FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Implementation of lambda expressions evaluator
Student:	Jan Sliacký
Supervisor:	Ing. Petr Máj
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	Until the end of summer semester 2019/20

Instructions

Using the definition of lambda calculus from the BI-PPA course, implement evaluator of lambda expressions to be used in the course. The evaluator should allow step by step interaction with the student and detect any mistakes. Evaluate the data mining methods to aid the detection of the type of the errors made by students and provide an implementation of this feature to be used interactively by students. The application should be runnable in browser and its code should be well designed and documented so that it can be used as teaching aid as well.

References

Will be provided by the supervisor.

Ing. Karel Klouda, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 15, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementation of lambda expression evaluator

Jan Sliacký

Knowledge engineering
Supervisor: Ing. Petr Máj

May 16, 2019

Acknowledgements

Author of this thesis would like to thank for exceptional support, inspiration, and guidance to his supervisor Ing. Petr Máj.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 16, 2019 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Jan Sliacký, Jan Sliacký. *Implementation of lambda expression evaluator.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019. Also available from: ⟨`https://github.com/lambdulus/`⟩.

Abstrakt

λ-kalkul je stěžejní koncept počítačových věd. Jako takový je učen na většině univerzit vyučujících informatiku a počítačové vědy včetně FIT ČVUT. Pro mnoho studentů může být studium λ-kalkulu a pochopení jeho významu a dopadu na současné programovací jazyky obtížnou úlohou. Tato práce vytváří evaluátor λ-kalkulu a jeho front-end navržený tak, aby prezentoval λ-kalkul jako programovací jazyk a umožnil snadnou integraci do výukových materiálů.

Klíčová slova lambda calculus, λ-calculus, evaluator, interpreter, interaktivní interpreter

Abstract

λ-calculus is a fundamental concept in computer science and as such is taught at almost all universities with a computer science programme, including FIT CTU. But for many students, learning the λ-calculus and understanding its significance and impact on programming languages is a challenging task. This thesis describes a λ-calculus evaluator and its front-end designed to help students understand λ-calculus by treating it more like a programming language and by effortless integration with existing course materials.

Keywords lambda calculus, λ-calculus, evaluator, interpreter, interactive interpreter

Contents

List of Figures

Introduction

λ-calculus is one of the fundamental concepts of computer science. As a theoretical concept, it is essential for constructing formal proofs for modern programming languages. Furthermore, it is a theoretical foundation for all functional languages. It is taught at most universities teaching computer science, including FIT CTU (course BI-PPA). Despite the core concept of λ-calculus being very simple, it often leads to confusion when approached for the first time – mainly because of its notable difference from practical programming languages. For students, it can be quite challenging to learn how to think and write programs and algorithms in λ-calculus in a similar way they are used from practical programming languages. On top of that, most of the students, already having experience with imperative programming languages, struggle with the purely functional paradigm, namely with the absence of state and sequential control flow.

The author of this thesis, as a former student of the course PPA, learned λ-calculus the hard way and realised there is significant room for improvement. The most important part of teaching and making λ-calculus easier for students, is to introduce and describe λ-calculus as a programming language. Students must be able to adapt it and use it the same way they are using other programming languages. For that reason, interactive and user-friendly evaluator – Lambdulus must be developed. Lambdulus should help students to realise, that λ-calculus can be considered the simplest pure functional language. To facilitate this goal, λ-calculus must be introduced to the students not as a purely theoretical concept, but as a programming language, complete with tools to help writing λ-calculus "programs", observing their execution and debugging their errors in ways similar to the workflow of programming languages the students already know and use.

Courses teaching λ-calculus often employ λ-calculus evaluators towards this goal. However, these suffer from two main drawbacks: While they evaluate λ-calculus expressions, they lack more advanced support such as debugging. They often use lambda-calculus definition and syntax far different from

what is taught at the courses.

This thesis describes Lambdulus, a λ-calculus evaluator developed to address the main shortcomings of existing λ-calculus evaluators from the educational point of view.

Lambdulus will be used primarily by the students of the PPA course mentioned above. As such, it supports directly the definition and notation of λ-calculus used in the course. But Lambdulus has been designed with modularity in mind and adopting other λ-calculus dialects is extremely simple.

Any application actively trying to support the learning of programming can only be effective if it can be integrated seamlessly into existing course materials so that students can effortlessly use that tool daily. To this end, Lambdulus allows simple, yet robust embedding into existing course materials as well as sharing examples and whole λ-calculus programs among students and teachers.

Finally, the purpose of this thesis is not only to create evaluator useful for teaching but also produce code-base illustrating an exemplary implementation of the λ-calculus interpreter so that students can view, explore and better understand the simplicity and expressive power of the λ-calculus.

The rest of this thesis is organised as follows: The following chapter defines the λ-calculus and its semantics. Chapter 2 describes existing λ evaluators. Chapter 3 proposes the design of the evaluator created as the topic of this thesis. The next chapter 4 explains the general architecture of developed evaluator and its major parts. Chapter 5 demonstrates the benefits of such a tool. The thesis concludes with a summary and discusses future work.

Chapter 1

Definition of λ-calculus

Despite the fact, that λ-calculus is one of the most fundamental concepts of computer science, its definitions differ across many materials. Specific definitions usually agree on the common core, then introduce some additional properties of the language. For this thesis, the definition of λ-calculus from the course PPA is used [1, Syntax of the λ-Calculus]

Valid λ expression is:

1. Single variable. Variable can be any sequence of lowercase letters from the English alphabet also followed by the sequence of number
2. When M and N are valid λ expressions, the followings are also valid λ expressions:
   - (M) – λ expression enclosed in parentheses
   - (λx . M) – where x is any variable, called λ abstraction; being a function which takes argument named x and which body is M
   - (M N) – called application, where M is applied to N, similar to calling function M with an argument N

For reasons of practicality and by PPA course [1, Simplifying the notation], additional features, which are easily transformed to the λ-calculus defined above are defined, as follows:

- implicit parentheses

  ```
  M N O P Q
  ```

  is equivalent to

  ```
  ((((M N) O) P) Q)
  ```

- λ function with multiple arguments

```
(λx y . M)
```

is λ abstraction equivalent to

```
(λx . (λy . M))
```

- square bracket closing all open parentheses

```
(A (B (C D]
```

is equivalent to

```
(A (B (C D)))
```

1.1 Free and Bound Variables

As declared in the materials of the course PPA [1, Free and Bound Variables]: Depending on the context where a variable name is used in the λ expression, that variable is either free or bound.

A *free* variable is variable with a name that does not correspond to the name of the enclosing function's argument.

A *bound* variable is variable in the body of λ function with the same name as the argument of λ function enclosing that variable. Such variable is said to be bound to the argument.

1.2 Reductions of λ expression

λ expression can be transformed using the following rules defined as [2, Reductions]:

- β reduction (*application*)

  "Given the λ expression (λ x . E) A, all bound occurrences of x in E will be replaced with A. However, the β reduction can only be applied if A does not contain any free variables that might collide (i.e. would become bound when substituted in E)."

- α conversion (*renaming*)

  "Builds on the observation that two λ expressions which are identical in all but names of bound variables are indeed identical (i.e. there is no difference between (λx . x) and (λy . y), they are both identity functions). α conversion is thus defined as renaming λ argument x and all its bound occurrences in the λ body to y as long as y does not appear as a free variable in the λ body."

- η conversion (*optimization*)

  "Given a λ expression (λ x . A x) where there are no bound occurrences of x in A:"

  $$(\lambda\ \mathtt{x}\ .\ \mathtt{A}\ \mathtt{x})\ \eta\rightarrow\ \mathtt{A}$$

  "This *is true* because if the expression above was ever followed by an application of B, the result of it after substitution would have been:"

  $$(\lambda\ \mathtt{x}\ .\ \mathtt{A}\ \mathtt{x})\ \mathtt{B}\ \beta\rightarrow\ \mathtt{A}\ \mathtt{B}$$

It is worth noting, that there is a formal difference between *evaluation* and *normalization* of λ expression [3]. Formally defined, former cannot reduce λ function if that λ function is not being β reduced. Latter can step into any λ function and proceed with a reduction inside its body. For the sake of simplicity, the course PPA does not distinguish between those terms and defines only the *normalization*, which it calls evaluation. From now on, when term *evaluation* will be used, it will be referring to *normalization*.

The course PPA defines two orders of evaluation [2, Normal vs Applicative Evaluation]:

- Normal order (*leftmost outermost*)

  "In this order, expression being currently reduced is the leftmost λ function with the corresponding argument."

- Applicative order (*leftmost innermost*)

  "This order is identical to the former in terms of finding the λ function to apply. But before the argument of the λ function is substituted, it is first evaluated itself."

1.3 Normal Form

According to the course PPA Normal Form is defined as follows [2, Normal Form]:

"λ expression is said to be in normal form if it cannot be further reduced using either β or η reductions. Note that α reductions are always possible since they do not change the λ expression."

Chapter 2

Analysis of existing evaluators

Due to the simple definition of the λ-calculus, general λ evaluator is an uncomplicated tool. What makes this tool relatively complex, however, is demand for illustrative features. To effectively teach and expose all of those operations, transforming inputted λ expression.

This chapter compares λ evaluators designed for educational purposes. Each of them has a unique set of features, which are described in detail. Many of them are currently used in the course PPA as a teaching tool. Lambdulus intends to replace them all.

In this chapter, terms macro and operator are used. Operator differs from macro in an only way; it must be arithmetical or logical operator such as + or *AND*. Otherwise, they are effectively the same thing.

2.1 Lambda Calculus

This evaluator [4] uses backslash instead of λ symbol and arrow symbol instead of a dot symbol between last argument name and function's body.

It also allows the use of multi-argument λ functions. It understands comments like most programming languages do.

This particular evaluator features user-defined macros. As seen in figure 2.1, each macro is defined as an assignment to the identifier, which can be named either with numeric or alphanumeric string. It is, however, not possible to create macro for arithmetic operators and other special symbols.

The evaluator does not offer stepping functionality. For that reason, it is not clear when or how are user-defined macros expanded. It evaluates input in the background and then shows the result.

Evaluation of the input is not limited to reducing single expression. User can define as many macros as they want and also specify as many λ expressions to reduce as needed. As long as each expression is on its own line, *Lambda Calculus* can parse and evaluate it. This is undoubtedly a very useful approach,

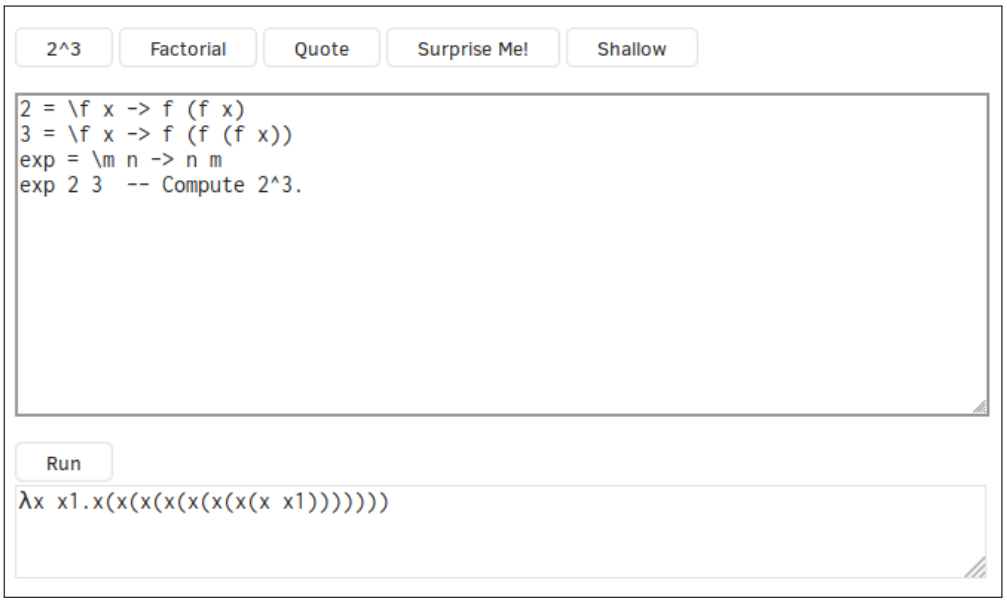

Figure 2.1: Lambda Calculus

which leads to quick and easy workflow when the user needs to evaluate more than a single λ expression.

Lambda Calculus supports only *normal evaluation order*.

2.2 λ Calculus Interpreter

This evaluator's UI differs noticeably from all of the others. Its user interface is composed of a terminal-like interactive console [5].

In terms of the syntax, λ *Calculus Interpreter* uses substitute keyword *lambda* instead of λ symbol in the inputted expression. Each λ abstraction can only be a function of a single argument. It also requires to enclose each application in parentheses, making the following expression ambiguous.

```
(λ x . λ y . x y)
```

Because when not parenthesized, it means

```
(λ x . (λ y . x) y)
```

And not

```
(λ x . (λ y . x y))
```

Which is its actual meaning.

λ *Calculus Interpreter* does not interpret numbers or operators. For the user, it is also not possible to define their own macros.

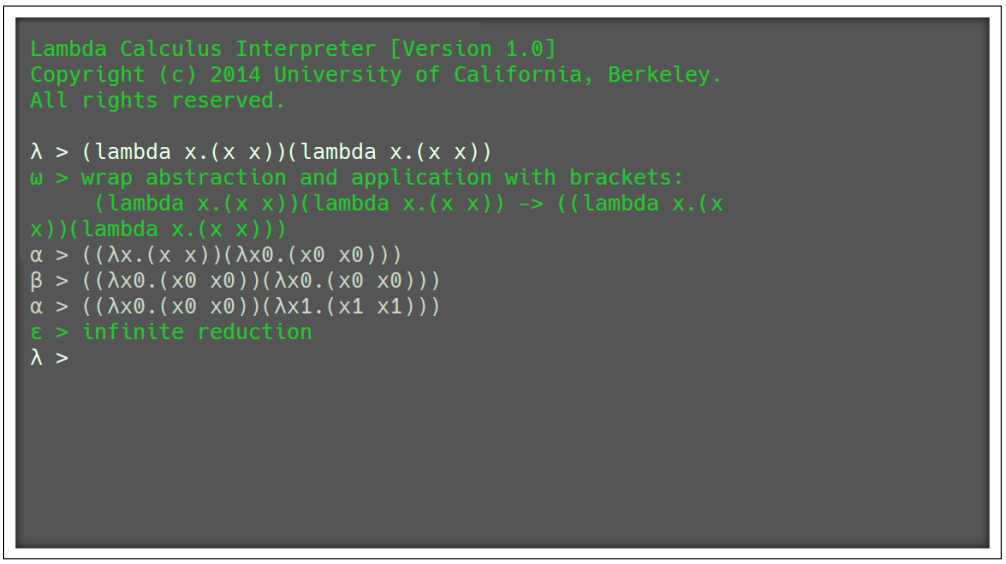

Figure 2.2: λ Calculus Interpreter

It, however, offers *Normal Order Reduction* and *Normal Order Evaluation*. Former reducing due to well-known and previously defined rules, latter ignoring possible reductions inside the function's body.

A notable feature of this evaluator is its detection of infinite recursions and exponentially expanding parts of the expression. This functionality may often prevent further evaluation of malformed expressions.

2.3 Lambda Calculator

This evaluator [6] also uses backslash as a convenient way to express λ symbol. Each λ function has a single argument, and it understands implicit closing inside function's body, allowing to omit body level parentheses.

Lambda Calculator supports numbers if the user enables them explicitly. In that case, they are treated strictly as numeric values. They can be passed as arguments to arithmetic operators, but they cannot be treated as expandable macros. A similar situation is with arithmetical operators. They can be applied to numeric values, but they are not expandable macros so they will not be applied to numbers in the form of Church numerals.

User is, however, allowed to define their own macros. In combination with the possibility to redefine built-in arithmetical operators and also numerical values, it is possible to modify the behaviour of said operators to work with Church numerals. When redefined by the user, all operators work only with native λ expression. And until the user deletes their macro, they can not be

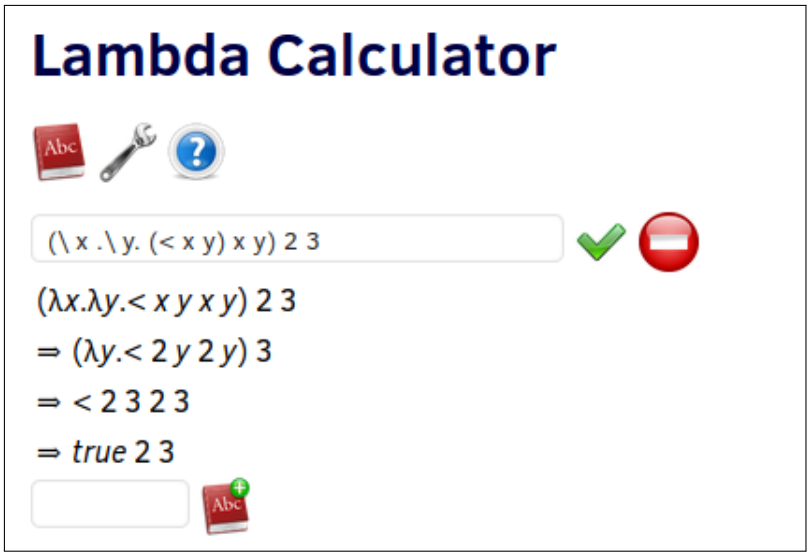

Figure 2.3: Lambda Calculator

used again with numerical or logical values. When enabled, *Lambda Calculator* can substitute symbols – user-defined macros and arithmetical operators. This substitution enables to expand macros and operators only when needed, but also tends to substitute one macro for another instead of fully unwrapping them. For instance, values *False* and number zero in the form of the Church numeral suffer from this inconvenience. They are equivalent λ functions and *Lambda Calculator* substitutes one for another instead of expanding them and letting the user see what they look like 2.4.

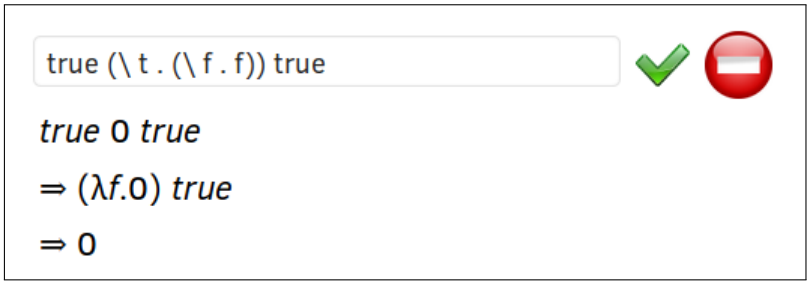

Figure 2.4: Lambda Calculator – substitution

Another option is never to substitute those symbols. This means every symbol will be expanded right at the begging of evaluation, making the steps noticeably more complex and hard to read. Similar confusion may occur, when the user selects *Pure Calculus With Numerals*. In this case, numbers are expanded immediately, and operators are left as they are, making the corresponding expression non-evaluable.

Lambda Calculator, as only one from analysed evaluators implements η-conversion.

It can be configured to use either *Normal Evaluation* or other strategies which are out of the scope of this thesis.

2.4 Lambdalab

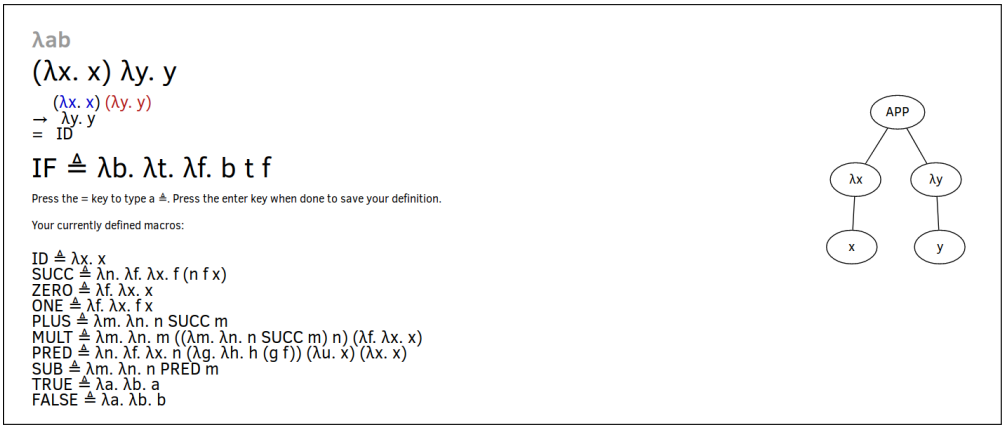

Figure 2.5: Lambdalab example

Lambdalab [7] is only one of the tested evaluators, which uses a λ symbol for representing λ abstraction. When the user writes backslash, it replaces it with a real λ symbol. This is a very convenient way to type the correct symbol without too many difficulties. Same as most of the previous evaluators, it also understands only single-argument λ functions. It also uses implicit closing around the function's body.

This evaluator does not implement any particular logic around numbers. They are primitive values, which cannot be used as corresponding λ functions.

Compared to that, operators are not allowed at all. If typed, they will cause a syntactic error.

It, however, supports user-defined macros. Macro can be an identifier with all uppercase letters in its name. *Lambdalab* also forbids the use of macro, which was not previously defined. When the macro is used in the inputted expression, it is expanded only when it is needed due to the corresponding evaluation order.

This particular evaluator also offers multiple evaluation strategies. Previously mentioned *Normal Order*, *Applicative Order* and some additional, which exceed the scope of this thesis.

Lambdalab offers a very convenient way to share λ expressions between multiple users or devices. It allows copying URL with encoded expression

and serialised settings for easy sharing. Sharing feature, however, does not serialise user-defined macros, which is something that could be very useful.

Finally, it allows to visualise the inputted expression and also every step of its evaluation. This feature can be really useful, especially when dealing with complex and long expressions. *Lambdalab*, however, seems to have issues with displaying those long expressions, rendering a substantial part of graph outside viewing area or otherwise malformed, as seen in figure 2.6.

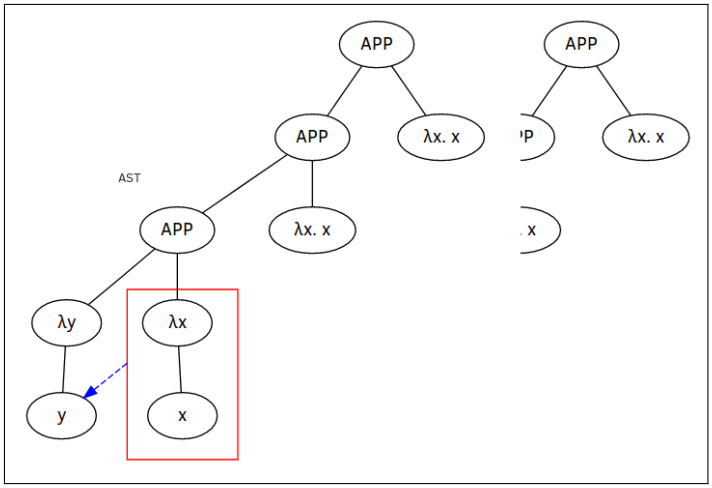

Figure 2.6: Lambdalab – visualisation of λ expression

This chapter introduced multiple λ evaluators currently available on the web. Many of them are created by computer science students. Others were created on the universities to support the teaching of the λ-calculus. Most of them share similar traits in terms of features. All of them implements well known *Normal Order* of evaluation. Some of them additionally feature *Applicative Order* and a few other specific ones. Except for one, all of them implements macros or numbers. In the way they treat macros and numbers lays the real difference between individual λ evaluators. Some implementations enforce norms, others, on the contrary, allows the user to modify virtually anything even built-ins.

Chapter 3

Design

This chapter considers positive and negative design decisions of previously introduced λ evaluators. It also states features, which Lambdulus should have, in order to compete with current evaluators on the internet.

To ensure future extensibility, implemented core and the front-end will be independent on each other. Any additional abstraction will be introduced as part of customizable front-end, whereas the core will remain small and universal.

This chapter will further describe specific design decisions and features of Lambdulus.

3.1 Evaluator Core

The core module will address parsing, representation of the expression in the form of the abstract syntax tree (AST) and expression evaluation. Every single step of evaluation, also referenced to as reduction or conversion, should be done independently to whole evaluation sequence. The evaluation process will be controlled within front-end's user interface. The core module will not feature any functionality directly related to continuous evaluation. The core creates AST, passes it to front-end and then is used to identify and perform the next step of evaluation when prompted by front-end; with both operations done atomically.

Lambdulus will implement two common evaluation strategies – *Normal Order* and *Applicative Order* to fulfil the objective of this thesis. Both orders are subject of course PPA, and as such, they should be implemented in the way they were defined earlier in chapter 1.

The core will not implement any features to support λ abstraction with multiple arguments. Every part of this syntax short-hand will be implemented within the front-end. For instance AST of λ expression (λ x y . x y) looks as shown on the figure 3.1. Moreover, the core will also not implement any

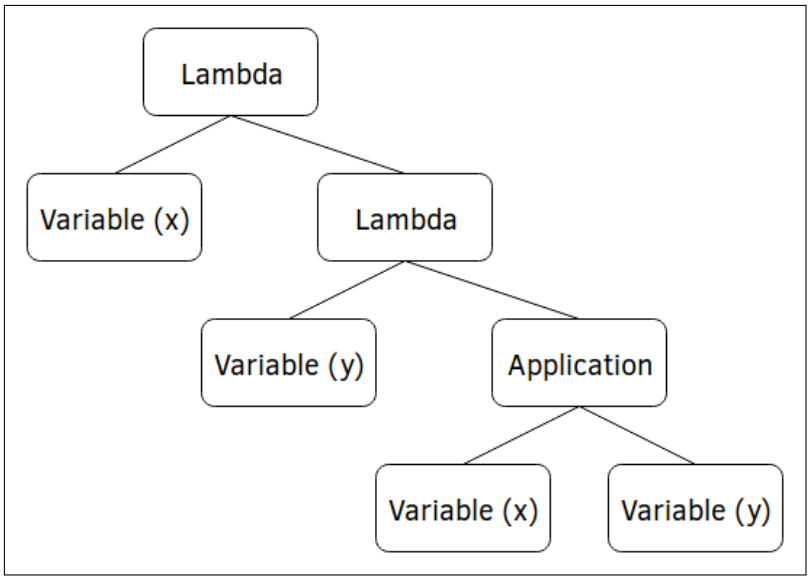

Figure 3.1: AST of λ abstraction

support for interaction between the user and the Lambdulus; core module is intended to only communicate with compatible front-end.

Following sections describe important features and the design of the core module.

3.1.1 Macros

Macros should expand only when it is needed. Macros are not a natural part of the λ-calculus. They are, however, useful addition, when treated as lexical substitutes for explicitly declared λ expressions. Use of the macros makes λ expressions more readable. The positive effect is especially noticeable when comparing two or more subsequent steps in the process of evaluation. Correct use of macros can greatly simplify parts of the expression and make it look cleaner and shorter. That is possible because the Lambdulus treats each macro as identifier until that macro is being applied to its arguments.

Not every macro must be λ abstraction. Lambdulus accepts any valid λ expression as the macro. This allows the user to encapsulate for example parts of the λ expression in complex algorithms or whole algorithms, as in listing 1. The listed expression is not traditional λ function, but still can be treated as such. [1] Once a macro is expanded, newly created sub-expression can contain

[1] Once the expression is declared as macro, it can be applied to the argument in the same way as any λ abstraction. This benefits from the fact that everything in λ-calculus is a function; even result of any application.

as many other macros as needed. This is especially useful for evaluation of recursive algorithms.

Arithmetical and logical operators are non-modifiable macros. These represent well-known abstraction and user will expect to be able to use them in their expression.

```
nth-item-of-list := Y (λ fn lst n .
                        if (= n 0)
                           (first lst)
                           (if (null (lst))
                               nil
                               (fn (second lst) (- n 1))
                            )
                       )


nth-item-of-list (cons 3 (cons 5 (cons 1 nil))) 2
```

Listing 1: Usage of macros – function nth-item-of-list

3.1.2 Numbers

Numbers in the λ-calculus are an abstraction for native Church numeral expressions. When expressed numerically, they offer a convenient and easy way to identify and observe actual values being passed into the λ functions. Despite that, they are still actual λ functions with all natural properties. They can be applied to the argument of any kind, and Lambdulus treats them virtually the same way as the macros. Numbers differ from macros in the way they decide to which form they should expand. While macros obtain their definition at the initialisation, native λ expression for numbers are constructed at the time of expansion. Because of that, λ expression can contain really big numbers without losing readability.

3.2 Front-end

Front-end module will implement user interface and have control over the evaluation process as a whole. The front-end will also be responsible for visualisation and representation of AST. However, front-end will not be allowed to modify AST directly; it would undermine the separation of responsibilities that both core and front-end have.

Because the Lambdulus will be mostly used as the teaching tool for the course PPA, it should consistently implement notation as used in the said course. Moreover, because this course, like many others, offers study materials

on the web, Lambdulus should be easily integrated within these materials. This consists, among other features, of ability to create shareable links to the particular λ expressions. Study materials would benefit from such a feature because any lambda expression would be directly linked to Lambdulus and evaluable within a moment.

As a static website, Lambdulus will offer an option to use it without an internet connection or any remote server after the first load. User is thus given two options. Either they can load web application in their browser and use it in the standard way as any other web page, or they can download the whole project from a public repository and serve it locally.

The first option is expected to be preferred by most users, not only because it offers an easy way to obtain the latest version of the tool, but also because it is easy and convenient.

However, there is a technology centred around the idea of distributing web applications as standalone desktop applications. It is entirely possible that Lambdulus will be distributed as the one in the future. However, at the time of writing, it would not bring any benefit to the students and other potential users of Lambdulus.

Next section describes the most important part of the front-end application.

3.2.1 Debugging and evaluation

To offer complete control over evaluation, Lambdulus implements the step-by-step and continuous evaluation together with debugging functionality. Simple step-by-step evaluation is built directly on top of the core's functionality. Each step passes current AST to the core and stores returned new AST. Similar to this, the continuous evaluation does the same without the need for user interaction for each step. User starts the procedure and then either interrupts evaluation or waits until the evaluation ends. Continuous evaluation ends when λ expression cannot be further reduced. Before this state is reached, the user can interrupt evaluation.

Making the user interrupt evaluation on their own every time does not lead to the best user experience. For that reason, Lambdulus implements break-points. Break-point can be attached to any λ expression. User can interrupt on the β reduction of λ function and even choose specific argument in case of multi-argument functions. Macro and number expansions are also breakable.

These features will be combined with advanced visualisation functionalities in the near future. For example, the interactive colouring of arguments would offer even more insight into the debugging process.

3.3 Error Detection

To encourage students to learn λ-calculus reductions, Lambdulus features validation functionality. User can ask the tool to let them perform the next step manually. Then Lambdulus checks if the λ expression is correct. In case user-submitted expression is not valid, Lambdulus uses an expert system to try and detect possible mistakes done by the user. It should help students understand the process of evaluation better and prepare them for any important exams.

Chapter 4

Realization

For this specific task, TypeScript language [8] was chosen. TypeScript is JavaScript [9] superset with structural static types. It is transpiled to any specified version of JavaScript. TypeScript compiler produces code, which runs in most modern browsers, is reasonable readable and efficient. Moreover, to write quality code that would be easily read by students of the course PPA, TypeScript seems like the best option – it has syntax very similar to the one of C++ [10]. Compilation step allows to catch and fix a significant number of potential bugs, and it also introduces some useful semantic concepts for keeping code cleaner.

There are numerous options for developing client-side – web-based application. React.js [11] was chosen for its relative simplicity and rich set of features. React is JavaScript library developed by Facebook, Inc. and community. It is created for declarative programming of UI and associated logic. React implements single way data flow, which helps to keep control over data and logic in clean and elegant shape. On top of that, TypeScript supports React sufficiently for the purposes of this thesis.

This chapter describes two main modules of Lambdulus – core and front-end. The core consists of the lexer, parser and evaluation procedure. Web front-end consists of interactive features and error validation functionality employing methods of knowledge engineering.

4.1 Evaluator Core

This module is responsible for parsing and representation of λ expression as AST. It also implements reductions making evaluation of that expression possible. This chapter describes all of its features and abilities.

4.1.1 Lexer

⟨*number*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*ident*⟩ ::= letter { ⟨*letter*⟩ } { ⟨*digit*⟩ }

⟨*digit*⟩ ::= '0'
| '1'
| '2'
| '3'
| '4'
| '5'
| '6'
| '7'
| '8'
| '9'

⟨*operator*⟩ ::= '+'
| '-'
| '*'
| '/'
| '<'
| '>'
| '<='
| '>='
| '='
| '^'

⟨*keywords*⟩ ::= ⟨*lambda*⟩
| '('
| ')'
| '.'
| ']'

⟨*lambda*⟩ ::= λ
| '\'
| ⟨*user-defined-token*⟩

Listing 2: Lexer EBNF grammar

The listing 2 describes EBNF [12] grammar for lexer. Numbers are represented exclusively as integers in decimal radix since they are translated to Church numerals. Operators available in Lambdulus are non-configurable. Each supported operator is strictly defined in implementation and user is not able to redefine it. Every identifier is at least a single alphabetic character. This character may be immediately followed by a sequence of alphabetic characters forming the word. In both cases, a single character or word is optionally immediately followed by a valid number. However in course PPA it is common practice to use single-letter names for identifiers while omitting white space in between them. This practice 3 allows for shorter expressions on the whiteboard and is also supported in Lambdulus.

```
(λ abc . cba) 1 2 3

≡

(λ a b c . c b a) 1 2 3
```

Listing 3: Example of λ expression with single letter names

4.1.2 Parser

$\langle LEXPR \rangle \quad ::= \langle SINGLE \rangle \{ \langle SINGLE \rangle \}$

$$
\begin{aligned}
\langle SINGLE \rangle \quad ::= & \langle number \rangle \\
| \quad & \langle operator \rangle \\
| \quad & \langle ident \rangle \\
| \quad & \text{`(' } \langle lambda \rangle \langle ident \rangle \{ \langle ident \rangle \} \text{ `.' } \langle LEXPR \rangle \text{ `)'} \\
| \quad & \text{`(' } \langle LEXPR \rangle \text{ `)'}
\end{aligned}
$$

Listing 4: Parser EBNF grammar

Parser for Lambdulus is implemented as simple recursive descent parser [13]. Recursive parser leads to simple and easy to understand code.

Parser traverses top-level expression until it encounters symbol '(' which signals nested λ expression, this expression can be either λ application or λ abstraction – every λ abstraction must be enclosed in parentheses. It is also valid to nest non-empty λ expression in many levels of redundant parentheses. Redundancy of parentheses does not have any effect as they are entirely removed by the parser. Once it is clear which kind of expression is enclosed in these parentheses, it is then parsed in the entirely same way as any top-level expression. The parser also keeps count of currently open sub-expressions, referencing level of nesting. If the parser encounters the end of the expression, it can only be valid, if this expression exists on the top level. Otherwise, it is considered syntax error because missing closing parenthesis is forbidden.

The parser is also able to detect and report all sorts of syntax errors. For instance, the empty sub-expression is considered invalid. Same goes for also mentioned missing closing parenthesis. Also, every token placed in conflict with formal grammar is considered syntax error, and as such, it will be raised together with information about where has the error been found and what was expected instead. Together with lexer's ability to raise a lexical error, Lambdulus can offer a useful and complete set of hints of what was entered incorrectly and how to fix it.

4.1.3 Abstract Syntax Tree

Parsing step produces AST – recursive data structure represented as a binary tree. This implies that every non-terminal node of AST has two ancestors. This also effectively describes the way Lambdulus represents λ abstraction. In traditional λ-calculus each function is a single-argument function.

4.1.3.1 Application

AST node application represents two expressions applied one to another. Due to the rules of the λ-calculus, left expression, being considered receiver of argument, is applied to the right one, being considered the argument.

```
class Application implements AST, Binary {
  constructor (
    public left : AST,
    public right : AST,
    public readonly identifier : symbol = Symbol()
  )

  clone () : Application

  visit (visitor : ASTVisitor) : void
}
```

Listing 5: Interface of Application AST node

4.1.3.2 Lambda

Lambda node holds two values; the left is of the type variable and is considered a λ function's argument, whereas the right value can be any valid λ expression. As said previously, every λ function is a function of a single argument. This pattern is based on the fact, that when needed, the visual representation layer will be able to represent multiple λ functions standing as the body of the previous ones. This also means, that due to this design this convenient representation is also available for user functions as no steps from Lambdulus's core are needed to represent nested functions as multi-argument functions.

4.1.3.3 Variable

Variable node is the simplest of all the members of AST. It acts as node wrapper for token identifier, and it additionally has utility features to satisfy AST interface.

Variables are not required to start with or contain only lowercase letters. It is however strongly recommended, as it keeps them visually distinguish-

```
class Lambda implements AST, Binary {
  constructor (
    public argument : Variable,
    public body : AST,
    public readonly identifier : symbol = Symbol()
  )

  ...

  clone () : Lambda

  visit (visitor : ASTVisitor) : void
}
```

Listing 6: Interface of Lambda AST node

able from macros, which are in contrast named with first or all characters in uppercase.

```
class Variable implements AST {
  constructor (
    public readonly token : Token,
    public readonly identifier : symbol = Symbol()
  )

  name () : string

  clone () : Variable

  visit (visitor : ASTVisitor) : void
}
```

Listing 7: Interface of Variable AST node

4.1.3.4 Macro

Macro is an expression such as operator or explicitly defined kind of the abstraction; for example, function ZERO would be considered macro. Macro can expand itself into corresponding AST when needed. It is possible because macro holds MacroDefinition object – already parsed definition of macro, in the form of the AST. Macro acquires this definition at the time of its construction. It is used lazily by macro – the definition is not touched until it is needed. Moreover, only when macro needs to be expanded, then this definition is used to substitute the original macro identifier. It is also worth noting,

that definition of one macro can contain another macro inside of it – allowing users to compose more complicated macros out of the simple ones. What is however forbidden is circular referencing of macros – every macro must be able to expand itself in the finite number of steps by simple literal substitution. It is also not possible to use any macro before it was formally defined. This restriction is due to only semantic nature of macros. Lambdulus does not identify macros lexically. Whether the identifier is or is not considered macro, depends solely on the corresponding definition being present or not.

```
class Macro implements AST {
  constructor (
    public readonly token : Token,
    public readonly definition : MacroDef,
    public readonly identifier : symbol = Symbol()
  )

  name () : string

  clone () : Macro

  visit (visitor : ASTVisitor) : void
}
```

Listing 8: Interface of Macro AST node

4.1.3.5 ChurchNumber

ChurchNumber node is a special kind of macro-like value. It represents a single number in the form of Church numeral. When prompted to expand and only then, churchNumber is replaced with full AST structure corresponding to the correct native value of the said number.

While the last three nodes express terminals, each of them can also stand as the root of the whole λ expression. It is worth noting that not every expression can be evaluated per se, but some of them can be expanded, for example, every churchNumber and macro.

```
class ChurchNumber implements AST {
  constructor (
    public readonly token : Token,
    public readonly identifier : symbol = Symbol()
  )

  name () : string

  clone () : ChurchNumber

  visit (visitor : ASTVisitor) : void
}
```

Listing 9: Interface of ChurchNumber AST node

4.1.4 Evaluation of λ expression

As stated earlier, term evaluation is being used to describe the sequence of λ reductions or conversions. For purposes of working with macros and Church numerals, Lambdulus implements additional reduction - expansion. Also, the empty reduction was added. It is called None and serves as a marker that expression is in normal form. Each reduction object is a simple record; it serves solely as a data structure for relevant information about the corresponding λ reduction or conversion.

Lambdulus also implements both *Normal* and *Applicative Order*, defined earlier chapter 1. These are called evaluators; for instance, NormalEvaluator or ApplicativeEvaluator. Due to the isolation of each reduction in the evaluation process, the user can change the evaluation strategy from step to step.

Listing 10 shows an essential part of the primary evaluation method.

As far as each reduction object is only data record, any real transformation happens in so-called reducers. Each reduction has appropriate reducer which implements transformation by traversing AST and changing its structure.

Any reducer returns whole AST every time some transformation is performed. This helps to unify cases where expression currently being transformed is also top-level expression – AST root. This design allows for identical control of evaluation for all cases as seen in listing 10. In the referenced listing, no reduction or reducer can be seen. This is because reductions and reducers are considered low abstraction, so they only appear encapsulated inside relevant evaluation strategy implementations – evaluators.

```
                                    ...
while (true) {
  const normal : NormalEvaluator = new NormalEvaluator(root)

  if (normal.nextReduction instanceof None) {
    break
  }

  root = normal.perform() // new root is returned
}
                                    ...
```

Listing 10: Evaluation control flow

4.1.4.1 Alpha Conversion

The Alpha conversion represents every renaming which will be issued in a single step. Each one renaming references single lambda abstraction, which binds variable occurring freely in the argument. Every alpha conversion is

```
class AlphaConverter extends ASTVisitor {
private converted : AST | null = null
private oldName : string = ''
private newName : string = ''
public readonly conversions : Set<Lambda>

  constructor (
    { conversions } : Alpha,
    public tree : AST
  )

  onApplication(application : Application) : void

  onLambda(lambda : Lambda) : void

  onChurchNumber(churchNumber : ChurchNumber) : void

  onMacro(macro : Macro) : void

  onVariable(variable : Variable) : void

  perform () : void
}
```

Listing 11: Interface of Alpha Converter

issued because of and before some beta reduction, so argument containing unbound or free variables here is meant concerning beta reduction taking place right after earlier referenced alpha conversion. In the specific case, where the free variable in argument value is identical to the name of function argument currently being applied to, no alpha conversion is being issued.

When issuing renaming new non-conflicting name must be created. At the time of writing, Lambdulus creates a new name by prepending an underscore to the original name.

Interface of AlphaConverter in figure 11 demonstrating visitor pattern.

4.1.4.2 Beta Reduction

The beta represents every substitution of single argument value into the λ function's body. Beta reduction object carries the information needed for the corresponding substitution. Parts of that information are the parent node of the application being issued, side of the parent this application occupies, argument name and also argument value related to the substitution. When reduction is being processed, implementation of reducer traverses targeted

```
class BetaReducer extends ASTVisitor {
  private substituted : AST | null = null
  private parent : Binary | null
  private treeSide : Child | null
  private target : AST
  private readonly argName : string
  private readonly value : AST

  constructor (
    { parent, treeSide, target, argName, value } : Beta,
    public tree : AST,
  )

  onApplication(application : Application) : void

  onLambda(lambda : Lambda) : void

  onChurchNumber(churchNumber : ChurchNumber) : void

  onMacro(macro : Macro) : void

  onVariable(variable : Variable) : void

  perform () : void
}
```

Listing 12: Interface of Beta Reducer

AST for occurrences of unbound argument name – unbound inside λ function's body expression – and replaces them accordingly.

Interface of BetaReducer in figure 12 demonstrating visitor pattern.

4.1.4.3 Eta Conversion

The eta conversion, also known as optimisation, transforms λ abstraction to another λ expression from former's body.

Interface of EtaConverter in figure 13 demonstrating visitor pattern.

4.1.4.4 Macro Expansion

The expansion was added to express the transformation of macro or Church numeral value to corresponding AST value. Its reducer – expander implements both transformations. Every macro or the numeric value is expanded right before they are beta reduced. Beta reduction of the expandable identifier

```
class EtaConverter extends ASTVisitor {
  private parent : Binary | null
  private treeSide : Child | null
  private target : AST

  constructor (
    { parent, treeSide, target, argName, value } : Beta,
    public tree : AST,
  )

  onApplication(application : Application) : void

  onLambda(lambda : Lambda) : void

  onChurchNumber(churchNumber : ChurchNumber) : void

  onMacro(macro : Macro) : void

  onVariable(variable : Variable) : void

  perform () : void
}
```

Listing 13: Interface of Eta Converter

is needed for expansion to take place in the evaluation sequence. If macro or number is never going to be applied to its argument, it is never going to be expanded. This only follows rules of λ-calculus and its lazy evaluation. The only exception to this rule is expandable identifier standing as individual expression. Reason for this is to allow the user to create a macro which abstracts λ application and then evaluate the whole program from a single macro.

Exemplary evaluation of a macro expression in listing 14 – consequent steps are read from top to bottom and macros T and F are defined on the right.

```
T T F                          T := (λ t f . t )
(λ t f . t ) T F               F := (λ t f . t )
(λ f . T ) F
T
(λ t f . t )
```

Listing 14: Lambdulus macro evaluation

```
class Expander extends ASTVisitor {
  private expanded : AST | null = null
  private parent : Binary | null
  private treeSide : Child | null
  private target : AST

  constructor (
    { parent, treeSide, target } : Expansion,
    public tree : AST
  )

  churchNumberBody (n : number) : AST

  churchNumberHeader (tree : AST) : AST

  onChurchNumber(churchNumber : ChurchNumber) : void

  onMacro(macro : Macro) : void

  perform () : void
}
```

Listing 15: Interface of Macro Expander

Interface of Expander in figure 15 demonstrating visitor pattern.

4.2 Front-end

Lamdbulus's primary front-end is built as a web-based application. It is built as a completely server-less application running only inside the user's web browser. The user has the option to either instantly use the software or download it from a public repository and build it themselves and use locally.

The main focus was placed on creating a functionally rich but otherwise plain application. Opportunity to enhance the user interface, as well as user experience (UX). Mostly in the process of integrating Lambdulus to the course PPA. The central part of the current UI is displayed in figure 4.1.

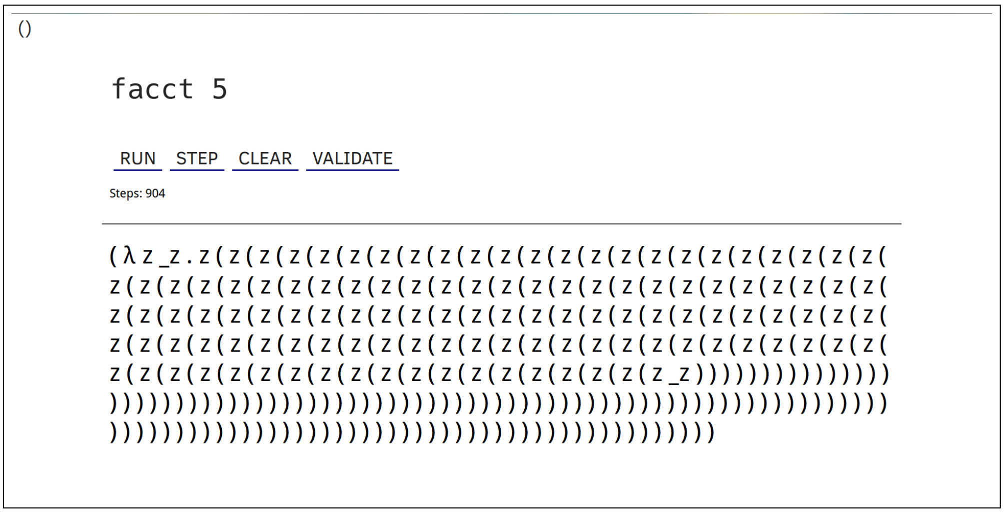

Figure 4.1: Lambdulus

As seen in figure 4.1 central part of the Lambdulus's front-end functionality is an evaluation of λ expression written by the user. The evaluation itself is done by Lambdulus's core module. Front-end's responsibility is to either continually run one reduction after other – in case of *RUN* action, or evaluate step by step – in case of *STEP* action. While step-by-step evaluation is controlled by the user, a continuous evaluation is being controlled by run-time, and thus, it is necessary to employ some additional features for easy control over the evaluation process. For instance, to avoid infinite evaluation of ever-expanding λ expressions, Lambdulus can stop evaluation process at any moment – user pushes button *STOP* which appears once button *RUN* is pushed. This feature is complemented by break-point functionality.

Because many λ expressions have very long evaluation process and many intermediate steps form very long and confusing expressions, Lambdulus supports break-points. User can attach break-point on the β reduction of the

whole function or specific argument by clicking on it. It can also be attached to macro and ChurchNumber. Each break-point can interrupt continuous evaluation. These break-points work in a similar way as the break-points in general debuggers. User can take advantage of skipping many steps and then observe and analyse the step-by-step evaluation of produced λ expression.

In addition to built-in macros, front-end allows the user to define their own macros. As mentioned in chapter 3 This is done in application settings, and the user can construct any complex macro expression.

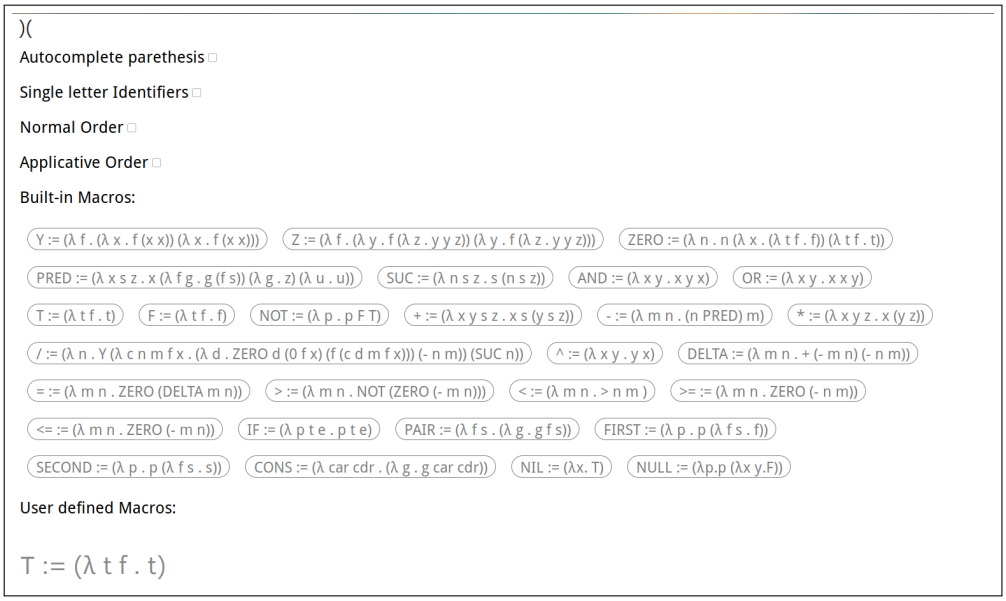

Figure 4.2: Lambdulus – Settings UI

As seen in the top part of Figure 4.2 user customise some parts of the parsing and writing process. Most significant customisation in terms of relevance to the course PPA is an option to enable single-letter-identifiers parsing mode. In this mode, λ expression containing identifiers which are longer than a single letter will be parsed as if each letter was separated by space from others. It is especially useful for quick typing of simple expressions.

In figure 4.3 is demonstrated colour highlighting feature. The green colour is used for λ abstraction, which is going to be applied to its argument, which is highlighted with a pink colour. For the expansion of the macro blue colour is used. Expanding numbers are highlighted with a different shade of blue.

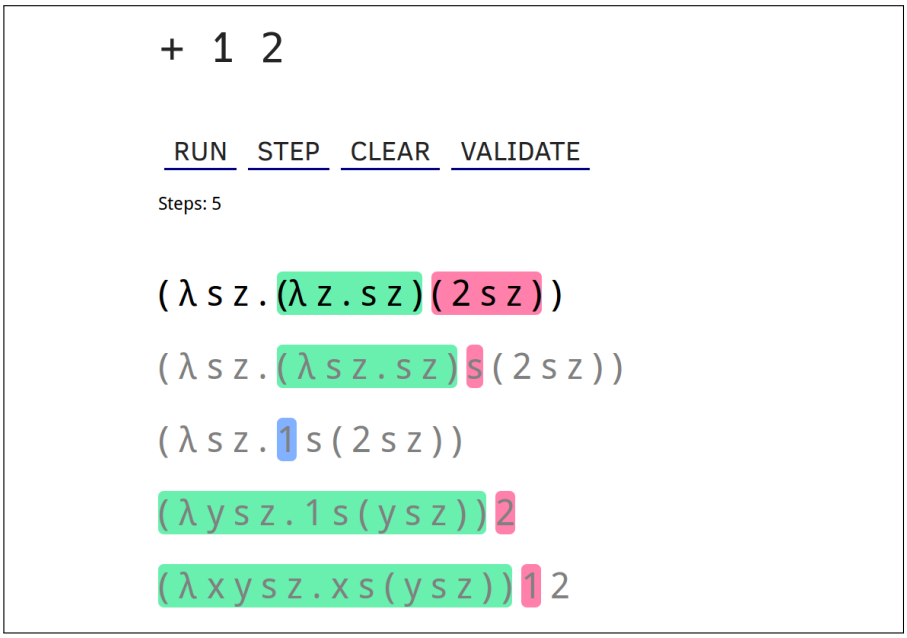

Figure 4.3: Lambdulus – Colouring of Expressions

4.3 Error Detection

For the purpose of analyzing user-inputted λ expression in step-by-step checking, various methods of knowledge engineering were considered. Expert systems were chosen because validating user input and analysing possible mistakes seems to be a problem, which methods of knowledge engineering suit best. The main advantage of expert systems over neural networks, decision trees and others, is their ability to behave in a deterministic way – if they are given the same knowledge base and same answers they produce the same result every time. This is significantly easier to use in these specific conditions.

Validation process proceeds as follows: User prompts validation of their ability to perform a single step of evaluation. User is given input element, types and submits their expression. Input is validated, in case of any difference between user input and referential expression, an expert system is being used to decide which mistakes were possibly made. The user is then informed about the result of the analysis – possible mistakes and reasons leading to their expression being invalid.

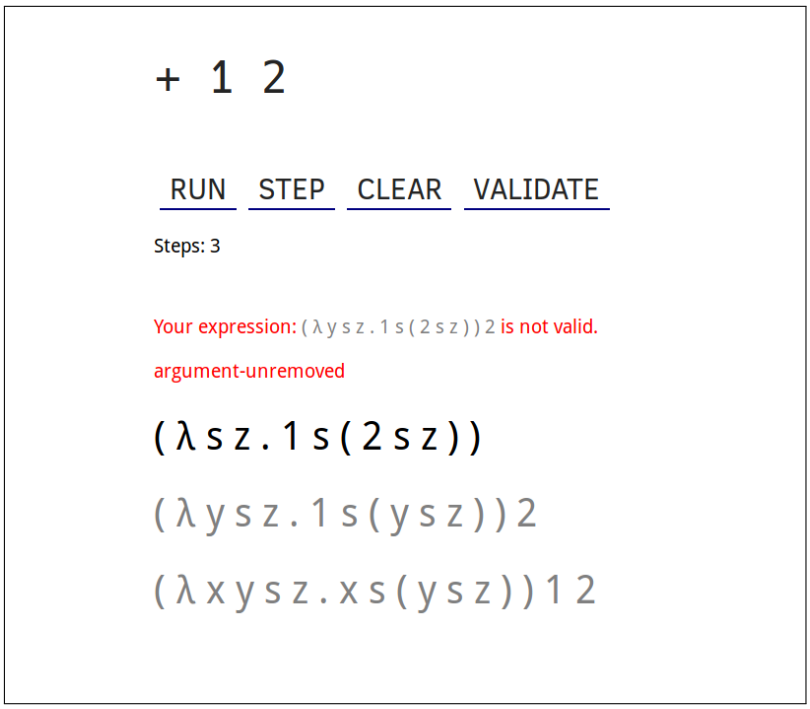

Figure 4.4: Lambdulus – Error Detection

The knowledge base needs to be defined clearly and effectively. The knowledge base used in this thesis is defined as JavaScript Object Notation (JSON) because it is easily parsed and also readable by people. Its structure looks as follows.

```
[{
  "name" : "tried-beta-instead-of-alpha",
  "type" : "dis",
  "predicates" : [
    "correct-beta-instead-of-alpha",
    "incorrect-beta-instead-of-alpha"
  ]
},
{
  "name" : "ignored-beta-reduction",
  "type" : "con",
  "predicates" : [
    "should-do-beta",
    "old-same-or-equivalent-to-user"
  ]
}]
```

Listing 16: Knowledge Base structure

In figure 16 are displayed two exemplary rules of the knowledge base. The entire base is the sequence of rules similar to those shown. Each rule must be named – name will be used as an outputted result in case the rule is satisfied. To satisfy the rule system must evaluate the sequence of queries called predicates. Each sequence specifies the relation between sole predicate by setting the type of the rule. Each rule stands either as disjunction – type *dis* or conjunction – type *con*.

The predicate in the sequence can be either primitive or name of another rule. In the former case expert system directly queries front-end of Lambdulus by implemented Application Programming Interface (API). The queried part of the Lambdulus traverses AST of the last valid expression, next valid expression, and the user-submitted expression and resolves the query as either *true* or *false*. Besides the Boolean value, each resolution adds a list of warnings or errors in case any of them have occurred. They help to further identify possible mistakes done by the user. As mentioned earlier predicate can be a reference to another rule. In that case, the expert system effectively queries itself. This scenario comes with the only limitation – it is not allowed to nest rules recursively, other than that there is no limitation of the depth of nested rules.

In case it is needed, it is permitted to create a rule with a single predicate. Such rule can serve as named primitive predicate and can also carry the same name as one.

The structure for the rule is noticeably flat. More semantical structures were also considered, but any intricacy comes with the undesired effect of

worse readability. The chosen structure allows for comfortable observation and easy modification. For example, adding the rule identifying that user ignored any reduction can be made like so.

```
{
  "name" : "ignored-beta-reduction",
  "type" : "con",
  "predicates" : [
    "should-do-beta",
    "old-same-or-equivalent-to-user"
  ]
}
```

Listing 17: Rule Ignored-Beta-Reduction

The corresponding handler to newly created primitive predicate must also be implemented by Lambdulus.

The core of the expert system is backwards-chaining inference engine. It iterates over all rules in the knowledge base and tries to prove one rule at the time. In this specific case of the expert system, there is no need for implementing the priority of the rules. Order of the rules in the knowledge base should not influence the result in any way. Reason for this is the fact that as an expert system is not querying human, it is not important to prefer any specific order. The expert system might implement no store for answers as well. That is true only because querying API is functionally pure by design – every query resolves to the same answer when given the same combination of three instances of AST. For that matter, any storage for answers would serve only as optimisation.

Chapter 5

Evaluation

The main objective of this thesis was to create λ evaluator suitable for use in course PPA. An important part of the integration of the Lambdulus to the mentioned course is the modification of current study materials on Course Pages. To evaluate the level of satisfaction of course's specific demands, author of this thesis edited teaching material for equivalent course BIE-PPA – the same course, but taught in the English language.

While each exercise on λ-calculus was being processed, Lambdulus proved itself to be a useful tool even for correcting typical mistakes done by teachers when preparing materials. In this process, several mistakes were discovered and corrected.

- Missing single or multiple parenthesis

  ```
  Original:  (λx y . (-x y) 5 2

  Corrected: (λx y . (-x y)) 5 2
  ```

- Incorrectly used precedence

  ```
  Original:  (λx. (* + 3 x - x 4)) 5

  Corrected: (λx. (* (+ 3 x) (- x 4))) 5
  ```

- Possibly missing argument or otherwise malformed expression

  ```
  Original:  (λx. (λy. * + y z - x 4) 5) 2

  Corrected: (λx. (λy. (λz. * (+ y z) (- 4 x))) 5) 2 4
  ```

- Infix instead of prefix

```
Original:  ((x or y) and (not (x and y)))

Corrected: (AND (OR x y) (NOT (AND x y)))
```

- Missing λ symbol

```
Original:  (p.(λq.(λp.p (p q))(λr.+ p r))(+ p 4)) 2

Corrected: (λp.(λq.(λp.p (p q))(λr.+ p r))(+ p 4)) 2
```

This proves that Lambdulus, as a λ evaluator, functions well even for quick checking of the validity of arbitrary λ expression.

To integrate Lambdulus into materials, a teacher must only copy URL of Lambdulus expression – which is automatically updated while typing and paste this link to teaching materials online. When clicked, this link will open prepared λ expression inside Lambdulus. Students can observe and learn about the evaluation of this expression without any preparation from their side.

Conclusion

The aim of this thesis was to develop Lambdulus, an interactive λ-calculus evaluator for the course PPA at FIT CTU. Lambdulus fulfils all the requirements of the course and has already been integrated into the teaching materials.

Lambdulus is built prioritising student's needs and taking into account other already existing evaluators. It consists of two separate modules to ensure future extensibility. The core module's code is written as an illustration and teaching aid for explaining the simplicity of implementation of λ-calculus interpreter to students. As such it has been designed with ease of use in mind and is well documented. The core of Lambdulus is extremely minimal step-by-step λ expression evaluator which can easily be extended with different λ-calculus dialects, or extra features (different evaluation orders, etc.) It is also intended to be used as a demonstration of the simplicity and power of the simple λ-calculus; all complex features, such as multi-variable λ functions are simplified into the most basic forms, and great emphasis has been placed on the readability of the code, including full documentation. The front-end module is built on top of the core evaluator and provides the visualization and user interface layer that in terms of workflow and usability brings λ-calculus much closer to practical programming languages. The front-end supports clear visualization of λ-calculus evaluation steps and proper debugging of λ expressions. It also features an error checking mode where the student attempts their own step-by-step solutions, and Lambdulus corrects their mistakes offering hints to the errors made. Finally, the front-end has features for simple sharing and embedding in existing course materials via state-of-the-art web methods.

The whole code-base of *Lambdulus* is on the Github and application is accessible online at this address: https://lambdulus.github.io It is easy-to-deploy on any type of web hosting because it is a client-only static web application.

In the moment of the writing, the author and his advisor are in the process of preparation for submitting the paper on the SPLASH-E conference.

Future work

For the future work on this thesis, its author was given the grant for the integration of Lambdulus to the PPA's teaching materials and to improve the user interface and experience. This section discusses the work that will be done as a fulfilment of this grant as well as more general improvements to be done in a more distant future.

- more evaluation strategies

  Optimized evaluation of arithmetical and logical expressions will be implemented.

- usability testing and refinement of UI

  The user interface of the web front-end application will be refactored to become more intuitive and easy to use.

- meaningful and semantic renaming when α converting

  Lambdulus will find an optimal and unique name for substitution. This name should correspond to some convention and make α converted λ expression more readable.

- reverse macro expansion

  Macros and numbers will be able to reverse-expand back to their abstract form. This will simplify the results of mathematical and logical operations and make them more readable.

- better visualization and debugging

  Visualisation and debugging are going to be significantly improved. Better control over evaluation and debugger with more features will make understanding of the evaluation process more clear and intuitive. Additional functionalities such as conditional break-points will give users much more control over the evaluation process. Also, highlighting the expression with colour combinations will be implemented. User will be able to click on any argument of the function, and every occurrence will be highlighted, offering a better way of orientation.

- operator precedence

  To make λ expressions even more easy to read and type, Lambdulus will implement operator precedence as known from math. User will be able to write expressions such as: $+ * 3\ 2 * 2\ 2$ being effectively equivalent to: $+ (* 3\ 2) (* 2\ 2)$

Bibliography

1. *Syntax of the λ-calculus* [online]. 2019. Available also from: `https://courses.fit.cvut.cz/BIE-PPA/tutorials/lambda-1.html`. [cit. 2019-4-20].
2. *Reductions* [online]. 2019. Available also from: `https://courses.fit.cvut.cz/BIE-PPA/tutorials/lambda-2.html`. [cit. 2019-4-20].
3. PLOTKIN, G.D. *Call-by-name, call-by-value and the λ-calculus*. 2nd ed. Theoretical Computer Science Volume 1, 1975. Available from DOI: `10.1016/0304-3975(75)90017-1`.
4. LYNN, Ben. *Lambda Calculus* [[web application]]. Available also from: `https://crypto.stanford.edu/~blynn/lambda` [cit. 2019-4-20].
5. GONG, Liang. *λ Calculus Interpreter* [online]. Available also from: `https://jacksongl.github.io/files/demo/lambda/index.htm`. [cit. 2019-4-20].
6. BURCH, Carl. *Lambda Calculator* [online]. 2019. Available also from: `https://capra.cs.cornell.edu/lambdalab`. [cit. 2019-4-20].
7. *Lambdalab* [[online]]. 2012–2019. Available also from: `http://www.cburch.com/lambda`. [cit. 2019-4-20].
8. MICROSOFT CORPORATION. *TypeScript* [[online]]. 2012. Available also from: `https://www.typescriptlang.org/`. [cit. 2019-5-12].
9. NETSCAPE COMMUNICATIONS CORPORATION; MOZILLA FOUNDATION; ECMA INTERNATIONAL. *JavaScript* [[online]]. 1995. Available also from: `https://www.ecma-international.org/publications/standards/Ecma-262.htm`. [cit. 2019-5-12].
10. STROUSTRUP, Bjarne. *C++* [[online]]. 1985. Available also from: `https://isocpp.org/`. [cit. 2019-5-12].
11. FACEBOOK, INC. *React.js* [[online]]. 2013. Available also from: `https://reactjs.org/`. [cit. 2019-5-12].

12. SCOWEN, Roger S. Extended BNF — A generic base standard. In: [[online]]. 1998. Available also from: `https://pdfs.semanticscholar.org/ed89/e6f749768cc4fc585e6ef406afeace436a19.pdf`. [cit. 2019-5-12].

13. BURGE, W.H. *Recursive Programming Techniques*. 1975. ISBN 0-201-14450-6.

Appendix A

Acronyms

FIT CTU Faculty of Information Technology at Czech Technical University in Prague

PPA Programming Paradigms

AST Abstract Syntax Tree

UI User Interface

UX User Experience

EBNF Extended Backus-Naur form

URL Uniform Resource Locator

Appendix B

Contents of enclosed CD

- `readme.md` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `core` ..... the directory of source codes of core module
    - `dist` ..... the directory of compiles JavaScript source code
  - `frontend` ..... the directory of source codes of front-end module
    - `build` ..... the directory of built static web-site
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format