Mjolner Tool Workshop

Mjølner Informatics Report
MIA 01–44
March 2004

Table of Contents

1 Mjolner Workshop

These notes contain five exercises which cover the most important features of the Mjolner tool. Furthermore an appendix containing keyboard shortcuts is supplied –– you might want to read that section before you continue.

To get started on the exercises, download the files syntaxedit.bet, useloops.bet og loops.bet to your own directory.
Then start the Mjolner tool ('mjolner' – not 'mjolner &').

Now you are ready for the exercises.

Please report any errors you may find in this document to support@mjolner.dk

Author: Christian Heide Damm (damm@daimi.au.dk), October 1999
Updated by Elmer Sandvad (ess@mjolner.dk), September 2000
Updated and translated by Karsten Jorgensen (karsten@mjolner.dk), May 2001

1.1 Exercise 1: The basics – editing and compiling

In this exercise we will construct a program that reads a list of integers and prints the minimum and maximum value.

Choose 'File–>New BETA Program...' Name the program 'minmax.bet'.
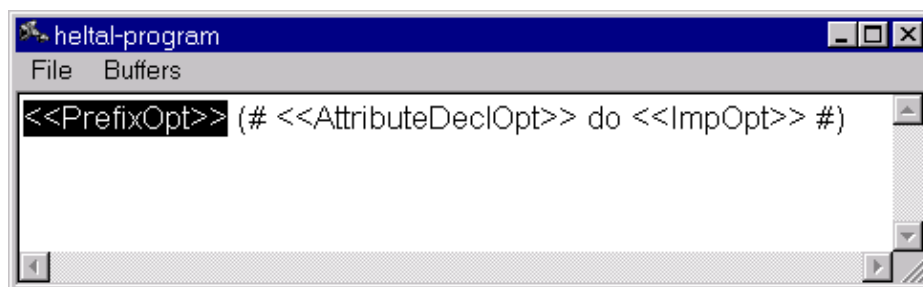
The `<<PrefixOpt>>` in the Form window means that you can type a super pattern here. We do not want that here, so click `<<PrefixOpt>>` and press Delete.

The `<<AttributeDeclOpt>>` means that you can declare attributes here. The program needs two integers: `min` and `max`.

Click `<<AttributeDeclOpt>>` and type `min,max: @integer`. Choose 'Edit–>Parse Text' or

press Ctrl+space to leave text editing mode.

The `<<ImpOpt>>` means that you can type imperatives here. We need to initialize `min` and `max`, so click `<<ImpOpt>>` and type `maxint->min; minint->max`. Leave text editing mode.

Now we need a loop. Mark `minint->max` by holding down the mouse button from the middle of `minint` to the middle of `max`.

```
heltal-program                                    _ □ ✕
 File   Buffers
(# min,max: @integer do maxint->min; minint->max #)
```
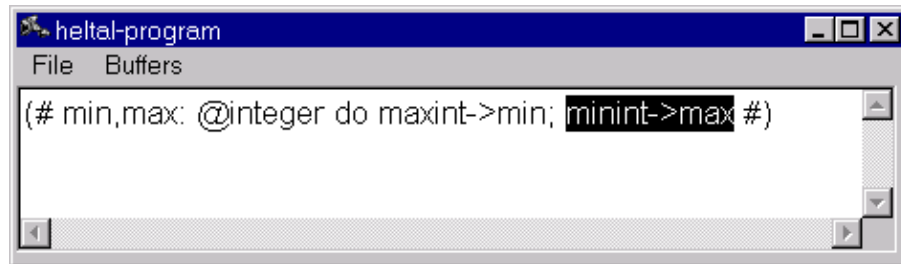
Then append some "empty code" by choosing 'Edit–>Insert After' or by pressing Enter.

```
heltal-program                                    _ □ ✕
 File   Buffers
(# min,max: @integer do maxint->min; minint->max; <<ImpOpt>> #)
```
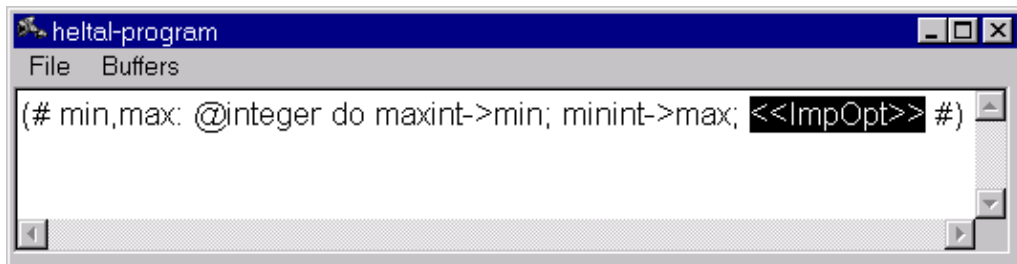
Replace the `<<ImpOpt>>` by `myloop: (# do restart myloop #)`. Leave text editing mode.

```
heltal-program                                    _ □ ✕
 File   Buffers
(# min,max: @integer
do maxint->min; minint->max; myloop: (#  do restart myloop #)
#)
```
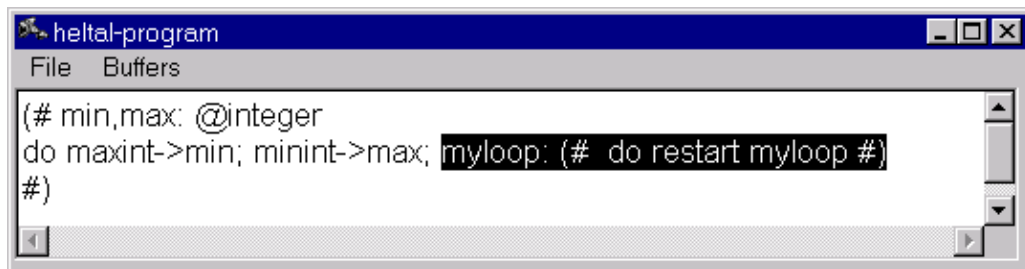
Mark `myloop` as shown above and re–enter text editing mode by pressing Ctrl+space or F2. Type the code in the loop, so that it looks like this:

Is the program correct?

Find out by checking it: choose 'Compile/Run–>Check Current (minmax)' or click on the Check button ✖ in the toolbar. If there are errors in the program a window showing the errors will appear:

You can click the 'Nodes' list to browse through the errors. At the bottom of the window information on the error is displayed: "i is not declared". Insert a declaration of `i` by clicking on `max` and

inserting "empty code" (using menus or pressing Enter). Type `i` and leave text editing mode.
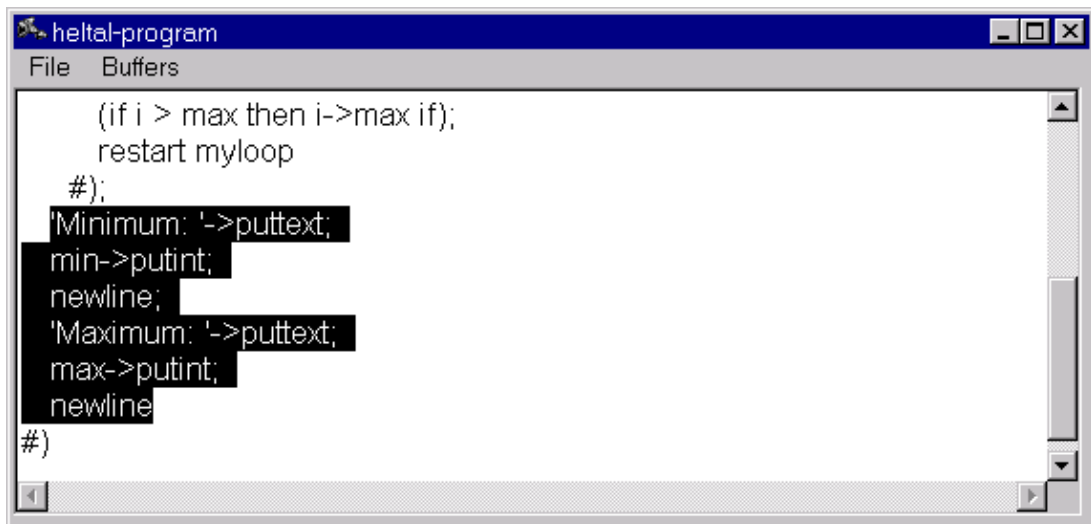
```
heltal-program
File   Buffers

(# min,max,|: @integer
do
   maxint->min;
   minint->max;
   myloop:
     (#
     do
        getint->i;
        (if i = 0 then leave myloop if);
        (if i < min then i->min if);
        (if i > max then i->max if);
        restart myloop
     #)
#)
```
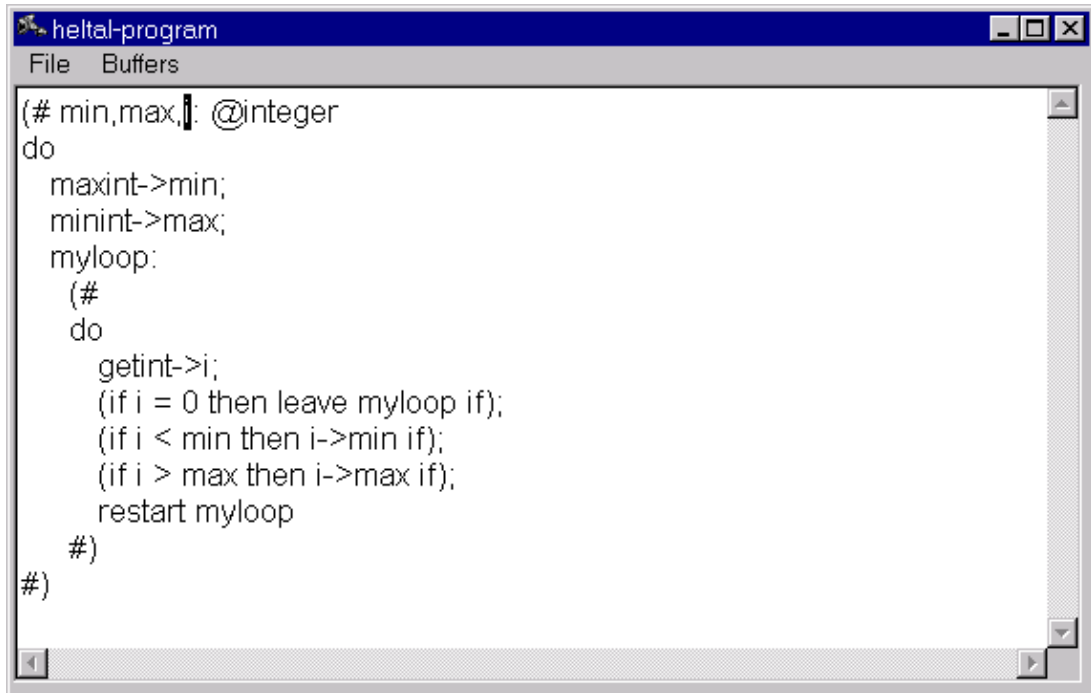
Check the program again. This time it should work.

Now we insert a few statements that print the results of the loop. Mark all of `myloop` and insert some "empty code" after the loop and add the lines shown below:

```
heltal-program
File   Buffers

        (if i > max then i->max if);
        restart myloop
      #);
   'Minimum: '->puttext;
   min->putint;
   newline;
   'Maximum: '->puttext;
   max->putint;
   newline
#)
```
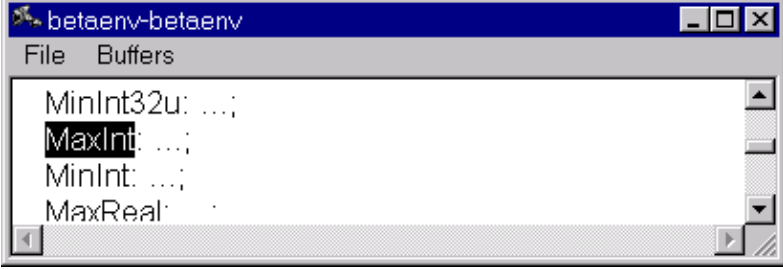
Compile and run the program from the Mjolner tool (NOT in a shell). You can interact with the program in the shell where the Mjolner tool was started.

1.2 Exercise 2: Browsing code

In this exercise we will use the program 'minmax' from exercise 1 to illustrate code browsing in the Mjolner tool.

1.2.1 What is maxint/minint?

In 'minmax' we use `maxint` and `minint`. Find out how they are defined by marking e.g. maxint and clicking the Definition button 🔲 in the toolbar (or double−click `maxint`, or press Alt+Right).
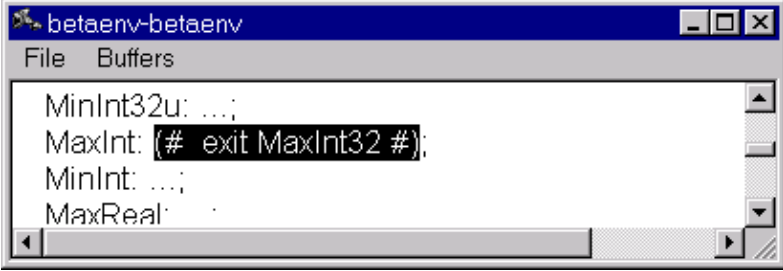
Maxint is apparently defined in the file 'betaenv'. Double−click the three dots beside `maxint`.
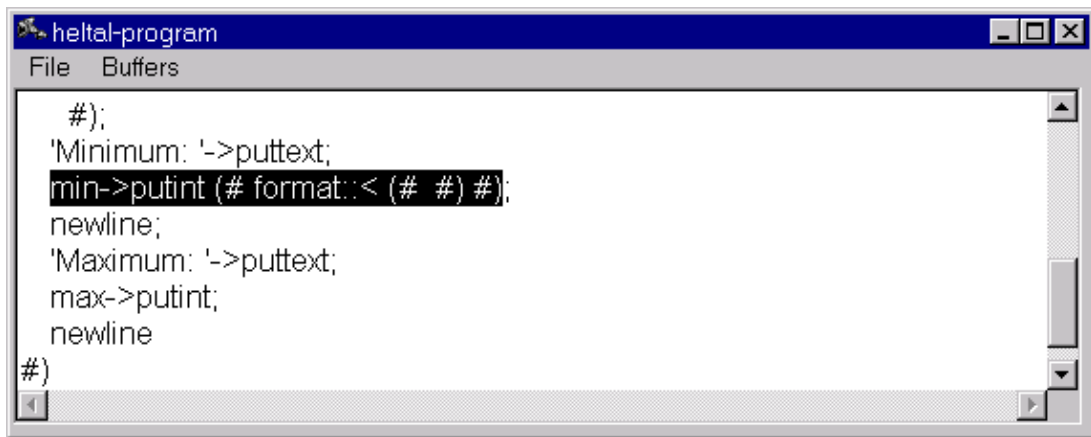
Continue the same way with `maxint32` to find out the actual value.

Now we return to 'minmax'. Click the Back button ⬅ in the toolbar (or right−click and choose 'back', or press Ctrl+Left). Go back until the 'minmax'−program is displayed.

1.2.2 Adjust the 'Minimum'−output right and add zero−padding

Mark `min->putint` and enter text editing mode. Add `(# format::< (##) #)` to `putint` as shown:

Double−click `format`. Because the compiler has not yet checked the text you have just typed, it does not yet know what it means. So answer yes when asked whether to check the program.

Now you can see the attributes of `putint`, they should be self−explanatory. Further bind `putint.format` in `minmax`, so that it prints `min` right adjusted with width 10 and zero−padding.

1.2.3 Which containers are available?

You have probably heard of lists and arraycontainers, but there are many other containers. At the top of the Projects window there is a line that reads 'Std. Libraries/'. Double−click that line or click '+' to open it. Then open 'containers'.

Now all the files in ~beta/containers/ are displayed. Browse through the files and inspect the

different kinds of containers. There are quite a few of them.

Whenever you find ⟨*⟩ in a program, it means the program contains a comment at that point. If you double−click the asterisk, the comment is displayed. Double−click again, and the comment is hidden. You can also use Alt+Right or the Definition button 🗋 in the toolbar.

If ever you find that too much code is being displayed and you would like an overview of the program, you can press Alt+Up. This hides all the details. Double−click to get more details (or mark and press Alt+Right or click the Definition button 🗋 in the toolbar). To show all details in a selection recursively, press Alt+Down.

1.2.4 What else is in the Beta system?

You should know about the files listed below −− have a quick look at them:

- basiclib
- directory (interface to directories on a disk)
- file (file handling)
- math (mathematical computation)
- random (random numbers)
- guienv
- controls (elements for graphical user intefaces, such as buttons, text fields, check−boxes)
- stddialogs (standard dialogue elements, such as file dialogues)
- persistentstore (for storing objects on disk)
- sysutils
- envstring (access environment variables, e.g. $USER)
- time (handle time and dates)

1.3 Exercise 3: Editing using syntactic code fragments

A central feature of the Mjolner tool is that it enables you to copy and move around large chunks of code quite easily −− without running into syntactic errors such as missing brackets etc. Unfortunately, it is difficult to illustrate the usefulness of this feature in a small example. Therefore this exercise consists of a description of the techniques required to use syntactic editing.

Load the file syntaxedit.bet.

1.3.1 Marking code fragments

First of all, you need to know how to mark a chunk of code. In the Mjolner tool it is impossible to mark something that is not a "whole", for instance you cannot select half a descriptor. The tool will always expand the marking until it fits a syntactic category.
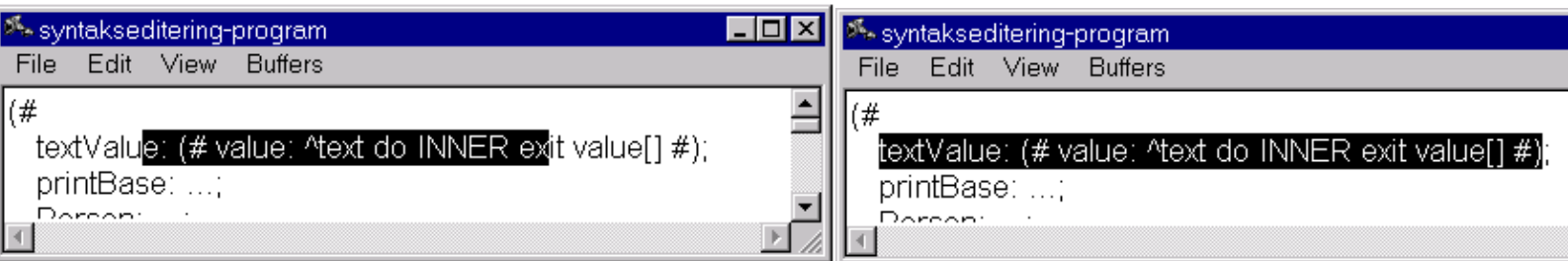
In the example above, the selection becomes the entire pattern.

Another example: Mark from the middle of one pattern to the middle of the next.
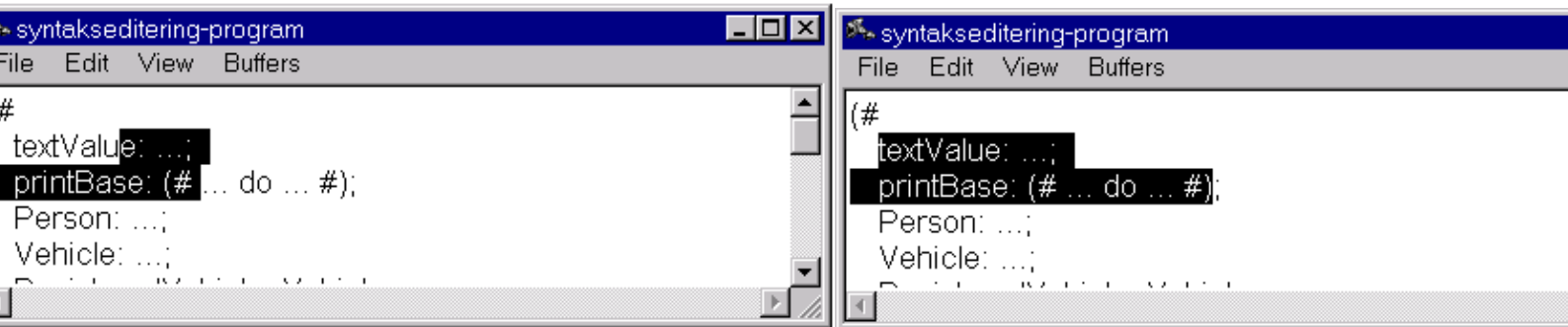
The tool selects both patterns.

Try marking different chunks of code and observe what is being selected. For example, try marking the following:

- `UnregisteredVehicle: Vehicle ...;`
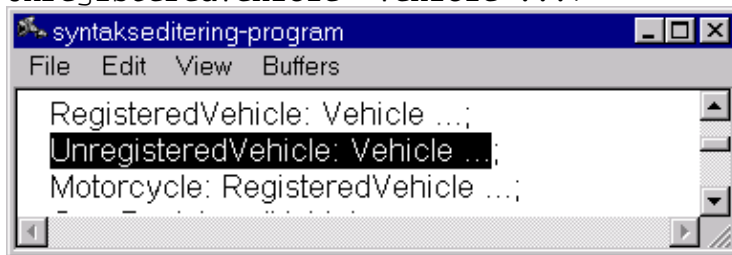
- `Person: ...` until `aPerson: @person`
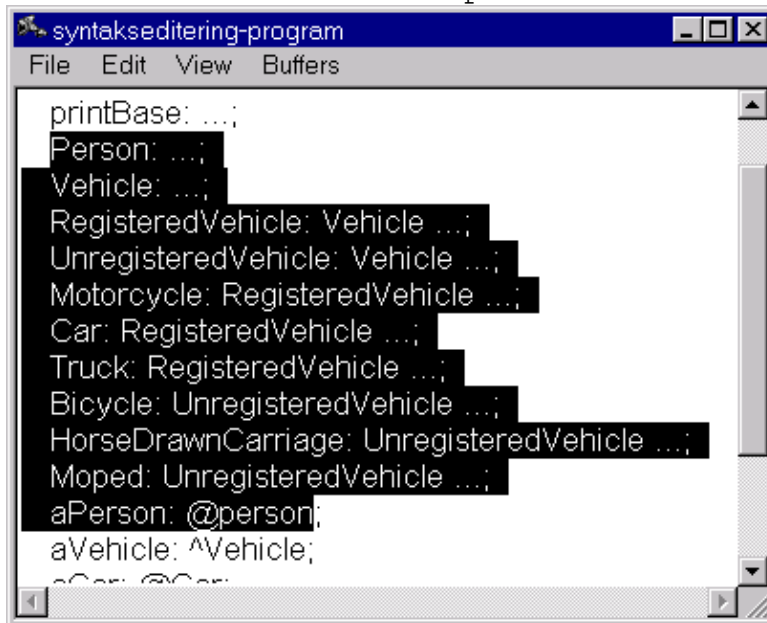
- Inside `Person.print` mark `'living at'->puttext`

```
syntakseditering-program                    _ □ ✕
File   Edit   View   Buffers

   Person:
    (#
       init: ...;
       print: printBase
         (# name:: ...;
         do
           nl;
           'living at '->puttext;
           Address[]->puttext;
```

- Inside `Person.print` mark the entire do–part

```
syntakseditering-program                    _ □ ✕
File   Edit   View   Buffers

   Person:
    (#
       init: ...;
       print: printBase
         (# name:: ...;
         do
           nl;
           'living at '->puttext;
           Address[]->puttext;
           nl;
           (if Age > 21 then 'already '->puttext else
           Age->putint;
           ' years old'->puttext;
           nl;
           (if not Kids.empty then ... if);
           (*)
         #);
       Name: @text;
```

Experiment with the use of Alt+Up (overview), Alt+Left (abstract recursively), Alt+Right (detail) and Alt+Down (detail recursively).
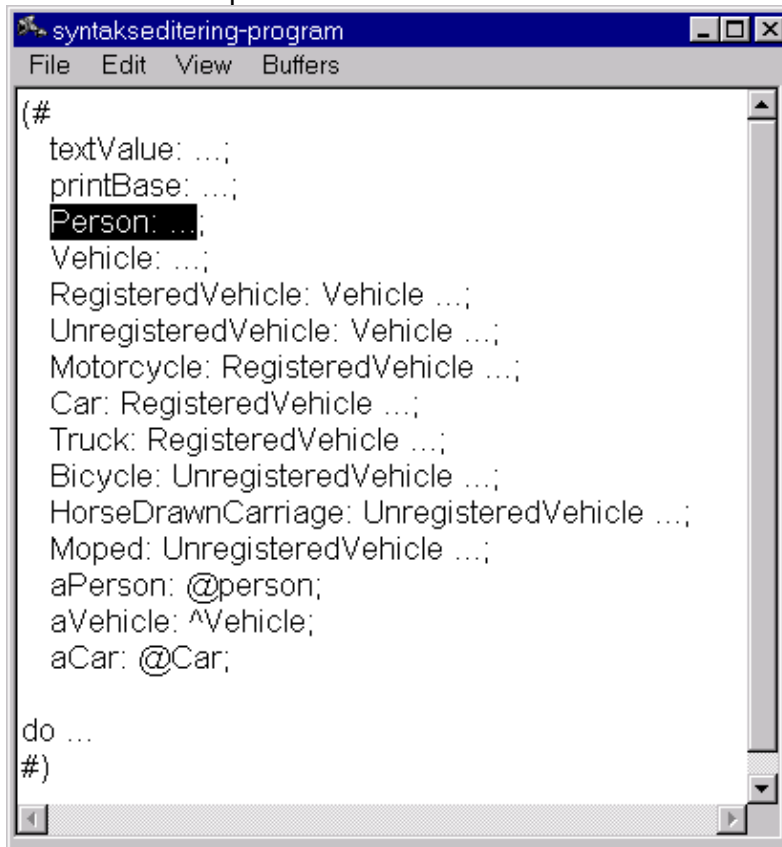
1.3.2 Code fragments: Cut, copy paste

The tool supports cut, copy, paste, undo and redo on the usual keys:

Command	Keyboard shortcut
cut	Ctrl+X
copy	Ctrl+C
paste	Ctrl+V
undo	Ctrl+Z
redo	Ctrl+Y

Now we want to move the Person pattern down to aPerson: @person:
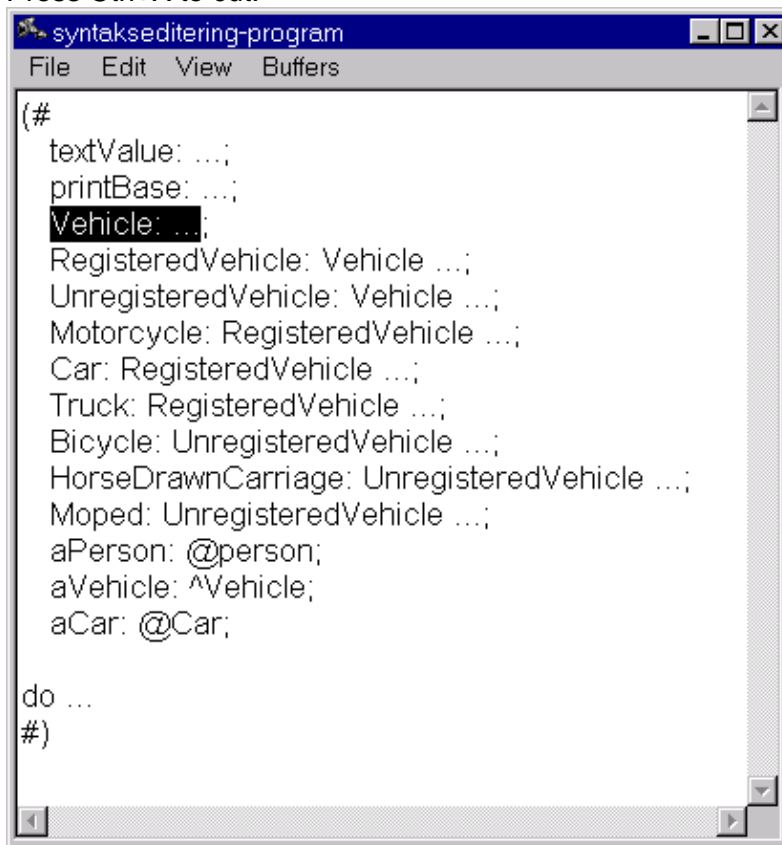
1. Mark the Person pattern.

```
syntakseditering-program                    _ □ ×
 File   Edit   View   Buffers
(#
  textValue: ...;
  printBase: ...;
  Person: ...;
  Vehicle: ...;
  RegisteredVehicle: Vehicle ...;
  UnregisteredVehicle: Vehicle ...;
  Motorcycle: RegisteredVehicle ...;
  Car: RegisteredVehicle ...;
  Truck: RegisteredVehicle ...;
  Bicycle: UnregisteredVehicle ...;
  HorseDrawnCarriage: UnregisteredVehicle ...;
  Moped: UnregisteredVehicle ...;
  aPerson: @person;
  aVehicle: ^Vehicle;
  aCar: @Car;

do ...
#)
```

2. Press Ctrl+X to cut.

```
syntakseditering-program                    _ □ ×
 File   Edit   View   Buffers
(#
  textValue: ...;
  printBase: ...;
  Vehicle: ...;
  RegisteredVehicle: Vehicle ...;
  UnregisteredVehicle: Vehicle ...;
  Motorcycle: RegisteredVehicle ...;
  Car: RegisteredVehicle ...;
  Truck: RegisteredVehicle ...;
  Bicycle: UnregisteredVehicle ...;
  HorseDrawnCarriage: UnregisteredVehicle ...;
  Moped: UnregisteredVehicle ...;
  aPerson: @person;
  aVehicle: ^Vehicle;
  aCar: @Car;

do ...
#)
```

3. Mark Moped: UnregisteredVehicle.

```
(#
  textValue: ...;
  printBase: ...;
  Vehicle: ...;
  RegisteredVehicle: Vehicle ...;
  UnregisteredVehicle: Vehicle ...;
  Motorcycle: RegisteredVehicle ...;
  Car: RegisteredVehicle ...;
  Truck: RegisteredVehicle ...;
  Bicycle: UnregisteredVehicle ...;
  HorseDrawnCarriage: UnregisteredVehicle ...;
  Moped: UnregisteredVehicle ...;
  aPerson: @person;
  aVehicle: ^Vehicle;
  aCar: @Car;

do ...
#)
```

4. Press Enter to insert code.

```
(#
  textValue: ...;
  printBase: ...;
  Vehicle: ...;
  RegisteredVehicle: Vehicle ...;
  UnregisteredVehicle: Vehicle ...;
  Motorcycle: RegisteredVehicle ...;
  Car: RegisteredVehicle ...;
  Truck: RegisteredVehicle ...;
  Bicycle: UnregisteredVehicle ...;
  HorseDrawnCarriage: UnregisteredVehicle ...;
  Moped: UnregisteredVehicle ...;
  <<AttributeDeclOpt>>;
  aPerson: @person;
  aVehicle: ^Vehicle;
  aCar: @Car;

do ...
#)
```

5. Press Ctrl+V to paste.
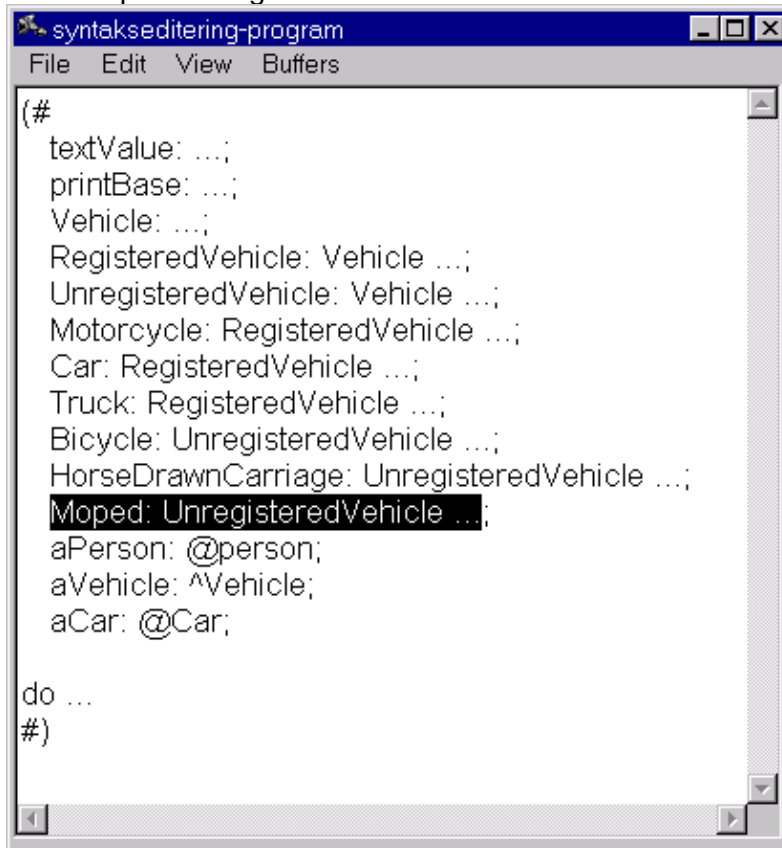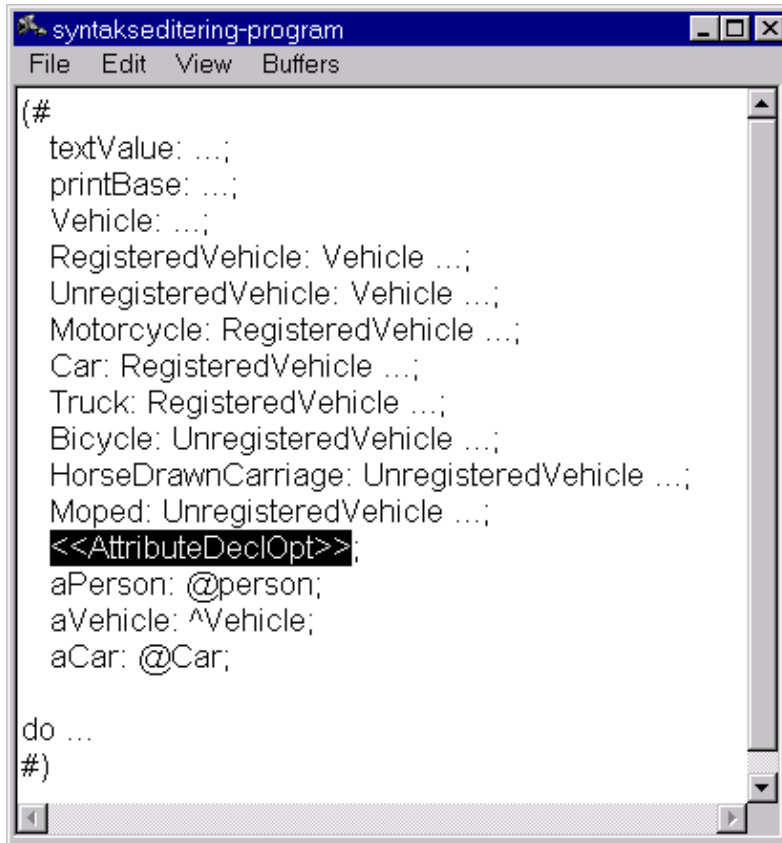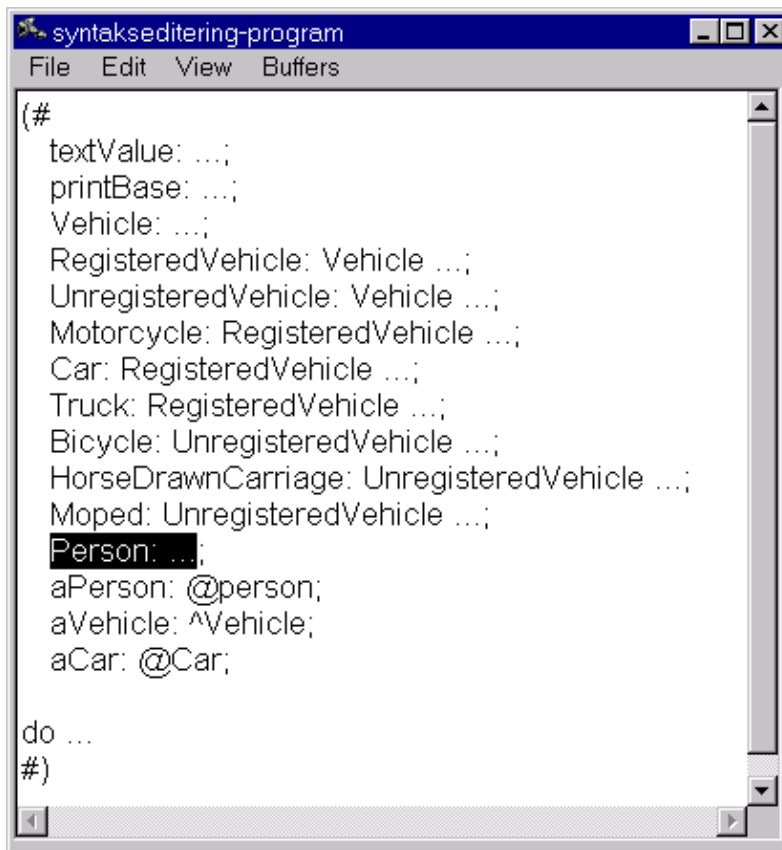
```
syntaksediting-program                    _ □ ✕
 File   Edit   View   Buffers
(#
  textValue: ...;
  printBase: ...;
  Vehicle: ...;
  RegisteredVehicle: Vehicle ...;
  UnregisteredVehicle: Vehicle ...;
  Motorcycle: RegisteredVehicle ...;
  Car: RegisteredVehicle ...;
  Truck: RegisteredVehicle ...;
  Bicycle: UnregisteredVehicle ...;
  HorseDrawnCarriage: UnregisteredVehicle ...;
  Moped: UnregisteredVehicle ...;
  Person: ...;
  aPerson: @person;
  aVehicle: ^Vehicle;
  aCar: @Car;

do ...
#)
```

Now try undoing it all using Ctrl+Z.

Notice that the `Person` pattern could have been thousands of lines of code, and it would still be very simple to move it around or copy it.

You can apply this to all sorts of other code fragments: imperatives, descriptors, do–parts, enter–parts, exit–parts, variabels, superpattern–prefixes. But it does take a little practice to be able to use it efficiently.

1.4 Exercise 4: The fragment system

This exercise is about the fragment system and how it is represented in the Mjolner tool.

1.4.1 Hiding details of your implementation

Load the files loops.bet and useloops.bet in the Mjolner tool.
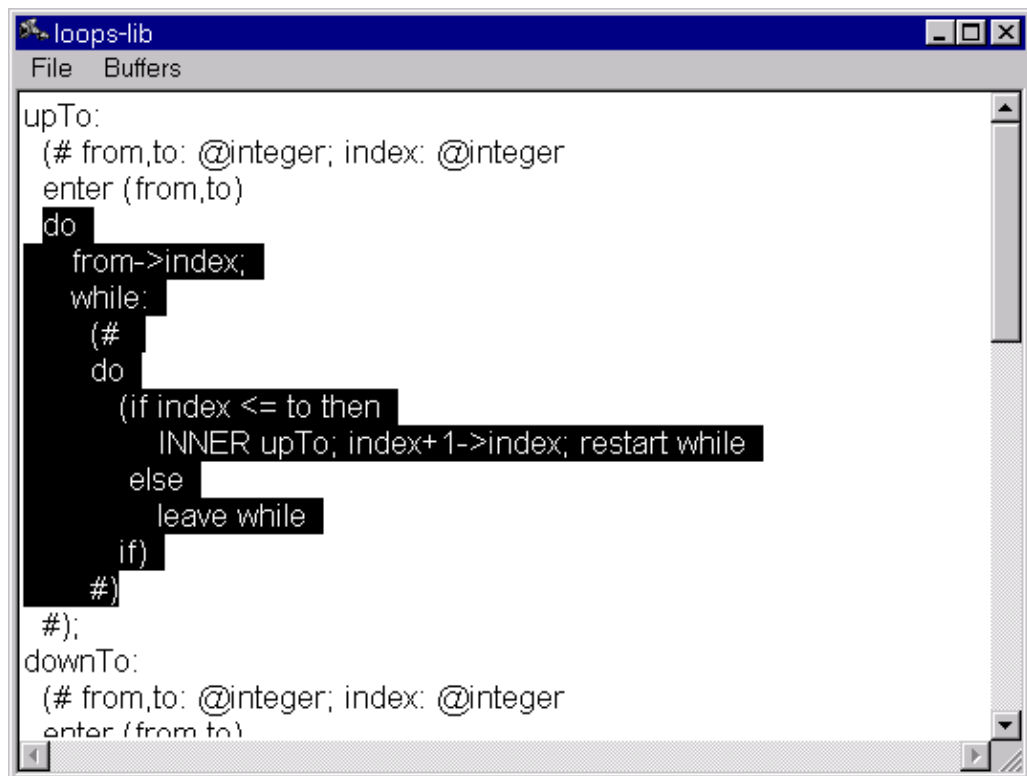
If you click the '+' next to the 'useloops' icon in the Projects window. vou will see that the program INCLUDEs the file 'loops' (symbolized by a single up–arrow:).

Click the line that says loops. We would like to save the do–parts in another file; our implementation file.

While viewing 'loops' choose 'SLOTs–>Create Implementation File...' and type `loopsbody`. Now you have created a new file 'loopsbody' whose ORIGIN is 'loops' (symbolized by a double up–arrow:). Conversely, 'loops' now has BODY 'loopsbody' (symbolized by a down arrow:).

Go back to 'loops' by clicking ⬆ `loops` in the Projects window.

Mark the do–part of `upTo` (press and hold the mouse button from the middle of 'do' to somewhere in the do–part).

Choose 'SLOTs–>Make DoPart SLOT', and name it `upToImplementation`. Now the implementation of `upTo` is automatically moved down into 'loopsbody'.

Double–click `<<SLOT upToImplementation: DoPart>>` in 'loops'. This takes you directly to the code in 'loopsbody' (or click the Definiton button 🔲 in the toolbar or press Alt+Right).

The Mjolner tool keeps track of your current implementation file at all times. This is the file into which the implementaion is moved when you choose 'SLOTs–>Make DoPart SLOT'. When you use 'SLOTs–>Create Implementation File...' to create a new file, that file becomes the current implementation file. If you wish to change it, use the SLOTs menu.

Move the do–parts of `downTo` and `stepTo` down into 'loopsbody' as well. This can be done in one step by selecting both `downTo` and `stepTo` and choosing 'SLOTs–>Hide Implementation'.

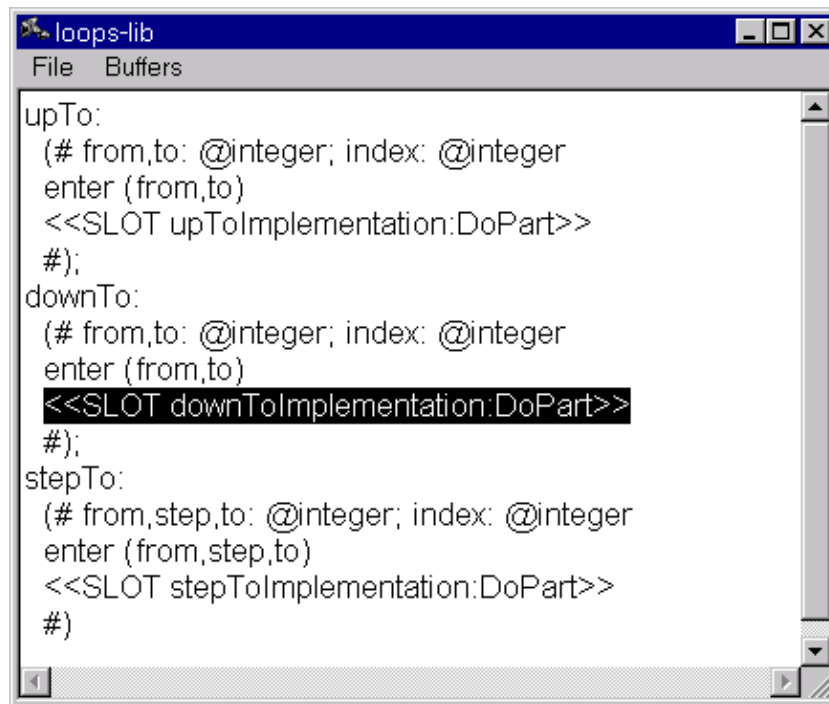## 1.4.2 Browsing through SLOTs and fragment forms

A SLOT is something of the form `<<SLOT blahblah: DoPart>>`. The corresponding fragment form contains a line such as `---blahblah: DoPart---`.

Using the Mjolner tool you can easily jump from a SLOT to the corresponding fragment form or vice versa.

While viewing 'loops', double–click the `stepTo` SLOT. This causes the implementation

of `stepTo` to be displayed. In the Fragment Forms window you can see that it contains three do–parts and that currently 'stepToImplementation' is displayed.

Jump to the implementation of `downTo` by clicking on the corresponding line in the Fragment Forms window. To find the SLOT mathing the 'downTo' fragment form, click the 'ToSlot' button in the toolbar or right click in the Form window and select 'Follow–>Link To SLOT'.

Now the tool searches through the chain of ORIGINs (adhering to the rules for doing so) looking for a matching SLOT. It ends up finding such a SLOT in 'loops'.

Three patterns are defined in 'loops': `upTo`, `downTo` and `stepTo`. To be precise, they are defined in a `lib:attributes` fragment form. as can be seen in the Fragment Forms window. But which slot will this fragment form be inserted into, i.e. where is the SLOT corresponding to `lib:attributes`? Find out the same way as above. Of course, the matching SLOT is located in 'betaenv'.

1.5 Exercise 5: Debugging

In this exercise we will take a brief look at how the Mjolner tool can help you locate errors in your programs.

Load the file syntaxedit.bet.

Compile ⬚ and run ⬚ the program. It contains a 'Reference is none'–error.

Click the Debug button ⬚ in the toolbar or right click in the code and choose 'Compile/Run–>Debug syntaxedit'. Now a debugging window pops up.

1.5.1 Reference is none

Click 'Go' in the debugging window. This time, when the program stops, the window tells you at the bottom that there is an error: 'Reference is none'. And in the Form window the command causing the error is marked (`'kg'->value`, which should be `'kg'->value[]`). Close the debugging window, correct the error and run the program again.

It is that easy to find and correct errors like 'Reference is none'!

1.5.2 Break points

For practice we will insert a breakpoint at `'The car is:'->putline`. Mark this line and right click. Choose 'Set Break' and mark something else (anything).

Click 'Go' in the debugging window. Now the program runs for a little while and then stops at the break point.

Click the Object button in the debugging window. This displays the active object, i.e. the object whose do–part is currently being executed. Here, the active object is of course the entire program.

You can see that the object has a component called `aPerson` which is an instance of the `program.Person` pattern. But we already knew that. Now double click that line. Double click `Name`, and double click `T`. You can see that the person is Santa Claus.

In this manner the object structure can be observed at runtime. Sometimes this is preferable to inserting debug–printlines.

Now click the break point `<<1>>`, then right click and choose 'Erase Break'. Choose 'Rerun' either from the right click menu or the new window. Now the program is ready to start over.

1.6 Appendix: Important commands and keyboard shortcuts

1.6.1 File operation commands

Operation	Keys	Description	Menu item
New BETA Program	Ctrl+N	Create a new BETA program	File –> New BETA Program
Open file	Ctrl+O	Open a BETA file	File –> Open
Save file	Ctrl+S	Saves the current open file on disk	File –> Save
Print	Ctrl+P	Print your code	File –> Print
Quit Mjolner Tool	Ctrl+Q	Of course, you wouldn't want to, but sometimes it is necessary to stop the fun.	File –> Quit

1.6.2 General navigation and editing commands

Operation	Keys	Mouse/Button	Description	Menu item
Re–prettyprint	F5		Redraws the code if something has gone awry	View –> Reprettyprint
Overview	Alt+Up		Abstract the code around the current selection	View –> Overview
Detail **Show definition**	Alt+Right	Double click or	This command has a number of different meanings	View –> Detail View –> Follow Semantic Link

Show code		Definition	depending on the current selection. Show more details around the current selection, find the definition of a name, open or close a comment	View–>Follow Link to SLOT
Abstract Recursively	Alt+Left		Abstract the current selection	View –> Abstract Recursively
Detail Recursively	Alt+Down		Detail the current selection until nothing remains abstract in it	View –> Detail Recursively
To slot	F3	To Slot	Search through the chain of ORIGINs looking for a SLOT matching the fragment form	View–> Follow Link to SLOT
Back	Ctrl+Left	Back	Go back to where you were before, e.g. before you pressed Show Definition and jumped to the definition	History–>Back
Forward	Ctrl+Right	Forward	The oppposite of Back	History–>Forward
Append empty code	Enter		Append "empty code" to the current selection	Edit–>Insert After
Prepend empty code	Ctrl+Enter		Prepend "empty code" to the current selection	Edit –> Insert Before
Paste before	Insert		Prepend the contents of the clipboard to the current selection	Edit–> Paste Before
Paste After	Ctrl+Insert		Append the contents of the clipboard to the current selection	Edit–> Paste After
Find	Ctrl+F		Open a dialog that lets you	Edit–>Find...

			search for a string in either the current selection, the current fragment form, a file or its "domain" or "extent".	
Replace	Ctrl+H		Like Find, but lets you replace the string by another one	Edit–>Replace...
Check current	F4		Check the current code for errors	Compile/Run–>Check Current
Check program	Ctrl+F4		Re–check the program that was last checked	Compile/Run–>Check Program
Compile current	F7		Compile the current code and (if it is a program) produce an executable file	Compile/Run–>Compile Current
Compile program	Ctrl+F7		Compile and create executable for the most recently selected program	Compile/Run–>Recompile
Run program	F8		Run the most recently selected program (if it has been compiled)	Compile/Run–>Run Program
Compile and run	Shift+F7		Compile and run the most recently selected program	Compile/Run–>Compile and Run Program
Debug the program	F9		Debug the most recently selected program (if it has been compiled)	Compile/Run–>Debug Program
Debug Executable	Ctrl+F9		Choose an executable to debug	Compile/Run–>Debug Executable
Get help	Ctrl+Alt+H		Open a browser with the Mjolner System Documentation	Help –> Web Documentation –> Mjolner System Documentation

1.6.3 Text editing commands

Command	Keyboard shortcut
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Undo	Ctrl+Z
Redo	Ctrl+Y

These commands are available in text editing mode as well as in syntax browsing mode.

1.6.4 Entering and leaving text editing mode

Command	Keys	Description	Menu item
Enter text editing mode	F2, Ctrl+Space	Enter text editing mode for the current selection	Edit−>Edit Text
Leave text editing mode	F2, Ctrl+Space	Re−parse the edited chunk of code. The code must be syntactically correct	Edit−>Parse Text
Cancel Text editing	Escape	Discards all changes to the edited chunk of code	Edit−>Cancel Textediting