UCLouvain

École polytechnique de Louvain

# OzDoc

## Development and implementation of a highly automated documentation generator for the Oz programming language

Authors: **Simon ALEXE, Péter FRENYO**
Supervisor: **Peter VAN ROY**
Readers: **Benoît DUHOUX, Nicolas LAURENT**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

*A baby has brains, but it doesn't know much.*
*Experience is the only thing that brings knowledge,*
*and the longer you are on earth*
*the more experience you are sure to get.*

L. FRANK BAUM
The Wonderful Wizard of Oz

# Acknowledgements

This project was not just the fruit of two people's work. It might have been born in our brains, but would've only stayed a dream if not for the precious help we received all throughout this journey.

We would like to give our most sincere thanks to our thesis advisor, Professor Peter Van Roy of the Ecole Polytechnique de Louvain and the Institute of Information and Communication Technologies, Electronics and Applied Mathematics at Université Catholique de Louuvain, for embarking with us on this adventure and supporting us all throughout. Prof. Van Roy would always welcome us to his office with a smile to answer our questions about our research, or when we ran into what seemed like dead ends until we got his invaluable advice. He allowed this project to be our own work, but still steered us in the right direction when we most needed it.

We would also like to greatly thank our readers, soon-to-be PhDs Benoît Duhoux and Nicolas Laurent, for accepting to be the reviewers of this final piece of work in our academic cursus, as well as their important advice.

Many thanks to the the Ecole Polytechnique de Louvain and UCLouvain, especially the teachers and teaching assistants to courses mentionned all throughout this report, without which we would've been stuck in development hell. Many thanks to Guillaume Maudoux for his good will and fantastic advice, as well as for providing us with sources and tools that ended up being quitessential to one of the sharpest turns OzDoc took during development.

Last but not least, we must express our profound gratitude to our respective families and all of our friends and colleagues from university for their constant support and encouragement, and for making this project a very fulfilling experience. This accomplishment would not have been possible without all of them. Thank you.

Simon Alexe and Péter Frenyo

# Abstract

The Oz programming language has been helping students in learning and easily understanding the intricacies of all the different types of programming paradigms and their associated programming languages, particularly at Université Catholique de Louvain, where several courses use Oz as a . Such an important learning instrument could not be left without a proper documentation generator, a feature taken for granted by most popular languages. For that reason, we have created OzDoc.

OzDoc has been inspired by modern state-of-the-art documentation generators. This paper explores the selling points and downsides of contemporary popular documentation generators in an attempt to take the best out of each and avoid anything potentially unfavorable. The focus is mainly on the design of the best possible parser and documentation file generator, for which we make several case studies.

We present the evolution of the OzDoc tool in time, what lead us to take specific implementation decisions concerning the design of OzDoc, and what the tool's final version became. The grammar, lexer, parser and file generator are all explored in details.

The end result is an incredibly flexible, robust and expandable framework, that is also compatible with other languages. Testing showed that the final documentation files are still considered a bit too verbose, but the great modularity and adaptability of the tool will allow anyone to comfortably improve it over the following years thanks to the feedback of users, researchers and students.

A user manual is available at the end to the user who simply wants to the use the tool, and a developer manual is also provided for anyone who wants to contribute to the growth of OzDoc.

# Contents

# List of Figures

# List of Tables

# Acronyms

**ANTLR** ANother Tool for Language Recognition.
**API** Application Programming Interface.
**AST** Abstract Syntax Tree.

**CSS** Cascading Style Sheets.

**DocGen** Documentation Generator.

**EBNF** Extended Backus-Naur Form.
**EPL** Ecole Polytechnique de Louvain.

**GPL** GNU General Public License.

**HTML** HyperText Markup Language.

**IDE** Integrated Development Environment.
**IEC** International Electrotechnical Commission.
**ISO** International Organization for Standardization.

**JDK** Java Development Kit.
**JS** JavaScript.
**JSON** JavaScript Object Notation.

**lexer** Lexical Analyzer.

**MOOC** Massive Open Online Course.

**NLP** Natural Language Processing.

**OzDoc** Oz Documentation Generator.

**PDF** Portable Document Format.
**PLDC** Programming Languages and Distributed Computing.

**RegEx** Regular Expression.

**repo** Node Repository.

**SQL** Structured Query Language.

**UCLouvain** Université Catholique de Louvain.

**XML** Extensible Markup Language.

**YAML** Yet Another Markup Language.

# Chapter 1

# Introduction

We live in a world ever more driven by computers and technologies. IT projects are growing bigger, involving more programmers and having not only more lines of code, but also more complex ones. With society focusing on a short development time paired with a high return of profit, it is nowadays expected to reuse already existing code for problems of similar categories.

This is where a good documentation comes into play. Written by programmers for programmers, a good documentation reduces what could be a day long exercise of digging through code down to a few minutes, especially when there is no way to ask the author for guidance. Documentation primarily aims at giving a clear overview of all aspects of the project. Its foundation usually lies in listing the different methods, defining their inputs and outputs as well as giving example uses.

To ironically close the loop, documentation itself is nowadays generated automatically by *documentation generators*. Programmers write comments in the code itself, comments that are then parsed and compiled by a tool to generate crisp, clean documentation on a beautiful HTML page where the function of just about everything contained in the code is detailed.

Many languages come with their automated documentation generator. At Université Catholique de Louvain (UCLouvain), inspiring computer scientists learn Java and Python, which respectively come with Javadoc and Pydoc. Taught by Prof. Peter Van Roy, Oz is a pedagogic programming language that covers different programming paradigms, making it a very effective learning tool for future graduates. Being mostly used for academic purposes, Oz does not come with its documentation generator, and documentation for tasks and group projects is written by hand.

# 1 Context

The subject of this Master's thesis is the development and implementation of an automated documentation generator for the Oz programming language. We aim for our tool to be used by students, assistants and teachers of the courses using Oz at UCLouvain, namely LFSAB1402 - Informatique 2 and LINGI1131 – Computer Language Concepts, both taught by Prof. Peter Van Roy. We also hope that our tool will be powerful enough to be of use to the Programming Languages and Distributed Computing (PLDC) Research Group [1], whose activities center around increasing the expressiveness of programming languages, and whose research oftentimes calls for the Mozart Programming System, a full-featured development platform based on the Oz multiparadigm programming language.

**Where does the idea of OzDoc come from?**

April 2018, it was time for Master students to choose the subject of their thesis and for 3rd year Bachelors to submit the so-called *Pacman* group project in Oz for the Computer Language Concepts course. As we were working in one of UCLouvain's computer rooms, we overheard a student complaining about how hard it was to work on a project with scarce documentation. Shortly after, we had a good laugh with a teaching assistant for a different course on how tedious the correction of students' projects was because of their lack of comments and documentation. These two events gave us the idea of OzDoc, that we immediately proposed to Prof. Peter Van Roy. We wanted to help both parties and add a new dimension of rigorousness to Oz's documentation. The subject of this thesis is then less of an original idea and more of a concrete solution to a contemporary problem that we encountered.

# 2 Objectives

The two main parts of this thesis, as its name suggests, are the *development* and the *implementation* of the aforementioned DocGen. We use the term *development* to describe the whole research and design process as well as the methodology, and the term *implementation* to describe the concrete building process of the tool, pushing it from concept to reality.

The 3 main objectives of the development process are as follows:

1. **Research on the subject of Documentation Generators** : Have a complete understanding of the different state-of-the-art examples we currently have in automated documentation generation. Identify the most widely spread functionalities and how they are implemented.

2. **Research on the subject of Parsers** : Following this thesis's ideal of reusing what already exists, we should aim to research what parsers could be used for Oz instead of making a new one from scratch, or make a robust one ourselves.

3. **Designing the code of the tool** : To be understood in a Software Engineering way: deciding the language to be used, along with the architecture of the code, level of modulability, choices of implementation, atomic separation of components, etc. And, of course, a good documentation.

The main objectives of the implementation process are as follows:

4. **Creation of a Parser** : Making of a tool capable of parsing the whole Oz programming language syntax plus predefined comments using standardized keywords. Modulability is a plus.

5. **Creation of the Documentation Generator** : Takes the output of the parser as input to build crisp, clear and complete documentation.

6. **Performance evaluation of the tool** : Analyze performances in domains such as compilation time, execution time, correctness.

7. **Integration to Emacs** : For the tool to be easily accessed by users of Emacs, it must be fully integrated it to the last version of Mozart2 on Github.

8. **Writing of a User and a Developer Manual** : For students and teachers alike, the User Manual should be exhaustive and simplify the use of the tool, while developers should comfortably be able to contribute to OzDoc's advancement.

And the two aforementioned final goals are:

9. **For OzDoc to be used by students and teachers of UCLouvain**

10. **For OzDoc to be used by the PLDC Research Group**

# 3 Structure of the thesis

We wish for this manuscript to not only describe the end result but also the methodology that lead to said result. The structure thus follows the main objectives. The thesis is split into 5 Chapters:

1. **Introduction** : Describes the thesis and establishes the context, its scope, its objectives and its structure.
2. **Research** : A little history course on the subject of documentation generators, and a study of state-of-the-art examples. Research and decisions taken on the subject of parsers.
3. **OzDoc** : The bulk of this thesis. Includes the design and implementation of everything from the documentation generation process and the parser to the documentation generator itself. Shows the documentation produced as output. Integration of OzDoc to different platforms.
4. **Evaluation** : Performances such as compilation time, execution time, . Testing of the different functionalities.
5. **Conclusion** : Discussion of the issues and future improvements. A complete review of the tool and a proper ending to the thesis.

And an Appendix containing:

A. **User Manual** : How to install the tool and how to use it. The predefined keywords to use in the comments of your code and their functionalities. The produced documentation and how to modify it.

B. **Dev Manual** : How to modify the tool and implement new functionnalities. How to use the already implemented framework to parse and generate documentation for any language.

# Chapter 2

# Research

In this chapter, we will explore the history of the good and bad parts of the most popular documentation generators. All of this research is what lead us to the final design of OzDoc, by trying to take the best of every world with the options available to us.

## 1 A bit of history

Technical documentation plays an essential part in the development and release of any product for people working on the project and end-users alike.

### The birth of documentation

Almost 2000 years ago, people already documented their inventions and tools the way we do. The first signs of technical documentation go back as far as 200 BC[2], with the Antikythera Mechanism, an ancient Greek machine capable of predicting astronomical positions and eclipses years in advance. Later on, most documentation we can still find is mostly in the form of patents. What we're interested in, of course, isn't patented design and copyright, but rather the way to make a clear, beautiful and easy-to-understand documentation. Specification documents are mostly where we turn our attention. To give a concrete historical example, the Difference Engine, a computer from the 1800s made to tabulate polynomial functions, received numerous improvement by other researchers to become the more general-purpose Analytical Engine, a calculator made for a wider mathematical usage, thanks to the in-depth

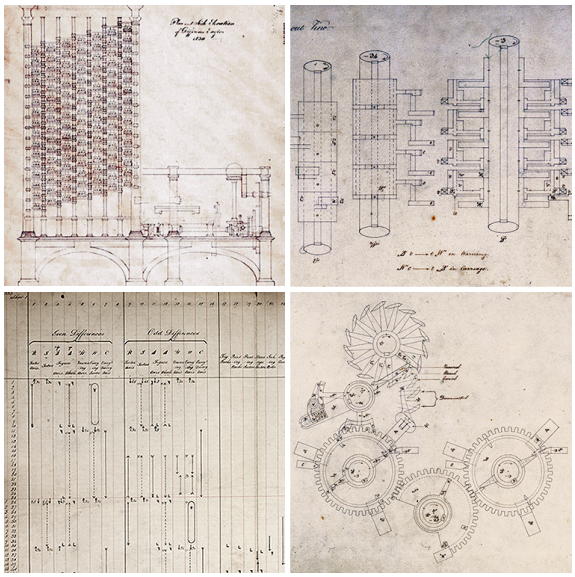

Figure 2.1: Charles Babbage's documentation for the Difference Engine[43]

documentation left by its creator, as visible in figure 2.1.

### The first documentation generator

With the realization of the importance of documentation and the emergence of technologically advanced computers, people turned their attention towards automatic documentation generation, especially for computer programs. The first popular automatic documentation generator to have surfaced, chronologically, was mkd (mkdoc) for UNIX in 1989[3]. It was developed at Université Montpellier II at a time when there was no known software for documentation usable with several languages (like Assembly, C, Fortran, etc.). It presented a rather rudimentary documentation system for UNIX programs: it generated software documentation according to the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) standards by simply copying and pasting all of the comments written in the code in order, so as to end up with text in a sort of "organigram" format ending up in a text file. The options given as arguments allowed to detect the type of comments depending on the programming language. mkdoc has since then taken back its old name, mkd, and has been evolving quite a lot due to being open source.[3] These simpler models are where it started, and are still used today for Unix packages. This shows us that sometimes, going for the simpler solution can still be very effective.

### The most widespread

In 1995, 6 years later, one of the most popular documentation generator, Javadoc, would be born. To this day, Javadoc is still widely used all over the planet and is still being improved on. Contrary to mkd, its output is in HTML format, allowing for a separation of files via hyperlinks and a hierarchical model for programs[4]. In 2000, 5 years later, we would have PyDoc, which is surprisingly similar to mkdoc but outputs a pretty HTML or a text file[5]. Both being references in documentation generation, we will have a more exhaustive look at Javadoc and PyDoc in section 2.

### Different levels of automation

Our case study of modern documentation generators brought to light differences in their core. To highlight said differences, we have decided to classify modern DocGen's automation levels into 3 categories, namely:

- Partially automated documentation generators
- Highly automated documentation generators
- Fully automated documentation generators

On the lower end of the automation spectrum, we have *partially-automated* DocGen tools, which we also like to call *user-controlled* DocGens. By that, we simply mean that the documentation generation tools aren't automated to a point where you would only need a carefully commented code for it to work. Most of them simply act as a framework or Integrated Development Environment (IDE) to write your documentation in a prettier way. There are indeed many popular Application Programming Interface (API) documentation generation tools that aren't fully automated and require quite a lot of input from the user, some having become popular among users. MkDocs (note the *s* at the end, different from mkdoc), for example, is a static site generator written in Python geared towards building project documentation. All the information still needs to be provided in information files written in Markdown and a Yet Another Markup Language (YAML) configuration file, but the pretty site is getting taken care of by the application[6]. Even more popular than MkDocs is Jekyll, another static site generator, and its Documentation themes, written in Ruby[7]. In the same category, OpenAPI, previously known as Swagger, now open source and being worked on by Oracle, Adobe, SAP, SalesForce and others, also provides Web API documentation generation by writing a configuration file in

JavaScript Object Notation (JSON) or YAML[8].

A bit higher on the scale when it comes to automation, we have *highly automated documentation generators*, which regroup most of the DocGens we've talked about so far (mkd, Javadoc, Pydoc, or even the final Oz Documentation Generator (OzDoc)). These mostly use tags inside comments of the source code (or in a separate file) to generate documentation. At first, we used to refer to Javadoc and the like as *fully automated documentation generators*, but we realized during our research that an even higher level of automation (true fully automated DocGens) were already in the works in the world of DocGens[9;10;11].

The last remaining category is true *fully automated documentation generators*. These use languages processing methods and data mining techniques to automatically generate documentation in natural language for the user.[12] These will be explained in a bit more detail in section 2.

### Documentation generators for Oz

An important side note that needs to be mentioned is the fact that during our research, we have also discovered that the name "ozdoc" already used to be present in versions of Mozart prior to 2.0 (1.4 and below)[13;14]. The ozdoc function was a partially automated DocGen (using an Extensible Markup Language (XML) file as input for the metadata of its specifications) that provided rudimentary markup technical documentation for Oz. This feature was abandoned when upgrading to Mozart2.[14]

### The present and future of DocGens

Nowadays, lots of research is being made in all 3 types of documentation generators. Even as recently as last September (2018), the Third International Workshop on Dynamic Software Documentation (DySDoc3) in Mardid held the first Software Documentation Generation Challenge.[15] Documentation generation tools' improvement gains more and more traction as time goes on, and the increase in open source projects being worked on by big and small companies (with the likes of OpenAPI) is helping this advance by a ton.

# 2 State of the Art

Defining what "state of the art" means in the context of current documentation generators can be a little tricky. It is still a matter of preference, and depends on the type of project and the context. No single DocGen is unanimously called number one.

We will thus discuss the most popular documentation generators we can find nowadays. Interestingly enough, there are contenders in all 3 categories of DocGens (described in the previous section on different levels of automation).

Partially automated ones are highly popular with Web applications because of the absence of a good JavaScript DocGen[16], and thus probably the most popular one when it comes to companies and businesses (which is not really our center of interest since we're looking for a documentation generator for the Oz programming language).

Highly automated DocGens are mainly used when coding the usual Java, Python, C, etc., be it by the lone developer by hobby or by companies specialized in software.

Fully automated DocGens are more of an experimental category of generators with a quite promising future, considering how fast the domain of machine learning, data mining and natural language processing is currently evolving.

### Case study : Javadoc

Due to the simplicity, availability and popularity of Javadoc, a deeper case study only seems natural. Do note that most of the information in this part about the choices and thought process behind Javadoc comes from the wonderful case study of Javadoc by Douglas Kramer, one of its creators. First of all, unlike in partially automated DocGens, Javadoc decided to choose the route of "documentation comments", or "doc comments", which other documentation generators rather call "docstring" nowadays. This means that instead of putting the instructions to generate the documentation in a separate file, the instructions are written in the source code itself via specially marked comments that describe the code and is intended for extraction by the Javadoc tool[17]. Those comments begin with `"/**"` and end with `"*/"`, and its content applies to the declaration that immediately follows it. This is different from mkd and PyDoc, where the comment refers to the previous (and not the next) declaration.

This is the center of a debate concerning whether or not API specifications belong in another file or in doc comments with the source code. There is sense in the option of separating the specification and the implementation, leaving them in the same workspace but in two separate files: it would raise visibility of specification changes, and would emphasize the fact that doc specifications should not be changed lightly (and implemnetation changes should follow with it). It does, however, act as excessive overhead, and leaving doc comments in the source file ends up taking less space and being more convenient, hence the choice of keeping it this way in Javadoc. Not only that, but with separated specs and implementation, programmers that aren't familar with the code will have a harder time trying to look at both files instead of just being able to look at the specs of the function they're about to see right above their eyes. Programmers might also be more inclined to get lazy and not modify the specification file, leading to even worse documentation. For a programmer, the documentation and the code definitely need to be bound, to help him or her implement the specification. Furthermore, keeping doc comments and source code together also guarantees that the Javadoc tool generates up-to-date accurate API signatures (only the doc comments might be out of sync with the implementation, not the API signature). It was thus decided for Javadoc to keep API specs with the source code via doc comments.[17]

Another important part of Javadocs development and improvement comes from the choice of the audience it was aimed at. A greater focus has been put on the API specification documentation than on the API development documentation. Both are taken into account, but applications developers are far more numerous and thus hold a more important place when it comes to needing documentation. API developers would need more elaborate comments, with examples and bug documentation, etc. This has decided to be scratched for a better general usage.[17]

The documentation in Javadoc has thus been reduced to comments in the source code that must follow conventions and look a certain way. This plays the role of establishing guidelines so as to unify the documentation. Indeed, people were writing documentation however they wanted and code was all over the place before unification of API specification documents by Javadoc. This, in conjunction with a focus on cross-platform compatibility (with the slogan "Write Once, Run Everywhere"), allows for an easy-to-use and working documentation working on any machine without needing any rewriting.[17]

In the end, all these choices leave us with a Javadoc which, in some way, might

| Tag | Meaning |
|---|---|
| @author | Identifies the author. |
| {@code} | Displays information as-is, without processing HTML styles, in code font. |
| @deprecated | Specifies that a program element is deprecated. |
| {@docRoot} | Specifies the path to the root directory of the current documentation. |
| @exception | Identifies an exception thrown by a method or constructor. |
| {@inheritDoc} | Inherits a comment from the immediate superclass. |
| {@link} | Inserts an in-line link to another topic. |
| {@linkplain} | Inserts an in-line link to another topic, but the link is displayed in a plain-text font. |
| {@literal} | Displays information as-is, without processing HTML styles. |
| @param | Documents a parameter. |
| @return | Documents a method's return value. |
| @see | Specifies a link to another topic. |
| @serial | Documents a default serializable field. |
| @serialData | Documents the data written by the **writeObject( )** or **writeExternal( )** methods. |
| @serialField | Documents an **ObjectStreamField** component. |
| @since | States the release when a specific change was introduced. |
| @throws | Same as **@exception**. |
| {@value} | Displays the value of a constant, which must be a **static** field. |
| @version | Specifies the version of a class. |

Figure 2.2: All possible tags in Javadoc from Java Development Kit (JDK)8 [44]

be considered state-of-the-art DocGen for the moment being, despite being so old. Its principles are indeed still used all over documentation generators, and Javadoc itself is still as popular as ever. It implements a lot of useful tags in defining the specifications of a function, as visible on figure 2.2. The specs are entirely defined by the doc comments and any reachable documents marked as specifications, via the `@see` tag (in other words, any specs that do not live in the doc comments must be reachable from the comments, via hyperlinks or by naming the title). [17] Forcing doc comments to be written in a particular way puts everyone on the same page, and thus, forcing syntax can be a good thing.

Of course, these type of specs do rely on the programmer to make an API containing sufficient assertions and specifications of boundary conditions, corner cases, etc. They do also rely on the programmer already having a certain knowledge from other programming languages (doc comments do not define each and every common computer term), but it needs to target the vast majority of users, i.e. avoid using obscure, scientific terms for no reason but rather use a clear writing style while being consistent with most existing doc comments. If any external documents still exist outside of the API specifications for any small part of the code, they should be specified and the exact interesting part should be linked to, as a ”See also”.

The specifications need to be platform-neutral, free of any implementation-specific

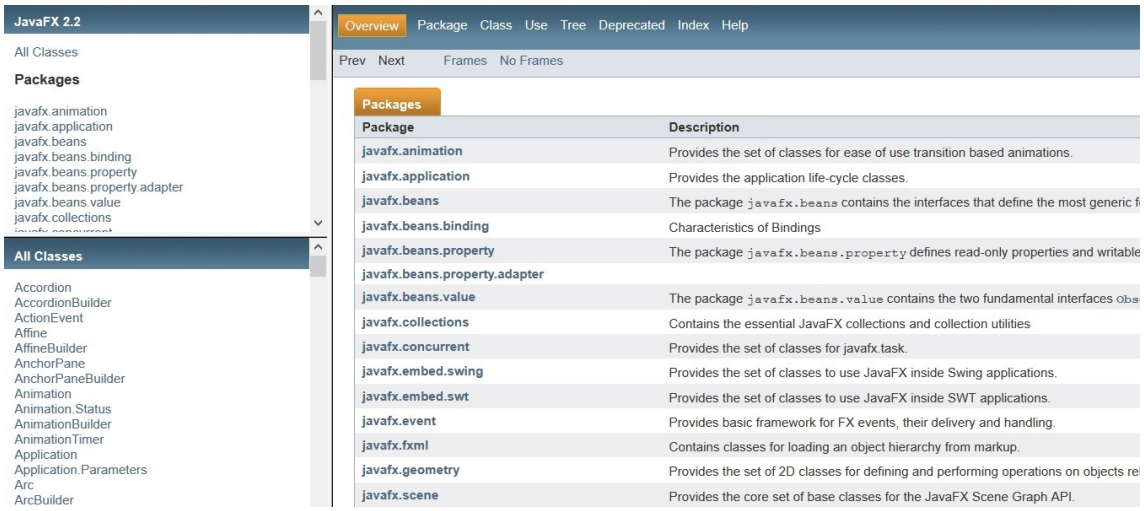


Figure 2.3: Javadoc's Standard Doclet applied to the JavaFX 2.2 package[45]

detail. If adding any platform- or implementation-specific detail, one can separate it via a NOTE paragraph.

Most people use the standard HTML doclet, which makes the default page we all probably know (as it ships with the Javadoc tool), which is visible on figure 2.3, but many other doclets can be used or even created. Tutorials on how to make pretty HTMLs and which tags to use are also readily available.

Javadoc is still improving (hence why people are still using it), for example with the addition of style markup classes to control the styles of the generated pages, allowing you to provide an alternate Cascading Style Sheets (CSS) to customize page navigation, headers and footers, the content and the overall document[18], and later on support for several stylesheets at the same time[19], but also updates for the traversing of Javadoc comments as abstract syntax trees[20], HTML5 support, a search box, etc.[21].

Despite all these positive features, there are actually several reasons why Javadoc's model for documentation generation is kind of flawed: the fact that Javadoc forces a certain model for doc comments and absolutely needs certain fields to generate documentation for a function will oftentimes lead to a huge part of the comments giving absolutely no added value, and most of the time being generated automatically by the Javadoc wizard for documentation comment generation, leading to bland comments with a lack of context and often not any more useful than the name of the variables and functions already are. Not only that, but the check for correct input and important edge cases are better checked with asserts than with comments in a documentation. Taking screen space in the source code also hinders readability.

As time passes, and code gets updated, programmers often forget to update the comments in the specifications along with said changes, leading to inconsistencies. Refactoring is also made slower and more difficult when making changes in a code using Javadoc, due to the fact that it must be done manually.[22]

Despite all this, why is Javadoc still one of the most popular documentation generators, and used so widely throughout the world? Simply put, it's easy and it works. Support for Javadoc is available in just about every IDE and available for just about any Java library. As metnionned, having doc comments in the same file as the source gives a higher chance of having up-to-date specifications, and takes out the need to retype information in two separate files. It's just convenient. In fact, the main attraction of Javadoc right now is its popularity. Coming up with something better means competing with Javadoc's universal availability and ease of use. Despite the look and technologies that could use a makeover, Javadoc is just so easy to pick-up for anyone and very natural to use, and will be compatible on anything and everything. Most IDEs also provide a ton of tools that improve their use with Javadoc. Although, it could use something to make programmers care more about doing their documentation seriously.

**Case Study : Doxygen**

If you want to use another popular documentation generator, compatible with not only Java but also other languages, Doxygen is very a popular choice and might be the way to go. Doxygen also use comments from the source file to make documentation. However, it provides a lot of features that Javadoc doesn't have, like class diagrams for hierarchies which is the main attraction for many people[23], but also support for additional tags like `todo` on a separate page while being entirely compatible with Javadoc tags (in fact, both can be used together without hindering each other, as Javadoc just ignores unknown tags, while Doxygen knows all Javadoc tags), output generation in Portable Document Format (PDF) and TeX format as well, additional summary pages, source-code browsing, and a lot of visual customization.[24]

This all makes it seem like Doxygen is just a better Javadoc yet it is visibly still used a lot less. The reason being the same again: popularity and simplicity. Doxygen offers a lot of interesting additions, but is painful to set up to work correctly. If set up correctly, it should provide you with a lot more formats and can be a

better documentation, but not a lot of things are well organized by default like in Javadoc. For example, `index.html` seems to default to a blank page if not set up correctly. Not only that, but most IDEs, as mentionned, already have great support for Javadoc, while not having the same for Doxygen. The best IDEs' tools and plugins for Javadoc allow for an even better visualization, easier navigation and greater maneuverability of your code and documentation.

### From highly automated to fully automated

Now that was for state-of-the-art highly automated documentation generators, but there is still one we haven't talked about yet: fully automated documentation generators, which was, at first, our aim for OzDoc. These ones do not require any special comments from the programmer, and intelligently work out the role of most things in the code by themselves.

In a paper by *McBurney et al.*[11], a state-of-the-art fully automated documentation generator is tested and compared to other DocGens of the same genre. They oppose it to what they call "manual" documentation generators, which we call partially or highly automated ones, i.e. ones where the generator needs to analyze either some external metadata or the comments in the source code. This time, we analyze the source code itself, the aim being to stitch keywords from the source code together to make natural language sentences[9;10]. While most of these fully automated DocGens went no further than what we just said, the one from *McBurney*'s paper tried to add an important plus to this: a context detector. In an attempt to analyze the reason for the existence of a function or method, they analyzed how the Java methods are invoked so as to create context information to make better natural languages descriptors for said methods. By context, we mean the dependencies of the method, and any other methods which rely on the output of the method.[11;12]

Indeed, as documentation generators depending on comments written by the programmers require a lot of time and energy from the programmer when writing the documentation, and needs constant updates, they put quite a burden on the programmer to make the documentation good and up-to-date.

To make a good, usable fully automated DocGen, the addition of context detection is important. This was done by applying the famous PageRank algorithm (the one made and used by Google)[25] to locate the most important methods as according to the number of calls, then gathered relevant keywords describing the behavior of the

most important methods and used a natural language generation system to create natural language text about the context in which the method is called. This is important because it helps answer questions about why a method exists and what role it plays in the software.[11;12]

The results of these fully automated DocGens were pretty good. It was effective in giving programmers information about what methods do internally, why they exist and how to use them.[10] However, many complaints emerged from the descriptions in the documentation being way too verbose, confusing and with too many details, sometimes not very useful ones. Automatic generation leads to more information than we need, and sometimes not the right one, but overall brings about pretty good results.[11]

# 3 Qualities of a good documentation generator

After checking the state-of-the-art documentation generators from all 3 categories (partially, highly and fully automated DocGens), we can cherry-pick the best parts of each and mention the parts that should be avoided, so as to determine the characteristics of a truly good documentation generator.

### Simplicity

Probably the single most important characteristic we can find in everything we've said so far, being basically the sole reason why Javadoc is so popular, is obviously ease of use. This is exactly why, even though Doxygen should technically be better than Javadoc, the latter is still the most popular and used one.[1] It basically all comes down to *simplicity*. And indeed, sometimes staying on the simple side can pay off: mkd may be from 50 years ago, its very elementary principles are still the foundation of the popular PyDoc. Basically, we need documentation that is easy to pickup and easy to maintain. Minimizing this hassle can be done with a few tricks: first off, we absolutely need platform-neutrality for our documentation generator. In the case of the Oz programming language, OzDoc needs to work just as well no matter the operating system (and also no matter the IDE or editor used). Little to

[1] This is based off of observations on the likes of StackOverflow and Reddit, where many people complain about the hassle of setting up Doxygen, as well as the decreasing popularity of Doxygen as visible on Google Trends

no setup should be required. That is, compatibility with any system is preferable, and implementing support with the most famous IDEs (with, at the bare minimum, some compatibility plugins and tools) is a huge plus.[17] In the case of OzDoc, it simply means implementing it into Mozart and maybe make some plugins for famous text editors like Atom and Notepad++.

We can safely assume that just about every programmer is lazy. Programmers are not willing to go out of their way for something they haven't heard of. There are thus two ways to lure a programmer into finally using a documentation generator to avoid the loss of some precious hair for the next person who would have to read and modify the code the first programmer has written: simplicity or efficiency. Either the documentation is so simple that it only requires a single click (like with a fully-automated DocGen), or we showcase some examples of beautiful documentation generated without much of a hassle (like by having an elegant, official Oz base API generated via OzDoc) to the programmer. A combination of both is best.

For those reasons, it is preferable to keep the doc comments in the source code. Putting them in another file makes the task of picking of someone else's code (or even documenting your own) daunting, and if people are intimidated by something, they get even lazier. While having the necessary features for documentation generation in another file would help having good and updated documentation when there is any, it would also discourage many programmers from doing it at all, so it is preferable to give privilege to having a documentation at all. However, it is important to note that readability of the source code shouldn't be sacrificed for the sake of convenience. Polluting the code with an unnecessary amount of mandatory comments might make the source code much harder to read and artifically increase the number of lines, sometimes for no apparent reason (i.e. bringing nothing to the table). The model for mandatory comments should be one that makes programmers write useful information only, and avoid redundance (especially when it just leads to everyone repeating variable names to satisfy the DocGen, e.g. by forcing to write what ends up being the likes of "nbr_uses: the number of uses", "powerlevel: the level of power", etc.).[22] The comments and tags shall thus always be optional, and the DocGen should have enough automation to make an interesting and helpful documentation even out of a comment-less code.

### Maximize usefulness

It is needed to encourage programmers to do good documentations: for that reason, tutorials and examples of pretty files should be readily available and recommended without annoying the user, so as to show how simple it is to use the documentation generator to get beautiful results. To that end, the most convenient aspects of the tool should be on the spotlight, like the very handy search function that is available thanks to HTML5.

Speaking of the output, the information on it should lead to beautiful, clear, convenient and easy-to-understand documentation. Forms of customization of the output are also appreciated by the community, but homogeneity between at least all the default forms of different documentations are a must. This makes it easier to read for everyone. This also means that some of the freedom of the user must be limited, so as to impose a few key technicalities for the sake of consistence. Giving options about what to show and how to show it will please the more advanced users, while the beauty of the default pages will encourage ordinary users to test the tool and document their code correctly.[17]

### Avoid boilerplate

When it comes to the content of the specifications, especially if going with a fully automated documentation generator, the information should be easy to read and useful (i.e. not too verbose for no reason). The "why" of a method is to be insisted on much more than the "what". This can mainly be done by explaining the dependencies and context of the method, where is it called, what does the method itself call and where is the method called.[11]

The DocGen should, however, encourage the programmer not to rely on it for everything, especially those that could be done better in other ways. This includes important assertions, which should be written as such instead of just mentionned in the specifications when needed, and boundary conditions, which can also simply be tested in the code.[22] This will reduce the amount of boilerplate and enhance readability, while giving a sturdier code which doesn't depend on something as unreliable as the goodwill of the programmer. However, part of this problem is mainly to be dealt with in the language itself rather than in the documentation.

### Quirky selling point

Additionally, a set of unique features need to make the generator attractive for people to use, or else users will just choose any other one available. For example, smart highlighting of the scope of variables when hovering one's mouse over it can be a little detail that might attract some people. Other quirks can be taken from popular DocGens, like a class diagram generator to visualize hierarchies[23], advanced summary pages and extra tags like the `@todo` tag in Doxygen[24]. A bigger focus can be made on correcting the mistakes of other popular documentation generators, like a focus on making the generator robust against refactoring so as not to have the problems Javadoc has[22].

### Expandability

Finally, as obvious as it may seem, the documentation generator should also be "modulable", meaning flexible and easy to modify and extend, and leave room for evolution. We believe making the tool open source can lead to a lot of improvements and great results, and could only be benefic to the user base.

# 4 Inner mechanisms: Lexer and Parser

It's a good thing to describe the input and output of a documentation generator, but there is an important part in the middle that has yet to be mentioned in our research, yet was the very first stepping stone for this thesis: the necessity of an Oz interpreter (i.e. a lexer and a parser) for our documentation generator. Indeed, we have yet to know how a documentation generator works.

Taking the example of a highly automated documentation generator with documentation comments in the source code, a documentation generator would take the source code as input and parse its content. To that end, the code needs to go through a lexer to tokenize the characters, then through a parser to make an abstract syntax tree or parse tree from the source code, then finally it can go through the final output generator and the beautiful HTML file can be born. But writing a lexer and parser from scratch can be hard, so we had to analyze the options available to us, and the characteristics of good parsers.

### Lexer and Parser

A Lexical Analyzer (lexer)'s first role is one of a tokenizer. The lexer will make tokens out of the characters by using Regular Expression (RegEx), searching for separators, typically whitespaces of any kind. The lexer also attaches context to the token, to describe the type of said token.[26] These will become the atoms of the parser, the base unit it will use to create a logical structure.[27]

We can either write our lexer manually or generate it by some way. In our case, since we are talking about Oz, we focused on manual lexers by using the entirety of the Oz grammar available in the canonical book.[28]

The parser will take the stream of tokens given to it by the lexer and transform it into an organized, hierarchical data structure, generally some kind of parse tree or Abstract Syntax Tree (AST), allowing to organize the data at the same time as it's checking the syntax.[26] This AST will be used by the documentation generation tool to create the final HTML file.

One important thing to remember while writing the grammar rules for the parser is that said parser cannot check for semantics! A parser only checks for syntax. That means that when you're not sure what to do with some token, it's probably better to let the parser pass the content to the program, and check the semantics to make sure the rule has a proper meaning later on.[27;29]

Lexers and parsers go hand in hand, we'll be talking about them together, and will mostly refer to them as simply the "parser" as a common metonymy, as the parser is the final and biggest part of the bunch. When it comes to creating a parser, several options are available. One would be to create one's own parser, like we students have learned in the Languages and Translators course at Université Catholique de Louvain (UCLouvain)[30]. The other one would be to use a lexer and parser generator. The most popular one, even used by hypergiants like Twitter[31], is a completely free software called ANother Tool for Language Recognition (ANTLR) (specifically, ANTLR4 in our case).

### Grammar

The gist of lexer and a parser is in the grammar it uses to parse the language. Grammar rules describing the language need to be written to make a parser. Two methodologies are available to achieve a good grammar: top-down or bottom-up.

A top-down approach goes from the final product slowly towards details. The first rules to be written are the top-level ones, like the rules representing the entire file. The focus is on the general organization of the file.[27] This approach is preferred in a case similar to OzDoc, when we already know the final form of the language we're designing a grammar for.

A bottom-up approach focuses on the low-level elements first and gradually builds the language from basic expressions and such. However, the focus thus becomes less on the parser and more on the details of the lexer.

### Why RegExes aren't enough

All of the previous steps are necessary regardless of whether you make your own parser or use a parser generator. However, one might consider another route for the parsing job: RegExes. For the case of OzDoc, or any DocGen, RegExes seem like a fair choice. While it could work for some fairly simplified languages, there are a lot of limitations, like the lack of support for recursion among other things. It has even become a recurring joke among web developpers that one should never attempt to parse HTML using RegEx, or that person might doom humanity.[2] This is the reason why we need true parser.

### ANTLR4

When it comes to choice of which to use to make parser itself, ANTLR4 being a very powerful and free tool makes it the obvious choice for making a parser the easy way, as it does indeed let one avoid the daunting task of creating a parser from scratch. For simplicity's sake only, we can see how attractive an ANTLR4 parser is.

In ANTLR4, grammar rules are all written for context-free grammars. That is, the rules are recursive, and there is always only one element on the left-hand side of the rule. On the right-hand side, rules are written with a syntax very similar to regular expressions. Of course, in the case of a documentation generator, we need to write a language that does not skip the comments and actually processes them.[32;31]

So far, ANTLR seems like the perfect solution to anyone's problems. However, there are also quite a few limitations when working with ANTLR, starting with the installations needed by the user. At the very least, a runtime would be necessary

[2] "RegEx match open tags except XHTML self-contained tags", https://stackoverflow.com/a/1732454

even for the user to run the parser, and that's simply not convenient.

Another point is that it not only forces the output to be an Abstract Syntax Tree (AST), but it also obligates you to travel said AST with an object following one of two design patterns: either a Listener or a Visitor. A Visitor allows for a little bit more freedom than a Listener, with the possibility of controlling the flow and return elements at each node through a function associated to said node (for example to generate code), but they are still limiting us in terms of what we're able to do with our AST.

Yet another point, closely linked to the last one, is that the movement through our parse tree can only be done through one type of algorithm: depth-first search. If we wanted to implement a documentation generator using breadth-first search or any other algorithm, we simply couldn't. In that regard, writing one's own parser gives a lot more freedom.

### Homemade Parser

If one were to write his own parser, it would be wise to take inspiration from parsers used by other documentation generators. For instance, we can notice that most parsers used by documentation generators can parse the documentation generator itself: PyDoc is written in Python and can thus parse PyDoc, Javadoc is written in Java and can thus parse Javadoc, etc. These kind of work like bootstrapping compilers, like Java currently has. Coding OzDoc in Oz should at least be considered.

Speaking of Java, it can be interesting to see how Javadoc parses Java code to make documentation. As Javadoc is open source and under GNU General Public License (GPL) license, the code of the entire JDK is available for everyone to see. We can see that it uses javac's API to use Java's own parser, which is currently written in Java as well (but used to be written in C). A few options have to be tweaked, of course, like how comments should be kept now instead of ignored during compilation.

One last thing worth mentioning is the special type of grammar that we ended up using as inspiration for our parser: *island grammars*. An island grammar precisely defines small portions of the syntax of a language. The rest is left imprecise on purpose, and is called the "ocean" or the "water". These types of grammar lead to faster parsing (thanks to having statistically less ambiguities overall), more robust parsing (thanks to less dependence on language differences)[33] and what is called ignorant parsing (i.e. not requiring full knowledge of the syntax of a given language).[34;35]

# Chapter 3

# OzDoc

The OzDoc tool that you experienced or will experience wasn't always the way it is. It has been improved and modified as we buried our noses deeper and deeper into research. Before describing the final tool, it is important to know how and why we ended up with OzDoc being the way it is.

## 1 Evolution

### 1.1 A fully-automated documentation generator

The name of this thesis wasn't always the one you currently see. The original name of our paper used to be: "Development and implementation of a *fully-automated* documentation generator for the Oz programming language", which, at the time, was indeed the premise of this project.

The tool we originally envisioned started on the simpler side with, however, a key unique component: making Oz the first language to have a fully-automated documentation generator as official DocGen tool. Being students in Artificial Intelligence and having recently had a course on Computational Linguistics[36], we could envision a parser using a bit of Natural Language Processing (NLP) work out well if done correctly.

The earliest vision we had for the tool basically took in a source code in the Oz language along with a hard-coded syntax file describing Oz's grammar that would be given to the parser to get the abstract syntax tree needed to generate the documentation. Everything would already be set up, and a single click would do the entire

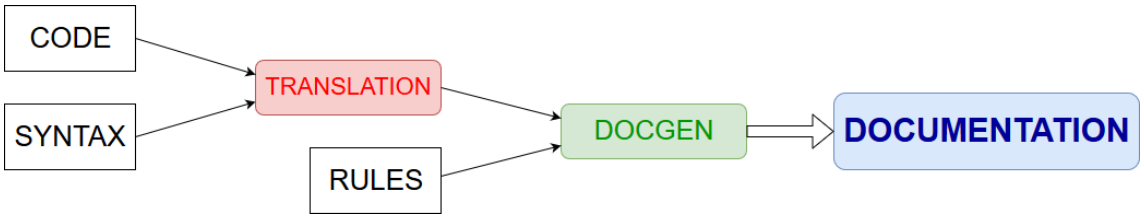


Figure 3.1: Early model representing OzDoc's framework

work. This would, however, require tools of NLP and assumptions made on the name of variables and methods, while not always leading to conclusive information on the source code.[12;11]

As the project evolved, however, we've started to want it to become widespread as well, and envisioned a fully "modulable" (i.e. flexible and easily expandable) tool allowing customizable styles of documentation for any language of our choice given a syntax, being able to generate all sorts of documents given tailor-made rules for the output file. This ended up giving us a framework we kept for a while, which you can see in figure 3.1 (where "translation" is now the lexer and parser, "syntax" is the input grammar, and "rules" is equivalent to style sheets).

However, even before creating something stylish like this, getting a good documentation tool to help students have a better understanding of instruction codes from their projects in their different Oz courses was higher on the list of priorities. Basing our work on popular DocGens seemed to be a better start, and lead us to the first implemented version of OzDoc.

## 1.2 Evolution of OzDocParser

When it comes to a grammar file for representing Oz, three main choices were visible:

- Writing it ourselves based on the canonical book[28], which we have learned to do in a course at UCLouvain about Languages and Translators[30], but would be very time consuming.
- Using the grammar, lexer and parser already written by students in 2016 as part of their Master's thesis called CorrectOz, a tool made with ANTLR to correct student's mistakes in Oz exercises from courses at UCLouvain.[37]
- Using the syntax available in Mozart2.[14]

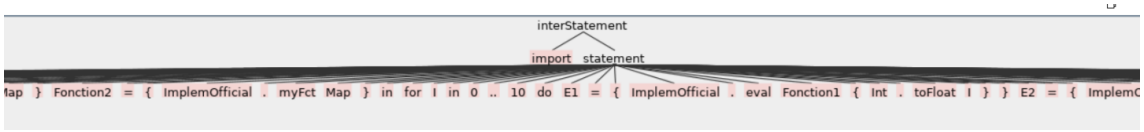


Figure 3.2: CorrectOz parser's resulting AST when meeting unrecognized syntax

### ANTLR Parser

For reasons of safety and simplicity, and mainly so we can focus on the intricacies of the documentation generator itself, we decided to go with the ANTLR4 grammar created by N. Magrofuoco and A. Paquot for their tool CorrectOz. This allowed us to continue working on other parts of our tool without worrying, and allowed us to feed the AST to the documentation generator, which was written in Python, but also to focus on other parts of the project, like the integration to Emacs. This did mean, however, that the user would need to have the ANTLR runtime for the target language as well, which takes out some of the "simplicity" component for the user.

While CorrectOz's parser worked well as long as we were using toy codes and short examples, its limitations appeared after we asked teaching assistants at UCLouvain for actual Oz code made by students during projects. Having received lots of anonymous codes of projects made in 2016, 2017 and 2018 (later than CorrectOz), some files having up to 1500 lines of code or more, several defects rose to the surface.

First of all, the parser became exponentially slower as file sizes rose. The parser was designed to detect and correct programming mistakes in Oz exercises in Massive Open Online Course (MOOC)s or the INGInious exercise assessment platform, which generally split the exercises into smaller parts requiring smaller codes. OzDoc would be the opposite: designed and needed mainly for bigger codes, so as to make them easier to understand. CorrectOz's parser thus needed approximately 10 minutes to parse a student's code made of 500 lines of code. Needless to say that for a documentation generator, these types of delays would be unacceptable. The time complexity of the parser needed to be optimized.

Secondly, the parser had an Achilles' heel with some of the syntax of Oz. Looking at the abstract syntax tree through a GUI, the entire tree structure would get ruined if a single piece of unrecognized syntax was met. These pieces of syntax include Oz syntax used in projects. For example, meeting a functor would always break the parser. Imports also had a similar impact. The effects on the tree stucture are visible on figure 3.2.

We thus modified the grammar file of CorrectOz's parser to fix the problems encountered, and make it respect general conventions of ANTLR parsers. After rearranging the grammar to meet conventions, tweaking it so it wouldn't ignore comments (since we are, in the end, making a generator based on documentation comments), making it skip unneeded information (expressions, statements, etc. don't need to be fully parsed) and making it recognize the few keywords it didn't, we were still left with the problems of speed and control, and were slowly starting to see the attraction in writing our own parser despite the workload it represented. While not only having this speed problem, we were also limited in the actions we could do with our Abstract Syntax Tree in that we were forced to use either a Listener or a Visitor design pattern to travel across our tree, which limits the freedom of our code. And while ANTLR4 is indeed used by the likes of Twitter for query parsing[31], the codes we work with are made of tens of thousands of words, which is quite longer than a query.

In the end, instead of staying stuck on the speed problem, we decided to code our own parser to have a tool that would be faster, need minimal installations from the user (and have minimal weight), and would give us more freedom when choosing what to do with the AST (for example, by opening up the option of cheating around any ambiguities a code might have that an ANTLR context-free grammar cannot easily fix).

### Oz Parser

The next iteration of OzDocParser was coded in Oz. Indeed, most parsers for languages are written in the language itself. That is why Javadoc or PyDoc are capable of parsing themselves.

Do note that we did consider using the parser used by Mozart2 itself[14], but not only did it parse too much information for us again (since we mainly need comment and can discard a lot of info), but the code for it is a mix of code from 20 years ago with eponymous code written only 5 years ago by students and PhDs of UCLouvain, which don't seem to interact at all, and was overall very confusing.

To facilitate the coding of the parser, we also had a prototype written in Python as a baseline, developed in parallel with only basic functions and based on island grammar[33]. An underlying problem of the Oz parser was reflected early on: not only did anything coded take much longer than the prototype to get written due to

the fact that some major documentation for Oz functions done through the years had dead links, but the obtained early prototype parser was fairly slow. That is, a lot slower than the Python prototype. This seemed to fit an underlying problem that has surfaced some years ago: the switch from Mozart 1.4 to 2.0 seemed to have slowed down parsing (this was, at first, an empirical observation made via student projects through the years). Indeed, the C++ parser was abandoned for an Oz parser on the official Mozart2 Github, so as to minimize extra installations. Ever since then, there have been complaints of slowed down code, and the question of going back to the old C++ parser remained.[38] For convenience sake, the Oz parser was kept, but the underlying problem of slower speed remained. The parser has not been touched since then.[14]

Our focus on widespread usage of our tool being at the highest on our priority list, we consider the question of speed to be quintessential. The writing of our parser in Oz not only multiplied coding time by a huge amount, but resulted in a loss of parsing speed as well. While it spited us to give up on the possibility of OzDoc being able to parse itself, we decided to make a hard switch to what used to be our failsafe, a Python parser. This had the extra advantage of allowing our DocGen to use powerful Python-to-Web tools as well as faster Python packages (thanks to the fact that most of them are implemented in C). Python being more documented and easier to use, this also gave us the possibility to make an easily malleable tool, another key aspect we were aiming for. However, the question of supreme convenience by having an Oz parser (which would allow the user not to install Python on their computer) is still on the list of priorities, but currently falls in the category of "*premature optimization*", meaning spending a lot of time on something that one might not actually need. And as Donald Knuth, father of Literate Programming, says it so well in one Pr. Peter Van Roy's favorite quotes: "Premature optimization is the root of all evil".[39]

# 2 The OzDoc Parser

The current OzDocParser is a Python tool coded from scratch. The tool uses an island grammar[33;34] (see 4) we have written for Oz. OzDocParser is, however, a modulable tool which uses a grammar file as one of its inputs, meaning it can already be used with other languages.

## 2.1 Grammar

We define the grammar considered by the parser as follows:

1. $S \rightarrow X_i \ S \ Y_i$
2. $S \rightarrow P_j \ \ text \ \ Q_j$
3. $S \rightarrow \epsilon$
4. $S \rightarrow \ text$
5. $S \rightarrow SS$

With a defined set of production rule given as input to the parser for each $X_i$, $Y_i$, $P_j$, $Q_j$ (only one set of two rules for every i, j):

1. $X_i \rightarrow \epsilon \mid X_i \rightarrow keyword_i \mid X_i \rightarrow regex_i$
2. $Y_i \rightarrow \epsilon \mid Y_i \rightarrow keyword_i \mid Y_i \rightarrow regex_i$
3. $P_j \rightarrow \epsilon \mid P_j \rightarrow keyword_j \mid P_j \rightarrow regex_j$
4. $Q_j \rightarrow \epsilon \mid Q_j \rightarrow keyword_j \mid Q_j \rightarrow regex_j$

Where we respectively call $X_i$ and $Y_i$ *non priority context opening and context closing rules* and $P_j$ and $Q_j$ *priority context opening and context closing rules*. We use the terminals `keyword` and `regex` to denote specific keywords and RegExes, and `text` to represent any text. $\epsilon$ is the empty string.

This grammar defines rules to open and close what we call *contexts* for simplicity (in a non technical way, this is still a context-free grammar).

To better explain, let's consider a concrete example given in Table 3.1.

| i | X | Y |
|---|---|---|
| 1 | fun | end |
| 2 | [A-Z][A-Za-z0-9]* | $\epsilon$ |

| j | P | Q |
|---|---|---|
| 1 | % | \n |
| 2 | /* | */ |

Table 3.1: Example production rules for OzDoc's grammar

One can easily recognize Oz's syntax.

The first line for X, Y states that the keyword `fun` must eventually be closed by a keyword `end`. The second line introduces variables, which are not open context but simple leaves, as they immediately end with the empty string.

The first line for P, Q states that comments can be introduced using the percent sign, and only ends at the end of the line. The second line adds block comments. The production rules prevent any other nested context once a priority opening rule

has been encountered, as long as its corresponding priority closing rule has not been met.

This grammar definition allows a highly modulable and somewhat easy parsing of the code following a standard pattern of recursive descent. We do both the tokenization and the parsing into an AST of desired complexity on the go while recovering as much information as we want along the way, such as starting and ending characters' index, starting and ending lines' index, some chunks of code, descriptions, etc.

A default input configuration file containing the settings needed to properly parse Oz for documentation generation could then be as short and as simple as follows:

```
priority_context_rules = [
                  ['%', '\n'],
                  ["/*", "*/"],
                  ["\'", "\'"],
                  ['\"', '\"'],
                  ["`", "`"]]
context_rules = [
                  ["{", "}"],
                  ["[]", ""]]
                  ["functor", "end"],
                  ["fun", "end"],
                  ["local", "end"],
                  ["case",  "end"],
                  ["try",  "end"],
                  ["if",  "end"],
                  ["for",  "end"],
                  ["raise",  "end"],
                  ["thread",  "end"],
                  ["lock",  "end"],
                  ["proc",  "end"],
                  ["meth",  "end"],
                  ["class", "end"]
```

Note that the version shown is simplified. The more complete version as well as more information on the configuration file can be found in the Developer manual in Appendix B.

## 2.2 Coding Strategy

The whole parser follows the Strategy design pattern : it defines its global behavior and replaces its inner methods dynamically using the given arguments.

As an example, we define a dictionary of different methods in the module `ParseEval` to check the context rules. During the parsing, the parser asks ParseEval for an evaluation, and the right method to evaluate this rule is automatically chosen.

The coding took longer but this was necessary to achieve our goal of having a tool easy to modify and implement onto. We followed as much Software Engineering precepts as we could to make the code debt as low as possible.

We also implemented different pseudo-parsing methods of the code that are to be used after the main parse to clean up the result if needed. Those methods include fusing succeeding comment lines into the same node, building a list of all method declarations, finding all comment tags and linking them to comment blocks,... All this always being, of course, entirely modulable. There is not a single string or RegEx in the entire Parser class except the aforementioned default Oz grammar rules, to allow better robustness.

The Parser class, `OzDocParser.py`, as well as the code is available on GitHub (link in Appendix A). More information on the ParseEval module is available in the Developer Manual in Appendix B.

## 2.3 The Abstract Syntax Tree

### Internal Representation of a Node

We represent Nodes as `ParserNode`, a standard Node structure with some added fields:

```
class ParserNode:
    def __init__(...)
        self.node_id = ...
        self.context_type = ...
        self.parent = ...
        self.children = ...
        self.start =...
        self.end = ...
        self.line_start = ...
        self.line_end = ...
        self.description = ...
        self.code = ...
```

Note that most of those fields are kept empty throughout the parsing, and are only computed to improve the DocGen's process. As an example, only the root node usually has a defined `code` field. All other children nodes use indexing to minimize

spatial complexity, and `start` and `end` are also used to reduce time complexity in other search algorithms. `Node_id` is a unique ID generated by the parser (for HTML id purposes); Context type is the name of the rule or the keyword used to create these node; Line indexes are compiled once during the parsing for the user's readability and to be used by the DocGen.

The class, of course, comes with some very useful methods. Two worth mentioning would be `iter_children` which returns an iterator for pythonic uses, or the `lowest_context_for_char` method which returns the lowest child node having its context active at a particular character, doing so very quickly by abusing the `start` and `end` fields.

### The Resulting Graph

On figure 3.3, we provide an example result of the AST resulting from the basic parse. The print function outputting this tree structure has been entirely coded by us, as the *Python tree pretty-print* package we used to work with didn't print the nodes in order of appearance, but rather reorganized them to help visualize the tree (which was unwanted in our case). Each line is a ParserNode, printed as follows:

```
context type (description) @L[line start:line end] @[char start:char end]
```

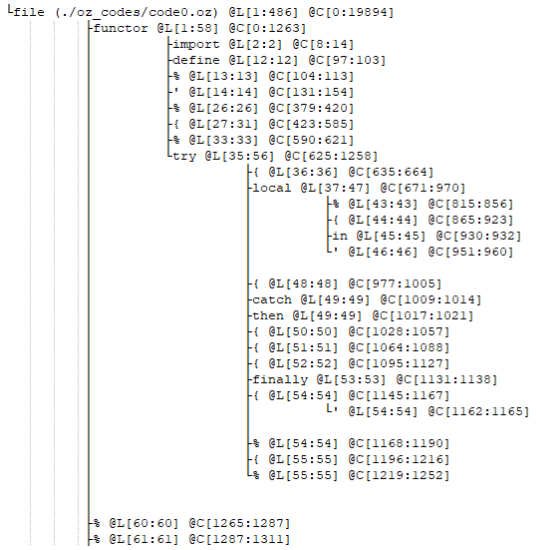


Figure 3.3: An abstract Syntax Tree root

### Improving the graph's completeness

Let's have a look at a specific block of code visible on figure 3.4. Its corresponding

```
438   %---------------------------CheckTranslate-PU-----------------------------------
439   % This function checks if the translate given in argument is a translate allowed here
440       fun {CheckTranslatePu translate(dx:X dy:Y 1:PokeUniverse)}
441          if {CheckIfFormula X} andthen {CheckIfFormula Y} then
442             {CheckPokeUniverse PokeUniverse}
443          else false
444          end % end of if {CheckIfFormula}
445       end % end of fun {CheckTranslatePu translate(dx:X dy:Y 1:PokeUniverse)}
446
```

Figure 3.4: A classic Oz function

```
-% @L[338:338] @C[13002:13032]
-% @L[340:340] @C[13033:13114]
-% @L[341:341] @C[13114:13196]
-% @L[342:342] @C[13196:13255]
-fun @L[343:348] @C[13257:13469]
          |-( @L[343:343] @C[13261:13315]
          |-if @L[344:347] @C[13321:13436]
          |          |-( @L[344:344] @C[13324:13340]
          |          |-( @L[344:344] @C[13349:13365]
          |          |-then @L[344:344] @C[13366:13370]
          |          |-( @L[345:345] @C[13379:13411]
          |          |-else @L[346:346] @C[13417:13421]
          |
          |-% @L[347:347] @C[13437:13464]
```

```
-% @L[338:338] @C[13002:13032]
-% @L[340:340] @C[13033:13114]
-% @L[341:341] @C[13114:13196]
-% @L[342:342] @C[13196:13255]
-fun @L[343:348] @C[13257:13469]
          |-( @L[343:343] @C[13261:13315]
          |          |-Variable (CheckTranslateRu) @L[343:343] @C[13262:13278]
          |          |-atom (translate() @L[343:343] @C[13279:13314]
          |          |          |-Variable (X) @L[343:343] @C[13292:13293]
          |          |          |-Variable (Y) @L[343:343] @C[13297:13298]
          |          |          |-Variable (RealUniverse) @L[343:343] @C[13301:13313]
          |
          |-if @L[344:347] @C[13321:13436]
          |          |-( @L[344:344] @C[13324:13340]
          |          |          |-Variable (CheckIfValue) @L[344:344] @C[13325:13337]
          |          |          |-Variable (X) @L[344:344] @C[13338:13339]
          |          |
          |          |-( @L[344:344] @C[13349:13365]
          |          |          |-Variable (CheckIfValue) @L[344:344] @C[13350:13362]
          |          |          |-Variable (Y) @L[344:344] @C[13363:13364]
          |          |
          |          |-then @L[344:344] @C[13366:13370]
          |          |-( @L[345:345] @C[13379:13411]
          |          |          |-Variable (CheckRealUniverse) @L[345:345] @C[13380:13397]
          |          |          |-Variable (RealUniverse) @L[345:345] @C[13398:13410]
          |          |
          |          |-else @L[346:346] @C[13417:13421]
          |
          |-% @L[347:347] @C[13437:13464]
```

Figure 3.5: A classic Oz function sub tree, with atoms and variables added.

sub tree can be seen in figure 3.5 on the left. By modifying the `settings` file to also look for atoms and variables (that is, simply adding two rules), we can easily obtain the sub tree on figure 3.5 on the right. If the DocGen really needed everything, we could add the keyword *andthen* and add a last regex to recover the fields inside the atom. Finally, after running a method to also group comments (`context_type` given as argument), we obtain the last graph, visible at 3.6. We did all those modifications in a matter of seconds without any changes to the code itself, proving the Parser's great modulability.

### Simplifying the Graph

There is a legal minimum to what the `settings` file needs to properly function. While we can go as far as we want towards a more complete AST by adding many correct rules, we can **not** go the other way around and simplify the graph as much as we want by removing more rules. This happens because, in Oz, there are many different nestable contexts that end with the same keyword, (`end`). So the code:

```
fun {Z} if X then Y else Z end end
```

The fun will only be correctly parsed if both the `fun` and `if` rules are present. However, it is very easy to ask the parser to delete unwanted nodes once the parsing

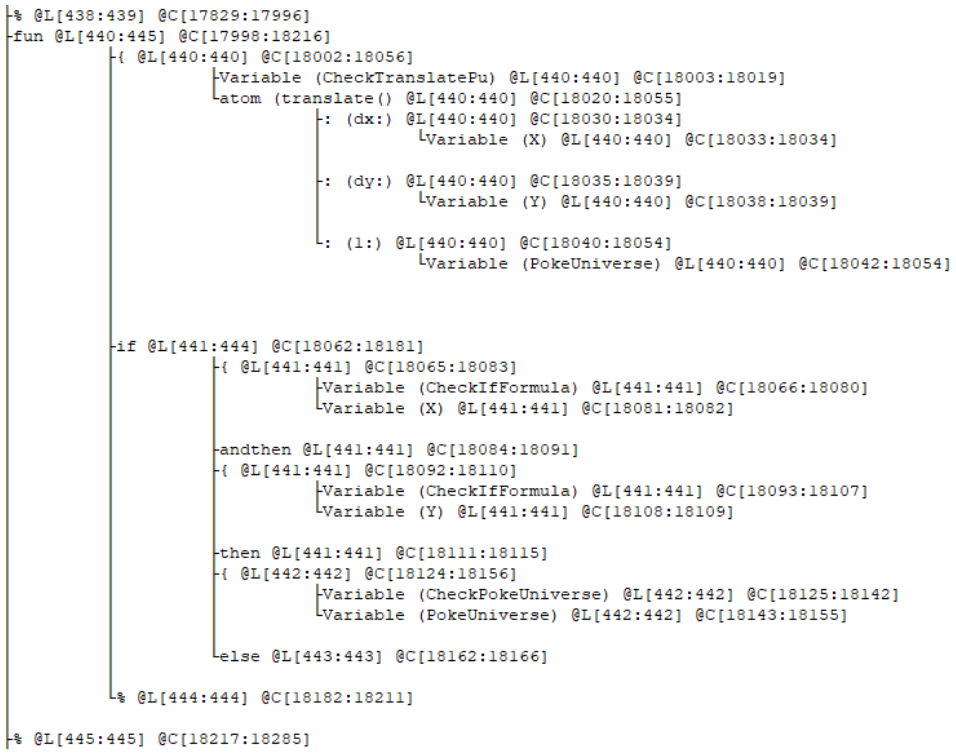


Figure 3.6: A classic Oz function sub tree, completely parsed.

process is over. In fact, it is done with a single one-liner. By only keeping the bare minimum, we can obtain the graph on 3.7.

### Folders and Files as nodes.

As you can also see on 3.7, the Parser sees and handles files and folders as nodes in the AST, making the parser jump between directories just as it does between lines of codes. This simplifies the DocGen's work, as it has only one tree to iterate over, giving it the possibility of cross-referencing elements between files.

### Node Repositories and AST clean up.

As briefly mentioned above and in section 2.2, there are many pre-coded modifications and clean ups that can be done to the AST once it has been generated.

Let us define the concept of a Node Repository (repo), that we use throughout the documentation generation and that are generated right after or during parsing. A repo is a data entity (typically a list of `lists` or a `dict`) holding information between the nodes' contexts and other file contents, much like a Structured Query Language (SQL) table would do.

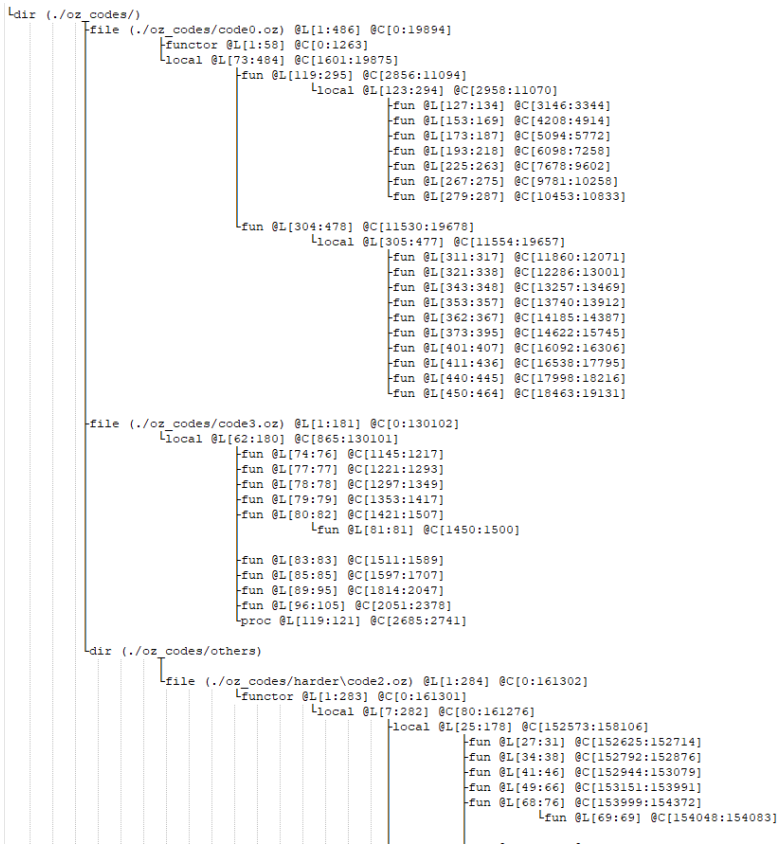


Figure 3.7: A reduced AST showing functions, files and directories

These tables are produced dynamically and are used to clean the AST and facilitate the DocGen's work. Here is a concrete example workflow on how to build a repo of functions and calls for Oz:

1. We have a function that builds a repo of nodes of a particular context type. We feed it the 3 function keywords in Oz (`proc`, `fun`, `meth`). It will thus contain a list of nodes having these contexts.

2. We have a function that can append a new column to a repo with, for each node in the first column, the following node of a certain context type in this new column. We actually run this function on our repo of functions to link the functions to their declaration (in braces {), allowing us to have them accessible when making their documentation.

3. Finally, we will use the repo-building function again to build another repo of nodes, this time for function calls only. We will thus look for the  (braces) context type, which also represents function calls in Oz. However, we only want real method calls to be added to the repo, and not method declarations (which also use braces). For that, we simply add an exception to the arguments

of the repo-building function, made with the function declarations recovered in the previous step.

4. We now have a repo of functions with their respective declarations and a repo of function calls, in 3 lines of code. Those can then be used in the DocGen process.

# 3 The documentation generation

Once the parsing and post-parsing work was done, it was time to finally generate documentation. There were many different options, but our research helped us head in the right direction.

## 3.1 Defining the documentation file format

It was clear to us that HTML-, CSS- and JavaScript (JS)-based documentation was the way to go. It has the same advantages as PDF, namely in that it is portable, readable on every media, well formatted and can be printed, but doesn't have all the verbosity of text and Excel files. It also has the additional advantage of being **easily modifiable** and with a **high level of interaction**, like an application. Contrary to PDF, one could also open an HTML document and copy or modify the layout however he pleases.

Modulability and accessibility being once again our main goals, we decided that the documentation should be *template* based, kind of mimicing recent Javadoc's stylesheet support [18;19] as well as Doxygen's [24]. The base template is accessible and editable, and the DocGen copies it before writing HTML into it. We implemented program arguments to manually chose the template folder and the DocGen code to be executed. We will talk about that in the following section, i.e. section 4.

The next step was then to define what the template needed. First, of course, it needed to be totally *serverless* and usable offline. That means that any content must be computed and generated in advance into the documentation, and that the only interactions will have to come from accessible JS. Likewise, all the CSS and resources should be available offline. We needed to do all that without any repercussions on the rich beauty or interactivity of the documentation.

To solve both issues, the obvious choice was to go for what is called a front-end

**framework**. Among the different options, we decided to go for the very popular, well-documented, free and open-source **Bootstrap**[1]. Made for developing responsive and mobile websites, Bootstrap is the most popular CSS Framework and comes with many tutorials[40].

## 3.2 Using the right HTML library : Yattag

Writing HTML directly from a Python script would be a pretty daunting task with many ugly hard-coded parts. As we were also aiming for ease of access, we decided to use a python library that would deal with writing HTML for us. Yattag[2], a free of use Python library, was a perfect fit. We show an example use of Yattag in the following code:

```
doc, tag, text, line = Doc().ttl()
with tag('ul', id='grocery-list'):
    line('li', 'Tomato sauce', klass="priority")
    line('li', 'Salt')
    line('li', 'Pepper')
print(indent(doc.getvalue()))
```

Note the use of the `klass` keyword because `class` is a reserved Python keyword. This code outputs the following HTML code:

```
<ul id="grocery-list">
  <li class="priority">Tomato sauce</li>
  <li>Salt</li>
  <li>Pepper</li>
</ul>
```

Note that we imported the library into our project so there isn't any extra installation to be done by the end user or end developer.

## 3.3 Code writing methodology

More than our specific code, our aim here is to present how any DocGen would be implemented in our framework. We consider that the parsing of the code was already done and the AST already produced, and that all that is left is the documentation

[1] https://getbootstrap.com/
[2] https://www.yattag.org/

to generate.

The first step in the documentation generation is to write the template. One should have a concrete idea of what he wants to add in the documentation and where. The main `index.html` as well as JS functions and CSS classes should be completely written before going any further. Spots of codes to be replaced in the Template by the DocGen can be specified using keywords, or the template can use *iframes*[3] of HTML files that will be created by the DocGen at a later stage. We provide one basic template for Oz documentation, but it could of course be used for any other programming language.

The second step is to write the DocGen Python script and add it to the OzDoc framework. The DocGen is simply a Python file containing code to be executed after the parsing. If written correctly (by following the Dev Manual B), our framework gives it access to the AST resulting of the parsing, the destination folder for the documentation with the template already copied in it, the configuration files and finally, of course, all other utility modules we have added to the project, including Yattag and other tools to handle files and build the aforementioned repos. The DocGen then simply computes what it needs and adds HTML to the copied template files and folders. We already gave examples on how building repo and using Yattag make that step trivial. One shall also keep in mind that his DocGen will be able to handle a single file just as much as a folder or text alone, and will need to take that into account.

To generate the documentation, all that is left is to run the OzDoc script with, as arguments, the template and DocGen to be used as well as the input code and output directory. We talk more about the OzDoc script in the following section 4, and on how to write DocGens in the Dev Manual in B.

## 3.4 Resulting Oz DocGen and Oz Documentation

Based on our ambition of combining the best parts of every good documentation generator available, we wanted to fully commit towards making a documentation generator using the entirety of Javadoc tags[44] combined with extra Doxygen tags and features (class diagrams, summaries, etc)[24], as well as features from the likes of Pydoc, Sphinx, Epydoc, etc.

[3]https://www.w3schools.com/tags/tag_iframe.asp

### Template-based generation

As presented before, we decided to go for a template based solution. The only thing OzDoc needs to generate a default documentation right now is a template HMTL. The definition of a "template" is quite open: it is basically any HTML containing the replacement tags we fixed. Said tags will be replaced to generate what their name describes (a table head or body, a list of function details, etc.). An example of a minimalist template is provided in figure 3.10. Of course, one can also choose to follow a different direction, and replace the current documentation generation algorithm entirely in a matter of hours.

This makes the tool incredibly easy to pick up: one can even get his favorite HTML template from anywhere, and simply stuff tags like functiondetails in an empty space or tablebody in the middle of a ¡table¿ tag, and the parser and template injector will take care of the rest. The default template is only a first design aiming to help students and researchers alike have a better grasp of the functions in their code. This should give programmers an incentive write more comments, and will only lead to better documentations as users get used to it.

### Tags

The tags currently taken into account by OzDoc's default documentation generator are a mix of Java and C, with tags like params, see, result being mixed with pre and post. Tags are also modular, and can be modified in a matter of seconds to be instantly recognized. They just need to be taken care of by the file generators to appear in the output file. The current template-based algorithm is only a first among an almost infinite amount of variety

### Some Observation on the Oz codes

The main enemy of all of our ambitions for the generated output was the heterogeneity and lack of rigor of Oz users themselves. Indeed, after the backend coding part for the parser, came the design part for the final documentation we would like our users to have with the default stylesheet, the one they will get if they launch OzDoc with a minimal amount of reconfiguration.

Having drafted several unique models of possible final pages, we decided to study the huge amount of data we had to see which one we would choose and how we

would implement it. That is when we noticed that most of what we thought was either impossible or fairly useless considering the way people use Oz.

Not only did we try to visualize documentation on the huge amount of student projects we had from the previous years, but we were also focused on making a full API of the mozart2 Github repository. However, that would mean that we would need a complete overhaul of the repository itself.

While in Java, the user is forced to have functions inside of classes, is encouraged to create a hierarchy between the classes and packages and play with inheritance, etc., in Oz, the programmers can do just about whatever they want. And so they do.

## 3.5 Lack of enthusiasm for comments

We noticed that most people just like to stack everything into lots and lots of functions even in the official Mozart2 code from the repository. Those functions rarely have comments attached to them, or at least useful ones. Most people just live a one-liner briefly explaining the aim of the function, but it is mostly crystal clear for the author rather than other potential readers.

The comments in the files of the mozart2 repository being way too sparse, we had to give up on the idea of creating an API simply with our tool. Help would be needed from the original writers of the programs, as, at the very least, short explanation comments are needed to translate it into tags and turn it into a documentation page.

This is when we turned our focus towards Python DocGens a bit more: while its users use classes a lot more, it is still a script language in which the programmer is also free to do just about whatever he wants. The inspiration for OzDoc grew gradually into being Sphinx, Pydoctor, etc. more than Javadoc.

For the main focus, we opted for what would, as we empirically concluded, be the most beneficial for the current users at first: prioritizing grouping and explaining of lone functions, as they make up the majority of most people's code. Following this, we ended up with the main center of focus of the page being similar to what you can see on figure 3.8: instead of making the classes the center of attention and the first thing to enter the viewer's field of view, it would be a table regrouping the functions. Details of said functions are then available below the table in a style very similar to Javadoc, as visible on figure 3.9.

An important step that will be needed to be able to do better designs is to change

| Function | Description |
|---|---|
| MyFunction | Begin of MyFunction |
| Flatten | This function takes a list with sub-lists in it and returns a list without lis |
| CheckValue | |
| GiveMeThePointWithModifications | This function returns a point with the modifications include in the List. |
| DoRu | we put ListOfMod at nil |
| CheckFormula | "If-then-else" and "comparaison". |
| DoTranslatePu | This function applies the translation (Dx Dy) to the elements of List. |
| DoPu | This function checks the pu part of Map and List is the list in pu. |
| CheckMap | All the following functions returns true or false |
| CheckPrimitiveRu | here (kind: building or road or water). |
| CheckIfValue | This function checks that the value X is a value allowed here |
| CheckTranslateRu | here (translate(dx:<value> dy:<value> 1:<RealUniverse>)) |
| CheckRotate | here (rotate(angle:X 1:<RealUniverse>)) |

Figure 3.8: OzDoc applied to a student's 2016 Oz project: function table

## CheckValue

*{CheckValue X }*

Validates a chess move.

<p>Use {@link #doMove(int theFromFile, int theFromRank, int theToFile, int theToRank)} to move a piece.

**Parameters:**

-- theFromFile file from which a piece is being moved

-- theFromRank rank from which a piece is being moved

-- theToFile file to which a piece is being moved

-- theToRank rank to which a piece is being moved

**Returns:**

-- true if the move is valid, otherwise false

**Since:**

-- 1.0

**Source Code:**

Show

```
fun {CheckValue X }
if {Float.is X} then X
else
   case X of exp( W) then {Float.exp {CheckValue W} }
   [] log(W) then {Float.log {CheckValue W}}
   [] neg( W) then ~{CheckValue W }
   [] plus( W Z)  then {CheckValue W} + {CheckValue Z}
   [] minus( W Z)  then {CheckValue W}- {CheckValue Z}
   [] mult(W Z) then {CheckValue W}*{CheckValue Z}
   [] 'div'(W Z) then {CheckValue W}/{CheckValue Z}
   [] sin(W) then {Float.sin {CheckValue W}}
   [] cos(W) then {Float.cos {CheckValue W}}
   [] tan(W) then {Float.tan {CheckValue W}}
   [] exp(W) then {Float.exp {CheckValue W}}
   end % end of case X
end % end of if {Float.is}
end
```

Figure 3.9: OzDoc applied to a student's 2016 Oz project: function details

```
    <section id="tables" class="my-3">
        <div class="container">
            <table class="table table-striped">
                @tablehead
                @tablebody
            </table>
        </div>
    </section>

    <section id="function_details">
        @functiondetails
    </section>

    <script src="assets/js/jquery.min.js"></script>
    <script src="assets/js/bootstrap.bundle.min.js"></script>
    <script src="assets/js/master.js"></script>
</body>
</html>
```

Figure 3.10: Example of HTML template usable by OzDoc

the general mindset, and forcing people into the conventions that our tags make up. While this limits freedom, it would also greatly increase visibility and would increase the scarce amount of comments in each code..

# 4 The OzDoc Framework

Historically, we first implemented the parser and then the DocGen, and used to call the parser from the DocGen. We needed a definitive main script and were hesitating about which one of the two it should be. In the end, following our aim of higher customizability and modulability, we decided to implement a whole framework for generating documentation through a new script that would encapsulate both parts of the documentation generation process : The `OzDoc.py` main module.

## 4.1 How the encapsulator works

The main script follows, once more, the Strategy design pattern.

The first step is to recover input arguments. As expected from such a project, the main module handles extensive arguments and prints their usage (using the `-h` argument). The arguments given to the program must include either a file, folder or text (mutually exclusive arguments) from which the documentation should be generated. It accepts as optional arguments :

- a configuration Python script containing the grammar and other settings that are primarily used by the parser, but might also be also accessed in the DocGen,

- a DocGen Python script containing the code to be ran,
- an input template directory,
- an output directory in which the generated documentation should be put.

Note that the script does not check for file presence and access for robustness, as The Python community uses an EAFP (easier to ask for forgiveness than permission) coding style.[41] It handles the configuration file by dynamically importing it as a module. It uses the retrieved settings to run the parser on the input code (be it a file, folder, or plain text).

It then copies the entirety of the template to the output destination for the DocGen to use.

The main module dynamically imports and runs the DocGen (be it the one by default or the one passed as argument). The DocGen receives access to the parser AST, the settings, the template source folder and the destination folder.

Finally, the main modules asks the `os` to open the documentation main entry page, that is the `index.html`.

By following that process, exactly implementing the dynamic Strategy design pattern, we have in fact developed a **Framework** for users and developers to create their own documentation and to implement their own DocGens, templates and grammars.

## 4.2 Integration to Emacs

The integration to Emacs is fairly simple, and can be done in five steps.

First, a new entry for Emacs' Oz context menu must be added to launch OzDoc. This menu is the variable `oz-menu` in `oz.el` at `mozart2-master/opi/emacs`. We can call this new entry "Generate Doc", and link a function `oz-ozdoc-buffer t` to it that will be implemented in `mozart.el`, same folder.

In `mozart.el`, one needs to add the fun `oz-ozdoc-buffer` that simply calls the external program OzDoc by any means (like the lisp library "UIOP, the Utilities for Implementation- and OS- Portability" function `uiop:run-program`) with arguments "-t" for text, followed by the text of the current buffer.

A complete build of the project is not necessary. Compiling the two modified files and replacing the ones in Emacs's folder `share/mozart/elisp` is enough to have

the integration.

We need OzDoc to be accessible by the os. For that, we can put it mozart2's folder hierarchy, and hardcode its location in the aforementioned `oz-ozdoc-buffer` function.

Finally, the os needs to be able to run OzDoc (have Python3 be installed).

We followed these steps to integrate a previous prototype version of OzDoc to Emacs with success.

# Chapter 4

# Evaluation

Evaluation is a key element of any project. It aims at assessing the quality of the deliverable according to different decision criteria.

Quantifiable criterion like process time or number of errors are very simple to both evaluate and judge. Qualitative criterion like quality, usability or robustness are what makes evaluation exponentially more complex the more features are added to a tool.[42]

In our case, for a project that creates a tool for end users, the most dominant criteria are the **usefulness** and **accessibility** of the tool, that are both not only qualitative but also very subjective. Because of that subjectivity, we needed to ask users, obviously other than us, what were their thoughts on our tool. Indeed, the most important objective of OzDoc is that **users** do actually use it, and evaluating the propensity of users to try out OzDoc by themselves and keep using is the center of our evaluation.

## 1 Sampling

While the evaluation of execution time can objectively be done in a few lines of code, evaluation of the accessibility and practicality is much more subjective and requires a form of feedback. As the initial objective of having students test it in early 2019 during their Oz project at UCLouvain couldn't be met, we decided to fall back and survey a population of interest however we could, despite the smaller sample size. This lead us to showing the tool to computer science and engineering students present in the several computer rooms accessible at UCLouvain for precious

feedback. While the audience is the right one, it is to be noted that the sample size is quite small.

We hope more feedback will be obtainable the next academic year, especially if the tool is added to the Oz course.

# 2 Evaluation of the Parser

While the Parser is the main culprit in requiring our attention and workforce due to the numerous changes and overhauls it had to go through, eating up time that could've been used to have a better documentation generator template, the final product is a powerful tool we honestly couldn't be more proud of.

## 2.1 Execution Time

Execution time is directly linked to the complexity of the grammar passed as argument and to the length of the code to parse, so it is hard to assess its speed. However, testing proved that in normal use cases, time shouldn't even be considered a problem. We are talking in a matter of tenths of seconds to extensively parse the biggest Oz code we had, and a maximum of 15 seconds to parse a tree of folders for a total of over 5000 lines of codes.

# 3 Evaluation of the whole OzDoc Framework

## 3.1 Ease of installation

Installing the tool has proven to be very easy. A simple download, unzip, and finally a drag-and-drop and the script was ready to be used. Our subjects all had Python already installed, so it simplified the installation. We can expect most of our future users to also have Python installed.

## 3.2 Ease of use

Running the tool was also very simple. The help message shown when running the script is very comprehensible and the code has been written specifically so that it can

be run from anywhere, take input from anywhere and output anywhere. We asked subjects to swap between different input parameters (including different settings, docgen, and layouts) we didn't see a rise in use complexity.

## 3.3 Ease of development

We asked two users at two different key point of our thesis to develop their own use-case (including grammar in setting file, a very easy template to make sentences, and a very easy docgen that just computes the data to be printed).

The first test was done without the current Developer Manual, only oral explanations about how the tool works. This is because we used to **not** have a Developer Manual, because we had not yet thought of it. In the first feedback we received, the user complained that he did not know where to start or what to do, and felt crushed under the hierarchy of the program. He still managed to make his use case work by copying our code and modifying it, but it wasn't conclusive.

The second test was done wit the Developper Manual and without any oral explanation. The first feedback we received was that the manual was too long, lacking in examples, and going all over the place. After some reading, it was however concluding : the user took entire control of the Framework, and even had his fun building trees of HTML.

The current version of the Developer Manual is the improved version after the feedback. We shortened it, made it follow a more concise straightforward top-down explanation and we added example uses at every point.

# 4 Evaluation of the Oz DocGen and produced documentation

## 4.1 Quality

Our focus being having been mainly on the parser and framework for a major part of this project, the default documentation generating part of OzDoc probably ended up being the weakest link. The template-based generator is a very powerful tool in itself, but the default design for the output page that we wanted to use to invite new

users to try OzDoc could use a better design. This only includes the default style sheet, however, as the tool in itself integrates smoothly with the framework and works quickly and effectively like the rest of the system. A small design overhaul of the default appearance of the output could indeed have its place.

## 4.2 Usefulness

Despite the potential need of raising the attractiveness of the visuals, the tool can still already be used optimally with the style sheet. New design sheets and new documentation generation variants can be seamlessly integrated in a matter of minutes. The documentation generator can thus fit the desires and needs of all kinds of people, as they're free to play with the framework to obtain anything they might want.

# Chapter 5

# Conclusion

### Why OzDoc

The idea of OzDoc was born to help students at UCLouvain have an easier time understanding Oz code written by others so they could spend more time making better projects. We also wanted the Programming Languages and Distributed Computing (PLDC) Research Group to have an easier time working with Oz and be able to document their research. We believe the tool we have produced has achieved all initial goals.

### The Research

The research aimed to better understand documentation generator as well as identify good practices and features. Through extensive investigation, we could recognize the following attributes to be quintessential: simplicity (ease of access, ease of use), usefulness, expendability, boilerplate avoidance, visual attractiveness and having unique features.

### Implementation

The development and implementation process proved itself to be on an even bigger scale than we expected. The creation of an optimized parser, that ended up being adaptable as well, along with the making of a documentation generator capable of matching or even surpassing modern documentation generators ended up being a project full of ups and downs, and mostly full of sharp turns in the development.

Catching up to documentation generators that have more than 20 years of development history in a just a few months proved to be a challenging but exciting task that thought us how to handle bigger projects.

### Result

The end result of these months of research and development is OzDoc, a flexible and adaptable tool/framework for documentation generation. The tool works for Oz but also any other language. It is highly modulable, allowing developers to add their own DocGens, templates and grammars in seconds. The tool comes with a User Manual and a Developer Manual.

### Evaluation/Improvements

We consider the parser to need little to no improvement, and are really proud of it. Evaluation did show, however, that our default documentation might need a little bit of tweaking to its design to become a more pleasant visual experience. Another good road for improvement would be to add more default grammar for other languages as well as more, better looking templates and associated DocGens.

### To the readers

We thoroughly encourage each and every reader to try out OzDoc with the help of the user manual, and hope we might convince developers to read the developer manual and join us as well either to improve the tool or make their own documentation generators. We hope the community will find our tool helpful and enjoyable.

# Bibliography


[1] Peter Van Roy. Programming Languages and Distributed Computing (PLDC) Research Group. https://www.info.ucl.ac.be/~pvr/distribution.html, 2016.

[2] J. Carroll. The history of technical documentation. https://medium.com/@Jacklyn_Lee/the-history-of-technical-documentation-ea7d42921933, 2019. [Online; last accessed 17-August-2019].

[3] J.-P. Louyot, C. Jimenez, Wikibooks Contributors. Mkd (Extracteur de documents). https://upload.wikimedia.org/wikipedia/commons/4/4d/Mkd_%28Extracteur_de_documents%29-fr.pdf, 2019. [Online; last accessed 17-August-2019].

[4] S. Heckman L. Williams, D. Ho. Javadoc. https://web.archive.org/web/20170613233020/http://agile.csc.ncsu.edu/SEMaterials/tutorials/javadoc/, 2004. [Online; archive from 13-June-2017].

[5] Ka-Ping Yee. Module pydoc. http://epydoc.sourceforge.net/stdlib/pydoc-module.html, 2001. [Online; last accessed 17-August-2019].

[6] T. Christie. Mkdocs - project documentation with markdown. https://www.mkdocs.org/, 2014. [Online; last accessed 17-August-2019].

[7] F. Gessert. Building static sites in 2017: Cloud-hosted, cms-backed, and api-driven. https://medium.baqend.com/building-static-sites-in-2017-cloud-hosted-cms-backed-and-api-driven-f68b5debc396, 2017. [Online; last accessed 17-August-2019].

[8] H. R. Donfack. Comprendre la spécification openapi (swagger) et apprendre à utiliser swagger editor. https://www.developpez.com/actu/178434/Comprendre-la-specification-OpenAPI-Swagger-et-apprendre-a-utiliser-Swagger-Editor-par-Hi

2017. [Online; last accessed 17-August-2019].

[9] L. Pollock E. Hill and K. Vijay-Shanker. Automatically Capturing Source Code Contextof NL-Queries for Software Maintenance and Reuse. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.215.4662&rep=rep1&type=pdf, 2009.

[10] D. Muppaneni L. Pollock G. Sridhara, E. Hill and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods . https://sites.udel.edu/se-research/2010/09/05/towards-automatically-generating-comments-for-java-methods/, 2010.

[11] P. W. McBurney and C. McMillan. Automatic Documentation Generation via Source Code Summarization of Method Content. https://www3.nd.edu/~cmc/papers/mcburney_icpc_2014.pdf, 2014.

[12] S. Mahakalkar and A. D. Gujar. Source Code Summarization of Context for Java Source Code. http://www.ijircce.com/upload/2016/july/56_Source.pdf, 2016.

[13] Benjamin Poulain. Linux Certif - ozdoc. http://www.linuxcertif.com/man/1/ozdoc/#NAME_15042h, 2008. [Online; last accessed 17-August-2019].

[14] Mozart Contributors. Mozart, an implementation of oz. https://github.com/mozart/mozart2. [Online; last accessed 17-August-2019].

[15] DocGen Team. Third International Workshop on Dynamic Software Documentation (DySDoc3). https://dysdoc.github.io/dysdoc3/, 2018. [Online; last accessed 17-August-2019].

[16] JavaScript Joel. Let's talk about the state of auto-generated documentation tools for javascript. https://dev.to/joelnet/lets-talk-about-auto-generated-documentation-tools-for-javascript-49ol, 2018. [Online; last accessed 17-August-2019].

[17] Douglas Kramer (Sun Microsystems). API Documentation from Source Code Comments: A Case Study of Javadoc, 1999. [Paper available at http://www.dougkramer.org/documents].

[18] Oracle Corporation. Java SE 7 Features and Enhancements. https://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html,

2011. [Online; last accessed 17-August-2019].

[19] Oracle Corporation. JDK 10 Release Notes. https://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html, 2018. [Online; last accessed 17-August-2019].

[20] Oracle Corporation. What's New in JDK 8. https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html, 2014. [Online; last accessed 17-August-2019].

[21] Oracle Corporation. What's New in JDK 9. https://docs.oracle.com/javase/9/whatsnew/toc.htm, 2017. [Online; last accessed 17-August-2019].

[22] Markus Sprunck. Top 5 Reasons for Not Using JavaDoc in the Next Project. https://www.javacodegeeks.com/2012/06/top-5-reasons-for-not-using-javadoc-in.html, 2012. [Online; last accessed 17-August-2019].

[23] Robert S. Laramee. Bob's Concise Introduction to Doxygen. https://cs.swan.ac.uk/~csbob/teaching/laramee07commentConvention.pdf, 2011.

[24] Dimitri van Heesch. Doxygen - Manual for version 1.5.3. http://www.literateprogramming.com/doxygen.pdf, 2007.

[25] Google. Facts about Google and Competition. https://web.archive.org/web/20111104131332/https://www.google.com/competition/howgooglesearchworks.html, 2011. [Online Archive; last accessed 14-August-2019].

[26] A. V. Aho et al. Compilers: Principles, Techniques, and Tools, 2006.

[27] G. Tomassetti. The ANTLR Mega Tutorial . https://tomassetti.me/antlr-mega-tutorial/#mistakes-and-adjustements, 2017. [Online; last accessed 17-August-2019].

[28] P. Van Roy S. Haridi. Concepts, techniques, and models of computer programming, 2004.

[29] T. J. Parr. ANTLR Reference Manual. https://www.antlr3.org/share/1084743321127/ANTLR_Reference_Manual.pdf, 2004.

[30] UCL P. Schaus. Languages and translators.
https://uclouvain.be/cours-2019-lingi2132, 2017.

[31] T. J. Parr. The Definitive Antlr4 Reference.
http://lms.ui.ac.ir/public/group/90/59/01/15738_ce57.pdf, 2007.

[32] R. W. Quong T. J. Parr. ANTLR: A Predicated-LL(k)Parser Generator.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.881&rep=
rep1&type=pdf, 1994.

[33] L. Moonen. Generating robust parsers using island grammars.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=957806, 2001.

[34] H. Boom A. Van Deursen, J. Visser. Island Grammars.
http://www.program-transformation.org/Transform/IslandGrammars.

[35] T. Kuipers A. Van Deursen. Building Documentation Generators.
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=792497, 1999.

[36] P. Dupont (UCL) F. Cédrick. Computational Linguistics.
https://uclouvain.be/en-cours-2019-LINGI2263.html, 2017.

[37] A. Paquot N. Magrofuoco. CorrectOz – Recognizing common mistakes in
theprogramming exercises of a computer science MOOC.
https://www.info.ucl.ac.be/~pvr/Magrofuoco_Nathan_33521100_&
_Paquot_Arthur_60371000_2016.pdf, 2016.

[38] S. Doeraene. Reply to 'Memory usage and execution time (compared to 1.4)'.
https://github.com/mozart/mozart2/issues/118#issuecomment-29453327,
2014. [Online; last accessed 17-August-2019].

[39] D. E. Knuth. Computer Programming as an Art.
http://delivery.acm.org/10.1145/370000/361612/a1974-knuth.pdf, 1974.

[40] Jake Spurlock. *Bootstrap: responsive web development.* " O'Reilly Media,
Inc.", 2013.

[41] QantifiedCode. Python Anti-Patterns.
https://docs.quantifiedcode.com/python-anti-patterns/readability/
asking_for_permission_instead_of_forgiveness_when_working_with_files.html,
2018.

[42] Andrew Vickers Antony Powell. A practical strategy for the evaluation of

software tools, 1996.

# Figure References

[43] Science & Society Picture Library. Science museum. Retrieved from https://blog.stephenwolfram.com/2015/12/untangling-the-tale-of-ada-lovelace/.

[44] O'Reilly. Javadoc tags. Retrieved from https://www.oreilly.com/library/view/java-a-beginners/9780071606325/appblev1sec1.html. [Contains all the tags from https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html].

[45] Oracle. Screenshot of https://docs.oracle.com/javafx/2/api/, as last seen on 17-August-2019. [From the official API].

# Appendix A

# User Manual

This Manual is for end users who only wish to use the tool to generate documentation. It covers the usage of the default parameters as well as externally provided parameters, but not how to implement them.

## 1 Installing Python

Python Version 3 (or more recent) is needed to run the tool. Please refer to https://www.python.org to install Python on your device. You can check if you have Python installed on your computer and verify the version by running the command:

```
> python --version
```

One thing to note is that if you have both Python 2 and Python 3 installed on your computer, running the above command will only tell you about Python 2. If the above command indicates that you have a version of Python 2, it is advised to also try using the following command:

```
> python3 --version
```

## 2 Downloading OzDoc

The tool can be downloaded for free from GitHub at https://github.com/AlexeSimon/OzDoc

# 3 Installing OzDoc

No installation is required for using OzDoc. Simply unzip the downloaded file (using tools like Winrar) and put it wherever you please.

# 4 Using OzDoc

You can launch the tool like any other Python script by executing the following command:

```
> python OzDoc/OzDoc.py code.oz
```

Where `OzDoc/OzDoc.py` is the path to `OzDoc.py`, and `code.oz` is the code you want to generate documentation from. The tool can be run from anywhere, as long as the path to `OzDoc.py` is correct.

The documentation will be generated in a new folder `./generated_doc`. You can view it by opening the `index.html` file inside it in any browser of your choice.

# 5 Program Arguments

Arguments are of the following form:

```
usage: OzDoc.py [-h] [-f FILE] [-d DIR] [-t TEXT] [-o OUT] [-s SETTINGS]
                [--template TEMPLATE] [--docgen DOCGEN]
                [file]

Generates documentation (for Oz code by default).

positional arguments:
  file                  specify input file

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  specify input file
  -d DIR, --dir DIR     specify input directory
  -t TEXT, --text TEXT  specify input text
  -o OUT, --out OUT     specify output destination (default: generated_doc)
  -s SETTINGS, --settings SETTINGS
                        specify setting file
```

```
                      (default: settings/oz_default.py)
  --template TEMPLATE   specify documentation template directory
                        (default:templates/oz_default)
  --docgen DOCGEN       specify documentation generating code
                        (default:docgens/oz_default.py)
```

The program always requires one and only one of the following arguments:

1. A file path. This is equivalent to the option -f file path
2. -f followed by a file path
3. -d followed by a directory path
4. -t followed by a string delimited by quotation marks ”

The documentation will be generated from the argument that has been passed.

Optional arguments are as follows:

1. -o specifies the output folder. By default, the documentation is generated in a new folder `./generated_doc` in the script path.
2. -s specifies a settings file for the parser and docgen. By default, the `oz_default.py` located in `OzDoc/settings` is used.
3. template specifies the template folder. By default, the documentation uses the template folder `oz_default` located in `OzDoc/templates`.
4. docgen specifies the documentation file generation Python script to run. By default, the file `oz_default.py` located in `OzDoc/docgens` is used.

All paths must always be given relative to current path or absolute.

If the options -s or –docgen are used, Python might automatically create a `__cache__` folder containing a compiled version of the arguments provided, for optimization purposes. You may want to keep it if you run the script often.

# 6 Example using custom arguments

If a modified docgen, template or settings file were provided to you, you can use them as follows:

```
> python OzDoc/OzDoc.py code -s settings.py --docgen docgen.py --template template
```

where `OzDoc/OzDoc.py` is the path to `OzDoc.py`, `code` is the code you want to

generate documentation from (or a folder or text, using -f or -t), `settings.py` is the path to your settings file, `docgen.py` is the path to your docgen script and `template` is the path to the template folder.

# Appendix B

# Developper Manual

This manual is for end users and developers who wish to modify or improve the tool for their own needs. It assumes the User Manual was read first. It covers how to modify or implement one's own docgen, template and settings as well as how to add new functionalities to the Framework.

## 1 The Framework's folder hierarchy

The Framework comes with a clean folder hierarchy that we recommend to keep intact. When adding new files to the tool, they should go in their respective folder. When adding a new functionnality, it should go into the correct src Python script.

```
./OzDoc:
    /docgens                The folder for docgen Python scripts.
    /settings               The folder for settings Python scripts.
    /src                    The folder for OzDoc src scripts.
        ArgumentsHamdler.py Handles input arguments.
        FileHandler.py      Handles files and folder (ex: copy).
        ListTools.py        Tools specific for lists.
        OzDocParser.py      The OzDoc parser.
        ParseEval           Evaluation functions for the parser.
        ParserNode.py       Internal data representing a Node in the AST
        PrintTools          Tools to print info for debugging
        StringTools         Tools specfic for strings.
        WebHandler          Handles HTML web pages
    /templates              The folder for templates.
```

| | |
|---|---|
| `/yattag` | `The Yattag module.` |
| `__init__.py` | `Defines OzDoc as module base for Python.` |
| `OzDoc.py` | `The main OzDoc script.` |

Starting from here, we will use the concise notation to specify modules in `src`.

# 2 Understanding the Framework

The Framework follows these steps at execution of the `OzDoc.py` script:

1. OzDoc recovers input arguments. This is done using ArgumentsHandler. The code is based on the argparse Python module.
2. The settings file is dynamically imported and executed using the FileHandler.import_module function. The code is based on the importlib Python module.
3. An OzDocParser object is instantiated using the received arguments: entry point (file, folder, text) and grammar rules from the imported settings module. The object parses itself and builds its abstract syntax tree at instantiation time.
4. The template directory received in argument is copied to the out directory by FileHandler.
5. The docgen file is dynamically imported and executed just like for the settings file.
6. The `run()` method of the docgen is executed. It receives as argument the OzDocParser object containing the AST, the settings, the template (unmodified source), and the out directory.
7. It is then the DocGen's responsibility to make the documentation in its run() method.

# 3 Modifying the setting file

The settings file's main purpose is to pass variables to the parser and DocGen. In particular, it defines the grammar to be parsed with its mandatory fields.

It is a regular Python script, meaning one can code in it just like in any other script of the framework. This is so that complicated settings variable requiring a script to be generated can be generated directly in the settings file. We highly recommend the developer to keep this file's settings related and put any code not related to the generation of settings inside the DocGen script or where it belongs in the hierarchy.

## 3.1 Representing grammar rules

Please refer to Chapter 3, subsection 2.1 to understand how the grammar works and what we define as priority and non priority context rules.

We represent any non-regex rule as a list of 5 strings:

0. The context type name for this rule
1. A keyword defining what type of rule follows directly (text, symbol, regex, ...). See ParseEval.
2. The context opening rule
3. Again, a keyword defining what type of rule follows
4. The context closing rule

```
["comment_line", "symbol", '%', "symbol", '\n']
    # Rule representing comments in Oz
```

## 3.2 Mandatory fields

There are 2 mandatory variables to be defined inside the setting file. Those variables will be used during the parsing:

1. priority_context_rules : a list of rules of exclusive priority in the parser. If one is detected and the context opened, no other context will be opened until the context is closed. Those rules are evaluated as-is.
2. context_rules : a list of non-priority rules

## 3.3 Example Use : Parsing comments in C

```
priority_context_rules = [
    ["comment_inline", "symbol", '//', "symbol", '\n'],
```

```
    ["comment_box", "symbol", "/*", "symbol", "*/"]]
context_rules = []
```

### 3.4 Example Use : Parse functions in Python

Let's consider one wants to generate documentation for Python using our tool. We want to recovers the comments, the class definitions and the function definitions. We obtain the following (non robust) code:

```
priority_context_rules = [
    ["comment_inline", "symbol", '#', "symbol", '\n'],
    ["comment_box", "symbol", "\"\"\"", "symbol", "\"\"\""]]
context_rules = [
    ["class", "text", "class", "symbol", ":"],
    ["fun", "text", "def", "symbol", ":"]]
```

## 4 Understanding and modifying the Parser

### 4.1 ParserNode

The parser uses instances of ParserNode to represent the AST. A ParserNode contains the following fields:

1. node_id : An unique id generated by the parser during the parsing
2. context_type : A string representing the context name (from the settings files)
3. parent : parent node
4. children : list of children nodes
5. start : offset in the file at which the context starts
6. end : offset in the file at which the context ends
7. line_start : line in the file at which the context starts
8. line_end : line in the file at which the context ends
9. description : usually empty, can contain extra information depending on the node, like the string that opened the context or the path of a file/dir node.

10. code : almost always None except for the node at the root of the tree. Other nodes use indexing with start:end to find their code. This is done for spacial complexity reasons.

## 4.2 Instantiating the Parser

```
def __init__(self, code=None, context_type=None, description=None,
    id_start=0, priority_context_rules=None, context_rules=None):
```

The parser requires a starting node with its code and all the grammar rules to make the AST. If the code is not provided, the parser will try to retrieve the code(s) if the starting node is a file or dir by looking for the path in the description.

The variable `id_start` simply specifies at which id the parser should start counting.

The parser runs the method `build_abstract_syntax_tree()` automatically at the end of it's initialization to build the tree from the starting node.

You can see examples of parser instantiation for the three standard user cases (file, folder, text) in `OzDoc.py`.

## 4.3 The Parser Algorithm

The Parser follows a standard predictive recursive-equivalent descent without backtracking algorithm. it can be found in `build_abstract_syntax_tree()`. Normally, one should not have to do any modification inside the parser.

However, the parser uses methods in the ParseEval module to evaluate the rules. Precisely, it asks `eval_rule()` to decide automatically what method to use, based on the already mentionned rule keyword at index 1 and 3 of the evaluated rule. The standard one are text, symbol, regex and empty:

- symbol : string is evaluated as it
- text : string is evaluated with a lookbehind and lookafter non alphanumeric check to find the exact text (ex: looking for Hey in Heyah will be evaluated as False)
- regex : string is evaluated as regex.
- empty : Should be used for closures only. Closes the context right after it has opponed, to make a leaf node.

Of course, new ones can be added.

## 4.4 Adding new ParseEval method and keyword

A parse eval definition **must** follow the following :

```
def eval_rule_keyword(node, offset, rule_string):
    # do eval
    # if eval says the rule is correct, return the result of the eval
    return result
    # else, MUST return None (or no return statement)
    return None
```

After coding your method, you must add its keyword and the method to the dictionary for it to be accessible:

```
evalfun["keyword"] = eval_rule_keyword
```

Then, one can use its keyword for any rule in the settings file to define new grammar.

## 4.5 Example Use : Evaluate Palindromes

Let us consider that we want the parser to recognize palindromes words. We add a new function in ParseEval that takes a regex to find a word and only returns if the word is a palindrome (using another, not shown, is_palindrome function), and we add it to the dictionary:

```
def eval_rule_palindorme(node, offset, regex):
    ans = regex.match(node.code[offset:])
    if ans is not None:
        ans = group(0)
        return ans if is_palindrome(ans) else return None
    else:
        return None
...
evalfun["palindrome"] = eval_rule_palindrome
```

We can then use this evaluation function in the setting to find palindrome leaves:

```
 ["pal", "palindrome", "[a-z]*", "empty", ""],
```

The parser will handle the rest.

# 5 DocGen script

The docgen file's main purpose is to create the documentation using everything that has been done up until this point.

It is a regular Python script, meaning one can code in it just like in any other script of the framework.

We recommand the developper to use Yattag to write HTML.

## 5.1 Mandatory fields

The docgen script file must contain a `run()` function with the following signature:

```
1 def run(parser, settings, out):
```

Here are the arguments received:

1. parser : The OzParser object with the AST already built.
2. settings : The settings **module**. That means it is already imported and one can access fields using the dot ".".
3. out : The path to the output folder, in which the template folder as already been copied.

## 5.2 Abusing OzDocParser's capabilites and methods

We highly recommend the developer to first clean his AST before writing docgen code. Also, one should not write his docgen from scratch, as many useful methods to build extensive node repository are already provided. We present here **some** of the methods provided by the module OzDocParser:

- `fuse_similar_successive_contexts(node,types)` : Fuse two successive sister nodes in the AST starting from node if they have the same context type and if this context type is in the types list given
- `remove_nodes_type` : Remove node (and all its children) from the AST if its context is in the list provided. Can also take an argument to delete them from memory.

- `build_context_repo` : Build a list of all the nodes whose context type is in the type list given.
- `build_context_repo_not_in` : Same, but allows another list for exclusion to be given
- `build_line_repo` : Build a list where every entry is a list representing a line in the code. Each line entry contains the different node that starts or end at this line, without repitition of the same node twice in the same line entry.
- `build_link_context_with_repo(node,context1,context2)` : Append [node1, node2] to repo if node1 not in repo, node1 is in context1, node2 in context2 and node2 is the first node in context2 to appear after node1. Ignores node depth. This can be used, as an example, to create a list that links function to their definition.

## 5.3 Example Use : Body of a table containing all functions

```
1  doc, tag, text = Doc().tagtext()
2  with tag('tbody'):
3      for function in fun_repo:
4          funcname = function[2]
5          if funcname != '$':
6              with tag('tr'):
7                  with tag('td'):
8                      text(funcname)
9                  with tag('td'):
10                     description = ""
11                     prev_sister = function[0].find_previous_sister()
12                     if prev_sister:
13                         if prev_sister.context_type in settings.
       comment_keyword:  # if previous sister is a comment
14                             description = code[prev_sister.start:
       prev_sister.end].split('\n')[0][:80]
15                             description = description.lstrip('/*% ').
       rstrip('/*% ')
16                     text(description)
17 new_tablebody = doc.getvalue()
18 fh.replace_in_file('@tablebody', new_tablebody, destination)
```

# 6 Templates

The template is the main body in which the docgen will put generated data.

Up until now, we only talked about generating documentation, but we could just as well use the whole framework to generate **anything** that is text based.

We recommend the developer to have a concrete and precise idea of how he will access and edit particular spot of the template. An example solution that we use is simply the use of tags.

If generating HTML, we recommend the developer to follow tutorials on HTML, CSS and Javascript.

## 6.1 Example use: A table waiting for its content

```
<section id="tableaux" class="my-3">
    <div class="container">
        <table class="table table-striped">
            @tablehead
            @tablebody
        </table>
    </div>
</section>
```

An example of @tablebody being replaced can be seen in the code of the previous section.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**