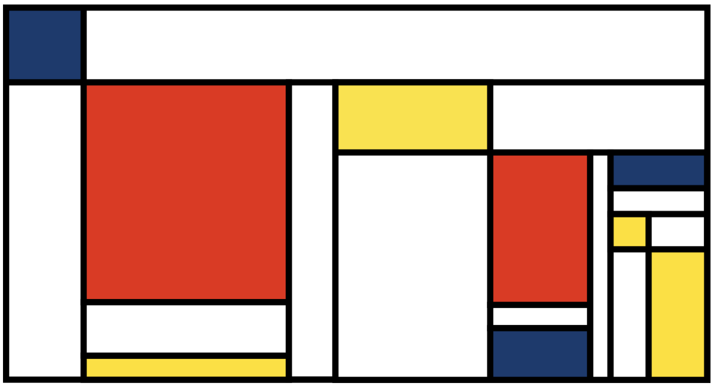
Student Research Project
Documentation

Mondriλn

A Visual Programming Language Based on the Lambda Calculus



Version: 1.0
Date: 2024-12-20

Authors: Anja Friedrich, Mona Panchaud
Advisor: Prof. Dr. Farhad D. Mehta

Department of Computer Science
Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Abstract

The lambda calculus is a core part of every functional programming class. However, students often find it difficult to grasp, especially the process of beta reduction. Hence, representing lambda terms and the beta reduction process visually could improve its comprehensibility to students and motivate them to learn more about the subject. The goal was to design a visual language for the representation of lambda terms which can aid students' understanding of lambda calculus. In addition, the language aims to be visually pleasing, simple to draw and reason about on paper, and straightforward to read for humans. A proof of concept application should be created to ensure that the implementation of this language is feasible.

The team conducted research about existing visual esoteric programming languages and visual lambda calculus representations. The results of this research influenced the design of the language. Additionally, the SVG format was chosen for storing the visual representations.

Mondriλn uses coloured rectangles and their relative positions to represent lambda terms. This design was inspired by the Dutch painter Piet Mondrian. The language allows for artistic freedom without sacrificing its understandability. Furthermore, it is able to represent all lambda terms, only limited by the number of colours displayable on screens. The proof of concept application is implemented as a command line tool and supports generating images from lambda terms, parsing a lambda term from an image and performing beta reduction on them. This proves that it is possible to implement the language we envisioned. Mondriλn appears to make lambda terms simpler to understand, particularly when it comes to the beta reduction process.

While the language shows promise, its potential usefulness to students remains an open question. Further evaluation is needed to determine how effectively it can aid students' understanding of lambda calculus, with a particular focus on the beta reduction process. Some unresolved issues remain in the proof of concept application, for example, when handling large lambda terms. A next step to improve the usability of the system could be to develop a web-based interface, which could also offer additional features. Overall, this project provides a good starting point for further development. By addressing the aforementioned limitations and opportunities for enhancement, the system can become even more usable and effective.

Management Summary

Background

A core part of every functional programming class is the lambda calculus as it forms the basis for functional programming. However, students often find it difficult to grasp, especially the process of beta reduction. Hence, representing lambda terms and the beta reduction process visually could improve its comprehensibility to students. Additionally, allowing experimentation with lambda terms in a visual way could fascinate and, therefore, motivate students to learn more about the subject. Furthermore, visual functional programming languages are an interesting concept, which is not yet well explored.

Objectives

The goal was to design a visual language for the representation of lambda terms which can aid students' understanding of lambda calculus. In addition, the language aims to be visually pleasing, simple to draw and reason about on paper, and straightforward to read for humans. A proof of concept application should be created to ensure that the implementation of this language is feasible.

Approach

Upon starting this project, research was conducted about existing visual programming languages and visual lambda calculus representations. The results of this research influenced the design of the language. Additionally, the team researched various image formats and concluded that SVG is the most suitable for storing the visual representations.

Results

Mondriλn uses coloured rectangles and their relative positions to represent lambda terms. This design was inspired by the Dutch painter Piet Mondrian. The language allows for artistic freedom without sacrificing its understandability. Furthermore, the visual language is able to represent all lambda terms up to the limit of how many colours can be represented by a screen.

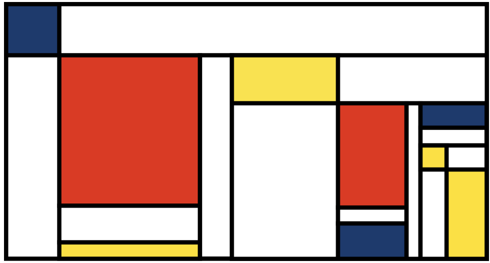


Figure 1: Piet Mondrian inspired representation of the term $\lambda a.b\,c\,(\lambda d.b\,a\,(a\,(\lambda c.c)))$

The proof of concept application is implemented as a command line tool and supports generating images from lambda terms, determining the lambda term contained within an existing image and performing beta reduction on them. This proves that it is possible to implement the language we envisioned. Mondriλn appears to make lambda terms simpler to understand, particularly when it comes to the beta reduction process.

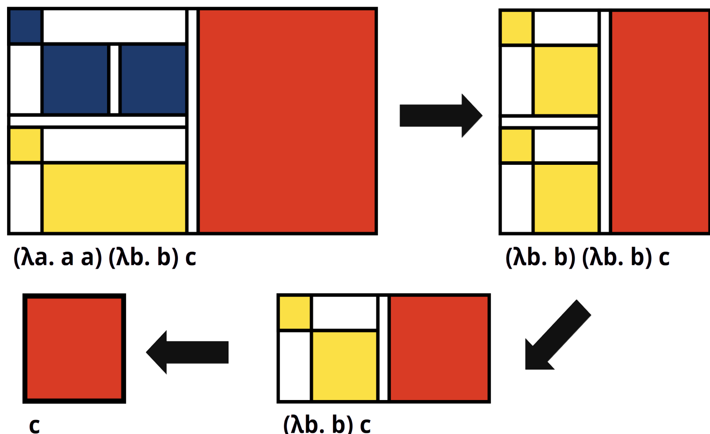


Figure 2: Beta reduction process of the term $(\lambda a.a\,a)\,(\lambda b.b)\,c$

Conclusion

While the language shows promise, its potential usefulness to students remains an open question. Further evaluation is needed to determine how effectively it can aid students' understanding of lambda calculus, with particular attention given to the beta reduction process and its clarity.

Despite the achievements, a few unresolved issues remain in the current proof of concept application. Handling large lambda terms currently presents a challenge under certain circumstances. Additionally, to improve the usability of the system, developing a web-based interface would be a valuable next step. It could also offer additional features, such as interactive animations to visualise the beta reduction process, direct manipulation of elements, and options for users to choose and customise colours for a more personalised experience.

Overall, this project provides a good starting point for further development. By addressing the aforementioned limitations and opportunities for enhancement, the system can become even more usable and effective.

Contents

I Product Documentation **1**

1 Objectives **2**
- 1.1 Background . . . 2
- 1.2 Goals . . . 2
- 1.3 Constraints . . . 3
- 1.4 Deliverables . . . 3

2 Existing esoteric programming languages **4**
- 2.1 Functional programming languages . . . 4
  - 2.1.1 Unlambda . . . 4
  - 2.1.2 Lazy K . . . 4
  - 2.1.3 🆒 . . . 5
  - 2.1.4 Husk . . . 5
  - 2.1.5 SKM calculus . . . 5
  - 2.1.6 Jot . . . 5
  - 2.1.7 Brainfunct-slash . . . 5
- 2.2 Graphical programming languages . . . 5
  - 2.2.1 Piet . . . 6
  - 2.2.2 Turnstyle . . . 6
  - 2.2.3 Pyramid Scheme . . . 6
  - 2.2.4 Rail . . . 6
  - 2.2.5 Skastic . . . 7
- 2.3 Visual Lambda Calculus representations . . . 7
  - 2.3.1 VEX . . . 7
  - 2.3.2 Visual Lambda Calculus . . . 7
  - 2.3.3 Alligator Eggs! . . . 8
  - 2.3.4 To Dissect a Mockingbird . . . 9

3 Requirements **10**
- 3.1 Priorities . . . 10
- 3.2 Use Cases . . . 10
  - 3.2.1 Actors . . . 10
  - 3.2.2 Use case 01: Convert lambda term to image . . . 10
  - 3.2.3 Use case 02: Parse lambda term from image . . . 11
  - 3.2.4 Use case 03: Beta reduce lambda term . . . 11
  - 3.2.5 Use case 04: Configure design of converted image . . . 12
  - 3.2.6 Use case 05: Use pre-defined definitions for input of lambda terms . . . 12
- 3.3 Functional and Non-Functional Requirements . . . 12
  - 3.3.1 Functionality . . . 13
  - 3.3.2 Usability . . . 15
  - 3.3.3 Reliability . . . 15
  - 3.3.4 Performance . . . 15
  - 3.3.5 Supportability . . . 15

4 Language Specification **16**
4.1 Design goals . . . 16
4.2 Lambda term rules . . . 16
4.3 Conventions . . . 16
4.4 Visual Representation . . . 17
4.4.1 Variable . . . 17
4.4.2 Abstraction . . . 17
4.4.3 Application . . . 18
4.4.4 Definitions applying to all building blocks . . . 19
4.4.5 Technical notes . . . 20

5 Design **22**
5.1 Architecture . . . 22
5.1.1 System Overview . . . 22
5.1.2 Application Architecture . . . 22
5.1.3 Architectural Decisions . . . 24
5.2 Visual Representation Design . . . 25
5.2.1 Piet Mondrian inspiration . . . 25
5.2.2 Application Separators . . . 26
5.2.3 Limitations in representing all terms . . . 27
5.3 Automatic colour selection . . . 27

6 Testing **29**
6.1 Testing methods . . . 29
6.1.1 Unit/Integration testing . . . 29
6.1.2 Property-based testing . . . 29
6.1.3 Manual testing scenarios . . . 29
6.2 Evaluation of test results . . . 30
6.2.1 Unit tests . . . 30
6.2.2 Property-based testing . . . 31
6.2.3 Manual testing scenarios . . . 31
6.3 Fulfilment of requirements . . . 32
6.3.1 Functionality . . . 32
6.3.2 Usability . . . 34
6.3.3 Reliability . . . 34
6.3.4 Performance . . . 35
6.3.5 Supportability . . . 35

7 Evaluation **36**
7.1 Church Numerals . . . 36
7.2 succ 0 . . . 37
7.3 Addition . . . 37
7.4 Endless Loop . . . 39
7.5 Y-Combinator . . . 40
7.6 Not True . . . 41

8 Conclusion and Outlook **42**

II Project Documentation **43**

9 Project Management **44**
9.1 Methodology . . . 44
9.2 Project Plan . . . 44
9.3 Tools . . . 45

9.3.1 Project Management . . . 45
9.3.2 Software Development . . . 45
9.3.3 Documentation . . . 45
9.4 Quality Measures . . . 45
9.4.1 Organisational Means . . . 46
9.4.2 Code Quality . . . 46
9.4.3 Documentation Quality . . . 46
9.5 Risk Management . . . 47
9.5.1 Risks and their impact . . . 47
9.5.2 Risk Assessment . . . 47

Bibliography **50**

List of Figures **52**

List of Tables **53**

Listings **54**

III Appendix **55**

A README document . . . 56
B Document with instructions for creating your own images . . . 59

Part I

Product Documentation

Chapter 1

Objectives

This chapter gives an overview of the projects' objectives.

1.1 Background

An esoteric programming language (esolang) is a programming language designed to test the boundaries of programming language design, as a proof of concept, or as software art [1]. Designing and implementing esolangs provides valuable hands-on experience in various disciplines including grammar rules, lexical analysis, and parsing. Most of the existing esolangs are imperative, meaning that their programs consist of statements which change a program's state. Functional esolangs, on the other hand, are not yet well explored. Additionally, in our research on existing esolangs, we found that visual functional esolangs are particularly interesting. This led us to the idea of creating a visual representation of the lambda calculus, which forms the basis for functional programming and is a core part of every functional programming class. Students often find the lambda calculus difficult to grasp, especially when it comes to the process of beta reduction. Hence, representing lambda terms and the beta reduction process visually could improve its comprehensibility to students. Furthermore, allowing experimentation with lambda terms in a visual way could fascinate and, therefore, motivate students to learn more about the subject.

1.2 Goals

The main goal of this project is to design a visual language for the representation of lambda terms which can aid students' understanding of lambda calculus. The visual language should be able to represent all possible lambda terms. Moreover, the language's syntax must be formally specified.

The team aims to conduct research about existing functional esolangs, visual esolangs, and visual lambda calculus representations and let the results of this research guide the design of the language.

To ensure that the implementation of the language is feasible, and to allow for easy experimentation, a proof of concept application should be created. This proof of concept application should be able to generate images from lambda terms, determine the lambda term contained within an existing image, and perform beta reduction on them.

1.3 Constraints

The proof of concept application should be implemented in the functional programming language Haskell.

1.4 Deliverables

The deliverables include:

- A formal language specification of a visual language which can represent all lambda terms
- A proof of concept application which can generate visual representations from lambda terms, parse existing representations, and perform beta reduction on them
- Detailed documentation of the project, the product, and the research on existing esolangs

Chapter 2

Existing esoteric programming languages

This chapter gives an overview of existing esoteric programming languages, both functional and graphical languages, as well as existing visual lambda calculus representations.

2.1 Functional programming languages

Esoteric functional programming languages are typically deeply rooted in the principles of lambda calculus. These languages place strong emphasis on immutability, first-class functions, and the use of higher-order functions, while deliberately deviating from conventional language design for the purposes of humour, novelty, or artistic expression. Many esoteric functional programming languages are Turing-complete, meaning they can theoretically compute any computable function, despite their unconventional syntax and semantics. Esoteric functional programming languages are often used for educational purposes, to explore the boundaries of programming language design, or for code golfing competitions, where the goal is to write the shortest possible program to solve a given problem [1].

The following sections provide a brief overview of several esoteric functional programming languages, each illustrating unique and unconventional approaches to computation.

2.1.1 Unlambda

Unlambda is designed around the concept of combinatory logic, which eliminates the need for variables by using function application and combinators. It is strongly inspired by the SKI combinator calculus, which consists of three basic combinators: S, K, and I. Unlambda is a Turing-complete functional programming language, which avoids variables, assignments, and control structures typically seen in imperative programming. Instead, it is purely centred on the concept of applying functions to arguments. Unlike many other functional esolangs, Unlambda is considered impure because it permits side effects, such as printing output to the screen through specific operations. Unlambda uses several basic functions (combinators): s, k, i, ., d [2].

2.1.2 Lazy K

Lazy K is based on combinatory logic and inspired by SKI combinator calculus. The language is notable for its use of lazy evaluation, a key concept in functional programming that delays the evaluation of expressions until their results are actually needed. Lazy K was designed to address the lack of functional purity of Unlambda and as such it is Turing-complete as well as purely

functional [3]. Functional purity is a key concept in functional programming which ensures that functions have no side effects and always return the same result for the same input.

2.1.3 🆒

🆒 is an esoteric programming language which exclusively uses emojis for source code, input, and output. It is easy to model the lambda calculus in 🆒, which makes the language Turing-complete. 🆒 consists of four types of elements: sequences, definitions, variables and literals. It offers several built-in functions, such as addition, subtraction, multiplication, division, comparison, logical operations, and print, as well as the variables True and False [4].

2.1.4 Husk

Husk is a minimalistic programming language that is based on combinatory logic and designed to be as concise as possible. It is primarily used for code golfing and operates exclusively with pure functions. Input and output are handled via command-line arguments, which can be accessed using one of the superscript number instructions in the language. Additionally, Husk employs lazy evaluation [5].

2.1.5 SKM calculus

SKM calculus is a minimalistic, Turing-complete esoteric programming language based on SKI combinator calculus, which uses the combinator M instead of the I combinator. For any expression a, expression $M\ a$ reduces to a if and only if a reduces to either K or S. This leads to only one possible reduction for any expression in the SKM calculus, making it a simple language. This change in the combinator set results in a language that allows for the implementation of safe data structures without the need for a type system [6].

2.1.6 Jot

Jot is based on the concept of combinatory logic, namely on SKI calculus, and is Turing-complete. It only uses 0 and 1 as elements and can encode lambda calculus directly in binary form [7].

2.1.7 Brainfunct-slash

Brainfunct is a functional programming language inspired by the esoteric language Brainfuck, and comes in two distinct versions: Brainfunct-slash and Brainfunct-nested. In Brainfunct-slash, functions are separated by the "/" symbol, with the final function serving as the main one, which cannot be called directly but is executed when the program starts. Unlike Brainfunct-nested, Brainfunct-slash does not support nested functions, making it non Turing-complete [8].

2.2 Graphical programming languages

There are programming languages which rely heavily or only on graphical elements. The following sections provide a brief overview of several of these visual programming languages.

2.2.1 Piet

Piet is a stack-based esoteric programming language where programs are represented as abstract paintings. The language is named after the painter Piet Mondrian. Instead of traditional code, Piet uses 18 different colours, made up of 6 hues, each with three variations of lightness. Commands in Piet are executed based on the changes in lightness and hue between adjacent colour blocks. Lightness can either remain the same or become darker by 1 or 2 steps, while hue shifts by 0-5 steps. Both lightness and hue are aligned in a loop, meaning that after the darkest shade or the maximum hue step is reached, the sequence wraps back around to the lightest shade or the initial hue, respectively. White pixels are treated as blank spaces and are skipped during execution. Black pixels act as barriers: when encountered, the “Codel Choser” (responsible for navigating through the painting) changes direction to continue interpreting the program [9].

2.2.2 Turnstyle

Turnstyle is Piet’s functional sister, designed for reading and evaluating expressions from images. Each expression is defined by specific positions and a designated heading (direction), allowing for organised and clear evaluation. There are five distinct kinds of expressions, each displayed in a unique manner: variables, lambda abstractions, function applications, symbols, and identity [10].

2.2.3 Pyramid Scheme

Pyramid Scheme is a functional, esoteric dialect of Lisp, where instead of traditional brackets, pyramids are used to determine evaluation order. Each pyramid can accept two arguments. The language supports a wide range of standard operations like addition, subtraction, loops, and conditionals [11]. Listing 2.1 shows an example program that does a conditional check and prints 1 if A is true, or 0 if A is false.

```
       ^
      /?\
     ^---^
    /!\ / \
   ^---/out\
  /?\  -----^
 ^---^     /0\
/A\ / \    ---
---/out\
  ^-----
 /1\
 ---
```

Listing 2.1: Pyramid scheme program that does a conditional check and prints 1 if A is true, or 0 if A is false

2.2.4 Rail

Rail is a two-dimensional functional programming language with immutable values and LISP-style lists. It is modelled after a train on a rail system, where the flow of execution follows tracks, reminiscent of an abstract railway network. Rail’s design includes local variables and a clean procedural model, as well as a rich set of built-in functions for handling conditionals,

mathematical operations, and more [12]. Listing 2.2 shows an example program that reads from standard input and prints to standard output.

```
$ 'main' (--):
 \
 | /---------\
 | |         |
 | \    /-io-/
 \---e-<
       \-#
```

Listing 2.2: Rail program that reads from standard input and prints to standard output

2.2.5 Skastic

Skastic is a functional programming language where the source code is not written in the usual text format but instead exists within an image file. Programs are visualised as nodes and edges, which form an abstract syntax tree. Instead of traditional code writing, nodes and edges are drawn to represent code blocks and their relationships, effectively building a tree-like structure of operations. What makes Skastic unique is that its interpreter can read and process these graphical trees. This allows programmers to create a program by sketching, scanning, or drawing their logic in a graphical format. The interpreter relies heavily on image analysis techniques [13].

2.3 Visual Lambda Calculus representations

There are several definitions of the lambda calculus represented exclusively or for the most part in a visual way. The following sections provide a brief overview of several of these definitions.

2.3.1 VEX

VEX is a visual representation of the lambda calculus. It uses the following rules to visually represent lambda terms [14]:

- Expressions are always contained in a circle.
- Parameters are circles, that must be contained inside a circle and touch the outer circle. The parameters are labelled with their name.
- Variables are circles. Which value they have is clear from the connecting line to a parameter circle.
- Applications are represented by arrows. The arrows need to be labelled, to clarify the order of application. This makes the representation not fully visual.

Figure 2.1 shows an example of how these rules result in a visual representation.

2.3.2 Visual Lambda Calculus

Visual lambda calculus is a purely visual representation of the lambda calculus using the "bubble notation", a term introduced by its creator. The bubble notation uses the following elements to represent lambda terms:

- Circles represent either a variable or an abstraction.

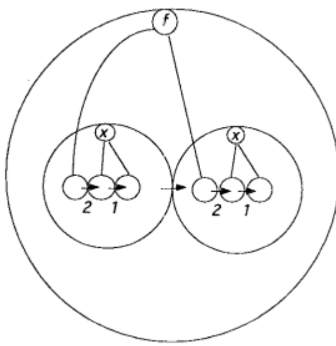


Figure 2.1: VEX Representation of the Y-Combinator $\lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$ [14]

- Variables are represented using empty coloured circles.
- Abstractions are represented by a circle with a colour (the parameter) and contain other circles. The contained circles represent the term inside of the abstraction.
- Applications are represented by partially overlapping circles, with the argument circle below the function circle.

Figure 2.2 shows an example of how these rules result in a visual representation.

The bubble notation has the advantage that it allows for smooth animations when beta reducing lambda terms. Additionally, this representation features a graphical environment to play around with lambda terms using the bubble notation [15].

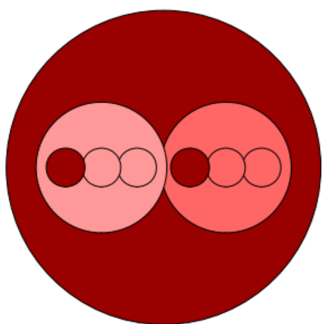


Figure 2.2: Bubble notation of the Y-Combinator $\lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$ [15]

2.3.3 Alligator Eggs!

"Alligator Eggs!" is a paper puzzle game designed for children. It is a purely visual representation of the lambda calculus.

The following elements are used for representing lambda terms:

- Coloured eggs represent variables. The colour represents the variable name.
- Hungry alligators represent abstractions. The colour represents the variable name of the function input.
- Applications are represented as two alligator families next to each other.
- Parentheses are represented by old alligators.
- All terms are represented in tree form but tree edges are not drawn.

Figure 2.3 shows an example of how these rules result in a visual representation.

To visualize beta reduction the so called “eating rule” is introduced which simply says that hungry alligators eat the family to their right. Afterwards, they die from overeating, and if they were guarding any eggs of their own colour, what they have just eaten will hatch out of the eggs [16].

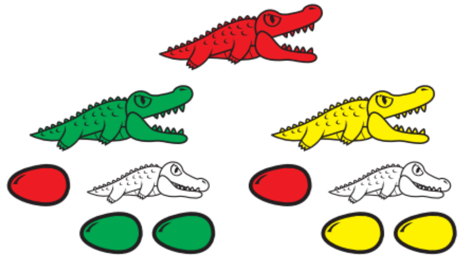


Figure 2.3: “Alligator Eggs!” representation of the Y-Combinator $\lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$ [15]

2.3.4 To Dissect a Mockingbird

“To Dissect a Mockingbird” is a purely visual representation of the lambda calculus. It imagines functions as birds where inputs are songs that the birds hear and outputs are the songs they sing.

To represent lambda terms, it uses boxes with a dotted outline (abstractions). These boxes have a semicircle as their input which are connected with lines to circles or the output semicircle. Circles represent the “brain cells” of the bird where the function application happens. Circles can also be connected to each other with lines. Figure 2.4 shows an example of how these rules result in a visual representation.

Thanks to the usage of lines, this representation is purely visual and additionally eliminates the need for alpha conversion. However, the multiple meanings of lines makes it more complex to read. Most importantly, this notation allows for smooth animations when beta reducing lambda terms [17]. Unfortunately the representation lacks a formal description.

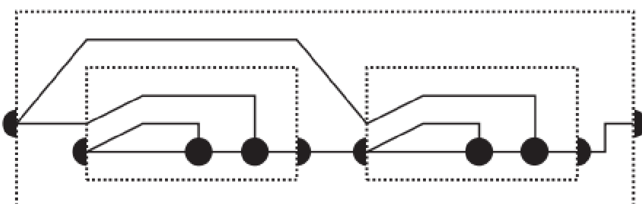


Figure 2.4: “To Dissect a Mockingbird” representation of the Y-Combinator $\lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$ [17]

Chapter 3

Requirements

This chapter gives an overview of the requirements for the developed system and how we prioritised them. As a first step, use cases were identified, from which functional and non-functional requirements were derived.

3.1 Priorities

To prioritise our work on the functional and non-functional requirements, we defined the following priorities.

Priority 1 These requirements are needed to build a minimum viable product, providing just enough features to be useful to users.

Priority 2 These requirements will be implemented after completion of the minimum viable product, if there is enough time left.

Priority 3 These requirements will be implemented in the improvements phase, if there is enough time left and we deem the requirements to be important enough.

3.2 Use Cases

The following use cases have been defined in casual format as described by Larman [18]. Each use case is assigned a unique ID, which is later used for reference throughout this chapter (e.g. UC01).

The diagram in figure 3.1 shows an overview of all defined use cases.

3.2.1 Actors

There is only one kind of user for this system, therefore we employ the ”User” as a primary actor for all use cases.

3.2.2 Use case 01: Convert lambda term to image

3.2.2.1 Main success scenario

The user submits a valid lambda term to the system. The system converts the given term into an image in the defined visual representation format. This image is saved to the users file system.

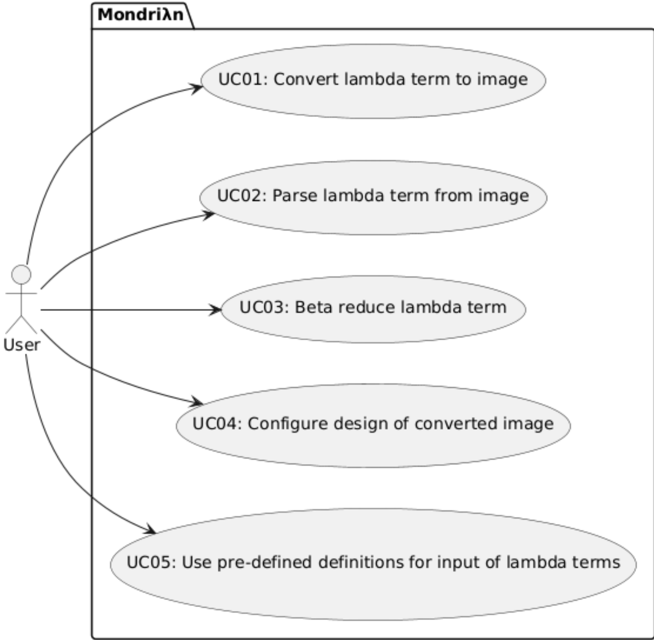


Figure 3.1: Use Case Diagram

3.2.2.2 Alternate scenarios

If the lambda term given by the user is not in a valid format, the system responds with an error message.

3.2.3 Use case 02: Parse lambda term from image

3.2.3.1 Main success scenario

The user presents the system with an image in the defined visual representation format. The system parses the given image to receive the lambda term contained within. The lambda term is shown to the user.

3.2.3.2 Alternate scenarios

If the image given by the user is not in the defined visual representation format, the system responds with an error message.

3.2.4 Use case 03: Beta reduce lambda term

3.2.4.1 Main success scenario

The user provides the system with a valid lambda term, either by submitting an image in the specified visual representation format, from which the lambda term is parsed as outlined in UC02, or by directly inputting a valid lambda term, as described in UC01. The system applies beta reduction using the "leftmost outermost first" strategy to reduce the term once. The resulting term of the reduction is shown to the user. The reduction step is converted to an image and saved to the file system as described in UC01. The file is labelled according to the number of the reduction step that it represents. If the term can be reduced further, the user is asked if the program should execute the next reduction step. If the user confirms, another reduction step is executed. If the term cannot be further reduced, the user is informed accordingly.

3.2.4.2 Alternate scenarios

If the lambda term or image given by the user is not in a valid format, the system will respond with an error message as described in UC01 and UC02.

Using a configuration option, the user is able to choose the alternative strategy "leftmost innermost first" instead of the default strategy for beta reduction.

3.2.5 Use case 04: Configure design of converted image

3.2.5.1 Main success scenario

The user presents the system with a valid lambda term and with a configuration regarding how the resulting image should look. The system converts the given term into an image in the defined visual representation format. When converting the image, the system adheres to the given configuration. This image is saved to the users file system.

3.2.5.2 Alternate scenarios

If the configuration provided by the user is not in a valid format, the system will respond with an error message.

If the lambda term given by the user is not in a valid format, the system will respond with an error message as described in UC01.

3.2.6 Use case 05: Use pre-defined definitions for input of lambda terms

3.2.6.1 Main success scenario

The system has a set of pre-defined definitions for lambda terms. The user submits a valid lambda term containing pre-defined definitions to the system. The system replaces all pre-defined definitions using delta reduction. Then, the system converts the resulting term into an image in the defined visual representation format. This image is saved to the users file system.

3.2.6.2 Alternate scenarios

The alternate scenarios described in UC01.

3.3 Functional and Non-Functional Requirements

From the use cases described above, we derived functional and non-functional requirements. These are grouped using *FURPS*, an acronym which stands for *Functionality*, *Usability*, *Reliability*, *Performance* and *Supportability* [19]. Each requirement has a unique ID so that it can be referred to in later chapters.

This ID is structured as follows:

Functionality RQ-F-{Number}

Usability RQ-U-{Number}

Reliability RQ-R-{Number}

Performance RQ-P-{Number}

Supportability RQ-S-{Number}

We assigned each requirement one of the aforementioned priorities so that they can be planned appropriately.

3.3.1 Functionality

The sections below show all requirements regarding functionality grouped by their use case.

3.3.1.1 UC01: Convert lambda term to image

ID	Name	Priority
RQ-F-100	The system is able to convert lambda terms containing only one variable.	1
RQ-F-110	The system is able to convert lambda terms containing only one lambda abstraction.	1
RQ-F-120	The system is able to convert lambda terms containing only one application of a lambda abstraction and a variable.	1
RQ-F-125	The system is able to convert lambda terms containing only one application of two variables.	1
RQ-F-130	The system is able to convert lambda terms containing only one application of two lambda abstractions.	1
RQ-F-140	The system is able to convert lambda terms containing only two applications of lambda abstractions and/or variables.	1
RQ-F-150	The system is able to convert lambda terms containing any combination of variables, abstractions and application.	2
RQ-F-155	The system is able to convert lambda terms containing parentheses that alter the default order of application (left-associative).	2
RQ-F-160	The system is able to detect an invalid lambda term and inform the user.	1

Table 3.1: Functional requirements for UC01

3.3.1.2 UC02: Parse lambda term from image

ID	Name	Priority
RQ-F-200	The system is able to parse lambda terms from images containing only one variable.	1
RQ-F-210	The system is able to parse lambda terms from images containing only one lambda abstraction.	1
RQ-F-220	The system is able to parse lambda terms from images containing only one application of a lambda abstraction and a variable.	1
RQ-F-225	The system is able to parse lambda terms containing only one application of two variables.	1
RQ-F-230	The system is able to parse lambda terms from images containing only one application of two lambda abstractions.	1
RQ-F-240	The system is able to parse lambda terms from images containing only two applications of lambda abstractions and/or variables.	1
RQ-F-250	The system is able to parse lambda terms from images containing any combination of variables, abstractions and application.	2
RQ-F-260	The system is able to detect an invalid visual representation and informs the user.	1
RQ-F-270	The system allows for a fault resistance of up to 0.2 units when parsing images.	3

Table 3.2: Functional requirements for UC02

3.3.1.3 UC03: Beta reduce lambda term

ID	Name	Priority
RQ-F-300	The system reduces lambda terms and images using the "leftmost outermost first" strategy, informs the user of the reduced term.	2
RQ-F-310	The system saves a file for each reduction step, labelled with the number of the reduction step.	2
RQ-F-320	The system asks the user if they want to continue the reduction process after each reduction step.	2
RQ-F-325	The system informs the user if a term cannot be further reduced.	2
RQ-F-330	The system ensures that the width and height of images remain consistent, and the overall appearance and visual characteristics of the images are preserved.	3
RQ-F-335	The system uses the same colour for a given variable name throughout the reduction process.	3
RQ-F-340	The system applies alpha renaming to resolve shadowing when two distinct variables have the same colour, assigning a new, unused colour to one of them.	3
RQ-F-350	The system is able to use the alternative strategy "leftmost innermost first" for reduction, if this configuration option is given.	3

Table 3.3: Functional requirements for UC03

3.3.1.4 UC04: Configure design of converted image

ID	Name	Priority
RQ-F-400	The system is able to accept configuration regarding how the converted image should look and adheres to this configuration when converting images.	3
RQ-F-410	The system is able to detect that a given configuration is invalid and informs the user.	3

Table 3.4: Functional requirements for UC04

3.3.1.5 UC05: Use pre-defined definitions for input of lambda terms

ID	Name	Priority
RQ-F-500	The system is able to identify pre-defined definitions in lambda terms and replace them using delta reduction before converting the lambda term to an image.	3

Table 3.5: Functional requirements for UC05

3.3.2 Usability

ID	Name	Priority
RQ-U-100	Users who have previous experience with lambda calculus and have read the documentation regarding how the visual representation works, are able to determine what lambda term is represented in images containing simple lambda terms.	1
RQ-U-200	Users are able to access documentation on the system's CLI from within the app.	3

Table 3.6: Usability requirements

3.3.3 Reliability

ID	Name	Priority
RQ-R-100	The system handles errors gracefully and provides users with a comprehensible error message.	2

Table 3.7: Reliability requirements

3.3.4 Performance

ID	Name	Priority
RQ-P-100	The system converts lambda terms to images within 8 seconds.	1
RQ-P-200	The system parses lambda terms from images within 10 seconds.	1
RQ-P-300	The system executes beta reduction on lambda terms with a maximum of 10 reduction steps within 20 seconds.	2

Table 3.8: Performance requirements

3.3.5 Supportability

ID	Name	Priority
RQ-S-100	All components used are open source.	1
RQ-S-200	The system provides a command-line application with clear usage instructions to guide users in executing commands and utilising its features effectively.	2
RQ-S-300	The system provides a client-side web application that can be accessed and used directly through a web browser without requiring any installation on the user's device.	3

Table 3.9: Supportability requirements

Chapter 4

Language Specification

This chapter introduces the rules of our visual representation of the type-free lambda calculus, named Mondriλn.

4.1 Design goals

We set the following goals for our visual representation:

- All lambda terms should be representable.
- The representation should be easy to read for humans.
- It should be easy to draw and reason about the representation on paper.
- The resulting images should be visually pleasing.

From these goals and our research on existing visual representations presented in section 2.3, we made the following decisions:

- The visual representation should use as few lines and different shapes as possible, namely just rectangles.
- Colours should be used to represent variables.
- There should be as few special cases for the representation as possible.
- The order of application must always be explicit in the visual representation. This should not be represented by using special shapes but by the visual layout of the shapes.

4.2 Lambda term rules

A lambda term ("term") in type-free lambda calculus is built using the following rules [20]:

- x: A **variable** is a character or string representing a parameter.
- $(\lambda x.M)$: A lambda **abstraction** is a function definition, taking as input the bound variable x and returning the body M.
- $(M\,N)$: An **application**, applying a function M to an argument N. Both M and N are lambda terms.

4.3 Conventions

The previous definition produces terms in the form of $((\lambda x.(\lambda y.(x\,y)))\,(a\,b))$. To make the example terms in this document easier to read, we use the following conventions:

- Parentheses of an abstraction extend as far right as possible, meaning $\lambda y.y\,\lambda x.x\,y$ represents $(\lambda y.y\,(\lambda x.x\,y))$
- The outermost parentheses are not written.

4.4 Visual Representation

The following sections show how the rules described in section 4.2 are visually represented.

Very similar to lambda calculus, a valid visual representation is built from the following "building blocks":

- A variable
- An abstraction containing a bound variable and another building block
- An application of two building blocks

How these building blocks are represented is described in the following sections. Section 4.4.4 provides more information that applies to all of the building blocks.

4.4.1 Variable

A variable is represented with a solid colour rectangle. The colour of the rectangle represents its variable name. The rectangle must have an outline.

Figure 4.1: Representation of the term: x

4.4.2 Abstraction

An abstraction in the form of $\lambda x.M$ is represented by the following elements:

- A white rectangle that has an outline and acts as a bounding box.
- On the layer above the bounding box and with the same coordinates as its top left corner, there is a solid colour rectangle. The rectangle must have an outline.
- Another building block which is diagonally adjacent to the solid colour rectangle. The building block must fill up the remaining space to the bottom/right until the outer edges of the bounding box.

The rectangle in the top left represents the input to the function with the colour representing its variable name.

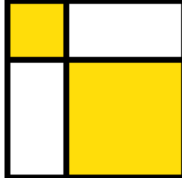


Figure 4.2: Representation of the term: $\lambda x.x$

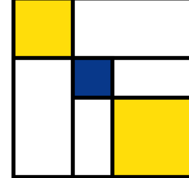


Figure 4.3: Representation of the term: $\lambda x.\lambda y.x$

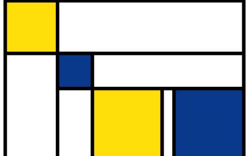


Figure 4.4: Representation of the term: $\lambda x.\lambda y.x\,y$ (see section 4.4.3 for more details)

4.4.3 Application

Applications are represented by building blocks adjacent to each other. They are separated by a white rectangle with an outline in-between them ("separator").

In the simplest case of just one application (e.g. $x\,y$) the building blocks are horizontally adjacent to each other, with a separator in-between. The building block on the left represents the first term (x), whereas the building block on the right represents the second term (y). The building blocks and the separator must all have the same height.

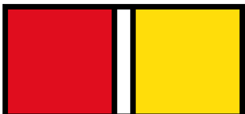


Figure 4.5: Representation of the term: $x\,y$

When there are multiple applications in the current abstraction context, the following rules apply:

- For each level into the term, the adjacency of the building blocks and the separator alternates between vertically adjacent and horizontally adjacent. It starts with horizontal adjacency for the outermost term.
  - In the case of vertical adjacency: The building blocks and the separator must all have the same **width**. The sum of the **heights** of the building blocks and the separator must equal the **height** of the outer building block, the block they are **horizontally** adjacent to. The building block on the top represents the first part of the application, whereas the building block on the bottom represents the second part of the application.
  - In the case of horizontal adjacency: The building blocks and the separator must all have the same **height**. The sum of the widths of the building blocks and the separator must equal the **width** of the outer building block, the block they are **vertically** adjacent to. The building block on the left represents the first part of the application, whereas the building block on the right represents the second part of the application.

As previously mentioned: If a new abstraction context is opened, the count of applications is reset i.e. for the next application, the simplest case is considered first.

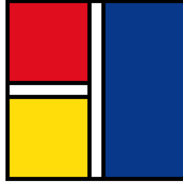


Figure 4.6: Representation of the term: $(x\,y)\,z$

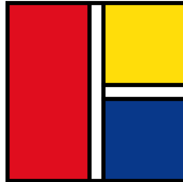


Figure 4.7: Representation of the term: $x(y\,z)$

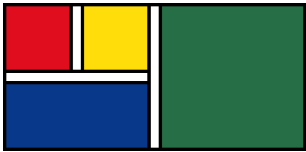


Figure 4.8: Representation of the term: $((a\,b)\,c)\,d$

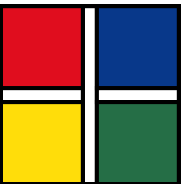


Figure 4.9: Representation of the term: $(a\,b)\,(c\,d)$

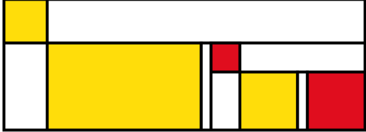


Figure 4.10: Representation of the term: $\lambda x.x\,(\lambda y.x\,y)$

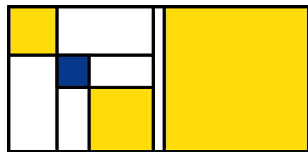


Figure 4.11: Representation of the term: $(\lambda x.\lambda y.x)\,x$

4.4.4 Definitions applying to all building blocks

The following sections provide several definitions which apply to all building blocks.

4.4.4.1 First building block

The first building block, meaning the leftmost innermost variable or abstraction, must always be positioned in the top left corner of the image.

4.4.4.2 Adjacency

A rectangle is considered adjacent to another rectangle if the content (i.e. solid colour) is adjacent. As a result, the outlines of the rectangles overlap.

4.4.4.3 Outlines

The outline of all shapes must use the colour black (#000000). Within one visual representation the outline must always have the same width.

4.4.4.4 Solid Colours

Black (#000000) is not allowed to be used as a solid colour.

For the following cases white (#ffffff) must be used as a solid colour:

- bounding boxes (rectangle surrounding an abstraction)
- separators (rectangle separating two building blocks in applications)

In other cases, white is not allowed to be used as a solid colour. All other colours are always allowed to be used as a solid colour.

Free and bound variables/colours In lambda calculus a variable (for example x) does not necessarily have the same meaning throughout a lambda term. The meaning depends on whether this variable is bound or free in a given context.

The same applies to colours used in an image. As colours represents a variable, they have the same properties and can also be bound or free.

4.4.5 Technical notes

The following notes refer to definitions and restrictions due to the usage of the SVG format.

4.4.5.1 Tags for shapes

The tags used for rectangles must be `<rect>` with a set height/width, fill, stroke colour and stroke width. The alpha channel of the specified colours is ignored. Other tags (e.g `<path>`, `<polyline>` etc.) are not allowed to be used.

4.4.5.2 Length units

For all dimensions, so-called “user units” (i.e. plain numbers) must be used.

4.4.5.3 SVG Tag

The width, height and viewbox of the `<svg>` tag are not considered. It is assumed that all elements in the SVG file are in the visible area.

4.4.5.4 Transform

If any transforms are used, they are ignored.

Chapter 5

Design

This chapter gives an overview of the design of the software system.

5.1 Architecture

The goal of this section is to give an overview of the projects architecture using the C4 model. The C4 model was originally introduced by Simon Brown and is inspired by UML and the 4+1 model for software architecture [21]. The code diagram is not covered because this level of detail is not needed for understanding the overall architecture.

5.1.1 System Overview

Figure 5.1 shows a high-level overview of the software system. The diagram shows that no additional systems are involved and that there is only one actor, as already described in section 3.2.1.

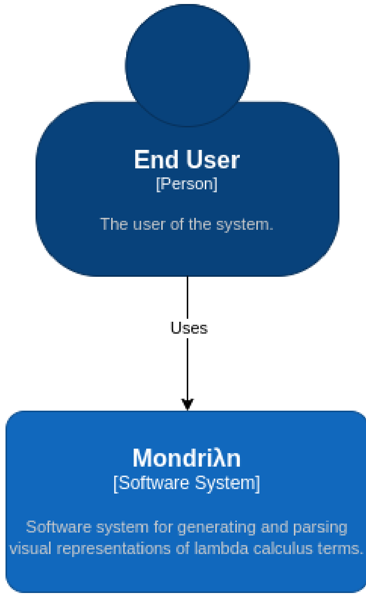


Figure 5.1: System Context Diagram

5.1.2 Application Architecture

Figure 5.2 gives an overview of the components of the software system. All users only interact with the CLI tool. The CLI tool interacts with the Core part of the system, which is responsible

for generating, parsing and beta-reducing lambda terms.

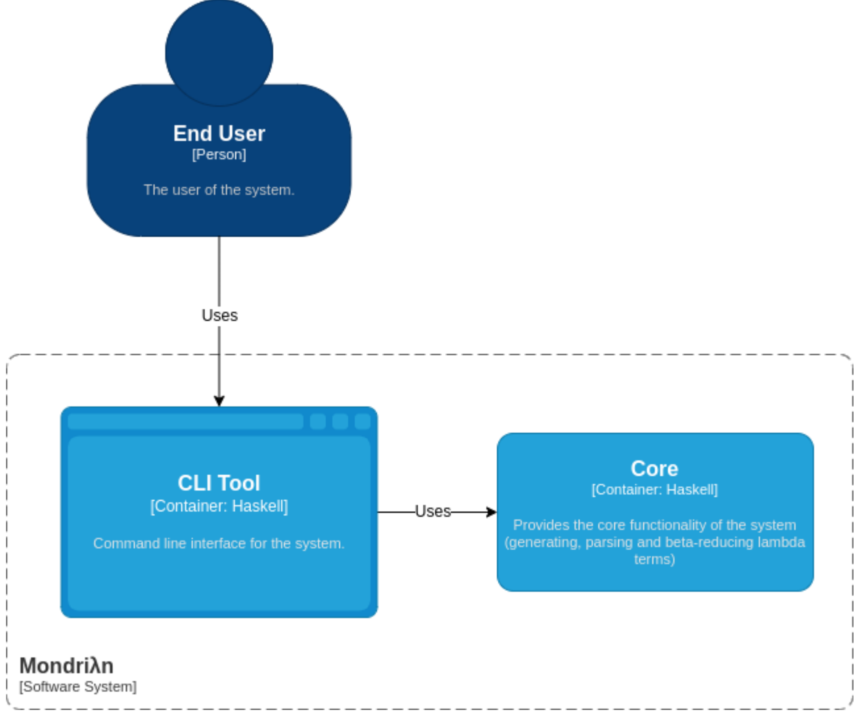


Figure 5.2: Container Diagram

5.1.2.1 CLI Tool

The CLI is the entry point for users to interact with the system. In this project the CLI serves the purpose of demonstrating the main functionality of the system. In the future, the entry point for the user could also be a web application, but this was out of scope for this project.

The CLI provides commands for the following operations:

- Generating an image from a given lambda term
- Parsing a lambda term from a given image
- Beta-reducing a given lambda term, and generating an image for the initial term and the subsequent reduction steps.
- Beta-reducing a given image, and generating images for the reduction steps.

Beta reduction The commands for beta reduction reduce a term only one step at a time. If the resulting term is still reducible after one step of beta reduction, the user is asked if they want to continue with the next reduction step. This choice is given to the user because of the following reasons:

- Trying to reduce every term until the beta normal form is reached could result in an endless loop because some lambda terms have no beta normal form.
- We cannot determine if a lambda term would result in an endless loop or has a beta normal form, as these are undecidable problems [20].

5.1.2.2 Core

Figure 5.3 shows all components inside the Core part of the software system. The following paragraphs describe the responsibility of each component and its relationship to the other components.

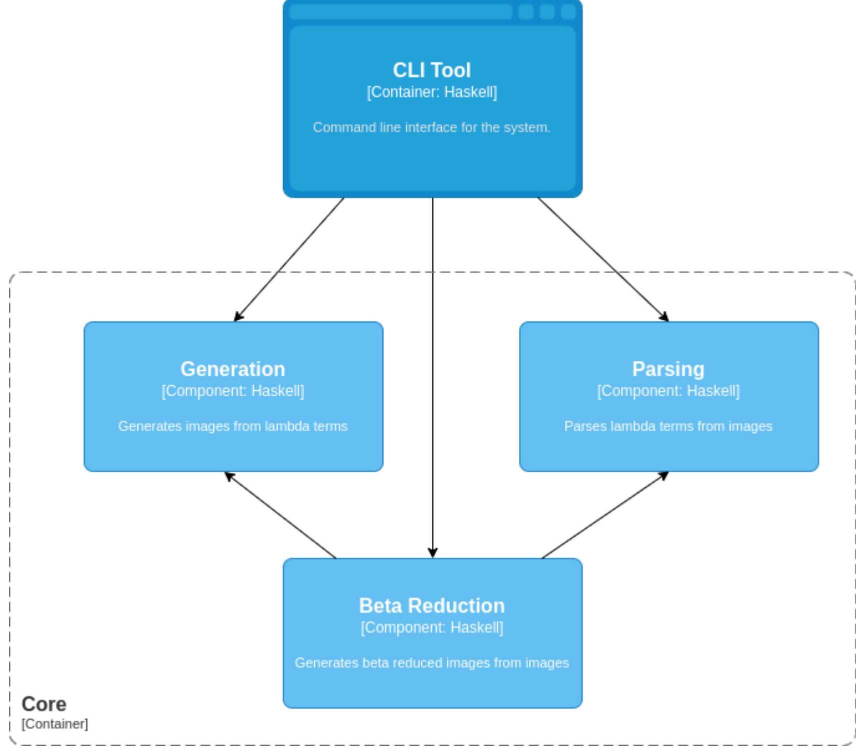


Figure 5.3: Component Diagram of Core

Generation The generation component generates images either from a user provided lambda term or from an intermediate representation. To achieve this, the steps shown in figure 5.4 are executed.

The diagram only shows the execution path where no errors occur. The process can fail both at step 1 (when the user provided term is invalid) and step 5 (when there are IO errors). If this is the case, it returns an appropriate error. When starting with an intermediate representation, the process starts at step 3.

Figure 5.4: Process of generating images from lambda terms

Parsing The parsing component parses existing images (either previously generated by the tool or by the user) either to a lambda term or to an intermediate representation. It checks whether the rules of the visual representation are followed and if not, fails with an appropriate error.

Beta Reduction The beta reduction component reduces existing images or intermediate representations by one reduction step. The beta-reduced term is then saved as an image again. To parse existing images and generate beta-reduced images, the component uses the image parser and image generation components.

5.1.3 Architectural Decisions

The previously described architecture is based on architectural decisions (AD) that we have made. To capture these architecturally significant decisions we use the Y-statement template proposed by O. Zimmermann which focuses on keeping architectural decision records (ADR) lean and still answer the most important "why" questions [22].

Each ADR has a unique ID, structured as AD-{Number}. The AD log is ordered chronologically.

AD-01 SVG format In the context of use cases UC01 and UC02, facing the need to save and read graphical representations of lambda terms, we decided to use the SVG format and neglected raster image formats and other vector graphic formats, to achieve easier writing/reading of images, accepting that images can only be read when they are in this specific format.

Additional rationale: SVG is an open standard, which is well documented and supported by many image viewers, and image editors and libraries to read/write SVG files exist.

AD-02 SVG parsing In the context of use case UC02, facing the need to read SVG files in a maintainable way, we decided to use the svg-tree library and neglected the libraries diagrams-input and markup-parse, to achieve ease of reading SVG files, accepting that we are using a library which lacks usage examples in its documentation.

Additional rationale: svg-tree allows us to use the already implemented types for basic shapes instead of defining these ourselves.

AD-03 SVG writing In the context of use case UC01, facing the need to write SVG files in a maintainable way, we decided to use the lucid-svg library and neglected the libraries diagrams, svg-tree and markup-parse, to achieve ease of writing SVG files, accepting that we use different libraries for parsing and writing SVG files.

Additional rationale: lucid-svg is well documented. Additionally, it allows us to keep the information on the shapes used, rather than saving them as paths like the library diagrams.

AD-04 SVG generation In the context of use case UC01, facing the need to generate graphical representations of lambda terms, we decided to split the generation into different distinct steps (colour generation, calculating dimensions, calculating points, writing svg), to achieve more maintainable code for generating graphical representations.

5.2 Visual Representation Design

In the context of the visual representation design presented in chapter 4, several design decisions needed to be taken. This section documents these decisions, the reasons behind them, and the limitations of the current design.

5.2.1 Piet Mondrian inspiration

The design of Mondriλn is inspired by the Dutch painter Piet Mondrian. The language is named after him, with the λ symbol used as the letter "a" to emphasise that it is a functional language.

Piet Mondrian was one of the pioneers of 20th century abstract art [23]. Many of his paintings consist of rectangles filled with primary colours as well as white rectangles. The rectangles are adjacent to each other and always have a black outline. A selection of his paintings can be seen in figure 5.5.

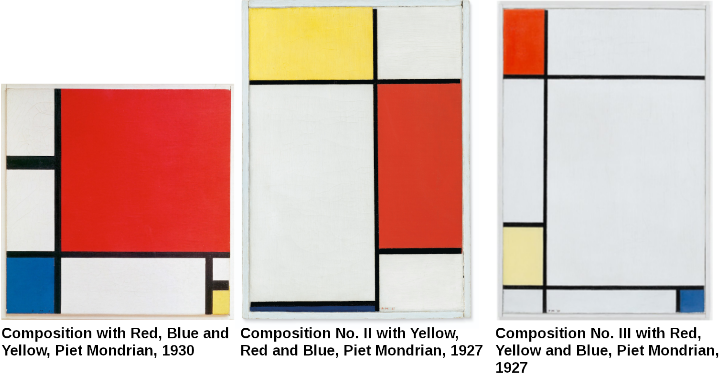


Figure 5.5: Selection of Piet Mondrian paintings [24] [25] [26]

Figure 5.6 shows an example of how the language can be used to create a representation which resembles the paintings of Piet Mondrian.

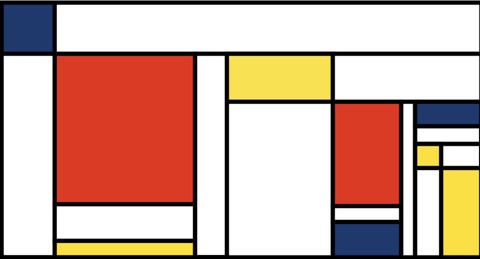


Figure 5.6: Piet Mondrian inspired representation of the term $\lambda a.b\,c\,(\lambda d.b\,a\,(a\,(\lambda c.c)))$

5.2.2 Application Separators

It is important that distinct lambda terms are visually distinguishable and unambiguous. In the initial design, applications were only visualised with adjacent rectangles. This produced the problem that some lambda terms are not visually distinguishable. Figure 5.7 shows such an ambiguous lambda term. From the image it is not clear whether term 1 or term 2 is represented.

$$\text{(red blue) (yellow green)} \qquad \text{(term 1)}$$
$$\text{(red yellow) (blue green)} \qquad \text{(term 2)}$$

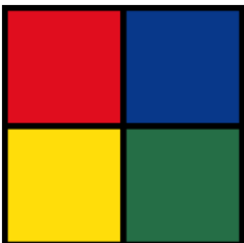


Figure 5.7: Ambiguous lambda term representation of $(a\ b)\ (c\ d)$

This issue was initially solved by adding an additional rule that 4 rectangles which all meet in the same central point are not allowed, and that in this case, their heights need to be adjusted. One possible visual representation using this rule is shown in figure 5.8. This has the advantage that this lambda term is now unambiguous. On the other hand, it has the disadvantages that a new rule needs to be respected and that this rule breaks the normal recursive generation/parsing flow (both for humans as well as for the program).

Figure 5.8: Unambiguous lambda term representation of $(a\ b)\ (c\ d)$ due to new special rule

To solve the issue without the disadvantage of breaking the normal recursive generation/parsing flow, the language now uses so called “separators” (see section 4.4.3). Figure 5.9 shows the

same term with “separators”. The addition of separators has the additional advantage that the created images resemble the paintings of Piet Mondrian more closely. On the other hand it has the slight disadvantage that users need to draw additional rectangles when creating their own images.

Figure 5.9: Unambiguous lambda term representation of $(a\ b)\ (c\ d)$ due to separator

5.2.3 Limitations in representing all terms

The visual language would in theory be able to represent all lambda terms if we assume that the amount of colours is infinite. In practice, however, the language can only represent a limited set of lambda terms due to practical limitations.

Firstly, humans can only distinguish between approximately 10 million colours [27]. Secondly, the current implementation of the visual representation uses the SVG format which only supports a limited set of colours. In SVG, colours are specified with values for red, green and blue, each in the range from 0 to 255 [28]. Using these 256 possible values per channel, a total of $256^3 \approx 16.7 * 10^6$ different colours are representable with SVG and therefore also with the current implementation. This could only be increased by using a different format which supports a greater bit depth per channel. Nevertheless, a certain limit would remain, as screens can only display colours up to a certain bit depth per channel, which, for most screens in use today, is the same amount that SVG supports.

However, the practical implications of this limitation are negligible; Understanding lambda terms of a size where this becomes relevant is a difficult task that is not meaningfully simplified by a visual representation. In order to remove the limit completely, a different way of representing different variables needs to be used, which in turn could make simple terms more difficult to understand.

5.3 Automatic colour selection

When generating visual representations from lambda terms, a mechanism to choose a colour for each variable is needed. This section describes how the mechanism works.

To choose colours, the system uses the HSV (Hue, Saturation, Value) model. The HSV model is a cylindrical representation of the RGB color model, with the parameters of hue, saturation and value/brightness uniquely identifying a coordinate. Figure 5.10 illustrates how the HSV cylinder is constructed. HSV simplifies the handling of generating colours in comparison to working directly with RGB because the relation between different coordinates and their resulting colour is more intuitive [29].

The colour generation process works in the following way:

1. The program counts the amount of distinct variables in the term.
2. It generates a random initial colour. This colour has a random hue, a random saturation within a certain range, and a fixed value/brightness.
3. Using the fixed brightness, the colour cylinder is simplified to a colour circle.

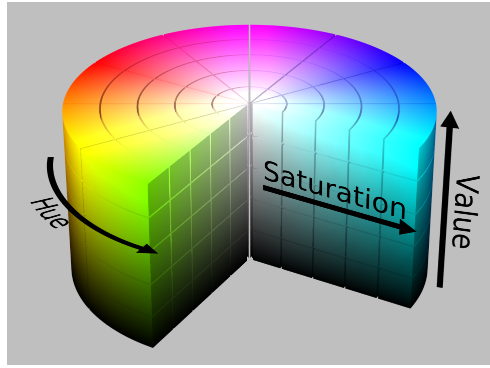


Figure 5.10: HSV cylinder [29]

4. The 360 degrees of a circle is divided by the previously calculated number of different variables to receive a step size for the hue.
5. To obtain the colours, the program “walks” around the colour circle with the previously calculated step size, while keeping the saturation constant. The coordinate of the initially generated colour is used as a starting point.
6. The variables are assigned a colour by order of appearance in the term.

A simplified illustration of this process is shown in figure 5.11.

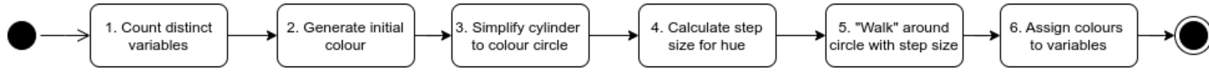


Figure 5.11: Process of choosing colours for lambda terms

This process keeps the brightness and saturation in one image the same, which results in a more visually pleasing and uniform image. In addition, the range of saturation and the brightness were constrained to increase the resemblance to the paintings of Piet Mondrian. A range is used to allow for some variability between different images.

However, keeping the saturation and brightness the same throughout an image has the disadvantage that if a large amount of colours is needed, the colours end up looking very similar due to the hue step size being very small. This disadvantage can only be resolved to a certain extent with a more complex algorithm because once a certain amount of variables is reached, the colours will always start to look quite similar. Additionally, assigning the colours by order of appearance of a variable has the disadvantage that similar colours will end up being displayed closely together in an image. To improve this, a more sophisticated algorithm for assigning colours to variables needs to be implemented which also accounts for the location of variables in an image.

Chapter 6

Testing

The functional and non-functional requirements, outlined in chapter 3, need to be verified. This chapter verifies if and how these requirements are fulfilled.

6.1 Testing methods

This section gives an overview of the different methods that were used for verifying the requirements.

6.1.1 Unit/Integration testing

Unit/Integration testing was set up for the functions that are responsible for parsing lambda terms from images, converting lambda terms to images and beta reducing lambda terms. These test all core functions for each step of the process and are automatically run in the continuous integration pipeline.

6.1.2 Property-based testing

Property-based tests using the library QuickCheck were set up to test a large number of terms. These property-based tests generate random lambda terms, generate their images and parse the created images back to lambda terms. It is then checked if the original terms and the parsed terms are alpha equivalent.

6.1.3 Manual testing scenarios

For the manual testing, a set of test scenarios were defined and are described in the following sections.

6.1.3.1 Test scenario 1: Convert lambda terms to image

To test the conversion of lambda terms to images a selection of different lambda terms are used. While transforming the terms to images, the time it takes to generate the image is tracked.

The following lambda terms are used for testing:

- x
- $\lambda x.x$
- $(\lambda x.x)y$
- $x\,y$

- $(\lambda x.x)\,(\lambda x.x)$
- $(\lambda x.x)\,(\lambda x.x)\;y$
- $a\,(b\,c)$
- $\lambda a\,a$ (example of an invalid lambda term)

6.1.3.2 Test scenario 2: Parse lambda terms from image

To test the parsing of lambda terms from images, the generated images from scenario 1 are used. While parsing the images, the time it takes to parse the image is tracked.

Additionally, an image that does not comply with the language specification is used to test the error handling. This image contains overlapping rectangles.

To test the fault tolerance of the parsing, an image that contains deviations of 0.2 units from the correct placement is used.

6.1.3.3 Test scenario 3: Beta reduce lambda terms

To test the beta-reduction process of lambda terms, the term $(\lambda x.(\lambda y.y)\,x)\,((\lambda z.z\,z)\,(\lambda z.z\,z))$ is used. During the reduction process, the time it takes to reduce the term is tracked. It is also analysed if the leftmost outermost strategy is used. Additionally, it is verified whether the user is asked if the next reduction step should be performed and whether the user is informed that the term cannot be reduced further. After the reduction, it is checked if a file is saved for each reduction step and if the file is labelled correctly. Finally, it is checked if the same colour is used for the same variable during the whole reduction process.

6.1.3.4 Test scenario 4: Understandability of the CLI

To test the understandability of the CLI, the CLI is opened and the instructions are followed by a user who has never used the tool before. The user is asked to explain the steps and to give feedback on the understandability of the CLI.

6.1.3.5 Test scenario 5: Usability

To test the usability of the language specification, a user who has no experience with our language specification is asked to read the language specification. The user is then given the images generated in test scenario 1, apart from the invalid term, and asked to explain the lambda terms.

6.2 Evaluation of test results

The following sections evaluate the test results.

6.2.1 Unit tests

The automatically executed unit tests showed that the core functions for parsing lambda terms from images, converting lambda terms to images and beta reducing lambda terms operate correctly.

The unit tests for parsing also include various cases where the image does not comply with the language specification, such as overlapping rectangles, missing rectangles, and rectangles that are not appropriately adjacent. The tests showed that the error handling for these cases is successful.

6.2.2 Property-based testing

The property-based test showed that the generation and parsing was successful for the majority of the terms. It was found that extremely large terms could not be generated correctly in some cases. This was for example the case for the following term.

$$a\ (b\ (c\ (d\ (e\ (f\ (g\ ((\lambda h.\ (\lambda i.j\ (k\ (\lambda l.\lambda m.(\lambda n.o\ (p\ (\lambda q.r\ (\lambda s.t)))) \ i\ j\ k)))\ l)\ m))))))$$
$$n\ (o\ (p\ (\lambda q.r\ (\lambda s.t\ ((\lambda u.(\lambda v.(\lambda w.\lambda x.y\ (z\ (aa\ (\lambda bb.cc\ (\lambda dd.ee)))))\ ff)\ gg)\ hh\ ii)))\ jj\ kk\ ll)))$$

This term, when looked at as a tree, has two branches that are very deep. The first deep branch sets the base dimensions. This results in a problem when the second deep branch is added. The problem is that a negative width for one of the rectangle is calculated. This is due to the static width of the application separator being subtracted from the available width. Because the available width is already very low in the second deep branch this resulted in negative dimensions. This is illustrated in figure 6.1.

We concluded that this problem could be solved by adjusting the application separator width to be dynamic, but deemed it out of scope for this project.

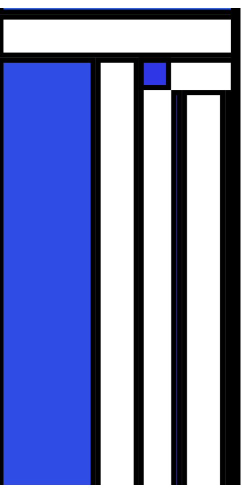


Figure 6.1: Problem in the second deep branch with the application separator

Due to this problem, we couldn't fully verify whether the parsing works correctly in all cases. However, when manually correcting this specific image to follow the language specification, the parsing was executed correctly. Additionally, the parsing worked correctly in all cases where the generation was executed successfully.

6.2.3 Manual testing scenarios

The following sections evaluate the manual testing scenarios.

6.2.3.1 Test scenario 1: Convert lambda terms to image

The test scenario showed that the conversion of lambda terms to images was successful. For all tested lambda terms, the images were generated as expected. The images were all generated in under 0.3 seconds seconds for all terms.

The generation of the term $\lambda a\, a$ failed, as expected, with the error message "The given lambda term is invalid.".

6.2.3.2 Test scenario 2: Parse lambda terms from image

The test scenario showed that the parsing of lambda terms from images was successful. For all tested images, the lambda terms were parsed as expected. The time it took to parse the images was under 0.3 seconds for all images.

Also the image with deviations of 0.2 units from the correct placement was parsed correctly and in under 0.3 seconds.

The parsing of the image with overlapping rectangles failed, as expected, with the error message "Parsing the file failed because there were rectangles found that overlap with other rectangles.".

6.2.3.3 Test scenario 3: Beta reduce lambda terms

The test scenario showed that the beta-reduction process of lambda terms was successful. The term $(\lambda x.(\lambda y.y)\,x)\,((\lambda z.z)(\lambda z.z))$ was reduced to $\lambda z.z$ as expected. It could be confirmed that the leftmost outermost strategy was used, as can be seen in the following reduction steps:

$$\begin{aligned} (\lambda x.(\lambda y.y)\,x)\,((\lambda z.z)\,(\lambda z.z)) && \because \beta \\ (\lambda y.y)\,((\lambda z.z)\,(\lambda z.z)) && \because \beta \\ (\lambda z.z)\,(\lambda z.z) && \because \beta \\ \lambda z.z && \end{aligned}$$

After each reduction step, the user was asked if the next reduction step should be performed. When no further reduction was possible, the user was informed that the term cannot be reduced further. Finally, it was confirmed that a file was saved for each reduction step and that the file was labelled correctly. The same colour was used for the same variable during the whole reduction process.

The time it took to reduce the term was under 1 second.

6.2.3.4 Test scenario 4: Understandability of the CLI

The test subject was able to understand the functionality of the CLI and explain what steps needed to be taken. The user was able to generate images from lambda terms, parse lambda terms from images, and reduce lambda terms.

6.2.3.5 Test scenario 5: Usability

The user was able to successfully identify six out of seven lambda terms. The user was not able to identify the term $a\,(b\,c)$, as the user assumed that the term was $a\,b\,c$.

6.3 Fulfilment of requirements

This section described how well the functional and non-functional requirements are fulfilled. Refer to section 3.3 for the requirements. The requirements are referred to by their ID.

6.3.1 Functionality

This section describes how well the functional requirements are fulfilled.

6.3.1.1 UC01: Convert lambda term to image

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-F-100	1	Scenario 1	-	yes
RQ-F-110	1	Scenario 1	-	yes
RQ-F-120	1	Scenario 1	-	yes
RQ-F-125	1	Scenario 1	-	yes
RQ-F-130	1	Scenario 1	-	yes
RQ-F-140	1	Scenario 1	-	yes
RQ-F-150	2	Property-based test	see section 6.2.2	mostly
RQ-F-155	2	Scenario 1	-	yes
RQ-F-160	1	Scenario 1	-	yes

Table 6.1: Test results for functional requirements concerning UC01

6.3.1.2 UC02: Parse lambda term from image

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-F-200	1	Scenario 2 and integration tests	-	yes
RQ-F-210	1	Scenario 2 and integration tests	-	yes
RQ-F-220	1	Scenario 2 and integration tests	-	yes
RQ-F-225	1	Scenario 2 and integration tests	-	yes
RQ-F-230	1	Scenario 2 and integration tests	-	yes
RQ-F-240	1	Scenario 2 and integration tests	-	yes
RQ-F-250	2	Property-based test	Due to failure in generation, see section 6.2.2	unknown
RQ-F-260	1	Scenario 2 and unit tests	-	yes
RQ-F-270	3	Scenario 2	-	yes

Table 6.2: Test results for functional requirements concerning UC02

6.3.1.3 UC03: Beta reduce lambda term

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-F-300	2	Scenario 3	-	yes
RQ-F-310	2	Scenario 3	-	yes
RQ-F-320	2	Scenario 3	-	yes
RQ-F-325	2	Scenario 3	-	yes
RQ-F-330	3	-	Due to time restraints this feature was not implemented	no
RQ-F-335	3	Scenario 3	-	yes
RQ-F-340	3	-	Due to time restraints this feature was not implemented	no
RQ-F-350	3	-	Due to time restraints this feature was not implemented	no

Table 6.3: Test results for functional requirements concerning UC03

6.3.1.4 UC04: Configure design of converted image

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-F-400	3	-	Due to time restraints this feature was not implemented	no
RQ-F-410	3	-	Due to time restraints this feature was not implemented	no

Table 6.4: Test results for functional requirements concering UC04

6.3.1.5 UC05: Use pre-defined definitions for input of lambda terms

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-F-500	3	-	Due to time restraints this feature was not implemented	no

Table 6.5: Test results for functional requirements concerning UC05

6.3.2 Usability

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-U-100	1	Scenario 5	-	yes
RQ-U-200	3	Scenario 4	The CLI shows which commands can be executed. Apart from this information, there is no documentation directly in the app. However, there is a documentation in the GitLab repository.	yes

Table 6.6: Test results for usability requirements

6.3.3 Reliability

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-R-100	2	Scenario 2 and unit tests	-	yes

Table 6.7: Test results for reliability requirements

6.3.4 Performance

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-P-100	1	Scenario 1	All conducted manual tests showed that the generation could be performed in under 8 seconds.	yes
RQ-P-200	1	Scenario 2	All conducted manual tests showed that the parsing could be performed in under 10 seconds.	yes
RQ-P-300	2	Scenario 3	The test scenario showed that the reduction could be performed in under 20 seconds.	yes

Table 6.8: Test results for performance requirements

6.3.5 Supportability

ID	Priority	Proven by test case	Remarks	Fulfilled
RQ-S-100	1	Verified by checking all components	-	yes
RQ-S-200	2	Scenario 4	-	yes
RQ-S-300	3	-	Due to time restraints this feature was not implemented	no

Table 6.9: Test results for supportability requirements

Chapter 7

Evaluation

This chapter explores the appearance of certain terms. The terms discussed are noteworthy for two main reasons: they either illustrate endlessly beta-reducible expressions or represent standard functions such as logical negation (not), booleans, addition, and Church numerals. Additionally, the chapter aims to provide a visual connection between different lambda calculus expressions and their corresponding beta reductions.

7.1 Church Numerals

Alonzo Church introduced Church numerals as a way to represent natural numbers in the lambda calculus [20]. The Church numeral n is represented as a function that takes two arguments, a function f and an argument x, and applies the function f n times to the argument x. The Church numerals 0, 1, 2 and 3 are represented in figure 7.1. The dusty rose coloured rectangles represent the function f. The visual representation clearly shows that for each Church numeral, the function f is applied n times to the argument x.

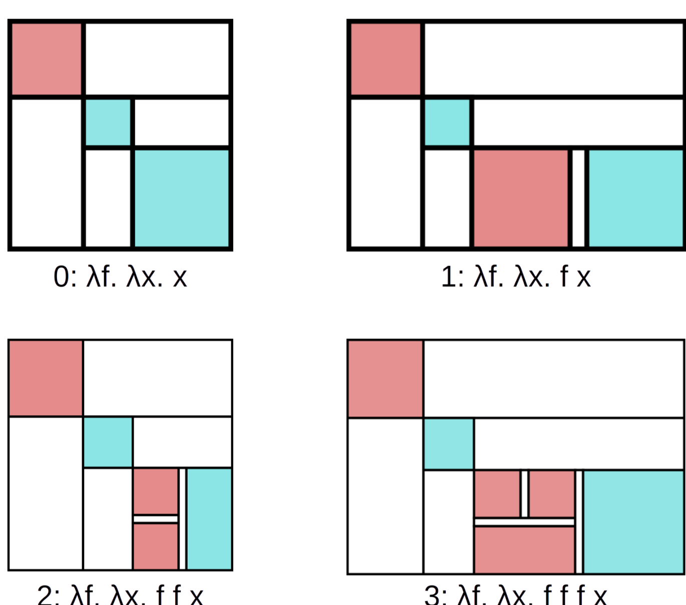


Figure 7.1: Visual representation of the Church Numerals 0, 1, 2, 3

7.2 succ 0

The successor function (succ) in the lambda calculus operates on a Church numeral n and produces the succeeding Church numeral $n+1$. The succ function is represented in figure 7.2.

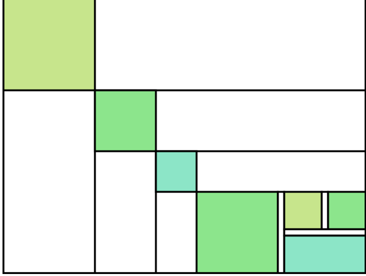


Figure 7.2: Visual representation of the succ function: $\lambda n.\lambda f.\lambda x.f(n\, f\, x)$

Figure 7.3 shows how the successor function is applied to the Church numeral 0 ($\lambda f.\lambda x.x$) and how it is beta-reduced. This results in the Church numeral 1 (see figure 7.1).

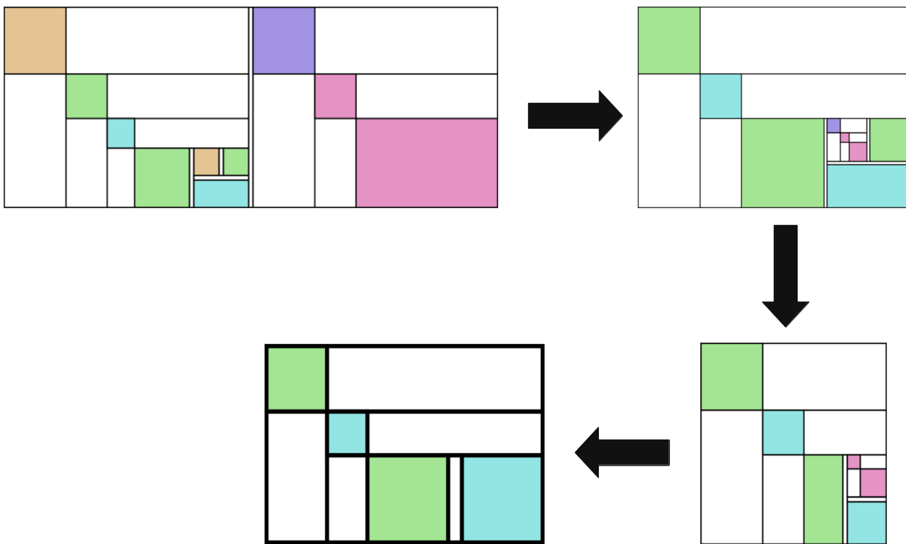


Figure 7.3: Beta reduction of ‘succ 0’ to 1

7.3 Addition

The addition function in the lambda calculus takes two Church numerals m and n and produces the Church numeral $m+n$. Addition in the lambda calculus uses the Polish notation, where the operator is placed before the operands. Note that the successor function is used to represent the addition function. The successor function is applied n times to the Church numeral m and is represented with the same colours as in figure 7.2. This is illustrated in figure 7.4.

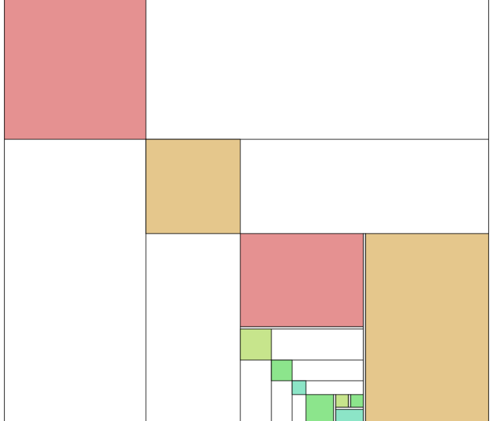


Figure 7.4: Visual representation of the addition function: $\lambda m.\lambda n.m(\lambda n.\lambda f.\lambda x.f(nfx))n$

Figure 7.5 shows the addition of the Church numerals 1 and 1. It can clearly be seen that the addition function is applied to the Church numerals 1 and 1.

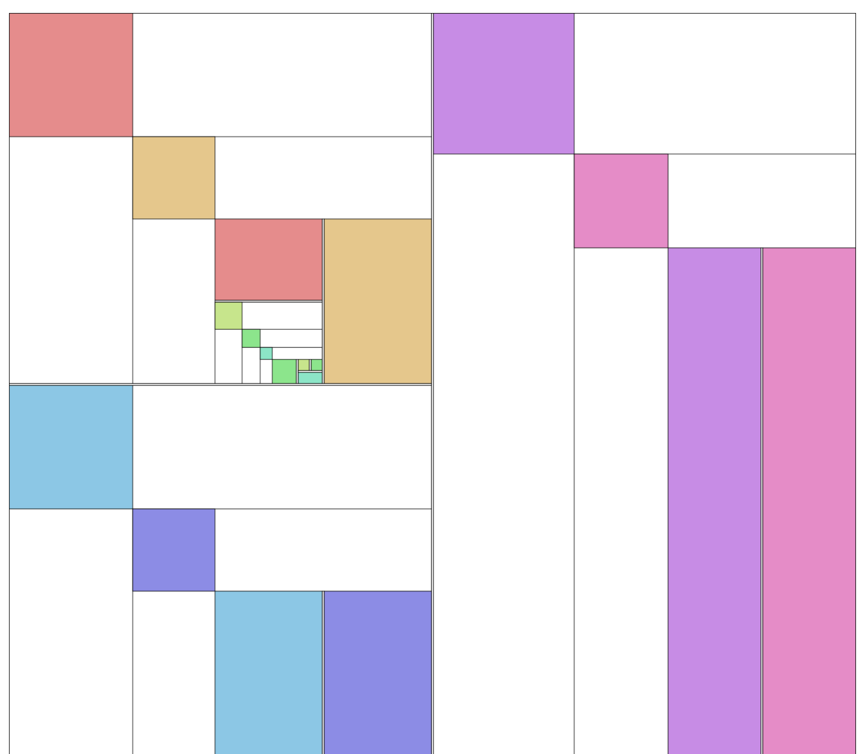


Figure 7.5: Visual representation of $+\,1\,1$

The beta reduction process of the addition of the Church numerals 1 and 1 is shown in figure 7.6, resulting in the Church numeral 2 (see figure 7.1).

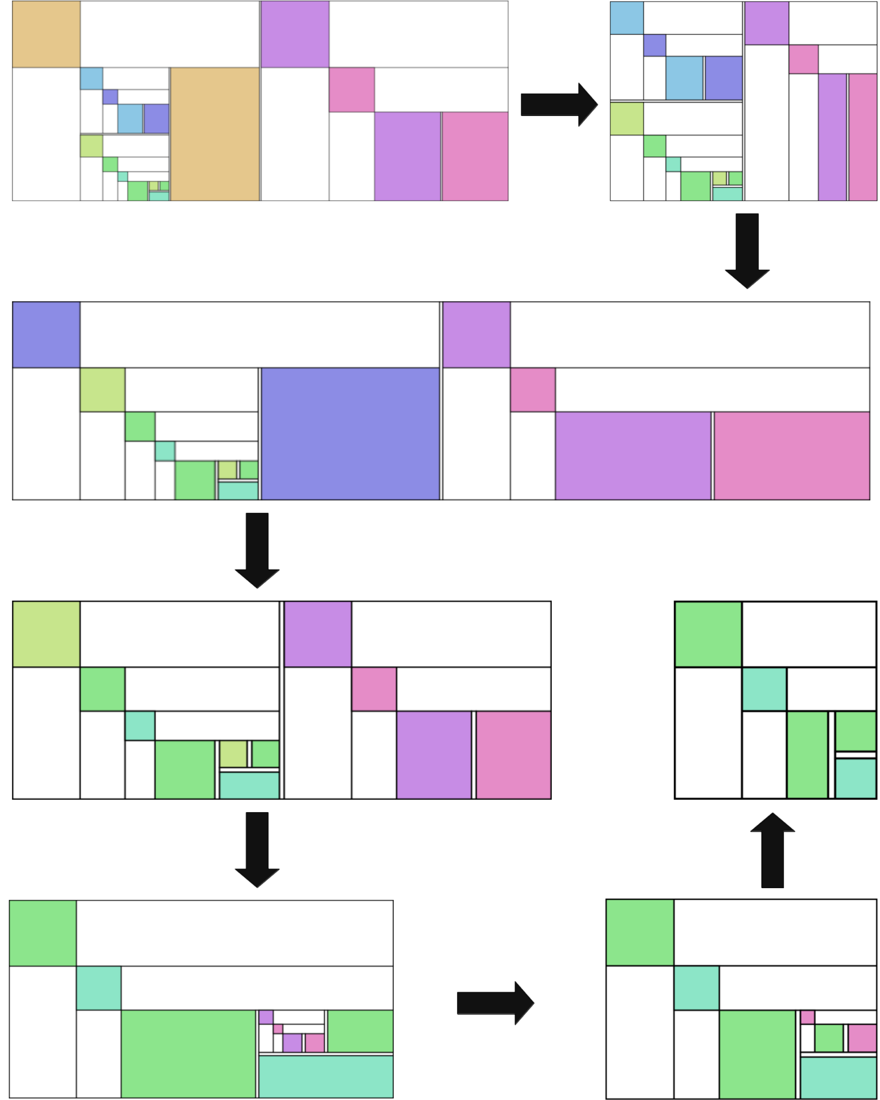


Figure 7.6: Beta reduction of $+\,1\,1$ to 2

7.4 Endless Loop

The lambda calculus allows for the creation of endless loops. Figure 7.7 shows the beta reduction of the expression $(\lambda x.x\,x)(\lambda x.x\,x)$, which results in an endless loop. It is apparent that the term continues to expand and never reaches a normal form.

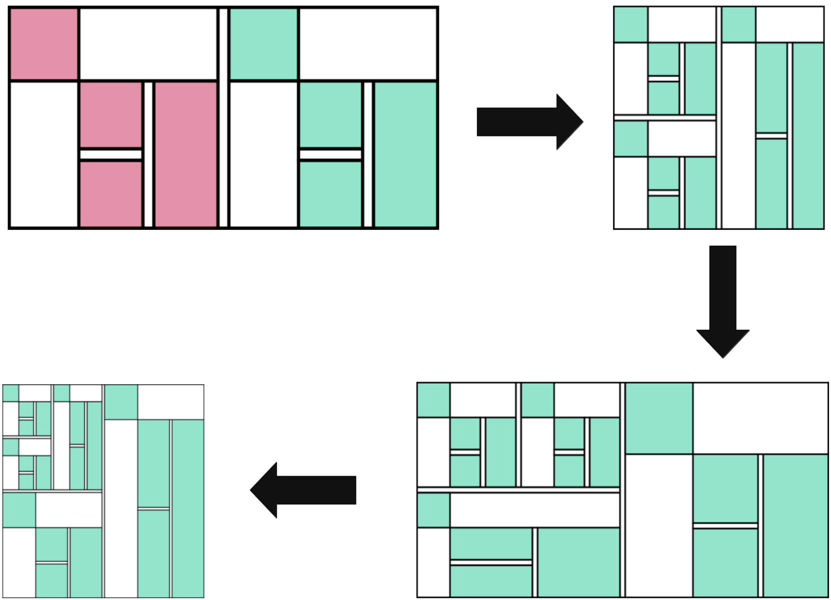


Figure 7.7: Beta reduction of $(\lambda x.x\,x)(\lambda x.x\,x)$, an endless loop

7.5 Y-Combinator

The Y-Combinator $\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$ is a fixed-point combinator in the lambda calculus. It allows for the creation of recursive functions without a function appearing in its own definition. The latter is known as "recursion" in other programming languages but is not allowed in lambda calculus [30]. Beta-reducing the Y-Combinator creates an endless loop as well. Figure 7.8 shows the beta reduction of the Y-Combinator.

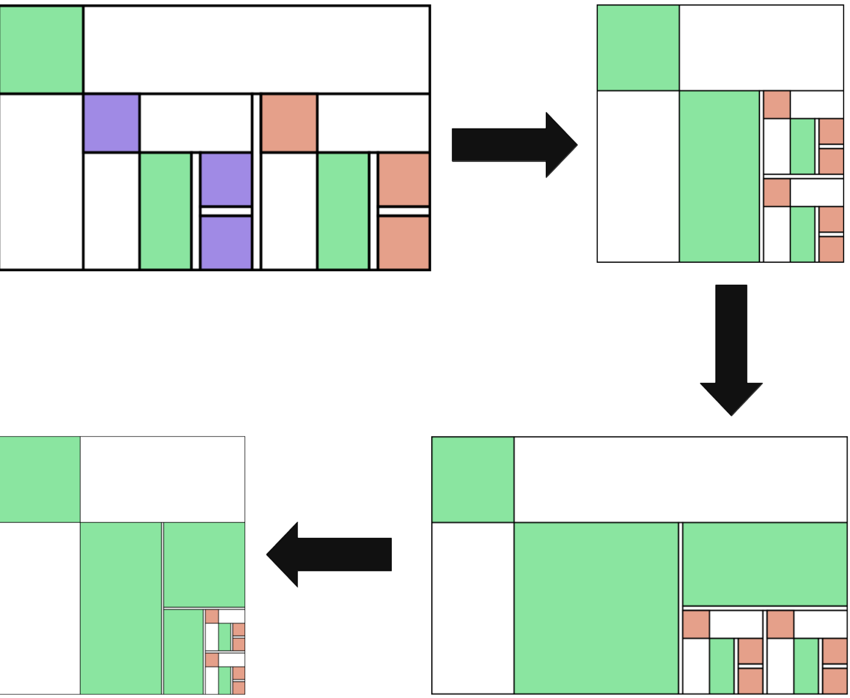


Figure 7.8: Beta reduction of the Y-Combinator

7.6 Not True

The logical negation function *not* in the lambda calculus is represented as $\lambda b.b\ False\ True$. In figure 7.9, the booleans $True$ and $False$ are represented.

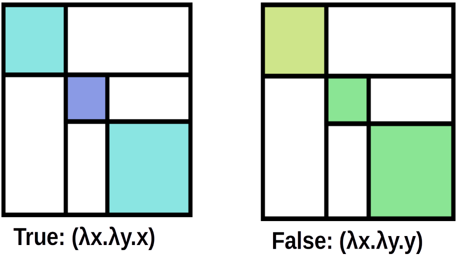


Figure 7.9: Representation of the booleans $True$ and $False$

This can be used to evaluate the expression *not* $True$. The beta reduction of *not* $True$ is shown in figure 7.10. As expected, the expression evaluates to $False$.

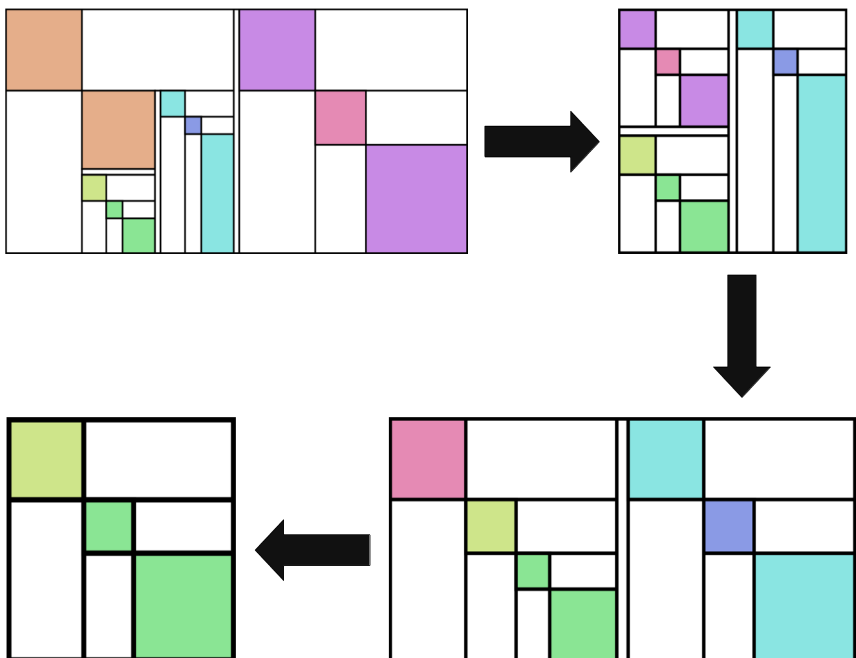


Figure 7.10: Beta reduction of *not* $True$ to $False$

Chapter 8

Conclusion and Outlook

This project demonstrates the successful visualisation of lambda terms as images and is a first step towards making lambda calculus more intuitive to learn for students. The language is capable of representing all lambda terms up to the limit of how many colours can be represented by a screen. Additionally, the language appears to make lambda terms simpler to understand, particularly when it comes to the beta reduction process. However, while the language shows promise, its potential usefulness to students remains an open question. Further evaluation is needed to determine how effectively it can aid students' understanding of lambda calculus, with particular attention given to the beta reduction process and its clarity.

The proof of concept application is implemented as a command line tool and supports generating images from lambda terms, determining the lambda term contained within an existing image and performing beta reduction on them. This accomplishment aligns with the project's objectives and requirements, as outlined in the previous chapters. To improve the usability of the system, developing a web-based interface would be a valuable next step. It could also offer additional features, such as interactive animations to visualise the beta reduction process, direct manipulation of elements, and options for users to choose and customise colours for a more personalised experience.

Despite the achievements, a few unresolved issues remain. Handling large lambda terms currently presents a challenge under certain circumstances as described in section 6.2.2. Additionally, alpha renaming has not yet been implemented due to time constraints.

Overall, this project provides a good starting point for further development. By addressing the aforementioned limitations and opportunities for enhancement, the system can become even more usable and effective.

Part II

Project Documentation

Chapter 9

Project Management

9.1 Methodology

For this project, we decided to use a combination of RUP and agile elements. We decided against using SCRUM because it would be too much overhead and too little benefit for our small team of two. Instead, we incorporate the following agile elements:

Iterative development We will iteratively expand the functionality of the lambda term converter. We will start with the conversion of simple lambda terms to images and back, and then expand the functionality to include beta reduction and possible additional features.

In-person working In-person working will take place at least once per week. Having compared our schedules, we decided to regularly meet on Tuesday afternoon. On this day we will discuss the progress of the project and plan the next steps, as well as work together on hard problems.

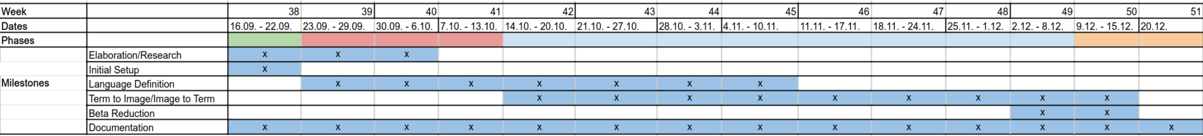
Active communication As individual work on the project is unavoidable, we will keep each other informed about the progress of tasks and problems that arise. For communication, we use the tools described in section 9.3.

Research phase We plan on having a short research phase at the beginning of the project to get a better understanding of the lambda calculus and its potential visual representation.

Meetings with advisor We plan to have regular meetings with our advisor to discuss the progress of the project and get feedback. These meetings will take place weekly or biweekly on Tuesday afternoons, or, in case of unavailability, on Fridays.

9.2 Project Plan

The project plan is structured in milestones, as can be seen in figure 9.1. The figure shows that the project is divided into the four phases of RUP: Inception (green), Elaboration (red), Construction (blue), and Transition (orange).



Week		38	39	40	41	42	43	44	45	46	47	48	49	50	51
Dates		16.09. - 22.09.	23.09. - 29.09.	30.09. - 6.10.	7.10. - 13.10.	14.10. - 20.10.	21.10. - 27.10.	28.10. - 3.11.	4.11. - 10.11.	11.11. - 17.11.	18.11. - 24.11.	25.11. - 1.12.	2.12. - 8.12.	9.12. - 15.12.	20.12.
Phases															
	Elaboration/Research	x	x	x											
	Initial Setup	x													
Milestones	Language Definition		x	x	x	x	x	x	x						
	Term to Image/Image to Term					x	x	x	x	x	x	x	x	x	
	Beta Reduction												x	x	
	Documentation	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 9.1: Project plan with milestones

The milestones are based on the RUP methodology and include the following elements:

Initial Setup For this milestone, we set up the project repositories and the tools that we will use. The first milestone is planned to be completed in the first week of the project.

Research and Language Definition The second milestone is the end of the research and language definition phase. We will research the lambda calculus and formally define a visual representation that is able to represent as many aspects of the lambda calculus as possible.

Term to Image/Image to Term The third milestone features the implementation of a 'lambda-term-to-image' and a 'image-to-lambda-term' function. By the end of this phase, we should have a minimum viable product that can convert lambda terms to images and back.

Beta reduction The fourth milestone is the implementation of beta reduction. We reserved two weeks for the implementation of beta reduction.

Documentation Throughout the project, we continuously update the documentation to ensure that it is always up to date.

9.3 Tools

This section describes the tools that we use for this project.

9.3.1 Project Management

Teams We use Teams as a central platform for communication within our project team and with our advisor. Additionally, in case we need to hold remote meetings, we use Teams for video calls.

GitLab issues For tracking the tasks that need to be worked on to further our project progress, we use GitLab issues.

Clockify For keeping track of the time spent on our project we use Clockify. This helps to ensure that we adhere to the guideline of spending ≈ 17 hours per week on our project. At the end of the project, we can create statistics on how much time we spent on each task and how much each team member contributed.

9.3.2 Software Development

GitLab Our code and documentation is stored in GitLab repositories. Furthermore, the CI pipelines for our project are also managed and executed in GitLab. For more details see section 9.4.1.

Inkscape To create example images for the language design document we use the vector graphics editor Inkscape.

Visual Studio Code For writing code we use the code editor Visual Studio Code (VS Code). In our experience, VS Code provides the best environment for writing Haskell code.

GitHub Copilot & Microsoft Copilot For more cost-efficient and fast development we use GitHub Copilot and Microsoft Copilot. These tools are used in accordance with the AI guidelines of the OST.

9.3.3 Documentation

LaTeX We use LaTeX to write all of our project documentation and the technical report.

9.4 Quality Measures

To help us achieve a high standard of quality, we put the following quality measures in place.

9.4.1 Organisational Means

Merge Requests

Every change must go through a merge request. This is an enforced rule, as no team member is allowed to commit directly to the main branch. All merge requests are reviewed by another team member, who gives feedback and may approve the merge request.

The person who opened the merge request can merge it only after the following conditions are fulfilled:

- The merge request must be approved by another person. As this rule can not be enforced in the OST GitLab, we trust each other to respect this rule.
- The CI pipeline that builds the project, checks the formatting (see section 9.4.2 for more details), and executes all automated tests must pass for each merge request. This is enforced by our GitLab settings.

Definition of Done

For a task to be considered complete and to close the issue, the following requirements must be met:

- The change is merged into the main branch.
- The documentation is up to date.
- If possible, automated tests have been written.

Boy/Girl Scout rule

The team commits to using the boy/girl scout rule for code and documentation: "Always leave the campground cleaner than you found it."

9.4.2 Code Quality

Clean Code

Code should be written in a way which is clear and understandable. By using clear naming and a consistent coding style, we ensure that our code is maintainable and extendable.

Formatting

To ensure that our code is formatted consistently, we use the automatic formatting tool Ormolu. On each merge request, the build automatically checks whether the code is formatted correctly.

Testing

All of our code should be tested, either automatically or manually. Automatic tests are preferred, as these can be executed with little overhead on every code change, and verification happens on each merge request. If automatic testing is not possible, we use manual testing.

9.4.3 Documentation Quality

To ensure high quality in our documentation, we put the following rules in place:

- The build for the documentation must run without errors.
- Each documentation change is spell checked with an automatic spell checker.
- All new figures, code listings, and tables have captions and an attached LaTeX label

9.5 Risk Management

This section describes the risks we identified for our project and how we plan to mitigate them.

9.5.1 Risks and their impact

We identified the following risks that could impact our project:

1. Project member falls ill If this risk occurs, the team's available capacity will be reduced. Moreover, if the affected team member is working on critical tasks and/or tasks that other important work depends on, it may cause delays in the implementation. This would impact the project timeline and the ability to meet the full scope of the project. Additionally, the team member who falls ill will face the challenge of making up for the 'lost' hours to reach the planned hours of work. This could also lead to a loss of motivation due to a higher workload.

2. Lambda calculus is not fully representable in the chosen format The chosen format may not fully support the representation of lambda calculus, which could restrict the ability to express certain functions or concepts. This would impact the ability to implement the project as planned. It may also lead to additional work to find workarounds or alternative solutions, potentially delaying the project timeline. In the worst case, the project could fail if we find that the chosen format cannot represent lambda calculus to a sufficient degree.

3. Parsing takes longer than planned We may underestimate the complexity of the parsing process, as we have no experience with image parsing. If the parsing takes longer than planned, the project timeline could be affected.

4. Scope cannot be fulfilled If the scope proves to be unrealistic within the given constraints, the project could fail or the deliverables would be incomplete.

5. Time-consuming use of third-party libraries There is a risk that integrating third-party libraries may take longer than expected due to the team's limited knowledge of these libraries. This lack of familiarity could lead to additional time spent on learning, troubleshooting, and resolving unforeseen issues, potentially delaying project progress.

6. Requirements are unclear Unclear requirements could lead to significant additional work as in the worst case the feature needs to be implemented again. This would affect the project timeline and may lead to the scope not being able to get fulfilled.

9.5.2 Risk Assessment

To assess the risks we identified, we evaluated the likelihood of the risk occurring and the impact it would have on the project, as displayed in table 9.1.

Risk	Likelihood	Impact
1. Project member falls ill	High	Medium
2. Lambda calculus is not fully representable in the chosen format	Medium	Very High
3. Parsing takes longer than planned	Medium	High
4. Scope cannot be fulfilled	Low	Medium
5. Time-consuming use of third-party libraries	Medium	Medium
6. Requirements are unclear	Low	Medium

Table 9.1: Risk Assessment

9.5.2.1 Risk Mitigation

To mitigate the risks we identified, we put the following measures in place:

1. Project member falls ill To mitigate this risk, the ill team member will notify the team as early as possible if a task cannot be completed on time and requires reassignment. We will ensure that all team members are kept informed of ongoing tasks and are prepared to take over responsibilities from one another. By highlighting this risk, we also aim to promote awareness of maintaining a healthy work-life balance within our team.

2. Lambda calculus is not fully representable in the chosen format We will do research on lambda calculus before defining the visual representation and evaluate possible pitfalls. Because exploring the feasibility of the visual representation is a core objective of our project, we will define the visual representation as early as possible. We aim to implement it as completely as possible, though we may need to settle for a subset of lambda calculus or omit certain features.

3. Parsing takes longer than planned To counteract this risk, we will begin the parsing as early as possible. We will also conduct preliminary research into image formats and parsing libraries to select the most suitable option for our project – prioritising one that is stable, straightforward, and user-friendly.

4. Scope cannot be fulfilled By defining our functional and non-functional requirements early in the project, we ensure that the scope is clear from the start. As a team, we define the priority of the individual work items, taking into account the importance of each feature for delivering the project scope. Through our iterative development approach we can regularly review and adjust our plan based on our progress and any possible challenges.

5. Time-consuming use of third-party libraries To mitigate this risk, we will only use third-party libraries that are well-supported and aim to use as few as possible. We will also conduct initial research on libraries we intend to use to ensure they are well-documented and user-friendly.

6. Requirements are unclear We define our functional and non-functional requirements early in the project in a way that is clear to everyone on the team. We ensure this by reviewing the defined requirements together. If any team member is uncertain about a requirement, they will ask for clarification using the appropriate communication channels.

Bibliography

[1] Wikipedia contributors, *Esoteric programming language*, en, Page Version ID: 1260788576, December 2024. [Online]. Available: https://en.wikipedia.org/wiki/Esoteric_programming_language (visited on 14th December 2024).

[2] D. Madore, *The Unlambda Programming Language*. [Online]. Available: http://www.madore.org/~david/programs/unlambda/ (visited on 12th October 2024).

[3] M. J. Sullivan, *The Lazy K Programming Language*. [Online]. Available: https://tromp.github.io/cl/lazy-k.html (visited on 12th October 2024).

[4] H. Rostedt, *Cool*, en. [Online]. Available: https://gitlab.com/fogity/squared-cool (visited on 12th October 2024).

[5] L. Ghignone, *Husk Wiki*, en. [Online]. Available: https://github.com/barbuz/Husk/wiki/Home (visited on 12th October 2024).

[6] *SKM Calculus*. [Online]. Available: https://esolangs.org/wiki/SKM_Calculus (visited on 12th October 2024).

[7] *Jot*. [Online]. Available: https://esolangs.org/wiki/Jot (visited on 12th October 2024).

[8] *Brainfunct*. [Online]. Available: https://esolangs.org/wiki/Brainfunct (visited on 12th October 2024).

[9] D. Morgan-Mar, *Piet*. [Online]. Available: https://www.dangermouse.net/esoteric/piet.html (visited on 12th October 2024).

[10] J. Van der Jeugt, *Turnstyle*, en. [Online]. Available: https://jaspervdj.be/turnstyle/spec/ (visited on 12th October 2024).

[11] C. O'Brien, *Pyramid Scheme*. [Online]. Available: https://github.com/ConorOBrien-Foxx/Pyramid-Scheme (visited on 12th October 2024).

[12] *Rail*. [Online]. Available: https://esolangs.org/wiki/Rail (visited on 12th October 2024).

[13] M. White, *Skastic*. [Online]. Available: https://github.com/mypalmike/skastic (visited on 12th October 2024).

[14] W. Citrin, R. Hall and B. Zorn, "Programming with visual expressions," in *Proceedings of Symposium on Visual Languages*, ISSN: 1049-2615, Sep. 1995, pp. 294–301. DOI: 10.1109/VL.1995.520822. [Online]. Available: https://ieeexplore.ieee.org/document/520822 (visited on 12th October 2024).

[15] V. Massalõgin, "Visual lambda calculus," *Master's thesis, Rijksuniversiteit Groningen, Estonia*, 2008.

[16] B. VICTOR, *Alligator Eggs!* [Online]. Available: https://worrydream.com/AlligatorEggs/ (visited on 12th October 2024).

[17] D. C. Keenan, *To Dissect a Mockingbird: A graphical notation for the lambda calculus with animated reduction*, April 2014. [Online]. Available: https://dkeenan.com/Lambda/ (visited on 12th October 2024).

[18] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004, ISBN: 0131489062.

[19] R. B. Grady and D. L. Caswell, *Software metrics: establishing a company-wide program.* Englewood Cliffs, N.J: Prentice-Hall, 1987, ISBN: 9780138218447.

[20] Wikipedia contributors, *Lambda calculus*, en, Page Version ID: 1249612766. [Online]. Available: `https://en.wikipedia.org/wiki/Lambda_calculus` (visited on 7th October 2024).

[21] S. Brown, *C4 model*, en-US. [Online]. Available: `https://c4model.com/` (visited on 10th December 2024).

[22] O. Zimmermann, *Architectural Decisions — The Making Of*, en, 2021. [Online]. Available: `https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html` (visited on 8th October 2024).

[23] Wikipedia contributors, *Piet Mondrian*, en, November 2024. [Online]. Available: `https://en.wikipedia.org/wiki/Piet_Mondrian` (visited on 17th December 2024).

[24] Wikipedia contributors, *Composition with Red, Blue, and Yellow.* [Online]. Available: `https://commons.wikimedia.org/wiki/File:Piet_Mondriaan,_1930_-_Mondrian_Composition_II_in_Red,_Blue,_and_Yellow.jpg` (visited on 17th December 2024).

[25] Wikipedia contributors, *Composition: No. II, With Yellow, Red and Blue.* [Online]. Available: `https://commons.wikimedia.org/wiki/File:Composition_no._III_with_red,_yellow,_and_blue_by_Piet_Mondrian.jpg` (visited on 17th December 2024).

[26] Wikipedia contributors, *Composition no. III with red, yellow, and blue.* [Online]. Available: `https://commons.wikimedia.org/wiki/File:Composition_no._III_with_red,_yellow,_and_blue_by_Piet_Mondrian.jpg` (visited on 17th December 2024).

[27] Wikipedia contributors, *Color*, en, 2024. [Online]. Available: `https://en.wikipedia.org/wiki/Color` (visited on 14th December 2024).

[28] *SVG specification, "Color"*, publisher: World Wide Web Consortium. [Online]. Available: `https://www.w3.org/TR/SVG11/color.html` (visited on 14th December 2024).

[29] Wikipedia contributors, *HSL and HSV*, en, November 2024. [Online]. Available: `https://en.wikipedia.org/wiki/HSL_and_HSV` (visited on 17th December 2024).

[30] Wikipedia contributors, *Fixed-point combinator*, en, Page Version ID: 1259282932, November 2024. [Online]. Available: `https://en.wikipedia.org/wiki/Fixed-point_combinator` (visited on 14th December 2024).

List of Figures

1 Piet Mondrian inspired representation of the term $\lambda a.b\, c\, (\lambda d.b\, a\, (a\, (\lambda c.c)))$ iii
2 Beta reduction process of the term $(\lambda a.a\, a)\, (\lambda b.b)\, c$ iii

2.1 VEX Representation of the Y-Combinator 8
2.2 Bubble notation of the Y-Combinator 8
2.3 "Alligator Eggs!" representation of the Y-Combinator 9
2.4 "To Dissect a Mockingbird" representation of the Y-Combinator 9

3.1 Use Case Diagram 11

4.1 Representation of the term: x 17
4.2 Representation of the term: $\lambda x.x$ 17
4.3 Representation of the term: $\lambda x.\lambda y.x$ 18
4.4 Representation of the term: $\lambda x.\lambda y.x\, y$ 18
4.5 Representation of the term: $x\, y$ 18
4.6 Representation of the term: $(x\, y)\, z$ 19
4.7 Representation of the term: $x(y\, z)$ 19
4.8 Representation of the term: $((a\, b)\, c)\, d$ 19
4.9 Representation of the term: $(a\, b)\, (c\, d)$ 19
4.10 Representation of the term: $\lambda x.x\, (\lambda y.x\, y)$ 19
4.11 Representation of the term: $(\lambda x.\lambda y.x)\, x$ 19

5.1 System Context Diagram 22
5.2 Container Diagram 23
5.3 Component Diagram of Core 24
5.4 Process of generating images from lambda terms 24
5.5 Selection of Piet Mondrian paintings 25
5.6 Piet Mondrian inspired representation of the term $\lambda a.b\, c\, (\lambda d.b\, a\, (a\, (\lambda c.c)))$ 26
5.7 Ambiguous lambda term representation of $(a\ b)\ (c\ d)$ 26
5.8 Unambiguous lambda term representation of $(a\ b)\ (c\ d)$ due to new special rule . 26
5.9 Unambiguous lambda term representation of $(a\ b)\ (c\ d)$ due to separator 27
5.10 HSV cylinder 28
5.11 Process of choosing colours for lambda terms 28

6.1 Problem in the second deep branch with the application separator 31

7.1 Visual representation of the Church Numerals 0, 1, 2, 3 . . . 36
7.2 Visual representation of the succ function: $\lambda n.\lambda f.\lambda x.f(n\, f\, x)$ . . . 37
7.3 Beta reduction of 'succ 0' to 1 . . . 37
7.4 Visual representation of the addition function: $\lambda m.\lambda n.m(\lambda n.\lambda f.\lambda x.f(nfx))n$ . . . 38
7.5 Visual representation of $+\,1\,1$ . . . 38
7.6 Beta reduction of $+\,1\,1$ to 2 . . . 39
7.7 Beta reduction of $(\lambda x.x\,x)(\lambda x.x\,x)$, an endless loop . . . 40
7.8 Beta reduction of the Y-Combinator . . . 40
7.9 Representation of the booleans $True$ and $False$ . . . 41
7.10 Beta reduction of *not* $True$ to $False$ . . . 41

9.1 Project plan with milestones . . . 44

List of Tables

3.1 Functional requirements for UC01 13
3.2 Functional requirements for UC02 13
3.3 Functional requirements for UC03 14
3.4 Functional requirements for UC04 14
3.5 Functional requirements for UC05 14
3.6 Usability requirements 15
3.7 Reliability requirements 15
3.8 Performance requirements 15
3.9 Supportability requirements 15

6.1 Test results for functional requirements concerning UC01 33
6.2 Test results for functional requirements concerning UC02 33
6.3 Test results for functional requirements concerning UC03 33
6.4 Test results for functional requirements concering UC04 34
6.5 Test results for functional requirements concerning UC05 34
6.6 Test results for usability requirements 34
6.7 Test results for reliability requirements 34
6.8 Test results for performance requirements 35
6.9 Test results for supportability requirements 35

9.1 Risk Assessment 48

Listings

2.1 Pyramid scheme program that does a conditional check and prints 1 if A is true, or 0 if A is false . . . 6

2.2 Rail program that reads from standard input and prints to standard output . . . 7

Part III

Appendix

A README document

MondriΛn - A Visual Programming Language Based on the Lambda Calculus

Create lambda terms that look like art!

Inspired by the iconic paintings of Piet Mondrian, we present MondriΛn (yes, we're clever with that lambda in the name 😉).

This CLI tool allows you to:

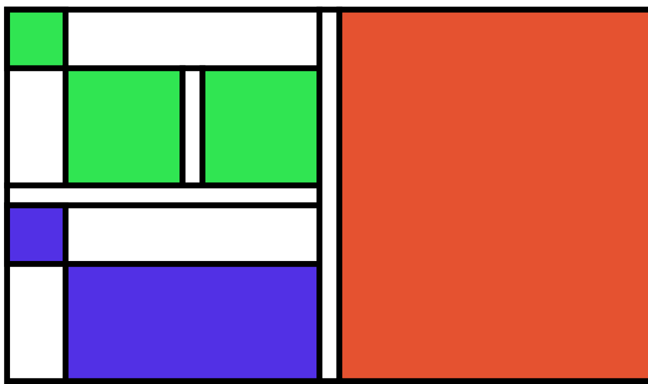
- **Transform your lambda terms into beautiful images**
- **Beta reduce lambda terms** to gain a clearer visual understanding of beta reduction
- **Parse lambda terms from images** — use pictures created by MondriΛn or draw your own (SVG format is required; we recommend using Inkscape, refer to additional resources for more details).

Getting Started

- **Run the application locally** (see instructions below).
- **Enter a command** in the CLI interface.

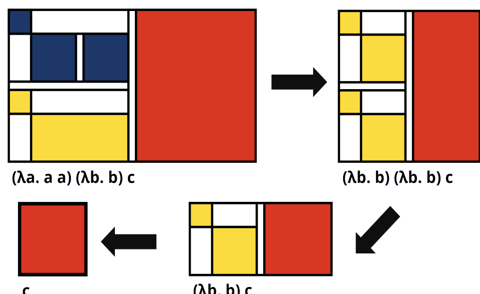
```
*** Commands ***
generate <term>                 Convert a lambda term to an SVG.
generate beta-reduced <term>    Beta-reduce a lambda term and convert to SVGs.
parse <file-path>               Parse an SVG located at the given file path and convert to a lambda term.
parse beta-reduced <file-path>  Parse an SVG located at the given file path to a beta-reduced lambda term.

Enter a command:
generate (\x. x x) (\y. y) z
```

Which can then result in the following visual representation:



Features

1. **Generate Pictures from Lambda Terms**
   - Input your lambda terms using either \ or λ for lambda.
   - Ensure there's a space between applications (variables can be multi-character).
   - Mondriλn will generate a visual representation of your term.
2. **Parse Terms from Images**
   - Provide the path to an SVG image.
   - Mondriλn will extract the corresponding lambda term.
   - Get creative! Draw your own images to parse — but don't worry, if there's an issue, Mondriλn will let you know what needs fixing.
3. **Beta Reduce Lambda Terms**
   - Input a term directly or provide the path to an image.
   - Mondriλn will beta reduce the term step-by-step, saving a visual representation of each step.
   - After each step, you'll be prompted to decide whether to continue the reduction process.



Additional Resources

- Refer to the Language Specification document for details on the visual representation.
- Check out the Inkscape Instructions to learn how to create your own SVG images for parsing.

Run locally

Install GHCup

1. Please refer to the official GHCup installation guide

Install Stack

1. Run `ghcup install stack` to install stack using ghcup
2. Verify the installation by checking the version: `stack --version`

Clone the project

1. Clone the repository: `git clone https://gitlab.ost.ch/esofunlang/esofunlang-code`
2. Navigate to the project directory

Run the project

1. Run the project by running `stack run`

How to input lambda terms

- The lambda in lambda terms is represented by a backslash '\' or by a 'λ'. Therefore, abstractions are written as `\x.x` or `λx.x`
- Applications are two terms with a space in between: `a b`
- Parentheses can be used to group terms and change the standard order (left associative) of operations
- Variable names can consist of multiple characters: `\xx.xx`

Example term λp.p (λx.λy.y) (λx.λy.x) will be generated by entering the following command:

```
generate \p.p (\x.\y.y) (\x.\y.x)
```

Development

Prerequisites

1. stack
2. HLS

We recommend setting up these tools using ghcup.

Run locally

1. Use `stack build --pedantic` to compile the project
2. Use `stack test --pedantic` to run all tests

Code completion / Syntax Highlighting

If you are using VS Code, use the following extensions to have automatic code completion and syntax highlighting:

1. Haskell
   i. On setup, you are asked whether you want to use the bundled ghcup or via PATH. We recommend using ghcup from PATH.
2. Haskell Syntax Highlighting

If the Haskell extension crashes, open the output view, choose the Haskell extension in the source dropdown and see what the error is.
In case the error is `GHC ABIs don't match`, you should be able to fix this by configuring Stack hooks as described here.
After restarting the extension, this error should be fixed.

Automatic formatting

Formatting is automatically checked using Ormolu in the CI pipeline.
To format your code locally use either of the following options:

1. Recommended: Use the integration into HLS, which is included in the Haskell extension for VS Code.
2. Setup one of the editor integrations.
3. If you want to format your code manually, e.g. as a `git hook`, you can install Ormolu locally as described here and then use the command `ormolu --mode inplace $(git ls-files '*.hs')`.

B Document with instructions for creating your own images

Creating your own images

To create your own images we recommend the vector graphics editor Inkscape.

Inkscape options

In Inkscape you should turn the following options on/off to make it easier to create these images:

- Turn "Node Snapping" on, turn off the other snapping options
- Turn off "When scaling objects, scale the stroke width by the same proportion"

Creation Flow

We recommend initially creating a document with a large width/height so that you have enough space for drawing your visual representation.

When creating your images, we recommend keeping the language specification document at hand. Remember that application separators and bounding boxes are drawn as white rectangles. Additionally, make sure that you keep the stroke width consistent within your image and that the stroke is black.

When you are done with drawing your image, use the "resize to content" button in the document properties of Inkscape and save your image.

After this step, you are ready to parse the image with the application.

Notes for editing generated images

When changing the fill colours manually in Inkscape, you may end up with two fill colours in the final svg.
Once as fill="..." and once as style="fill:...".
This can produce wrong result for the variables when parsing the image.
Unfortunately, this is a bug in Inkscape and you will need to remove the duplicated fill colours manually (fill="").