LambdaCpp: a C++-framework for interpreting lambda-calculus

Benedikt Franke

August 14, 2020

1 Introduction

Lambda calculus (often written λ-calculus) is an abstract model of computation, an alternative to the well-known Turing machines. It was invented around 1928 by Alonzo Church and published in *"A set of postulates for the foundation of logic"*[1] [2].
Church invented λ-calculus as formal system that could serve as a foundation for logic, while being based on the concept of a function, contrary to the already-existing set-theory. As contradictions were found in the original formulations, the system was revised over the following years, resulting in *pure λ-calculus*, colloquially referred to as λ-calculus today[2].
During these revisions, it could be proven that the class of *λ-definable functions* is equal to the class of Turing-computable functions [2], giving rise to λ-calculus as a model of computation.
Formally, λ-calculus can be defined by the following grammar [3]:

$$\begin{aligned}\langle formula\rangle &\rightarrow \langle variable\rangle\\ &\rightarrow \lambda\ \langle variable\rangle\ .\ \langle formula\rangle\\ &\rightarrow (\ \langle formula\rangle\ )\ \langle formula\rangle\end{aligned}$$

As this grammar looks pretty simple, it is even more surprising that it describes a computational model that is equally powerful as the Turing machine. The second rule of the definition gave λ-calculus its name: a *λ-abstraction* of the form $\lambda\ v\ .\ M$ works by binding the specified variable v in the term M. This way, v effectively becomes a parameter to the function M. When this function is applied to another function N (third rule of the grammar), all occurrences of v in M simply get replaced by N. This shows the nature of λ-calculus: Everything is a function, even the variables.
The goal of this paper is to describe the implementation procedure of a framework written in C++ that is able to parse and interpret lambda expressions. The framework, called *LambdaCpp*, includes a small example application, a *Read-Evaluate-Print-Loop* (*REPL*), that is included as an example of what can be done easily with the help of the framework. The project is available via

[1] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932, ISSN: 0003486X. [Online]. Available: `https://github.com/emintham/Papers/blob/master/Church-%20A%20Set%20of%20Postulates%20for%20the%20Foundation%20of%20Logic.pdf`.

[2] F. Cardone and J. Hindley, "History of lambda-calculus and combinatory logic," Jan. 2006.

[3] Andreas F. Bochert, *Lecture slides of Objektorientierte Programmierung in C++, Ulm University*, `http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/cpp-2018-06-19.pdf`, Accessed: 2020-08-07.

GitHub [4].

The remainder of this paper is structured as follows: In the next chapter possible design decisions in the implementation process are discussed. In the third chapter, the resulting implementation is presented. Last but not least, a summary of the work is given.

2 Design decisions, their advantages and drawbacks

During design of this framework, multiple decision had to be made:

Most obviously, the grammar of the language had to be be developed. Here, ease of use and ease of parsing had to be weighted against each other: a context-sensitive grammar might be easier to use, an ϵ-free LL(1) grammar is easier to parse. As the goal of this project is to interpret λ-calculus and not to develop a general-purpose programming language, more weight was assigned to parsing the language. Therefore a LL(1) grammar (not ϵ-free) was chosen. As a result, a recursive descent parser with one lookahead is enough to parse the designed mini-language. However, as the mini-language does not need to define a whole lot of operations, it does still have a concise and clear syntax. Some examples can be found in subsection 3.4.

As λ-calculus is defined by a formal grammar itself, the next decision to be made was whether the syntax tree for a formal λ-calculus should be separated from the syntax that purely stems from the mini-language. Examples for this are assignments or commands for beta-reduction, which should be included in the framework while not being part of λ-calculus itself. While building only one unified syntax tree for the entire language would make the code simpler and easier to maintain, it would also make using only the λ-calculus logic more difficult. This would result in making the library less versatile for re-using. For this reason, this project follows the modular approach, where lambda expressions and commands are separated from each other. Hence, it is easily possible to exclusively include this part of the framework in an application, making use of the syntax tree for λ-expressions, while not using the mini-language, the tokenizer or the parser at all.

Another consideration was adaptability of the mini-language. While designing the parser and the tokenizer in a way that could handle a wide range of language-customizations was not a priority of this project, two interfaces are still supplied that enable a user of the library to adapt parts of the mini-language to his or her needs. These are the possibility to change the ”special characters” of the language (e.g. the lambda symbol) via a template parameter, and to register *reserved symbols* with the tokenizer by specifying a tuple of a symbol and a function. When the specified symbol is encountered during the lexical analysis, the corresponding function supplied by the user is executed. A possible use case for this will be demonstrated in the example application.

One of the most central design choices was how to represent a λ-expression internally. As shown by the grammar in section 1, λ-expressions can expand into three different forms, and two of these expansions contain other lambda expressions themselves. Therefore, a valid lambda expression can be modelled as a binary tree, where the leafs are always variables.

[4] Benedikt Franke, *Repository of LambdaCpp at GitHub*, `https://github.com/BeFranke/LambdaCpp`, Accessed: 2020-08-11.

A dynamic polymorphism with three subclasses, each one implementing the same base class was therefore chosen. It will be explained in more technical detail in subsection 3.1. Of course, this polymorphism can be extended with more classes by an user of the framework, should the need to do so arise.

3 Implementation

The framework is divided into six different modules, each with a distinct function. Here, a *module* consists of one header file, optionally with an *.cpp*-file of the same name.
An overview of these modules can be seen in figure 1. A more detailed explanation for each module is given in their individual sections.

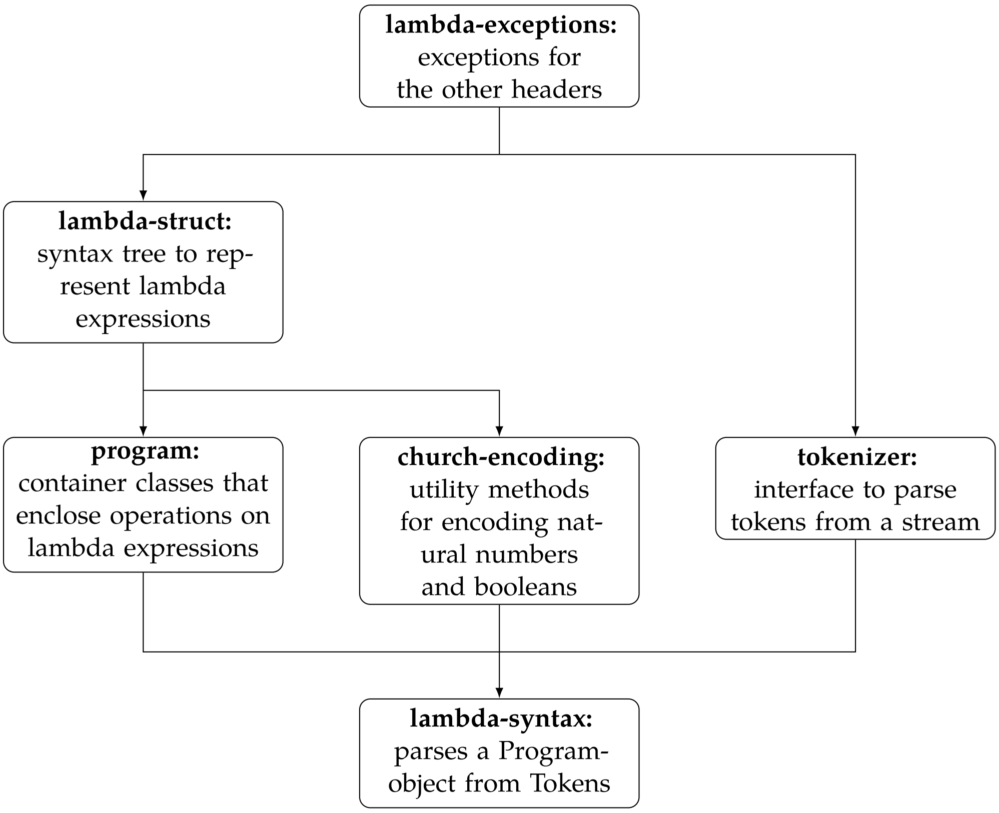

lambda-exceptions: exceptions for the other headers
lambda-struct: syntax tree to represent lambda expressions
program: container classes that enclose operations on lambda expressions
church-encoding: utility methods for encoding natural numbers and booleans
tokenizer: interface to parse tokens from a stream
lambda-syntax: parses a Program-object from Tokens

Figure 1: Overview of the modules. Arrows from module A to B represent that B depends on A.

3.1 lambda-struct - a syntax tree for lambda-expressions

The central and most important module of the package is called *lambda-struct*. It contains a polymorphism of four classes, where *Expression* is the base class used to represent all valid lambda expressions. Together with its subclasses, *Variable*, *Lambda* and *Application*, it forms a binary syntax tree, where *Lambda* and *Application*-objects form the branches, as these classes have two other *Expression*-objects as fields. The leafs of the tree are always *Variable*-objects. Figure 2 shows the class diagram of this package.

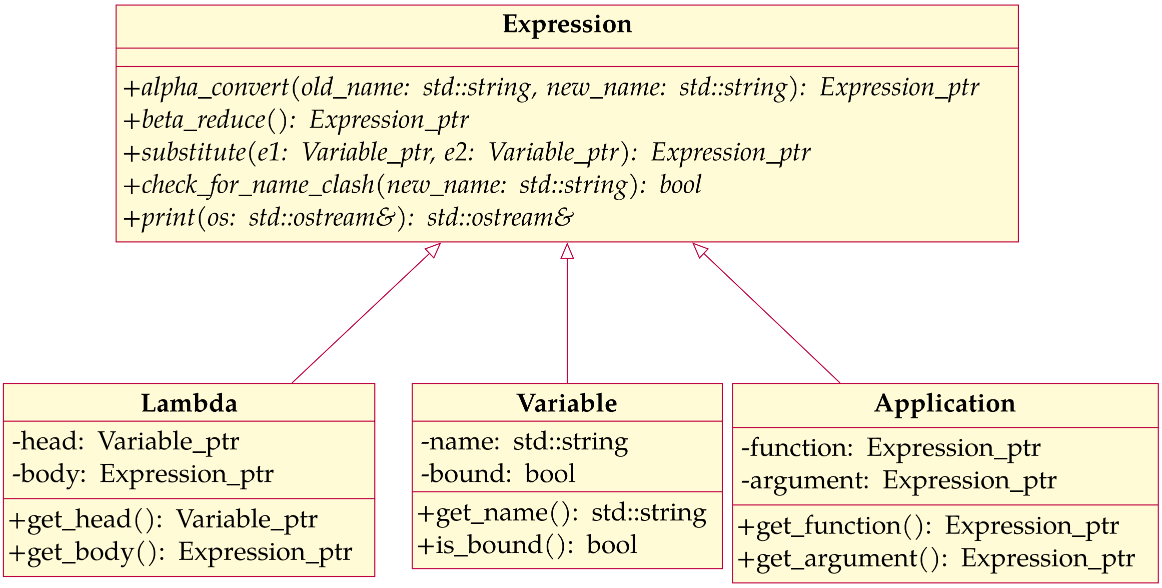

Expression
+alpha_convert(old_name: std::string, new_name: std::string): Expression_ptr
+beta_reduce(): Expression_ptr
+substitute(e1: Variable_ptr, e2: Variable_ptr): Expression_ptr
+check_for_name_clash(new_name: std::string): bool
+print(os: std::ostream&): std::ostream&
Lambda
-head: Variable_ptr
-body: Expression_ptr
+get_head(): Variable_ptr
+get_body(): Expression_ptr
Variable
-name: std::string
-bound: bool
+get_name(): std::string
+is_bound(): bool
Application
-function: Expression_ptr
-argument: Expression_ptr
+get_function(): Expression_ptr
+get_argument(): Expression_ptr

Figure 2: Class diagram of lambda-struct. *Italic* method names are purely virtual and implemented by the child classes.

The three subclasses correspond to the possible expansions of the grammar shown in the introduction. The pure virtual methods of the base class *Expression* are used to define operations on λ-expressions:

- *substitution* means replacing a variable x in a λ-expressions M by a specified other term N, and can be written as $S^x_N M|$ [5].
- *alpha conversion* represents the renaming of bound variables.
- *beta reduction* means replacing terms of the form $(\lambda\ x.M)N$ by $S^x_N M|$.

The fourth method, *check_for_name_clash* is an utility method that is used to assert that an invoked alpha conversion does not rename a variable into a name already used in the expression. The last method, *print*, is used to output the *Expression*-object to a stream. All important functions are therefore polymorphic, only subclass-dependent getter-methods exist in the corresponding subclasses.
As this module only depends on *lambda-exceptions*, it can be re-used directly to represent a λ-calculus syntax tree in any other context, too.

3.2 tokenizer - processing a stream into tokens

The first step in every compiler or interpreter is usually *tokenization*, i.e. lexical analysis of the input, along with splitting valid input into *tokens*. A token can be a single symbol (e.g. the lambda operator, here "\" is used similar to anonymous functions in the *Haskell* language [6]), or multiple characters, like a variable name or a number of multiple digits.

[5] Andreas F. Bochert, *Lecture slides of Objektorientierte Programmierung in C++, Ulm University*, `http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/cpp-2018-06-19.pdf`, Accessed: 2020-08-07.

[6] *Homepage of the Haskell language*, `https://www.haskell.org/`, Accessed: 2020-08-08.

For this task, the header defines the class *Tokenizer*, which takes a *std::istream* as argument, which defines where the tokens are extracted from. It is also possible to specify *reserved symbols*. These are symbols that, when encountered during tokenization, lead to the execution of a user-defined function. For each symbol that gets registered as a *reserved symbol*, exactly one function may be specified. This ensures better re-usability of the class, as it is possible to inject special commands into the language which are specified by the caller, e.g. a help-function in a REPL, as shown later.

Another important interface is supplied as the possibility to change the *special characters* of the language by providing a suitable *enum class* as a template parameter. The default is shown in Figure 3.

Most importantly, the class *Tokenizer* defines a public method, *get*(), which parses and yields the token from the input stream. Tokens are represented by objects of the class *Token*, which is a simple class with a *TokenType* (indicating the kind of token) and a *std::string* as members. The string contains the exact parsed symbols that lead to the creation of this *Token*.

```
enum class Symbol {
    lambda = '\\',
    body_start = '.',
    bracket_open = '(',
    bracket_close = ')',
    separator = ';',
    comment = '#',
    assignment = '=',
    conversion_end = '>',
    name_definition = '\''
};
```

Figure 3: Default enum class *Symbol*. By providing a custom enum class that defines the same members, the mini-language can be adapted.

The *Tokenizer* also reserves a symbol, by default "#", as the start of a line-comment. Therefore, if this symbol is encountered all following characters are ignored until a linebreak occurs.

3.3 program - combining lambda-expressions and operations

As described earlier, one of the key points of the framework's architecture is that the lambda syntax tree is split away from user commands, e.g. to beta reduce an expression or to assign a name to an expression. For this reason, the module *program* defines several data structures:

A polymorphism of three classes is used to represent beta reduction and alpha conversion. These are:

- *Conversion*, the base class of the polymorphism that is used to represent the identity conversion.
- *AlphaConversion* stores the name of the variable that should be renamed as well as the new name of this variable.

- *BetaReduction* stores how many steps of beta reduction should be performed, and how many steps may be performed at max before an expression is considered to have no normal form (indicated by an exception).

Furthermore, the class *command* is defined to bind one *Conversion*-object to one *Expression*-object each, and contains the method *execute* that should be used to apply the reduction to the expression. But as a statement can depend on previous inputs by referencing a previously-defined name, one last data structure is needed. This is basically a wrapper around a *std::unordered_map* that is used to store a table of *Command*s that have been assigned a name, as well as the last input under a special key. Using these data structures, it is possible to save the state of a lambda-program over multiple inputs.

3.4 lambda-syntax - assembling a lambda program from tokens

While we now have everything together to store a program written in the lambda mini-language, one important thing is still missing: Before being able to store or execute anything, it is often necessary to parse the desired program from a stream containing user input that has been parsed into *Token*s by the *Tokenizer* class described earlier. For this purpose, a recursive descent parser has been designed to parse the following grammar:

$$
\begin{aligned}
\langle statement \rangle &\rightarrow \langle assignment \rangle \ ; \\
&\rightarrow \langle rvalue \rangle \ ; \\
\langle assignment \rangle &\rightarrow \ 'NAME' = \langle rvalue \rangle \\
\langle rvalue \rangle &\rightarrow \langle expression \rangle \ \langle conversion \rangle \\
\langle expression \rangle &\rightarrow \ \backslash \ VARIABLE \ . \ \langle expression \rangle \\
&\rightarrow (\ \langle expression \rangle \ ) \ \langle expression \rangle \\
&\rightarrow VARIABLE \\
&\rightarrow LITERAL \\
&\rightarrow NAME \\
\langle conversion \rangle &\rightarrow \langle alpha \rangle \\
&\rightarrow \langle beta \rangle \\
&\rightarrow \epsilon \\
\langle alpha \rangle &\rightarrow VARIABLE > VARIABLE \\
\langle beta \rangle &\rightarrow LITERAL \ > \\
&\rightarrow >
\end{aligned}
$$

All symbols not enclosed into angular brackets are terminal symbols, uppercase words are terminal symbols that do not have a fixed representation. For example, the symbol $VARIABLE$ might match x, *hello* or $a123$, but they must start with a lowercase letter. $NAME$s are named lambda-functions, they must start with an uppercase letter to distinguish them from variables. *LITERAL*s currently match positive integers as well as the strings "true" and "false". These labels are all assigned by the *Tokenizer* described earlier. The parser can therefore treat them like symbols even though they do not have a fixed representation.

It is possible to recognize the basic λ-calculus-grammar presented in section 1 by looking at the expansion of the symbol ⟨*expression*⟩. However, for the mini-language to be well usable, the expansion has been extended by literals and named functions. In its current form the grammar expects every statement to be terminated by a semicolon, enabling input to be split among multiple lines. If this is not desired, these symbols can be changed as described in subsection 3.2.

As previously mentioned, comments are split away by the*Tokenizer* and therefore do not need to be included in the grammar.
In program code, this parser is embodied by the class *Parser* in *lambda-syntax*. It defines a single public method, *statement*(), that requests *Tokens* from the*Tokenizer* one by one, either returning a *Program*-object on success or throwing an error. It also exposes the accumulating *Program* object as a public member, which makes it possible to manipulate the accumulator between *statement*()-calls. For example, this is used in the *REPL* that is presented in subsection 3.6 to store the last command for access by the user.
Internally, *Parser* has one method per non-terminal symbol that gets called when the corresponding part of the grammar has been recognized. As the grammar shown above belongs to the $LL(1)$-grammars, this can be done purely by requesting a token, checking its type and calling the right method, without the need for backtracking or multiple lookaheads.
The biggest drawback of this language design is that brackets are not optional: A lambda expression enclosed in unneeded brackets will not be parsed correctly. However, this should not hinder readability considerably, as brackets are already mandatory for function application.

Using this mini-language, all relevant operations on λ-expressions are possible, for example:

Command	Explanation
$'ID' = \backslash\ x\ .\ x;$	Assign a name to an expression
$ID\ y;$	Use assigned function
$\backslash y\ .\ (x)\ y\ \ \ y > a;$	Invoke alpha conversion to rename 'y' to 'a'
$(\backslash x\ .\ (x)\ x)\ a\ \ >$	Invoke beta-reduction with '>'
$(2)\ 1$	Literals get automatically expanded into λ-calculus
$(true)\ x$	'true' and 'false' are reserved as boolean values
$\backslash ab\ .\ (ab)\ c$	variable names can be longer than one letter

By including literals like natural numbers or booleans, simple arithmetic operations are possible out-of-the-box, e.g. the above presented expression (2) 1 would expand to

$$(\backslash f\ .\ \backslash\ x\ .(f)(f)x)\backslash f\ .\ \backslash\ x\ .(f)x$$

which could then be beta-reduced and alpha-converted to $\backslash\ f\ .\ \backslash\ x\ .(f)(f)x$, which is the λ-calculus-expression for 2, the presented expression is therefore equal to the multiplication of 2 and 1.
In the following section, it will be clarified how this expansion of numbers into λ-expressions works.

3.5 utilities - church-encoding and lambda-exceptions

There are two modules containing methods for other modules while not having real value when used stand-alone:

The module *church-encoding* contains methods to conveniently get the λ-calculus representations for natural numbers and boolean values, computed by *Church encoding*. Using this module, the parser can automatically expand literals into λ-calculus. When using *Church encoding*, a number $n \in \mathbb{N}_0$ is represented by applying a function n times, e.g:

$$0 \rightarrow \lambda f . \lambda x . x$$

$$1 \rightarrow \lambda f . \lambda x . (f) x$$

$$2 \rightarrow \lambda f . \lambda x . (f) (f) x$$

and so forth. Also implemented are boolean values, where $true \rightarrow \lambda a . \lambda b . a$ and $false \rightarrow \lambda a . \lambda b . b$.

The other module, *lambda-exceptions*, contains a variety of exceptions that can be thrown in the other headers. All of these exceptions derive from a common base class, *LambdaException*, in order to make it possible to catch all exceptions thrown by the framework at once by simply catching a *LambdaException*.

3.6 Example application

As an example for demonstration, a REPL for lambda-expressions is included in the framework. It demonstrates how the framework can be used by taking input from *stdin*, interpreting and executing the user-supplied expression and printing the result to *stdout*.
As the framework does most of the work, the file *repl.cpp* only needs to handle in- and output as well as errors, like catching Syntax Errors and informing the user about it or providing a help message when it is requested.
By inputting statements written in the mini-language, a user can freely evaluate any λ-calculus expression by beta reduction, change variable names via alpha conversion or assign a name to a λ-expression to store it for later. Taking inspiration from many calculators, the name *Ans* can be used to retrieve the last result, making it easy to iteratively evaluate expressions without having to re-assign a name in every step. A short help text via the reserved symbol ″?″ is also provided.
This function is implemented using the *reserved symbol*-interface of the modules *lambda-syntax* and *tokenizer*, by registering ″?″ as a reserved symbol with the parser and specifying a function that should be called when this symbol is encountered.
As described previously, the parser executes the correspondingly registered function as soon as it encounters a reserved symbol, which enables the caller to inject its own logic into the parsing process. In this case, for ″?″ a help message is printed, while for ″exit″, the REPL terminates.

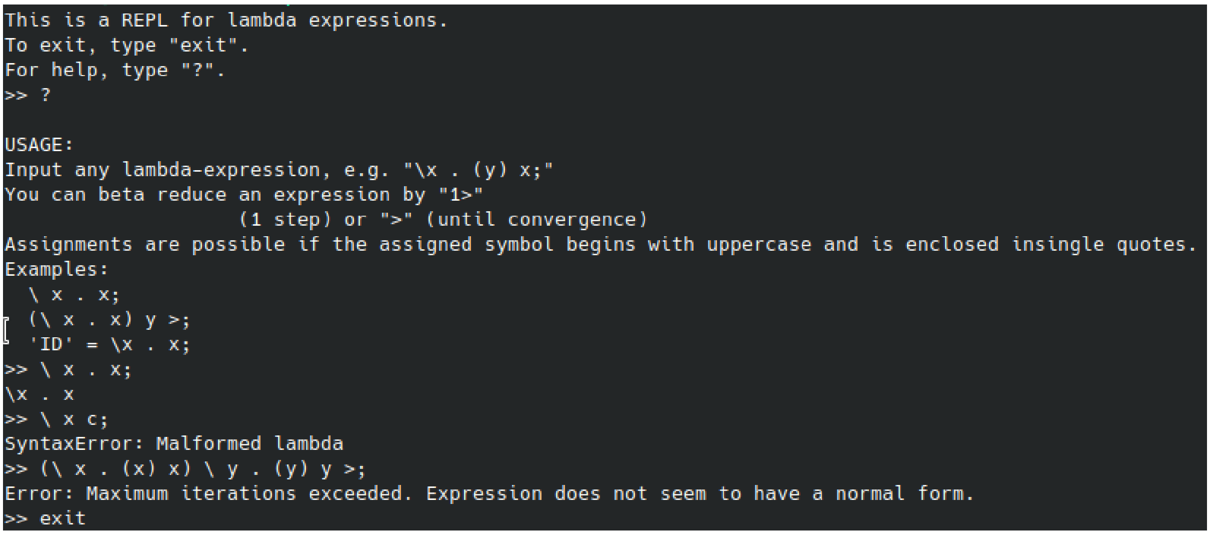
3.7 Testing and toolchains

On the technical side, *LambdaCpp* uses *CMake*[7] to generate build-scripts. The project was built with C++14 and g++ version 9.3.0.
Testing is done using *googletest* [8], which is automatically downloaded by the CMake-script if *CMAKE_BUILD_TYPE* is set to *DEBUG*, making it easy to reproduce the testing process.

[7] *CMake homepage*, `https://cmake.org/`, Accessed: 2020-08-12.

[8] *Repository of googletest at GitHub*, `https://github.com/google/googletest`, Accessed: 2020-08-12.

For all library files, unit tests are provided that amount to C0-coverage. The example application, *repl.cpp*, can be manually C0-tested by inputting just five commands, shown in Figure 4. A coverage report made with *lcov* is included in the repository.



```
This is a REPL for lambda expressions.
To exit, type "exit".
For help, type "?".
>> ?

USAGE:
Input any lambda-expression, e.g. "\x . (y) x;"
You can beta reduce an expression by "1>"
                    (1 step) or ">" (until convergence)
Assignments are possible if the assigned symbol begins with uppercase and is enclosed insingle quotes.
Examples:
  \ x . x;
  (\ x . x) y >;
  'ID' = \x . x;
>> \ x . x;
\x . x
>> \ x c;
SyntaxError: Malformed lambda
>> (\ x . (x) x) \ y . (y) y >;
Error: Maximum iterations exceeded. Expression does not seem to have a normal form.
>> exit
```

Figure 4: These five commands lead to every line of *repl.cpp* being executed at least once.

4 Summary

In this paper, the implementation of *LambdaCpp*, a framework for interpreting λ-expressions was presented. In the beginning, design decisions made during the development process were discussed while considering the specific benefits and limitations of the resulting abstractions and interfaces. Following this discussion, the resulting implementation was presented module by module, with attention to inter-module dependencies and reusability. The central syntax tree used to represent λ-expressions was discussed in detail, as well as the grammar of the designed mini-language with its advantages and drawbacks.

The implemented framework is modular to enable a user to make use of certain features exclusively. For example, the syntax tree for λ-calculus is separate from the mini-language used in the REPL. Where appropriate, configuration options via template parameters or other interfaces are given.