The Cedalion Workbench

Boaz Rosenan

22997: Graduate Project in Computer Science
Software Engineering Lab
Dept. of Mathematics and Computer Science
The Open University

Supervisor: Prof. D. H. Lorenz

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 The Cedalion Programming Language . . . 6
- 1.3 Development Status . . . 7

2 Requirements **8**
- 2.1 Scope . . . 8
- 2.2 Cedalion Workbench Overview . . . 8
- 2.3 Functional Requirements . . . 10
  - 2.3.1 Projectional Editing . . . 10
  - 2.3.2 The Text Bar . . . 11
  - 2.3.3 Auto-completion . . . 11
  - 2.3.4 Aliases . . . 12
  - 2.3.5 Context Menu . . . 12
  - 2.3.6 Adapters . . . 12
  - 2.3.7 Definition Search . . . 13

3 Software Design **14**
- 3.1 Software Architecture . . . 14
  - 3.1.1 Client: The Eclipse Plug-in . . . 14
    - 3.1.1.1 Eclipse . . . 14
    - 3.1.1.2 The Cedalion Workbench Plug-in . . . 14
    - 3.1.1.3 Figures and Commands . . . 16
  - 3.1.2 Server: The Logic Engine . . . 16
    - 3.1.2.1 SWI-Prolog . . . 16
    - 3.1.2.2 The Cedalion Logic Engine . . . 17
    - 3.1.2.3 The Cedalion Base Code . . . 17
    - 3.1.2.4 User DSLs and Programs . . . 18
- 3.2 Functionality . . . 19
  - 3.2.1 Design Principles: Client/Server Approach . . . 19
    - 3.2.1.1 Client/Server Examples . . . 19
    - 3.2.1.2 Procedures and Commands . . . 19
  - 3.2.2 Requirements Walkthrough . . . 20
    - 3.2.2.1 Projectional Editing . . . 20

3.2.2.2 The Text Bar 21
3.2.2.3 Auto-Completion 21
3.2.2.4 Aliases 22
3.2.2.5 Context Menu 22
3.2.2.6 Adapters 23
3.2.2.7 Definition Search 23

Bibliography **24**

A Communication Protocols **25**
A.1 The Cedalion Public Interface (CPI) 25
A.2 Built-in Predicates 25
A.3 Service Predicates 25

Abstract

Cedalion is a programming language designed as a host for internal domain specific languages (DSLs). It features syntactic freedom and composability of DSLs, thanks to the use of projectional editing.

This software project provides a projectional editor for the Cedalion programming langauage, named the Cedalion Workbench. It is developed in Java, Prolog and Cedalion, as a plugin for the Eclipse platform.

Chapter 1

Introduction

This paper describes the design and implementation of the Cedalion Workbench, which is a significant part of my research. The implementation itself is an open-source project hosted in SourceForge (http://cedalion.org). The source-code to this project is accessible through the Mercurial repository, and there are regural releases containing new features and bug-fixes.

1.1 Background

Cedalion is a *Language-Oriented Programming Language* [6]. As such, it is a programming language designed for the *Language Oriented Programming* paradigm. To our knowledge, this is the first language with this attribute.

Language-Oriented Programming (LOP) is, at least by name, a relatively new software development paradigm. In the traditional approach, the programming language is selected first, and then the software design process translates the concepts of the problem domain into the concepts of that programming language (such as identifying "problem-domain objects" in OOD and building class diagrams based on them). In contrast to that, with LOP we first *design the programming-language* to suite our needs, and then write the software using that language. In other words, instead of bringing the problem-domain to the language, LOP brings the language to the problem-domain.

The "programming languages" we end up with when using the LOP approach are usually oriented at a specific problem-domain, i.e., these are ususally *Domain-Specific Languages (DSLs)*. However, the term *problem-domain* here is not restricted to a end-user problems. For example, in a billing system for a Telecom carrier, the problem-domain is not restricted to the domain of billing phone conversations. The analysts building the billing schemes are indeed most interested in that problem domain, but there is much more into this kind of system than just that. Querying the system logs for relevant (billable) information, communicating with internal and external servers to collect information, formatting the bills and providing user interface can all be considered "problem-

domains", in the wider sense. As such, they all "deserve" their own DSLs for effective implementation à la LOP.

The contemporary LOP approach [2, 3] advocates the use of multiple, interoperable DSLs, each one focused on one problem-domain, such that together they can cooperate to address large multiparadigm software. This approach is very favorable for code reuse, since DSLs dealing with globally-relevant problem-domains, such as user-interface or networking, can be shared in a very wide scope (e.g., posted on the Internet).

Indeed, LOP seams promising, but there are challenges to overcome. The main challenge associated with LOP is the need to effectively define and implement DSLs. This is especially difficult when we need these DSLs to be interoperable with one another, that is, allow code in one DSL be wrapped around code in another. Traditional language implementation techniques, such as the use of Lex and Yacc will not provide us with a solution for that problem.

Currently, there are two approaches that allow effective and modular DSL development: *Internal DSLs* and *Language Workbenches*.

Internal DSLs These are DSLs that are defined from within an existing programming language[1], named the *host* language. Lisp and its dialects, Ruby, Smalltalk and Haskell are known to be good hosts for internal DSLs. For example, Ruby-on-Rails [1] is a web-development platform that makes extensive use of internal DSLs.

Language Workbenches These are Integrated Development Environments (IDEs) for defining, implementing and using *External DSLs* [3]. Their power is in the fact they do not pose any limitations in terms of syntax and semantics on the DSLs they support. Semantic freedom is inherent to the use of external DSLs. The syntactic freedom is provided by the use of *projectional editing* [4], which can be seen as the Model/View/Controller (MVC) Architecture, applied to a textual programming language. With projectional editing, the code is being edited through a view, displaying an underlying model, where the *projection* (the "controller") is determined as part of the language definition. Projectional editing allows such DSLs to have any syntax, regardless of the limitations of a certain parsing algorithm (parsing does not take place), and even syntactic ambiguity. Disambiguation is done when entering code, by selecting the correct construct from a menu. This way, while two different phrases may look similar, they are different down at the model level.

LOP Languages There are significant trade-offs between these two approaches. On the one hand, internal DSLs are easier to implement than external DSLs (even when using language workbenches), but they are bound by the syntax and

[1] The term *internal* DSLs was coined by Fowler [3]. Others [5] use the term *Embedded* DSLs to describe the same thing. We follow Fowler's terminology to avoid confusion with languages such *Embedded SQL*, languages implemented using pre-processors on the host language. These languages are actually external DSLs.

semantics of the host language. On the other hand, language workbenches provide freedom to create better DSLs, but implementing them is harder relative to internal DSLs.

To overcome these trade-offs, our research introduces the concept of *LOP Languages*. These are programming languages designed to take the best of both worlds, providing a good solution for LOP.

LOP Languages are good hosts for internal DSLs, allowing easy DSL implementation. However, they import two important features from language workbenches:

1. Projectional Editing, as a way to provide syntactic freedom; and
2. DSL Schema, a method to allow formal definition of the validity of DSL code.

Cedalion is an instance in this class of languages, providing a proof of concept. The rest of this document will focus on Cedalion specifically, where the discussion brought in this section provides the motivation.

1.2 The Cedalion Programming Language

In Section 1.1 we explored the motivation for Cedalion. In this section we provide a brief overview of the Cedalion programming language.

Following the description in Section 1.1 and in [8], to qualify as an LOP language, Cedalion needs to be able to host internal DSLs and support DSL schema and projectional editing.

Cedalionis a logic-programming langauge, mostly based on Prolog. As such, it can host internal DSLs following the method described by Menzies [7].

To support DSL schema, Cedalion provides a static type-system, based on Hindley/Milner type inference. This provides the basic mechanism for defining and enforcing schemata for DSLs. Each new construct needs to have a type-signature defined, defining the construct's type in terms of the types of its arguments. Collecting the type-signatures of all the constructs of a certain DSL can be seen as its schema. They provide a set of rules for the structural validity of DSL code. The type-system will then enforce these rules, so that any well-typed piece of DSL code will be considered "valid" in the eyes of the DSL schema. In addition to the type-system, DSL developers can define their own "checkers" to check for domain-specific terms of validity. This allows Cedalion to report errors in a way that makes sense to its DSL users.

Finally, to support projectional editing, Cedalion includes *projection definitions*, which are statements that transform language constructs to visualization objects. These objects are supported by the *Cedalion Workbench*, which displays them to the user.

A complete definition of a language construct in Cedalion consists of a type-signature, a projection definition (optional, defaults to a Prolog-like syntax), and a semantic definition. The semantic definition depends on the type of

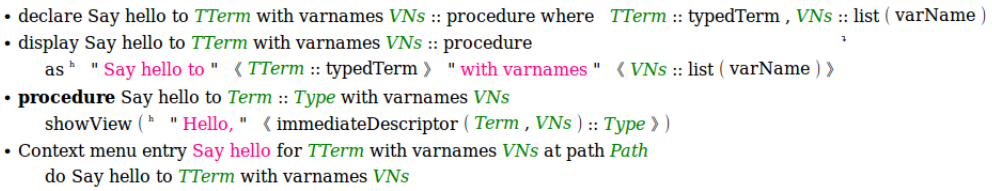

Figure 1.1: A screenshot of a Hello, World program in Cedalion

language construct being defined. In Cedalion parlance, such a construct is called a *concept*.[2]

Figure 1.1 shows a "Hello, World" program in Cedalion. The program contributes a context-menu entry labeled "Say Hello", that once clicked it shows the object that was right-clicked (to receive the context menu) in the Cedalion View, to the right of the word "Hello". The code begins with a definition of a new concept - the "Say hello to" procedure. It begins with a type signature (the *declare/where* statement), followed by a projection definition (the *display/as* statement), followed by a semantic definition (the *procedure* statement), and finally, the definition of the context menu entry, associating the label "Say Hello" with the procedure.

1.3 Development Status

The Cedalion implementation is an open-source project hosted in SourceForge. Its declared state of maturity is "pre-alpha", meanning it is a work-in-progress and is not yet ready for prime-time use. Nevertheless, Cedalion gets around ten downloads per week from different countries around the globe. Judging by the reported statistics, some of these downloads are made by random visitors to the website, while a few of them seem to be made by repeating users, coming to get a newer version. The on-site documentation includes a "hello, world" tutorial, introducing the concepts of projectional editing to the users. However, judging from the number of views on that page, it has not been tried by too many people.

The main challenge in the implementation of Cedalion is stability. Cedalion's projectional editing allows for user code to run from within the projectional editor, as it is being edited. Buggy code, even ill-typed code can still run in this context. The key in stabilizing such software is to localize the effect of bugs. The effect of bugs resulting in an exception is easy to localize (catch the exception and move on). However, some bugs result in non-termination, which is significantly harder to contain.

[2] The same term is also used in [2].

Chapter 2

Requirements

This chapter describes the requirements from the Cedalion Workbench.

2.1 Scope

The scope of this project (and thus of its requirements) is the Cedalion Workbench, an Eclipse plug-in desinged as a dedicated editor for the Cedalion programming language.

2.2 Cedalion Workbench Overview

Cedalion code cannot be edited using traditional text editors and can only be edited using a special *projectional editor*, a dedicated editor for projectional editing. The Cedalion Workbench [?], an Eclipse-based IDE, implements such a projectional editor. *Projectional editing* offers an alternative to the traditional parsing approach. With projectional editing, instead of editing the code in a text editor and then parsing it to form an abstract syntax tree (AST) of the code, we edit the AST directly, and present it to the user using a *projection*, that is, a transformation to some human readable representation, which is usually (but not necessarily) textual.

Figure 2.1 is a screenshot of the Cedalion workbench. The editor screen is structured similar to the user interface of a Web browser. The top of the window contains a *text bar* (similar to the address bar of a Web browser) with a few action buttons to its left. The text bar displays the text representation of the currently selected Cedalion code. The rest of the window's real-estate (*code area*) is dedicated to projecting the content of the file being edited. The tab label indicates the name of the file.

Even at first glance, one can see that the Cedalion workbench differs from a text editor. The projected Cedalion code contains special symbols, and it is displayed using varying font sizes and unorthodox layout. The code comprises a hierarchy of rectangular elements (we call *terms*), nested inside one another.

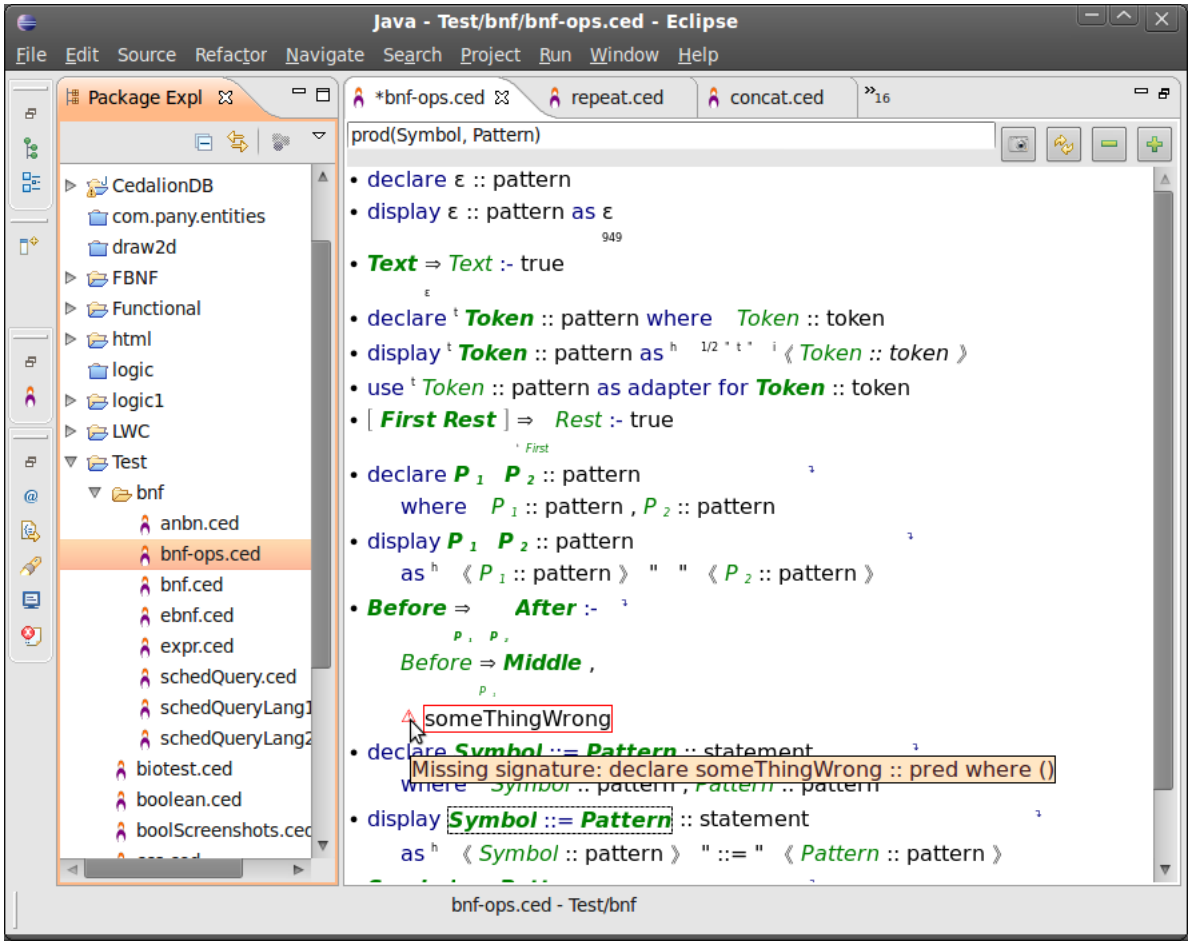

Figure 2.1: A screenshot of the Cedalion workbench

When the user clicks on a term, a selection box appears around it, and the content of the text bar is replaced with a textual "Prolog-like" projection of that term. This textual representation can be edited, and when hitting Enter, the changes are applied to the code area, assuming the text complies with the simple Prolog-like syntax.

The edited code can be either valid or invalid Cedalion code. Unlike many syntax driven editors, Cedalion does allow invalid code to be edited. For example, an undefined term can be used. In such a case, Cedalion will mark this term with an error marker (a red rectangle with a small red warning sign symbol at its top left), and provide the details of the error as a tooltip. As can be seen in Figure 2.1, in the case of an undefined concept, Cedalion claims a missing signature, and uses type inference to suggest what this signature might be. Indeed, double clicking the red warning sign will suggest inserting such a type signature before the current statement. This is the way new concepts can be introduced. Concepts and type signatures are discussed in [6].

2.3 Functional Requirements

2.3.1 Projectional Editing

Cedalion's syntax is based on projectional editing. Supporting this is the essence of what the Cedalion Workbench is made to do. To support projectional editing, the Cedalion Workbench provides an editor. This editor supports a model-view-controller behavior, where:

- The *model* is the Cedalion program.
- The *view* is a visual, human readable representation of that program, and
- The *controller* consists of a set of rules defined in the Cedalion program itself, defining how nodes in the model are translated to a view.

Applying the controller rules on the model provides the human readable representation of the program. However, the view provides more than that. To allow editing, the editor maintains traceability between view elements and their underlying model elements. Each portion of the view representing a node in the model is selectable. Editing operations made when such a view element effects the underlying model node. Such operations replace the current node with a different one. After the editing operation has been performed, the modified model is re-projected to a view, synchronizing the view and the model.

The view consists of several types of elements (*figures*), providing different visualization abilities. These include:

- *Labels*, displaying plain text.
- *Symbols*, displaying a single unicode character.
- *Horizontal flow*, displaying its contained figures horiozntally.

- *Vertical flow*, displaying its containd figures vertically.
- *Font modifiers*, modifying the style of the font in the figures they contain.
- *Text-color modifier*, modifying the color of the contained text.
- *Line-border*, displaying a solid rectangle around the contained figure.
- *Background*, providing a colored background for the contained figure.
- *Raized / Lowered border*: display a raized/lowered border around the contained figure.
- *Expand figure*: contains two figures: expanded and collapsed. Switches between them when clicking the collapse/expand icon displayed next to the figure.
- *Action figure*: performs some action when the contained figure is double-clicked.
- *Brackets*: surrounds the contained figure with "brackets", made of two Unicode characters, for which the font size is adapted to match the height of the contained figure.

2.3.2 The Text Bar

At the top of the Cedalion Workbench editor window there is a text box, named the Cedalion *text bar*. This text bar provides a textual representation of the selected element. When the user selects a node in the projectional editor, the content of the text bar is replaced with its textual representation. This textual representation uses a Prolog-like syntax. The user can edit the content of the text in the text bar. When the user presses the Enter key, if the content of the text bar is valid with regard to the Prolog-like syntax, the selected node and all the nodes below it are replaced with the content of the text bar, parsed.

The user can select nodes with a lot of content below them. To avoid very long text in such cases, the sub-tree displayed in the text bar is trimmed to a fixed depth. Trimmed nodes are represented with a dollar sign ($) followed by a number. When parsing the text in the text bar, the Cedalion Workbench replaces the dollar-sign-number sequences with the sub-trees they represent. The trimmed sub-trees are stored for the duration of the editor's operation, to allow text-level copy and paste of large trees.

2.3.3 Auto-completion

Plain use of the text bar as described in Subsection 2.3.2 is good when creating a new language or creating new language constructs, as demonstrated in the Hello, World example [**?**]. However, most of the time, when using existing language constructs, auto-completion can be used to help the users find the constructs they want, and avoid misspelled names.

When entering text in the text bar, the user can press a key combination (Control+Space) to get a list of suggestions. The Cedalion Workbench should query the collection of concepts that can be used in the selected location, and finds ones which have *aliases* (explained next) starting with the string entered. The list is then presented to the user for choosing the appropriate concept, which is then inserted as a new term.

2.3.4 Aliases

Aliases are strings associated with concepts. Each concept has a "natural" alias, which is its internal identifier (the name Cedalion uses internally, regardless of projection), without namespace prefixing. Additional aliases can be defined by the user. The Cedalion Workbench also infers aliases in some cases from projection definitions (e.g., when the projection is a label, the content of the label is used as an alias for this concept). Concepts and aliases have a many-to-many relationship, where a single concept can have multiple aliases (e.g., its internal name and something based on its projection), and several concepts can share the same alias. In the latter case, disambiguation should be done by choosing the desired entry from the auto-completion list of choices.

2.3.5 Context Menu

The Cedalion Workbench provides a context menu for every term (a code element, represented as a rectangular feature by the projectional editor). Right clicking a term causes a pop-up menu to appear, listing operations relevant to this term. Selecting this operation executes it, performing an action such as modifying code or displaying content in the Cedalion view (a part of the Cedalion Workbench made for displaying information and interacting with the user). User code can contribute context menu entries by using *context menu entry* statements. These statements associate the caption of the entry with an action. They also specify a pattern for the term to be matched, in a form of a typed term. This allows entries to be specified for specific concepts, or specific types, thus making the menu context-dependent.

2.3.6 Adapters

An adapter of type T_1 to type T_2 is a concept of type T_2, which takes one argument of type T_1, and semantically acts as a proxy, adding no additional meaning to its argument. Cedalion has a special declaration for declaring a concept as an adapter. This allows the Cedalion Workbench to reconcile concepts of type T_1 in the context where a concept of type T_2 is needed. Adapters allow the Cedalion Workbench's auto-completion to offer concepts of type T_1 in these cases, and when a type mismatch between T_1 and T_2 is presented, the Cedalion Workbench automatically inserts the adapter to fix this error.

2.3.7 Definition Search

Concepts are centric to the way Cedalion software is programmed. Concepts are introduced in Cedalion in both the DSL definition and the DSL code. To allow Cedalion users to be able to understand the different concepts and be able to track their definitions, the Cedalion Workbench provides a mechanism for searching concept definitions. When selecting a compound term, Cedalion's context menu displays the option "Show Definitions." Selecting this option will display the full story behind the concept associated with the selected term. This "story" includes all aliases assigned to this concept, the type signature, projection definition and all semantic definitions. The nature of the semantic definitions for a concept depend on the concept type. For example, for predicates, the semantic definition includes all clauses contributing results to that predicate. A semantic definition of a statement includes all rewrite rules translating this statement into others. For a type, it includes all type signatures of concepts of that type. The user can relate new defining statements to concepts using "defines" statements. Each defining statement is displayed along with the file name in which it is defined. Clicking that definition will open that file, and highlight the relevant definitions with a green background.

Chapter 3

Software Design

3.1 Software Architecture

This chapter we describe the "anatomy" of the Cedalion Workbench implementation, identifying its main componenets, describing the role of each of them and the interactions between them.

Cedalion is designed to be used in client/server settings, where the "client" is responsible for interacting with the outer world, and the "server" contains the Cedalion program, and provides the client with guidance as to how to perform its job. The Cedalion Workbench is no exception for this. Here we describe its structure, which is presented graphically in Figure 3.1.

3.1.1 Client: The Eclipse Plug-in

In the Cedalion Workbench, the "client" side consists of an Eclipse Plug-in implemented in Java. This plug-in contributes the Cedalion Editor, the projectional editor for Cedalion, and the Cedalion View, which allows Cedalion code to display visuals. In addition, the plug-in conatins a collection of Java classes, that can be used to perform client-side operations. The client side is depicted at the right side of Figure 3.1.

3.1.1.1 Eclipse

At the bottom of the client stack is Eclipse, a third-party piece of software that was initially developed by IBM, and then became open-source and is now developed by a large group of voulenteers, sponsored by IBM. Eclipse is almost entirely implemented in Java.

3.1.1.2 The Cedalion Workbench Plug-in

On top of Eclipse, is the Cedalion Workbench Plug-in. It is implemented in Java, with an XML configuration file (plugin.xml) describing the contributions of this

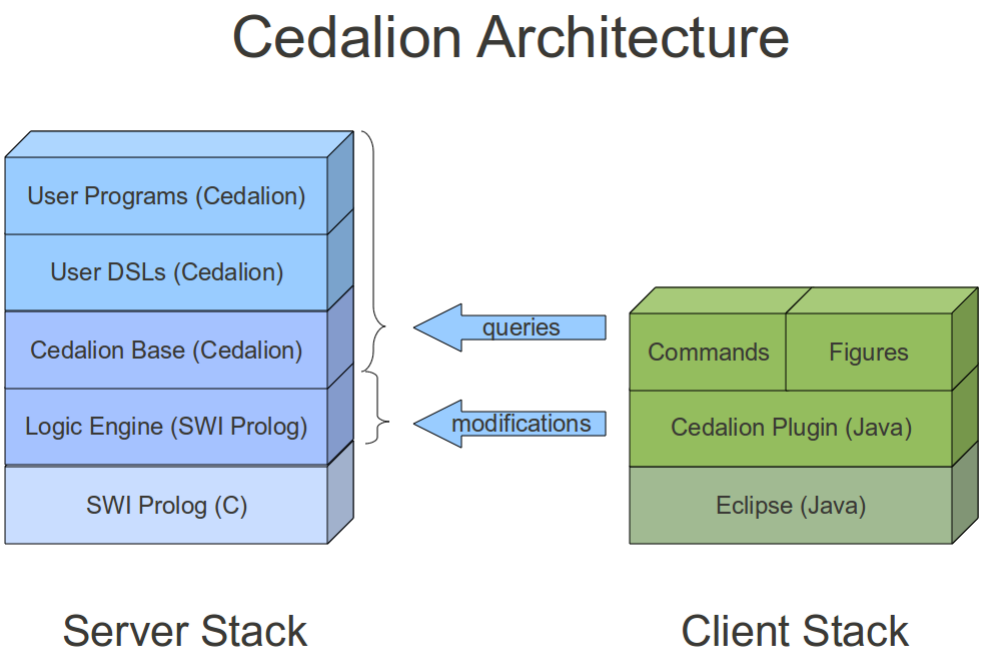

Figure 3.1: The Cedalion Workbench Client/Server Architecture

plug-in to the Eclipse environment. As stated above, the two main contributions provided by this part are the Cedalion Editor (class net.nansore.cedalion.eclipse.CedalionEditor), and the Cedalion View (class net.nansore.cedalion.eclipse.CedalionView). The editor is associated with the file extension ".ced", so that files holding this extension will be opened using this editor, and will display the Cedalion logo in the file browser. In addition to the editor and view, the Cedalion plug-in is also responsible for scanning the Eclipse workspace for files with the ".ced" extension, and loading them into the server. This is how the user determines what program runs on their system.

3.1.1.3 Figures and Commands

The Cedalion Workbench plug-in is an extension to the Eclipse platform, and it is extensible by itself. Two kinds of extensions exist for the Cedalion Workbench plug-in: Figures and Commands.

Figures are visual objects used for displaying Cedalion code. Figures are the most basic elements, providing the basic display capabilities. Combining several figures together allows us to display complex terms.

Commands are objects providing the *actions* a Cedalion program can perform. Since Cedalion is a pure declarative language, the server side (containing the Cedalion program) cannot *do* anything on its own. Instead, the client can query it to get answers. These answers may include, which *commands* need to be executed. The client holds a collection of commands available. These include commands for manipulating the display (e.g., showing something in the Cedalion View), modifying the Cedalion program (load a file, add or remove a statement), etc. Some of these commands, such as the ones for manipulating the Cedalion program, require the assistance of the server. There are also control commands, such as *doAll* (class net.nansore.cedalion.cmd.DoAll), which takes a list of commands and executes them sequentially.

Both collections of commands and figures are extensible, meanning that a third-party user can add an Eclipse plugin containing their own implementations, and write a Cedalion program that makes use of the new figures/commands.

3.1.2 Server: The Logic Engine

The server side's role is to contain the Cedalion program, and to allow the client to query it. It is depicted at the left hand side of Figure 3.1.

3.1.2.1 SWI-Prolog

As can be seen at the left side of Figure 3.1, the server stack begins with SWI-Prolog, a Prolog interpreter developed at the University of Amsterdam as an open-source project. The choice of SWI-Prolog was rather arbitrary, based on its popularity. Another candidate to consider is YAP (Yet Another Prolog), developed at the University of Porto, Portugal, and the University of Rio Dejenero, Brasil. YAP is also open-source, it is not as popular and probably less

stable, but provides siginificantly better performance. Replacing one Prolog implementation with another requires a reasonable amount of porting work.

3.1.2.2 The Cedalion Logic Engine

Above SWI-Prolog, a Prolog program implements the Cedalion Logic Engine. This is actually the implementation of the "core" of Cedalion. Contained in the file *service.pl*, this layer provides implementation for a communication protocol between the client and the server, providing access to its predicates. These predicates include services for manipulating the Cedalion program, and the predicates of the Cedalion program themselves. The services provided by the logic engine include:

1. Loading (or re-loading) a Cedalion file into the Cedalion Program.
2. Adding / Removing a statement from the Cedalion program.
3. Loading / Saving a Cedalion file into / from an in-memory representation.

The Cedalion Logic Engine is also responsible for Cedalion's module system. When loading Cedalion program files, it adds module-specific prefixes to names, thus preventing collision between concepts with the same local name. Modules can still access concepts defined in other modules by explicitly adding their prefix.

In addition, the Cedalion Logic Engine provides Cedalion with its builtin predicates. These are Prolog predicates that are exported to the Cedalion environment by naming them within the *builtin* namespace. At present there are around thirty such predicates.

3.1.2.3 The Cedalion Base Code

The Cedalion Logic Engine is capable of loading and running a Cedalion program, but at this point, it cannot do much. The "core" Cedalion has its extensible core, but it does not have a type-system and does not support projectional editing. These features (and more) are provided by the Cedalion Base Code, the *bootstrap* module. It has the following roles:

1. Provides implementation for the predicates in the *Cedalion Public Interface* (*CPI*) namespace. These are interface predicates used by the Cedalion Workbench Plug-in. They answer questions such as "how do I open a file?" or "how do I visually display a construct?". These predicates complete the implementation of the Cedalion Workbench.
2. Provide Cedalion programmers the ability to control these predicates (and thus, the Workbench's behavior), by providing language constructs that manipulate the results of these predicates. This includes, for example, a statement for defining projection definitions. Through rewrite rules these statements effect the answer to the visualization predicate (cpi:visualizeDescriptor).

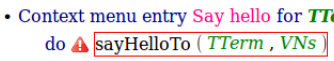

Figure 3.2: Cedalion code with an error

Another example is a statement that defines a context-menu entry. It manipulates the results of the query for context-menu entries for a certain selection (cpi:contextMenuEntry).

3. Implement the Cedalion type-system. The type-system is implemented as a collection of *checkers*, logic clauses contributing error messages in certain situations. Checkers are queried by the predicates facilitating the projectional editing, to provide decorations over the visuals. The most common kind of such decorations is an error marker, a red rectangle with a warning sign at its left. These inform the user that the code is invalid. Recall that the type-system (a provider of such error markers) is used for enforcing a schema for DSLs. Another kind of marker, also comming from the type-system, is a tooltip associated with variables. These tooltips display the inferred type for each variable. Figure 3.2 shows Cedalion code from the Hello, World example in Figure 1.1, this time where the "Say hello to" (or sayHello, as it appears here) procedure is not defined. The screenshot shows the error marker around the undefined concept.

4. Implement basic language constructs and basic languages to allow users to start programming in Cedalion. This includes the procedural programming constructs, and some simple language constructs for handling sets.

3.1.2.4 User DSLs and Programs

With the base-code (the *bootstrap* package) in place, users have all they need to start working. Typically, this will start with defining DSLs, and continue to implementing the software using these DSLs. However, following the Cedalion philosophy, there is no strict destinction between the two. As in traditional programming, where programming can be seen as defining more and more abstractions (functions, classes, etc), where some of them (e.g., the *main* function) provide an interface to the outer world, Cedalion programming is also about building abstractions one on top of the other. Some of these abstractions can be conveniently called DSLs, while others are "simple" extensions, such as procedures, functions and predicates, which also exist in traditional programming languages. As in traditional programming, these abstractions are worthless unless they have a real-life meaning. They need to interface to something in the outer world. This can either be the CPI (if the program is intended to run from within the Cedalion Workbench), or some other public interface, designed as the interface between the Cedalion program and some dedicated client.

3.2 Functionality

In Section 3.1 we discussed the "anatomy" of the Cedalion Workbench. In this chapter we shall discuss its "physiology", how it does the things it does. We start by describing the main design principle: the client/server approach. Then we revisit the functional requirements defined in Section 2.3, and provide the design behinds them.

3.2.1 Design Principles: Client/Server Approach

Cedalion is a pure logic programming language. This means that code in Cedalion can only provide results to queries (answer questions), but by itself, it cannot *do* anything. It cannot display anything on the screen, and cannot write anything to the disk. It cannot even change the contents of its own logic database (as do the predicates *assert* and *retract* in Prolog). Cedalion's way of doing things is related to its client/server approach.

3.2.1.1 Client/Server Examples

As mentioned in Chapter 3.1, Cedalion is based on a client/server approach. The Cedalion Workbench is one example for this approach (where the client is an Eclipse plug-in), but definitely not the only one. For practical applications, Cedalion code should be deployed in a client/server architecture as well. The server side remains the Cedalion program running on top of a Cedalion logic engine, but the client side can differ from one application to the other.

One possible client is a client for GUI applications. Such a client should have the capabilities to open windows and interact with the user, but should have no knowledge on the actual application that needs to run. The same client can thus be used for different applications. The client in this case shall make queries to the server to receive the layout of the windows to open, and what to do on user events.

Another example is supporting web applications. For these, the "client" is actually a web server (e.g., a Java Servlet). It is a relatively simple and generic server, capable of transforming HTTP requests into Cedalion queries, and transforming the answer to these queries into HTML. The same web server (the "client" in Cedalion's terms) can be used for various applications. The (Cedalion-level) server provides the actual application logic.

3.2.1.2 Procedures and Commands

Regardless of the nature of the client, it is sometimes required to perform actions. For example, the web-server acting as a Cedalion client may often be requested to add records to a database (for data-driven web applications). In Cedalion, we use procedures and commands for this purpose.

Commands are terms understandable by the client, which stand for performing some action. A command can be for adding a record into the database,

openning a window, saving some data to a file, etc. Commands should have all the information needed to perform their action, such as the name of the table and the data to insert, or the name of the window to open, which in turn can be used to query its contents. Commands can be compound, including other commands. For example, the *doAll* command supported by the Cedalion Workbench contains a list of commands and performs them one by one.

Procedures are higher level entities, still representing actions to be performed. Unlike commands, procedures are understood by the Cedalion program, and not the client. The Cedalion program can translate a procedure into a command using the *procedureCommand* predicate. Implementing this (mainly through *procedure* statements) provides a way for procedural programming in Cedalion. Procedures can call one another through the *doProc* command, which runs a procedure by querying the server (the Cedalion program) for the underlying command, and then performing it. This creates a dialog between the client and the server.

3.2.2 Requirements Walkthrough

3.2.2.1 Projectional Editing

Cedalion's projectional editing is implemented in parts at both the client and the server sides. The client/server protocol for supporting projectional editing defines the term *descriptor*, to refer to a handle to a part of the code, one that can be projected to a view, and in most cases, modified. A "normal" descriptor (/bootstrap:descriptor), one representing a part of the code being edited, contains a *path* to the code element. A path consists of the file name and a list of indexes representing the path that needs to be taken in the abstract syntax tree (AST) of the code, in order to reach that code element. This is a unique identifier for a code element. Given a path, the current contents can be queried and modified. Cedalion's bootstrap package provides a predicate (/*bootstrap*:termAtPath) for querying the contents of a code element (given a path), and a procedure (/bootstrap*:*setAtPath) for modifying its contents. By providing the path, a descriptor allows the projectional editing mechanism in Cedalion to edit code elements.

The CPI includes the predicate *visualizeDescriptor*, which provides the visualization for a given descriptor. This predicate is Cedalion's entry point for the implementation of its projectional editing. The implementation fetches the underlying code element, consults its projection definitions to find an appropriate one (and uses a default projection if none is found). When doing so, it relpaces the child elements with descriptor for the child elements, to allow them to be visualized as well. The result provided by the Cedalion program is a term representing visuals.

On the client side, each node in the returned term is turned into a Java object representing some figure to be displayed. Cedalion uses *draw2d* for visualization, and all visualization objects are derived from draw2d's *Figure* class. Cedalion has a mechanism for converting terms into Java objects, by querying Cedalion's

cpi:termClass predicate. This predicate matches a Java class to a term. The client can assume that the class name does not change, and therefore the results are cached on the client side.

3.2.2.2 The Text Bar

The text bar is implemented in part as part of the *CedalionWidget* (of package net.nansore.cedalion.eclipse), which is used as the editor control by the *CedalionEditor* (of the same package). When a *VisualTerm* (package net.nansore.cedalion.figures) is selected, it takes over the text bar. It first replaces the content of the text bar with a textual representation of the term represented by the *VisualTerm* object. This representation is provided by calling the procedure *cpi:termAsString*, which unifies a given variable with the term at the given position as a string.

The text bar is editable by the user. The active *VisualTerm* listens to all keystrokes. When the user hits Enter, the active *VisualTerm* calls the procedure *cpi:editFromString* to replace the content represented by this *VisualTerm* with the result of parsing the text in the text box using Prolog's syntax.

The *cpi:termAsString* and *cpi:editFromString* procedures use the *termToString* and *stringToTerm* commands respectively, which are implemented by calling Prolog predicates of these names, implemented in *service.pl*.

3.2.2.3 Auto-Completion

The Cedalion Workbench uses Eclipse's built in mechanism for auto-completion. Cedalion provides an adapter (class *CedalionProposalProvider*, internal to *CedalionEditor* in package net.nansore.cedalion.eclipse) to provide the completion proposals. This class calls the selected *VisualTerm* to retrieve the proposals (method *getProposals()*). It performs a query to the *cpi:autocomplete* predicate. This predicate takes the prefix – the characters to the left of the caret in the text box, as a filter on the results. It also takes the code element path, to allow it to retrieve the type of the requested concept, as another filter.

The bootstrap package of the Cedalion program implements the predicate retrieving the auto-completion options. It uses an internal predicate for retrieving all concepts for the current type, and another predicate to check the aliases of each concept. These aliases are matched against the prefix entered by the user. If the prefix is a prefix of the alias, the option is displayed.

Auto-completion is not only made for replacing content; it is also made for inserting content. The suggestions for auto-completion take the current content of the selected code element into account, and tries to place it as the first argument of each suggestion, if that fits. Doing so provides Cedalion users a possibility to edit terms, not just from the outside in (as would be the case where each entered concept is a new one), but also from the inside out. For example, to enter the expression $X+2*Y$, one could start by entering "X", then "+" and selecting the appropriate concept from auto completion, then select the empty right-hand operand of the "+", and enter "2", then enter "*" and select

the desired concept, and finally – select the right-hand operand of the "*" and enter "Y".

3.2.2.4 Aliases

Aliases are implemented entirely in Cedalion. They are answers to the /bootstrap:aliasString predicate. This predicate has a clause providing each concept its default alias: the name without the namespace prefixing. More results are provided using a rewrite rule (see Section 3.3 of [6]) translating *alias* statements into clauses of the *aliasString* predicate. The *alias* statement allows DSL developers provide custom aliases for their DSLs' concepts.

Additional rewrite rules provide aliases based projection definitions. They rewrite certain forms of projection definitions into *alias* statements. These forms include:

- Projection definitions where the projection is a label, where the label string is taken as the alias.
- Projection definitions which specify a horizontal flow, where either the first or second element is a label. The label string is taken as the alias.

3.2.2.5 Context Menu

The context menu is displayed by the *VisualTerm* class (package net.nansore.cedalion.figures), as a response to a right click. The private method *createContextMenu()* creates and displays the context menu. The construction of the context menu is done by calling the *cpi:contextMenuEntry* predicate, implemented in Cedalion. The arguments to this predicate are the descriptor associated with the *VisualTerm* object (providing its location), and an unbound variable to retrieve a term describing the menu item to display, including the procedure to be executed once the .

Then a popup menu is constructed for each result. Each entry is built by instantiating a *CedalionMenuItem* (package net.nansore.cedalion.eclipse), which builds the menu item, and registers the associated action. This mechanism is polymorphic, so that other classes can be implemented to provide other kinds of menu items, such as groups (which are not currently implemented). The *CedalionMenuItem* class associates a callback with each menu item, one that executes the associated procedure.

On the Cedalion side, the *cpi:contextMenuEntry* predicate is implemented with a rewrite from the |*bootstrap:contextMenuEntry* statement. This statement provides the caption to be written on the menu item, a pattern to be matched against the part of the code which is right-clicked (the "context"), place-holders for the path (the location of that piece of code), and the variable name assignment, and the procedure to be executed. A newer version of this statement also takes an image identifier, for the icon to be presented besides the caption. This statement is used extensively in Cedalion's bootstrap code, and is also used in libraries.

3.2.2.6 Adapters

Adapters are implemented solely in Cedalion. The entry point is the */bootstrap:checkAdapter* predicate. A rewrite rule contributes to this predicate for every */bootstrap:adapter* statement.

Adapters are used by the bootstrap package in two places: For autocompletion, and for resolving type mismatches automatically.

In autocompletion, the *autocomplete* predicate consults the *checkAdapter* predicate to expand the search beyond the needed type, to also include concepts that can be placed in that position wrapped in an adapter. This makes editing much easier in some cases. In addition, when considering the existing content as the first argument for the new content, adapters are taken into account as well.

When a type mismatch occurs (either by entering code manually, or by copying-and-pasting code into a certain location), Cedalion tries to resolve this error automatically. Adapters are considered for the resolution of such errors.

3.2.2.7 Definition Search

Definition searches are also implemented in Cedalion. The context menu entry providing the search runs the */bootstrap:doShowDefinitions* procedure. That procedure is defined for every non-variable, and calls */bootstrap:showDefinitions*. The reason for this split is to make the creation of the context menu faster, by not waiting to actually build the output when the menu is constructed.

The *showDefinitions* procedure builds a visualization term containing all the definitions of the concept it takes as argument. Then it displays it in the Cedalion view, by calling the *showView* command, implemented by the *ShowView* class (package net.nansore.cedalion.cmd).

Bibliography

[1] M. Bachle and P. Kirchberg. Ruby on Rails. *IEEE SOFTWARE*, pages 105–108, 2007.

[2] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[3] M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[4] Martin Fowler. Projectional editing. Martin Fowler's Bliki. `http://martinfowler.com/bliki/ProjectionalEditing.htmlx`.

[5] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[6] David H. Lorenz and Boaz Rosenan. Cedalion: a language for language oriented programming. *SIGPLAN Not.*, 46:733–752, October 2011.

[7] Tim Menzies. DSLs: A logical approach, 2001. Lecture Notes, EECE 571F, `http://courses.ece.ubc.ca/571f/lectures.html`.

[8] Boaz Rosenan. Designing language-oriented programming languages. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 207–208, New York, NY, USA, 2010. ACM.

Appendix A

Communication Protocols

A.1 The Cedalion Public Interface (CPI)

The list of concepts in the CPI is given in Figures A.1 and A.3. Please note that they are given in arbitrary order.

A.2 Built-in Predicates

Built-in predicates are given in the "builtin" namespace. This namespace also contains other concepts used by these predicates. Figures A.3, A.4 and A.5 provide a listing of all concepts in the builtin namespace.

A.3 Service Predicates

The following are Prolog predicates that provide services used by the Eclipse plug-in.

insert(Statement) Adds the given *Statement* to the logic database.

remove(Statement) Removes the given *Statement* from the logic database.

generateFile(FileName, StringVar, Goal) Generates a file named *FileName*, containing a line per each result of *Goal*. The content of the line is the string bound to *StringVar*.

readFile(FileName, Namespace, FileContent) Reads a Cedalion source file named *FileName*, and binds its content into *FileContent*. *Namespace* is the default namespace to be used when reading the file.

writeFile(FileName, FileContent) Writes *FileContent* into a Cedalion source file named *FileName*.

namespace **cpi**

- *// Provide auto-complete solutions for TTerm, where the user has already entered Prefix. CompletionString is to contain the full completion term, and Alias will contain the alias to be displayed*
  autocomplete (*TTerm* , *Prefix* , *CompletionString* , *Alias*) :: pred ↪ [*TTerm* :: typedTerm , *Prefix* :: string , *CompletionString* :: string , *Alias* :: string]
- *// Succeeds for every context menu entry provided for Descriptor. MenuItem is bound to a description of the menu item to display*
  contextMenuEntry (*Descriptor* , *MenuItem*) :: pred ↪ [*Descriptor* :: typedTerm , *MenuItem* :: menuEntry]
- *// Deprecated. Moved to the /bootstrap namespace*
  redo (*Res*) :: procedure ↪ [*Res* :: string]
- *// Load a file named FileName into memory. ResourceName is to be used for the paths, and Namespace is the default namespace to be used*
  openFile (*FileName* , *ResourceName* , *Namespace*) :: procedure ↪ [*FileName* :: string , *ResourceName* :: string , *Namespace* :: string]
- *// Close a file: remove its contents from memory*
  closeFile (*ResourceName*) :: procedure ↪ [*ResourceName* :: string]
- *// A simple procedural expression that returns the Constant it is given*
  const *Const* :: expr (*T*) ↪ [*Const* :: *T*]
- *// Save a file to the disk. Given are both the ResourceName by which it is represented in memory, and the FileName to be used on the disk*
  saveFile (*ResourceName* , *FileName*) :: procedure ↪ [*ResourceName* :: string , *FileName* :: string]
- *// Defines procedure Proc, by associating it to Command*
  procedure pred *Proc* :: pred ↪ [*Proc* :: procedure , *Command* :: command]
  Command
- *// This is how a procedural function looks as a procedure*
  func *T Result* = *Expr* :: procedure ↪ [*Expr* :: expr (*T*) , *Result* :: ref (*T*) , *T* :: type]
- *// Associating a concept (represented by TTerm) with a Java class*
  termClass (*TTerm* , *ClassName*) :: pred ↪ [*TTerm* :: typedTerm , *ClassName* :: string]

Figure A.1: CPI Listing (part I)

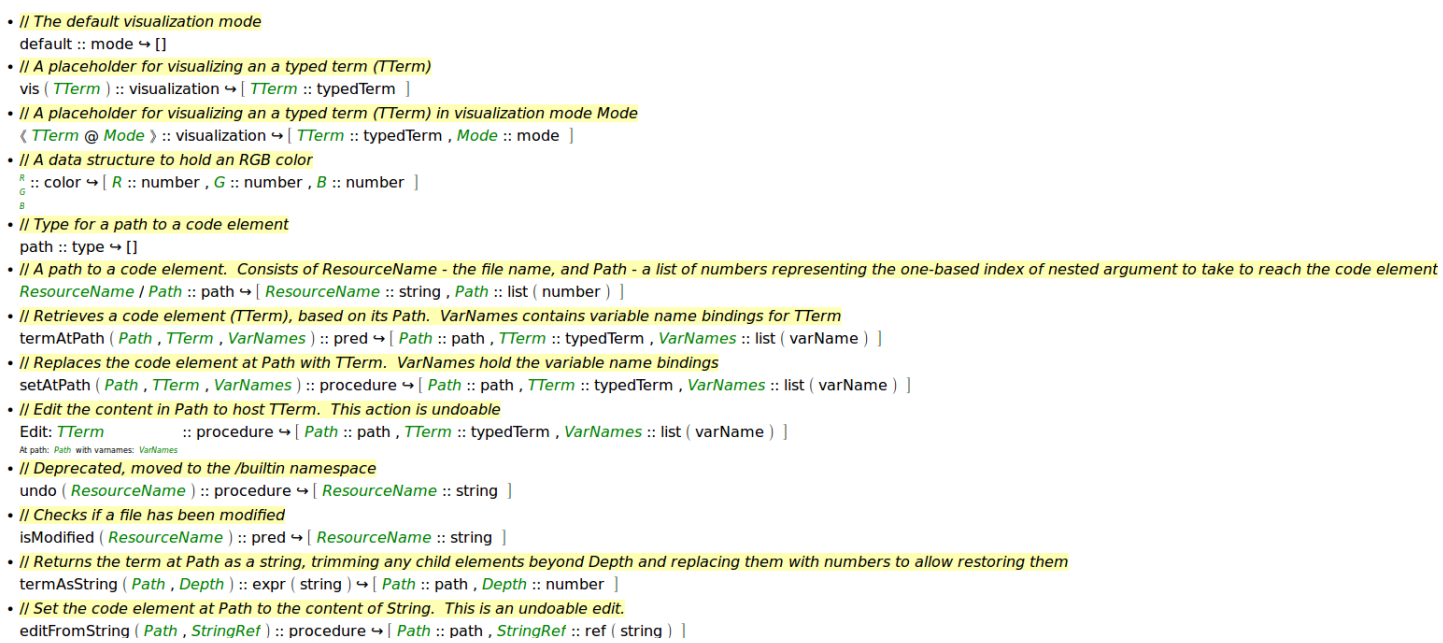

Figure A.2: CPI Listing (part II)

namespace **builtin**

- *// Succeeds if A > B*
  greaterThen (*A* , *B*) :: pred ↪ [*A* :: number , *B* :: number]
- *// Succeeds if C equals (numerically) AV+BV*
  plus (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Succeeds if C equals (numerically) AV-BV*
  minus (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Succeeds if C equals (numerically) AV*BV*
  mult (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Succeeds if C equals (numerically) AV/BV (real division)*
  div (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Succeeds if C equals (numerically) AV/BV (integer division)*
  idiv (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Succeeds if C equals (numerically) AV mod BV*
  modulus (*AV* , *BV* , *C*) :: pred ↪ [*AV* :: number , *BV* :: number , *C* :: number]
- *// Is TTerm a compound typed-term?*
  compound (*TTerm*) :: pred ↪ [*TTerm* :: typedTerm]
- *// Succeeds at Nominator out of Denominator times*
  coinToss (*Nominator* : *Denominator*) :: pred ↪ [*Nominator* :: number , *Denominator* :: number]
- *// Codes are the ASCII codes of Str*
  charCodes (*Str* , *Codes*) :: pred ↪ [*Str* :: string , *Codes* :: list (number)]
- *// Z is a concatenation of strings X and Y*
  strcat (*X* , *Y* , *Z*) :: pred ↪ [*X* :: string , *Y* :: string , *Z* :: string]
- *// Parse a typed term (TTerm) into a Name and a list of typed arguments (TArgs). Can also be used to construct a typed term out of a name and typed args. This predicate is not type-safe. Use safeParseTerm instead.*
  parseTerm (*TTerm* , *Func* , *TArgs*) :: pred ↪ [*TTerm* :: typedTerm , *Func* :: string , *TArgs* :: list (typedTerm)]
- *// Calculate the next of previous integer*
  Index = *IndexMinusOne* +1 :: pred ↪ [*IndexMinusOne* :: number , *Index* :: number]

Figure A.3: builtin Namespace Listing (part I)

- *// Conditional predicate. If Cond succeeds, Then applies. Otherwise Else applies.*
  if *Cond* : :: pred ↪ [*Cond* :: pred , *Then* :: pred , *Else* :: pred]
  Then
  else:
  Else
- *// A representation for the contents of a generated file for the writeFile command. For every result of Goal, AnnotatedStatement represents a statement to be written to file*
  fileContent (*AnnotatedStatement* , *Goal* , *NSList*) :: fileContent ↪ [*AnnotatedStatement* :: annotatedTerm , *Goal* :: pred , *NSList* :: list (nsElem)]
- *// A data structure to hold a list of statements with variable name bindings, and a list of namespace aliases*
  fileContent (*Terms* , *NsList*) :: fileContent ↪ [*Terms* :: list (annotatedTerm) , *NsList* :: list (nsElem)]
- *// A data structure to hold a pair of a statement along with its variable name bindings*
  statement (*S* , *VN*) :: annotatedTerm ↪ [*S* :: statement , *VN* :: list (varName)]
- *// A data structure to hold a single typed variable (TVar) along with its Name*
  varName (*TVar* , *Name*) :: varName ↪ [*TVar* :: typedTerm , *Name* :: string]
- *// Checks if two typed terms are equal. Variables are equal if they are already bound to each other*
  TVar1 == *TVar2* :: pred ↪ [*TVar1* :: typedTerm , *TVar2* :: typedTerm]
- *// An import statement. Binds a namespace name to an alias (deprecated)*
  import *NS* as *Alias* :: statement ↪ [*Alias* :: string , *NS* :: string]
- *// A goal that always succeeds*
  true :: pred ↪ []
- *// List is a list of elements such as Element, for every result of Goal*
  Find all *Element* of type *Type* such that *Goal* into *List* :: pred ↪ [*Element* :: *Type* , *Type* :: type , *Goal* :: pred , *List* :: list (*Type*)]
- *// Creates TTermCopy, a copy of TTermOrig, such that they are structurally equal, but have different variables.*
  copyTerm (*TTermOrig* , *TTermCopy*) :: pred ↪ [*TTermOrig* :: typedTerm , *TTermCopy* :: typedTerm]
- *// Succeeds if TTerm1 and TTerm2 have the same structure, but potentially different variables*
  structurallyEqual (*TTerm1* , *TTerm2*) :: pred ↪ [*TTerm1* :: typedTerm , *TTerm2* :: typedTerm]

Figure A.4: builtin Namespace Listing (part II)

- *// Succeeds if TTerm matches Var::Type, where Var is an unbound variable*
  var (*TTerm*) :: pred ↪ [*TTerm* :: typedTerm]
- *// Succeeds if TTerm matches Str::Type, where Str is a Cedalion string*
  string (*TTerm*) :: pred ↪ [*TTerm* :: typedTerm]
- *// Succeeds if TTerm matches Num::Type, where Num is a number*
  number (*TTerm*) :: pred ↪ [*TTerm* :: typedTerm]
- *// Perform unification of TTerm1 and TTerm2, not allowing variables to be unified with terms containing them*
  safeUnify (*TTerm1* , *TTerm2*) :: pred ↪ [*TTerm1* :: typedTerm , *TTerm2* :: typedTerm]
- *// Throw an exception*
  throw (*Exception*) :: pred ↪ [*Exception* :: exception]
- *// A goal that never succeeds*
  fail :: pred ↪ []
- *// Succeeds if Goal succeeds, and if no exceptions have been thrown from it. If an exception matchin Exception has been thrown, AltGoal is evaluated.*
  try: :: pred ↪ [*Goal* :: pred , *Exception* :: exception , *AltGoal* :: pred]
    Goal
  catch *Exception* :
    AltGoal
- *// Succeeds if TTerm contains no unbound variables*
  ground (*TTerm*) :: pred ↪ [*TTerm* :: typedTerm]
- *// Succeeds for all Statements loaded from file FileName. VarNames is bound to a list of variable name bindings*
  loadedStatement (*FileName* , *Statement* , *VarNames*) :: pred ↪ [*FileName* :: string , *Statement* :: statement , *VarNames* :: list (varName)]

Figure A.5: builtin Namespace Listing (part III)

stringToTerm(String, NSList, Term, VarNames) Converts *String* into *Term*. It uses the namespace bindings in *NSList*, and binds the variable name bindings to *VarNames*.

termToString(Term, VarNames, Depth, NSList, String) Converts *Term* into *String*. It trims all sub-terms deeper then *Depth*. *VarNames* are assumed to contain the variable name bindings to be used, and *NSList* contains the namespace aliases.

loadFile(FilePath, Namespace) Loads a file named *FilePath* into the logic database. It uses *Namespace* as the default namespace for the file. If the file has already been loaded, its previous content will first be removed.