBO.



7.5 PBO Window

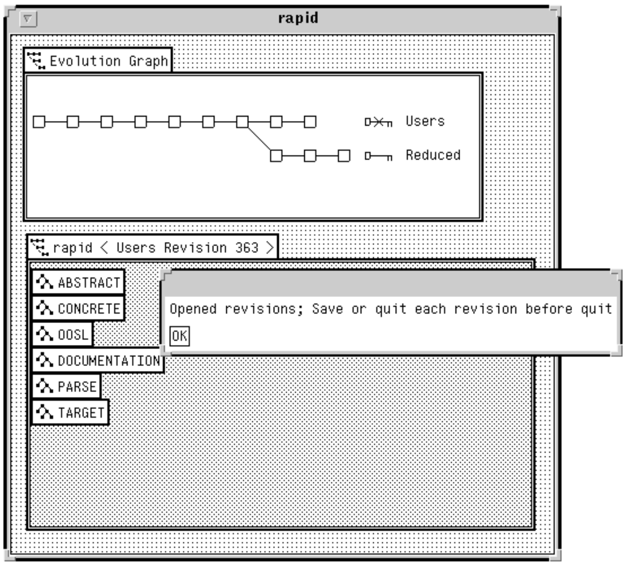
In this section the functionality of the PBO window is described.

7.5.1 Commands

-> ***Quit*** This command removes the PBO window and terminates APPLAB.

Restriction:

It is not allowed to quit from the PBO window if any revision window currently is open.



7.5.2 Undocumented Features

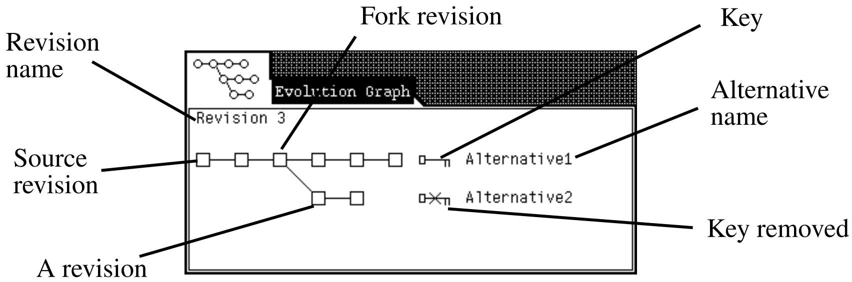
-> Garbage Collect

-> Memory Statistics

-> Memory Statistics without GC

7.6 Evolution Graph Window

The evolution graph window displays all revisions of the PBO.



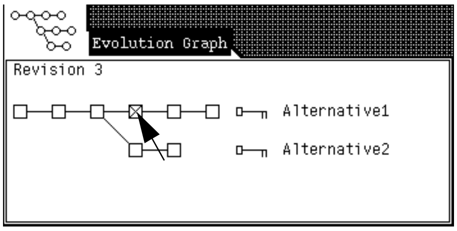
The first revision is called the *source revision* and is the original empty revision. A number of consecutive revisions are called an alternative. A revision may belong to several alternatives. A branch starts in a *fork revision*. For every alternative there is a key to enable synchronization between users. The user first opening the latest revision in an alternative removes the key. Only the user holding the key may add new revisions at the end of the alternative. All other users have to branch and thus create new alternatives.

The names of the alternatives are always visible in the evolution graph window. In order to see the name of a revision move the mouse to the symbol of the revision and the name will be displayed in the upper left hand corner of the window. This operation will only be in effect until the evolution graph is changed.

7.6.1 Interactive Selection of Alternatives and Revisions

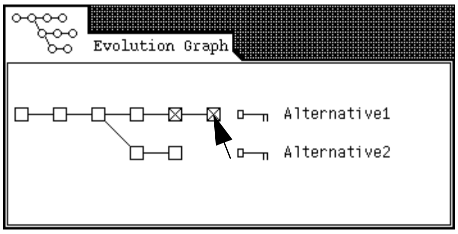
The selection of alternatives and revisions is by direct manipulation with the mouse in the evolution graph. The left mouse button is the selection button. Commands in the revision graph window's popup menu affect the selected objects in the graph. If more than one revision is selected the operation will be applied to each revision in turn.

Select a revision — Click on the revision of your choice. Any previous selections will be de-selected.



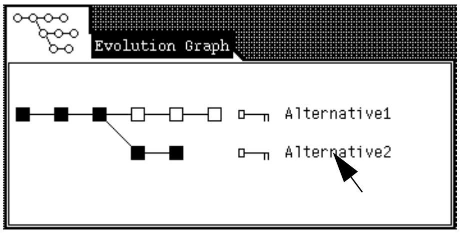
Click on the revision

Toggle revision — Click on a revision while pressing the shift key on the keyboard. The state of the revision toggles between selected and not selected. The operation will not affect previous revision selections. The toggle operation can be used for selecting more than one revision.



Click + Shift

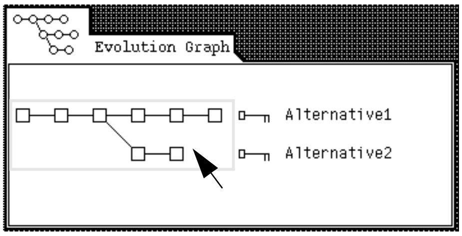
Select alternative — Click on the alternative name. It is only possible to select one alternative at a time.



Click on the
alternative name

Remove selections

It is possible to remove all previous selections by clicking in the area of the evolution graph without selecting a revision.



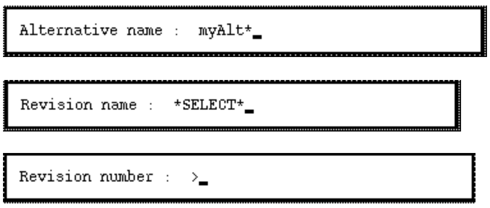
Click inside the
marked area

7.6.2 Selection of Revisions by Rule

An alternative way of selecting revisions. Revisions are selected by specifying the binding rule which is applied to the evolution graph.

-> Enter Selection Criteria

Sets the binding rule.



-> Show selection

Evaluates the binding rule and shows the selected revisions in the evolution graph.

7.6.3 Open

Creates and opens virtual copies of selected revisions. The copies are not added to the PBO and are not shown in the graph until the user saves them. It is possible for a user to have several revisions open at the same time.

-> Open -> Dynamic Rebinding

Opens selected revisions and binds the relations according to their binding types.

-> Open -> Explicit Rebinding

Opens selected revisions and prompts the user for interactive bindings of relations. The binding types of the relations are overruled.

This menu alternative is used for interactively rebinding all relations of the revision, e.g. when the relations should be rebound to another PBO.

-> Open -> Fixed Relation

Opens the revisions using the revision description of the relation. No rebinding of the relations are done. The binding types of the relations are overruled.

This menu alternative is used when the existing binding of the relation should be used, e.g. in the case of error corrections.

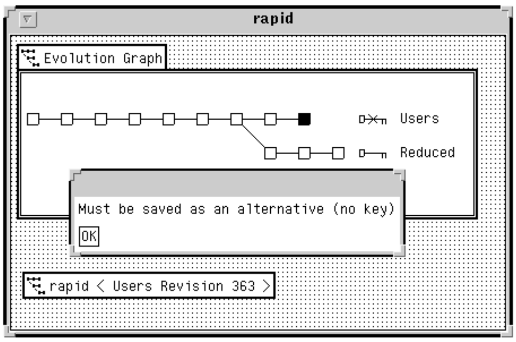
Short-cut: A double click on a revision is a short-cut for choosing a revision and then performing the ***-> Open -> Dynamic Rebinding*** command.

None of the commands will affect the binding rule of the relations. The revision identification will be changed to the bound revision.

When the last revision of an alternative is opened the key is removed from the alternative. Only a revision with a key is allowed to be appended at the end of the alternative. When quit is done on a revision with a key, the key is returned.

Restrictions:

If the key is removed when opening the last revision of an alternative or when opening a revision within the alternative the user will be notified. In these cases the opened revision has to be saved in a new alternative.



7.6.4 Delete

Deletes a selected alternative or selected revisions.

-> Delete

Restrictions:

- If the evolution graph has only one alternative, this alternative cannot be deleted.
- It is not allowed to delete the source revision in the evolution graph (the leftmost box).
- It is not allowed to delete a revision which forms a fork between two alternatives.

7.6.5 Rename

Renames a selected alternative or selected revisions.

-> Rename

The rename operation will not affect the binding of relations since the RevisionId is not used for selecting revisions at binding time.

7.6.6 Attribute Support

An attribute is an attribute name with an associated attribute contents. In the current implementation both the name and the contents are text strings. The attribute can be attached to the PBO or to any revision. Attributes can be used for example to describe the state of a revision in the development process.

-> Attached Info -> Attributes -> Object attributes -> ......

These commands apply to the PBO attributes.

...... ***-> Show***	Show the attributes of the PBO.
...... ***-> Set***	Define a new attribute or change the contents of an attribute in the PBO.
...... ***-> Delete***	Delete a PBO attribute.

<u>Restriction:</u>

- See *Predefined Attributes* (section 7.9) for a summary of all attributes with an already defined meaning.

-> Attached Info -> Attributes -> Revision attributes -> ......

These are the commands applicable to the revision attributes.

...... ***-> Show all***	Show all attributes for each revision in the PBO.
...... ***-> Show***	Show all attributes of the selected revisions.

...... -> ***Set*** Define a new attribute or change the contents of an attribute for every selected revision.

...... -> ***Delete*** Delete an attribute for every selected revision.

7.6.7 Other Commands

-> Show Revision Names

The selected revisions will have their names printed above its symbol.

-> Misc -> Restore all keys

If anyone takes the key from an alternative and the execution of the program is unintentionally interrupted (crashed) the key will be lost. This command resets the key for every alternative.

Note: Do not use this command when there are other simultaneous users of the same PBO.

-> Misc -> Purge Removes revisions in the evolution graph according to the “Purge” attribute.

Uses the “Purge” attribute. See the definition of the Purge attribute in the *Predefined Attributes* (section 7.9) for more information.

7.6.8 Undocumented Features

-> Attached Info ***-> Object Relation Menu -> ......***

-> Misc -> Compact

-> Misc -> Redisplay

-> Misc -> Debug -> ......

7.7 Revision window

The revision window contains the functionality to save and quit a revision of the PBO.

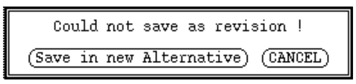
7.7.1 Save

The save command stores the revision in the alternative it was created from and automatically gives the revision a default name. After the save operation, the key or the inserted revision is owned by the current revision.

-> ***Save***

<u>Restriction:</u>

- If the revision is not owner of a key it is not possible to save the revision in the alternative it was created from.



Instead the revision may be save in a new alternative or the user may cancel the save operation. If the revision is saved in a new alternative the user is prompted for an alternative name and a revision name.

7.7.2 Quit

This command lets the user leave a revision without saving changes in the revision.

-> Quit

If the user has received the key it is returned.

7.8 Grammar Document Characteristics

This section describes some characteristics of the grammar documents from the version handling point of view.

7.8.1 Revision Commands

The revision commands are located in the revision window menu as follows:

-> Revision -> Save as revision

This command behaves as the Save command except the user has to provide an explicit revision name for the revision to be saved.

-> Revision -> Save as alternative

The current revision is saved in a new alternative and the user is prompted for names of the new alternative and revision.

-> Revision -> Insert Revision Window

The PBO window and evolution graph window is created (if it was not already present) and the revision window is inserted as a son to the PBO window. This enables the user to view and manipulate the revisions in the revision graph.

7.8.2 Undocumented Features

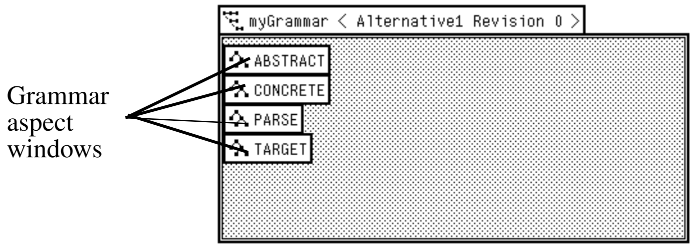
-> Miscellaneous -> Create relation

-> Miscellaneous -> Delete relation -> ......

-> Miscellaneous -> Relation info

7.8.3 Binding of Relations

One relations is connected to each grammar aspect window in the grammar document. The relation is bound to the meta grammar.

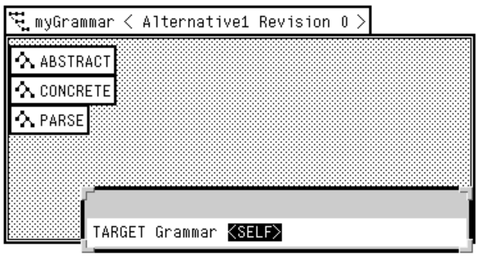


■ Revision window

-> Add (Abstract, Concrete, ...)

When adding a new grammar aspect window the meta grammar is bound using an explicit binding.

A Target grammar window may be bound to the current revision. This is achieved by assigning the PBOId to ‘<SELF>’.



-> Change meta grammars

All relations are rebound using explicit binding.

■ Grammar aspect window

-> Binding Rule -> Set Binding Rule -> Dynamic

Sets the binding type to dynamic rebinding and asks for the binding rule. The command will not affect the current value of the relation.

-> Binding Rule-> Set Binding Rule -> Explicit

Sets the binding type to explicit rebinding. The command will not affect the current value of the relation.

-> Binding Rule -> Set Binding Rule -> Fixed

Sets the binding type to fixed rebinding. The command will not affect the current value of the relation.

-> Binding Rule -> Show Binding Rule

Shows the binding rule of the relation.

7.9 Predefined Attributes

This section introduces the predefined attributes on both the PBO and the revisions.

7.9.1 PBO Attributes

Currently there are only two predefined PBO attributes.

- Purge

  The attribute defines how many revisions should be left at the end of each alternative when the purge command is executed.

  The Purge contents is defined as.

  PurgeContents ::= 'KEEP=' (Integer | 'ALL')

Restrictions:

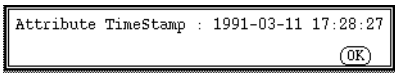
The source revision and fork revisions will not be deleted. This means that the number of remaining revisions is greater or equal to the specified number of revisions to keep. If no Purge attribute exists the Purge command will use the PurgeContents = "KEEP=ALL'.

Examples:

KEEP=4	! Keep the 4 last revisions in every alternative.
KEEP=ALL	! No revisions will be removed.

- TimeStamp

  Contains the creation date and creation time of the PBO. This attribute is automatically set and should never be modified manually.



7.9.2 Revision Attributes

There are currently no predefined revision attributes.

8.0 Unix Level

This chapter explains how APPLAB works at the Unix level. This information can be useful if you want to change the system configuration or if you get into some trouble. For normal usage you do not need to know this.

8.1 The "applab" Script

The `applab` script starts an APPLAB session for a grammar file.

The synopsis for applab is

```
> applab [-m=#] [-g] [-p]
      [-text=importFile:[grammar]:[startProd]]
      [-socket] [-msgport=#]
      filename
```

`-m=#`	start APPLAB with a heap of # Mbyte. Default is 1 Mbyte.
`-g`	print a message for each garbage collection.
`-p`	prints a message concerning the current heap size
`-socket`	start APPLAB with open socket communication
`-msgport=#`	initiate socket communication on port #. Only meaningful when starting APPLAB with the socket option.
`-text=textFile:[grammar]:[startProd]`	starts APPLAB with a text link window (see section 4.5) containing the imported text from `textFile`. The file is assumed to contain a program written in the metagrammar `grammar` starting at production `startProd`. If no grammar name is given the current grammar is used. If no start production is given the first production of the grammar is used.
`filename`	Must have the extension ".gram" . If the file does not exist, a new grammar is created.

8.2 The "ormmessage" Script

The ormmessage script sends a message to an active APPLAB session.

The synopsis for ormmessage is

```
> ormmessage [-ormclient=<orm-machine>] [-msgport=#]
      STOP |
      ((READ|EDIT) <FileName> <grammar> <startProd>)
```

`-ormclient=<orm-machine>`

Send message to APPLAB running on `orm-machine`.

`-msgport=#`	Use port # for socket communication.
`STOP`	Close the current APPLAB session

`(READ | EDIT) <FileName> <grammar> <startProd>`

Initiates a Text Link Window (see section 4.5) with the contents of the text file `FileName` interpreted according to the grammar `grammar` with start production `startProd`

8.3 Grammar Files

Grammars in APPLAB are stored in a binary format as so called *program-base objects*, or PBO:s. For each PBO there is a “basefile” and a number of “revisionfiles” as in the following example:

aGrammar.gram	*(basefile)*
.aGrammar.gram_1	*(revisionfile)*
.aGrammar.gram_5	*(revisionfile)*

The revisionfiles contain the “contents” information whereas the basefile only contains information about the revisionfiles.

Note that the revisionfiles have filenames starting with a dot. They will therefore normally not be listed by the UNIX *ls* command. To list all the files, including the revisionfiles, use e.g. the command

```
> ls -lsa
```

It is important that the basefile and the revisionfiles are kept together in the same directory and that they are consistently named. Therefore, you should not move, copy, or remove any of these files using normal UNIX commands. Instead, use the following scripts:

`> ormmv lang.gram x.gram`	to change the name of a grammar
`> ormmv lang.gram dir`	to move a grammar to another directory
`> ormcp lang.gram x.gram`	to copy a grammar
`> ormcp lang.gram dir`	to copy a grammar to another directory
`> ormrm lang.gram`	to remove a grammar

These scripts work similarly to the standard UNIX mv, cp, and rm, but they do not take options or multiple arguments.

8.4 Environment Variable

APPLAB makes use of the following environment variable:

`setenv MJOLNERHOME dir`

used for finding grammars and icon raster files

8.5 Files in the Release

This section lists the files included in the APPLAB release and their purpose.

```
bin/
   applab       Sun sparc Solaris binary executable for APPLAB
   ormmessage   Sun sparc Solaris binary executable for socket
                communication with an open APPLAB session
   ormcp        script for copying grammar files
   ormmv        script for moving grammar files
   ormrm        script for removing grammar files

demo/           demonstration programs and grammars

doc/            documentation

grammars/
   abs.gram     metagrammar for abstract grammars
   con.gram     metagrammar for concrete grammars
   code.gram    metagrammar for code generation grammar
   doc.gram     metagrammar for documentation grammar
   oosl.gram    metagrammar for oosl grammars
   par.gram     metagrammar for parse grammars
   rapid.gram   metagrammar for rapid grammars
   sem.gram     metagrammar for semantic checking grammar (not
                OOSL)

Images/
   UILrasters/      raster files for icons
      Grey/             raster files for background grey scales

lib/
   greyscales   configuration file for the grey scales used in
                background windows
```

9.0 Trouble Shooting

9.1 Problems With Starting APPLAB

9.1.1 APPLAB Prints Error Message on Standard Output

This may happen after a previous crash during a save operation, which has left the program document you are trying to open in an inconsistent state. Usually, the following error message is given in the Unix shell window:

```
Warning: Wrong version of bytestring
```

In this case, you must remove the last inconsistently saved revision of the document, as described in section 4.7.

9.2 Problems With Saving a Revision

If APPLAB crashes when you save a revision of a document, the reason could be that you are out of disk quota. In this case, you may need to restore consistency in the document files as described in section 4.7

If an empty revision name has previously been given then no further revisions can be saved in that alternative. This is a bug that will be fixed in future releases.

9.3 “Dead” Windows

Sometimes, a window seems “dead” and does not respond to mouse clicks etc. The reason may be that there is an unanswered promptbox connected to the window. If you can not see any promptbox, try closing the outermost APPLAB window and other top-level windows on your screen. The promptbox may be behind one of these windows.

It is also possible that the Caps Lock-, or Num Lock-button is activated. This also causes the window to act “dead”.

9.4 Problems With Editing

9.4.1 Confusing Behaviour at Expand

See section 4.4.5.

9.4.2 The System Hangs

If the whole APPLAB system suddenly hangs when you are text editing, the reason could be that the grammar-interpreting parser has reached a loop in the grammar. You will need to crash the APPLAB system from the Unix shell window (<CTRL>-C) and restart it. Your changes since the last save are lost.

9.5 Frequent “brooms”

The “broom” is the mouse cursor shown during garbage collection (see section 3.8). In case the broom shows up very frequently, you probably need to run APPLAB with a larger heap. This may be the case if you are working with larger grammars. See section 8.1 for how to start APPLAB with a larger heap.

Index