RASMUS User's Manual

Kim S. Larsen

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C
Denmark

August 1992

Contents

1 Introduction

The manual is divided up into several parts describing the following:

- the window system
- the relation editor (XMODIFY)
- persistence
- the interpreter (XRASMUS)
- the editing facilities (also used in XRIKKE)

XRASMUS is intended to be run under the window system X. In chapter 2, we describe the window system's relationship to X. Chapter 3 contains a description of the relation editor, called XMODIFY. Chapter 4 is about persistence. We discuss how results from one session can be saved and used again in a later session. This is partly a discussion of how the XRASMUS system relates to the underlying UNIX operating system. In chapter 5, we describe the XRASMUS interpreter. Finally, chapter 6 details the general purpose editing facilities, which are also available in XRIKKE, the stand-alone text editor.

2 The Window System

To start XRASMUS, use the UNIX command `xrasmus` D_1 D_2 ... D_n, where the D_i's are paths of directories. Chapter 4 will explain about these directories. After typing `Return`, two windows will pop up. The larger of these is the XRASMUS interpreter which will be described in chapter 5. If you place the pointer inside this window, then you can type expressions which the interpreter evaluates. As a simple example, you can try to type $1 + 2$ `F1` in the lower part of the window. Notice that the interpreter evaluates this and responds with the answer 3. The respond could also be an error message if, for example, the text does not form a syntactically correct expression.

The other window is an icon, i.e., a closed window. Placing the pointer on the icon and clicking the middle mouse button will open the window. This window contains the relation editor, XMODIFY, to be described in great detail in chapter 3.

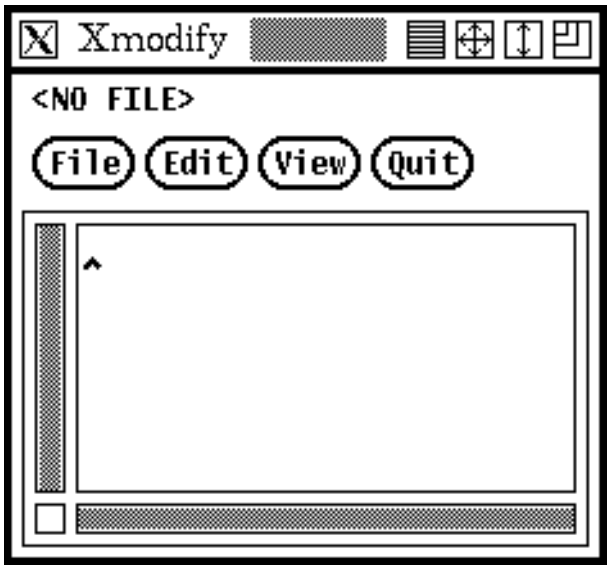

Figure 1: An empty XMODIFY window.

3 The Relation Editor

When you click the XMODIFY icon, an empty XMODIFY window will pop up; see figure 1. This window shares all the properties common to X-windows, i.e., it can be made active by moving the pointer into it, it can be resized, etc. To perform these actions, the mouse buttons are used in the manner standard to X-windows.

3.1 Starting a Modify Session

In the window of figure 1, you see four buttons: **File**, **Edit**, **View**, and **Quit**. If you click the **File** button, the menu shown in figure 2 will be presented to you. If you select the entry **Load**, you will be prompted for a file name. Writing the name of an existing file (with default extension `RAS_REL`) which contains a relation will have the effect of loading that relation into the XMODIFY window. This is illustrated on an example relation in figure 3. The relation has attribute names: Name, Age, Married, Code, and Address. As the relation is too large to fit entirely in the window, only a part of it is shown. The scroll bars (the left-most column and the bottom row) indicate which part of the relation is currently shown. The left scroll bar, for example, indicates that approximately the top half of the relation is shown in the window.

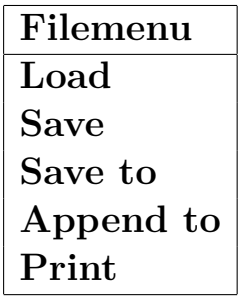

Figure 2: The File menu.

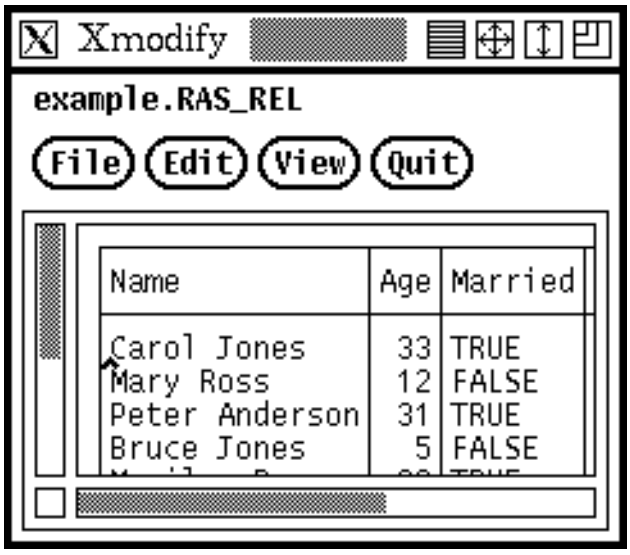

Figure 3: Initial appearance after loading.

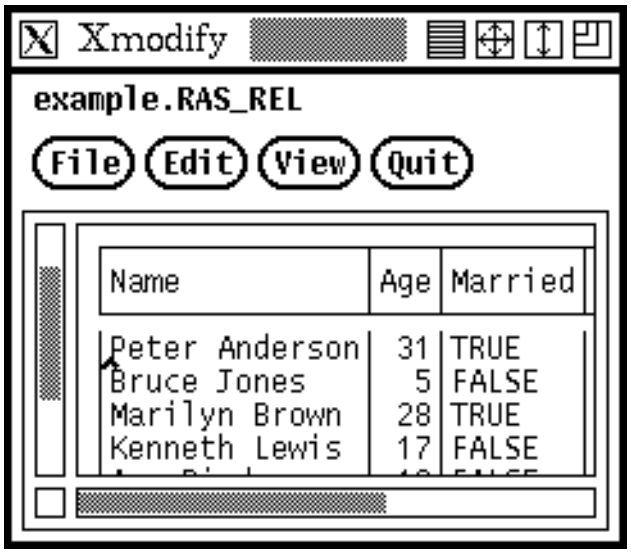

Figure 4: After clicking the left scroll bar.

Viewmenu
Ps
Setify

Figure 5: The View menu.

3.2 Navigation

As for X-windows, the scroll bars can be used for navigation. For example, clicking below the shaded area of the left scroll bar will bring you further down in the relation; see figure 4 and notice the effect the action had on the left scroll bar. You can use the bottom scroll bar to scroll horizontally. The arrows on the keyboard can also be used for navigation in the obvious way.

3.3 Views

Clicking the **View** button will give you the menu shown in figure 5.

If you select the **Ps** entry, then a **p**ermute/**s**ort window will pop up. An example of such a window is shown in figure 6. In this window, all the attribute names from the relation currently in the XMODIFY window are listed (twice). As can be seen on the left scroll bar, not all of the attribute names could fit in the window.

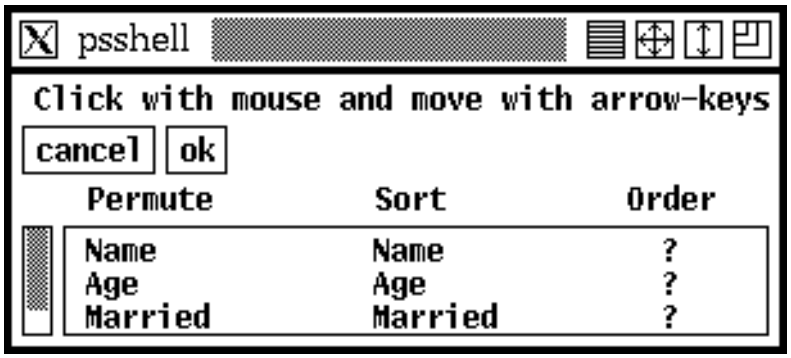

Figure 6: A permute/sort window for the example relation.

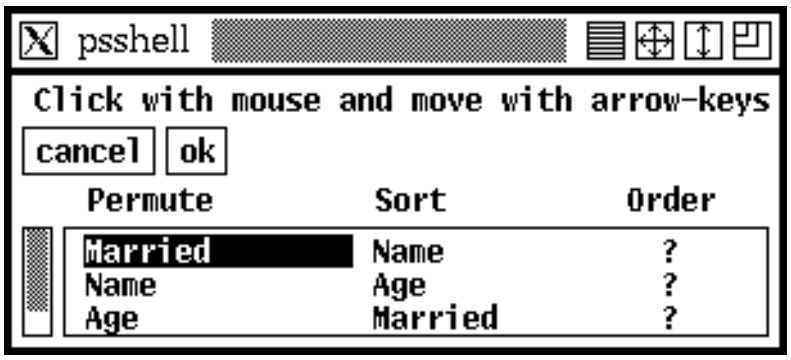

Figure 7: The permute/sort window after moving Married.

Now you can move attribute names around. For example, if you place the pointer on top of the attribute name Married in the Permute column, click the mouse button, and twice press the $\boxed{\uparrow}$ button, then Married will be moved to the indicated position; see figure 7. Pressing the $\boxed{\downarrow}$ button will similarly move a field down.

Clicking one of the Order fields containing a "?" will change the contents of the field to a "<". Clicking again changes it to a ">" and clicking one more time changes it back to "?". A "<" indicates that you want the column corresponding to the attribute name in the Sort column sorted in ascending order, while a ">" indicates descending order. A "?" means that you will let the system choose the order.

The attribute names in the Sort column can also be moved in exactly the same way as the attribute names in the Permute column. However, when an attribute name in the Sort column is moved, the corresponding Order field will move accordingly.

Let us assume that our actions have created the picture shown in figure 8. This window now indicates that we want our tuples sorted according to

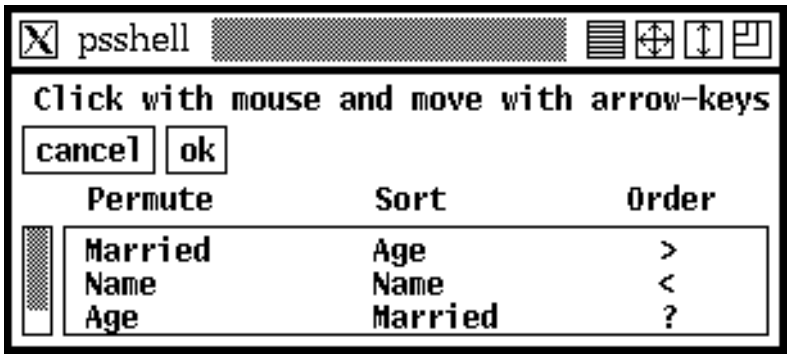

Figure 8: The permute/sort window with sortings indicated.

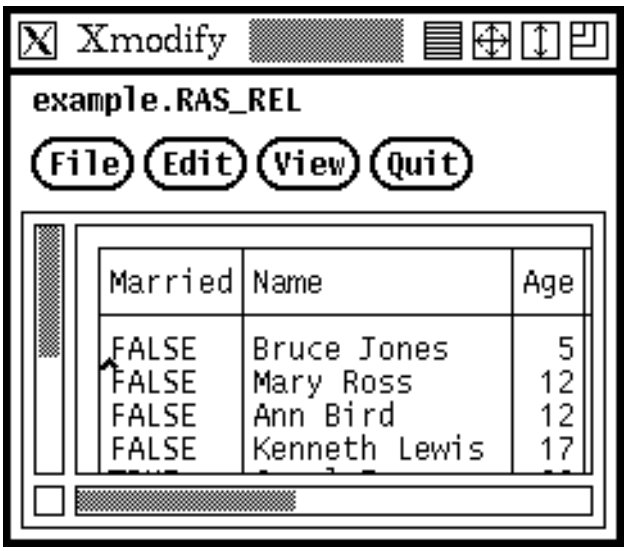

Figure 9: A permuted and sorted relation.

the Age field (in ascending order). However, if two or more tuples have the same Age value, then we want these tuples sorted according to the Name field.

If you click the **ok** button, then the permute/sort window will disappear, and the relation in the XMODIFY window will be changed according to the description you have just given; see figure 9. If you regret your actions, you can always edit things back to what they were, or you can simple click the **cancel** button and the permute/sort window will disappear leaving the relation unchanged.

If you select the **Setify** entry from the **View** menu, then the system will remove duplicate tuples from your relation.

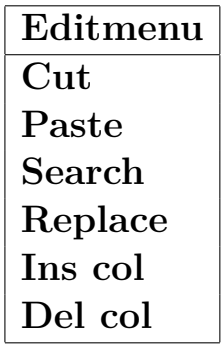

Figure 10: The Edit menu.

3.4 Editing

Every field in the relation can be edited; also the attribute names. However, the *types* of the columns cannot be changed. Using the mouse, the pointer can be moved to any field. A click on the mouse button will place the *cursor* at the position indicated by the pointer.

When the cursor has been placed in a field, hitting the carriage return has the effect of moving the cursor to the field immediately to the right of its current position. At the right-most field of a row, this action will have the effect of inserting a new tuple right after the one in which the cursor is placed. This new tuple will contain the standard value "?" in all fields.

You can edit the field in which the cursor is placed almost as you edit any file using the editing facilities described in chapter 6. If the field is a Text field, then you can insert any character by simply typing it, and you can delete characters using `Delete` on the keyboard. A carriage return cannot be inserted. In Int fields, you can only type digits (except that the character "−" can be placed at the left end of the number in the field) and in Bool fields, you can only place one of the characters "t" or "f", which expands into "true" or "false", respectively. Additionally, pressing the `F2` button will place the standard value in any field. A standard value is shown as a grey field.

If you click the **Edit** button, then the menu shown in figure 10 will pop up. The first four entries have the same effect in XMODIFY as they do in the general purpose text editor. There are just a few exceptions. First, searching and replacing is limited to fields, i.e., you cannot find an occurrence of a text which is spread out over several fields. Second, you can cut and

Quitmenu
Quit
Quit - no save

Figure 11: The Quit menu.

copy a whole field or part of a field. You can also cut and copy a number of tuples, but you cannot select, say, three fields of a four field tuple.

The last two entries, **Ins col** and **Del col**, are use to insert and delete columns. To insert a column, select the entry **Ins col**. You will then be prompted for an attribute name and a type. When you have responded, the new column is added to the relation. All the tuples currently in the relation will be extended with the standard value of the type specified in the new column. To delete a column, select the entry **Del col**. Then click on an attribute name and that column will disappear.

3.5 Ending a Modify Session

If you want to leave XMODIFY and close the window, then you click the **Quit** button. The menu depicted in figure 11 will pop up. If you select the **Quit** entry, then all the changes (if any) that you have made to the relation will be saved. If you have regretted all the changes you have made, then you click the **Quit - no save** button to close the XMODIFY window without making any changes to your relation.

If you want to save your changes without leaving XMODIFY, then click the **File** button to get the menu from figure 2. Selecting the **Save** entry will have the effect of saving the relation in the XMODIFY window on the file from which it was loaded, i.e., your original relation will not exist thereafter. You can also select the **Save to** button to save your changed relation on a new file. You will be prompted for a file name.

3.6 Creating a New Relation

If you try to load a nonexisting file, then you will not get an error message as you might expect. Instead, you will get a relation like the one depicted in figure 12. This is actually the totally empty relation, i.e., it has no

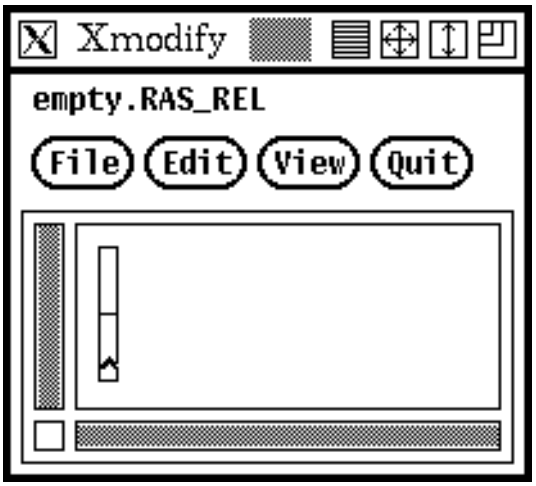

Figure 12: A totally empty relation.

attributes and it contains no tuples. You can now edit this relation and add columns as described earlier. Later you can work on the contents of the relation by adding tuples.

4 Persistence

One of the primary differences between an ordinary programming language and a database language is that the latter works on persistent data, i.e., we want to be able to start a session with the data we had when we ended the last session. We discuss this further in section 4.1. Another difference connected to the above is that a database should be able to deal with huge amounts of external data. In particular, a database language is required to include a specification of how external data can be read by the system. This is the subject of section 4.2.

4.1 Starting and Ending a Session

As described earlier, a session is started by typing `xrasmus` $D_1\ D_2\ \ldots D_n$, where the D_i's are directories. These directories contain XRASMUS definitions. For instance, if a directory contains a file named A and this file contains the number 47, then the name A will be associated with the value 47 in the XRASMUS interpreter. If different directories contain identical file names, then only the *last* definition is used. To be precise, if D_i and D_j contain identical file names and $i < j$, then the definition from D_j is

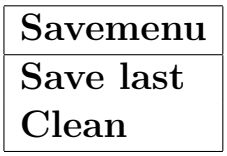

Figure 13: The save menu.

Figure 14: The window of the **Clean** entry.

used.

When starting a session, XRASMUS reads all the definitions from the specified directories and these can then be used in the interpreter. See chapter 5 for more details.

During a session, new variables can be defined and old variables can be redefined. How this is done is described in section 5. At the end of the session, the user may or may not want to save these changes. If you click the **Save** button of the interpreter window, you will see the menu depicted in figure 13. The entry **Save last** will be described in section 5. If you select the **Clean** entry, then a window will pop up; see figure 14. You can select the definitions you want deleted by clicking on the entries. If you click the field containing A and the field containing TheMostCarefulTest, then the window will change to reflect this; see figure 15. The fields become inverted to indicate that they are selected. Clicking an inverted field will undo that particular selection.

As it is indicated by the shaded area in the scroll bar, only some of the definitions fit in the window, and you can scroll down to access the remaining.

At any point, you can undo all of your actions by clicking **cancel**. If you

Figure 15: The example window with selections.

click **delete**, then the selected definitions will be deleted from the current directory.

4.2 External Relation Format

Data can be typed into a relation by hand as we have described it in chapter 3. However, sometimes large amounts of data is produced by other programs or have been collected by people over a large period of time and the question naturally arises: how can these data be read into a relation *automatically*? For this purpose, we describe the *external relation format*. This is the format on which relations are kept in the directories. So, obviously, if we can transform data to this format (automatically), then this data can be read into the system.

A relation on external format looks as it is illustrated in figure 16. Here, ⟨type⟩ can be either T, I, or B, representing the three atomic types. As an example, assume that the relation in figure 17 is so small that you can actually see it all in the window. Then this relation's external format would be as depicted in figure 18.

5 The Interpreter

As explained earlier, the largest window that pops up after you have typed `xrasmus` contains the interpreter; see figure 19. As you can see, the window consists of two separate parts. The purpose of the interpreter is to evaluate expressions typed by the user. In this chapter, we describe the facilities

⟨number of attribute names⟩
⟨type⟩ ⟨name⟩
.
.
⟨type⟩ ⟨name⟩
⟨field⟩
.
.
⟨field⟩

Figure 16: External relation format.

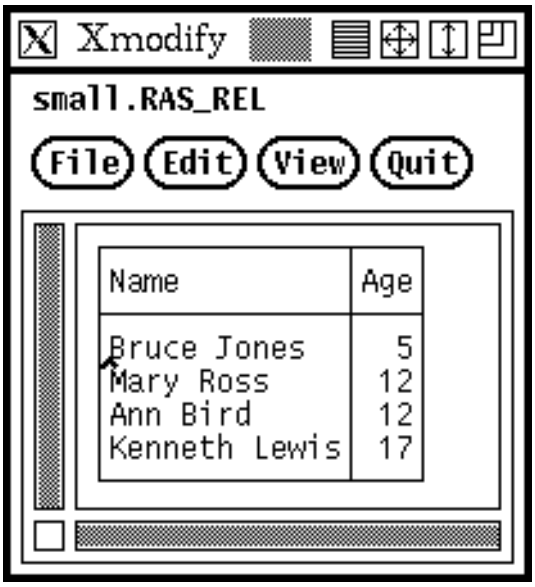

Figure 17: A small relation.

2
T Name
I Age
Bruce Jones
5
Mary Ross
12
Ann Bird
12
Kenneth Lewis
17

Figure 18: External relation format of example relation.

available for the interpreter and we define the set of syntactically legal expressions.

5.1 Interpreter Facilities

The window of figure 19 consists of an upper part and a lower part. These will in the following be referred to as the *passive* and the *active* part, respectively. Both parts will initially be empty. The passive part will always contain a complete transcript of the expressions which have evaluated along with the responses from the system. If you place the pointer inside this window, then you can type expressions in the active part; see figure 20. You cannot type in the passive part, but you can select text from the passive part and copy it to the active part or to files.

When typing in the active part, hitting `Return` will move the cursor to the next line. When you press `F1`, a number of things will happen. First, the expression you have just typed is moved to the passive part. The expression is then evaluated by the system and the value of the expression is printed immediately after your expression in the passive part. The active part is cleared such that you can type a new expression; see figure 21. Notice that a bold face font is used in the passive part to distinguish your expression from the answer which the system returned in response.

If there is a syntax error in the expression, then it will remain in the active

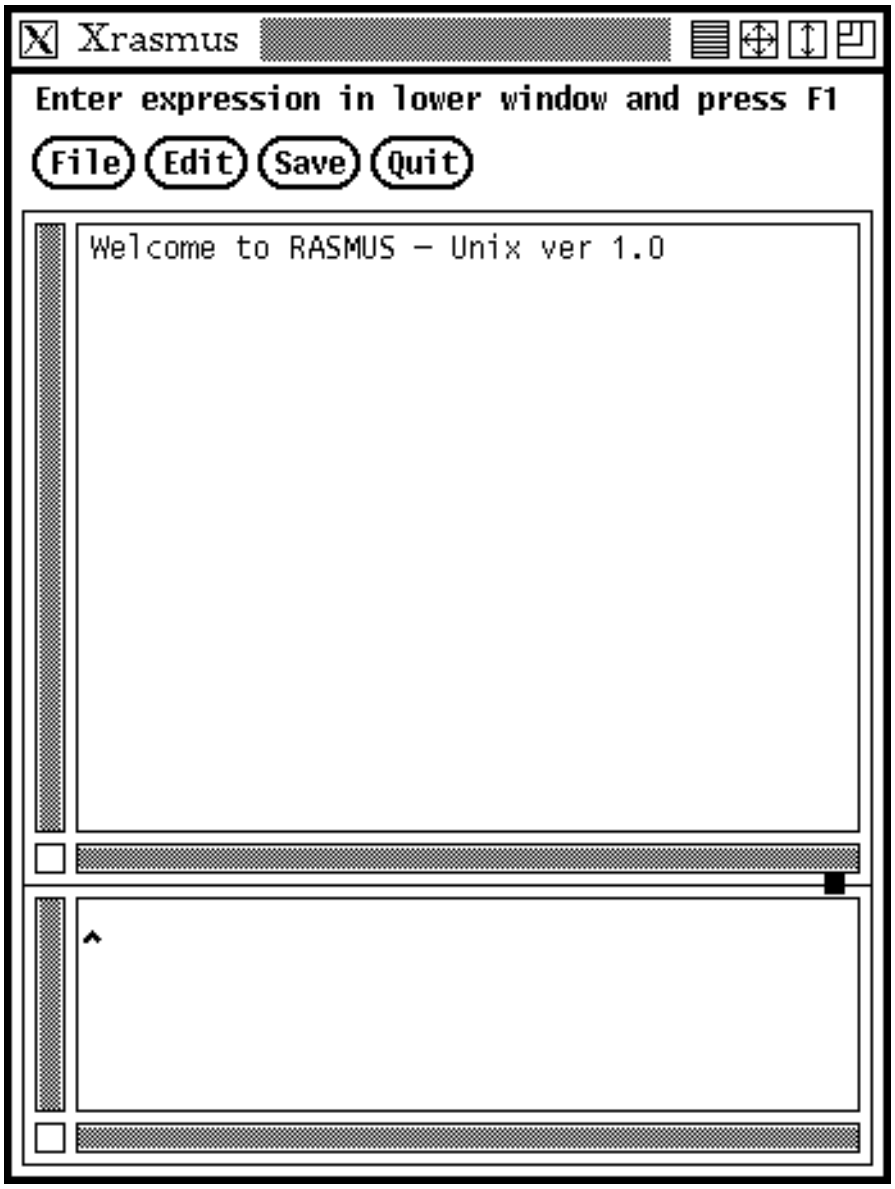

Figure 19: Initial appearance of interpreter.

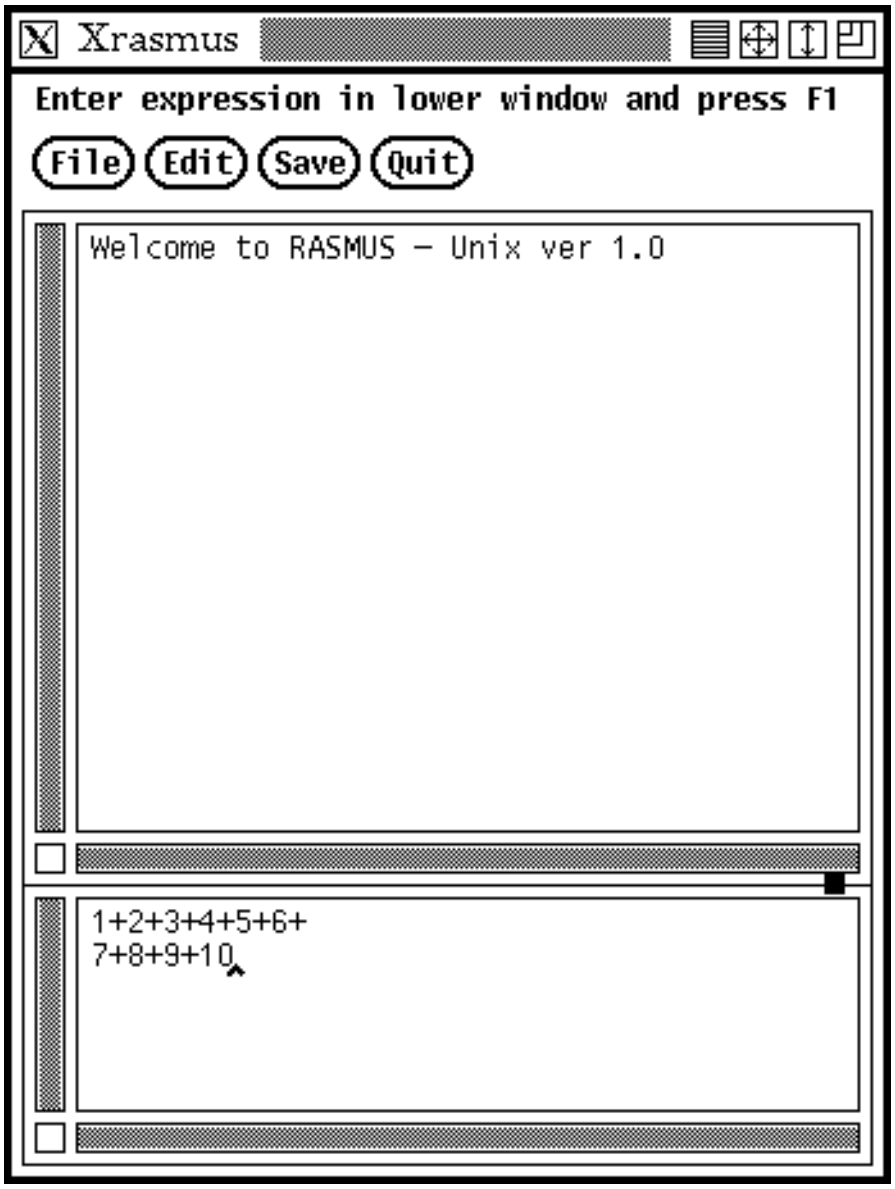

Figure 20: Typing the first expression.

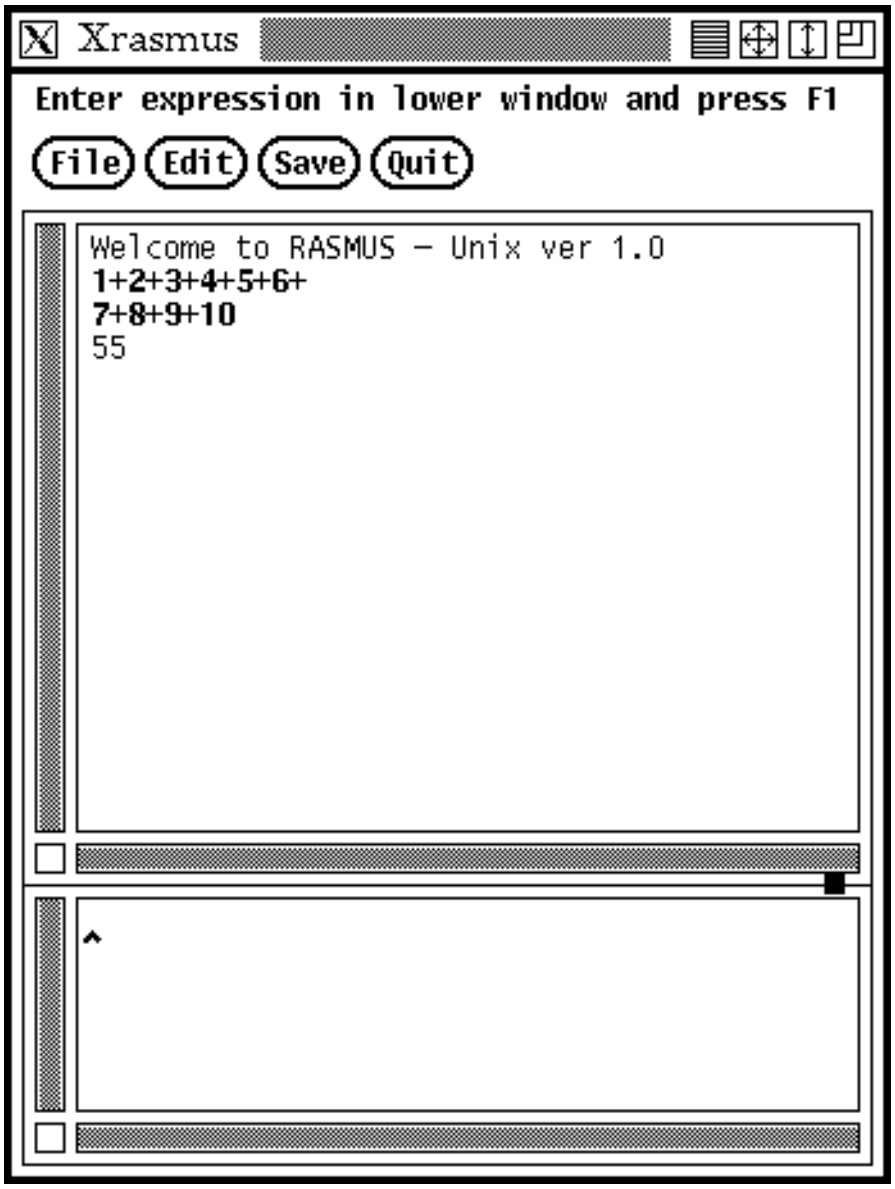

Figure 21: After one evaluation.

window and the cursor will be positioned at the error.

The active part can be used exactly like an editor. Thus, you can move the cursor around either by clicking the mouse or by using the keyboard arrows, you can delete and insert characters and lines, etc. At any time, if you press F1, then the interpreter will evaluate the expression in the active part, after which this expression along with the response will be a part of the passive part.

The interpreter window has four buttons; see figure 19. The **Save** button has been described partly in section 4. The remaining entry in this menu (see figure 13) is the **Save last** entry. By selecting that entry, you can always name the result of the last expression evaluated in the interpreter. If you click the **Quit** button, then you leave the XRASMUS system. The **Edit** button contains a menu identical to the ones from the text editor and XMODIFY. The entries in this menu can be used as described there, with the restriction that only the contents of the active area can be changed.

The last button is called **File**. Clicking this button will give you the menu shown in figure 2. If you select the entry **Print**, then a special window will pop up and allow you to print various parts of your file.

5.2 Expressions

In this section, we define the set of XRASMUS expressions. We do this in several steps. First, we give a grammar from which expressions must be generated. Afterwards, we give an informal semantics of XRASMUS expressions. In connection with this semantics, we restrict the set of expressions further by adding type constraints.

5.2.1 Grammar

In the following, we define some notation:

- **Category** * is a sequence of **Category**
- **Category**$^+$ is a nonempty sequence of **Category**
- **Category** $^{*\lambda}$ is a comma separated sequence of **Category**
- **Category**$^{+\lambda}$ is a nonempty comma separated sequence of **Category**

- **Category**$^{\circ}$ is either 0 or 1 of **Category**

In the following, a BNF grammar for XRASMUS expressions is presented.

Exp ::= **AtomConst** |
RelConst |
StandardConst |
`tup (` **NameExp**$^{*\lambda}$ `)` |
`rel (` **Exp** `)` |
`func (` **NameType**$^{*\lambda}$ `) -> (` **Type** `)`
Exp
`end` |
Name |
`#` |
`@ (` **Exp** `)` |
`not` **Exp** |
Exp `and` **Exp** |
Exp `or` **Exp** |
`-` **Exp** |
Exp `+` **Exp** |
Exp `-` **Exp** |
Exp `*` **Exp** |
Exp `/` **Exp** |
Exp `mod` **Exp** |
Exp `++` **Exp** |
Exp `(` **Exp** `..` **Exp** `)` |
`|` **Exp** `|` |
Exp `<<` **Exp** |
Exp `.` **Name** |
Exp `\` **Name** |
`has (` **Exp** `,` **Name** `)` |
Exp ProjSym Name$^{+\lambda}$ |
Exp `?` **Exp** |
Exp `[` **RenamePair**$^{+\lambda}$ `]` |
`!` `(` **Exp**$^{+\lambda}$ `)` **Restrict**$^{\circ}$ `:` **Exp** |
`max (` **Exp** `,` **Name** `)` |
`min (` **Exp** `,` **Name** `)` |
`count (` **Exp** `,` **Name** `)` |

		`add ( ` **Exp** `, ` **Name** `)` |
		`mult ( ` **Exp** `, ` **Name** `)` |
		Exp `=` **Exp** |
		Exp `<>` **Exp** |
		Exp `<` **Exp** |
		Exp `>` **Exp** |
		Exp `<=` **Exp** |
		Exp `>=` **Exp** |
		`if` **Guards** `fi` |
		`(+` **NameValue*** `in` **Exp** `+)` |
		Exp `(` **Exp**$^{*\lambda}$ `)` |
		`(` **Exp** `)` |
		IsType
AtomConst	::=	**BoolConst** | **IntConst** | **TextConst**
BoolConst	::=	`true` | `false`
IntConst	::=	**Digit**$^{+}$
Digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
TextConst	::=	`"` **Ascii*** `"`
Ascii	::=	**Letter** | **Digit** | **SpecialChar**
Letter	::=	`a` | `b` | $\cdots$ | `z` | `A` | `B` | $\cdots$ | `Z`
SpecialChar	::=	`!` | `@` | `#` | `$` | `%` | `^` | `&` | `*` | `(` | `)` | `-` |
		`_` | `+` | `=` | `|` | `\` | `~` | `‘` | `{` | `}` | `[` | `]` |
		`;` | `:` | `’` | `"` | `,` | `.` | $<$ | $>$ | `?` | `/`
NameType	::=	**Name** `:` **Type**
Name	::=	**Letter** **AlphaNum***
AlphaNum	::=	**Letter** | **Digit**

Type ::= `Bool` | `Int` | `Text` | `Atom` | `Tup` |
`Rel` | `Func` | `Any`

StandardConst ::= `?-Bool` | `?-Int` | `?-Text`

RelConst ::= `zero` | `one`

NameExp ::= **Name : Exp**

ProjSym ::= `|+` | `|-`

RenamePair ::= **Name** `<-` **Name**

Restrict ::= `|` **Name** $^{+\lambda}$

Guards ::= **Exp** `->` **Exp Choice** $^{\circ}$

Choice ::= `&` **Guards**

NameValue ::= `val` **Name** `=` **Exp**

IsType ::= `is-Bool (` **Exp** `) |`
`is-Int (` **Exp** `) |`
`is-Text (` **Exp** `) |`
`is-Atom (` **Exp** `) |`
`is-Tup (` **Exp** `) |`
`is-Rel (` **Exp** `) |`
`is-Func (` **Exp** `) |`
`is-Any (` **Exp** `) |`
`is-Bool (` **Exp** `,` **Name** `) |`
`is-Int (` **Exp** `,` **Name** `) |`
`is-Text (` **Exp** `,` **Name** `)`

5.2.2 Keywords

A **Name** cannot be one of the *keywords* listed in the following:

```
add     and     any      atom     bool     count
end     false   fi       func     has      if
in      int     is-Any   is-Atom  is-Bool  is-Func
is-Int  is-Rel  is-Text  is-Tup   max      min
mod     mult    not      one      or       rel
text    true    tup      val      zero
```

For these keywords, the system does not distinguish between upper- and lower-case letters.

5.2.3 Informal Semantics

We divide this section up into several subsections depending on the type of the expressions being discussed. Some operations involve more than one type, but are only discussed once. We try to place such operations under the main type involved.

First, we describe the *atomic* values: booleans, integers, and text.

Booleans

The constants `true` and `false` along with the operations `not`, `and`, and `or` have the standard interpretations. They require boolean arguments and they return a boolean value.

Values of the same type can be compared and the result of a comparison is a boolean.

Any two values can be compared using `=` or `<>`. Values of type `Bool`, `Int`, `Text`, `Tup`, and `Rel` can also be compared using `<`, `>`, `<=`, and `>=`. In following, we list the results of comparing types using the test `<`. The remaining test have the obvious complementary interpretation.

`Bool` `x<y` is true iff `x` is `false` and `y` is `true`.

`Int` `x<y` is true iff `x` is an integer less than `y`.

`Text` `x<y` is true iff `x` is a genuine prefix of `y`.

`Tup` `x<y` is true iff the set of names in `x` is strictly contained in the set of names in `y`. In addition, the intersection of names in `x` and `y` should have the same associated values.

`Rel` `x<y` is true iff the set of tuples in `x` is strictly contained in the set of tuples in `y`. An error occurs if the schemas of `x` and `y` are different.

Integers

The integer constants along with the operations `+`, `-`, `*`, `/`, and `mod` have the standard interpretations. Only `Int` values in the range -2147483647 to 2147483647 are available. A system error will occur if operations evaluate to values outside that range.

The operations listed above require integer arguments and they return an integer value. We point out that the value of `x/y` is the integer part of `x` divided by `y` and `x mod y` is the remainder of the same operation.

The expression `-i` gives the same value as `0-i`.

Text

A `Text` is a sequence of characters. A constant text is written as a sequence of characters surrounded by double quotes, i.e., like `"Rasmus"`.

- `t1++t2` denotes the concatenation of the texts `t1` and `t2`.
- `t(i..j)` denotes the subsequence from `t` starting with the `i`th character and ending with the `(j` $-$ `1)`th character, where `i` and `j` are integers. A character sequence of length n is numbered from 0 to $n-1$.
- `|t|` denotes the length of the text `t`. For example, any text `t` is equal to `t(0..|t|)`.

Standard values

The atomic types have *standard values*. These are written `?-Bool`, `?-Int`, and `?-Text`. These values are outside the orderings. This means, for example, that if `i` is not the standard value `?-Int`, then the expressions `i<?-Int`, `i=?-Int`, and `i>?-Int` will all evaluate to `false`.

Operators cannot be applied to standard values. This means that if expressions like `5+?-Int`, for example, are evaluated, then an error will occur.

Tuples

A tuple is a set of pairs, where each pair consists of an attribute name and an atomic value. If `A1`,...,`An` are attribute names and `v1`,...,`vn` are atomic values, then a tuple can be specified as

```
tup(A1:v1,...,An:vn)
```

We have the following operations on tuples.

- `t1<<t2` denotes the tuple `t1` updated with the tuple `t2`. The expression `t1<<t2` basically evaluates to the union of `t1` and `t2` except that whenever an attribute name appears in both `t1` and `t2`, only the attribute name and corresponding value from `t2` is used. If the same attribute name appears in both arguments, but the values are of different types, then an error occurs.
- `t.A` denotes the value associated with the attribute name `A` in `t`. If `A` does not appear in `t`, then an error will occur.
- `t\A` denotes the tuple `t` except that the attribute name `A` and its associated value is left out. If `A` does not appear in `t`, then an error will occur.
- `has(t,A)` denotes the boolean value `true` if the attribute name `A` appears in `t`. If not, then the value `false` is returned.

Relations

A relation is a set of tuples such that every tuple contains the same set of attribute names and such that for any two tuples, the values associated with identical attribute names are of the same type. This set of attribute names with their associated types is called the *schema* of the relation.

If `t` is a tuple, then `rel(t)` is a relation with one tuple `t`.

There are the following operations on relations.

- `|r|` denotes the length of the relation `r`, i.e., the number of tuples in `r`.
- `has(r,A)` denotes the boolean value `true` if the attribute name `A` appears in the schema of `r`. If not, then the value `false` is returned.
- `r1+r2` denotes the union of the tuples in `r1` and `r2`. If `r1` and `r2` does not have the same schema, then an error will occur.

- `r1-r2` denotes the set difference of `r1` and `r2`, i.e., `r1-r2` is the set of tuples from `r1` which do not belong to `r2`. If `r1` and `r2` does not have the same schema, then an error will occur.

- `r1*r2` denotes the join of `r1` and `r2`. The attribute names appearing in both schemas are required to be of the same type. Otherwise an error will occur. The relation `r1*r2` consists of all the tuples `t` such that `t` restricted to the schema of `r1` belongs to `r1` and such that `t` restricted to the schema of `r2` belongs to `r2`.

- `r1 |+ A1,...,An` denotes the projection of `r` onto the attributes `A1`,...,`An`. These attributes have to belong to the schema of `r1`. The relation consists of all the tuples from `r1` restricted to the attributes `A1`,...,`An`.

  `r1 |- A1,...,An` denotes the projection of `r` onto the attributes in the schema of `r` *except* the attributes `A1`,...,`An`.

- `r? b`, where `b` is a boolean expression, contains the tuples `t` from `r` which make `b` true when the special symbol `#` is replaced with `t`.

- `r[A1<-B1,...,An<-Bn]` denotes a renaming of the attributes in `r`. The attribute names `A1`,...,`An` are changed to `B1`,...,`Bn`, respectively. An error occurs if the attributes `A1`,...,`An` do not belong to the schema of `r`. The `A`i's must be pairwise different. Also, the `B`i's must be pairwise different and no `B`i is allowed to belong the schema of `r` minus `A1`,...,`An`.

- `!(r1,...,rn)|X: exp` denotes a factor expression. The result of a factor expression is the union the evaluation of a family of expressions to be described in the following. These must all evaluate to relations with the same schema. Otherwise an error occurs.

  The family of expressions is constructed by taking `exp` and substituting `#`, `@(1)`,...,`@(n)` with different tuple and relation values. The values for `#`, `@(1)`,...,`@(n)` are determined as follows. `X` must be a comma separated list of the attributes in the intersection of the schemas of the relations `r1` through `rn`. The result of evaluating `(r1 |+ X) + ...+ (rn |+ X)` is called the *base relation*. The symbol `#` is instantiated with the tuples in the base relation; one at a time. Now assume that `#` has been given a value, then `@(`i`)` is the relation

`(rel(#)*r`i`) |- X`. If `|X` is not specified, then it is assumed to be all the common attributes of the `r`i's.

If a list of attributes is specified (a restriction list), then `X` is this list. An error occurs if `X` is not contained in the intersection of the schemas of the n relational arguments.

- `zero` denotes the empty relation with the empty schema.
- `one` denotes the relation with the empty schema containing one tuple: the empty tuple.

Aggregation

The five operations `max`, `min`, `count`, `add`, and `mult` are very similar. They are all used like `max(r,A)`, where `r` is a relation and `A` an attribute name. They perform the action indicated by their name, e.g., `max(r,A)` returns the maximal value in the `A` column of the relation `r`. An error occurs if the relation does not have an `A` column. The standard values are always ignored. If the `A` column contains nothing but standard values, or if the relation is empty, then `max` and `min` returns the standard value, `count` and `add` returns `0`, and `mult` returns `1`.

Miscellaneous

- `if b1->exp1 & ...& bn->expn fi` is the *conditional expression*. The `b`i's are boolean expression. This conditional expression is evaluated as follows. The boolean expressions are evaluated in order until one is found which gives `true`. If none of the boolean expressions evaluate to `true`, then an error occurs. If `b`i was the first expression evaluating to `true`, then the result of the conditional expressions is the result of evaluating `exp`i.
- `(+ val x1=exp1 ...val xn=expn in exp +)` is a *block*. The expressions `exp1` through `expn` are evaluated in order and the results are named `x1` through `xn`, respectively. Then `exp` is evaluated and returned as the result of the block. Note that the values are named as soon as they are evaluated, so the names `x1` through `x`$(i-1)$ can be used in `exp`i, and all the names can be used in `exp`.

- `func (x1:T1,...,xn:Tn) -> (T) exp end` is a *function definition* and denotes a function. The names `x1` through `xn` are the *formal parameters* and `T`, `T1`,...,`Tn` are types; `T` is called the *result type*.

- `f(exp1,...,expn)` is a *function application*. The function `f` must be defined in the environment. The expressions `exp1` through `expn` are evaluated in order, and the values are bound to the formal parameters of the function definition. An error occurs if the number of expressions does not equal the number of formal parameters. An error also occurs if an expression evaluates to a value of type different from the one specified in the function definition. The body of the function definition is now evaluated and it can depend on the formal parameters. The result of the function application is the value of the function body unless this value is not of the type specified in the function definition (the result type).

- `(exp)` denotes whatever the expression `exp` denotes.

- `is-Bool(exp)` denotes `true` if `exp` evaluates to a boolean. Otherwise, it denotes `false`. The constructions `is-Int`, `is-Text`, `is-Atom`, `is-Tup`, `is-Rel`, `is-Func`, and `is-Any` are defined similarly.

  For the atomic types, there is an additional construction. If `exp` evaluates to a relation and `A` is an attribute name of that relation, then `is-Bool(exp,A)` denotes `true` if `A` is of type `Bool` and `false` otherwise. If either `exp` does not evaluate to a relation or `exp` evaluates to a relation and `A` does not belong to the schema of that relation, then a runtime error occurs. The expressions `is-Int(exp,A)` and `is-Text(exp,A)` have similar semantics.

6 Editing Facilities

This chapter describes the editing facilities that are common to XMODIFY and XRASMUS. When invoked as XRIKKE this also serves as a general purpose text editor.

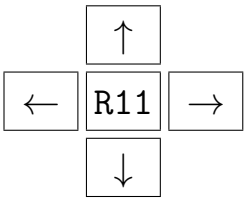

Figure 22: Keyboard arrows.

6.1 Navigation

In a vindow the text height and width was unlimited. This implies that sometimes not all the text will fit in the window. As you cannot place the cursor at a position not shown, there has to be a way of changing ones view of the text.

This can be done by using the scroll bars. If you position the pointer in the left scroll bar and click the right-most mouse button, then you will move down in the text. Clicking the left-most mouse button will bring you up. Your actions will be reflected by the scroll bar. The grey area indicates which part of the text you are currently viewing. The bottom scroll bar works similarly, but in the horizontal direction.

To the right on the keyboard, there are four arrows arranged like it is depicted in figure 22. The arrows can be used directly to move the cursor in any of the four directions. The key `R11` can be viewed as an "accelerator" key. If it is pressed once or twice before an arrow key is pressed, then the cursor will be moved further. In figure 23, there is a complete listing. In figure 23, a *page* is the number of lines which exactly fit in the window.

6.2 Files

After having typed some text, you would probably want to save it for later use. You do this as follows. Move the pointer to the **File** button and click on the mouse. Then the file menu will pop up; see figure 2. Click the **Save to** entry. You will now be prompted for a file name, i.e., a very small window with room for only one line will appear on the screen. You type a file name and press `Return`. You have now saved the text on that file. The editing process can continue after this action.

Later, if you want to get hold of that text again, then you select the **Load**

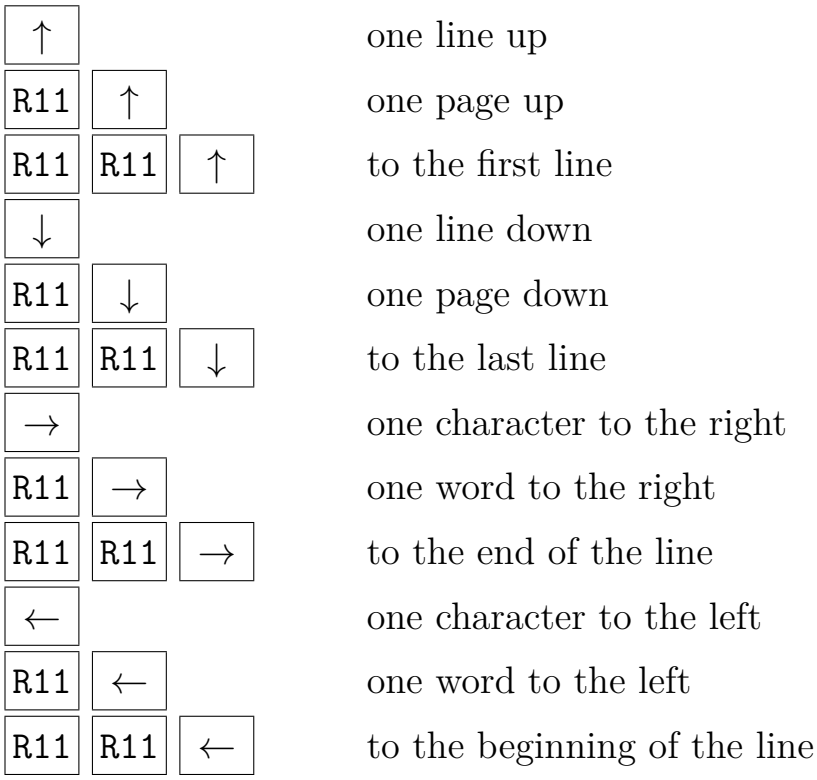

Figure 23: Moving the cursor.

entry. Again, you will be prompted for a file name and if you type the one on which you previously saved your text, then that text will be brought into the editor. The text, if any, which was in the editor when you selected the **Load** entry will first be saved.

Finally, if you load a text from a file, change it, and want to save it on the *same* file from which you loaded it, then you simply select the **Save** entry. Whatever was previously on that file will be lost.

6.3 Edit

Clicking the **Edit** button will make the edit menu appear; see figure 10. Single characters are inserted directly from the keyboard and deleted with the `Delete` button. We now focus on larger pieces of text and on how such pieces of text can be deleted, copied, or moved easily.

To select a piece of text, you position the pointer on top of the first character in the text you want to select. Then you press the left-most mouse button. While holding the button down, you move the pointer to the last

character in the text you want to select, and there you let go. Notice that the selected text has been inverted on the screen. If the text has been selected incorrectly, then you can start all over, or you can *extend* the selection by pressing the right-most mouse button and move the pointer like described above. This can be practical when you want to select more text than fits in the window. The system will remember a selection until you make another, even after the inverted text on the screen looks normal again.

Selected text can be deleted. This is done by selecting the **Cut** entry in the edit menu. Text can also be copied to other locations in the file. To do this, move the cursor to the position where you want the text inserted. Then select the **Paste** entry from the edit menu (clicking the middle mouse button will have the same effect). Text can be moved by first cutting and then pasting.

The editor provides facilities for searching for a text and for making simple systematic changes.

If you select the **Search** entry from the edit menu, then you will be prompted for a (one line) text. Enter a text and press `Return`. The system will now find the first occurrence of that text appearing after the position of the cursor. The cursor will be moved to this position. You will be notified if the text did not appear in the file below the current cursor position.

If you select the **Replace** entry, then you will be prompted for two texts: a search text and a replace text. Enter the two texts, each of them followed by a `Return`. The purpose of the replace command is to replace occurrences of the search text with the replace text. When an occurrence of the text has been found, a window with four buttons will pop up: **Replace and find next**, **Find next**, **Replace all**, and **Quit replace**. Now, click on the button with the desired action.

The displayed font size can be changed by pressing the `CTRL` button and simultaneously clicking the rightmost mouse button.

List of Figures