Mjölner/Orm User's Guide

Boris Magnusson et al.

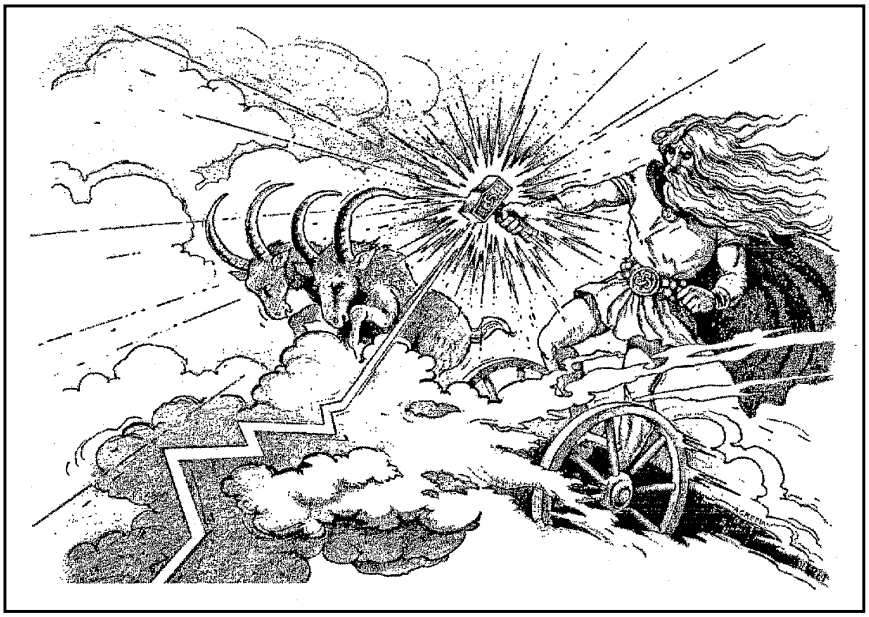
LU-CS-IR:91-4

Department of Computer Science
Lund University

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

Mjølner/Orm
User 's Guide

(version 1.3)



The Mjølner Group
Department of Computer Science
Lund University , Sweden

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

Front cover: The god Thor with his hammer Mjølner

© 1991 by Boris Magnusson, Sten Minör, Görel Hedin, Mats Bengtsson, Magnus Taube,
Lars-Ove Dahlin, Dan Oscarsson, Göran Fries, Anders Gustavsson, Pär-Anders Aronsson

Mjølner is a Nordic research project on programming environments. The Mjølner project is partially financed by Nordisk Industrifond (The Nordic Fund for Technology and Industrial Development) and Styrelsen för Teknisk Utveckling (The Swedish National Board for Technical Development).

Orm is not an acronym, but the name of a fiction character (a Scanian viking).

Contents

1.0 Introduction 1

1.1 System requirements 2
1.2 Scope and usage 2

2.0 Programming in Orm. A Guided Tour. 3

2.1 Enter the environment and a demo program 3
2.2 Browse the source code 3
2.3 Execute the program. 3
2.4 Change the program slightly and then re-execute it. 3
2.5 Observe the program execution by stepwise execution. 4
2.6 Edit the source program 5
2.6.1 Structure editing 5
2.6.2 Text editing 5
2.6.3 Semantic editing 6
2.6.4 Static analysis errors 6
2.7 Exit the system. 8

3.0 Grammar Editing in Orm. A Guided Tour. 9

3.1 Enter the grammar editor and a demo grammar 9
3.2 The target window 9
3.3 The abstract grammar window 10
3.4 The concrete grammar window 10
3.5 Make a small modification of the concrete grammar 11
3.6 Extend the grammar with a new statement 12
3.7 Things that may go wrong 14
3.8 The SEMANTIC and CODEGEN windows 15

4.0 Basic Interaction 16

4.1 Windows 16
4.2 The Mouse 16
4.3 Popup menus 17
4.4 Basic commands on windows 17
4.5 Additional menu commands on windows 18
4.6 Text 19
4.6.1 Positioning and Selection 19

4.6.2 Cut, copy and paste 19
4.6.3 Scrolling 19
4.7 Textprompts 19

5.0 The Orm File Browser 20

6.0 The Orm Programming Environment 22

6.1 Window structure 22
6.2 Entering, Exiting, and Saving a program 24
6.3 Editing in Declarations and Body windows 25
6.3.1 Placeholders 25
6.3.2 Selection 25
6.3.3 Expansion 26
6.3.4 Cut, copy, paste 26
6.3.5 Some hints 26
6.3.6 Textual Editing 27
6.3.7 Semantic Editing 27
6.3.8 Incremental Semantic Checking 28
6.3.9 Other commands in Declaration and Body windows 28
6.3.10 Shortcuts 29
6.3.11 Shortcuts to the Expand menu 29
6.4 Block-level Editing 30
6.4.1 Inserting and deleting blocks 30
6.4.2 Editing block titles 30
6.4.3 Copying blocks 30
6.5 Executing programs 31
6.5.1 Terminology 31
6.5.2 Commands 31
6.6 Documenting programs 33
6.7 Modules 33
6.7.1 Quick guide to how to import a module 34
6.7.2 Import declarations 35
6.7.3 Changing and rebinding import declarations 36
6.8 Other commands (printing, unparsing, etc.) 37
6.9 Grammar W indow 37
6.10 Purge 37
6.11 Error recovery 39
6.11.1 Restoring a crashed program 39
6.11.2 Error message when opening a program. 40
6.11.3 Other problems 41

7.0 The Orm Grammar Editor 42

7.1 Window structure 42
7.2 Entering, Exiting, and Saving a grammar 42
7.3 Creating a new grammar 42
7.4 Editing in the target and grammar aspect windows 42
7.5 Editing in the grammar window 43
7.5.1 Inserting and deleting grammar aspects and targets 43
7.5.2 Other commands 44
7.6 The grammar formalisms 44
7.6.1 The structure of abstract grammars 44
7.6.2 The structure of concrete grammars 45
7.6.3 Semantic and code generation grammars 47
7.6.4 Editing metagrammars 48
7.7 Changing a grammar used by the Orm programming environment 48

8.0 The Orm Version Handling System 49

8.1 Introduction 49
8.2 Terminology 49
8.3 Relations 49
8.3.1 Lookup of PBOId 51
8.3.2 Explicit binding of relations 51
8.3.3 Evaluation of relations at binding time 51
8.4 Window Structure 52
8.5 PBO W indow 53
8.5.1 Commands 53
8.5.2 Undocumented features 53
8.6 Evolution Graph W indow 54
8.6.1 Interactive selection of alternatives and revisions 54
8.6.2 Selection of revisions by rule 56
8.6.3 Open 56
8.6.4 Delete 57
8.6.5 Rename 57
8.6.6 Attribute support 57
8.6.7 Other commands 58
8.6.8 Undocumented features 58
8.7 Revision window 59
8.7.1 Save 59
8.7.2 Quit 59
8.7.3 Revision commands 59
8.8 Orm Program Browser Characteristics 60
8.8.1 Save 60

8.8.2 Quit 60
8.8.3 Revision commands 60
8.8.4 Undocumented features 60
8.8.5 Binding of Relations 61
8.9 Orm Grammar Editor Characteristics 62
8.9.1 Save 62
8.9.2 Quit 62
8.9.3 Revision commands 62
8.9.4 Undocumented features 62
8.9.5 Binding of Relations 62
8.10 Predefined attributes 63
8.10.1 PBO attributes 63
8.10.2 Revision attributes 64
8.11 Trouble shooting 64
8.11.1 Can not open a revision 64
8.11.2 Can not save a revision 64
8.11.3 Lost revisions 64

9.0 Orm Simula 65

9.1 Virtual Procedures 65
9.2 Modules 66
9.3 Details 66
9.3.1 Not implemented features 66
9.3.2 Not enforced restrictions 67
9.3.3 Known bugs and errors 67

10.0 Unix level 69

10.1 Running browser and gred “stand-alone” 69
10.1.1 Browser 69
10.1.2 Gred 70
10.2 Program and grammar files 70
10.3 Environment variables 71
10.4 The .ormlog file 71
10.5 Files in the release 71
10.6 Configuring the file browser 72
10.6.1 Configuration file syntax 72

11.0 The OrmSimLib modules 75

11.1 environment.mjol 75
11.1.1 Basic operations 75
11.1.2 Mathematical functions 77
11.1.3 Extremum functions 77

11.1.4 Environmental enquiries 78
11.1.5 Random drawing 78
11.1.6 Temporary output operations 79
11.2 simset.mjol 80
11.2.1 class Linkage 80
11.2.2 Linkage class Link 81
11.2.3 Linkage class Head 81
11.3 text.mjol 82
11.3.1 Text utilities 82
11.3.2 class Text 83
11.4 uil.mjol 84
11.4.1 class Window 84
Index 87

1.0 Introduction

Mjølner is a Nordic research project with the goal of combining the advantages of highly interactive and integrated programming environments with compiled languages featuring block structure and strong typing. One of the results of the Mjølner project is the *Orm* environment.Orm is an interactive compiling environment for object oriented languages. It is based on incremental compilation, loading, and execution, and has a meta-level for grammar based language design. Currently , grammars for Simula have been developed. The environment runs on Sun-4/SP ARC and is available for teaching and research purposes. Orm includes the following functionality .

- *An incremental compiling programming environment for Simula* . The source code is presented by a hierarchy of windows, reflecting the lexical nesting of classes and procedures. Editing inside a class or procedure is structure-oriented, although text editing of expressions and simple statements is also supported. Semantic editing is also supported which takes the structure-oriented approach a step further by making use of static semantic information such as context and visibility of identifiers.

  Programs can be executed and the execution can be observed and controlled by setting breakpoints and stepwise execution. Objects and procedure activations are shown as windows. The object structure can be browsed by using reference variables as a kind of hyperlinks.

  Compilation and loading is automatic and incremental. The user does not issue any commands for this. Syntactic correctness is guaranteed by the structure-oriented editing approach. Feedback on static semantic errors is given after each edit operation by displaying markings on the erroneous constructs. Such errors can be corrected at any time and the execution of erroneous constructs will simply act as a breakpoint during execution. Other facilities available include a module concept and version handling.
- *A grammar editor* . The grammar editor is structure-oriented and based on a technique for interpreting grammars. This 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 .
- *A graphical file browser* . Files and directories are presented as icons and windows. Opening a file icon automatically starts the appropriate application working on the file (similar to the Macintosh finder). Additional functionality is available via menu commands on the icons and in the windows.
- *Tailorability* . The programming environment is grammar driven and can handle programs also in other languages than Simula. T o make Orm work for another language it is only needed to supply another grammar . Since grammars are interpreted, the programming environment does not have to be regenerated. The system is, however , currently not sufficiently

general to completely handle other existing languages. This kind of tailorability is not made available to the user in this release.

1.1 System requirements

Machines	SUN-4/SP ARC
Operating system	SunOS release 4.1 or later
Window-system	OpenW indows 2.0
Primary memory	12 MB minimum, 16 MB recommended
Swap area	30 MB minimum, 40 MB recommended

On special request, the system is also available for SUN-3, but we do not recommend this as the system is rather slow on these machines.

If you have the minimum memory configuration you will only be able to run one program or module at a time.

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

1.2 Scope and usage

Orm 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 programming small things. Programs of about 1000 lines of code can be handled.

Orm has been successfully used in teaching introductory programming. It may be used as a demonstration system in classes on e.g. object-oriented programming, interactive programming environments, and compiler construction. Orm may also serve as an introduction to programming in Simula.

We do not guarantee any correction of bugs or compatibility with future versions of Orm. W e are, however , very interested in comments on the system and bug reports.

2.0 Programming in Orm. A Guided Tour.

This guided tour will show some features of Orm as a programming environment. We assume here that the system has been installed as described in the installation guide.

2.1 Enter the environment and a demo program

- Log on to a SUN-4.
- Start the OpenWindows 2.0 window system:
  ```
  > openwin
  ```
- In a shelltool, go to the Orm demo directory and start Orm.
  ```
  > cd .../Orm/demo
  > orm
  ```
  A directory window appears, showing some demo programs and grammars as icons.
- Double-click on the "fractals" icon (left mouse button). A *program window* for "fractals" appears on the screen. (This may take a few seconds since this involves starting a separate Unix process.)

2.2 Browse the source code

- The *source window* contains the source code of the program. The different program parts, e.g., classes and procedures, are represented by icons or windows. Double-click on the icon FractalExample. A *procedure window* is opened and the procedure can be inspected.
- The different parts of the procedure are represented by icons or windows inside the procedure window. In the opened procedure window, double-click on the icon that looks like a part of a flow chart. A *body window* containing the statements of the procedure body appears.
- Click on the icon of the FractalExample window (The gear wheels in the upper left corner). The window is closed.
- Double-click on the icon FractalExample again. The window will now be opened in the same state as when the last time it was closed, i.e, the body window is still open.

2.3 Execute the program.

- Choose ***Execute*** from the popup menu in the program window (right mouse button). An *observation window* appears.
- Choose ***Start*** from the popup menu in the observation window. The code is now loaded (and generated if needed). After a while, a *program I/O window* appears and a fractal tree is drawn in the window.

2.4 Change the program slightly and then re-execute it.

- Find the *body window* of the procedure FractalExample. It reads:
  ```
  W.MoveTo( 10 , 10 );
  factor := 2;
  ```

```
root :- new Tree( 0 , 9 , 2 );
new branch( 13 , 7 , root , factor ).into( root );
new branch( -13 , 5 , root , factor ).into( root );
new leaf( -2 , 6 ).into( root );
root.Draw( 8 , 250 , 10 )
```

- Add an extra branch to the fractal tree as follows:

  Select the statement

  ```
  new branch( 13 , 7 , root , factor ).into( root )
  ```

  by clicking the left button. Click repeated times at the same spot until the whole statement is selected (shown in inverse video).
- Choose ***Copy*** from the popup menu.
- Choose ***Paste after*** from the popup menu. A copy of the statement is now inserted after the original one.
- Change each of the values 13 and 7 of the new branch as follows.

  Select 13.

  Choose ***Edit as text*** from the popup menu [1]. A promptbox with the old value selected appears.

  Type another value (using the keyboard) and click OK [2].
- Re-execute the program by choosing ***Start*** in the observation window . The changed parts of the program are now reloaded and after a short time the new fractal tree is drawn in the program I/O window .

2.5 Observe the program execution by stepwise execution.

- Pin up the menu of the observation window by choosing the small needle in the top of the menu. Place the menu to the left in the observation window. (Take hold of the edge of the menu and drag it with the left mouse button.)
- Do a ***step/into.*** An *activation window* of the program appears.
- Do a ***step/into*** once more. An activation window of the procedure FractalExample appears. The activation window contains the source code and the local variables of the procedure.
- Do ***step/over*** a number of times until the program execution point is located at “Draw” in the last statement.
- An object structure has now been built. It can be explored by clicking on reference variables which causes the referred *object window* to appear on the screen. Do as follows:

  Click on the “root” reference variable in the FractalExamples activation window . An object window of class T ree appears.

---

1. Clicking the middle mouse button is a shortcut for choosing the ***Edit as text*** command from the menu.

2. Be careful to keep the cursor in the text-line during editing - the cursor should have the shape of a “bar”. Pressing the NEWLINE key has the same effect as clicking OK in the promptbox.

Click on the “localSUC” reference variable in the T ree object window . Another object window appears.

Go on exploring the object structure by clicking on reference variables in the appearing object windows. If the referred object is already on the screen its window is flashed to give you feedback on which one it is. Y ou can close object windows to icons or remove them.

- The program execution point is still at “Draw” in the FractalExamples activation. Do a ***step/into*** . The “Draw” procedure activation turns up inside the “T ree” object (this reflects the lexical nesting of objects and activations).
- Make the “Draw” procedure window larger so all of its contents can be seen. This is done by taking hold of the window border and dragging a corner with the middle mouse button. Using the lower right corner , the surrounding windows will expand if necessary . The menu command ***Wrap*** in the frame menu is also useful.
- Do ***step/over*** a number of times and parts of the fractal tree will be drawn in the program I/O window .
- Do ***continue*** and the program execution will run to its end. The object and activation windows then disappear .

2.6 Edit the source program

- Close the Observation W indow and find the Source W indow . Find the body window of the procedure FractalExample again.

2.6.1 Structure editing

Add a new “while-statement” as follows

- Select one of the statements by clicking the left button. Click repeated times at the same spot until the whole statement is selected.
- Add a new statement by ***Expand After*** . A question-mark placeholder for a new statement appears after the previously selected statement.
- Replace the placeholder by a “while-statement” by
  Expand -> while ? do ?
  (The “***->***” denotes a submenu).

Try further structured editing by selecting placeholders and using the ***Expand*** menu. There are three types of placeholders:

?	construction placeholder
@	lexeme placeholder
_	list placeholder

2.6.2 Text editing

Statements and parts of statements can be edited as text.

- Select a statement or part of a statement.

- Click on the middle mouse button. A textprompt appears with a text copy of the selection.
- Edit the text in the textprompt and click on the OK button. The text is parsed and if it is syntactically legal it replaces the previous selection. If there is a syntactic error in the text you are given a chance to correct it.

2.6.3 Semantic editing

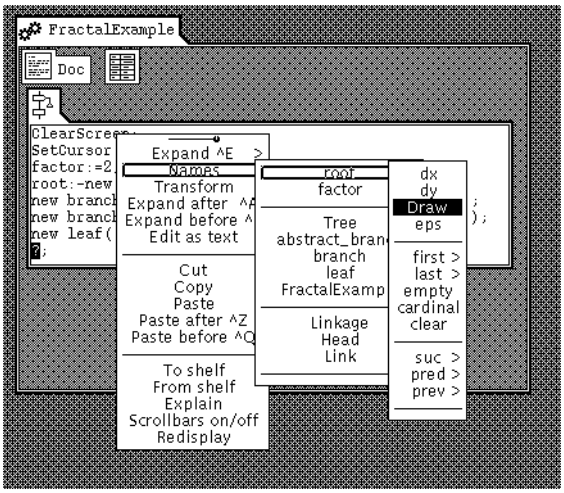
Semantic editing gives you quick access to all visible names in the current context. It provides a powerful way of constructing dot-expressions, procedure calls, etc.

Construct a procedure call, say

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

with semantic editing:

- Add a placeholder for a statement by selecting a statement and then ***Expand After***
- Try the Names menu. ***Names -> root -> Draw*** gives the desired result. Note that the generated procedure call contains the correct number of placeholders for parameters.



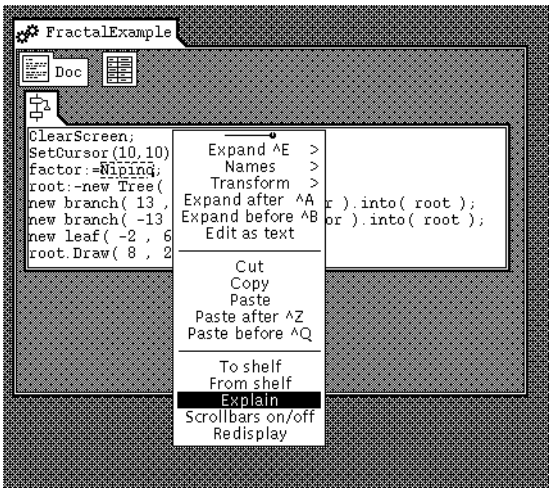
The Names menu shows all the names visible at the current selection. “root” is a reference variable and its pull-right menu shows all the visible procedures (“messages”) and variables in “root”.

2.6.4 Static analysis errors

Static analysis such as typechecking, declaration checking, etc., is done automatically and incrementally after each edit operation. A static error is shown as a dotted rectangle around the erroneous construct.

Change an identifier to one which is not declared.

- Select an identifier in one of the statements.
- Change it to something else, e.g. your name. (Use the middle mouse button to get a textprompt for textual editing.)
- A static analysis error occurs which is shown as a dotted rectangle around the erroneous construct:

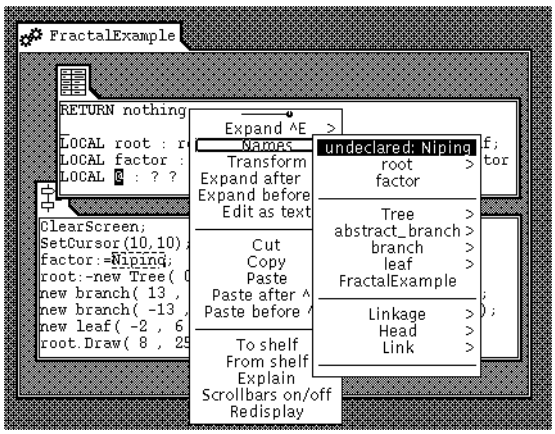


- Select the erroneous construct and ***Explain*** to get an explanation of the error .

You may correct the error any time you wish. An undeclared identifier can be corrected either by changing the identifier or by declaring it. T ry declaring it:

- Find the declaration window for the procedure FractalExample.
- Add a new local variable by selecting one of the existing ones and ***Expand After*** .

- Set the name of the variable to the one used in the previous example. You may do this with the ***Names*** menu since it contains also undeclared variables.



Notice that when you have declared the variable, the dotted rectangle immediately disappears.

2.7 Exit the system.

- Do ***Quit*** in the program window . If you have followed the whole procedure above, the program has been changed and a prompt with three buttons, Save, Quit, Cancel will appear . Since you probably do not want to change this demo program permanently you should press the Quit button. The program window will disappear and its icon reappear in the directory window .
- Exit the file browser by choosing ***Exit*** in the directory window . The directory window disappears and gives control back to the shelltool in OpenWindows.
- Exit OpenWindows by choosing ***Exit*** in the popup menu of the background screen.

3.0 Grammar Editing in Orm. A Guided Tour.

The Grammar Editor , Gred, is a structure-oriented environment used for editing the grammars interpreted by the Orm programming environment. It also allows experimenting with grammars in an explorative fashion.This guided tour will give an introduction to Gred. For further details of using it, please refer to chapter 7 [3].

3.1 Enter the grammar editor and a demo grammar

- If not done already , log on the system and start OpenW indows and Orm as described in section 2.1
- Double-click on the “demogrammar” icon (left mouse button). Gred starts and a window with a number of icons appears.
- Resize the window to fill most of the screen by grabbing a window corner with the 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 defined the syntactic sugar and screen layout of different constructs in the language. Several icons labeled “SEMANTIC-...” describe the static semantics of the language, and “CODEGEN” specifies the translation to intermediate code. Finally , “TARGET” is a target structure which can be edited in accordance with the abstract and concrete grammars.

3.2 The target window

The target window is used for editing the program according to the abstract and concrete grammars. It is similar to the body windows in the Orm programming environment as described in chapter 2, where program bodies are edited by means of the structure-oriented editor . For a more thorough presentation of the functionality of the structure-oriented editor , please refer to section 7.4.

- Double-click on the “T ARGET” icon. The target window is opened showing a program fragment.
- 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

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

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.

3.3 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:

```
StatementList ::* Statement                    List
Statement ::! If ! While ! Assign              Alternation
If ::= Expression & Statement & Statement      Construction
Expression ::! IdUse ! Add ! Sub ! Mult ! Div Alternation
Plus::= Expression & Expression                Construction
IdUse ::= LEXEME                               Lexeme
```

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.
- Scroll down a few pages to the productions for statements (AssignStmt, IfStmt, WhileStmt, etc.). Scrolling a page is done by clicking at the small box at the bottom of the scroll “elevator”. (N.B. the scrolling in the windows sometimes give surprising effects. This is an error that may be corrected in future versions)
- Pin up the editor menu.

3.4 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 consist 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:

```
StatementList ::=                          List
   before:”begin” >>> <nl>
   in:”;” <nl>
```

```
   after:<<< <nl> "end"

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

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

IdUse::=                                Lexeme
   before:_
   after:_
```

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>); No concrete syntax is represented with an underscore (_);

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

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

3.5 Make a small modification of the concrete grammar

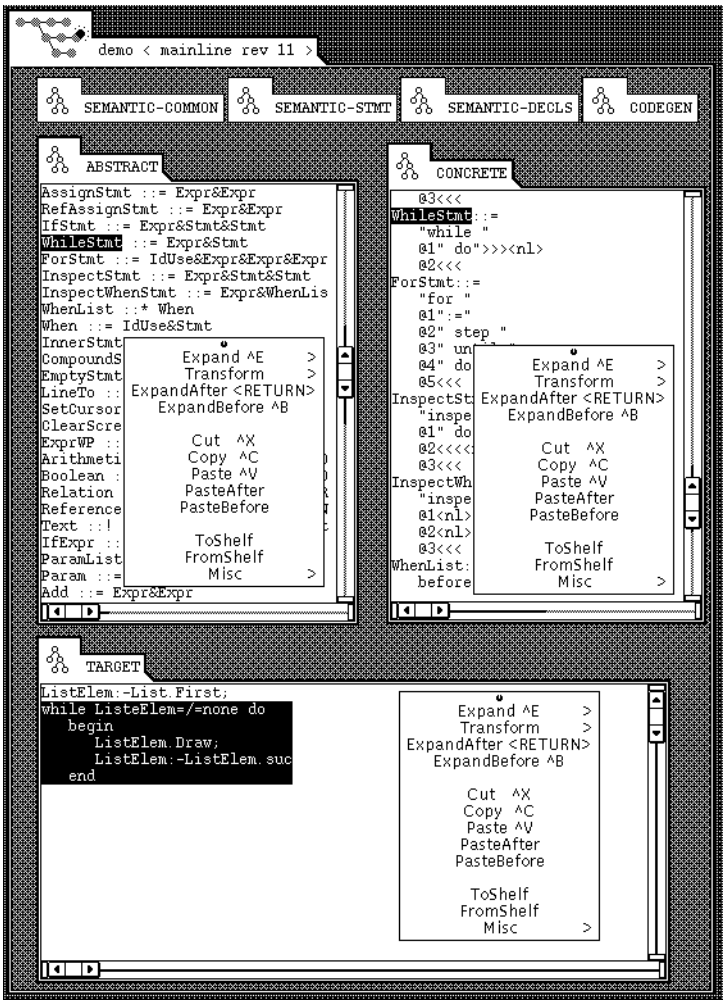
The program in the target window is edited in accordance with the grammars. A modification of the grammars will affect the target program immediately . A simple modification is to change the keywords of a statement.T o start with, let us change the keywords of a while statement from “while -do” to, say , “as long as-do”.

Do the following in the CONCRETE window:

- Find the while-stmt production in the concrete grammar by scrolling the window .
- Select the “while“ token at the second line of the production by clicking at it.
- Choose ***Expand -> Edit lexem*** from the editor menu. A one-line text editor appears. Change the text from “while “ to “as long as “ and press return.

Move to the T ARGET window .

FIGURE 1



- Choose ***Misc -> Reunparse*** from the editor menu in this window . The program display will change according to the modified concrete grammar .
- Choose ***Misc -> Rebuild Menus*** which will generate new expand menus for the modified 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 “as long as-do” statement by selecting it.

3.6 Extend the grammar with a new statement

In the previous section only the surface syntax of the language was modified. W e 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' s make one. It shall have the following abstract syntax:

```
RepeatStmt ::= Stmt & Expr
```

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 lexem*** 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 lexem*** and fill in "Stmt".
- Do ***Expand after*** and ***Expand -> Edit lexem*** and fill in "Expr". W e 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 lexem*** . Be careful spelling the name right.

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

```
RepeatStmt::=
   "repeat" >>> <nl>
   @1 <<< <nl> " 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 lexem*** . 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 lexem*** to fill in "repeat ".
- Select the "repeat" token you just filled in and do ***Expand after*** . Expand the new placeholder to ">>>".
- Select the ">>>" token and do ***Expand after*** . Expand the placeholder to "<nl>". The second line of the production is now complete.

- Select the "@@" [4]placeholder and do ***Edit lexem*** . Fill in "1". Expand the following placeholder to "<<<". Select it, do ***Expand after*** and expand to <nl>. Select the <nl> token, do ***Expand after*** , expand to " ", do ***Expand -> edit lexem*** , 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 lexem*** to "2". Select the following "?" placeholder and choose ***Cut*** from the menu.
- Save the grammar on file, just for safety . Select ***Save*** in the popup menu of the window containing all the grammar windows.

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

Move to the T ARGET window .

- Do ***Misc -> Rebuild menus*** .
- 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, do ***Misc -> Rebuild menus*** and ***Misc -> Reunparse*** 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 production in the abstract grammar . Do ***Expand -> Edit lexem*** to change it to "StmtList".
- Do ***Misc -> Rebuild menus*** and ***Misc -> Reunparse*** in the T ARGET window. Cut the statement parts of the repeat-statements in the program. You can now insert several statements in the repeat-statement using ***Expand after*** .

3.7 Things that may go wrong

One mistake that results in that the changes of a grammar does not come into effect in the target window is that the production name has been misspelled. Be careful not to add extra blanks and to use the same upper/lower case letters.

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

If you forget to do ***Misc -> Reunparse*** and ***Misc -> Rebuild menus*** after a grammar change it will not come into effect.

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 contain 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.

3.8 The SEMANTIC and CODEGEN windows

Gred has been used for editing the semantic and code generation attribute grammars for the Orm programming environment too. However , it does not support interactive evaluation of attributes within Gred at present. The grammars can be modified in Gred, but the effect of grammar modifications have to be evaluated in the programming environment. To get an idea of how the static semantic checking and the code generation is specified in Mjølner/Orm you can open the SEMANTIC and CODEGEN windows.

The static semantics production for the while-statement, for instance, is located in the SEMANTIC-STMT window . Open the window , resize it, add scollbars, and scroll until you find it.

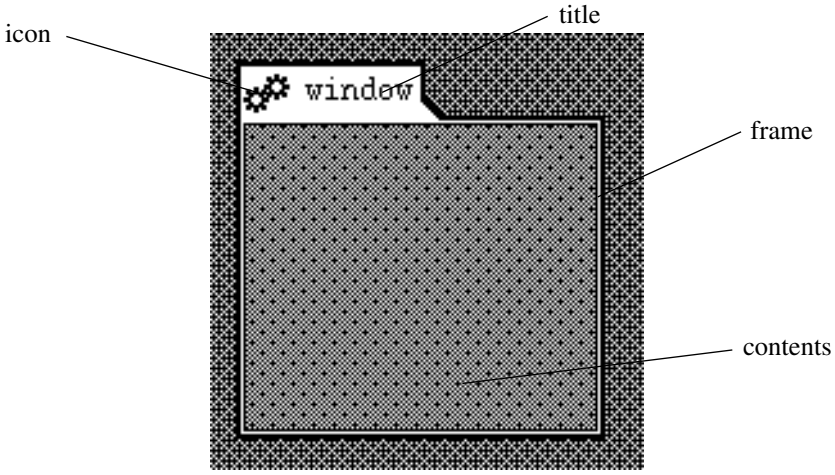
The code generation production for the while-statement is in the CODEGEN window . Open the window , make it larger , add scrollbars, and scroll several pages.

4.0 Basic Interaction

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

4.1 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.



4.2 The Mouse

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

LB	Moving windows and selecting things in window contents.
MB	Resizing windows, initiate text editing.
RB	Brings up popup menu

In addition to this general usage, there may be specialized behavior 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.

4.3 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
window menu	on the icon and frame of an open window
contents menu	on the contents part of an open window

The icon and window 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. 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.

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.

4.4 Basic commands on windows

Open	Open window by double clicking (LB) on the icon or title of a closed window . Or choose ***open*** in icon menu.
Close	Close window by click on the icon of an opened window . Or choose ***close*** in window menu.
Move	Move window by dragging icon, title, or frame (LB).

Resize	Resize window by holding on to the window frame (MB) and drag. The cursor turns into a corner shape to show which corner is dragged. Dragging past another corner of the window will switch dragging to that corner . Resizing an interior window by the lower right corner 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. If the window was already in front of its siblings, the surrounding window is placed in front of *its* siblings. Thus, repeated clicks on an interior window will eventually bring its outermost window to the front. Instead of clicking, the ***front*** command in the window menu can be used.
Back	Bring a window to the back of all its sibling windows by the ***back*** command in the window menu. Instead of using the menu command, one may hold down the shift button on the keyboard and click on the title or frame.

4.5 Additional menu commands on windows

Refresh	Redraws the window or icon.
Organize	Moves interior icons and windows into rows in the currently visible area of the window . This command is useful if you have "lost" a window or an icon, e.g. if it is located outside the visible area of its surrounding window .
Wrap	Resizes a window to be just large enough to make interior windows and icons fully visible. This command is useful to shrink a window as much as possible without hiding interior windows. It is also very useful when interior windows are "lost" (outside the visible area). In contrast to the "Organize" command the interior windows and icons are not moved.
De/Attach	The "Detach" command temporarily detaches a window from the window hierarchy and makes it reside directly on the screen. The detached window is surrounded by a shadow to make it clear that it is not in its proper place in the hierarchy . The "Attach" command moves a detached window back to its proper place in the window hierarchy . Detaching a window is useful if the window hierarchy constrains you too much and makes it hard to see several things at the same time. It is also useful when you are temporarily uninterested in the context of a window and want to use the screen more freely .

4.6 Text

Text based interaction is possible in text prompts and in text-editable windows, e.g. the documentation windows in the Orm programming environment.

4.6.1 Positioning and Selection

LB	click	position the text cursor
LB	double click	select a word
LB	triple click	select a line
LB	down, drag, up	select a sequence of characters
MB	click	extend the selection

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

4.6.2 Cut, copy and paste

Use the cut, copy , and paste keys on the keyboard (L10, L6, and L8 respectively on old SUN keyboards).

4.6.3 Scrolling

The text scrolls automatically as needed when you enter text. Using the arrow keys to change the cursor position also causes the text to scroll. In windows it is usually possible to add scrollbars by a popup menu command.

4.7 Textprompts

Textprompts usually contain an OK and a CANCEL button. Pressing the NEWLINE key on the keyboard is a shortcut for clicking on the OK button.

Some hints:

- If the text is too large to be completely visible in the prompt you can scroll by using the left and right arrow keys on the keyboard.
- The textprompt can be moved by holding on to its frame and dragging it.

  Only a part of the textprompt area is sensitive to keyboard input. If you have “Mouse-moved” focus (as is recommended for other reasons in Orm) rather than “Click-to-type” focus, the mouse cursor has to be in this sensitive area when you type on the keyboard. Y ou can see if the mouse is in the right area by the shape of the mouse cursor .

If you have “Mouse-moved” focus you should leave the mouse alone after issuing a command which brings up a textprompt. Y ou will then not need to reposition the mouse before starting to type. If you move the mouse in this situation you must reposition the cursor onto the line to edit before typing. The cursor will have the form of a “|” (vertical bar) when correctly positioned.

5.0 The Orm File Browser

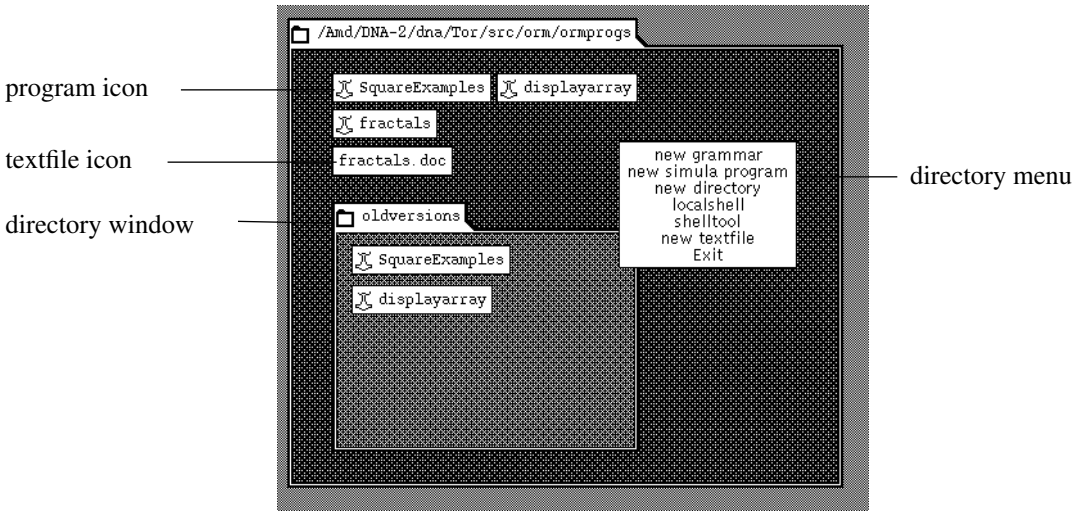
The Orm file browser is started by the command

```
> orm
```

The current directory will then appear on the screen as a window (opened or closed).

The Orm file browser presents directories as windows, nested inside each other reflecting the hierarchical directory structure (FIGURE 2). Files appear in the directories as icons. Orm program icons have the form of a small hammer . Files can be opened, and an appropriate application is then started to take care of further interaction with the file. T o provide additional functionality , both files and directories have associated popup menus. In the standard setup there are the following commands. (The configuration of the file browser is described in section 10.6).

FIGURE 2 Presentation of files and directories in the file browser.



Directory window commands:

new grammar	creates a new grammar
new simula program	creates a new simula program
new directory	creates a new directory
localshell	creates a window which runs csh
shelltool	creates an OpenW indows shelltool
new textfile	creates a new empty textfile
exit	exit the file browser

Commands on icons (only the icon figure, not the title)

open	open the document in the normal way . Double-clicking on the icon is a shortcut for this command.
front	put the icon in front of other windows and icons
back	put the icon behind other windows and icons
refresh	redraw the icon

Commands on Orm programs and grammars:

On the icon figure:

open	open the program the normal way (without revisionhandling). The latest stored version of the program will be opened.

On the icon title:

open with revisionhandling	open the program with the revision handling system included in the programming environment
copy	copies the program to another name or directory
move	moves the program to another name or directory
remove	removes the program

Commands on textfiles:

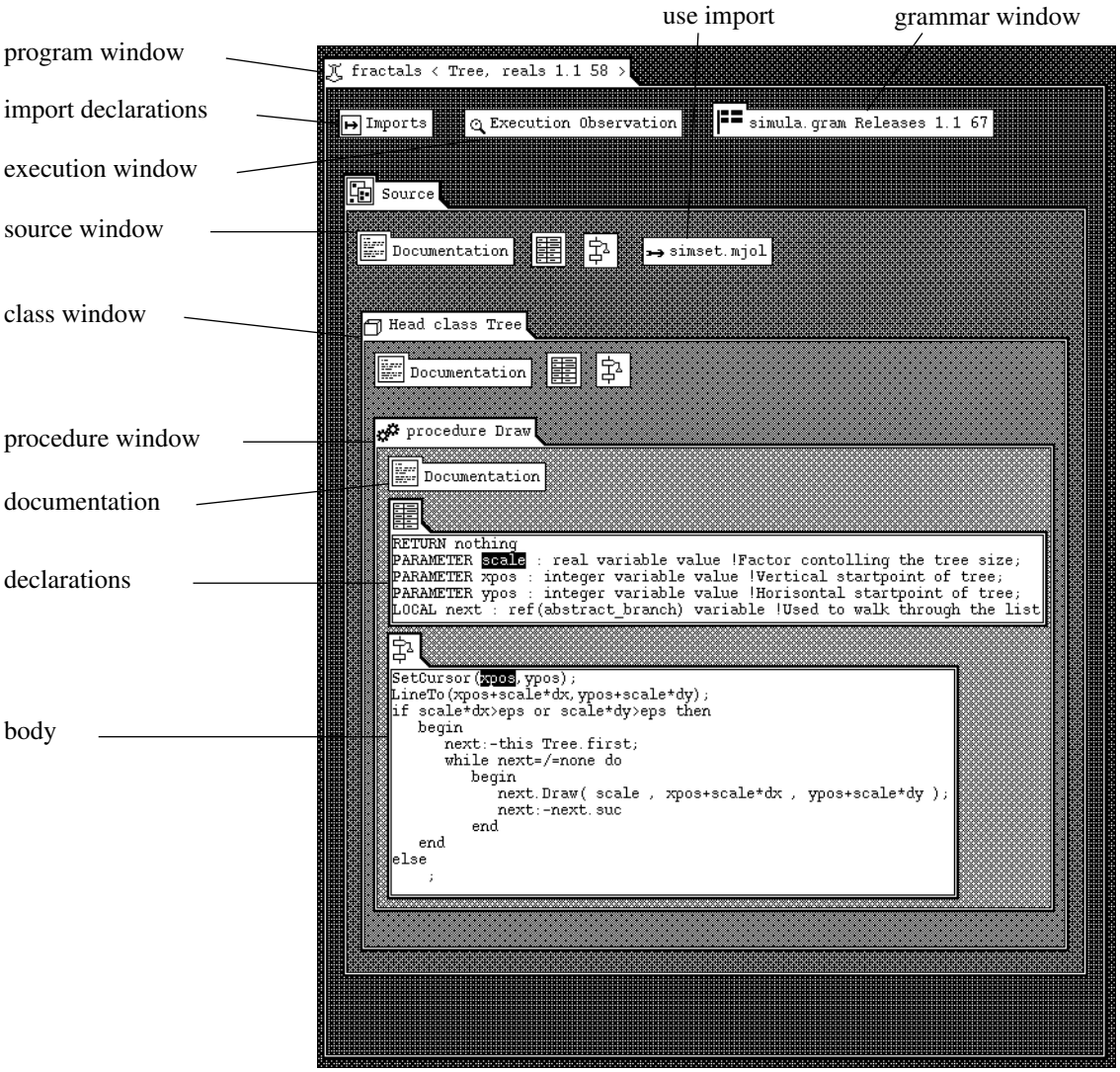
open	runs the OpenW indows `textedit` program on the file

6.0 The Orm Programming Environment

6.1 Window structure

A program and its execution is presented as a hierarchy of windows. In the source code the window nesting reflects the lexical nesting of classes and procedures (FIGURE 3). In the execution, the window nesting reflects the lexical nesting of objects and procedure activations (FIGURE 4).

FIGURE 3 Presentation of the source code



The top of the window hierarchy is as follows.

Directory window	The directory containing the program
Program window	
Source window	Contains the source code of the program
Execution window	Presents objects and activations in an active execution (not always present).
Declared imports	For declaring module imports
Grammar window	For controlling the grammar binding
Program I/O window	Contains the output of the program execution (not always present)

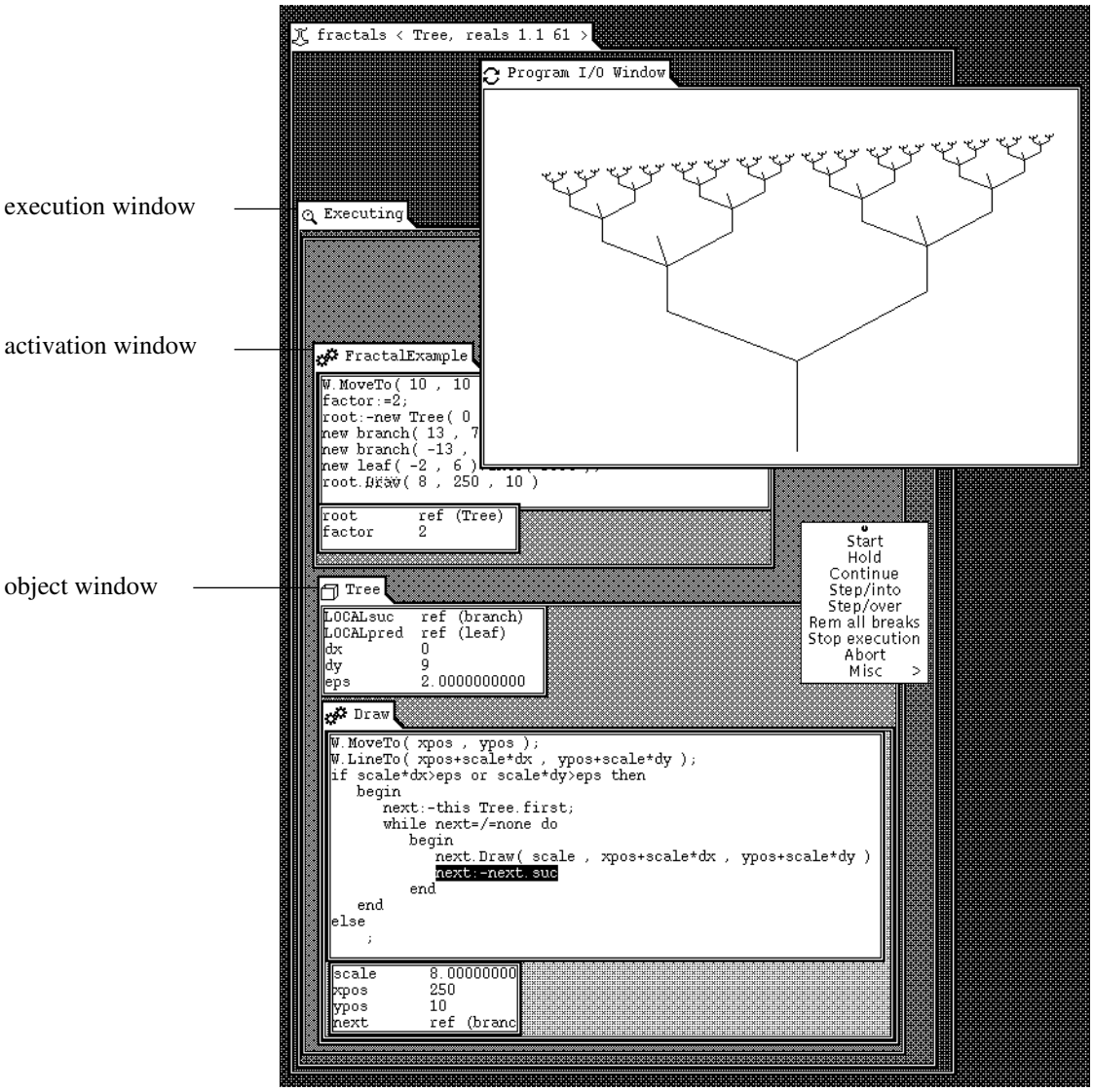
The source window has the following structure.

Source window	Represents the main program block
Documentation window	Contains text documenting the program
Declaration window	Contains variable declarations
Body window	Contains statements
Use import windows	Represents bindings to imported modules
Block window	Represents a class or a procedure
Documentation	Contains text documenting the class/proc
Declaration window	Contains parameters and variables
Body window	Contains statements
Block windows	
Block window ...	
...	

The execution window has the following structure:

Execution window	Changes the title from *Execution Observation* to *Executing* when the program is started.
Program activation	Instance of the main program block.
Active declarations	Shows values of variables
Active body	Shows program execution point or reactivation point.
Active block	Object or procedure activation
Active declarations	
Active body	
Active blocks	
Active block...	
...	

FIGURE 4 Presentation of the program execution



6.2 Entering, Exiting, and Saving a program

A program can be *entered* from the file browser . This will start an application (the Orm programming environment) through which it is possible to edit and execute the program.

Entering Enter the program by double clicking on the icon of the program in the directory . Or ***Program icon -> Open*** .

Exiting — Exit the program by ***program window -> Quit***. If you have edited the program a prompt box appears with “Save”, “Quit”, and “Cancel” buttons.

Saving — Save the program by ***program window -> Save***. It is wise to save the program rather often since the system crashes occasionally. Typically, you should save the program before executing it if you have made substantial changes. Each save creates a new revision of the program. A revision cannot be changed.

Note that closing the program window does not mean exiting the programming environment. When a program is in the entered state its title contains version information in addition to the program name.

6.3 Editing in Declarations and Body windows

Source code in declarations and body windows is represented as abstract syntax trees, *AST s*. Editing in these windows is basically structure-oriented. The program can contain *placeholders*, which represent incomplete parts of the program. The placeholders can be replaced by syntactically correct language constructs by choosing from a menu. This means that it is impossible to construct programs which are syntactically incorrect.

6.3.1 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 of a certain category. In Simula, there are the following categories:

identifier	e.g. Square, H2, moveT o
integer constant	e.g. 0, 1, 471 1
real constant	e.g. 3.14, 2.0, 0.015
text constant	e.g. “The Art of Programming”
character constant	e.g. 'c'

6.3.2 Selection

Language constructs are selected by clicking with the left mouse button. Repeated clicks at the same position extends the selection to the enclosing con-

struct. 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.

6.3.3 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 lexem*** 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*** .

6.3.4 Cut, copy, paste

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

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.

6.3.5 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 you actually have selected the list. In these situations you can find out by repeated selections which construct you have actually selected.

Another source of confusion is that some language constructs have extents which are very similar. In our Simula implementation, this is the case for actual parameters to procedures and classes. For implementation reasons, each parameter is enclosed in an extra construct which has a slightly larger extent than the parameter itself. E.g. in

```
p( a+b , c*d )
```

the actual parameter "a+b" is selected, whereas in

```
p( a+b , c*d )
```

the extra enclosing construct is selected. This is important to know, since if you want to edit the parameter (e.g. by expand, copy, or paste) only the parameter should be selected, whereas if you want to remove the whole parameter or add a new parameter before or after it, the extra enclosing construct should be selected.

In future versions of the system we hope to have removed these sources of confusion.

6.3.6 Textual Editing

The "expand" command is usually rather tedious to use for constructing and changing expressions. Therefore, textual editing is provided for expressions and for the assignment statements ("?:=?" and "?:-?"). To edit a construct as text, select the construct and choose ***Edit as text***. A promptbox appears in which you can change the construct textually. When you are ready, press the NEWLINE key on the keyboard, or click on the OK button. The text is then parsed, and if it syntactically legal, the corresponding AST will replace the current selection. If there was a syntax error in the text a new promptbox will appear. If you change your mind and don't want to change the construct, press the CANCEL button.

There is a convenient shortcut to textediting. Click on the middle button of the mouse instead of choosing the ***Edit as text*** menu command.

6.3.7 Semantic Editing

As a complement to structure-oriented editing and text editing it is also possible to do *semantic editing*. Semantic editing utilizes static semantic information such as scope rules, properties of declared identifiers, and context of the current selection to support high-level editing. Qualified references, as present in statically typed object-oriented languages, makes it possible to determine the legal set of messages to an object and provide quick access to these. This gives a powerful way of constructing message sends.

The ***Names*** menu in Orm gives a submenu of all names visible at the current selection. E.g. when in a normal expression in a procedure, the Names menu shows the declared identifiers in that procedure, in the surrounding class, in the superclasses of the surrounding class, etc. In a dot-expression the

Names menu instead shows the identifiers declared in the class (and superclasses) of the remotely accessed object.

If you choose a name in the Names menu, the current selection will be replaced by that name (if it is syntactically legal). Dot-expressions (message sends or remote access to variables) can be constructed by pulling right at the menu items for reference variables or procedures returning reference values. Furthermore, both the current context and the declared properties of the identifier are taken into account in constructing the resulting expression. Some examples:

- If a menu item denotes a procedure, array , or a class, the resulting expression will automatically contain placeholders for the appropriate number of parameters or indices.
- If a menu item denotes a class, a “new” is automatically inserted (if appropriate in the context)
- Context is taken into account in order to not insert parameters in inappropriate places, e.g. to the left of an assignment when returning a function value.
- Locally used but undeclared names also appear in the Names menu.

By using the semantic editing facility you very seldom have to type anything on the keyboard. Normally you can type a name just once and from then on retrieve it from the Names menu. The mechanism has proven especially valuable when using classes provided by someone else.

6.3.8 Incremental Semantic Checking

After each edit command, incremental semantic checking is performed. This means that you get immediate feedback on any static semantic errors in the source code. Examples of static semantic errors are incompatible types in expressions and undeclared variables. The erroneous construct is marked with a dotted rectangle. T o find out more about the error you can select the marked construct and choose

Explain

in the menu. Y ou can correct the error at any time you wish and the marking will then disappear . It is even possible to execute programs with static semantic errors in them, but execution will stop at the point of the error .

6.3.9 Other commands in Declaration and Body windows

Transform	Not implemented
To shelf	Not documented
From shelf	Not documented
Scrollbars on/off	Add or remove scrollbars on the window
Redisplay	Redisplay the text in the window .

6.3.10 Shortcuts

There are a number of shortcuts when editing in Declaration and Body windows.

Mouse shortcuts

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

Keyboard shortcuts

- 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 lexem
^M (or RETURN)	Expand after
^B	Expand before
^Z	Paste after
^Q	Paste before

- Function keys

L6	Copy
L8	Paste
L10	Cut

6.3.11 Shortcuts to the Expand menu

Instead of selecting syntactic constructs from the Expand menu with the mouse, the keyboard can be used. T ype 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 do <T AB>”, “wh d <TAB>”, and “wh<T AB>” 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.

In a similar manner identifiers and constants may be inserted by typing them followed by <ESC>. If an expression is selected and “3.14<ESC>” is entered, the real constant 3.14 will be inserted.

These shortcuts 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>.

6.4 Block-level Editing

6.4.1 Inserting and deleting blocks

New procedures and classes can be inserted by menu commands in the block windows (with *block* we mean a procedure, a class, or the main program). A procedure is created by the command

Block window/icon -> new procedure

and a class is created by the command

Block window/icon -> new class

A class or procedure is deleted by the command

Block window/icon -> delete .

6.4.2 Editing block titles

Currently , editing the title of a block window is a little awkward. The title can be edited either as text or as syntactic structure. In order to edit the title as text, select the following menu command

Block window -> edit title as text

and a text prompt appears. In order to use structure editing, choose

Block window -> edit title as structure

An edit window appears, containing the title of the block window . Editing is done in the same way as for Declarations and Body windows. After each edit step the title is updated. The edit window is removed by

Edit window -> Hide window

The syntax for class headings is

```
@ class @
```

where the name of the superclass should replace the first placeholder and the name of the class itself should replace the second placeholder . If the class does not have any superclass, simply leave the first placeholder as it is.

6.4.3 Copying blocks

Blocks can be copied and moved via a block clipboard. If the block clipboard is present it is shown as a window sibling to the Source window . The following commands are provided.

Block window -> MoveFromClipboard
: The block in the clipboard is removed from the clipboard and placed as a sonblock of Block.

Block window -> CopyFromClipboard
: The block in the clipboard is copied and the copy placed as a sonblock of Block.

Block window/icon -> MoveT oClipboard
: The block is removed from its current place and placed in the clipboard.

Block window/icon -> CopyT oClipboard
The block is copied and the copy is placed in the clipboard.

6.5 Executing programs

The execution observation window is created by

Program window -> Execute

6.5.1 Terminology

An *execution c* onsists of a collection of *objects* and *activations* . An object is an instance of a class. An activation is an instance of a procedure or of the main program. The execution may be *running* or *stopped* . A *point* is a position in the code where the execution is in a well-defined state. In the Simula implementation there is such a point before each statement, before each procedure call, and before each "new". A stopped execution has a *current execution point* , which is the point in the code where execution will be resumed. There are also points in the code which correspond to procedure calls or "new"s which have not yet returned. These points are called *reactivation points* .

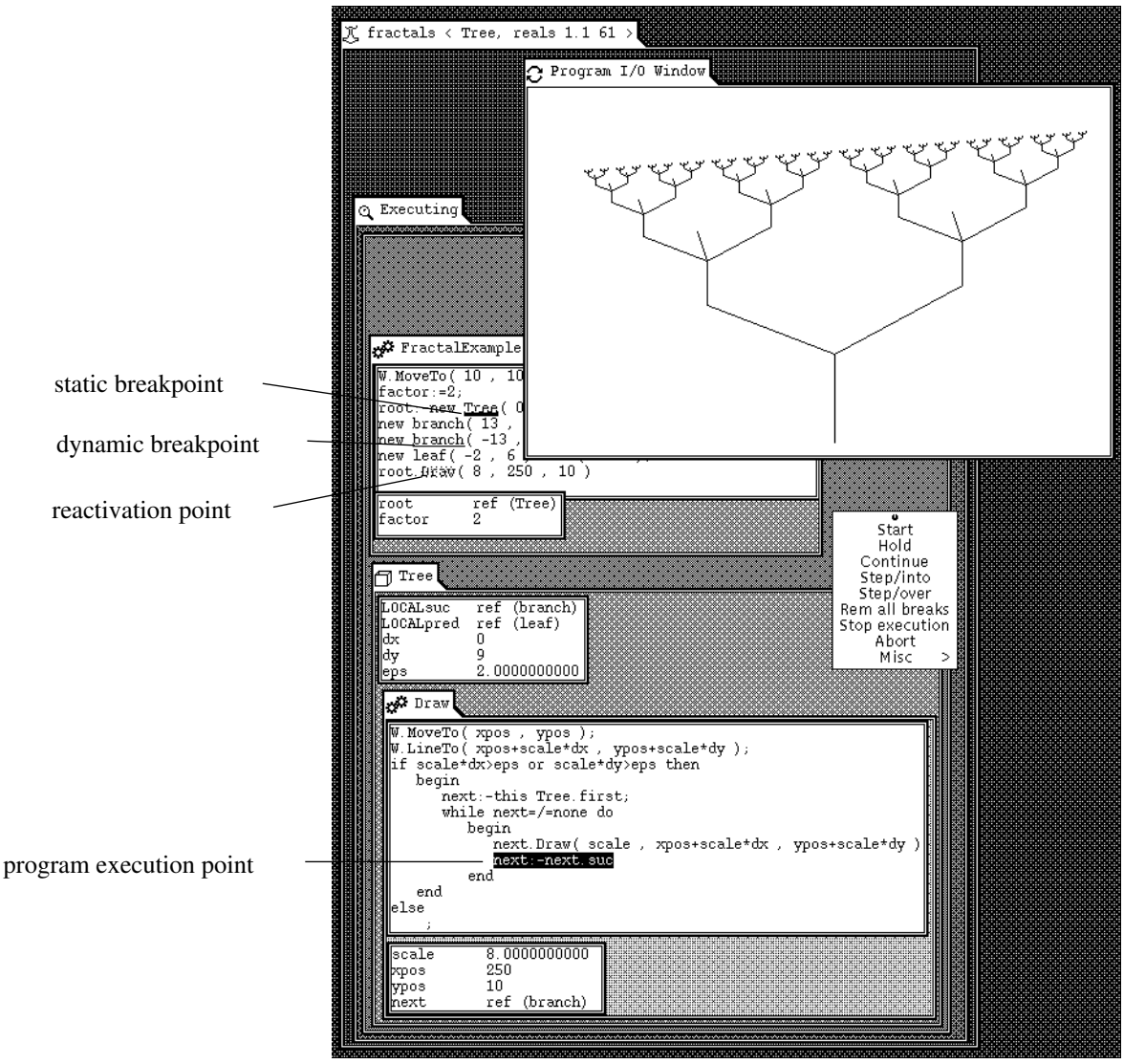
It is possible to set *breakpoints* at all points in the program. If a running execution encounters a breakpoint, the execution will be stopped. A *static breakpoint* applies to all instances (existing or future) of a procedure or a class. A *dynamic breakpoint* applies only to one specific existing object or activation.

6.5.2 Commands

The following menu commands are available in the execution window:

Start	Executes the program.
Hold	Stops the execution if it is running.
Continue	Resumes a stopped execution
Step/into	Executes to the next point, including points in called procedures and new objects.
Step/over	Executes to the next point in the current object or activation.
Rem all breaks	Removes all breakpoints.
Stop execution	Stops the execution and makes it finish prematurely . All objects and activations are removed.
Abort	Finishes the execution prematurely and also removes the whole execution window .
Misc -> Reset	Not documented
Misc -> Stop in -> Stop in ALL modules	Makes the Step commands apply to points also in imported modules.
Misc -> Stop in -> Stop in main only	

FIGURE 5 Program execution



Makes the Step commands apply to the main module only . This is the default case.

Misc -> InvCode

Invalidates all generated code. This will force regeneration and reloading of code at the next execution.

Menu commands in object windows and activation windows:

Delete window Deletes the window .

Show dynamic father

If the window of the dynamic father is open the window is flashed, otherwise it is opened.

Currently, it is not possible to continue running an execution after changes to the source code. Trying to do so will simply prematurely finish the existing execution and create a new execution starting from the beginning.

Breakpoints are set by selecting a language construct in an active body window and performing one of the menu commands

Set static breakpoint

Set dynamic breakpoint

Currently, breakpoints can only be set in existing object and activation windows. In future version of the system we hope to allow breakpoints to be set also in the source code windows.

Remove a breakpoint either explicitly by

Active Body window -> Remove breakpoint

or remove all breakpoints by

Execution observation window -> Rem all breaks

6.6 Documenting programs

There is one documentation window in each block window where it is possible to write free text. These documentation windows are intended for writing interface documentation. It is possible to generate a textual documentation file for a module. This generated file will contain the programming interface (block headings with parameters) and the text in the documentation windows. By using this feature the Orm environment can be employed as a (yet primitive) design tool.

6.7 Modules

There is no difference between programs and modules, i.e. any Orm program can also be used as a module. To import a module, the module import first has to be *declared* in the Imports window. The module import can then be *used* in a block. Currently, it is only possible to use imports in the outermost block (the Source window).

To declare a module, one has to give the file name of the module. These filenames have the ".mjol" suffix, e.g. "environment.mjol", "simset.mjol". The modules are found via an environment variable, ORMMODULES, which contains a list of directories. Module files are searched for first in the current directory, then in the directories listed in the ORMMODULES variable. In the standard setup, the ORMMODULES variable contains two libraries: ormsimlib and ormlab/lib. The ormsimlib library contains some general purpose modules:

environment.mjol	contains standard functions as sin, cos, etc.
simset.mjol	contains the classes for Simula's linked list facility .
uil.mjol	contains “userinterfacelayer” classes, i.e. windows etc.
text.mjol	contains a class for text manipulation.

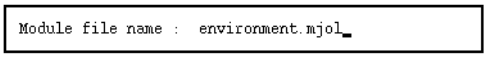
The ormlab/lib contains modules used in an introductory programming course based on Orm.

To delete an import one first has to delete all uses of it, then the import declaration can be deleted. If a module has been changed, other modules and programs importing that module have to be *rebound* to use the new version.

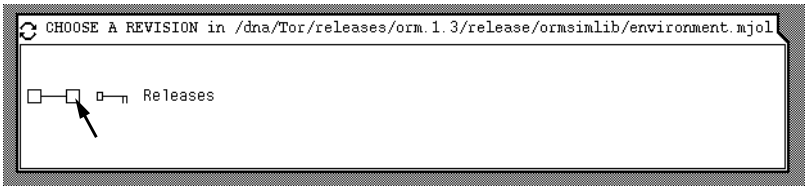
6.7.1 Quick guide to how to import a module

Suppose we have a program P and want to import the standard module environment.mjol. Proceed as follows:

1. Add a new import declaration to P by ***Imports window -> Declare new import*** .
2. A promptbox appears which prompts you for the filename of the module. Answer “environment.mjol” as below and press return.

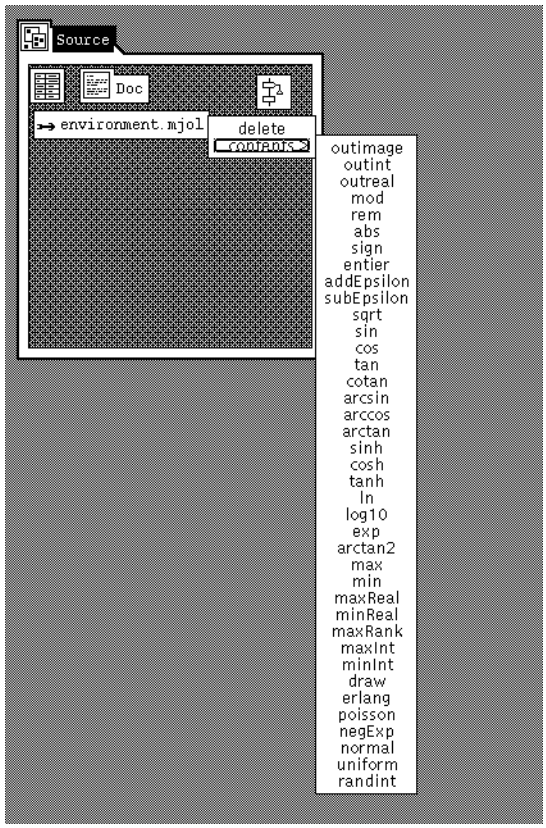


   The system will find the actual location (directory) of this module by consulting the ORMMODULES environment variable.
3. Now the system prompts you for which revision of this module to use. Usually you should choose the latest revision. Double-click on the rightmost small box in the prompt.



4. Now decide in which block in the program P to use this import. Currently, there is only one place to choose: the source block. In future versions of the Orm system it may be possible to add uses of imports also to classes and procedures.
   Select ***(Block window -> Use import -> environment.mjol)*** . A “Use import” icon now appears in the block window.

5. All the declarations in the imported module are now visible in the program P. Try the ***(Use window -> contents -> ...)*** menu to see which declarations the module contains.



6.7.2 Import declarations

The following screendump shows the pinned up menu of the Imports window.

- ***Declare new import*** adds a new import declaration. You will be prompted for the filename and the revision of the file. (As in section 6.7.1.)
- ***Change import binding -> ...*** allows the import declaration to be changed, e.g. to update the revision. By pulling right on this menu, without actually selecting, you will see all modules imported in this program. If any of the imports is not bound to the latest revision, this is indicated.
- ***Change import rule -> ...*** allows one to see and change the binding rules of the import declarations. The binding rules is used to determine which version of a module should be used. There are two types of binding rules, fix, and dynamic. These rules have effect when rebinding modules using

the “Rebind latest import” command. Fix-bound imports will not be affected by the “Rebind latest import” command, whereas dynamic-bound imports will be bound to the latest revision.

- ***Rebind “latest” imports*** . This command will rebind all dynamic-bound imports to the latest revision of their current alternatives.
- ***Show all implicit imports -> ...*** . This shows all imported modules including those implicitly imported (i.e. also those imported to imported modules). This command is useful in order to find out if a module happens to be imported in several revisions (which is usually a mistake).
- ***Delete import -> ...*** shows all imported modules. Selecting one of the modules will delete that module. However , deleting is only allowed if there are no uses of that import in the Source.

6.7.3 Changing and rebinding import declarations

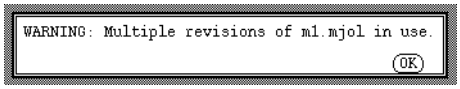
Suppose we have a program P which imports a module M. If M is changed we have to do the following steps to let the new version of M come into effect in P .

1. Save M ***(Program window -> Save)***
2. In P, rebind the import declaration of M. This is most simply done by choosing ***Import window -> Rebind “latest” imports*** . It can also be done by ***Import window -> Change import binding -> M*** .

If you are working with several modules it may occur that more than one revision of a module is implicitly imported. E.g. if we have the following modules

- Program P imports modules M1 and M2
- Module M2 imports module M1.

Suppose M1 is updated and that the import in P is rebound to the latest version of M1. However , M1 is also implicitly imported to P via M2. If the import in M2 is still bound to the older version we get a situation where two different revisions of M1 are imported into P . If this happens at rebinding, the following warning will appear:



This problem may also manifest itself as static semantic errors in P , since different revisions of the “same” class are regarded as type incompatible.

One can also find out if multiple revisions are in use by the command ***(Imports window -> Show all implicit imports -> ...)*** . The same module, but with different revisions, will then appear in more than one place in this menu.

In our example above, the problem can be remedied by the following steps:

3. Open the M2 module

4. Rebind the import to M1
5. Save the M2 module
6. Open the P program.
7. Rebind the import to M2.

There are some bugs in the system concerning rebinding of modules. If strange things happen, or if the system crashes during rebinding, you may be able to solve this problem by starting the Orm programming system directly from the UNIX level using the `-rebindmodules` option. This will prompt you for all imported modules and recompile the started program or module from scratch. (See section 10.1)

6.8 Other commands (printing, unparsing, etc.)

Other commands in the program window include the following.

as text -> Print program
: Unparses the program to Standard Simula text format and prints it.

as text -> Save program as text
: Unparses the program to Standard Simula text format and saves it on a file.

as text -> Save interface
: Generates a documentation text file with the class and procedure interfaces and the text from the documentation windows.

as text -> Save Documentation only
: Generates a documentation text file including only the text from the documentation windows.

other -> ...
: Not documented

6.9 Grammar Window

Each Orm program is developed according to an Orm grammar . The *grammar window* in a program shows which grammar is used and makes it possible to change this connection. Changing the grammar connection is not of interest to the normal user and is not documented further here.

6.10 Purge

Each Orm program is subject to revision control. Since storing all revisions consumes rather much disk space it is usually of interest to have automatic purging of revisions. Each Orm program has a purge attribute to control this. The purge attribute can either be set to keep a specific number of the latest revisions, or to keep all revisions. A new Orm program, created via the Orm file browser [5], automatically keeps the two latest revisions of the

5. If the program has been created directly from the Unix level (i.e. via the "browser" command in a Unix shell) it keeps *all* revisions by default.

program. Programs distributed in the release (e.g. modules and example programs) may have other purge attribute values.

In order to display or set the value of the purge attribute, the program has to be opened with revisionhandling. This is done by the menu command ***program icon -> open with revisionhandling*** in the Orm file browser. Instead of opening the (latest revision) of the program, a window appears which gives access to all the program revisions. Now you can display the purge attribute value by:

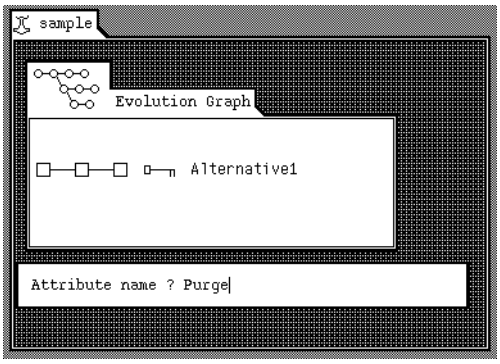
Evolution Graph Window -> AttachedInfo -> Attributes -> Object Attributes -> Show

This will display all attributes set for the program. First a timestamp attribute is displayed. Click on OK in the notifier which appears, and the purge attribute will be displayed.

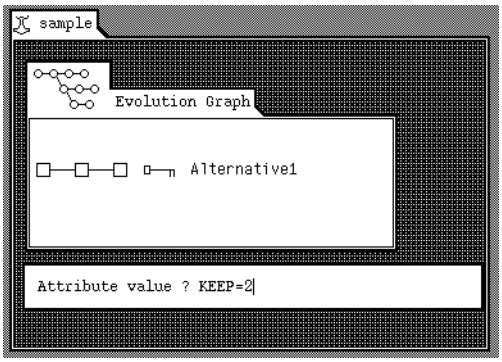
The purge attribute can be set by:

Evolution Graph Window -> AttachedInfo -> Attributes -> Object Attributes -> Set

A promptbox for the attribute name appears. Answer “Purge” as below (case is significant) and press return.:



As a result, a prompt for the attribute value appears. Answer “KEEP=2” as below if you want to keep the 2 latest versions. Answer “KEEP=ALL” if there should be no automatic purging.



The automatic purging applies only when quitting a program opened in the normal way . It is also possible to do a manual purge by

Evolution Graph W indow -> Misc -> Purge

The boxes in the evolution graph window represent the individual revisions. The leftmost box is the “source” revision which is very small and is not subject to purge. Purging will thus leave n+1 boxes if the purge value is n.

Finally , leave the revision handling by ***Quit*** in the revision handling window (the background window). Then you can open the program in the normal way again (double-click on the program icon in the file browser window).

6.11 Error recovery

6.11.1 Restoring a crashed program

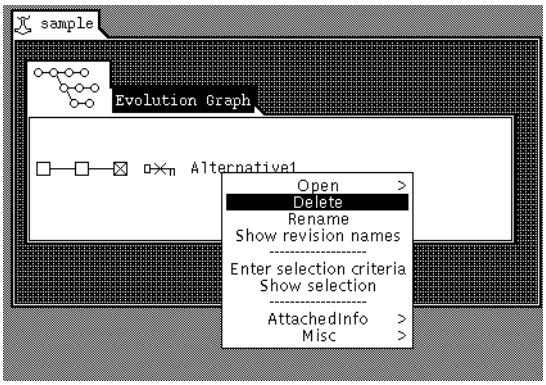
If the Orm system or part of it crashes while you have an opened program, this is normally no problem. Y ou will have lost only the changes you made since the last Save command. However , if the system crashes *during* opening or saving the program, the stored program may be left in an inconsistent state. 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, one of the following things may happen when you open it:

- The program icon simply disappears when you try to open it. No program window appears. In this case you have to kill the whole Orm system with CTRL-C and start `orm` again to see the icon.
- The program icon reappears a short while after you try to open it. No program window appears.

In these cases, the latest revision of the program is inconsistent and has to be removed. T o do this, proceed as follows.

1. Save a backup copy of the program (in case this cure does not work) by ***Program icon -> copy -> "backup.mjol"*** .
   (If you are out of disk quota this might not be possible).
2. Open the program with revision handling by the menu command ***Program icon -> open with revision handling*** .
   A revision handling window for the program appears. The revision handling window contains an "Evolution Graph" window showing small boxes which represent all the existing revisions of the program.
3. 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 program crashed the previous time it was run, the key is probably crossed over, indicating that the key has not been returned properly. To remedy this, do the menu command ***Evolution Graph W indow -> Misc -> Restore all Keys*** . The cross over the key now disappears.
4. 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 W indow -> Delete*** .



   A promptbox asks if you want to delete the revision. Click on the Yes button.
5. Leave the revision handling by ***Quit*** in the revision handling window (the background window).
6. Check if the cure worked by opening the program normally again (by double-clicking on the program icon in the file browser). If the program 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.

6.11.2 Error message when opening a program.

In some situations, an error message is displayed when you try to open a program. Further , a revision handling window containing an "Evolution Graph" window appears instead of the normal program window .

This occurs in the following situations:

- The file is missing for the latest revision of the program. This could happen if you have moved Orm program files with the Unix `mv` command and only moved the "basefile". In this case, try to move back those files (see the chapter on Unix level for details about the files).
- The `/tmp` directory is full
- You do not have write access in `/tmp`

6.11.3 Other problems

- The whole window system crashes. This may happen if you have too many Orm programs and grammars open and the window system runs out of swap space.
- The file browser says "All applications must be terminated first" when you try to exit the Orm system. This happens when there are open programs, grammars or other documents which have not been properly quitted. Use CTRL-C if you cannot find the open documents.
- Quitting a program or grammar leaves a big white space on the screen. Sorry , this is a bug which happens sometimes. If the white space is inside a directory window you can only get rid of it by exiting Orm. If the white space is directly on the screen background, i.e. left over from Orm itself (after exiting Orm) or after a stand-alone session with `gred` or `browser` you may try the Open W indows `pam` command.
- The Program I/O W indow "disappears".
  The I/O window is on the same level as the program window (i.e. inside the same directory window). T ry closing the program window . Probably you will find the I/O window behind it.

  Due to a bug, the I/O window does not always become visible when you start the program execution. T o fix this, you may add a call to "Close" as follows:

  ```
  W :- new Window(...);
  W.Close;
  W.Open;
  ```

  The call to Close will make the subsequent call to Open put the window in front.
- A body or declaration window seems "dead".
  There may be an unanswered promptbox connected to the window . If you cannot see any promptbox, try closing the outermost Orm window (probably a directory window) to an icon. The promptbox may be behind it.

7.0 The Orm Grammar Editor

7.1 Window structure

The window hierarchy in the grammar editor is the following:

Directory window	The directory containing grammars
Grammar window	Contains the parts of a 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 and concrete grammar aspects.

7.2 Entering, Exiting, and Saving a grammar

A grammar is normally entered from the file browser. By double-clicking on a grammar icon, the grammar editor is started.

Entering	Double-click on a grammar icon. Alternatively, choose the "open" alternative from the popup menu of the icon.
Exiting	Exit the grammar editor by selecting "Quit" from the menu in the grammar window. If the grammar has been edited a promptbox will appear with "Save", "Quit", and "Cancel" buttons.
Saving	Save the grammar by selecting "Save" from the menu in the grammar window. We recommend saving a grammar now and then since the system occasionally crashes. Typically, you should save the grammar after a substantial modification before trying it out in the target window.

7.3 Creating a new grammar

A grammar is created by selecting "new grammar" in the directory window. The system prompts for the name of the new grammar. An icon for the new grammar appears. The grammar is opened by double-clicking on the icon. An empty grammar window appears. Add an abstract, a concrete, and a target window as described in section 7.5.1. Don't forget to set the startproduction in the target window before editing in the target window (see section 7.4).

7.4 Editing in the target and grammar aspect windows

All grammar aspects and the target program are represented as abstract syntax trees. They are all edited by means of the structure-oriented editor. An editor is started when a grammar aspect window is opened.

The editor is basically the same as in the Orm program browser. Please refer to section 6.3 for a description of its functionality. However, the following features are not yet available in the grammar editor:

1. Textual editing (as described in section 6.3.6) is not supported.
2. Semantic editing (section 6.3.7) is not supported.
3. Incremental semantic checking (section 6.3.8) is not supported.

Furthermore, a “Misc” submenu has been added with functionality specific for grammar editing. It has the following entries:

Scrollbars on/off	Turns on or off the scrollbars of the window .
Redisplay	Redisplays the contents of the window
Reunparse	Reunparses the contents of the window according to the concrete grammar and displays it. Used in the target window after the concrete grammar has been modified.
Rebuild menus	Recreates the contents of the expand menu. Used in the target window after the abstract or concrete grammar has been modified.
Save as text	Saves a grammar aspect or target structure on a text file. You will be prompted for the filename.
Set title	Allows the title of 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.

7.5 Editing in the grammar window

7.5.1 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 Semantic	Inserts a new semantic grammar aspect
Add Codegen	Inserts a new code generation grammar aspect
Add Lexical	Not implemented
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. T o 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 on the title of the window .

7.5.2 Other commands

The following entries in the grammar window menu are not documented. We recommend not using them.

Change meta grammars	Not documented.
Revision	Not documented.
Miscellaneous	Not documented.

7.6 The grammar formalisms

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

7.6.1 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 ::! IdUse ! Add ! Sub ! Mult ! Div Alternation
(5) Plus::= Expression & Expression             Construction
(6) IdUse ::= 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 IdUse, Add, Sub, Mult, and Div , and that a Plus is a construction of two Expressions. (6) Finally , an IdUse (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 [6]:

```
(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::=_
```

(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 .

7.6.2 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 an 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 <<<
```

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

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

(4) IdUse::=                                 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 IdUse is a lexeme production. It has two parts, "before" and "after". The before part specifies the concrete syntax presented before the lexem contents. The lexem 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 have knowledge of the structure of concrete grammars. It can be expressed in an abstract grammar for concrete grammars [7]:

```
(1) ConcreteGrammar::* Production
(2) Production ::= Id & Definition
(3) Id ::= LEXEME
(4) Definition ::! Construction ! List ! Lexeme
```

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

```
 (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 T emplateList describing the concrete syntax presented before the descendants. The concrete syntax of the descendants is described in the SonList production. (6) A T emplateList consists of a number of templates containing keywords and formatting information. (7) A T emplate 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. (9,10,1 1) 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 T emplateList containing 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 lexem and its contents should be 1 for the first son, 2 for the second, etc. (15) A List definition has three parts. One T emplateList that specifies the concrete syntax to be presented before the first element of the list, one T emplateList that specifies the concrete syntax separating list elements, and one T emplateList for the concrete syntax after the last list element. (16) A lexeme definition contains a T emplateList containing the concrete syntax before the lexeme contents, and one T emplateList for the concrete syntax after it.

The “bread and butter” of the concrete grammars are the T ext, Indent, EndIndent, and NewLine templates. A T ext template will cause the lexeme contents of the T ext 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.

7.6.3 Semantic and code generation grammars

The semantic and code generation grammars can also be edited using Gred. They are expressed in their own formalisms specified in metagrammars, i.e.

an abstract and concrete grammar for the semantic grammar and an abstract and concrete grammar for the code generation grammar .

The formalisms for these grammars are *not* documented here.

7.6.4 Editing metagrammars

All grammars in the Orm programming environment and grammar editor are represented as abstract syntax trees. They are edited by means of the structure-oriented editor in 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 semantic and code generation 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 Gred. 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 Gred interprets while editing. Changing the abstract aspects of the metagrammars will cause unpredictable results.

The metagrammars are named abs.gram, con.gram, sem.gram, and code.gram. They are located in the "grammars" directory . The grammars may be inspected using Gred, but we strongly recommend you to *not* edit them.

7.7 Changing a grammar used by the Orm programming environment

All grammars, including the metagrammars and the Simula grammar , have been developed in the grammar editor . It is possible to modify a grammar , e.g. the Simula grammar , and start the programming environment with the new grammar . However , the semantic and code generation grammars have not been documented in this user 's guide and the commands for rebinding a program to a new revision of the grammar is not documented.

This documentation only describes experimenting with the abstract and concrete aspect of a language in the T ARGET window in the grammar editor .

8.0 The Orm Version Handling System

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

8.1 Introduction

All programs, modules and grammars within the Orm system are uniformly version controlled using the same basic mechanisms. The version handling system supports the user in organizing versions of programs, modules and grammars and maintains relations between those stored objects. T wo users may simultaneously work on the same revisions of the same program or grammar .

The operations for entering, exiting, and saving programs, modules and grammars (introduced in the sections: The Orm Program Browser and The Orm Grammar Editor) uses the version handling system for storing the information. These operations uses 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.

8.2 Terminology

A program, module or grammar 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. T ypical examples are a program that depend on a grammar and a program that import a module. 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.

8.3 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 is not yet 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.

8.3.1 Lookup of PBOId

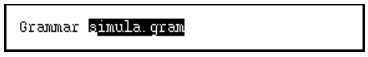
At binding time the PBOId is looked up in the file system in the following order:

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

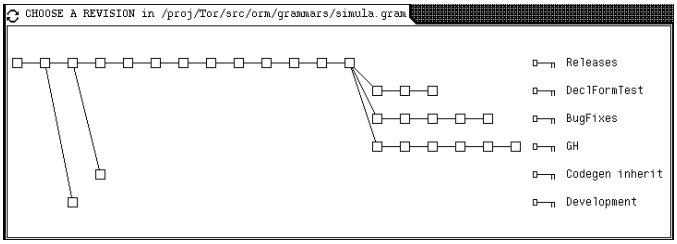
If no PBO is found an explicit binding is initiated.

8.3.2 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

8.3.3 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 is not 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

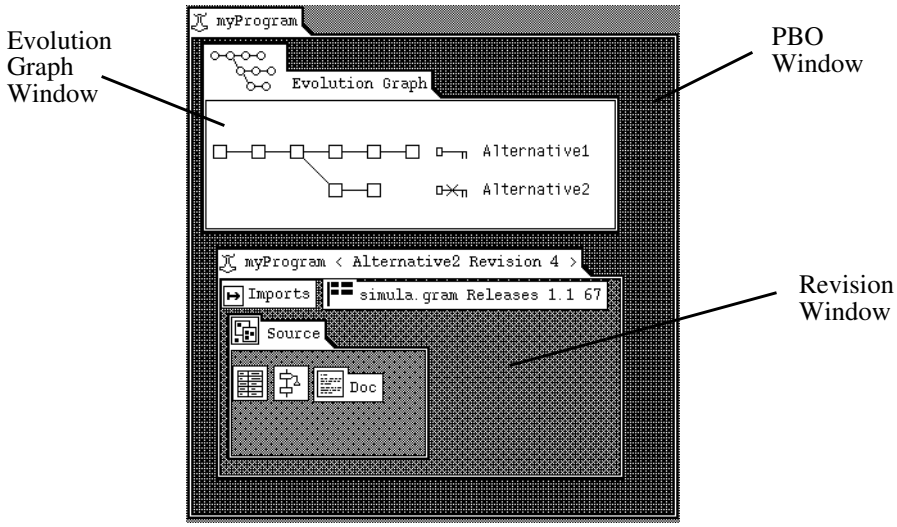
   An explicit binding is initiated. The binding rule is overruled.

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.

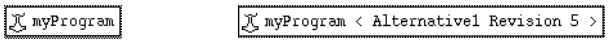
8.4 Window Structure

The window corresponding to a program, module or a grammar represents the *revision window* and is correspondingly labelled with the PBO-name and the RevisionId. A *PBO-window* can be inserted between the revision window and its father window and it will represent the PBO in the window hierarchy . Within the PBO-window there are functionality available for viewing and affecting the revisions in the evolution graph. The PBO-window is labelled with the PBOId. The use and functionality of the PBO-window is independent of the initial tool, i.e. independent of whether the ORM Program Browser or the ORM Grammar Editor is used. W ithin the PBO-window the *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.



If the PBO-window is omitted in the window hierarchy the revision window will be connected to the window enclosing the PBO-window . The icon of the

revision window and the PBO window are the same, but the title of the revision window icon is extended with the revision name.



PBO W indow icon Revision W indow icon

8.5 PBO Window

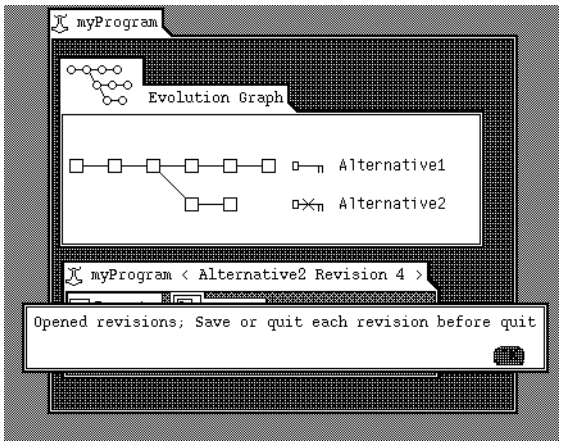
In this section the functionality of the PBO window is described.

8.5.1 Commands

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

Restriction:

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



8.5.2 Undocumented features

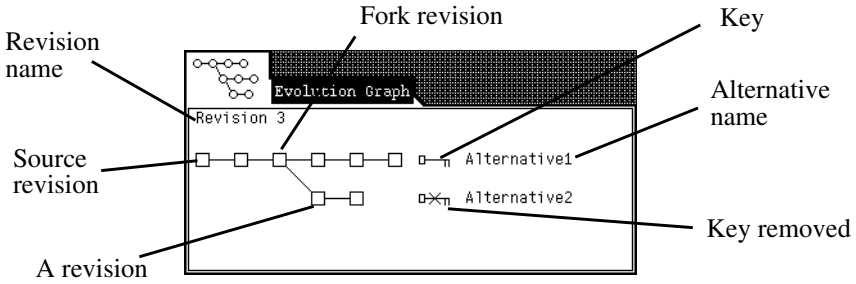
-> Garbage Collect

-> Memory Statistics

-> Memory Statistics without GC

8.6 Evolution Graph Window

The evolution graph window displays all revisions of the PBO.



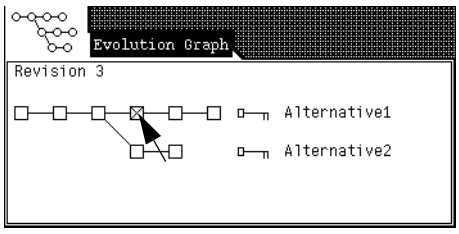
The first revision is called the *source revision* and is the origin empty revision. A number of consecutive revisions are called an alternative. A revision may belong to several alternatives. A branch starts in a *fork revision*. For every alternative there is a key to enable synchronization between 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.

8.6.1 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 affects the selected objects in the graph. If more than one revision is selected the operation will be applied to each revision in turn.

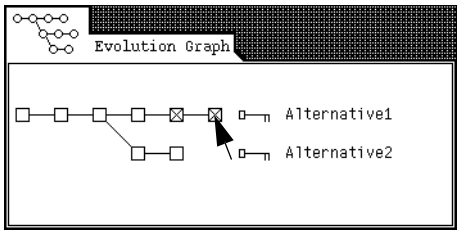
Select a revision — Click on the revision of your choice. Any previous selections will be deselected.



Click on the revision

Toggle revision

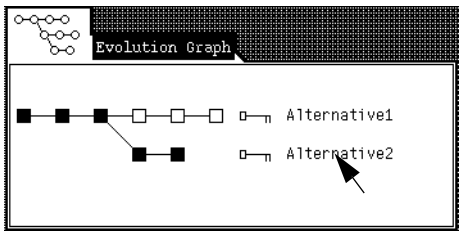
Click on a revision while pressing the shift key on the keyboard. The state of the revision toggles between selected and not selected. The operation will not affect previous revision selections. The toggle operation can be used for selecting more than one revision.



Click + Shift

Select alternative

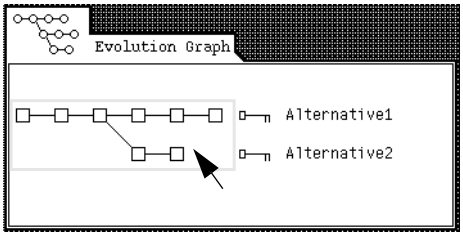
Click on the alternative name. It is only possible to select one alternative at a time.



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.



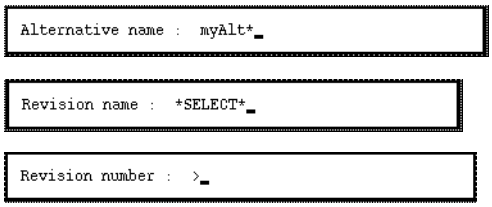
Click inside the marked area

8.6.2 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.

8.6.3 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 rebind 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.

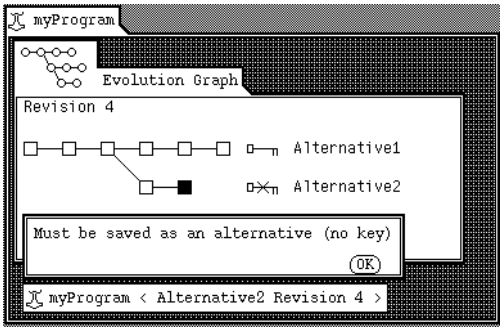
Shortcut: A double click on a revision is a shortcut for choosing a revision and then performing the ***-> Open -> Dynamic Rebinding*** command.

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 a revision with a key is quitted 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.



8.6.4 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.

8.6.5 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.

8.6.6 Attribute support

An attribute is an attribute name with an associated attribute content. In the current implementation both the name and the content 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.

Restriction:

- See *Predefined Attributes* (section 8.10) 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.

8.6.7 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 8.10) for more information.

8.6.8 Undocumented features

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

-> Misc -> Compact

-> Misc -> Redisplay

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

8.7 Revision window

The revision window contains the functionality to save and quit a revision of the PBO. The type of the revision window is application dependent, e.g. The Orm Program Browser or The Orm Grammar Editor .

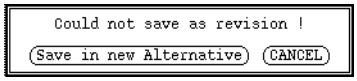
8.7.1 Save

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 of the inserted revision is owned by the current revision.

-> Save

Restriction:

- 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 saved 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.

8.7.2 Quit

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

-> Quit

If the user has received the key it is returned.

Note:

If used in “latest” mode, i.e. without PBO-window , the program will terminate and the Purge command is performed.

8.7.3 Revision commands

The revision command will be found in different menus depending on the application used. See the application characteristics of the revision window for further details of where to find the menus.

-> ... -> 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.

-> ... -> 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.

-> ... -> Insert Revision W indow

The PBO window and evolution graph window are shown 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.

8.8 Orm Program Browser Characteristics

This section describes some characteristics of the Program Browser from the version handling point of view .

8.8.1 Save

If the Program Browser has been started with the "-latest" option [8] an optimistic approach for inserting new revisions is used. This means that a revision may be inserted in the alternative from which it originates from without holding the key . Thus, if several users have opened the same program in the "-latest" mode, the first user who tries to save the program will be able to save it in the current alternative. All the other users will then have to create new alternatives when saving. The system will prompt them for the alternative names.

8.8.2 Quit

If the Program Browser has been started with the -latest option an automatic purge is performed.

8.8.3 Revision commands

Location of the revision commands within the revision window . See the section concerning the general revision window for definitions.

-> Other -> Revision commands -> Save as revision

-> Other -> Revision commands -> Save as alternative

-> Other -> Revision commands -> Insert Revision W indow

8.8.4 Undocumented features

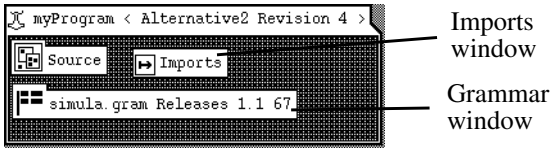
-> Other -> <os-cmd> revisionfile

-> Other -> Relations -> .....

8. This is the case when opening a program in the normal way in the Orm file browser (double-click).

8.8.5 Binding of Relations

In the program browser there are two different types of relations. The program grammar relation and the relations to the imported modules.



- Imports window

  Used for importing modules into a program. A simplified version is used for the binding rule. The rule is always set to "* * >", i.e. the latest created revision of any alternative.

-> Declare new import

Imports a new module in the program by letting the user perform an explicit binding for the new module.

-> Change import binding -> ..... (All imported modules)

Explicit rebinding of the selected import.

-> Change import rule -> ..... (All imported modules)

Used for selecting dynamic or fixed binding of the selected imports.

-> Rebind "latest" imports

All dynamically bound imports are rebound to the latest created revision. This command is used if the user knows that a new revision of a module has been created during the current session.

- Grammar window

  Points out the language grammar used when developing the program, e.g. the Simula grammar or the Beta grammar .

-> 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.

-> Set Binding Rule -> Explicit

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

-> Set Binding Rule -> Fixed

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

-> Show Binding Rule

Shows the binding rule of the relation.

-> Show Grammar Path

Shows the full file name of the current grammar revision.

8.9 Orm Grammar Editor Characteristics

This section describes some characteristics of the Grammar Editor from the version handling point of view .

8.9.1 Save

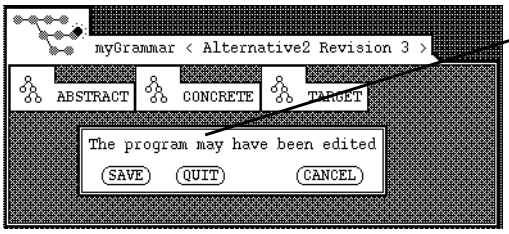
This works the same way as for programs. See section 8.8.1

8.9.2 Quit

If the Grammar Editor has been started with the -latest option an automatic purge is performed.

Restriction:

The grammar editor do not check if the grammars have been edited. A warning is always shown in the confirmation prompt.



8.9.3 Revision commands

Location of the revision commands within the revision window . See the section concerning the general revision window for definitions.

-> Revision -> Save as revision

-> Revision -> Save as alternative

-> Revision -> Insert Revision W indow

8.9.4 Undocumented features

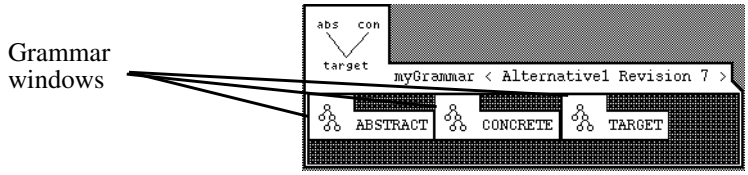
-> Miscellaneous -> Create relation

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

-> Miscellaneous -> Relation info

8.9.5 Binding of Relations

One relations is connected to each grammar window in The Orm Grammar Editor . The relation is bound to the meta grammar .



- Revision window

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

When adding a new grammar 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 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.

8.10 Predefined attributes

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

8.10.1 PBO attributes

Currently there are only two predefined PBO attributes.

- Purge

  The attribute defines how many revisions which 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 ’.

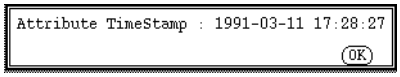
Examples:

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 .



8.10.2 Revision attributes

There are currently no predefined revision attributes.

8.11 Trouble shooting

8.11.1 Can not open a revision

- Read/W rite privileges
- /tmp
- Inconsistent revision
- Inconsistent evolution graph

8.11.2 Can not save a revision

- Read/W rite privileges
- Disk quota
- Inconsistent evolution graph
- Lost key

8.11.3 Lost revisions

- Purge attribute

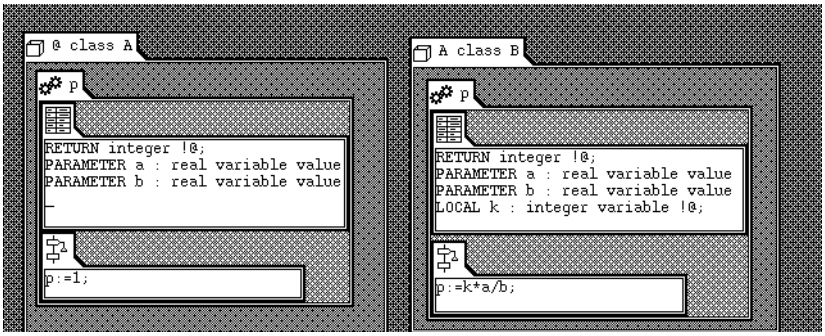
9.0 Orm Simula

This chapter reports on how Orm Simula differs from Standard Simula [9]. Orm Simula includes the most important constructs: classes, virtual procedures, statements, and expressions. Some notable constructs *not* included are coroutine sequencing (call, detach, and resume), teletype input-output, file handling, name parameters, and text. There are also slight differences in the syntax for the declarations. The module concept in Orm is also different from the “external” construct in Standard Simula.

9.1 Virtual Procedures

In Orm Simula, all procedures which appear inside classes are automatically virtual. The outermost procedure with a particular name is regarded as both a virtual procedure specification and a virtual procedure implementation. All procedures with the same name which appear in inner classes (subclasses) are regarded as virtual procedure implementations and must have exactly the same signature (the same parameters and the same result type)

Example : The Orm Simula fragment



corresponds to the following fragment in Standard Simula:

```
class A;
   virtual: procedure p is
      integer procedure p(a,b); real a,b;;
begin
   integer procedure p(a,b); real a,b;
   begin
      p := 1;
   end;
end -- A --;

A class B;
begin
   integer procedure p(a,b); real a,b;
   begin
      integer k;
      p := k*a/b;
   end;
end -- B --;
```

9. Data processing - Programming Languages - SIMULA, Swedish standard SS 63 61 14. SIS, Stockholm, Sweden, June 1987.

The example above also shows how the syntax for local variables, parameters, and return values differ in Orm and Standard Simula.

9.2 Modules

The module concept in Orm makes it possible to define fragments of programs as separately stored (and compiled) units. A module is in many ways similar to a statically allocated object. It may contain classes, procedures, local variables, and code. Importing a module is semantically similar to inspecting an object: all the declarations in the module become visible in the block which contains the import.

There is no difference in Orm between programs and modules. Any module can be both executed and imported. When executing a program, the code of its imported modules will be executed as well. Code in a module can thus serve several purposes:

- initialization of variables local to the module
- temporary test code when developing the module
- "main program code" when the module plays the role of a program

If a module is imported via several routes, it will still only result in one object during execution. E.g., suppose a program P imports modules A and B, which both import module C. When running P , there will only be one instance of the C module.

The module concept in Orm differs from the "external" construct in Standard Simula which allows a class or a procedure to be compiled separately . In some ways the Orm module is a more flexible construct. It makes it possible to group several classes together in one module. E.g., in Orm, the Linkage, Head, and Link classes are defined together in one module (simset.mjol). In Standard Simula these classes are nested inside another class (class SIMSET) which is sometimes rather impractical.

9.3 Details

This section gives a more detailed account of the differences between Orm and Standard Simula, following the structure of the Simula Standard.

9.3.1 Not implemented features

- algol subblocks (4.10) — Use a parameterless procedure or class instead.
- prefixed blocks (4.10.1) — Use a parameterless class instead.
- coroutine sequencing (7) — Not yet implemented.
- class Simulation (12) — Uses coroutine sequencing, not yet implemented.
- inspect statement 4.8) — Use is/in and qua instead.
- full for statement (4.4) — Only the simple variant with one step-until part is implemented.

- goto statement, switches and labels (4.5, 3.9 and 5.3) — Not implemented.
- parameters to procedures and classes (4.6 and 4.7) — Only default parameter passing mode for value types and reference variables is implemented. Arrays and procedures can be passed if they are hidden inside an object.
- name parameters (4.6.4) — Not implemented but present in the syntax. At least var semantics for simple variables will be implemented.
- long reals, short integers — Use reals or integers.
- I/O and files — primitive graphic I/O is available in the module uil.mjol.
- text — instead of a built-in text type, a class text with similar semantics is available in module text.mjol.
- overloading of standard procedures (9) — E. g. the max function is defined only for real type values.
- attribute protection (5.5.4) — The hidden-protected mechanism is not implemented.
- constant declarations (5.8) — Not yet implemented.

9.3.2 Not enforced restrictions

- block level for prefix classes (5.5.1) — A class can be used as a prefix at any block level. This works well across module boundaries but not in certain cases where the prefix class and the subclass are defined in the same modules.
- transplantation (2.4.1 and 3.1.5) — No runtime check is performed to prevent the transplantation effects mentioned in 2.4.1. No restriction on remote access is made as in 3.1.5.

9.3.3 Known bugs and errors

A few errors will cause an infinite loop in the semantic checker . Unfortunately the only escape is to kill the program (e.g. hit ctrl-C).

- An attempt to define a class as prefix to itself (directly or indirectly) will result in an infinite loop in the semantic analysis.
- A parameter of reference type to a class can not have a qualification which is equal to or inner to the class itself (causes infinite loop).

Some typechecking mainly on parameters is missing.

- The runtime check mentioned in case 2 of 4.1.4 is not performed.Thus it is possible to pass an object of a superclass to the qualification of a reference variable.
- A parameter of undefined ref-type gives no semantic error -marking.
- Passing an integer as actual parameter when the formal parameter is of ref-type causes a crash during execution.

During code generation many things are performed in an order depending on the order the names of variables, procedures etc. are entered. In some

cases this causes problem, which makes it necessary rearrange the program somehow . A name can be reentered by selecting it, ***Cut*** it and ***Paste*** it at the same place.

- The allocation structure during execution for a procedure or a class is decided by the order the parameters and local variables are entered. As a consequence the parameters to a virtual procedure implementation must be entered in the same order as in the corresponding specification. (***Cut*** / ***Paste*** the names of all parameters and local variables in order .)
- If the name of a formal parameter is changed, it might cause a change in the allocation structure for the class or procedure, but the corresponding instantiations of the (sub-)class or procedure are not always recompiled. (Keep the parameters ordered or ***Cut*** / ***Paste*** the type of the parameter .)
- If the name of a subclass is entered before the name of its prefix, the code for the prefix class will be loaded twice during execution. This results in inconsistencies during runtime. (***Cut*** / ***Paste*** the name of the subclass.)

10.0 Unix level

This chapter explains how Orm works on the Unix level. This information can be useful if you want to change the system configuration, if you want to run one of the components "stand-alone", or if you get into some trouble. For normal usage you don' t need to know this.

The functionality of Orm is composed of the following binary executables

- *finder* , manages the file system. Starts applications, e.g. browser and gred.
- *browser* , manages editing, compilation, and observation of programs
- *rts* , the runtime system which actually executes a program. The rts is started by the browser when the user performs the -> execute menu command.
- *gred* , the grammar editor .

10.1 Running browser and gred "stand-alone"

Normally , the browser and gred applications are started by the file browser when the user double-clicks on a program or grammar icon. It is also possible to start these applications directly from a unix shell. First do the command

```
> source orm
```

This will not start the Orm system, it merely sets the path and other environment variables. After this you can start the applications by e.g.

```
> browser -m=4 -latest aProgram.mjol
> gred -m=3 -latest aGrammar.gram
```

10.1.1 Browser

The synopsis for browser is

```
> browser [-m=#] [-g] [-latest] [-reevaluate]
            [-rebindmodules] [filename]
```

`-m=#`	start application with a heap of # Mbyte.
`-g`	print a message for each garbage collection.
`-latest`	start application on the latest version of the file. Do not display the version handling window .
`-reevaluate`	recompile the program from scratch.
`-rebindmodules`	interactive rebinding of all imported modules. W ill also recompile the program from scratch.
`filename`	If the file does not exist, a new program is created. If no filename is given, the system will make one up. The filename should have the extension *.mjol* .

10.1.2 Gred

The synopsis for gred is

```
> gred [-m=#] [-g] [-latest] [filename]
```

`-m=#`	start application with a heap of # Mbyte.
`-g`	print a message for each garbage collection.
`-latest`	start application on the latest version of the file. Do not display the version handling window.
`filename`	If the file does not exist, a new grammar is created. If no filename is given, the system will make one up. The filename should have the extension *.gram.*

10.2 Program and grammar files

Programs and grammars in Orm 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:

prog.mjol	*(basefile)*
.prog.mjol_1	*(revisionfile)*
.prog.mjol_5	*(revisionfile)*

The revisionfiles contain the "content" 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:

`> mvPBO prog.mjol x.mjol`	to change the name of a program
`> mvPBO prog.mjol dir`	to move a program to another directory
`> cpPBO prog.mjol x.mjol`	to copy a program
`> cpPBO prog.mjol dir`	to copy a program to another directory
`> rmPBO prog.mjol`	to remove a program

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

10.3 Environment variables

The Orm system makes use of the following environment variables (set in the `orm` script):

`setenv MJOLNERHOME dir`
: used for finding grammars and program templates

`setenv IMAGEPATH dir[:dir]*`
: the directories which contain icon images. The directory names should be absolute pathnames in the environment of the OpenW indows server .

`setenv FINDER_CONFIG file`
: the configuration file for the file browser

`setenv ORMMODULES dir[:dir]*:`
: the directories which contain program modules. Note, because of a bug, this path must end with a colon.

10.4 The .ormlog file

When running the browser via the Orm system, some information including possible error messages will be logged in the file *~/.ormlog* . If you run the browser directly from a UNIX shell, this information will appear in the shell window .

10.5 Files in the release

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

```
bin/
   orm           the system startup script
   cpPBO         script for copying program and grammar files
   mvPBO         script for moving program and grammar files
   rmPBO         script for removing program and grammar files
   generic       a script for handling many machine architectures
   DoBrowser     redirects IO to log-file when running browser
   ...           soft links to generic script to handle many
                    architectures
   sun4/         sun 4 binary executables
      finder        file browser
      browser       program development system
      rts           runtime system
      gred          grammar editor
      wexec         used by finder
      simulaunparser used by browser for unparsing to textfiles
lib/
   orm-config    configuration file for the finder
grammars/
   abs.gram      metagrammar for abstract grammars
   con.gram      metagrammar for concrete grammars
   code.gram     metagrammar for code generation grammar
   sem.gram      metagrammar for semantic checking grammar
   simula.gram          the grammar for simula used in Orm
   standardsimula.gram grammar for unparsing to standard simula
hopehome/
   sun4/         fonts and postscript code
rasters/         raster files for icons
templates/       empty program and grammar templates
```

```
ormsimlib/      standard library modules
ormlab/         material from the Orm programming course
   lib/            modules used in the course
   skeletons/      skeleton modules to copy
   solutions/      solution programs
demo/           demonstration programs and grammars
doc/            documentation
```

10.6 Configuring the file browser

The file browser gives a configurable graphical interface to the UNIX file system. Its synopsis is

```
> finder [directory]
```

Configuration is done by associating types to files, based on file prefixes or suffixes (extensions). Each type can be given an icon, a default command and a type-specific menu.

To activate an object presented by the file browser , double-click on its icon. If it is a directory , a window is opened with the contents of the directory . If it is a file, the default command is executed. The right button shows the object' s menu.

The position, state, and size of icons and windows is retained between sessions. The information is saved in a file .finder -info in each directory .

If a directory is given when starting the file browser , it is used as a startup directory for the file browser . Otherwise the current directory is used. The file browser uses the IMAGEP ATH and FINDER_CONFIG environment variables as described above.

In the standard setup, the configuration file is `.../Orm/lib/orm-config`.

10.6.1 Configuration file syntax

Each line in the configuration file is a keyword followed by options.

<config> ::= <line>*

<line> ::= <comment> | <type-decl> | <suffix-rule> | <prefix-rule> | <type-menu> | <directory-menu> | <background> | <chain> | <head>

<comment> ::= **%** *everything to the end of line*

<type-decl> ::= **type** <type-id> <type-option>* [**command** <command-string>]

<suffix-rule> ::= **suffix** <suffix> <type-id>

<prefix-rule> ::= **prefix** <prefix> <type-id>

<type-menu> ::= **type-menu** <type-id> <menu-selection-name> <menu-option>* [**command** <command-string>]

<directory-menu> ::= **menu** <menu-selection-name> <menu-option>* [**command** <command-string>]

<background> ::= **background** <image-name>

<chain> ::= **chain** <type-id>+

<head> ::= **head** <type-id> <head-id>

where

<type-id> ::= *1-4 characters or an integer*

<type-option> ::= **flags** <flag>+ | **icon** <image-name> | **parameter** <prompt> [**default** <default-value>]

<image-name> ::= *name of icon image file*

<prompt> ::= *characters (may be enclosed in ')*

<default-value> ::= *characters (may be enclosed in ')*

<command-string> ::= *everything to end of line*

<suffix> ::= *text*

<prefix> ::= *text*

<menu-selection-name> ::= *characters (may be enclosed in ')*

<menu-option> ::= **flags** <flag>+ | **parameter** <prompt> [**default** <default-value>]

<flag> ::= H | I | W | L | S

H means hide the icon when the command is executed

I means that the file should be invisible (its icon is not shown)

W means wait after the command is executed without removing the command window . The window is remove by selecting *exit* in the window menu.

L means position the icon to the left of the title instead of above.

S means show the filename on the icon without its prefixes and suffixes.

A <type-decl> declares a file type and its default command. The type is identified by one or several suffix or prefix rules. Directories have the predefined type 1, and all files without explicitly defined types have the type 2.

Commands are executed in the bourne shell (sh) and if any input or output is needed, a command window will appear for this. If a parameter option is present, a textprompt window will appear to allow the user to supply the parameter . The command text may contain %s, %p or %p0, %p1, ... The %s is replaced with the file name. The %pn is replaced with the n' th parameter . %p is the same as %p0.

A <type-menu> rule defines a menu entry associated with a file type. A <directory-menu> rule defines a menu entry associated with open directory windows.

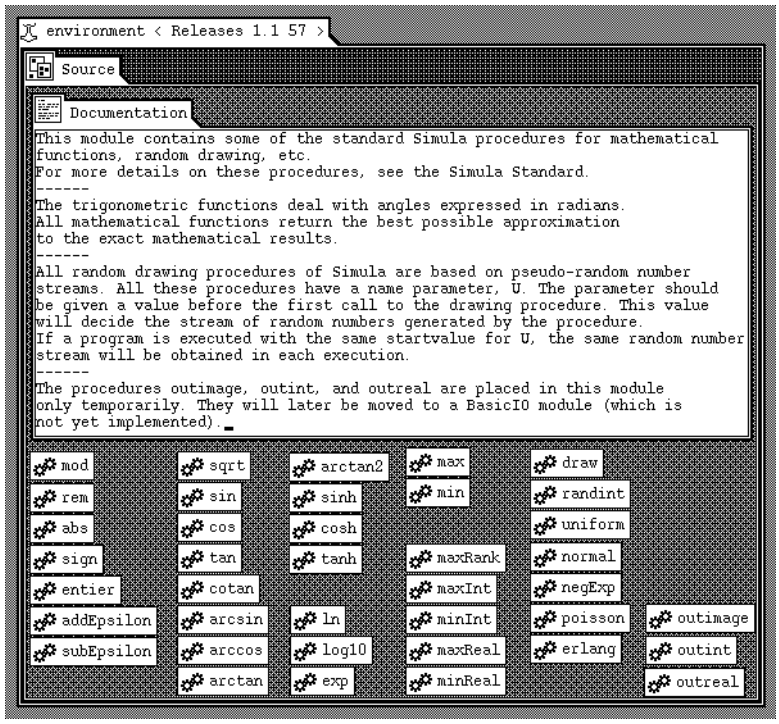
The <background> rule sets the background image of the outermost directory.

The <chain> and <head> rules are used for grouping several files together so only one object is shown on the screen. This is useful e.g. to group a source file together with its corresponding compiled object file. T o use this, define a type for each interesting suffix/prefix. Then use the <chain> rule to define these types as a “chain”. Then define another type which will act as a “headtype”. The “headtype” should not be associated with any suffix or prefix, but instead defined as the head of the chain by a <head> rule. <type-id> in the <head> rule should be one of the types in a chain (usually the first one). The <head-id> will then group all files in that chain together . A special menu is automatically added to grouped objects so individual files in the group can be displayed on request.

11.0 The OrmSimLib modules

This chapter documents the modules in the ormsimlib library .

11.1 environment.mjol



N.B. The seed parameters (U) to random functions are not updated properly because Orm Simula does not yet support out-parameters. Instead, the runtime system contains one global seed used for all random functions. The first time a random function is called, the seed is initialized. Then this seed is used for all subsequent calls to random functions. The seed is initialized only once per “Execution Observation” window . (If it is initialized to 0 the random procedures will home in on one value. Abort the execution and execute again.)

11.1.1 Basic operations

mod mod(i,base) (“i modulus base”) returns the integer remainder after dividing i by base, and adjusting the value to be between 0 and base.
Examples:
mod(9,7) = 2
mod(8,7) = 1
mod(7,7) = 0
mod(6,7) = 6

...
mod(2,7) = 2
mod(1,7) = 1
mod(0,7) = 0
mod(-1,7) = 6
mod(-2,7) = 5
mod(1, 7) = 1
mod(-1, 7) = 6
mod(1,-7) = -6
mod(-1,-7) = -1
RETURN integer
PARAMETER i : integer variable value
PARAMETER base : integer variable value

rem rem(i,j) returns the integer remainder after dividing i by j.
I.e.: rem(i,j) = i-(i//j)*j
RETURN integer
PARAMETER i : integer variable value
PARAMETER j : integer variable value

abs abs(e) returns the absolute value of e
Examples:
abs(4) = 4
abs(-4) = 4
RETURN real
PARAMETER e : real variable value

sign sign(e) =
0, if e = 0
1, if e > 0
-1, if e < 0
RETURN integer
PARAMETER e : real variable value

entier entier(e) returns the “floor” of the real value.
Examples:
entier(1.8) = 1
entier(-1.8) = -2
RETURN integer
PARAMETER e : real variable value

addEpsilon
addEpsilon(r) returns (r+epsilon) where epsilon is the smallest value such that (r <> r+epsilon) for this Simula implementation. However , addEpsilon(maxReal) = maxReal.
RETURN real
PARAMETER r : real variable value

subEpsilon
subEpsilon(r) returns (r -epsilon) where epsilon is the smallest value such that (r <> r -epsilon) for this Simula implementation. However , subEpsiolon(minReal) = minReal
RETURN real
PARAMETER r : real variable value

11.1.2 Mathematical functions

sqrt sqrt(r) returns the squareroot of r .
Examples:
sqrt(2) = 1.414...
RETURN real
PARAMETER r : real variable value

sin RETURN real
PARAMETER r : real variable value

cos RETURN real
PARAMETER r : real variable value

tan RETURN real
PARAMETER r : real variable value

cotan RETURN real
PARAMETER r : real variable value

arcsin RETURN real
PARAMETER r : real variable value

arccos RETURN real
PARAMETER r : real variable value

arctan arctan(r) returns a result in the range (0, pi/2) for nonnegative parameters and in the range (-pi/2, 0) for negative parameters
RETURN real
PARAMETER r : real variable value

arctan2 arctan2(y ,x) returns arctan(y/x). The result is in the range(-pi, pi) and a negative value is returned whenever y is negative. Positive y values always result in a positive result.
arctan2(0,x) returns zero if x is positive and pi if x is negative.
arctan2(0,0) results in a runtime error .
RETURN real
PARAMETER r1 : real variable value
PARAMETER r2 : real variable value

sinh RETURN real
PARAMETER r : real variable value

cosh RETURN real
PARAMETER r : real variable value

tanh RETURN real
PARAMETER r : real variable value

ln RETURN real
PARAMETER r : real variable value

log10 RETURN real
PARAMETER r : real variable value

exp RETURN real
PARAMETER r : real variable value

11.1.3 Extremum functions

max max(r1,r2) returns the greatest value of r1 and r2.
Example:

max(4,7) = 7
RETURN real
PARAMETER r1 : real variable value
PARAMETER r2 : real variable value

min min(r1,r2) returns the least value of r1 and r2.
Example:
min(4,7) = 4
RETURN real
PARAMETER r1 : real variable value
PARAMETER r2 : real variable value

11.1.4 Environmental enquiries

maxRank
Returns the greatest legal argument to the standard procedure "char".
RETURN integer

maxInt Returns the greatest integer value in this Simula implementation.
RETURN integer

minInt Returns the least integer value in this Simula implementation.
RETURN integer

maxReal
Returns the greatest real value in this Simula implementation.
RETURN real

minReal
Returns the least real value in this Simula implementation.
RETURN real

11.1.5 Random drawing

draw draw(a, U) is
TRUE with the probability a, F ALSE with the probability 1-a. It is always TRUE if a >= 1 an always F ALSE if a <= 0.
RETURN boolean
PARAMETER a : real variable value
PARAMETER U : integer variable value

randint randint(a,b,U) returns one of the values a, a+1, ..., b-1, b with equal probability . If b < a, the call leads to a runtime error .
RETURN integer
PARAMETER a : integer variable value
PARAMETER b : integer variable value
PARAMETER U : integer variable value

uniform uniform(a,b,U) returns a value drawn from the uniform distribution. The value is uniformly distributed in the interval a <= uniform(a,b,U) <= b. If b > a the call leads to a runtime error .
RETURN real
PARAMETER a : real variable value
PARAMETER b : real variable value
PARAMETER U : integer variable value

normal normal(a,b,U) returns a value drawn from the normal distribution with mean a and standard deviation b.
RETURN real
PARAMETER a : real variable value
PARAMETER b : real variable value
PARAMETER U : integer variable value

negExp negExp(a,U) returns a value drawn from the negative exponential distribution with mean 1/a. This is the same as a random "waiting time" in a Poisson distributed arrival pattern with expected number of arrivals per time unit equal to a. If a <= 0, a runtime error occurs.
RETURN real
PARAMETER a : real variable value
PARAMETER U : integer variable value

poisson poisson(a,U) returns a value drawn from the Poisson distribution with parameter a.
RETURN integer
PARAMETER a : real variable value
PARAMETER U : integer variable value

erlang erlang(a,b,U) returns a value drawn from the Erlang distribution with mean 1/a and standard deviation 1/(a*sqrt(b)). Both a and b must be greater than zero.
RETURN real
PARAMETER a : real variable value
PARAMETER b : real variable value
PARAMETER U : integer variable value

11.1.6 Temporary output operations

Note! These operations are here for historical reasons. W e do not recommend using them since the standard output in Orm is usually redirected to a file (`~/.ormlog`) so you will not see the output. Use the W riteInteger and WriteT ext operations in uil.mjol instead.

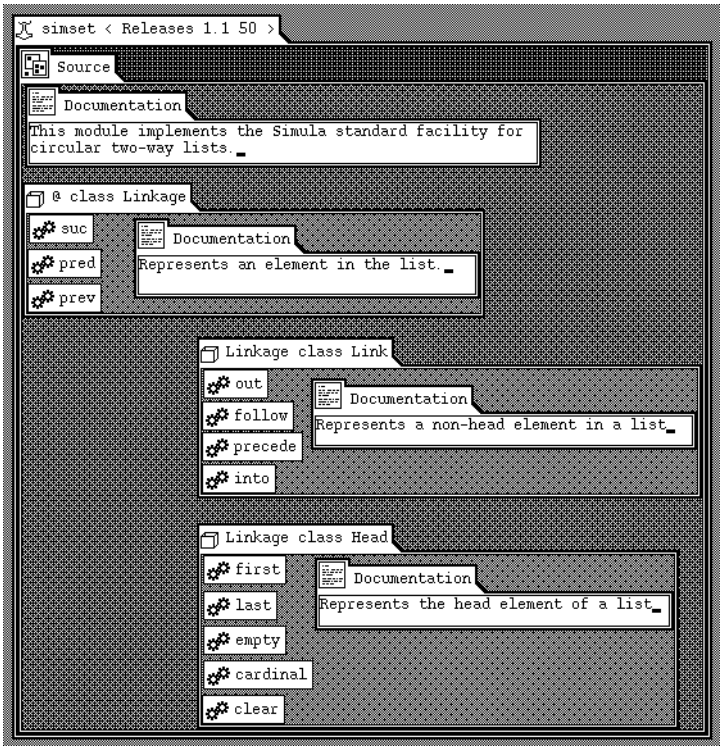
outimage Writes a linefeed on "standard output". Note that previous "outint" and "outreal" are buffered until this call.
RETURN nothing

outint outint(i,w) writes an integer i on the "standard output". w determines the width of the output field.
w > 0, the item is right-adjusted in w positions.
w = 0, the item is output in as many positions as needed (no trailing or leading spaces).
w < 0, the item is left-adjusted in (-w) positions.
w < 0, the item is left-adjusted in (-w) positions.
RETURN nothing
PARAMETER i : integer variable value
PARAMETER w : integer variable value

outreal outreal(r ,d,w) writes a real, r , on the "standard output". The output item will contain an exponent part and a part preceding the

exponent.
d determines output of the exponent.
d = 0, the exponent is only preceded by a sign part
d = 1, the exponent is preceded by a one-digit integer
d > 1, the exponent is preceded by a one-digit integer followed by a fraction of d-1 digits.
w determines the width of the output field.
w > 0, the item is right-adjusted in w positions.
w = 0, the item is output in as many positions as needed (no trailing or leading spaces).
w < 0, the item is left-adjusted in (-w) positions.
d < 0 causes a runtime error .
RETURN nothing
PARAMETER r : real variable value
PARAMETER d : integer variable value
PARAMETER w : integer variable value

11.2 simset.mjol



11.2.1 class Linkage

suc Returns the successor Link of this object. This object may be either a Link or a Head. Returns NONE if this is the last Link in the list.
RETURN ref(Link)

pred Returns the predecessor Link of this object. This object may be either a Link or a Head. Returns NONE if this is the first Link in the list.
RETURN ref(Link)

prev Returns the predecessing element of this object. This object may be either a Link or a Head. Returns the Head if this is the last Link in the list.
RETURN ref(Linkage)

11.2.2 Linkage class Link

out out removes this Link from the list of which it is an element. The procedure has no effect if this Link is not in any list.
RETURN nothing

follow Follow(L) inserts the Link L after this Link. If L was previously a Link in another list, it is removed from that list.
RETURN nothing
PARAMETER L : ref(Linkage) variable ref

precede Precede(L) inserts the Link L before this Link. If L was previously a Link of another list, it is removed from that list.
RETURN nothing
PARAMETER L : ref(Linkage) variable ref

into into(H) inserts this Link last in the list with Head H
RETURN nothing
PARAMETER H : ref(Head) variable ref

11.2.3 Linkage class Head

first Returns the first Link of this list
RETURN ref(Link)

last Returns the last Link of this list.
RETURN ref(Link)

empty Returns TRUE if there are no Link elements in this list
RETURN boolean

cardinal Returns the number of Link elements in this list.
RETURN integer

clear Removes all the Link elements from this list.
RETURN nothing

11.3 text.mjol

The text.mjol module differs from the Standard Simula text concept:

- Use operations *denote* and *assign* to simulate ":-" and ":=".
- Use operations *refequal* and *equal* to simulate "==" and "=".
- Use the function *TextConstant* to generate a constant text.

11.3.1 Text utilities

LOCAL **notext** : ref(Text) variable

TextConstant

Creates a new Text extObject with the contents of aString
RETURN ref(Text) !The text containing aString;
PARAMETER aString : string variable value !simple string;

blanks Creates a new Text extObject of length n filled with space characters
RETURN ref(Text)
PARAMETER n : integer variable value

copy Creates a new Text extObject which is a copy of the TextObject denoted by aText
RETURN ref(Text)
PARAMETER aText : ref(Text) variable value

11.3.2 class Text

start Returns the current start of the text
RETURN integer

length Returns the current length of the text
RETURN integer

main Returns a T ext denoting the whole string of which this T ext denotes a substring
RETURN ref(T ext)

pos Returns the current position in the text
RETURN integer

setpos Sets the current position in the text
RETURN nothing
PARAMETER newPos : integer variable value

more Test if the current position is beyond the last position in the text
RETURN boolean

getchar Returns the character at the current position in the text. Increments the position by 1
RETURN character

putchar Inserts aChar at the current position in the text. Increments the position by 1
RETURN nothing
PARAMETER aChar : character variable value

sub Returns a T ext denoting a substring of this T ext's contents
RETURN ref(T ext)
PARAMETER first : integer variable value
PARAMETER length : integer variable value

strip Returns a T ext denoting a substring of this T ext's contents without any trailing blanks
RETURN ref(T ext)

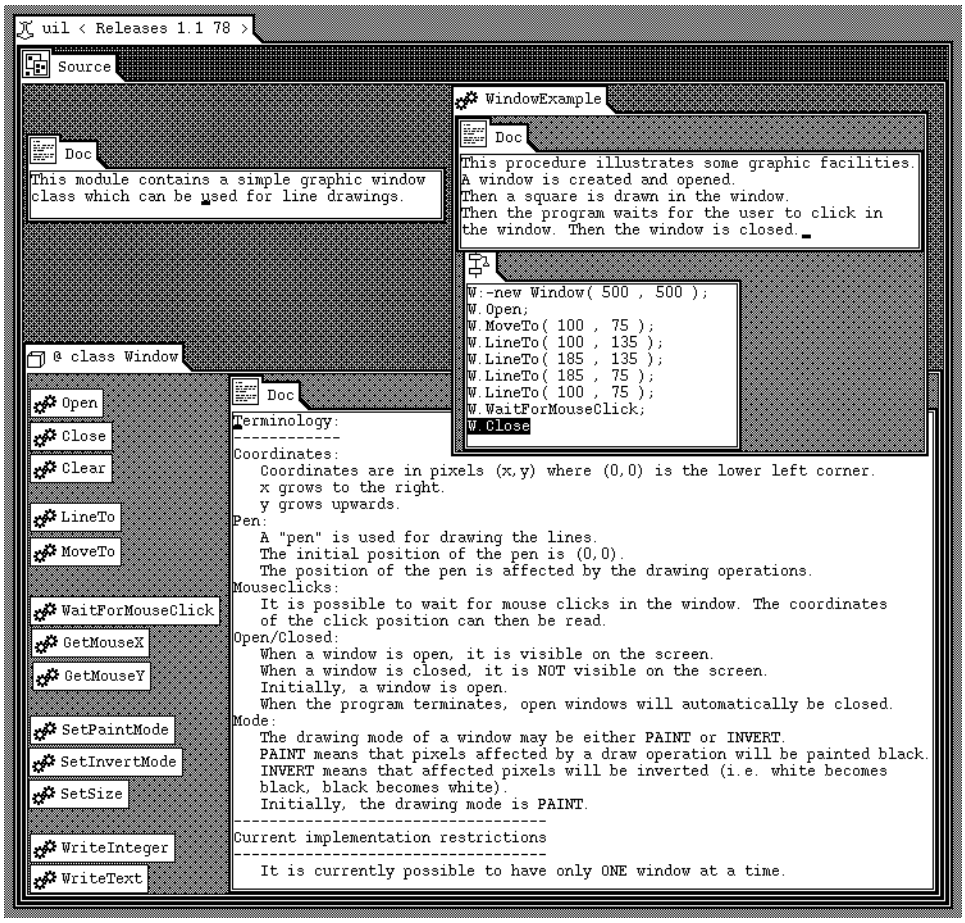
denote Simulates the :- operation
RETURN nothing
PARAMETER aT ext : ref(T ext) variable value

assign Simulates the := operation
BUGS: Does not cope with overlapping texts
RETURN nothing
PARAMETER aT ext : ref(T ext) variable value

refequal Simulates the operation == between this T ext and aT ext
RETURN boolean
PARAMETER aT ext : ref(T ext) variable value

equal Simulates the operation = between this T ext and aT ext
RETURN boolean
PARAMETER aT ext : ref(T ext) variable value

11.4 uil.mjol



11.4.1 class Window

PARAMETER w : integer variable value !W indows width in pixels;
PARAMETER h : integer variable value !W indows height in pixels;

Clear Clears the window .
RETURN nothing

SetPaintMode
Sets the drawing mode to P AINT.
RETURN nothing

LineT o The pen is moved from the current position to the position (x,y) while drawing a straight line.
RETURN nothing
PARAMETER x : integer variable value
PARAMETER y : integer variable value

MoveT o The pen is moved from the current position to the position (x,y) while in a “lifted” state.
RETURN nothing

PARAMETER x : integer variable value
PARAMETER y : integer variable value

SetInvertMode
Sets the drawing mode to INVERT .
RETURN nothing

Open Makes the window visible on the screen.
RETURN nothing

Close Closes the window (the window disappears from the screen).
RETURN nothing

WaitForMouseClick
Stops the execution until the user has clicked in this window .
RETURN nothing

GetMouseX
Returns the x position of the latest mouse click. Initially 0.
RETURN integer

GetMouseY
Returns the y position of the latest mouse click. Initially 0.
RETURN integer

SetSize Sets the size of the window to have width w and height h.
RETURN nothing
PARAMETER w : integer variable value
PARAMETER h : integer variable value

WriteInteger
Writes the integer value 'i' right-adjusted into a character field with the length 'w' characters. The lower left corner of the character field is placed on current output position.
RETURN nothing
PARAMETER i : integer variable value
PARAMETER w : integer variable ?

WriteT ext
Writes the characters of aT ext at the current output position.
RETURN nothing
PARAMETER aT ext : ref(T ext) variable ref

Index

A
Abort 31
ABSTRACT 9
abstract grammar 9, 44
abstract grammar window 10
activation window 4
activations 31
Active block 23
Active body 23
Active declarations 23
alternation 10
alternative 49
alternative name 49
attribute content 49
attribute name 49

B
Back 18
back 21
Binding of Relations 61
Binding rule 43, 50
Binding type 50
block 30
block clipboard 30
Block window 23
block windows 30
Block-level Editing 30
Body window 23
body window 3
breakpoint 31
browser 69

C
Change import binding 35
Change import rule 35
Choose selected 51
Close 17
CODEGEN 9
CONCRETE 9
concrete grammar 9, 45
concrete grammar window 10
construction 10
contents 16
contents menu 17
Continue 31
continue 5
Copy 4, 26
copy 21
CopyFromClipboard 30
Copying blocks 30
CopyToClipboard 31
current execution point 31
Cut 26

D
De/Attach 18
Declaration window 23
Declare new import 34, 35
Declared imports 23
Delete 57
delete 30
Delete import 36
Delete window 32
Directory window 23, 42
Documentation window 23
Documenting programs 33
dynamic breakpoint 31
Dynamic Rebinding 56

E
Edit as text 4, 27
Edit lexem 11, 26
edit title as structure 30
edit title as text 30
Editing block titles 30
Enter Selection Criteria 56
Entering 24
Environment variables 71
environment.mjol 34
Error recovery 39
evolution graph 49
Evolution Graph Window 54
evolution graph window 52
Execute 3, 31
execution 31
Execution window 23

Exit 8
exit 20
Exit the system 8
Exiting 25
Expand 5, 9, 26
Expand After 5
Expand after 9, 26
Expand before 26
Expand list 26
Expansion 26
Explain 7, 28
Explicit Rebinding 56

F
File Browser 20
finder 69
Fixed Relation 56
fork revision 54
frame 16
Front 18
front 21

G
Grammar aspects 42
Grammar Editor 9, 42
grammar formalisms 44
Grammar window 23, 42
grammar window 37
Gred 9
gred 69

H
Hide window 30
Hold 31

I
icon 16
icon menu 17
import a module 34
Import declarations 35
Incremental Semantic Checking 28
Insert Revision Window 60
Inserting and deleting blocks 30
InvCode 32

K
Keyboard shortcuts 29

L
LB 16
lexeme 10
list 10
localshell 20

M
MB 16
metagrammars 48
Misc 9
Mjølner 1
Modules 33, 66
Mouse 16
Move 17
move 21
MoveFromClipboard 30
MoveToClipboard 30

N
Names 6, 27
new class 30
new directory 20
new grammar 20
new procedure 30
new simula program 20
new textfile 20

O
object window 4
objects 31
observation window 3
Open 17, 24
open 21
Organize 18
Orm 1
orm 20
Orm Simula 65
ormlab/lib 33
ORMMODULES 33
ormsimlib 33

P
Paste after 4
Paste before 26
PBO 49
PBO Window 53
PBO-window 52
placeholder 5
Placeholders 25
point 31
Popup menu 17
Print program 37
procedure window 3
production types 10
Program activation 23
Program Base Object 49
Program I/O window 23
program I/O window 3
Program window 23
program window 3
Purge 37

Q
Quit 8, 25

R
RB 16
reactivation point 31
Rebind “latest” imports 36
rebindmodules 37
Rebuild Menus 12
Rebuild menus 43
Redisplay 28, 43

Refresh 18
refresh 21
Relations 49
Rem all breaks 31, 33
remove 21
Remove breakpoint 33
Rename 57
Reset 31
Resize 18
Reunparse 12, 43
Revision description 50
revision name 49
revision number 49
Revision window 59
revision window 52
RevisionId 49
rts 69
running 31

S
Save 25
Save as alternative 59
Save as revision 59
Save as text 43
Save Documentation only 37
Save interface 37
Save program as text 37
Saving 25
Scrollbars on/off 9, 28, 43
Selection 25
SEMANTIC 9
Semantic Editing 27
Semantic editing 6
semantic editing 27
Set dynamic breakpoint 33
Set startprod 43
Set static breakpoin 33
Set title 43
shelltool 20
Shortcuts 29
Show all implicit imports 36
Show dynamic father 33
Show selection 56
simset.mjol 34
source revision 54
Source window 23
Standard Simula 65
Start 3, 31
Static analysis errors 6
static breakpoint 31
static semantic errors 28
Step/into 31
step/into 4
Step/over 31
step/over 4
Stop execution 31
Stop in ALL modules 31
Stop in main only 31
stopped 31
Structure editing 5
structure-oriented 25
surface syntax 10

T
Tailorability 1
TARGET 9
Target window 42
target window 9
Text 19
Text editing 5
text.mjol 34
Textprompts 19
Textual Editing 27
title 16
title menu 17

U
uil.mjol 34
Use import 34
Use import windows 23

V
Virtual Procedures 65

W
Window 16
window menu 17
Window Structure 52
Window structure 22
Wrap 5, 18