Contents

1 Introduction

This manual describes how to use the RASMUS system together with the EMACS editor. It also describes how to obtain *persistence*, i.e. how to save results between sessions. Finally, it describes what RASMUS expressions look like, and what they mean. Basic familiarity with EMACS is assumed, cf. the TRINE manual.

2 The Interpreter

The purpose of the RASMUS interpreter is to evaluate expressions typed by the user. This chapter, describes the RASMUS interpreter.

2.1 Getting started

To start RASMUS from within EMACS, use the `Modes` menu. It has two entries relating to RASMUS:

- **Rasmus**
- **Rasmus...**

The first entry will open up a new frame called `RASMUS`, for the communication with the RASMUS interpreter. The second entry will start by prompting for directories, before opening up the new frame, but is otherwise identical to the first. The second entry will be further explained in chapter 3.

The RASMUS frame contains a new menu, `Rasmus` menu. Its content looks something like figure 1. The explanation of `Evaluate` is described in the following, while the explanation of `Save last...` , `Change name...` and `Clean` are deferred to chapter 3. The `Modify` entry will be described in a separate note.

2.2 Getting along

The RASMUS frame consists of two separate parts (cf. figure 2). These parts are in the terminology of EMACS called *windows* (not to be confused with X-windows). Each window has its own status-line. The top window is called the *output* window, or the *passive* part. The bottom window is called the *input* window or the *active* part. Input to the interpreter is typed into the active window, and the interpreter will respond in the passive window.

In fact, the active and passive windows are ordinary EMACS buffers, which operate in a special RASMUS major mode. This means that anything you can do with a buffer, you can also do with the RASMUS windows, including printing and saving.

In order to type in an expression for the interpreter to evaluate, the input buffer must be made active. This is done either via the `Buffers` menu, or by placing the cursor in the input window and pressing the left mousebutton (you may also place the cursor in the passive window, but the system will not allow you to alter the buffer it displays).

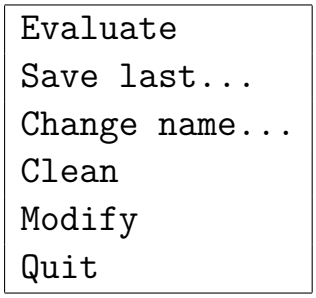

Figure 1: The Rasmus menu.

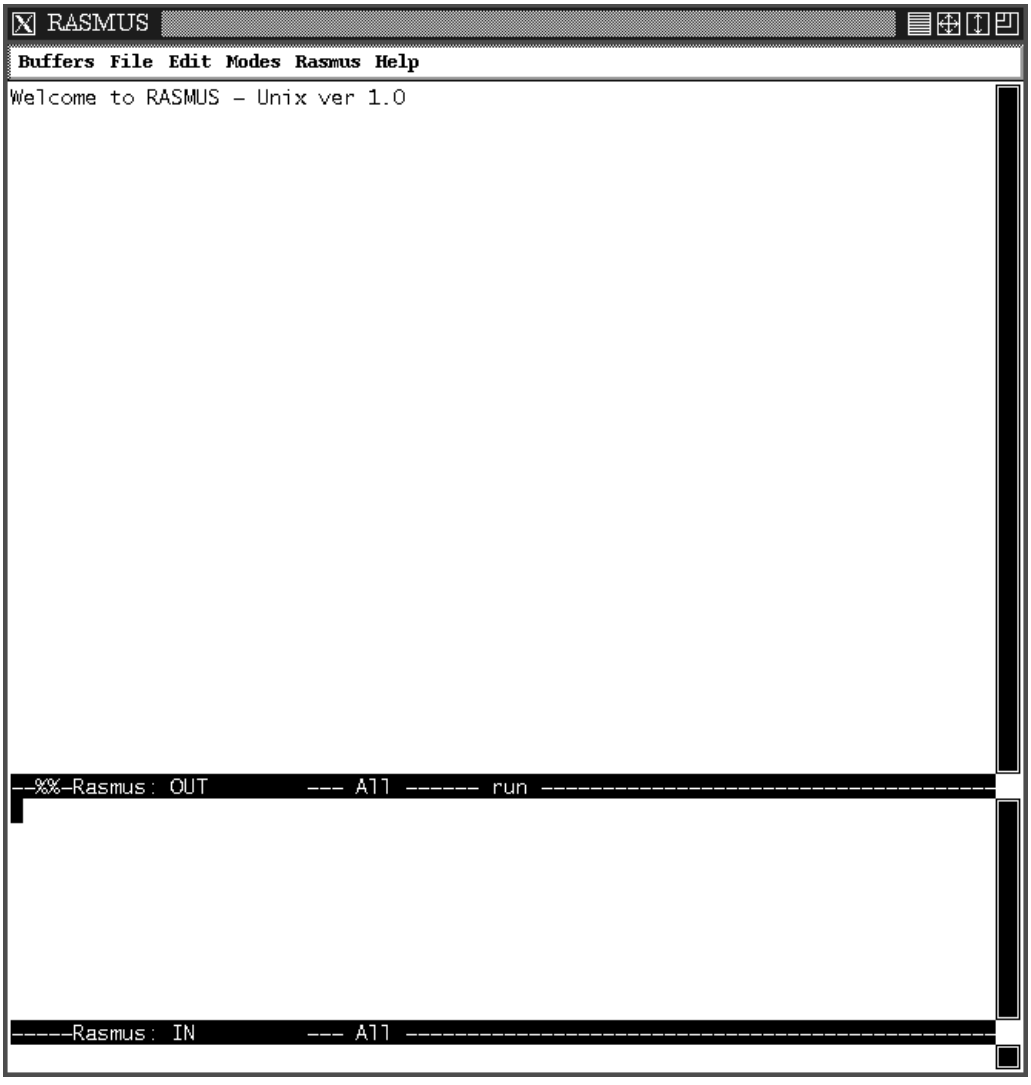

Figure 2: Initial appearance of interpreter.

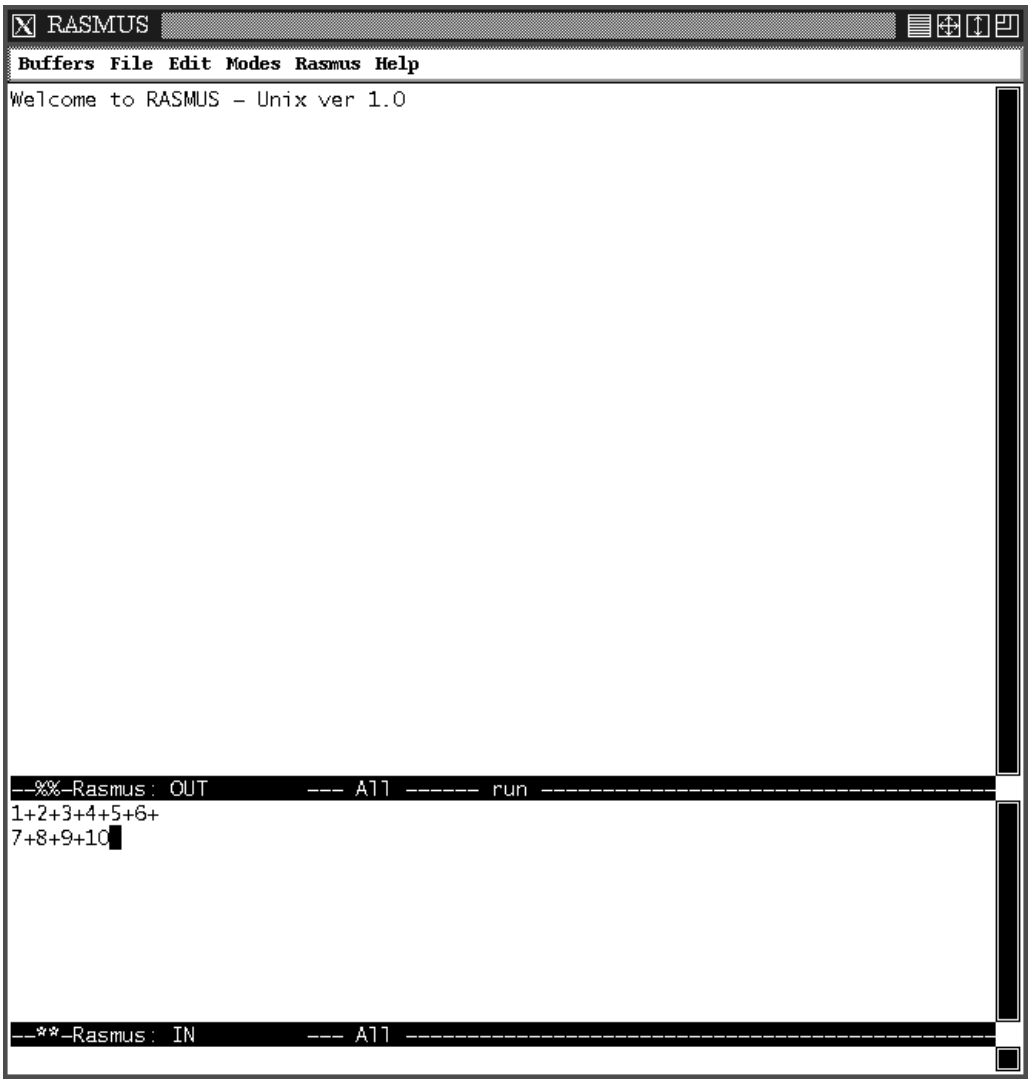

Figure 3: Typing the first expression.

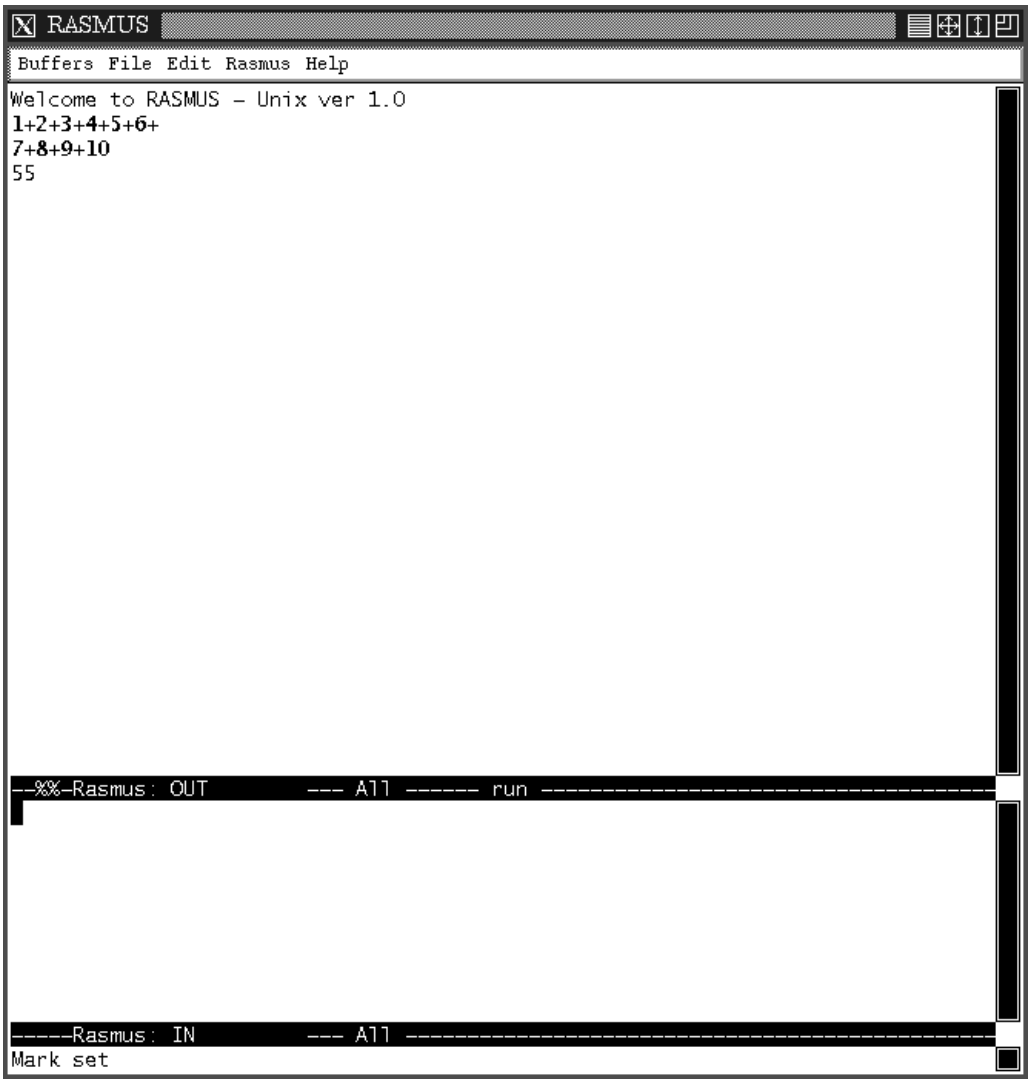

Figure 4: After the evaluation.

Suppose you have typed in the following expression:

```
1+2+3+4+5+6+
7+8+9+10
```

Your RASMUS frame will now look as in figure 3. In order to have the interpreter evaluate the expression, you must use the Rasmus menu. If you select Evaluate, your frame will end up looking like figure 4. The expression has been copied from the active window into the passive, using a different font, so that you may distinguish your input from the output of the interpreter. The interpreter evaluates the expression, and inserts the result into the output buffer. The active window is then cleared.

If you have typed in an illegal expression, the active part is not cleared, which means that you can edit the expression. What constitutes legal expressions is further described in chapter 4.

To get rid of the interpreter, you can use the Quit entry of the Rasmus menu. This will delete the RASMUS frame and the corresponding input and output buffers.

3 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 3.1. Another difference 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 3.2.

3.1 Starting and Ending a Session

As described in the beginning, RASMUS is started using the `Rasmus` or `Rasmus...` entries of the `Modes` menu. The second form allows optional directories to be supplied to the RASMUS session to be started. The current directory is always implictly included. So the first form really corresponds to the second, with the current directory as the only argument (or without any arguments at all, the current directory is still included!).

These directories contain RASMUS 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 RASMUS interpreter. If different directories contain identical file names, 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 used. The current directory is always included, and it will always appear as the last directory.

You will be prompted in the minibuffer for directories, one at a time, until you type `Return` on an empty line. Each directory must exist.

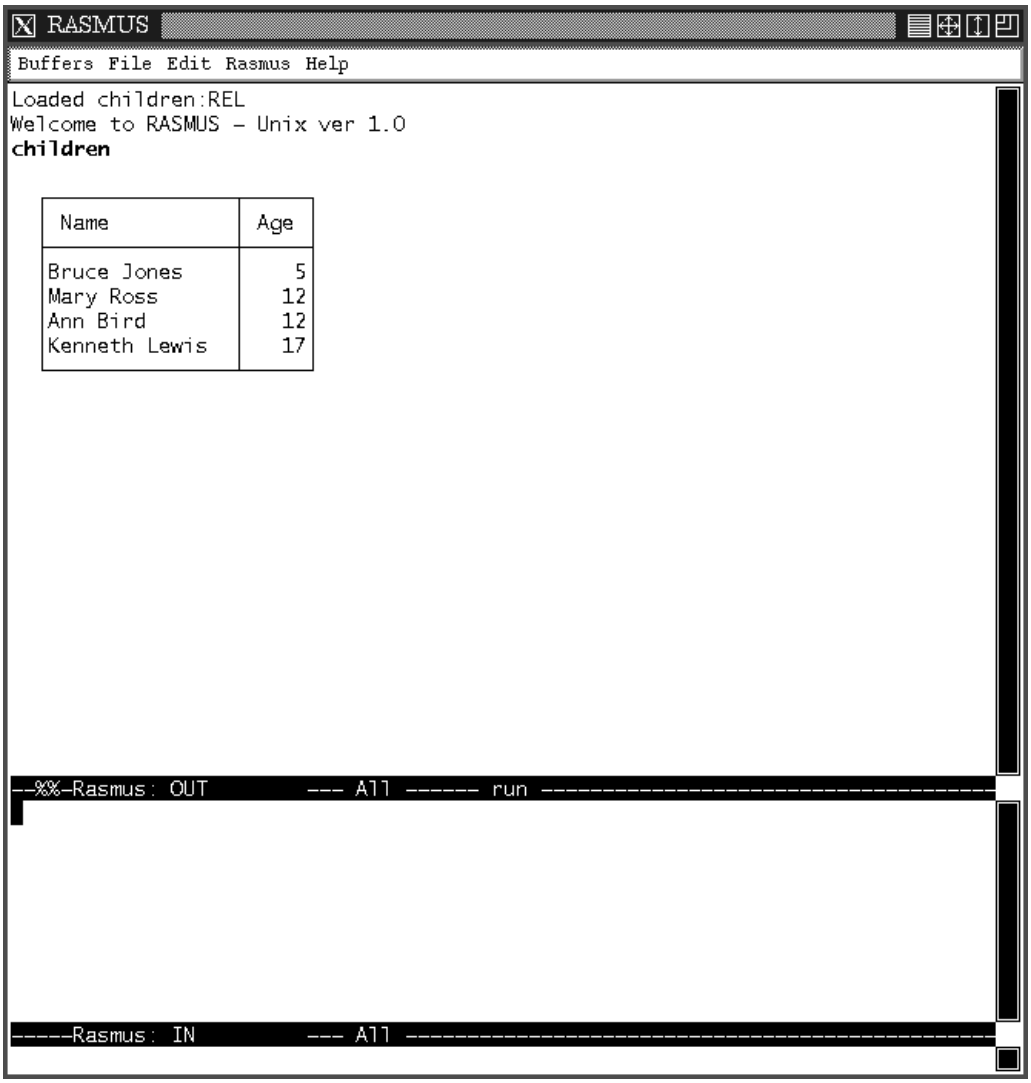

Figure 5: A small relation.

When starting a session, RASMUS establishes the definitions represented by the directories. As an example, suppose we have started RASMUS in a directory where there is a relation called `children`. After evaluating `children`, the display will look as in figure 5.

The `Save last...` entry of the `Rasmus` menu can be used to name and save the result of the last evaluation. EMACS will prompt for the name in the minibuffer, and the result will appear as a file in the current directory.

You can only save the last result evaluated, but you can do it as often as you please. You can even save the same result under different names.

If you want to save the result of earlier evaluated expression, you must reevaluate it. Remember that the expression is listed somewhere in the passive window, so you can use `Copy` and `Paste` from the `Edit` menu.

The `Change name...` option will allow you to rename a definition. It will prompt for the old and new names. The new name will persist for future sessions.

The [Clean] entry will bring up a list of the currently known definitions, allowing you to permanently delete any of them. The active buffer will be replaced by the `*Clean*` buffer, where all names are listed. By placing the cursor over one of the names and pressing [D], that defintion is *flagged* for deletion, as marked by the `D` appearing to the left of the name. Pressing [x] will *execute* the deletions, by removing the files corresponding to the flagged names. Pressing [q] will remove the `*Clean*` buffer.

> **Note that a relation is irretrieveably lost when it is cleaned.**

3.2 External Relation Format

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 such data be read into a relation *automatically*? For this purpose, we describe the *external relation format*. This is the format in 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 in external format looks as illustrated in figure 6. Here, ⟨type⟩

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

Figure 6: External relation format.

can be either T, I, or B, representing the three atomic types TEXT, INTEGER and BOOLEAN.

The external format of the `children` relation would be as depicted in figure 7.

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

Figure 7: External relation format of example relation.

Relations are stored in external format as ordinary text files. The text file is named after the relation name, by adding an extension of `.RAS_REL`. So our example relation was stored in a file `children.RAS_REL` with the exact contents of figure 7. This file can be loaded into EMACS as any other file and edited as required.

This also gives an alternative for the `Change name...` command, as it corresponds to renaming the file that corresponds to the definition, except that this would only take effect for the next session.

A better way of manipulating relations, will be accessible through the `Modify` option of the `Rasmus` menu. This will however be the subject of a separate note, as mentioned in the beginning.

4 Expressions

In this chapter, we define the set of RASMUS 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 RASMUS expressions. In connection with this semantics, we restrict the set of expressions further by adding type constraints.

4.1 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 grammar for RASMUS expressions is presented.

Exp ::= **AtomConst** |
RelConst |
StandardConst |
`tup (` **NameExp** $^{*\lambda}$ `)` |
`rel (` **Exp** `)` |
`func (` **NameType** $^{*\lambda}$ `) -> (` **Type** `)`
Exp
`end` |
Name |
Name `:=` **Exp** |
`#` |

`@ (` **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 `«` **Exp** `|`
Exp `.` **Name** `|`
Exp `\` **Name** `|`
`has (` **Exp** `,` **Name** `) |`
Exp **ProjSym** **Name** $^{+\lambda}$ `|`
Exp `?` **Exp** `|`
Exp `[` **RenamePair** $^{+\lambda}$ `] |`
`!` `(` **Exp** $^{+\lambda}$ `)` **Restrict** $^{\circ}$ `:` **Exp** `|`
`!<` `(` **Exp** $^{+\lambda}$ `)` **Restrict** $^{\circ}$ `:` **Exp** `|`
`!>` `(` **Exp** $^{+\lambda}$ `)` **Restrict** $^{\circ}$ `:` **Exp** `|`
`max (` **Exp** `,` **Name** `) |`
`min (` **Exp** `,` **Name** `) |`
`count (` **Exp** `,` **Name** `) |`
`add (` **Exp** `,` **Name** `) |`
`mult (` **Exp** `,` **Name** `) |`
`days (` **Exp** `,` **Exp** `) |`
`before (` **Exp** `,` **Exp** `) |`
`after (` **Exp** `,` **Exp** `) |`
`today |`
`date (` **Exp** `,` **Exp** `) |`
`open (` **Exp** `) |`
`close |`
`write (` **Exp** `) |`

		`system ( ` **Exp** ` ) |`
		Exp `=` **Exp** `|`
		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** `)`

4.2 Keywords

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

```
add     after    and      any      atom     before
bool    close    count    date     days     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      open     or       rel
system  text     today    true     tup      val
write   zero
```

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

4.3 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.

The comparison `x~y` on texts holds true if `x` occurs as a subtext of `y`.

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`.

The expression `days(s,t)` yields the number of days between the dates `s` and `t`. A date is represented as a text of the form `dd/mm/yy`.

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|)`.
- `today` yields the current date as a text of the form `dd/mm/yy`.
- `date(t,i)` yields the date which is `i` days later than the date `t`. A date is represented as a text of the form `dd/mm/yy`.
- `before(s,t)` yields the prefix of `t` that occurs before the first occurrence of the subtext `s`. If `s` does not occur, then the result is the empty text. `after(s,t)` yields the suffix of `t` that occurs after the first occurrence of the subtext `s`. If `s` does not occur, then the result is the empty text.

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.

- The variants `!<` and `!>` have the same semantics, except that the `#` tuples are processed in a non-decreasing respectively non-increasing order according to the attributes `X`.
- `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 the result is `0`. 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`.
- `exp1 ; exp2` is a *sequence* which evaluates first `exp1` and then `exp2`; the result is that of `exp2`.
- `n:=exp` is an *assignment*, which evaluates `exp` and assigns the result to the identifier `n`; the result is that of `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.

- `open(exp)` evaluates the text expression `exp` and opens the file with that name, which is then the current file.
- `close` simply closes the current file.
- `write(exp)` writes the result of the text expression `exp` in the current file.
- `system(exp)` evaluates the text expression `exp` and executes the corresponding command in a UNIX shell; the result is the lines of the standard output concatenated with separating `$` characters.

List of Figures