For it is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used.

Gottfried Wilhelm Leibniz (1646 - 1716)

CONCISE MANUAL

version 0.9

Ferenc Domes, Arnold Neumaier, Kevin Kofler, Peter Schodl, Hermann Schichl

University of Vienna, Austria

Contents

1 Introduction

The programming system ***Concise*** is a graph-based universal programming system for manipulating semantic information stored in the **semantic memory** (2.1). It combines in a novel way the capabilities of imperative programming and object-oriented programming.

Graphical and textual **views** (3.6) enable the user to see, understand, analyze, and modify the content of the semantic memory, and hence of documents, programs and data structures. In particular, there are (currently rudimentary) facilities for semantically accurate automatic document creation in LaTex, written in a user-extensible controlled form of natural mathematical language.

Although currently only an English version is available, ***Concise*** is set up to facilitate the easy generation of multiple-language versions of the documentation and the text appearing in the IDE. In particular, **type systems** (2.11) and dictionaries are implemented in a way that incorporates multiple language names and multiple meanings of names created by different authorities.

The web page on **FMathL** (`http://www.mat.univie.ac.at/~neum/FMathL.html`) (Formal Mathematical Language) contains detailed information about the theoretical background of ***Concise***.

Note: The present help information is available both within ***Concise*** via the Help button, and separately as a PDF document in `<devdir>/docu/Manual.pdf`. See **Editing Help** (3.12) for making changes in this document.

1.1 MSC Classification

Language of mathematics [education; new]: 97E40
Languages and software systems (knowledge-based systems, expert systems, etc.): 68T35
Text processing; mathematical typography: 68U15
Abstract data types; algebraic specification: 68Q65 18C50
Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc.): 68N30
Knowledge representation: 68T30, 68T35, 97R50
Natural language processing: 68T50, 03B65, 91F20
Logic of natural languages: 03B65 68T50, 91F20
Philosophy of mathematics: 00A30 03A05
Specification and verification (program logics, model checking, etc.): 68Q60
Mathematical modeling: 97Mxx, 00A71
e-Learning: 97U50
nonnumerical algorithms: 68W05
Graph theory (in relation to CS): 68R10
Grammars and rewriting systems: 68Q42
Formal languages and automata: 68Q45 03D05, 68Q70, 94A45
Semantics: 68Q55 03B70, 06B35, 18C50
Compilers and interpreters: 68N20
Mathematical problems of computer architecture: 68M07
Graph algorithms: 05C85 68R10, 68W05

1.2 About Concise

Concise was created by `Ferenc Domes` and `Kevin Kofler` based on the theoretical foundation elaborated by `Arnold Neumaier`, `Peter Schodl` and `Hermann Schichl`.

Developement History and Funding:

Concise 0.9: (2010-2011) was created during the MoSMath project, funded by the Austrian Science Fund (FWF) under contract number P20631. Their support is gratefully acknowledged.

1.3 Disclaimer

The authors are furnishing this software "as is". The authors does not provide any warranty of the software whatsoever, whether express, implied, or statutory, including, but not limited to, any warranty of merchantability or fitness for a particular purpose or any warranty that the contents of the software will be error-free.

In no respect shall the authors incur any liability for any damages, including, but limited to, direct, indirect, special, or consequential damages arising out of, resulting from, or any way connected to the use of the software, whether or not based upon warranty, contract, tort, or otherwise; whether or not injury was sustained by persons or property or otherwise; and whether or not loss was sustained from, or arose out of, the results of, the software, or any services that may be provided by the authors.

The short version:
NO WARRANTY!

2 Theory

This section describes the concepts needed to understand the ***Concise*** system.

2.1 Semantic Memory

The semantic memory is an organized collection of **objects** (2.2); parts of the semantic memory are typically displayed as **semantic graphs** with edges connecting these objects. The graph representation of the semantic memory can be viewed in a **graph view** (3.6.2).

The semantic memory changes during a **session** (3.3); its **current state** at any time is given by the **semantic mapping** that associates to any two objects `a` and `b` a third object `c=a.b`, which must be empty if `a` or `b` is empty. Formally, the semantic memory of a ***Concise*** session is therefore a function of time that assigns a semantic mapping to each point in time during which the session exists. But the time-dependence is suppressed in notation and terminology. Depending on the context, we use **SM** as abbreviation both for the semantic memory and for the semantic mapping at a particular time.

The contents of the semantic memory can be also viewed as an ordered collection of semantic units called **sems** (2.3). Each sem is given by an expression `handle.field=entry`, where handle, field and entry are nonempty objects. By definition, the fields of each handle are unique. Therefore adding a sem to the semantic memory replaces an existing sem having the same handle and field, which now points to the newly added entry.

In ***Concise***, the semantic memory is implemented as an associative array of associative arrays of integers:

```
TreeMap<Integer, TreeMap<Integer, Integer>> sems.
```

The entry of a sem is found by looking up the handle from the sem, thereby obtaining the associative array

```
TreeMap<Integer, Integer> fieldEntryMap
```

and then looking up the field from the `fieldEntryMap`, thereby finding the entry.

Ordinary objects are represented as positive integers in the semantic memory, while negative integers reference **external** (**??**) objects. The number 0 represents the empty (undefined or null) object. Ordinary objects may be associated with **names** (2.6).
The root object of the semantic memory of ***Concise*** is of type **Secretary** and has a field `name` which specifies the name of the secretary given by the user (default is 'Matilde', see **starting concise** (3.1)).

2.2 Object

Objects are the contents of the **semantic memory** (2.1) and are organized through **sems** (2.3).
Associated to each object is a **record** (2.4), which consists of all objects that can be reached from the object by following a sequence of moves along sems from their handle to their entry.

Each object may have a **value** to which it refers. A value is a datum **external** (**??**) to the semantic memory but accessible through it by ***Concise***.
Examples of values are wall clock times, pictures, files, but also (depending on the implementation) arrays, floating-point numbers, and integers. ***Concise*** acts on these values according to information stored in the semantic memory, by accessing external processors that can read or change values.
In addition, certain values can be converted into semantic information represented in the semantic memory, or in the reverse direction.

2.3 Sem

A semantic unit (short sem) is a triple `h.f=e` consisting of three nonempty objects, its **handle** `h`, its **field** `f`, and its **entry** `e`. The sems are contained in the **semantic memory** (2.1).

- According to this definition, given a handle `h` and a field `f`, the semantic memory uniquely determines the entry `h.f`; but if this entry is empty, there is no associated sem.
- If the handle `f` is given, we refer to `e` also as the entry of the field `f`.
- If the entry `e` is nonempty, we call `f` a field of the handle, and say that `h` **has** the field `f`.
- By definition of a state of ***Concise***, it depends on the current state of the semantic memory whether a handle has a particular field and which entry a field has.

Roughly, fields are used to look up something, while entries are used to contain something.

A **sem template** is a temporary version of a sem, not part of the current state of the semantic memory but ready to be added to it.

2.4 Record

A record originating from an **object** (2.2) (handle) is the set of all unique **sems** (2.3) contained in the sem sequences originating from the given handle.

A **sem sequence** of length n is given by `handle.field`$_1$`.field`$_2$`. ... . field`$_n$`=entry` and interpreted as the sequence of sems `handle.field`$_1$`=#1, #1.field`$_2$`=#2, ..., #(n-1).field`$_n$`=entry`.
A sem sequence of length 1 is just a sem.

2.5 Externals

Negative object ids from the **semantic memory** (2.1) are associated with external objects by the **externals table**. Formally the externals table maps each (negative) object id to an external by:

`HashMap<Integer, External> id-ExtTable`.
There are two major categories of externals, the **unique externals** and the **mutable externals**. Unique externals are objects which are unique in the way that no two unique externals of the same type with the same value may exist in the external table. Technically, each unique external is mapped to a (negative) object id by

`HashMap<UniqueExternal, Integer> uniqExt-IdTable`.

The 1-1 correspondence of ids and unique externals allows one to find newly inserted unique externals in the external table; if a unique external of the same type and value already exists then no new id is created for it but the existing id is returned. On the other hand, there is no way to change the value of a unique external.
This is, however, possible for mutable externals. There can be several mutable externals having the same type and value in the external table, and their values can be changed freely. Typical examples for unique externals are names, colors, fonts etc, while typical examples for mutable externals are strings, numbers, matrices, etc..

External tables are serialized as described in the corresponding part of the **Serialization** (4.1.1) Section. For the currently implemented external types refer to the **External types** (4.3) part of the Appendix.

2.6 Names

An ordinary object (with a positive id) can be associated with a name. Since programs created by different authors must cooperate, one needs a way to disambiguate names, and this must be recorded in the interpreter-generated section of the semantic memory. ***Concise*** proceeds via a dictionary when a word has several meanings, by distinguishing meanings by their source, here called an **authority**. For example, there is an authority called `System`, referring to the standard distribution of ***Concise***. Language support is realized by introducing different **languages**, for instance all default system names have the language `English`. Formally, the dictionary entries are created in `English` and in the `System` authority, by entering for each quadruple `(#ob,#lang,#name,#auth)` the six sems

```
findname.#ob.#lang = #name
findauth.#ob = #auth
findob.#lang.#name.#auth = #ob
```

where `findname, findauth, findob` are located in:

```
Dictionary.findname=findname
Dictionary.findauth=findauth
Dictionary.findob=findob
```

The dictionary root can be found under (`System`, `English`) in:
`Secretary.Library.Dictionary=Dictionary`

2.7 Semantic Graph

Semantic graphs allow users of ***Concise*** to graphically view the part of the semantic memory they currently focus on. (Besides such **graphical views** (3.6.2), which may be of a varying kind, there are also various **textual views** (3.6.3).)

A **semantic graph** is a labelled directed graph whose **nodes** (2.8) and directed **edges** (2.9) (arcs) are labelled by objects in such a way that every arc from a node labelled `h` to a node labelled `e` is labelled by a field `f` such that `h.f=e` is a sem.
The rules selecting the sems represented in the view can be configured by the user, also whether different nodes are allowed to have the same label.

In a drawing on the ***Concise*** screen, each arc defines an arrow labelled by the name of the field going from a node labelled by the name of the handle to a node labelled by the name of the entry. In the standard

view, the arrows are not drawn explicitly but always go either downwards or horizontally from left to right. The horizontal arrow is reserved for sems whose field is `type`; thus the type of a node (if displayed) is found on its right side. If the value of a node is drawn in a semantic graph, it appears in a square box to the left of the node.

Although several nodes of a semantic graph in graphical display may carry the same label, all nodes with the same label denote the same object in the semantic memory. This enables transparent drawings even for complex semantic relations by decomposing them through the use of multiple nodes for the same object.

2.8 Node

A node is an object shown in the **graph view** (3.6.2). A node has a certain shape containing its name or its identifier. Nodes without names are called **unnamed nodes**. Each node has a number of outgoing and incoming **edges** (2.9). The outgoing edges are called the **child edges** of the node while the incoming edges are called the **parent edges** of the node. The child (parent) edges are unique, that means there cannot be two child (parent) edges having the same name (under the same language and authority, see **names** (2.6)) or identifier.
- Left clicking on the node label selects the node.
- Right clicking on it brings up the menu of all actions applicable for the clicked node. The selectable actions may depend on the previous actions done on the node as well as the number of children, node type etc.
- Left clicking on a node two times centers the view on it and folds or unfolds its children.
- Pressing the left button on a node and moving the mouse drags the node in the view.
- Node shapes, colors, fonts etc. can be adjusted in the **configuration** (3.10).
- The configuration also contains special options for customizing the representation of unnamed nodes.

2.9 Edge

An edge is an object shown in the **graph view** (3.6.2). An edge has a label showing its name or its identifier. An edge has a source (parent) and a target (child) **node** (2.8). This defines its direction. In terms of sems, the label of an edge is the field, that of the source is the handle, and that of the target is the entry. The direction of an edge in the graph view (if not given by the default mentioned above) is signalled by an arrow shown either in the edge label or at the end of the arrow line. - Left clicking on the edge label selects the edge.
- Right clicking on it brings up the menu of all actions applicable for the clicked edge. The selectable actions may depend on the previous actions done on the edge as well as the parent and child node etc.
- Pressing the left button on an edge and moving the mouse bends the edge.
- Edge widths, colors, label fonts etc. can be adjusted in the **configuration** (3.10).
- The configuration also contains special options for customizing the visualization edge directions.

2.10 Types and Categories

Each handle or entry can have a **type**. A type defines some requirements for the sems originating from the typed object. If a handle or entry abides the restrictions given its type it is called **well typed** and **ill typed** otherwise. If a **record** (2.4) contais only well typed objects it is called a well typed record.
A **category** is a generalization of type allowing to define abstract unions of types. Categories (and types) are defined in **type systems** (2.11). Type systems are stored as type sheets (`.cnt` files) which can be checked for correctness by using the **type sheet checker** (3.7).

2.11 Type Systems

A type system is a collection of a set of objects which are called the **categories** of that type system. Type systems are written as **type sheets**. The type sheets from the <`concisedir`>`/sheets/import` directory are

automatically loaded when ***Concise*** starts. This directory contains type sheets for essential type systems like BasicTypes.cnt and optional ones. The optional ones have numbers in their file names indicating the order in which they are imported, e.g., 2-ElementaryActs.cnt means that the type system ElementaryActs is the second optional type system. Adding a new, numerated type sheet to this directory results in automatic import of the new type system at the next time ***Concise*** is started.
Note that errors in the imported type sheets may prevent ***Concise*** from starting up, and must be corrected or removed in order to get the system work again. See **Type Sheet Check** (**??**).

In ***Concise*** all imported type systems can be found under (`System`, `English`) in:
`Secretary.Library.Index.TypeSystem.`
All type system names are inserted in the `System` authority and all type names of the type system under the authority of the type system itself. This enables one to have distinct type systems with distinct type definitions sharing the same name.

For more details on type systems refer to the paper: Peter Schodl, Arnold Neumaier – The FMathL type system, available at the **FMathL homepage** (`http://www.mat.univie.ac.at/~neum/FMathL.html`).

3 Concise

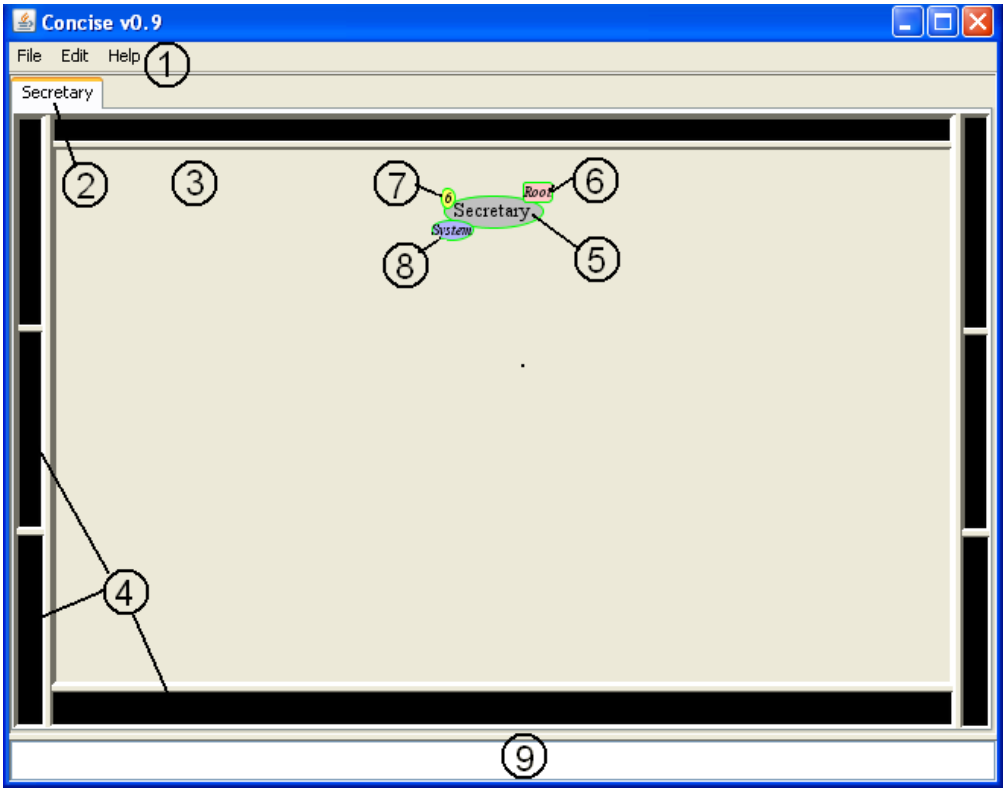
This section describes all features of the ***Concise*** system. The theoretical background needed to understand this part is described in the **theory** (2) section.

3.1 Starting Concise

When ***Concise*** is **started the first time**, a small window called the **session selector** will ask for the location where the **session** (3.3) files shall be stored. The suggested default location is <`concise dir`>`/data/Default.cns`. Later, after the session was saved more often than once, the session selector also allows one to revert to an older saved session.
Then a name of the **secretary** has to be choosen. The secretary is the root of the ***Concise*** session (see **sections of the semantic memory** (3.4)). The default secretary name is Matilde.
Finally, username and password have to be provided, used for identification of the different users inside ***Concise***. (For more details on this topic, see Section **Users** (3.9).)
Then the **main window** (3.2) of ***Concise*** will appear.

3.2 Main Window

When you start ***Concise***, and entered your user name and password, the following window will appear:
The enumerated parts are listed as follows:
- **1** - this is the menu bar containing basic actions like saving and exiting the application showing the help or editing the **configuration** (3.10).
- **2** - this is the tab where the name of root of the active **view container** (3.6.1) is shown. If you open more view containers you get new tabs like this one and you can switch between them by cliking on the tabs.
- **3** - this is the main **graph view** (3.6.2) of the current view container. Right click on it to open the menu of with different options or hold the left mouse button and move the mouse to pan the view. Using the mouse wheel you can zoom in or out. More mouse actions are listed in the **Graph View** (3.6.2) Section.
- **4** - these are space holders for other views in the same container. Right click on one of them to open a new **view** (3.6) in the corresponding part of the view conatiner.
- **5** - this is your root **node** (2.8) of the central graph view, **named** (2.6) the `Secretary`. Right click on it to open the menu of with different options. Double click on it to unfold its child **edges** (2.9) and child nodes. For different mouse actions and applicable keyboard shortcuts see the **Graph View** (3.6.2) Section.
- **6** - this label show the **type** (2.10) of the node.



- **7** - the number of children is shown there.
- **8** - the name of the **authority** (2.6) (owner and creator) of the node can be found here.
- **9** - this small text field is used to dispay system messages.

3.3 Session

The full **semantic memory** (2.1) can be saved as a **session**.

When ***Concise*** is started, either a new session is created or a saved session is restored.
- Upon creating a new session, the user is asked to enter a name for the root object of the semantic memory. This object is the secretary, has the type `Root` and contains as a record the complete semantic memory.
- The session selector allows to load the last or older saved sessions. The default save location can also be adjusted in the session selector. The last saved session has the extension `.cns` while the older sessions end with `.cnb`. For more details on the session selector as well as starting ***Concise*** for the first time, we refer to the Section **Starting Concise** (3.1).

When ***Concise*** is closed, the current session can be saved by choosing `Exit and Save` from the appearing dialog. Obsolete saved sessions can be deleted manually by deleting the corresponding `.cns` and `.cnb` files. For removing all saved sessions and restarting the system, choose `Edit->Reset Session` from the top menu bar of ***Concise***. Note that, since secretary name and user information are also stored in the session, one will be prompted again to choose them.

Compatibility issues: If the basic functionality of ***Concise*** is drastically changed in a new version, the saved session may contain data incompatible with the new version. In this case one may continue to use the

old session on one's own risk, or create a new empty one. To ensure compatibility with a new version, it is therefore advisable to additionally **serialize** (3.5) important user data. In particular, keep the previous version of ***Concise*** to be able to do this even after an upgrade!

3.4 Sections of the semantic memory

The **semantic memory** (2.1) contains the full semantic information relevant for having ***Concise*** work correctly, including the information about names of concepts and their translation to different languages.

The root object of the semantic memory is of type `Root` is called the **Secretary** and contains as a record the complete semantic memory. The name of the secretary is given by the user upon the creation of a new **session** (3.3). This name (and additional information created later) is stored as children of the object `Root.Identity=Identity`.

Concise has access to the values of objects, to auxiliary information needed for navigation through and searching the semantic matrix, and to external processors with which operations can be performed on the values.

A complete semantic memory therefore has four different sections, represented by records whose handles (children of the `Secretary`) hav the names **Library**, **Index**, **Dictionary**, and **Workspace**.
(i) The **Library** stores original information, including programs and background data. This is also the place where a user works and edits.
(ii) The **Index** stores tables for referencing and searching the library. In particular, it has nametables for disambiguation, as standard lexica and dictionaries. (But the contents is in the library, not in the index.)
(iii) The **Dictionary** stores lexical and grammatical information about the terms used, in various languages.
(iv) The **Workspace** stores, separately for each user, their preferences, views of the semantic memory, and private objects, It also contains, separately for each user, all auxiliary information generated by the IDE during a session or via the execution of objects.
However, all four sections use identical forms of representation and the same programs work on them in the same way.

All data generated by the ***Concise*** interpreter for more than intermediate calculations is stored in the workspace of the semantic memory, separately for each concurrent user (if there are several). The complete semantic memory is storable as a **session**, a compact, but platform dependent binary file (with ending `.cns`), from which the SM can be reconstructed quickly and uniqely on the same platform. Therefore, an interrupted interactive session can be quickly resumed later by reloading the stored session.

The workspace of the semantic memory can also be used *later* to reason statistically about the interpreting process, or to create permanent summaries of interest about a library.

3.5 Serialization

Concise can perform a standard serialization of views to .cnv files according to specification given in the **Serialization** (4.1) Section of the Appendix. From the resulting `.cnv` file the deserialization can recover the serialized view (even when the SM was essentially empty before loading it).

3.6 Views

Graphical and textual views enable the user to see, understand, analyze, and modify the content of the **semantic memory** (2.1), and hence of documents, programs and data structures.

For each view, there is an object that serves as the root of the view, also called the **view root**. Some

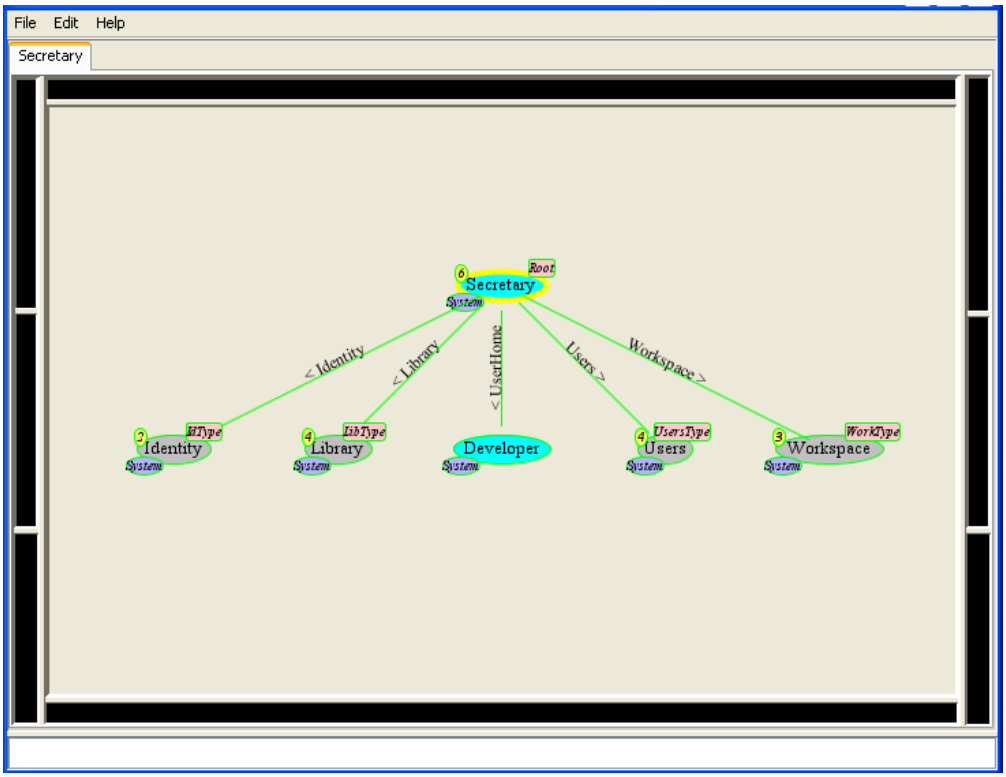
views (e.g., **graph view** (3.6.2)) can have any object as root, while others need an object of a certain type (e.g., the root object **text view** (3.6.3) must be of type `TextDocument`).

Each view has its own management in the semantic memory; different views are completely independent, working on the semantic memory but not on the other views. For example, each graph view only (un)folds his own nodes and not the underlying object, and of two nodes for the same object, one may be folded and the other may be unfolded.
The information required for managing the view information is stored in the **workspace** (see **sections of the semantic memory** (3.4)).

It is possible to see several views at the same time, or have multiple starting objects in the same view; see **view containers** (3.6.1) .

3.6.1 View Containers

Each view in ***Concise*** is contained in a view container. A view container can contain up to nine views. The center view is the root view of the view container and cannot be closed. When a new container is created, the center view is active, and the other eight empty parts are shown as thin black areas around it: In these



empty parts of the view container, new views can be opened or closed by right clicking on them and choosing the corresponding items of the menu.
- Changing the active view container is done by left clicking on the tabs containing its name.
- Each view container can be renamed by right clicking on their tabs.
- Pressing the left mouse button on the tab of a nonactive view container and dragging it into a non center area of the active view container copies the center view of the dragged view container into the selected area and closes the the dragged view container.

3.6.2 Graph View

Using a graph view is one way to navigate, modify, and use the objects stored in the **semantic memory** (2.1). The objects in the graph view are represented either as a **node** (2.8) or as an **edge** (2.9).
- All edges and nodes have a label. A label either shows the name of the node or edge, or a number after a hash (#) sign. The latter case indicates that the node or edge is unnamed, but it can be referenced by the given number. The numbers are unique, and the names are unique under a given language and authority (for more on authorities see the Section **Names** (2.6).
Left clicking on a node or an edge label selects the corresponding node or edge. Right clicking on a node or an edge label opens a small menu showing the list of options applicable for the clicked object.
There are a number of global options that do not require a selected node or edge. These can be accessed by right clicking outside a node or an edge label.

In general, one can issue commands in a graph view by using the mouse and selecting an option from small popup menus. Alternatively, the menu items below the view can be used to issue commands.
- Clicking the left button once on a node or an edge selects the node or the edge.
- Clicking the left button twice on a node centers the view on it and folds or unfolds it.
- Pressing the left button on a node or an edge and moving the mouse drags the node or the edge.
- Pressing the left button outside a node or an edge and moving the mouse moves the view inside the viewport.
- Turning the mouse wheel zooms the view in or out with respect to the position where the mouse pointer is located.
- Pressing the middle button on a node and moving the mouse allows to add sems.
- Pressing the middle button on an edge and moving the mouse to a node sets a new target for the edge.
- Clicking the right button on a node or an edge brings up the allowed options for the clicked node or edge.
- Clicking the right button elsewhere brings up the general options applicable for the current view.

Keyboard shortcuts:
- The `arrow keys` move the view inside the viewport.
- The keys `+` and `-` zoom in and out at the current mouse position.
- Pressing `backspace` centers on the root and resets the zooming.
- If a node or edge is selected, pressing `space` centers the view on it.
- When an edge is selected, pressing `delete` deletes the selected edge.

3.6.3 Text View

Records of the type `TextDocument` can be shown in a text view.
For an example of text views, read the compressed record sheet `con1.cnr.bz` from the <`concisedir`>`/test` directory into the semantic memory by right clicking on the current graph view and choosing `Read->Record Sheet`. The default location for the record sheet read is `Library.TextDocuments.con2=con2`. Right click on one of the black borders in the current view, select the option `New Text View`, enter `con1` or select the root node `con1`, press ok and the text view will appear.
It is possible to mark parts of the text view and inspect the record corresponding to the marked part by right clicking on the text view and selecting `Edit->New view` from the menu. Close a text view by selecting `Apperance->Close view`.

3.6.4 Record View

Records views show a whole record originating from a selected node. This part is unfinished and untested.

3.7 Type Sheet Check

A type sheet consistency check is automatically performed on all imported type sheets, but may also be called manually on an arbitrary type sheet by calling
`java -cp Concise.jar concise.TypeSheetChecker <-flags> <full path>/<sheetname>.cnt`.
Flags:
`c` - convert .cnt to .cnr,
`s` - show the type system.
It is also possible to modify the `checksheet` (for Linux) or `checksheet.bat` (for Windows) scripts located in the main ***Concise*** directory. These scripts can be used to conveniently call the type sheet consistency checker for type sheets under development. A type sheet should be loaded to ***Concise*** only after it was validated by this script.

3.8 Code Record Execution

A code record is a record containing an executable ***Concise*** program. To create a code record, ***Concise*** currently needs a code record sheet. (Later, these will be replaced by code sheets, which simplifies their creation and editing.)

The code record sheet can be loaded from the GUI by selecting the **read>code record sheet** option and executing it by choosing **execute>act**. Code record sheets can be loaded and executed automatically from the operation system command line by writing:

```
java -cp Concise.jar concise.CodeSheetExecuter [-h] [-x] [-n] cnrfile [cnrfile2 ...]
   -h display help
   -x execute the loaded code records
   -n no graphical output
   cnrfile...  one or more .cnr code record sheet files
```

The scripts `execcs` (for Linux) or `execcs.bat` (for Windows) can be also modified and used in order to support the development of new code record sheets.

3.8.1 File View

File views open an external file. Currently only text files can be opened.

3.9 Users

The first time ***Concise*** is started, it prompts for creating a new user or entering an existing **username** and a **password**. In the semantic memory, the **user root** of the active user (referenced as `<userroot>`) can be found in: `Secretary.Users.CurrentUser`,
while other existing users can be accessed from: `Secretary.Users.<username>`
The `<userroot>` contains the username, password, **user home** and the user **configuration**.
- The active user should work and store its data in: `<userroot>.UserHome`
which can be conveniently accessed from: `Secretary.UserHome`
- The user **configuration** (3.10) can be accessed from: `<userroot>.Config`
- The view information is stored in: `Secretary.Workspace.UserData.<username>`
If ***Concise*** was closed and the session was saved, this information is used to restore the working state when the system is started again.

3.10 Configuration

The configuration of the currently active **user** (3.9) contains a number of global settings that can be used to customize the look and feel of ***Concise***. Like all user settings, it can be accessed from the top menu bar

`Edit->Configuration`, alternatively from `<userroot>.Config` (see the Section **Users** (3.9)), or by right clicking on a graph view and selecting selecting `user configuration` from the appearing menu.

3.11 Debug

To enable the debug mode of ***Concise***, place a file called `debug` in the main ***Concise*** directory. Starting ***Concise*** while the debug mode is active skips the authentication procedure and automatically logs into the system with the user name `Developer` and password `developer`. In addition to this, more details about the system will be printed on the console when ***Concise*** starts.

3.12 Editing Help

If one has checked out the whole ***Concise*** development version (in the directory `<devdir>`) from the repository, one can edit this help by modifying
`<devdir>/progs/concise/src/concise/resources/help.txt`.
The formatting commands can be found at the beginning of the `help.txt` file.
The modifications will be visible when ***Concise*** is restarted or the refresh button on the right hand side is pushed.
Note that editing the help this way influences the help of the distrubuted versions only when the jar files are rebuilt.

A PDF version of this Help can be built (or rebuilt after an edit) by running one of the scripts `makehelp` (for Linux) or `makehelp.bat` (for Windows), or alternatively by calling `java -cp Concise.jar concise.HelpGenerator`.

The generated `Manual.tex` and `Manual.pdf` files can be found in the `<concisedir>/docu` directory; the images required for the LaTex document are located in the `<concisedir>/docu/images` directory.

4 Appendix

The appendix contains implementation details, system parameters and a list of Java classes used inside ***Concise***.

4.1 Serialization

Concise can perform a standard serialization of views to .cnv files according to the following specification, and a corresponding deserialization that recovers the view from a .cnv file (even when the SM was essentially empty before loading it).

The basic structure of a serialization is divided in several parts: the **external table** (4.1.1), the **authority codes** (4.1.2), the **language codes** (4.1.3), the **dictionary entries** (4.1.4), the **view roots** (4.1.5) and the **semantic memory** (4.1.6).

There are also some additional **requirements** (4.1.7) and rules which each implementation of the serialization should abide.
A simple serialization example can be found in the section **Example** (4.1.8).

4.1.1 External table

The external table serializes each external object like names, strings, numbers, colors etc. This part of the serialization starts with the line

```
* EXTERNAL TABLE *
```

and each following line contains an entry

`id=<type;value>`
where the `id` is a unique number identifying the external value, the `type` is the a three character type code of the external value (`nam, str, int, dbl`, etc.) and `value` is the string representation of the external value.

More information about external tables can be found in the **Externals** (2.5) Section. The currently implemented external types - divided into mutable and unique externals (explained in the External Section) - are listed in the following table. Each line of the table contains the name of the external and its serialization format `<type;value>`. Note that for the `value` string more explanation is given in this table as it contains formatting information and `meaning(expected format)` items.

```
********************************** MUTABLE EXTERNALS ****************************
Boolean <bol;decision(true or false)>
Dimension <dim;width(integer >0) x height(integer >0)>
Double <dbl;number(double)>
DoubleInterval <din;lower(double),upper(double)>
EscapedString <est;escapedString(java escaped string)>
Integer <int;number(integer)>
IntegerInterval <iin;lower(integer),upper(integer)>
Matrix <mat;[a-11(double),...,a-1n(double);...;a-m1(double),...,a-mn(double)]>
Point <poi;x(double),y(double)>
Rectangle <rec;x(integer),y(integer):width(integer) x height(integer)>
String <str;string(string)>
TextLine <tln;textLine(string without linebreaks)>
Vector <vec;[a-1(double),...,a-n(double)]>
********************************** UNIQUE EXTERNALS *****************************
Abstract2DShape <sha;shapeName(string)>
Array <arr;entry1(external) & ...  & entryn(external)>
Character <chr;char(character)>
Color <col;red(integer 0-255),green(integer 0-255),blue(integer 0-255)>
EscapedCharacter <ech;escapedChar(java escaped character)>
EscapedUniqueString <eus;escapedUniqueString(java escaped string)>
File <fil;pathedFileName(java escaped path string)>
Font <fon;name(string),size(integer >0),"italic"(optional),"bold"(optional)>
IntegerName <ind;uniqueInteger(integer)>
Name <nam;name(string)>
Picture <pic;pathedFileName(string)>
Timer <tim;notificationIntervalls(integer),running(boolean)>
UniqueString <ust;uniqueString(string)>
```

4.1.2 Authority codes

The authority codes are used in the dictionary for assigning each name to a certain owner. This part of the serialization starts with the line

`* AUTHORITY CODES *`

and each following line contains an entry

`aid=eid`

where the `aid` is a unique id identifying the authority while `eid`, is the id of the associated external value.

4.1.3 Language codes

The language codes are used in the dictionary for assigning each name to a certain language. This part of the serialization starts with the line

`* LANGUAGE CODES *`
and each following line contains an entry
`lid=eid`
where the `lid` is a unique id identifying the language while `eid`, is the id of the associated external value.

4.1.4 Dictionary

The dictionary assigns to a triple of an external name, authority and language a new id. This part of the serialization starts with the line
`* DICTIONARY ENTRIES *`
and each following line contains an entry
`eid,iid,aid,lid`
where the `iid` is a unique id identifying the internal entry while `eid`,`aid` and `lid` are the associated external, authority and language ids.

4.1.5 View roots

This part of the serialization starts with the line
`* VIEW ROOTS *`
and the next line contains a list of comma separated of internal or external ids specifying the roots of the serialization.

4.1.6 Semantic memory

This part encapsulates the main part of the data and starts with the line
`* SEMANTIC MEMORY *`
while each following line contains an entry
`handle: field`$_1$`=entry`$_1$`, ... ,field`$_n$`=entry`$_n$
where `handle` and all `field`$_k$ are internal and all `entry`$_k$ are internal or external ids. Each line is thus a compact representation of `n` sems originating from the same handle `handle`. For more details on sems see the section **Semantic memory** (2.1).

4.1.7 Requirements

Every implementation of the serialization should abide the following basic rules:
- Neither the external table nor the dictionary should contains the special name `type` used for defining the type of a Concise object. This special name is marked only by the reserved id `1`.
- For the basic authority the name `System` and for the basic language the name `English` is reserved. Each serialization should contain these.

4.1.8 Example

Minimalist example for serialization of a single type `Trailer` in the type sheet `TextDocument`. In this example the comments start with a `%` sign.

```
* EXTERNAL TABLE *
-1=<nam;English>
-7=<nam;System>
-424=<nam;TextDocument>
-1778=<nam;Trailer>
...
```

```
* AUTHORITY CODES *
12=-7 % System
219=-424 % TextDocument
...
* LANGUAGE CODES *
120=-1 % English
...
* DICTIONARY ENTRIES *
-7,12,12,120 % System(English,System)
-1,12,12,120 % English(English,System)
-424,219,12,120 % TextDocument(English,System)
-1778,4690,219,120 % Trailer(English,TextDocument)
...
* VIEW ROOTS *
...
* SEMANTIC MEMORY *
6155:1=4690 % the type of the node #6155 is 'Trailer'
...
```

4.2 System parameters

```
--------------------------------
Properties of Concise 0.9315 at 2012.04.01 at 11:29:57 CEST
--------------------------------
Sources* Files:400, Lines:67238, Words:202357, Total Size:2052kb
Imports* TypeSheets:6, Views:0
Semantic Memory* Objects:12584, Sems:27430
--------------------------------
Comprehensive word count reference:
Short...  7,000 to 20,000
Novella...  20,000 to 40,000
Catagory...  40,000 to 65,000
Novel...  65,000 to 90,000
Novel Plus...  90,000 to 120,000
Super Novel...  120,000+
--------------------------------
```

4.3 External types

```
ExternalTypes::
!  ***** Type Definitions *****
Color:
nothingElse>
File:
nothingElse>
EscapedCharacter:
nothingElse>
Point:
nothingElse>
Integer:
```

```
nothingElse>
EscapedString:
nothingElse>
Character:
nothingElse>
Boolean:
nothingElse>
Matrix:
nothingElse>
EscapedUniqueString:
nothingElse>
Vector:
nothingElse>
UniqueString:
nothingElse>
IntegerName:
nothingElse>
String:
nothingElse>
Abstract2DShape:
nothingElse>
Array:
nothingElse>
Double:
nothingElse>
TextLine:
nothingElse>
Dimension:
nothingElse>
Picture:
nothingElse>
Name:
nothingElse>
Font:
nothingElse>
Timer:
nothingElse>
Rectangle:
nothingElse>
IntegerInterval:
nothingElse>
DoubleInterval:
nothingElse>
UniqueExternal:
union> Abstract2DShape, Array, Character, Color, EscapedCharacter, EscapedUniqueString, File,
Font, IntegerName, Name, Picture, Timer, UniqueString
MutableExternal:
union> Boolean, Dimension, Double, DoubleInterval, EscapedString, Integer, IntegerInterval,
Matrix, Point, Rectangle, String, TextLine, Vector
External:
union> UniqueExternal, MutableExternal
```

4.4 External functions

```
********************************** NOTATION ******************************
!  - required input parameter, <> - required in-out parameter
******************************************************************************
->package:  math
[Vector x] = linSol(!Matrix A, !Vector b)
Solve the linear system equation Ax=b
[Matrix res] = matAdd(!Matrix mat1, !Matrix mat2)
Add the two given matrices
[Double det] = matDet(!Matrix mat)
Compute the determinant of the given matrix
[Matrix res] = matDiv(!Matrix mat1, !Matrix mat2)
Divide the first matrix with the second
[Double dat] = matGet(!Matrix mat, !Integer row, !Integer col)
Get the given data from the given position in the given matrix
[Matrix L, Matrix U, Vector rowPiv] = matLU(!Matrix mat)
Compute the LU decomposition of the given matrix
[Matrix res] = matMul(!Matrix mat1, !Matrix mat2)
Multiply the two given matrices
matSet(<>Matrix mat, !Integer row, !Integer col, !Double data)
Set the given data in the given position in the given matrix
[Matrix res] = matSub(!Matrix mat1, !Matrix mat2)
Subtract the second matrix from the first
[Matrix matT] = matTran(!Matrix mat)
Transpose the given matrix
->package:  system
alert(!  out)
The string representation of the given external is displayed in a small
message window
[Boolean ok] = confirm(!  question, title)
Display a small window with optional title, containingthe given question,
a yes and a no button.For yes/no, true/false is returned
disp(!  out)
The string representation of the given external is displayed in the console
[Boolean equal] = equal(!  ex1, !  ex2)
Compare the two given externals (as strings) and return true is they are
equal
[Array outArgs] = executeMatlabComm(!String command, !Integer outArgNum)
Execute a matlab command.  The 'command' string contains the command, while
the 'outArgNum' specifies the required number of output arguments.
Currently only Integer, Double and String outputs and arrays of the same
types are supported.
executeShellComm(!String command, String commArgs, String envVars, String workDir)
Execute a shell command.  The 'command' string can contain command
arguments or they can additionally be specified in 'commArg'.  'commArgs'
is either empty or a list of ';' separated command arguments.  Optional
environment variables can be given in 'envVars' as a ';' separated list of
pairs <varName>=<value>.  The full path of the working directory can be
optionally given in 'workDir'
```

```
[String time] = getTime
Return the system time as a string.
[String text] = input(!  question, text, title)
Display a small input window with an optional title, the given question,
an input field containingthe given optional text, and an ok and a cancel
button.The input text or an empty text for cancel is returned
strApp(<>String s, !String s2)
Append the second string to the first.
[String res] = strMerg(!String s, !String s2)
Merge the two string return the result
writeFile(!  out, !File file, Boolean append)
Write the string representation of the given external data to a file.
->package:  text
ifNotEmpty( testarg, text)
If the first argument is empty, the second will be written in the output.
resetAttribs
Reset all style attributes to their default values.
resetColor
Reset the color attribute to its default value.
resetFontSize
Reset the font size attribute to its default value.
setBold
Set the bold attribute.
setColor(!Integer red, !Integer green, !Integer blue)
Set the color attribute to the given rgb value.
setFontSize(!Integer size)
Set the font size attribute to the given value.
setItalic
Set the italic attribute.
setNonBold
Clear the bold attribute.
setNonItalic
Clear the italic attribute.
toggleBold
Toggle the bold attribute.
toggleItalic
Toggle the italic attribute.
->package:  tokensource
out(!Integer mainId, Integer supplementaryId)
This function has either one or two integer args which are passed to the
tokensource.
```

4.5 Source Listing

```
main:
   CodeSheetExecuter,
   ConciseAboutBox,
   ConciseApp,
   ConciseView,
   HelpGenerator,
   SystemInfo,
```

```
   SystemVersion,
   TypeSheetChecker,
   ViewPlotGenerator
acts:
   ElementaryAct,
   ElementaryActRegistry,
   ExecutionException
acts.core:
   ConstInfo,
   Constant,
   ExternalFunction,
   Function,
   InternalFunction,
   VarConst,
   VarConstInfo,
   VarInfo,
   Variable
acts.impl:
   Ask,
   Assign,
   Call,
   Convert,
   Do,
   ForAllFields,
   Get,
   Goto,
   Identical,
   Resume,
   Return,
   Set,
   Supervise,
   Vcopy
acts.parser:
   ActRecordParser
acts.run:
   AbstractEnvironment,
   ElementaryActReturn,
   ExecutionResult,
   LocalEnvironment,
   RuntimeEnvironment,
   Supervisor
acts.run.action:
   EndSuperviseAction,
   EndSuperviseDialogAction,
   ExecuteActAction,
   ExecutionAction,
   JumpAction,
   LeaveLocalAction,
   LoopAction,
   ResumeAction,
   ReturnAction,
```

```
   StartSuperviseAction,
   StartSuperviseDialogAction
core:
   ConciseException,
   ConciseVersionMismatchException,
   Config,
   DeserializationException,
   Serializable
core.action:
   Action,
   ActionList,
   SemAction,
   SemActionUI
core.graph:
   SemEdge,
   SemNode,
   SemNodeCollectorWalker,
   SemanticGraph,
   SemanticGraphEvent,
   SemanticGraphExecutionWalker,
   SemanticGraphListener,
   SemanticGraphUISettings
core.memory:
   AuthedName,
   LangedAuthedName,
   LangedName,
   Name,
   NameSequence,
   Sem,
   SemSequence,
   SemTemplate,
   SemanticIterator,
   SemanticMemory,
   SemanticMemoryEvent,
   SemanticMemoryListener,
   SimpleSem,
   SingleName
develop:
   ComplexRecord,
   EditorRecordView,
   Misc,
   Record,
   SemSequence,
   SemanticMemory-old,
   SimpleRecord
editor:
   Editor,
   EditorAction,
   EditorAdapter,
   EditorException,
   EditorListener,
```

```
   EditorView,
   EditorViewContainer,
   EditorViewEvent,
   EditorViewListener,
   EditorViewSelectorPanel
editor.browserview:
   BrowserView
editor.fileview:
   FileView
editor.graphview:
   GraphItemSelectionEvent,
   GraphItemSelectionListener,
   GraphView,
   GraphViewPanel,
   GraphViewToolset
editor.matlabview:
   MatlabConsoleView
editor.recordview:
   RecordEditor,
   RecordUnitTreeNode,
   RecordUnitTreeNodeEvent,
   RecordUnitTreeNodeListener,
   RecordView,
   RecordViewPanel
editor.textview:
   TextView
external:
   External,
   ExternalConversionException,
   ExternalConverter,
   ExternalConverters,
   ExternalDynamicObject,
   ExternalObjectEvent,
   ExternalObjectListener,
   ExternalRegistry,
   ExternalSelectorUI,
   ExternalTable,
   ExternalUI,
   ExternalVisualObject,
   ExternalsEditorUI,
   MutableExternal,
   UniqueExternal
external.convert:
   ExternalFileToString,
   ExternalIntegerToString,
   ExternalNameToString,
   ExternalStringToFile,
   ExternalStringToInteger,
   ExternalStringToName
external.functions:
   ExternalFunArgInfo,
```

```
   ExternalFunException,
   ExternalFunRegistry,
   ExternalFunResult,
   ExternalFunction,
   FunArgInfo
external.functions.math:
   LinearSolve,
   MatrixAdd,
   MatrixDet,
   MatrixDiv,
   MatrixGet,
   MatrixLU,
   MatrixMul,
   MatrixSet,
   MatrixSub,
   MatrixTrans
external.functions.system:
   Alert,
   Confirm,
   Disp,
   Equal,
   ExecuteMatlabComm,
   ExecuteShellComm,
   GetSystemTime,
   Input,
   StrApp,
   StrMerg,
   WriteFile
external.functions.text:
   IfNotEmpty,
   ResetAttribs,
   ResetColor,
   ResetFontSize,
   SetBold,
   SetColor,
   SetFontSize,
   SetItalic,
   SetNonBold,
   SetNonItalic,
   TextConstants,
   TextFun,
   TextReturn,
   ToggleBold,
   ToggleItalic
external.functions.tokensource:
   Out
external.interfaces:
   ExternalCharSequence
external.objects:
   ExternalAbstract2DShape,
   ExternalArray,
```

```
   ExternalCharacter,
   ExternalColor,
   ExternalEscapedCharacter,
   ExternalEscapedUniqueString,
   ExternalFile,
   ExternalFont,
   ExternalIntegerName,
   ExternalName,
   ExternalPicture,
   ExternalTimer,
   ExternalUniqueString
external.values:
   ExternalBoolean,
   ExternalDimension,
   ExternalDouble,
   ExternalDoubleInterval,
   ExternalEscapedString,
   ExternalInteger,
   ExternalIntegerInterval,
   ExternalMatrix,
   ExternalPoint,
   ExternalRectangle,
   ExternalString,
   ExternalTextLine,
   ExternalVector
matlab:
   MatlabConnector
parser:
   ConciseTokenSource,
   Grammar,
   GrammarException,
   TextParagraphIterator,
   Unlexer
records:
   RecordException,
   RecordRoot,
   RecordSheet,
   RecordUnit
session:
   Dictionary,
   DictionaryEntry,
   Session,
   SessionAuthority,
   SessionConstant,
   SessionConstants,
   SessionLanguage,
   SessionSettingSelector,
   SessionSystemConstant,
   SessionUI,
   Workspace
test:
```

```
   ActsTest,
   ConstantsTest,
   ConvertTex2cnr,
   CoreTest,
   EditorTest,
   ExternalMatrixTest,
   ExternalTest,
   GrammarTest,
   GraphTest,
   LatexTest,
   MatlabTest,
   OEdit,
   ParserTest,
   PgfTestJava,
   RecordTest,
   TextViewTest,
   TypeCheckTest,
   TypeSystemTest,
   UsageTest,
   ZoomPanelTest
types:
   TypeCheckException,
   TypeChecker,
   TypeException
types.def:
   TypeDef,
   TypeDefUI,
   TypeRestriction,
   TypeRestrictionUI
types.entries:
   TypeEntry,
   TypeEntryCollection,
   TypeEntryEquationList,
   TypeEntryNameList,
   TypeEntryOneName,
   TypeEntryRegistry,
   TypeEntryTrivial,
   TypeEntryUI,
   TypeEntryUnited
types.entries.data:
   TypeEntryData,
   TypeEntryEquationData,
   TypeEntryNameData
types.entries.impl:
   TypeEntryAllOf,
   TypeEntryArray,
   TypeEntryAtomic,
   TypeEntryComplete,
   TypeEntryFixed,
   TypeEntryIndex,
   TypeEntryItself,
```

```
   TypeEntryNothing,
   TypeEntryNothingElse,
   TypeEntryOneOf,
   TypeEntryOnly,
   TypeEntryOptional,
   TypeEntrySomeOf,
   TypeEntrySomeOfType,
   TypeEntryTemplate,
   TypeEntryUnion
types.system:
   TypeSheetParser,
   TypeSystem,
   TypeSystemImport,
   TypeSystemImports,
   TypeSystemRecordParser
types.usage:
   LitSubst,
   LitVar,
   LitVars,
   TargetLabels,
   Usage,
   UsageConstants,
   UsageException,
   UsageTarget,
   UsageToken,
   UsageTokenArray,
   UsageTokenMap,
   UsageTokenRegistry,
   UsageWalker,
   UsageWalkerAdapter,
   Usages
types.usage.atoms:
   CatVar,
   CatVarId,
   CatVarName,
   CatVarRec,
   ChrRan,
   FunArgs,
   Function,
   LineBreak,
   LitId,
   LitVal,
   Literal
types.usage.expressions:
   Alternative,
   Anytimes,
   ChrRanges,
   Expr,
   Multiple,
   Once,
   Optional,
```

```
   RegExpr
types.usage.matches:
   Except,
   Expect,
   MatchCase,
   Maximal,
   PatternMatch,
   Taboo
users:
   User,
   UserException
```