



# Towards Project-Based Workflows in Twelf

*Author:*
Alin IACOB

*Supervisors:*
Prof. Dr. Michael KOHLHASE
Dr. Florian RABE

A thesis submitted for the conferral of a *Master of Science*
in Smart Systems

August 2011

## Acknowledgements

I would like to thank my supervisors, Prof. Dr. Michael Kohlhase and Dr. Florian Rabe, for their excellent guidance during the past 2 years.

I would also like to extend a warm thank you to my family and close friends, who offered me unconditional encouragement and support.

Finally, I am greatly indebted to Prof. Dr. Michael Kohlhase and Prof. Dr. Dierk Schleicher for their understanding while helping me overcome a critical moment of my life.

## Declaration

This thesis is submitted in partial fulfillment of the requirements for receiving a Master of Science degree in Smart Systems at Jacobs University Bremen. The research presented in this work has been conducted under the supervision of Prof. Dr. Michael Kohlhase and Dr. Florian Rabe. All material in this Master Thesis is my own, unless otherwise stated. References to and contributions from other sources are explicitly marked and acknowledged.

I hereby certify that this work contains original and independent results and has not been submitted elsewhere for the conferral of a degree.

Alin Iacob
Bremen, 23rd August 2011

# Towards Project-Based Workflows in Twelf

Author: Alin Iacob

Supervisors: Prof. Dr. Michael Kohlhase, Dr. Florian Rabe

A thesis submitted for the conferral of a *Master of Science*
in Smart Systems

August 2011

# Contents

# Chapter 1

# Introduction

Mathematical Knowledge Management (MKM) systems represent formal knowledge in languages suitable for machine processing, such as XML. Some systems aim to achieve interoperability with others or even act as a knowledge interchange format.

Since interchange languages often become too verbose for human processing, MKM systems extensively use domain-specific languages for human input, which are then compiled into the interchange format. The compiled representations are then accessed by software services.

A problem with these compilers is that they often follow a functional approach: they take a predefined block of code as input and produce the compiled output, without any side effects. This poses two conceptual problems.

The first one is that the compiler does not have access to project-level information, which constrains the language by making it mandatory that identifiers from different compilation units are referenced by their physical location. This precludes the introduction of pure namespaces as a language feature, because, for each external symbol that needs to be imported, it is the author's task to specify the physical location of its enclosing namespace block, which defeats the purpose of using namespaces.

A second problem stemming from the functional approach is the absence of a permanent state with which the users can interact. Such interaction is desirable in many scenarios. For example, an IDE can provide autocompletion, search services and an outline view, all of which require a maintained state. Without one, each user query requires many files to be re-compiled.

Solving these problems involves switching to a project-based setting. The state of a project is best represented using the object-oriented concept of a class, with methods implementing various services.

Once in a project-based framework, various dimensions of knowledge can be managed alongside the compiled sources, such as a presentation dimension containing the documentation, or a relational dimension with predicates expressing $n$-ary relationships in the source data.

In this work, we restrict our attention to the Twelf framework, a system for modular encodings of logics and mathematical theories. Twelf is an excellent use case for our research. At the start of this work, Twelf documents were developed manually, with no tool support except the feedback given by the compiler. Compilation, the only automated workflow, was file-based and, despite modularity, all module names resided in a single global namespace. For small projects, this did not present a challenge. However, larger projects such as the Atlas library [CHK+11], an large graph of interrelated encodings, were experiencing serious usability and scalability problems.

Our work introduces the infrastructure for project-based authoring workflows in Twelf, motivated by the needs associated with the increase in complexity of the Atlas library. To this end, we established

a set of guiding research questions.

(Q1) Introduce namespaces in Twelf and build a project-based catalog service to assist in compilation. Find what kind of indexing is needed to support the compiler, as well as the infrastructure needed to maintain the index. Determine the best interface or set of interfaces for communicating with the compiler.

(Q2) Determine how resources should be identified in order to permit global collaboration. This question was already solved abstractly in MMT, a module system for mathematical theories which Twelf borrows its modularity from. The solution are namespaces encoded as globally unique URIs.

(Q3) Devise and automate a packaging and distribution method for Twelf projects, and its internal file structure.

(Q4) Find which other knowledge dimensions are suitable for inclusion in Twelf projects, and their uses in providing services.

(Q5) Enable IDE authoring services for Twelf.

The contribution of this work can be summarized as follows:

1. We enable project-based workflows in Twelf by implementing namespaces, along with a catalog service that critically assists the file-based Twelf compiler in the new project-based setting.

2. We refactor the Atlas library by introducing namespaces, de-cluttering the naming scheme and file hierarchy and pruning the obsolete parts of the tree.

3. We provide a reference implementation for mathematical archive management and project build workflows.

4. We implement an error-resilient parser for Twelf in the SIDER framework (Appendix B).

Chapter 2 is a theoretical exposition of MMT, Twelf and the Atlas. The main part of the work is Chapter 3, which details our contributions from a theoretical perspective. Chapter 4 describes the catalog service, which is at the core of the research developments.

# Chapter 2

# Formalizing Knowledge in Twelf

The Edinburgh Logical Framework (LF) [HHP93] is a type theory, closely based on dependently typed $\lambda$-calculus [Bar92], that provides a means to encode logics and other formal knowledge. Logical encodings in LF follow the Curry-Howard isomorphism: axioms and judgements are represented as types and proof are represented as terms. Thus, the question of finding a proof for a judgement $A$ is the same as finding a term of the given type $A$.

The Twelf system [PS, PS10] is a framework that specifies a concrete language for LF encodings and provides a compiler which, among other functions, checks and infers the type of each symbol. Twelf, which is itself derived from the Elf system [Pfe91], was further enhanced by adding modularity [RS09], opening up a multitude of applications. The modularity concept in LF constitutes the main use case of MMT [RK11, Rab11], a foundation-independent scalable module system.

We offer an overview of the MMT project in Section 2.1, and explain the Twelf system in Section 2.2. Finally, Section 2.3 discusses the most important use case of Twelf, the LATIN Logic Atlas [CHK+11], in which we evaluate our work.

## 2.1 MMT

The MMT Module system for Mathematical Theories [RK11, Rab11] is a formal language designed to combine a module system, a foundation-independent semantics, and scalability. Its concrete syntax extends the OMDoc language [Koh06] with modularity, by grouping definitions in theories and offering a way to express translations between two theories. The concept carries some similarities to object-oriented design in software engineering, and is motivated by the success of the "little theories" approach or organizing mathematics, popularized by the encyclopedic work of Nicolas Bourbaki [Bou74].

MMT organizes knowledge in a theory graph, where nodes represent theories and edges, also known as *theory morphisms* or *links*, indicate translations between theories. Four levels of abstraction can be distinguished in the MMT architecture:

**document level** A document is a sequence of modules.

**module level** Theories are sequences of constant or structure declarations, while views are sequences of constant or structure assignments.

**symbol level** Constant declarations introduce new constants, axioms, lemmas, or definitions. Structure declarations import a translated version of another theory; in the special case when the

translation is the identity function, they are called *theory inclusions*. Constant or structure assignments specify a mapping from symbols of the domain theory to terms over the codomain theory; in the special case when a mapping from a structure (defined in the domain) to the codomain is already defined somewhere else, it can be imported using a so-called *morphism inclusion*. Structures and views belong to the unitary concept of a *link*.

**object level** There two kinds of objects in MMT: terms and morphisms. Atomic MMT terms are constant references, variables, as well as the special objects $\bot$ and $\top$. New terms can be composed from existing ones by application, binding, attribution, and morphism application. Atomic MMT morphisms are references to links, or the identity morphism $\texttt{id}_T$, for any theory $T$. New morphisms can be composed from existing ones by morphism composition.

Each module name, symbol declaration and symbol assignment is uniquely and systematically identified by an absolute MMT URI, which is a valid absolute URI as defined by [BLFM05], but without a fragment part. There are three kinds of absolute MMT URIs:

**document URI *D*** references a document or a general module container and does not contain a query part.

**module URI *M*** references a theory or a view and is of the form *D*?*modName*, where *modName* is the name of the module.

**symbol URI *S*** references a symbol declaration or assignment and has the form *M*?*symbID*, where *symbID* is the identifier of the symbol. *symbID* is of the form $part_1/\ldots/part_n$, where each part contains valid URI characters excluding whitespace, "/", "?", "#", "]", "]" and "%". The character "/" between identifier parts separates structure names. For example,

```
http://latin.omdoc.org/math?Group?multiplication/*
```

refers to the $\star$ constant imported in theory `Group` via the structure `multiplication`.

The next section discusses modular Twelf, the most well-developed use case of MMT.

## 2.2 Twelf

LF is a $\lambda$-calculus with dependent types and, in the framework of the $\lambda$-cube [Bar80], it is also known as $\lambda P$. Thus, types may depend on terms, but terms cannot depend on types. At its object-level, LF expressions are grouped into kinds, types and terms:

$$\begin{aligned} \text{kinds } K &::= \mathit{type} \mid \{x:A\}\ K \mid A \to K \\ \text{types } A,B &::= a \mid A\ M \mid \{x:A\}\ B \mid A \to B \\ \text{objects (terms) } M &::= c \mid x \mid [x:A]\ M \mid M_1\ M_2 \end{aligned}$$

where $\{\cdot\}$ is $\Pi$-abstraction and $[\cdot]$ is $\lambda$-abstraction.

A Twelf document has the extension `.elf` and consists of a sequence of *signatures* and *views*.

**Identifiers** are dot-separated sequences of identifier parts. The two production defining them are

```
identifierPart ::= identifierPartCharacter+
identifier   ::= identifierPart ('.' identifierPart)*
```

where `identifierPartCharacter` is any Unicode character except any of `.:()[]%"` and space characters.

**Comments** are of three types: single-line, multi-line and semantic. The first two are ignored by a compiler, while the last provides metadata information which can be later used for providing documentation and other semantic services.

A semantic comment is, abstractly, a set of key-value pairs, where two notable keys are `short`, denoting a one-line description, and `long`, denoting the full version of the comment, which typically spans several lines. The keys can have arbitrary other names. The first line of the comment constitutes the `short` description and the following lines constitute the `long` description. However, from the first line that starts with `@` onwards, each line encodes a key-value pair (the `commentProperty` rule below).

In this listing, `[^\s@]` resolves to a single character whish is not a space character or `@`.

```
singleLineComment ::= '%' ((lineWhiteSpaceCharacter | '%') lineCharacter*)?
                       lineDelimiter

multiLineNonSemanticComment ::= '%{' .* '}%'

commentShortForm ::= [^\s@] lineCharacter*
commentLongForm ::= (lineWhiteSpaceCharacter*
                     ([^\s@] lineCharacter*)? lineDelimiter)*
                    lineWhiteSpaceCharacter* ([^\s@] lineCharacter*)?
commentPropertyValue ::= ([^\s] (lineCharacter* ))?
commentProperty ::= '@' identifier lineWhiteSpaceCharacter+
                    commentPropertyValue
multiLineSemanticComment ::= '%*' lineWhiteSpaceCharacter* commentShortForm?
                             (lineDelimiter commentLongForm)?
                             (lineDelimiter commentProperty)*
                             [^\s]* '*%'
```

**Read declarations** are akin to preprocessor include directives in C. They tell the Twelf compiler to replace the declaration with the contents of another file before continuing.

```
readDeclaration ::= '%read' '"' fileURI '"' '.'
```

**Terms** At the object level, Twelf entities are grouped into kinds, types and terms:

$$\begin{aligned} \text{kinds } K &::= \text{type} \mid \{x : A\}\ K \mid A \to K \\ \text{types } A, B &::= a \mid A\ M \mid \{x : A\}\ B \mid A \to B \\ \text{objects (terms) } M &::= c \mid x \mid [x : A]\ M \mid M_1\ M_2 \end{aligned}$$

where $\{\cdot\}$ is $\Pi$-abstraction and $[\cdot]$ is $\lambda$-abstraction.

A document can contain declarations of type and object constants:

$$\begin{aligned} \text{type declaration } & a : K. \\ \text{const declaration } & c : A. \end{aligned}$$

or definitions:

$$\text{type definition } a : K = A.$$
$$\text{const definition } c : A = M.$$

However, Twelf kinds, types and terms all correspond to MMT terms, where there is no such distinction. Therefore, for simplicity, we use the same grammar production in all cases, which we call "term".

```
term ::= 'type'
       | identifier
       | '(' sc? term sc? ')'
       | term sc '->' sc term
       | term sc '<-' sc term
       | '{' sc? identifierPart (sc? ':' sc? term)? sc? '}' sc? term
       | '[' sc? identifierPart (sc? ':' sc? term)? sc? ']' sc? term
       | term sc term
       | '(' sc? term sc? ':' sc? term sc? ')'
       | '_'
```

**Signatures** , or theories, are lists of *symbol declarations* and *structures*.

```
theory ::= '%sig' identifierPart '=' '{'
          (sigMetaDeclaration | theoryIncludeDeclaration | structDeclaration
           | termDeclaration)* '}' '.'
```

**Symbol declarations** are at the core of the Twelf language. They introduce new well-typed symbols.

```
termDeclaration ::= identifier (':' term)? ('=' term)? '.'
```

The first optional term is the type and the second is the definiens. Unlike MMT, at least one of the two optional terms must be provided. The type is optional as long as Twelf can infer it from the definition.

**Structures** are a type of *links*, along with views, and have the simplest form

```
simpleStructDeclaration ::= '%struct' identifierPart ':' identifier '='
    linkBody '.'
```

A structure is a translation from a domain signature, referenced by the identifier after the `:`, to the current signature. Semantically, it imports all the symbols from the domain, making them accessible by prepending the name of the structure to their name.

**Link bodies** are constructs that can appear as the definiens of both structures and views.

```
linkBody ::= '{' ( (linkMetaDeclaration | viewIncludeDeclaration
                  | constantAssignment| structAssignment))* '}'
```

The set of symbol names that can appear in the constant and struct assignments is a subset of the symbols with no definiens declared in the domain of the link. The symbols from the domain signature which are not assigned to anything in the are simply imported as-is into the current signature.

**Include** and **Meta** are syntactic sugar for structure declarations with empty body, where every symbol from the domain is imported as-is into the current signature. They can also appear inside link bodies, in which case they provide existing assignments to imports in the domain, obviating the need for defining complicated struct assignments directly in the body of the link.

```
includeDecl ::= '%include' identifier '.'
metaDecl ::= '%meta' identifier '.'
```

**Views** are a type of *links* along with structures. A view between two signatures has the (simplified) form

```
view ::= '%view' identifierPart ':' identifier '->' identifier '=' linkBody
    '.'
```

The first identifier after `:` is the domain and the next one is the codomain of the view.

Unlike structures, a *view* does not add symbols to a signature. Its role is independent from those of the structures. If we regard signatures as mathematical theories, where axioms are encoded as symbols according to the Curry-Howard isomorphism, then a *view* encodes an `is-a` relation between its domain and codomain.

Once we have a view from `domain` to `codomain`, every expression over the `domain` can be translated (via structural induction, using the view assignments for the atoms) into an expression over the `codomain`. Mathematically, any constructions or theorems of the `domain` theory can be automatically translated to constructions and theorems of the `codomain`.

We ommitted several topics, such as notation declarations (fixity declarations, alias lists), views and structures declared via a morphism, implicit views, spacing issues, complex codomains, morphisms and certain lexer rules. They are provided in the complete CFG grammar for Twelf, listed in Appendix A.

## 2.3 The LATIN Logic Atlas

The LATIN project [CHK+11] is a foundationally unconstrained framework for the development of logics and translations between them. A major part of the project is the LATIN Logic Atlas [KMR08], a library of interdependent LF encodings of logics, type theories, set theories and mathematical concepts, along with translations between them.

The Atlas is a use case for both MMT and modular Twelf, and contains formalizations of type theories, set theories, logics and mathematics. Among the logic formalizations in the Logic Atlas are

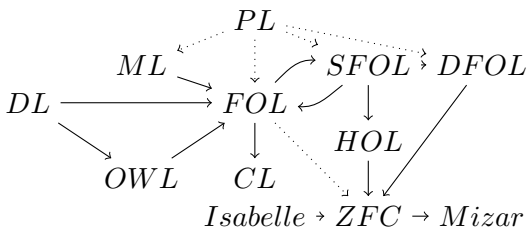


**Figure 2.1:** Logical Languages in the LATIN Logic Atlas

the formalizations of propositional ($PL$), first ($FOL$) and higher-order logic ($HOL$), sorted ($SFOL$)

and dependent first-order logic ($DFOL$), description logics ($DL$), modal ($ML$) and common logic ($CL$) as illustrated in Fig. 2.1. Continuous arrows denote translations between formalizations and dotted arrows denote imports. The figure only shows a small subset of the encodings since, for instance, it omits the modules for individual logical connectives. The entire library has more than 600 modules.

# Chapter 3

# Project-Based Workflows in Twelf

The authoring of Twelf content is best carried out in a project-based instead of file-based setting. At the start of this work, the Twelf workflow was file-based. Even though for small projects this did not present a challenge, once the LATIN graph grew larger and more interdependent, several shortcomings came into spotlight, some of them seriously impeding the authoring process.

An immediate shortcoming of the file-based workflow was related to external module references. A typical author opens an existing file in an editor with the intention of modifying a signature. The signature has several imports and structures referencing other modules. Unless the modules follow a naming scheme which somehow indicate the file in which they are declared, or the author is familiar with all the referenced modules, she has no direct way of knowing where an imported module is declared. Instead, she must first search through the modules previously declared in the file. If the module is not there, she has to open every needed file using `%read` declarations (the so called first-degree includes), search through them as well, and continue this search recursively. This situation is occurs almost always, since one of the goals of Twelf is high modularity and code reuse. For example, there are files in the LATIN graph which include, via `%read` declarations, tens of other files and more than 100 modules.

Another situation where the file-based workflow was falling short involves diamond situations, which occur when file A is included in B and C, and the last two are included in D. Even though the compiler treats this as one inclusion, an author who searches for included modules through a large graph of files can waste time by searching through the same file twice.

The resulting time cost of searching for modules is so high that a potential author can only do significant work after being fully accustomed to the structure and particular names of the files and modules that are being used.

Since modules are intended to be reused as much as possible, their names must not clash with any other module names that might be imported together with them in a third file. This requirement has led to extremely long and cumbersome names in the LATIN repository, such as `SEqualPF` (**p**roo**f** axioms for **s**orted **equal**ity) and `IFOLEQPFExt` (**ext**ended **p**roo**f** theorems for **i**ntuitionistic **f**irst-**o**rder **l**ogic with **eq**uality).

The module searching problem, as well as the name clutter problem can be solved with the addition of namespaces at the syntax level. Section 3.1 introduces them and explains why they require special catalog software to aid in authoring. We introduce the software solution for supporting namespaces, the Twelf catalog service, in Section 3.2. Section 3.3 discusses several knowledge dimensions that can be auto-generated in a project context, and their relevance for third-party tools, while Section 3.4 presents a distribution and packaging format for mathematical knowledge that subsumes the

previously discussed dimensions. Various added-value services can be implemented on top of the catalog and the project-based infrastructure; they are discussed in Section 3.5. Finally, Section 3.6 discusses the difficulties and outcome of the migration of the LATIN repository from the file-based to a project-based setting.

## 3.1 Namespaces

The introduction of namespaces in LF is the answer to the problems associated with a file-based workflow, outlined at the beginning of this chapter. Every document is required to have one or more *current namespace* declarations, using the syntax

`%namespace "`*N*`".`

*N* is either a URI or a relative URI reference (defined in RFC 3986 [BLFM05]), with no query or fragment.

Each such declaration is followed by a sequence of modules which are considered in the scope of the most recent *current namespace* declaration. Semantically, if a module *M* is in the scope of the namespace *N*, its MMT URI is *N*?*M*. The URIs of its children are prefixed by the URI of the module.

When *N* is a relative URI reference, it is resolved against a base namespace URI, determined using the following inductive algorithm:

**step** If this is not the first current namespace declaration in the document, the base URI is the absolute URI determined from the previous current namespace declaration.

**base** If this is the first current namespace declaration in the document, the base URI is externally specified; in the case of mathematical archive projects (Section 3.4), the base URI is given in the manifest.

While in the scope of a current namespace declaration, we need a mechanism for referencing modules in a different namespace. The syntactical construct is a *namespace prefix declaration*:

`%namespace` *pref* `= "`*N*`".`

where *pref* is a Twelf identifier part and *N* is a namespace URI, relative to the current namespace. From the point of this prefix declaration and until the end of the document, modules *M* in namespace *N* ca be referenced with *pref.M*. Since the names of the referenced modules can no longer clash with other local or referenced module names, they do not have to be unique. This greatly improves readability and eliminates the need for refactoring files with clashing names.

A person – or a compiler – who reads a file with prefixes pointing to other namespaces needs to know where the modules in those namespaces are physically located. This means knowing the file path and, for even more convenience, the start and end position within the file. This requirement highlights the need for an automated catalog service who can provide the translation from logical to physical locations, both to the human programmer and to the Twelf compiler.

## 3.2 The Role of the Catalog

The Twelf compiler is exclusively file-based: it compiles a file by reading and checking the declarations linearly. When encountering a `%read` statement, it reads and compiles the included file, adds its compiled declarations to an internal heap and continues compiling the initial file.

The namespace alias declarations, however, present a conceptual dead-end for the compiler, because the author-defined namespace hierarchy does not necessarily correspond to the physical, system-understandable file URLs. The synchronization of the two types of arrangements can be achieved only by tedious efforts, which is a strong motivation for automatizing it.

To make the problem worse, each file can contain references to any number of other namespaces, allowing for various kinds of inconsistencies, such as circular imports and hierarchy violations, which can be easily managed using automatic graph algorithms.

Therefore, we have implemented a standalone catalogue infrastructure that translates from MMT URIs (namespaces) to URLs (filenames). Let $U \subset URI$ be the set of MMT URIs and $F \subset URL$ be a set of URLs pointing to Twelf source files. Furthermore, we say that a MMT URI is *valid* over $F$ if $F$ contains a file which declares an MMT entity possessing that URI.

Abstractly, the catalog implements a partial function

$$f : U \to F \times \mathbb{N}^4$$

which, given a MMT URI $u$ valid over $F$, returns the URL of the file where $u$ is declared, together with the start and end of the declaration block, given as line and column indices.

Having access to the Twelf sources, the catalog also provides basic relational information about the Twelf entities, such as their meta information, the dependency list, or the list of children. This information is then requested by semantic IDEs for auto-completion, hovering tooltips or other services. Unlike the MMT software, which offers part of these services based on the compiled OMDoc files, the catalog is based on the Twelf sources, which makes it suitable for usage during the editing phase. During the authoring process, multiple source files are typically in an ill-formed state and do not compile, hence there are no OMDoc files that the MMT software can use. The catalog is designed to ignore many errors when parsing (especially the content of the proofs – see Section 4.10), hence it is the only candidate for offering semantic services during editing.

The catalog can also provide an OMDoc skeleton of a MMT entity or even an entire document, which can then be queried using XPath for the same information that the catalog provides via queries. The skeleton follows the OMDoc structure generated by the Twelf compiler, but notably omits terms, i.e. the right side of declarations and assignments. Terms could be included if necessary by performing a trivial translation from LF syntax to OMDoc, but their presence in the OMDoc skeleton would be of limited use, since unlike the compiler, the catalog does not perform any type inference. The advantage of saving OMDoc skeletons alongside the OMDoc files compiled by Twelf is that tools like MMT can extract information in a unitary way from the two sources.

Note that it is straightforward to build a catalog using the compiled OMDoc, which already contains a source location as a XML attribute for each Twelf entity. However, the catalog is already needed at compilation time, where the OMDoc is not yet available.

## 3.3 Compilation and Building

In the context of our project-based approach, it is useful to store different interrelated dimensions of knowledge in a unitary way, using MMT URIs extensively for interlinking between knowledge items as well as across dimensions. The dimensions are organized in a hierarchical folder structure, suitable for IDE support, archiving for easy distribution, as well as direct web access.

The dimensions presented below are derived from previous research and are neither final, nor the only ones possible. On the contrary, various build workflows can be added or modified, depending on the actual usage patterns that will take shape with the increased use of mathematical projects.

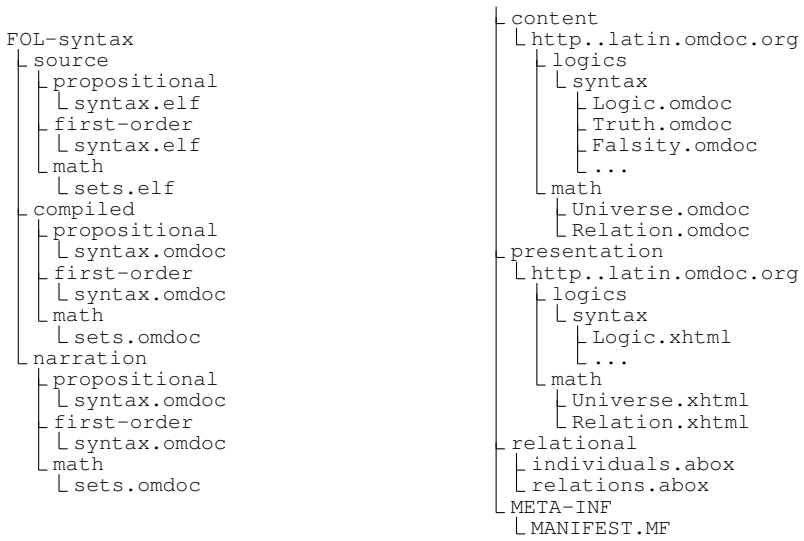


**Figure 3.1:** Directory structure of the FOL-syntax project

For example, a project containing the syntax of first-order and propositional logics, using set theoretic modules in the *math* namespace for the FOL universe, might have the folder structure shown in Figure 3.1, where each top-level subfolder corresponds to one of the dimensions discussed in this section. The explanation of the `META-INF` subfolder is deferred until the next section, as it pertains to the MAR file format.

**compiled** The most important workflow in a project is the compilation process, which is conceptually straightforward: the Twelf compiler reads a file and writes an OMDoc representation in a separate file.

MMT automatizes this process for all the Twelf files in a `source` directory tree, by instructing the Twelf compiler to put the OMDoc output in a separate file in a `compiled` folder, with the same name as the input file but with the `.omdoc` extension. Moreover, the folder structure of the generated `compiled` folder is the same as the original `source` folder: each file `source/`*path*`/`*file*`.elf` is mapped into `compiled/`*path*`/`*file*`.omdoc`, where *path* is of the form $dir_1/\ldots/dir_n$ and *file* is a file name.

Each MMT entity in the OMDoc files has an XML attribute

$$\texttt{source="}path/file\texttt{.omdoc?}col_i\texttt{:}line_i\texttt{-}col_f\texttt{:}line_f\texttt{"}$$

which provides a link to its exact location in the `source`.

**content** So far, we distinguished two possible ways of referencing a MMT entity in a project: (i) by file path in the `source` folder together with the position within the file and (ii) by its MMT URI. The first referencing method is straightforward, since a user or program can just traverse the `source` folders according to the given path. The second method, however, involves sending a query to the catalog, which requires having the catalog installed and configured. Since the namespace hierarchy of a project is orthogonal to the file hierarchy, the folder structure in either `source` or `compiled` is not appropriate when searching for an entity using its MMT URI.

Therefore, we devise a new dimension, `content`, which stores the modules in files according to their namespace. The path of each file is determined by the URI components of the namespace of the modules it stores. The compiled versions of all the modules with namespace

$$[scheme://][userinfo@]D_1.\ldots.D_m[:port]/S_1/\ldots/S_n$$

are stored sequentially into the OMDoc file

$$\texttt{content}/[scheme..]D_1.\ldots.D_m/S_1/\ldots/S_n\texttt{.omdoc}$$

For example, the modules in the namespace

`http://latin.omdoc.org/logics/syntax/`

are put into individual files in the folder

`content/http..latin.omdoc.org/logics/syntax/.`

**narration** In most forms of human communication, including academic publications, learning materials, and encyclopedia articles, the conveyed information is organized linearly, contains both formal and informal fragments, and includes transition texts, examples, re-iterations of previously defined elements, and backward and forward references. This constrasts with the structure of information in the `source` and `content` dimension: each file can be viewed as a list of $n$-ary trees, where the nodes are Twelf entities, with an optional comment attached to each node. This way of organizing information is not appropriate for the purposes described above, hence a new dimension is needed, called `narration`, containing one subfolder for each user-edited narrative exposition.

The `narration` contains documents with references to the `content` representations, transition texts, examples, discussions, and sections. General work in this direction has been done within the Kwarc group [Mül10]. In OMDoc-based narration, references are of the form

`<mref target="`*uri*`"/>`

where *uri* is the MMT URI of a module that can be retrieved from the `content` dimension.

In the case of Twelf, a basic form of narration follows the same file structure as the `source` and is simply a list of references to each module in the file. It is automatically generated by MMT, hence it slightly beats the purpose of this dimension, since it does not introduce any information already present in the `source`. It is, however, as a starting point from which more useful narrations can be derived.

Narration documents can also be based on STEX, a LATEX-based semantic markup language developed in Kwarc [Koh04]. Since STEX compiles to OMDoc, it can include MMT URI references, thus qualifying for inclusion in a separate `narration` dimension. The other compilation flow of STEX ends in a PDF document, making the STEX an excellent medium for presenting the information in `content` in the form of lecture notes, articles, printable tutorials or even books. The integration of the STEX tool hierarchy with Twelf workflows is the topic of future research.

**presentation** MMT can also convert modules compiled from OMDoc into the XHTML + MathML format with JOBAD annotations [GLR09], which intended to be viewed within a browser. We created a project workflow which maintains a `presentation` folder with the automatically generated XHTML files, obeying the same folder structure as the `content` dimension. Specifically, each XHTML file is a web page containing Each module in the presentation files has an XML attribute `jobad:href="`*URI*`"` giving its MMT URI.

If a project contains more than one `narration`, it could be useful to associate a `presentation` for each of them. Furthermore, other presentation formats are conceivable, such as PDF or even video files.

As an alternative to the Twelf-based workflow, STEX constitutes the `source` dimension, the `content` is the compiled OMDoc, and the `presentation` consists of XHTML+MathML and/or PDF files.

**relational** This dimension consists of two files in the `relational` folder: `individuals.abox` and `relations.abox`, containing an list of unary and, respectively, binary RDF predicates, encoded in XML.

The unary predicates are of the form

`<individual type="`*typePredicate*`" uri="`*uri*`">`

where *typePredicate* is the type of the MMT entity, such as `isTheory`, `isView` or `isCstDeclaration`, and *uri* is its MMT URI. Each module, declaration and assignment from the `content` dimension has a corresponding unary predicate in `individuals.abox`.

The binary predicates are of the form

`<relation subject="`$uri_1$`" predicate="`*pred*`" object="`$uri_2$`">`

where $uri_1$ and $uri_2$ are two MMT URIs, *pred* is a relationship predicate such as `hasChild`, `hasDomain`, `importsFrom`, `dependsOnTypeOf` or `dependsOnDefinitionOf`.

Storing separate relational information enables RDF extraction tools to access essentially the same information that can be obtained by querying MMT or, partially, the catalog. This allows integration with generic RDF-based tools, which adds many other possible workflows.

## 3.4 Mathematical Archives

This section presents a distribution and packaging format for the dimensions discussed above. We define the mathematical archive (MAR) file format to be a zip file, similar in purpose to Java `jar` files [Ora11a], containing the dimensions developed in Section 3.3, each in its own folder. All the dimensions are optional, except `source` and `content`.

Additionally, a mathematical archive contains a mandatory `MANIFEST.MF` file in the `META-INF` folder, conveying essential information required for understanding the content of the archive. The manifest file contains key-value pairs, one per line. The key is a sequence of characters excluding column and spaces. The first line, and the only one required, is

```
manifest-version:  1.0
```

and indicates the version of the manifest syntax. The version number can be changed if the syntax is updated and becomes ambiguous for a parser which only understands version `1.0`.

The manifest can optionally convey information about the various dimensions included in the package. However, a dimension is allowed to exist even if not mentioned in the manifest. A software that manages mathematical archives must first inspect the top-level directory entries to determine which dimensions are present, and only then extract additional information from `MANIFEST.MF`. Once the number and complexity of dimensions increases, it might be more sustainable to use special manifest files for each dimension. In this case, we will provide a mechanism for the main manifest file to reference the others.

The main role of the manifest is to avoid interpretation ambiguities, by specifying the language each dimension is written in, along with the tool that generated it. Table 3.1 lists the various optional key-value pairs in the manifest specification. The *D*`-generated-by` is intended to be used when

| Key | Value |
|---|---|
| `id` | archive file name |
| `version` | archive version number |
| `description` | one-line description of the archive |
| *D* | the language *D* is written in |
| *D*`-generated-by` | the tool that generated the dimension |
| `source-base` | an absolute URI against which the URIs in `source` are resolved |

**Table 3.1:** Key-value pairs in MANIFEST.MF. In the manifest, *D* is replaced by a dimension name

dimension *D* is generated either entirely automatically, or via tool-assisted human intervetion.

The directory structure from Figure 3.1 can be compressed into `FOL-syntax.mar`, with the contents of the manifest shown in Listing 3.1.

```
manifest-version: 1.0
id: FOL-syntax
version: 2
description: syntax of FOL and propositional logic
source: Twelf
source-base: http://latin.omdoc.org/
compiled: OMDoc
compiled-generated-by: Twelf
content: OMDoc
content-generated-by: {\MMT}
narration: OMDoc
narration-generated-by: {\MMT}
presentation: XHTML
presentation-generated-by: {\MMT}
relational: RDF/XML
relational-generated-by: {\MMT}
```

**Listing 3.1:** Contents of `META-INF/MANIFEST.MF`

## 3.5 IDE Services

In the previous sections, we presented several compilation and building workflows that are possible in a project-based setting. The authoring process can also subsume editing workflows, such as autocompletion, hover overlay, code correction, definition search and outline view. Both building and editing workflows are best automated within an

A specific requirement for the editing workflows is resiliency. All services must function even if the code being edited is partially incorrect. In case of true ambiguity, the services will degrade gracefully by acknowledging the ambiguity and helping the user rectify the problem.

Each of the editing workflows can be implemented efficiently using either the dimensions of knowlege from Section 3.3 or the services of the catalog presented in Section 3.2. More research is needed for assessing which source is a better fit and more sustainable.

**autocompletion** assists the user in getting context-sensitive suggestions while typing an identifier. The exact type of autocompletion depends on the position within the document, as well as what characters have been written so far in the identifier. We analyse each situation below.

*module references* This type of autocompletion is offered in places where module references are expected, i.e. `include` and `meta` declarations, as well as the (co)domain and definition of `view` and `structure` declarations. When the user starts typing, two separate lists are suggested: a list of module names declared anywhere in the current namespace, and a list of namespace prefixes previously declared in the document. After the user writes a namespace prefix followed by a dot, the IDE presents only the list of module names declared in that namespace.

The IDE can easily obtain the list of modules declared in a specific namespace using two approaches: by inspecting the corresponding folder in `content`, which contains one file for each module declared in the namespace, or by inquiring the catalog. The list of modules returned can be further restricted in the definitions of links and in link inclusions, by excluding references to modules which do not type-check. This information can be inferred from the `relational` dimension. In both cases, it is the task of the IDE to maintain the map from the list of prefixes in scope to the denoted namespace and to identify the MMT construct in which the user is positioned.

*symbol references* This service is offered in places where symbol references are expected, i.e. inside terms, as well as on the left-hand side of assignments in the body of a link. In the case of signatures, the IDE initially shows a list of symbol names, both local and imported, which are visible at that point in the module. Only names consisting of a single component are shown, i.e. not containing dots, to avoid excessive clutter. Once the user writes an identifier component followed by a dot, the IDE presents a new list of symbols whose reference starts with that component.

To encourage modular thinking and encourage a clean split between abstraction levels, the first symbols shown in the list are the ones declared locally, followed by the first-degree imports, the second-degree imports, and so on. If the symbol reference being typed is on the left side of an assignment and is preceded by the keyword `%struct`, only structure references are shown in the autocompletion list.

The IDE can obtain this information by either extracting it from the `relational` dimension, or directly inquiring the catalog. In both cases, it is the task of the IDE to identify the MMT construct in which the user is positioned.

*keywords* In any place of a document, if the user writes `%`, the IDE shows a list of keywords which are valid in that position. Determining this list does not require any access to external information and can be simply determined from the type of the current MMT construct.

*namespaces* In a namespace alias declaration, after the user has written the opening quote, the IDE shows the entire list of namespaces declared in the project, relative to the current namespace. Namespaces which are "closer" to the current one are shown first, i.e. those who have the same scheme and authority, and the common path prefix is as large as possible. The list of namespaces can be obtained either by recursively traversing the `content` folder or directly inquiring the catalog.

Suppose we want to enhance the project in Figure 3.1 with the encoding of proof theory in first-order logic. Therefore, we create a file and establish the current namespace `http://latin.omdoc.org/logics/proof_theory`. Since we need to reference the syntax modules, we start writing a namespace alias declaration `%namespace syn = "`. At this point, the autocompletion service shows the list

```
syntax
../math
```

Note that both namespaces are relative to the current one, and the namespace which is closer is shown first.

**hover overlay** provides in-place meta information when the user hovers the pointer over a module or symbol reference. The best candidates for display are the full URI, the associated semantic comment and, for object-level symbols, the original and inferred type.

We define the *original type* of a constant to be its type as it is written in the source file, while the original type of a structure is *Domain* `->` *Codomain*. The original type can be missing, since Twelf does not require the symbols to be explicitly typed, as long as they have a definition.

By contrast, the *inferred type* of a constant is the type determined by the Twelf compiler, which must be parsed from the OMDoc output in the `compiled` or `content` dimension. Note that, as a particuliarity of the Twelf language, the length of the inferred type can, in certain cases, be exponentially large in terms of the original type. In practice, even though the exponential increase in size does not happen too often, the IDE must check for this situation and choose whether to display it or not.

A semantic comment is, abstractly, a set of key-value pairs, where two notable keys are `short`, denoting a one-line description, and `long`, denoting the full version of the comment, which typically spans several lines. The keys can have arbitrary names.

Since the comment, as well as the type, can be fairly large, the IDE implements a two-step hover service. When the user first points to a module or symbol reference, the bubble which immediately appears contains the `short` description and the original type of the symbol. If the user presses a predefined key, a temporary window appears with the full information described above.

The semantic comment associated with an MMT entity can be extracted from the `content` dimension, as well as by querying the catalog.

**search by metadata** An interesting IDE service is enabling the user to search for a module or symbol by entering words from its description in the metadata. Character-by-character search similar to

Google Instant [Goo11] is well-suited in this situation. This service works best if most entities have well-written and informative semantic comments. The search experience in larger projects can be improved in many ways, such as sorting and grouping the results based on relevance, or allowing the logical operators AND, OR and NOT.

Searching by metadata can be integrated with autocompletion, to help a user who is not familiar with the notations to reach the desired symbol faster.

To provide this service, the IDE can obtain all the comments from the catalog or directly from the `relational` dimension, along with their associated MMT entities.

**outline view** and **project explorer** are two widgets which visualize the project tree based on the file structure and, respectively, the MMT URI structure. The file structure can be obtained by recursively traversing the `source` folder, while the MMT URI structure is obtained either by traversing the `content` folder, or inquiring the catalog.

For all of these services, we discussed two alternative sources that the IDE can consult: one or more dimensions stored in the project, and the catalog. An important difference between them is that the catalog is an appropriate source of information even for files that are under construction or incomplete, while the build workflow that produces the various dimensions can only use those files that are syntactically and semantically correct. Consequently, if multiple files are being edited in the same session, the catalog is a much better source of information than reading the dimensions, which are incomplete in this situation.

A third source of information that the IDE can use is the MMT software, which in turn extracts the information from the `compiled` dimension. MMT is the reference implementation of the language and is also responsible for producing all the dimensions and managing the Twelf compiler and the catalog. Therefore, it is preferable to access the dimensions through MMT API, which takes care of all the semantic processing required for IDE services.

These workflows have been partially implemented in the Eclipse IDE as a test case, using the SIDER framework [Juc11, JK10].

## 3.6 Migration to a Project-Based Repository

We verify the infrastructure described above on the Atlas library described in Section 2.3. The first step was to prune out encoding which had become obsolete. After the pruning, the Atlas contained 632 modules in 193 files. In the second step, we added namespace declarations to each file and eliminated the `%read` statements, which enabled large-scale module name simplifications. We took the opportunity to redesign the folder and file structure, which now no longer determines the logical hierarchy of the modules. After the redesign, the number of files was reduced from 193 to 145. However, the number of modules stayed roughly the same, since the refactoring did not modify their internal structure.

The Atlas defines a relatively small number of namespaces. Their URIs are self-describing. To reduce the learning curve for new users, namespace prefixes are used consistently in all documents. Table 3.2 shows the namespaces and their prefixes.

Once the Atlas library was redesigned, we were able to apply the project-based workflows discussed in Section 3.3 and, for the first time, obtain a unified list of issues that needed to be addressed, i.e. unfinished modules, documents written in an obsolete version of Twelf, and various outdated dependencies resulting from the practice of storing multiple branches of certain subprojects in the same folder.

| Prefix | Namespace URI |
|---|---|
| cat | http://latin.omdoc.org/category_theory |
| minimal | http://latin.omdoc.org/foundations/minimal |
| hol | http://latin.omdoc.org/foundations/hol |
| isabelle | http://latin.omdoc.org/foundations/isabelle |
| mizar | http://latin.omdoc.org/foundations/mizar |
| zfc | http://latin.omdoc.org/foundations/zfc |
| syn | http://latin.omdoc.org/logics/syntax |
| pf | http://latin.omdoc.org/logics/proof_theory |
| mod | http://latin.omdoc.org/logics/model_theory |
| mod_cat | http://latin.omdoc.org/logics/model_theory/categorical |
| mod_kripke | http://latin.omdoc.org/logics/model_theory/kripke |
| sound | http://latin.omdoc.org/logics/soundness |
| sound_cat | http://latin.omdoc.org/logics/soundness/categorical |
| sound_kripke | http://latin.omdoc.org/logics/soundness/kripke |
| math | http://latin.omdoc.org/math |
| prop_cat | http://latin.omdoc.org/translations/prop_cat |
| fol_ifol | http://latin.omdoc.org/translations/fol_ifol |
| prop_iprop | http://latin.omdoc.org/translations/prop_iprop |
| fol_sfol | http://latin.omdoc.org/translations/fol_sfol |
| ml_fol | http://latin.omdoc.org/translations/ml_fol |
| prop_hol | http://latin.omdoc.org/translations/prop_hol |
| sfol_fol | http://latin.omdoc.org/translations/sfol_fol |
| sfol_hol | http://latin.omdoc.org/translations/sfol_hol |
| tt_zfc | http://latin.omdoc.org/translations/tt_zfc |
| tt | http://latin.omdoc.org/type_theories |

**Table 3.2:** Namespaces and their prefixes in the revised Atlas library

This information enabled the authors of each part of the library to quickly rectify all of the problems, some of which had not been addressed since early stages of the Atlas project.

# Chapter 4

# Catalog Service Implementation

The basic role of the catalog service is to maintain a list of directories with Twelf files and periodically crawl them, including subfolders. Each crawl updates in-memory maps and data structures containing metadata and dependency information about each file and module. While reading each file, the parser checks for some aspects of well-formedness, without doing type checking and without following references. In particular, the system computes the MMT URI and exact position within the file (URL) of each module, declaration and assignment. With this information, it maintains a map from logical (MMT URI) to physical (URL) locations, which is critically needed by external tools and compilers.

This section will first outline the requirements for the catalog, followed by a description of the development approach. I will then present its architecture, configuration parameters, the three interfaces, and relevant implementation details.

The catalog is fully integrated with the Twelf compiler [RS09,RS11] and the MMT project [Rab11].

The code is licensed under the GNU General Public License v3.0 [Fre97] and is publicly available at the following URLs:

source, docs: `https://svn.kwarc.info/repos/MMT/src/lfcatalog`
binaries: `https://svn.kwarc.info/repos/MMT/deploy/lfcatalog`

## 4.1 Requirements

Once we enriched the Twelf language with namespaces, the issue of MMT URI resolution arises. Each namespace MMT URI must be translated to a URL composed of a file path on the disk and the line/column numbers of the start and end of the declaration. Since the namespace is independent from the file path, the MMT URI to URL mapping can only be computed after reading all the files in the project. Thus, we determined that we needed a separate piece of software able to do namespace resolution.

Stemming from the issue discussed above, we proposed several requirements for the catalog software:

(C1) Take as argument a list of files and folders containing Twelf documents and provide a MMT URI $\Rightarrow$ URL map that translates logical to physical locations, i.e. file path and start and end positions within the file.

(C2) Instead of crawling the locations every time, which is time-consuming if there are many files, crawl them only once and run in the background, answering queries.

(C3) Provide an interface which can be used by the Twelf compiler without much additional programming effort.

(C4) The software should stand on its own, but it must be able to seamlessly integrate with the MMT project as well.

In order to provide IDE support for Twelf on the semantic level, additional requirements had to be met:

(C5) Re-crawl a file once it is modified and periodically check for new or deleted files.

(C6) Provide a mapping from MMT URIs to the semantic comment associated with that MMT URI, in a language-independent form.

(C7) Given an exact position within a file, provide a list of identifiers which are valid at that position, to aid in auto-completion.

Finally, the software is a tool which should blend into the background and not require much user or programmer assistance, so additional requirements can be discerned:

(C8) Backward and forward compatibility with versions of the Twelf language.

(C9) Robustness and reliability: Most parsing errors should not result in dropping the file. Incorrect requests must be handled gracefully. All incidents must be logged.

(C10) Efficiency: parsing should be done in linear time and all requests should be answered in (amortized) constant time, avoiding disk access as much as possible. Since Twelf files can be quite large, only relevant information about them must be stored in memory.

## 4.2 Approach

I decided to use the Scala programming language based on four factors. First, API-level compatibility was desired with the MMT code base, which is written in Scala. Second, Scala makes use of all the Java libraries and has a rich collections library of its own, which greatly shortens the programming time. Third, the ability to program in a functional style and its advanced but unobtrusive syntactic constructs make the code much shorter than Java or C++ and less error-prone, as well as shortening the gap between intuition and implementation. Fourth, Scala classes can be used by Java code, with certain restrictions, explained in Section 4.7.

Several available open-source web servers such as the Java-based Jetty [Web11] or Tomcat (using JSP) or the Scala-based Lift [ea11] added too much overhead to the code, as well as significantly increasing the size of the deployed jar file. A minimal web server called Tiscaf [Gay11] was deemed to be more suitable, since its code base is short enough to be modified to our needs (the jar is only 324 KB), is written in Scala, allowing the calling code to take advantage of functional language features, and performs excellently in stress tests and concurrency situations.

The Twelf grammar is LL($k$) [RS69] for a finite and small value of $k$, hence it can be parsed by a top-down parser with constant lookahead. Several open-source LL($k$) parser generators for Java/Scala exist, most notably ANTLR [Par11]. However, I chose to use a hand-crafted parser for several reasons. First, some comments in Twelf are semantic and must be kept and, then, associated with other parsed structures in a non-trivial fashion. Second, the line between the parser and the lexer is blurred for Twelf, since identifiers can contain dots inside and can also be immediately followed by a dot. The syntax for modeling this situation is generally awkward and bloated in parser tools. Third, the tool

must completely recover from specific types of syntax errors, such as those inside terms, but not from other types of errors, such as not finishing a declaration with a dot, which makes the subsequent constructs ambiguous. Automatic parsers usually do not provide this level of customization. Fourth, keeping in mind the simplicity requirement, I tried to avoid dealing with automatically generated code.

Since the parser and lexer are both top-down with constant lookahead, the implementation makes no artificial distinction between them, providing one method for each lexer and parser rule.

Twelf is a research language in active development, with features constantly being added and removed. The code goes to great lengths to be backward- and forward- compatible with other versions of Twelf. Non-modular Twelf code is supported, as well as `%read` statements. Furthermore, future module or declaration types are supported, as long as they start with a directive, match curly brackets internally and end with a dot. Any such construction, whether from the past or the future, is then simply ignored by the catalog when answering queries.

The catalog internally uses HashSets and HashMaps from the Scala library for storing the parsed information, updating and deleting, and answering queries. Thus, every operation takes amortized constant time and does not access the disk, except for the queries that ask for the entire text of an entity. The disk is accessed as sparingly as possible, by ensuring that a file is crawled in at most three situations: first, when a location is added and the file is a descendant; second, when it is modified and the BackgroundCrawler thread schedules a re-check; and third, when the crawlAll method of the Catalog is called, either via the API or the web interface.

## 4.3 Architecture of the Catalog

The catalog is split into four loosely coupled components. The first three – frontend, backend and controller/storage – are pictured in Figure 4.1, while the fourth component, the **data model**, is omitted for clarity reasons, but is explained in detail in Section 4.8.

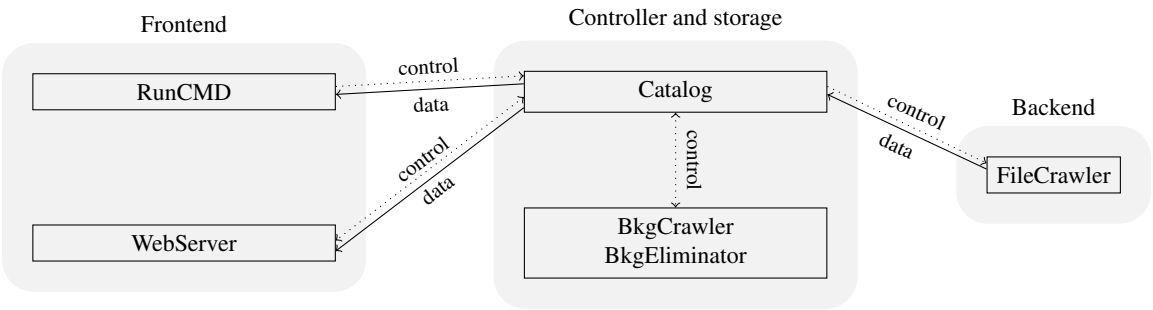


**Figure 4.1:** The architecture of the catalog

The **frontend** handles user interaction and consists of a command-line interface (Section 4.5) and a web interface (Section 4.6), the latter fulfilling requirement (C3). The class that runs the command-line interface also starts the web server, and allows only a limited set of operations, intended primarily for debugging.

The **backend** provides parsing functionality, encapsulated in the FileCrawler class and discussed in Section 4.9. The only exposed method is the object constructor

$$\text{FileCrawler} : \text{File} \Rightarrow \text{Document}.$$

Enforcing the loose coupling principle, FileCrawler only depends on the data model, and the only component that depends on FileCrawler is the Catalog.

The main class in the **controller/storage** component is Catalog, which performs three major tasks. First, it keeps a set of disk locations and continuously checks them for modifications, insertions or deletions (requirements (C2) and (C5)). New and deleted files are handled by two separate threads, BkgCrawler and BkgEliminator respectively, which check for modifications in the directory tree at user-defined time intervals.

Second, it provides a Scala- and Java-compatible API (see Section 4.7), used by the MMT software (requirement (C4)) and the two other interfaces.

Third, it maintains three maps, with public access level for convenience:

**urlToDocument** : Map[URL, Document], which maps a file path to the Document structure extracted from that file. Its key set is the set of file URLs that are currently indexed.

**uriToNamedBlock** : Map[MMT URI, NamedBlock], mapping a logical address to its NamedBlock (module, declaration or assignment).

**uriToModulesDeclared** : Map[MMT URI, Set[MMT URI]], mapping a namespace MMT URI to the modules declared in it. Since the same namespace can be declared in different files, the set of modules returned is collected from multiple files as well.

These indexes enable the Catalog to quickly answer queries about the data, namely requirements (C1), (C6) and (C7).

Table 4.1 specifies, for each component, the source files containing its implementation and the objects and classes defined in those files.

| Component | Source file | Classes |
|---|---|---|
| Frontend | run.scala<br>webserver.scala | Run<br>WebServer |
| Controller/Storage | catalog.scala | Catalog (object and class)<br>BackgroundCrawler<br>BackgroundEliminator |
| Backend | fileCrawler.scala | FileCrawler (object and class) |
| Data model | document.scala | Document<br>Block, NamedBlock<br>SemanticComment<br>Module, Sig, View<br>Assignment, CstAssignment, StrAssignment<br>Declaration, CstDeclaration, StrDeclaration<br>Position |

**Table 4.1:** Summary of the catalog components

The architecture can also be thought of as a simple Model-View-Controller design [Bur87], where the **Controller** is formed by the first three components (frontend, controller/storage and backend), the **Model** is our data model, and the **Views** are the return values of the three interfaces, namely Java, XML and plain text query answers.

## 4.4 Configuration

The software is packaged in **lfcatalog.jar**, which does not have any external dependencies. The following assumes that the system has the Java 6 Runtime installed and that `java` is in the current path. From the shell, the server can be started with

java -jar lfcatalog.jar (port <port>)? (location|+inclusion|-exclusion)*

*port* the port number on which the server runs. This argument is optional and must occur before the others. The default port number is 8080[1]. If the port is already in use by another server, the catalog searches for available ports incrementally, starting with the given port. The port on which the server is running is then printed to stdout.

*location* absolute path to a file or directory in the local file system. The program continuously and recursively crawls these locations and updates the hashes with the information from them.

*inclusion* an inclusion pattern, where *pattern* is a file or directory name, without its path. The star (*) is the only special character and matches any sequence of characters.

*exclusion* an exclusion pattern.

Inclusion and exclusion patterns affect which files and folders are crawled recursively, starting from the locations provided as command-line arguments and, later, given by web requests:

(i) *folders* are crawled if they do not match any exclusion pattern.

(i) *files* are crawled if they match at least one inclusion pattern, but no exclusion patterns. However, if no inclusion patterns are provided, only the second condition remains.

## 4.5 Command-Line Interface

The command-line interface is started by running lfcatalog.jar as described in Section 4.4, which simply calls the main function in the object info.kwarc.mmt.lf.Run.

This interface serves as a simple log to the web server, which, for every request received, prints a timestamp and the text of the query. If the query is getPositionInHeader, it also indicates success by printing ”OK” if the MMT URI points to an entity, or ”NOT FOUND” otherwise. The server also logs the setter requests with a message describing the changed state. In addition to requests, the server also logs the result of crawling each file, with a timestamp and the file path followed by either ”OK” or a list of error messages found while parsing.

The interface accepts four kinds of requests:

**delete** *file path* Deletes the location given by the path. The path must be absolute.

*file path* Adds a file or folder location. Again, the path must be absolute.

**errors** Prints all the errors that occurred during parsing the most recent versions of the files.

**exit** Quits the server.

These requests are not only for convenience and debugging, but also a replacement for the web setter requests, which are a security hazard in a production environment.

[1] Historically, a common choice for web servers operated in non-root mode - see GRC - Port Authority, http://www.grc.com/port_8080.htm

## 4.6 Web Interface

The web server is started either by running lfcatalog.jar as described in Section 4.4, or by calling the constructor of info.kwarc.mmt.lf.Catalog from the API, as explained in Section 4.7 and the project Javadocs.

As pointed out in Section 4.2, the initial reason for developing a web interface was to meet the requirement

> (C3) Provide an interface which can be used by the Twelf compiler without much additional programming effort.

Once this was established, we realized that it is possible to add other query types, with great potential utility in IDE tools.

The web server is started automatically when running the command-line interface:

java -jar lfcatalog.jar (port <port>)? (location|+inclusion|-exclusion)*

The server accepts HTTP GET requests and returns either an XML node or plain text. Table 4.2 lists the getter requests, their accepted query parameters and the returned entities. In case of an error,

| Request | Query param | MIME type | Description |
|---|---|---|---|
| getPosition | uri=MMT *URI* | text/plain | position in the HTTP body |
| getPositionInHeader | uri=MMT *URI* | text/plain | position in the HTTP header |
| getMeta | uri=MMT *URI/file path* | text/xml | semantic comment in XML |
| getMetaText | uri=MMT *URI/file path* | text/plain | semantic comment in plain text |
| getText | uri=MMT *URI/file path* | text/plain | full text |
| getOmdoc | url=*file path* | text/xml | OMDoc skeleton of file |
| getDependencies | uri=MMT *URI* | text/plain | list of dependencies as MMT URIs |
| getChildren | uri=MMT *URI* | text/plain | list of children as MMT URIs |
| getNamespaces | url=*file path* | text/plain | namespaces declared in file |
| getNamespaces | - | text/plain | all namespaces in managed locations |
| getModules | uri=MMT *URI* | text/plain | modules in given namespace |
| help | - | text/plain | summary of available requests |

**Table 4.2:** List of getter server requests

such as an unknown MMT URI, the server returns the error message in plain text. The case when the returned value is empty, such as when an existing entity does not have a comment or dependencies, does not constitute an error. In such situations, the server simply returns an empty HTTP body.

**getMeta** and **getMetaText** Print the semantic comment associated with an entity, either in a parsed XML form (getMeta) or plain text (getMetaText). If the parameter is a file path, the server returns the semantic comment from the beginning of the file; if it is a module, declaration or assignment MMT URI, the semantic comment associated with the respective entity is returned. In both cases, if the document or the entity does not have a semantic comment associated with it, an empty text is returned. If the parameter is neither a valid MMT URI, nor a file path in the list of managed locations, the reply is an error message.

**getOmdoc** and **getText** Print an entity, either in a parsed OMDoc form (getOmdoc) or plain text (getText). If the parameter is a file path, the server returns the entire document; if it is a module,

declaration or assignment MMT URI, the respective entity is returned. Note that getOmdoc only returns a skeleton of the entity, omitting all the Twelf terms and notation declarations. If the parameter is neither a valid MMT URI, nor a file path in the list of managed locations, the reply is an error message.

**getNamespaces** Print the namespaces introduced by the document given by its disk address. If the given path is not a a descendant of any of the managed locations, then an error message is returned. The command also has a variant with no parameters, returning all namespaces defined in all managed locations.

**getModules** Print a list of MMT URIs of the modules declared in the given namespace. If the namespace is unknown, the command returns an error message.

**Get dependencies** and **getChildren** Print the MMT URIs of the dependencies or, respectively, the children of an entity, one per line. The MMT URI given as parameter must belong to a module, declaration or assignment, otherwise an error message is returned.

The dependencies of a signature are the imported modules, the domains of the structs, imports and meta declared and, for views, the domain and codomain. The only dependency of a structure declaration is its domain, while assignments and constant declarations have no dependencies.

The children of a signature are all the declarations which have a MMT URI, i.e. struct and constant declarations. The children of a view or a struct declaration are the assignments.

**getPosition** Print the file path and position within the file of an entity. All the position information is encoded in an URL consisting of the file path, followed by a fragment part of the form

*#firstLine.firstCol-secondLine.secondCol*

The given MMT URI must belong to a module, declaration or assignment, otherwise an error message in plain text is returned.

**getPositionInHeader** Return the same information as **getPosition**, but in the HTTP header *X-Source-url*, while keeping the HTTP message body empty. In case of an error, it simply omits the custom HTTP header and writes the error message in the HTTP body. This request is sufficient for satisfying requirement C3, since the Twelf compiler only needs to know where to look for a referenced module. Due to bugs and limitations in the SML/NJ socket library, it is much easier for the compiler to use the information in the header instead of the HTTP body.

**help** Print a summary of the HTTP requests accepted by the server, their parameters and what they return.

In addition to the getter requests, the server accepts a number of requests, summarized in table 4.3, which modify its state. These requests can also be sent from a simple purpose-built administration page (Figure 4.2), which also provides information about the locations being maintained. Since the setter requests do not return any useful value, they simply redirect to the administration page.

**crawlAll** Crawl again all files in the managed locations. Only the files that have changed since the last crawl are re-crawled. This request is not needed in normal circumstances, since the server crawls all locations periodically. However, it could be used, for example, by an IDE after the user modified the structure of a project, to make the catalog up-to-date. A variant of this request

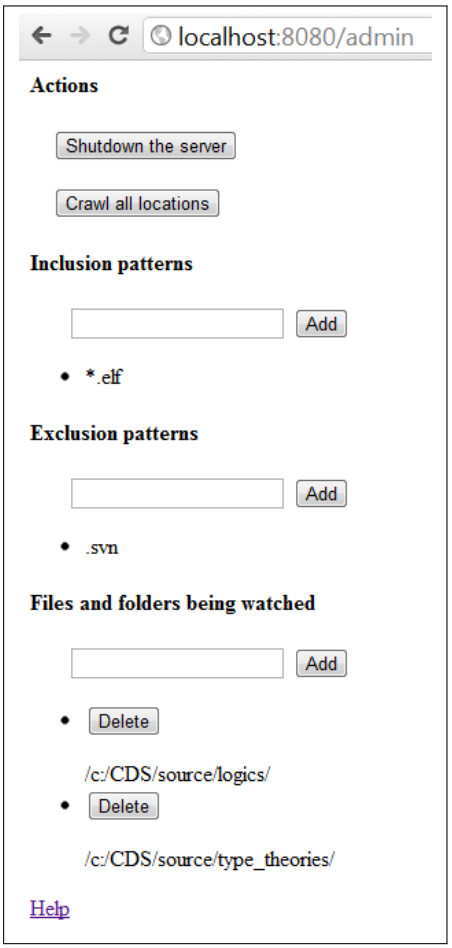


**Figure 4.2:** Web server administration page (http://localhost:*port*/admin)

which re-crawls a single document can also be used by an IDE after the user has modified a document, to ensure that future requests return updated information.

**admin?addLocation=***path* Add a disk location to the list of watched locations. This operation is equivalent to adding a location via program arguments when starting the server.

**admin?addInclusion=***pattern* This operation is equivalent to adding an inclusion pattern via program arguments when starting the server.

**admin?addExclusion=***pattern* This operation is equivalent to adding an exclusion pattern via program arguments when starting the server.

In a production environment, one can eliminate the security risk posed by these methods by either requiring authentification or simply disabling them. The latter option is possible since often the initial configuration remains the same throughout the lifetime of the web server. Even if locations or patterns have to be modified later, this can be accomplished using the command-line interface or the API, both of them from a trusted environment.

| Request | Query param | Effect |
|---|---|---|
| crawlAll | - | recrawl all modified files in managed locations |
| admin | - | print the administration page |
| admin | addLocation=*file path* | add a managed location |
| admin | addInclusion=*pattern* | add an inclusion pattern |
| admin | addExclusion=*pattern* | add an exclusion pattern |

**Table 4.3:** List of setter server requests

## 4.7 API

Since Scala runs on the Java virtual machine, a Scala jar is the same as a Java jar and can be run as a console application using

java -jar *name*.jar *parameters*

provided that the Scala library is either packaged inside the jar or available in the class path. To keep the package size small, lfcatalog.jar does not contain the Scala library. However, it can be added if necessary, to improve portability.

In order to use a Scala jar file in a Java project, one can simply add the jar to the classpath and use the classes and methods as usual. One important caveat is that the parameters and return values must either be defined in the jar, be part of the standard Java libraries, or have (implicit or not) converters available in the Scala library. In the last case, the Java project must import scala.collection.JavaConversions and scala.collection.JavaConverters.

The catalog API classes conform to these requirements, hence are inter-operable with Java code.

To start the catalog, one has to merely call the constructor of the class Catalog, which offers the same options as those described in Section 4.4. The methods of Catalog closely mirror the requests described in Section 4.6.

## 4.8 Data Model

Internally, the catalog uses an inheritance tree (Figure C.1) for storing all the information extracted from a Twelf file that closely mirrors the EBNF grammar discussed in Section 2.2.

**Block** an abstract class that is extended with different case classes for modules, declarations and assignments. The class stores the Position of the block, which is a wrapper around Pair(Pair(Int,Int), Pair(Int,Int)). The method toOmdoc, returning an OMDoc skeleton of the block as a scala.xml.Elem, is declared but not implemented.

**NamedBlock** inherits from Block and adds a name, a MMT URI and an optional semantic comment. A NamedBlock also stores the URL of the Twelf entity, which is computed from the file path and position within the file.

**Module** abstract superclass for Sig and View, inheriting from NamedBlock. It adds a set of dependencies.

**Sig** additionally stores a list of declarations, in the order in which they appear in the signature body.

**View** additionally stores the domain and codomain, as well as the list of assignments, in the order in which they appear in the body of the link.

**Assignment** with its children **CstAssignment** and **StrAssignment**, inherits from NamedBlock and models an assignment occurring in the body of a view or structure.

**Declaration** with its children **CstDeclaration** and **StrDeclaration**, inherits from NamedBlock and models a declaration occurring in the body of a structure. A StrDeclaration also optionally stores the MMT URI of the domain, if it is explicity specified in the Twelf code.

**SemanticComment** stores a list of parsed key-value properties found within a semantic comment, as well as the full text of the unparsed comment.

## 4.9 Parser

The parser implementation is contained in fileCrawler.scala. The only public method provided is the object constructor

$$\text{FileCrawler} : \text{File} \Rightarrow \text{Document}$$

that creates a FileCrawler and then calls its crawl method. The latter parses a file and returns a Document object, and throws either a FileOpenError if it cannot open the file, or a ParseError, if it encountered a syntactical error from which it could not recover. The method does not have any other side-effects, i.e., it does not communicate with the rest of the code. In particular, it does not follow references to other files.

As a rule, the parsing methods take at least two arguments: an index to the input string, and the MMT URI of the enclosing block. They always assume that the index points to the start of the respective directive. They return a Block or Document data structure, as well as an index to the position immediately following the last character of the parsed item, which is later used by the calling method to proceed with parsing.

The main parsing methods which correspond to the data structures are summarized in table 4.4.

| Method name | Return type | Parsed Twelf block |
|---|---|---|
| crawl | Document | *file* |
| crawlSemanticComment | SemanticComment | `%* ... *%` |
| crawlView | View | `%view ... .` |
| crawlSig | Sig | `%sig ... .` |
| crawlStrAssignment | StrAssignment | `%struct` *name* `:= ... .` |
| crawlCstAssignment | CstAssignment | *name* `:= ... .` |
| crawlStrDeclaration | StrDeclaration | `%struct ... = ... .` |
| crawlCstDeclaration | CstDeclaration | [`%abbrev`] *name* [`: ...`] [`= ...`] `.` |

**Table 4.4:** The most important parsing methods, returning structured data

Each method, apart from crawl, has the side-effect of updating the class-scope field keepComment with the last semantic comment it encounters. Although it is a departure from functional programming, this side-effect greatly simplifies the code, and is an intuitive way to attach the correct comment to each module, declaration or assignment. We now describe each method in turn.

**crawl** Builds a list of modules, namespace prefixes and current namespaces in the document. To simplify the parsing, it first reads the entire file into a flattened string, while remembering the position of each line start. The positions enable the computation of line and column positions for each Twelf element, as well as specific error reporting.

The main loop performs a constant look-ahead for the next `%`-directive (`%namespace`, `%sig`, `%view`) and calls **crawlNamespace**, **crawlSig** or **crawlView** respectively. The unknown directives are ignored and jumped over using skipUntilDot. The loop continues at the position returned by the callee. If a module is immediately preceded by a semantic comment, the loop is responsible for updating the Module data structure with this information.

**crawlSemanticComment** Reads a `%*` ... `*%` comment, extracts key-value properties and stores them in a Map[String, String]. The comment is first split in two parts: the first part are the lines which do not start with `@`, and the second are the lines starting with `@`, which must be grouped at the end. Both parts are optional.

Within the first part, the lines are treated as implicit key-value lines: the first line is saved as a property with key=`short`, while the rest of the lines, if they exist, form a property with key=`long`. This distinction is similar in intent to the short and long comments found in JavaDoc [Ora11b], or the brief and detailed descriptions found in Doxygen [vH11]. An IDE can, for instance, display the short comment on hover, and the long comment if the user presses the F1 key while hovering.

In the second part of the comment, each line is an explicit key-value, with the syntax

`@`*key space+ value*

where *key* contains no spaces.

**crawlView** Extracts the view name, domain and codomain, and adds the last two to a Set[MMT URI] of dependencies. Then, it calls **crawlLinkBody**, which further updates the set of dependencies.

**crawlSig** Extracts the signature name and calls **crawlSigBody**, which updates its set of dependencies.

**crawlStrAssignment, crawlCstAssignment, crawlCstDeclaration** Extracts the name of the symbol being declared or assigned to, and skips to the end using skipUntilDot. If the name has a prefix and it is not a valid namespace alias, a ParseError is thrown.

**crawlStrDeclaration** Extracts the name of the structure and optionally its domain. If the name has a prefix and it is not a valid namespace alias, a ParseError is thrown. If the structure is defined via a morphism, which is a space-separated sequence of identifiers, the method dadds them to the Set[MMT URI] of dependencies. Otherwise, the structure is defined via a link body, so the method calls **crawlLinkBody** to update the set of dependencies.

The methods in table 4.4 are helped by a number of convenience methods that simplify the parsing. Tables 4.5 and 4.6 provide an overview, while table 4.7 contains several lexer rules.

**crawlSigBody** This method parses a list of symbol, `structure`, `import` and `meta` declarations, and ignores any declarations starting with an unknown `%`-directive.

| Method name | Return type | Parsed Twelf block |
|---|---|---|
| crawlSigBody | Set[MMT URI] | { *declaration*⋆ } |
| crawlLinkBody | Set[MMT URI] | { *assignment*⋆ } |
| crawlNamespace | MMT URI, Option[String] | `%namespace` .... |

**Table 4.5:** Additional parsing methods, part 1

**crawlLinkBody** This convenience method parses a list of imports, structure assignments and constant assignments, and ignores any declarations starting with an unknown `%`-directive.

It takes advantage of the fact that the bodies of views and structure declarations are unified under the concept of a Link, hence are syntactically (and semantically) identical. For a parser which does not check references, however, there is a caveat. The identifiers in an `%include` declaration within a structure body can refer to a symbol from another module, but previously imported in the enclosing signature. Thus, the method has no way to determine, without checking other modules or even files, whether a prefix is a namespace alias or the name of a structure declared in another module.

The method treats this by taking a boolean parameter isView which tells is whether it is parsing a view or a structure body. In the view case, prefixes which are not among the namespace aliases declared in the document will generate a ParseError. In the structure case, the unknown prefixes are simply ignored.

Theoretically, this caveat could also apply to the identifiers in a view body; however, the Twelf compiler currently disallows "deep" assignments, i.e. assignments to a symbol from another module, which is imported via a structure or include. Consequently, this problem is not present in views.

Note that we have only discussed problems related to the identifier being declared or assigned to, or the so-called l-values. Once the parser looks into the declaration or assignment *terms* (r-values), the caveat will occur everywhere: in views, structure declarations, and even constant declarations, since terms can contain references to deeply nested symbols.

**crawlNamespace** For namespace declarations, it returns the (absolute or relative) MMT URI. In the case of a namespace *alias* declaration, it returns the alias and its relative MMT URI. In both cases, the calling method (**crawl**) takes care of computing absolute MMT URIs from the relative ones and updates the currentMMT URI and the Document's prefixes and namespaces fields.

| Method name | Return type | Parsed Twelf block |
|---|---|---|
| skipUntilDot | - | an arbitrary Twelf expression ending with dot |
| closeCurlyBracket | - | { arbitrary list of Twelf expressions } |
| expectNext | - | performs constant lookahead for the given string |

**Table 4.6:** Additional parsing methods, part 2

**skipUntilDot** Traverses an arbitrary expression until encountering a dot, on the assumption that any other dots are either enclosed within curly brackets, or part of an identifier. It deals with these two cases by calling closeCurlyBracket and crawlIdentifier, respectively. Strings and comments are treated as such, so dots within them do not count.

**closeCurlyBracket** Traverses a body of expressions until encountering a closing curly bracket, on the assumption that any internal curly brackets are matched. Strings and comments are treated as such.

**expectNext** Takes as arguments a string, an error message and a boolean acceptComments. The method skips over white spaces first (if acceptComments is true, then over any comments as well), then checks whether the next string matches exactly the one given as parameter. If not, it sends a ParseError with the message given as parameter.

The methods in table 4.7 implement greedy lexer rules with a lookahead of 1.

| Method name | Return type | Parsed Twelf block |
|---|---|---|
| crawlIdentifier | String | dot-separated identifier parts |
| crawlString | String | " . . . " |
| skipws | - | a succession of white spaces |
| skipwscomments | - | a succession of white spaces and comments |

**Table 4.7:** Lexer methods

**crawlIdentifier** If an identifier is followed immediately by a dot, the method always looks ahead to determine if the character after the dot is valid in an identifier (for characters allowed in identifiers, see Appendix A). If it is valid, then the method continues reading an identifier part.

**skipwscomments** As a side effect, it updates the class-scope field keepComment with the last semantic comment it encounters.

## 4.10 Error Management

The catalog is designed for robustness and reliability (requirement (C9)). All events are logged to the console, as explained in Section 4.5). The Document object returned by the parser contains a list of ParseErrors from which the parser was able to partially recover, along with recovered information. If there are no errors, the list is empty. Typical errors which can be recovered from are errors inside comments.

Since the parser ignores the content of terms, most errors within them are simply ignored.

There are situations when the parser encounters a syntactical error that makes the document ambiguous. In that case, there is no point in returning a partially recovered document, as the extracted information might be misleading. The default action is to throw a ParseError, which is then logged to the console.

## 4.11 Efficiency

The catalog was built from the ground-up with efficiency in mind, influencing three aspects of the implementation. First, it has influenced the choice of parser, which is a recursive descent parser with constant lookahead, ensuring linear-time crawling. Second, the various indexing data structures used in answering requests are HashSets or HashMaps, with amortized constant lookup and insertion time. Third, the data asked in requests is cached into memory, thus avoiding disk access as much

as possible. The only exception is the getText request, which needs to read the original Twelf code from the disk. Since Twelf files can be quite large, storing the original text threatens to overflow the available memory, hence only selected information about the files is cached. Requirement (C10) is thus fulfilled.

A future feature is to fetch documents from a web server, in case their MMT URI is not found in the list of managed locations in the file system. If the MMT URI is also a valid web URL, the catalog can try to fetch the document from a remote site and save it on disk, then crawl it and save the relevant information in memory. Thus, the catalog can serve as a web cache.

# Chapter 5

# Conclusion and Future Work

This work has brought four interrelated contributions for the improvement of knowledge management in Twelf:

1. The primary contribution is the implementation of namespaces, along with a catalog service that assists the file-based Twelf compiler in the new project-based setting.
2. The Atlas library is now refactored and fully ready for further expansion.
3. Several project build workflows are now integrated in MMT, along with the management of mathematical archive files.
4. We contributed a full Twelf parser to the SIDER development environment, as well as adapted the catalog for IDE-specific queries.

There are several directions for future work. The STEX tool hierarchy can be integrated with the Twelf workflows to enable interrelated informal and formal knowledge in a single package. The catalog can be adapted to fetch documents from a web server using their MMT URI, in case they are not found locally. The catalog can then save the document on disk, then crawl it and save the relevant information in memory. Thus, the catalog can serve as a web cache and remove the need for the user to download all the necessary dependencies before compiling, thus greatly simplifying the workflow. More IDE services can be supported, and the catalog can integrate better with MMT.

# Appendix A

# The Twelf Syntax (EBNF)

```
lineCharacter         ::= [^\x0A\x0C\x0D]
stringCharacter       ::= [^"\x0A\x0C\x0D]
lineWhiteSpaceCharacter ::= [ \t\x0B]
lineDelimiter         ::= \x0D\x0A | [\x0A\x0C\x0D]
keyword               ::= '->' | '<-' | '_' | '=' | 'type'| ':='
nonIdentifierPartCharacter ::= [\s\Q.:()[]{}%"\E]
identifierPartCharacter ::= [^\s\Q.:()[]{}%"\E]
identifierPart        ::= identifierPartCharacter+
identifier            ::= identifierPart ('.' identifierPart)*

singleLineComment     ::= '%' ((lineWhiteSpaceCharacter | '%') lineCharacter*)?
    lineDelimiter

multiLineNonSemanticComment ::= '%{' .* '}%'

commentShortForm      ::= [^\s@] lineCharacter*
commentLongForm       ::= (lineWhiteSpaceCharacter* ([^\s@] lineCharacter*)?
    lineDelimiter)*
                          lineWhiteSpaceCharacter* ([^\s@] lineCharacter*)?
commentPropertyValue  ::= ([^\s] (lineCharacter* ))?
commentProperty       ::= '@' identifier lineWhiteSpaceCharacter+
    commentPropertyValue
multiLineSemanticComment ::= '%*' lineWhiteSpaceCharacter* commentShortForm?
                          (lineDelimiter commentLongForm)?
                          (lineDelimiter commentProperty)*
                          [^\s]* '*%'

comment               ::= singleLineComment | multiLineNonSemanticComment
                          | multiLineSemanticComment
sc                    ::= ([\s] | comment)+

readDeclaration       ::= '%read' sc? '"' stringCharacter* '"' sc? '.'
namespaceDeclaration  ::= '%namespace' sc? '"' stringCharacter* '"' sc? '.'
namespaceAliasDeclaration ::= '%namespace' sc
                          identifier sc '=' sc?
                          '"' stringCharacter* '"' sc? '.'

termDeclaration       ::= ('%abbrev' sc)? identifier
                          (sc? ':' sc? term)?
                          (sc '=' sc term)?
                          sc? '.'
```

```
term                 ::= 'type'
                       | identifier
                       | '(' sc? term sc? ')'
                       | term sc '->' sc term
                       | term sc '<-' sc term
                       | '{' sc? identifierPart (sc? ':' sc? term)? sc? '}' sc?
                           term
                       | '[' sc? identifierPart (sc? ':' sc? term)? sc? ']' sc?
                           term
                       | term sc term
                       | '(' sc? term sc? ':' sc? term sc? ')'
                       | '_'

infixDeclaration     ::= '%infix'
                         sc ('none' | 'left' | 'right')
                         sc number
                         sc identifier
                         sc? '.'
number               ::= '0' | ([\d&&[^0]] [\d]*)
prefixDeclaration    ::= '%prefix'
                         sc number
                         sc identifier
                         sc? '.'
postfixDeclaration   ::= '%postfix'
                         sc number
                         sc identifier
                         sc? '.'
fixityDeclaration    ::= infixDeclaration | prefixDeclaration |
    postfixDeclaration

nameDeclaration      ::= '%name' sc identifier sc identifierPart sc?.

theoryIncludeDeclaration ::= '%include'
                         sc identifier
                         (sc? '%open' aliasList)?
                         sc? '.'
sigMetaDeclaration   ::= '%meta' sc identifier sc? '.'
viewMetaDeclaration  ::= '%meta' sc morphism sc? '.'
viewIncludeDeclaration ::= '%include' sc morphism sc? '.'
aliasList            ::= (sc aliasDeclaration)*
aliasDeclaration     ::= identifier sc? ('%as' sc identifierPart)?

morphism             ::= identifier (sc identifier)*

constantAssignment   ::= identifier sc? ':=' sc term sc? '.'
structAssignment     ::= '%struct' sc identifierPart sc? ':=' sc morphism sc? '.'
linkBody             ::= '{'
                         (sc? (linkMetaDeclaration | viewIncludeDeclaration
                             | constantAssignment| structAssignment))*
                         sc? '}'


simpleStructDeclaration ::= '%struct' (sc? '%implicit')? sc identifierPart
                         sc? ':' sc? identifier
                         sc '=' sc? linkBody
                         (sc? '%open' aliasList)?
                         sc? '.'
```

```
complexStructDeclaration ::= '%struct' (sc? '%implicit')? sc identifierPart
                         (sc? ':' sc? identifier)?
                         sc '=' sc morphism
                         (sc? '%open' aliasList)?
                         sc? '.'

emptyStructDeclaration ::= '%struct' (sc? '%implicit')? sc identifierPart
                         sc? ':' sc? identifier
                         (sc? '%open' aliasList)?
                         sc? '.'

structDeclaration    ::= simpleStructDeclaration | complexStructDeclaration |
    emptyStructDeclaration

signatureBody        ::= '{'
                         (sc? (sigMetaDeclaration | theoryIncludeDeclaration |
                             structDeclaration | termDeclaration |
                             fixityDeclaration |
                              theory | view))*
                         sc? '}'
theory               ::= '%sig' sc identifierPart sc '=' sc? signatureBody sc? '.
    '
view                 ::= '%view' (sc? '%implicit')? sc identifierPart sc? ':' sc?
     identifier sc '->' sc morphism sc '='
                         ((sc? linkBody) | (sc morphism)) sc? '.'

document             ::= (sc? (namespaceDeclaration | namespaceAliasDeclaration |
     readDeclaration |
                              theory | view |
                              termDeclaration | nameDeclaration |
                                  fixityDeclaration))* '%.'?
```

# Appendix B

# LL(*) Grammar for Twelf (Xtext)

Below is a LL(*) Grammar for Twelf, written for the Xtext parser generator [Ite11]. The header and minor processing-related tweaks are omitted. The grammar is maintained online at

https://github.com/jucovschi/MMT-Eclipse-Project/blob/master/info.kwarc.mmt.lf/src/info/kwarc/mmt/LF.xtext.

```
Model : {Model}
      WS* (constructs+=(modelConstructNOSP | termDeclaration)
            (WS* (constructs+=modelConstructNOSP | WS constructs+=termDeclaration
               ))* WS*)?
            (PERCENT DOT notInDocument+=anyTerminal*)?;

modelConstructNOSP : namespaceDeclarations+=namespaceDeclaration |
 readDeclarations+=readDeclaration |
 sigDeclarations+=sigDeclaration |
 viewDeclarations+=viewDeclaration |
 rawDeclarations+=(abbrevTermDeclaration | nameDeclaration | fixityDeclaration |
    unknownConstruct);

anyTerminal     : anyDirective|WS|PERCENT|COLON|QUOTE|LCBRACKET|RCBRACKET|
    LSBRACKET|RSBRACKET|LBRACKET|RBRACKET|
      DOT|ARROW|LARROW|EQUALS|ASSIGN|TYPE|UNDERSCORE|CID|URISTRING|ILLEGAL_TOKEN;

terminal LINESP   : ' '|'\t';
terminal LINEDELIM : '\r'? '\n';
terminal ML_COMMENT  : '%{' -> '}%';
terminal ML2_COMMENT : '%*' -> '*%';
terminal SL_COMMENT : PERCENT ((LINESP | PERCENT) !('\r'|'\n')*)? LINEDELIM;

// Reserved characters (can't be part of an identifier)
terminal PERCENT  : '%';
terminal COLON    : ':';
terminal QUOTE    : '"';
terminal LCBRACKET : '{';
terminal RCBRACKET : '}';
terminal LSBRACKET : '[';
terminal RSBRACKET : ']';
terminal LBRACKET : '(';
```

```
terminal RBRACKET : ')';
terminal DOT      : '.';

// Keywords (can't constitute an identifier)
terminal ARROW    : '->';
terminal LARROW   : '<-';
terminal EQUALS   : '=';
terminal ASSIGN   : ':=';
terminal TYPE     : 'type';
terminal UNDERSCORE : '_';

terminal CID         : !(LINESP|'\r'|'\n'|DOT|COLON|LBRACKET|RBRACKET|LSBRACKET|
    RSBRACKET|LCBRACKET|RCBRACKET|PERCENT|QUOTE)+;
terminal UNKNOWNDIRECTIVE : '%' ('a'..'z'|'A'..'Z'|'0'..'9')+;
terminal URISTRING : QUOTE -> QUOTE;

// Can't write these directly in rules because 'none' would become a token and
    would shadow CID
terminal INFIXNONE : '%infix' LINESP+ 'none';
terminal INFIXLEFT : '%infix' LINESP+ 'left';
terminal INFIXRIGHT : '%infix' LINESP+ 'right';

// To avoid lexer errors
terminal ILLEGAL_TOKEN : .;


// Object level

WS          : LINESP | LINEDELIM | ML_COMMENT | ML2_COMMENT | SL_COMMENT;
ID          : CID (DOT CID)*;

term        : termArr;
termPi      : LCBRACKET WS* name=(CID|UNDERSCORE) WS* (COLON WS* type=term WS*)?
    RCBRACKET;
termLambda  : LSBRACKET WS* name=(CID|UNDERSCORE) WS* (COLON WS* type=term WS*)?
    RSBRACKET;
termArr returns term:
                      termAppOrLambdaOrPi ({term.left=current} WS+ (ARROW|LARROW)
                          WS+ right=termArr)?;
// non-empty term application intermixed with Lambda and Pi quantifiers (possibly
     in the first position)
termAppOrLambdaOrPi returns term:
                          termLambdaOrPi |
                          left=termAppNoLambdaNoPi (WS* right=termLambdaOrPi)?;
// non-empty term application intermixed with Lambda and Pi quantifiers (but NOT
    in the first position)
termAppThenLambdaOrPi returns term:
                          left=termAppNoLambdaNoPi (WS* right=termLambdaOrPi)?;
// non-empty sequence of non-quantified terms
termAppNoLambdaNoPi returns term:
                          termBase ({term.left=current} WS+ right=termBase)*;
// [..] {..} .. [..] termApp
termLambdaOrPi returns term:
                          (quantifiers+=(termPi | termLambda) WS*)+ right=
                              termAppThenLambdaOrPi;
// (..) ---or--- TYPE ---or--- UNDERSCORE ---or--- ID
termBase returns term :
```

```
                        term=termAtomic |
                        {term} LBRACKET WS* (term=term WS* (COLON WS* type=term
                            WS*)?)? RBRACKET;
termAtomic returns term:
                        atom=TYPE | atom=UNDERSCORE | atom=ID;

morphism    : linkRefs+=[linkDeclaration|ID] (WS+ linkRefs+=[linkDeclaration|ID])
    *;



// Symbol level - signatures

termDeclaration  : (name=ID) ((WS* COLON WS* type=term)? & (WS+ EQUALS WS+ def=
    term)?) WS* DOT;
abbrevTermDeclaration : '%abbrev' WS+ termDeclaration;
nameDeclaration  : '%name' WS+ name=ID WS+ alias=CID WS* DOT;
fixityDeclaration : (fixity=INFIXNONE | fixity=INFIXLEFT | fixity=INFIXRIGHT |
    fixity='%prefix' | fixity='%postfix') WS+ fixityLevel=CID WS+ name=ID WS* DOT;

sigMetaDeclaration : '%meta' WS+ ref=[sigDeclaration|ID] WS* DOT;
sigIncludeDeclaration : '%include' WS+ ref=[sigDeclaration|ID] (WS* incOpt=
    includeOps)? WS* DOT;
structDeclaration : '%struct' WS+ ('%implicit' WS+)? name=ID (WS+ EQUALS WS+ def=
    morphism | // defined via morphism, no domain
      WS* COLON WS* from=[sigDeclaration|ID] // domain is given
      (WS+ EQUALS WS* (def=linkBody | WS def=morphism))? )? (WS* incOpt=
          includeOps)? WS* DOT;

includeOps       : {includeOps} '%open' (WS+ aliases+=aliasDeclaration)*;
aliasDeclaration : old=ID (=> WS* '%as' WS+ new=CID)?;



// Symbol level - links (views and structures)

constantAssignment : name=ID WS* ASSIGN WS+ def=term WS* DOT;
structAssignment : '%struct' WS+ name=ID WS* ASSIGN WS+ def=morphism WS* DOT;
viewMetaDeclaration : '%meta' WS+ def=morphism WS* DOT;
viewIncludeDeclaration : '%include' WS+ def=morphism WS* DOT;



// Symbol/module level - unknown constructs

anyDirective     : UNKNOWNDIRECTIVE | '%struct' | '%meta' | '%include' | '%
    namespace' | '%read' | '%sig' | '%view' | '%implicit' | '%open' | '%as' | '%
    abbrev' | '%name' | INFIXNONE | INFIXLEFT | INFIXRIGHT | '%prefix' | '%postfix
    ';

// { list of unknown or known constructs }
unknownBrackets  : {unknownBrackets} LCBRACKET (WS+ t+=anyConstruct)* WS*
    RCBRACKET;
// an unknown construct, optionally starting with a known or unknown directive
anyConstruct        : (t=unknownBody | directive=anyDirective (WS+ t=unknownBody)
    ?) WS* DOT;
// the part of the construct between the directive and the final dot, without
```

```
    surrounding WS
unknownBody       : t+=term (WS* (WS EQUALS (t+=unknownBrackets | WS+ (=> t+=term
    | t+=unknownBrackets)) | ASSIGN (t+=unknownBrackets | WS+ (=> t+=term | t+=
    unknownBrackets)) | COLON WS* (=> t+=term | t+=unknownBrackets) | anyDirective
     ))*;
unknownConstruct : directive=UNKNOWNDIRECTIVE (WS+ t=unknownBody)? WS* DOT;



// Module level - %namespace and %read declarations

namespaceDeclaration : '%namespace' WS+ (name=CID WS+ EQUALS WS*)? uri=URISTRING
    WS* DOT;
readDeclaration  : '%read' WS* file=URISTRING WS* DOT;



// Module level - signatures

sigConstructNOSP : abbrevTermDeclaration | nameDeclaration | fixityDeclaration |
    unknownConstruct | sigMetaDeclaration | sigIncludeDeclaration |
    structDeclaration | sigDeclaration | internalViewDeclaration;
sigConstruct  : sigConstructNOSP | termDeclaration;
sigBody       : {sigBody} LCBRACKET WS* (constructs+=(sigConstructNOSP |
    termDeclaration) (WS* (constructs+=sigConstructNOSP | WS constructs+=
    termDeclaration))* WS*)? RCBRACKET;
sigDeclaration : '%sig' WS+ name=CID WS+ EQUALS WS* def=sigBody WS* DOT;


// Module level - views

linkConstructNOSP : structAssignment | viewMetaDeclaration |
    viewIncludeDeclaration | unknownConstruct;
linkConstruct : linkConstructNOSP | constantAssignment;
linkBody      : {linkBody} LCBRACKET WS* (constructs+=(linkConstructNOSP |
    constantAssignment) (WS* (constructs+=linkConstructNOSP | WS constructs+=
    constantAssignment))* WS*)? RCBRACKET;
viewDeclaration : '%view' WS+ ('%implicit' WS+)? name=CID WS* COLON WS* from=[
    sigDeclaration|ID] WS+ ARROW WS+ to=morphism WS+ EQUALS WS* (def=linkBody | WS
     def=morphism) WS* DOT;
// view inside signature. Codomain is optional
internalViewDeclaration : '%view' WS+ ('%implicit' WS+)? name=CID WS* COLON WS*
    from=[sigDeclaration|ID] WS+ ARROW WS+ (to=morphism WS+)? EQUALS WS* (def=
    linkBody | WS def=morphism) WS* DOT;
linkDeclaration : viewDeclaration | internalViewDeclaration | structDeclaration |
     sigIncludeDeclaration | sigMetaDeclaration;
```

# Appendix C

# The Data Model (UML)

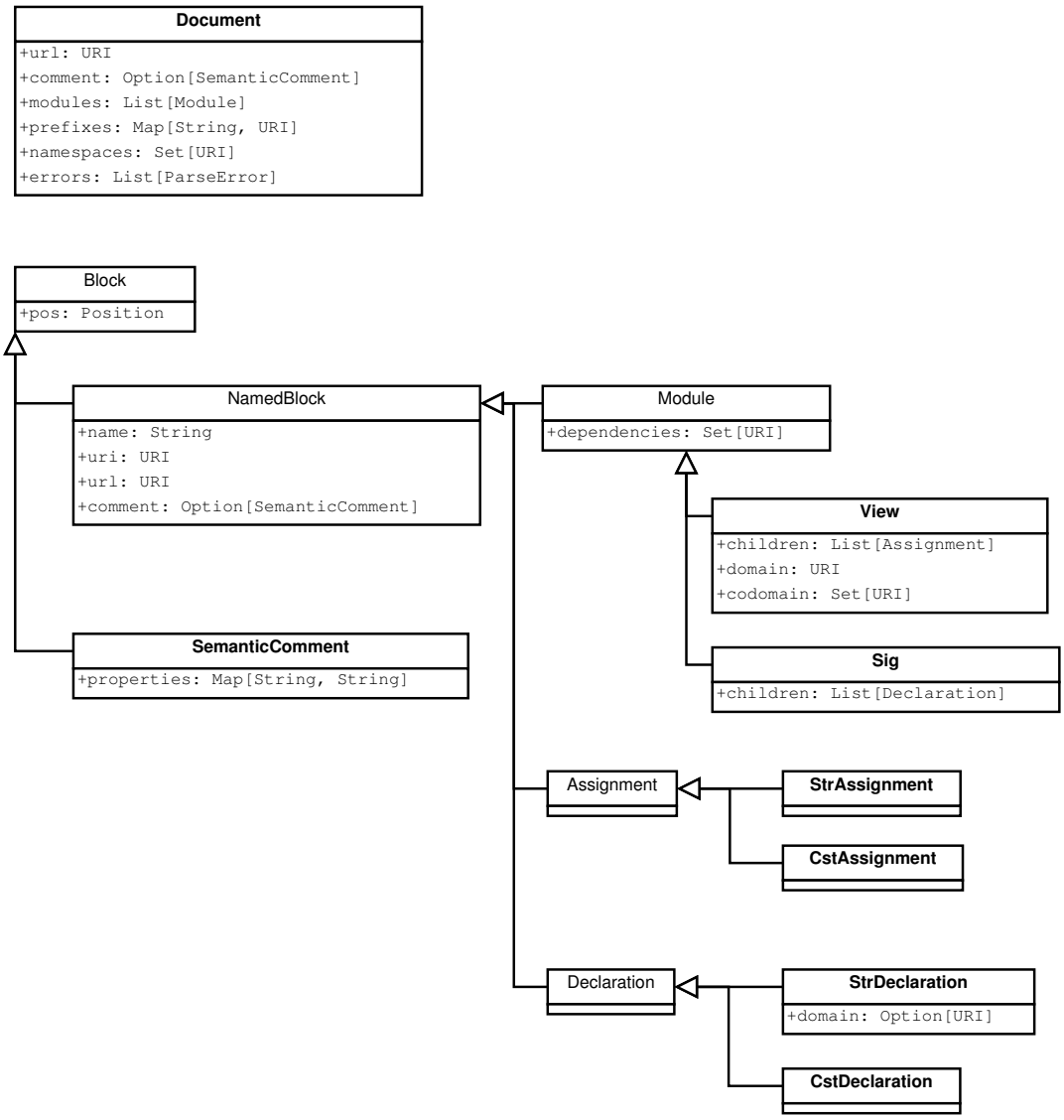


**Figure C.1:** UML diagram of the data model

# Bibliography


[Bar80] Hendrik P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland, 1980. 4

[Bar92] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992. 3

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force (IETF), 2005. 4, 10

[Bou74] Nicolas Bourbaki. *Algebra I*. Elements of Mathematics. Springer Verlag, 1974. 3

[Bur87] Steve Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html. 23

[CHK+11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011. 1, 3, 7

[DFRU11] James Davenport, William Farmer, Florian Rabe, and Josef Urban, editors. *Intelligent Computer Mathematics*, number 6824 in LNAI. Springer Verlag, 2011. in press.

[ea11] David Pollak et al. Lift, 2011. http://liftweb.net. 21

[Fre97] Free Software Foundation. Gnu general public license, 1997. http://www.gnu.org/copyleft/gpl.html. 20

[Gay11] Andrew Gaydenko. TIny SCAla Framework, 2011. http://gaydenko.com/scala/tiscaf/httpd. 21

[GLR09] Jana Giceva, Christoph Lange, and Florian Rabe. Integrating web services into active mathematical documents. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *MKM/Calculemus Proceedings*, number 5625 in LNAI, pages 279–293. Springer Verlag, July 2009. 14

[Goo11] Google. Google Instant, 2011. http://www.google.com/instant/. 18

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993. 3

[Ite11] Itemis. Xtext, 2011. http://www.eclipse.org/Xtext. 38

[JK10] Constantin Jucovschi and Michael Kohlhase. stexide: An integrated development environment for stex collections. *MKM 2010*, abs/1005.5489, 2010. 18

[Juc11] Constantin Jucovschi. SIDER, 2011. http://code.google.com/p/sider. 18

[KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Mathematical Knowledge Management*, volume 5144 of *Lecture Notes in Computer Science*, pages 504–519. Springer, 2008. 7

[Koh04] Michael Kohlhase. Semantic markup for TEX/LATEX. In Paul Libbrecht, editor, *Mathematical User Interfaces*, 2004. 13

[Koh06] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006. 3

[Mül10] Christine Müller. *Adaptation of Mathematical Documents*. PhD thesis, Jacobs University Bremen, 2010. 13

[Ora11a] Oracle. Java SE 7 Java Archive (JAR), 2011. http://download.oracle.com/javase/7/docs/technotes/guides/jar. 14

[Ora11b] Oracle. JavaDoc, 2011. http://java.sun.com/j2se/javadoc. 30

[Par11] Terence Parr. ANTLR v3 Parser Generator, 2011. http://www.antlr.org. 21

[Pfe91] Frank Pfenning. *Logic programming in the LF logical framework*, pages 149–181. Cambridge University Press, New York, NY, USA, 1991. 3

[PS] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th Conference on Automated Deduction*, pages 202–206. Springer-Verlag LNAI. 3

[PS10] Frank Pfenning and Carsten Schürmann. The twelf project, 2010. http://twelf.plparty.org/. 3

[Rab11] Florian Rabe. MMT Project, 2011. https://trac.kwarc.info/MMT. 3, 20

[RK11] Florian Rabe and Michael Kohlhase. A scalable module system. Manuscript, submitted to Information & Computation. http://arxiv.org/abs/1105.0548, 2011. 3

[RS69] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the first annual ACM symposium on Theory of computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM. 21

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009. 3, 20

[RS11] Florian Rabe and Carsten Schürmann. Twelf - MMT, 2011. http://trac.kwarc.info/MMT/wiki/Twelf. 20

[vH11] Dimitri van Heesch. Doxygen, 2011. http://www.stack.nl/~dimitri/doxygen. 30

[Web11] Webtide. Jetty WebServer, 2011. http://jetty.codehaus.org/jetty. 21