Online Lambda Calculus Evaluator

D R A F T, april 29 2018. uroven koherencie: okolo 50 %.

Contents

1 Introduction

Lambda calculus is a formal system of logic, concerned with functions described as substitution-based evaluation rules, invented by Alonzo Church in the late 1920s. It plays an important role in computer science and mathematics, with applications in other fields, such as linguistics and philosophy.[5]

The aim of this thesis is to design and implement a web application that evaluates user-input lambda calculus terms, both pure and typed, with a possibility of selecting a particular evaluation strategy and step-by-step evaluation, multiple importing and exporting options, and several other features lacking in publicly available preexisting solutions. The application should be useful to students of the course *IA014 – Advanced Functional Programming*, as it is intended to aid in the understanding of fundamental principles of lambda calculus taught therein.

Firstly, a general overview of the importance of λ-calculus will be presented, including, to some extent overlapping topics of historical significance and real-world applications—especially contributions to the theory of programming languages, then a rigorous definition of lambda calculus' syntax and semantics, together with descriptions of the implemented evaluation strategies, type systems, and extensions will be provided, and lastly, a thorough description of the evaluator web application itself will be given. In the appendix, one can find examples of some of the possible inputs to the application, along with an enumeration of pre-defined aliases and constants, potentially serving as a reference manual.

1.1 Historical background

"Wir müssen wissen – wir werden wissen!"[1] were the well-known words of David Hilbert, countering the appliance of the Latin saying *"Ignoramus et ignorabimus"*[2] to the natural sciences, and expressing his belief that any mathematical problem posed in an appropriate formal language can be solved, preferably by a mechanical means. In 1928 Hilbert stated the challenge, titled *Entscheidungsproblem*[3], as a search for an algorithm deciding whether any first-order logic formula passed as an input can be proven, given a finite set of axioms, using the inference rules of the logic system.[10] In order to prove or disprove an existence of such an algorithm, there was a need to formalize the intuitive notion of an algorithm itself first.

Turing machine, originally named *a*-machine (automatic machine), one of such proposed models of computation, is an abstract machine devised to encapsulate the intuitive notion of effective computability. It generalizes a computer—a person mindlessly following a finite set of given instructions *"in a desultory manner"*—by introducing a device of infinite sequential memory, finite table of instructions, and a state the device is in. The instructions determine what to write into the current memory cell based on its content and of the state of the machine, how to change the state, and what adjacent memory cell to read next.[20]

Adopting his earlier work on the foundation of mathematics, Alonzo Church has taken a different approach; the lambda calculus is defined as a system of nameless substitution-based functions with no explicit state or external memory, and due to its simplicity and expressivity has become a successful model of computation, with a multitude of contributions to the theory of programming languages.

[1] German for "We must know—we will know!"

[2] Latin for "We do not know and we will not know"

[3] German for "Decision problem"

Turing has shown that Turing machine and lambda calculus are both equally powerful formalizations, i.e. that the classes of computable functions defined by each of them are the same.[19] Assuming that functions on natural numbers computable by a human with limitless resources are the same as functions definable by Turing machine *(Turing-Church conjecture)*, the answer to the Entscheidungsproblem was found to be negative—no algorithm deciding on a solution of any problem posed in a formal language can exist—by both Turing and Church independently in 1936.[20][7]

There are a number of conflicting accounts of the origin of Church's choice to use the Greek letter λ. Barendregt argues that such usage is a result of a typesetting mistake, erroneously replacing circumflex (^), then already used as a class abstraction operator in Russel's and Whitehead's *Principia Mathematica*, with lambda, similar in appearance.[3] Dana Scott, a PhD student of Church's and a Turing Award laureate, opposes this hypothesis and claims the lambda letter was chosen arbitrarily.[17]

1.2 Real-world applications

Lambda calculus is regarded as a conceptual backbone of the functional programming, usually described as based on an evaluation of expressions, as opposed to the usual, contemporarily more popular approach of the imperative programming, based on a consecutive modification of a program's state.[4] A parallel between Turing machines (together with the underlying von Neumann architectures) and imperative languages, and between λ-calculus and functional programming languages can be drawn.[2]

lisp, r.milner a ML,

Anonymous functions One of the most conspicuous of contributions are *lambda expressions*, commonly referred to as *anonymous functions*—function definitions not bound to a name as the usual function/procedure definitions. First appearing in LISP back in 1958, anonymous functions are ubiquitous in functional programming languages, and recently[5] swiftly became popular in mainstream imperative languages. Anonymous functions are useful to create short functions relevant only to a specific local scope, without affecting the global namespace, or when passing a function as an argument or constructing a functional return value.

For instance, lambda expressions realized in several languages, defining a function comparable to a (named) mathematical function $f(x,y) = (x+y)\cdot y$, are as follows:

λ-calculus	$\lambda xy.\mathrm{TIMES}\,(\mathrm{PLUS}\,x\,y)\,y$
LISP	`(lambda (x y) (* (+ x y) y))`
Haskell	`\x y -> (x + y) * y`
C++11	`[](auto x, auto y) { return (x + y) * y; }`
Java SE 8	`(int x, int y) -> return (x + y) * y`

Higher-order functions A concept essential to λ-calculus, higher-order function is a function taking another function as an input, or returning function as a result. For example, the commonly used mathematical function composition operator $\circ$, defined as $(f \circ g)(x) = f(g(x))$, can be understood as a (named) higher-order function, accepting two functions as an input, and returning a third as an output. example

[4] This dichonomy is somewhat simplified, as the usual counterpart to the imperative paradigm is considered to be the declarative programming paradigm, the functional being a part thereof.

[5] For instance, lambda expressions were introduced in 2011 version of C++ and 2014 version of Java.[12][8]

Partial application

Type derivation

2 Lambda calculus

2.1 Pure lambda calculus

The original presentation of λ-calculus, without any extensions as *Let*-expressions, constants, type systems, etc., is called pure. It is worth mentioning that the referenced extensions do not add to the expressivity of the pure system, and they can even impose additional restrictions—the pure λ-calculus is already computationally universal – Turing-complete.

2.1.1 Term syntax

During the history, there have been minor variations in term syntax used by various authors, each with their respective syntactical abbreviations, for example $\{\lambda x[\{x\}(\{y\}(z))]\}(w)$,[7] $((\lambda x((x\,(y\,z))))\,w)$,[21] or $((\lambda x.(x\,(y\,z)))\,w)$,[11] which do all represent the same term. In this work, the grammar used the most frequently in the contemporary literature, coinciding with that presented in the course IA014[14] will be adopted, along with the common abbreviation methods (§ 2.1.2).

In the following definitions, the symbols M or N represent some terms, and, depending on the context, the variable x stands for any variable. The language of λ-calculus, called Λ (capital lambda), is generated by the following grammar, written in Backus-Naur form, consisting of only three production rules:

M	::=	x	*Variable*
	|	$(M\,M)$	*Application*
	|	$(\lambda x.M)$	*Abstraction*

Generally, a *variable* (a set of which will be written as VAR) could be any unambiguous identifier that is a member of a chosen countable set, but in this work, the set of variables is restricted to lowercase characters optionally followed by a numerical subscript, as the general case could interfere with the syntactic conventions of which the definition is given in the following section 2.1.2. Thus an equivalent, but more minimalistic definition is given by $\Lambda = (\Lambda\Lambda) \mid (\lambda\text{VAR}.\Lambda) \mid \text{VAR}$, where $\text{VAR} = \text{lowercase character} \mid \text{lowercase character}_{n\in\mathbb{N}}$. The set of terminals consists of the elements of VAR, lambda character, and parentheses. Some of the syntactically correct words are w_{32}, $(((l\,e)\,t_9)\,(i\,n))$, or $((\lambda o_3.f)\,p)$.

Throughout this work, some parts of term may be referred to by the following names: an *argument* for the right-hand side of an application, a *function* for a term that is an instance of the abstraction rule, a *parameter* of a function for the abstracted variable, and *function body* for the abstraction's term.

x

$$\overbrace{(\lambda \underbrace{w}_{\textit{parameter}}.\underbrace{(w\,w)}_{\textit{function body}})}^{\textit{function}}\ \overbrace{(\lambda o.(o\,o))}^{\textit{argument}}$$

2.1.2 Syntactic conventions

To make terms more succinct and convenient to write and read, the following syntactic conventions, describing parentheses omission, are being used:

$$\lambda x_1 x_2 x_3 \ldots x_n.M \equiv (\lambda x_1.(\lambda x_2.(\lambda x_3.(\ldots(\lambda x_n.M)\ldots))))$$
$$M_1\, M_2\, M_3 \ldots M_n \equiv (\ldots((M_1\, M_2)\, M_3)\ldots M_n)$$
$$\lambda x.x_1\, x_2 \ldots x_n \equiv (\lambda x.(x_1\, x_2 \ldots x_n))$$

The intuition behind the first two rules is rather straightforward, as the rules seemingly emulate the process of *currying*, i.e. a translation between function of multiple arguments *(left-hand side)* and multiple functions of a single argument *(right-hand side)*.[15] Clearly, this is only a matter of syntactic simplicity and has no effect on the actual semantics. The third rule necessitates a precedence of the application rule over the abstraction rule.

Terms can also be given a name—the choice of term names, or *aliases*, is restricted to strings of uppercase letters, and strings of numerals. In this work, the aliases are set in roman type. The means of assigning terms to numeral names is described in section 2.1.6 – Church encoding.

2.1.3 Variable binding and substitution

The substitution of one term for another lies at the very heart of λ-calculus' evaluation mechanics, as the idea of evaluating a function is expressed via replacing abstraction-bound variables with the provided argument. A variable x is bound in M, if occurs as an abstraction variable $\lambda x.M$. The lambda symbol is also called an abstraction operator, and M the scope thereof. In order to define reductions and reduction strategies later, first the concept of free and bound variables must be introduced formally, using inductive function $\mathrm{FV} : \Lambda \to \{\,\mathrm{VAR}\,\}$, mapping terms to the respective sets of free variables in the following manner:[21]

$$\mathrm{FV}(x) = \{x\}$$
$$\mathrm{FV}(M\, N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$$
$$\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) \setminus \{x\}$$

For example, $\mathrm{FV}(\lambda xyz.x) = \{\}$, $\mathrm{FV}(\lambda e.y\ e) = \{y\}$, and $\mathrm{FV}\big((\lambda w.w)\ w\big) = \{w\}$. Some variable x occurring in a term M is said to be bound, if it is not a member of the set of free variables of the term M. A term M, such that $\mathrm{FV}(M) = \{\}$, is called *combinator*.

Although apparently trivial, the substitution of free variables of some term with an another term, the result being written as $M[x := N]$, can be precarious, for some of the free variables of term N can become bound in the resulting term. A simple illustration of this phenomenon is the following function, $\lambda xy.x$, that could be understood as a function taking two parameters and returning the first. When this function is partially applied to a particular argument, $(\lambda xy.x)\ y$, a naïve replacement of the bound variable x for the argument y would yield $\lambda y.y$, that is, the identity function, a very different result from the expected function returning the first, already applied parameter y, for any argument. The solution to this problem leads to renaming variables in the process of a non-capturing substitution.

$$
\begin{aligned}
x[x := N] &\equiv N \\
y[x := N] &\equiv N, \ \textit{where}\, x \neq y \\
(M_1\, M_2)[x := N] &\equiv (M_1[x := N])\,(M_2[x := N]) \\
(\lambda x.M)[x := N] &\equiv \lambda x.M \\
(\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]), \ \textit{where}\, x \neq y,\ x \notin \mathrm{FV}(M) \vee y \notin \mathrm{FV}(N) \\
(\lambda y.M)[x := N] &\equiv \lambda s.(M[y := s][x := N]), \ \textit{where}\, x \neq y,\ x \in \mathrm{FV}(M) \wedge y \in \mathrm{FV}(N)
\end{aligned}
$$

in the last rule, the variable s stands for such a variable y_n,
where n is the least natural number such that $y_n \notin \mathrm{FV}(M) \wedge y_n \notin \mathrm{FV}(N)$

2.1.4 Reductions

A single evaluation step, expressing the notion of function application, is formalized via the mechanism of so-called *β-reduction*, consisting of substitution of all abstraction-bound variables with an application argument. In this work, the terms *evaluation* and *reduction* are used interchangeably. The formal β-reduction axiom is as follows:

$$(\lambda x.M)\, N \ \rightarrow_\beta \ M[x := N]$$

The term of the form $(\lambda x.M)\, N$ is called a *β-redex* (*reducible expression*), or, if not causing any ambiguity, simply a *redex*. As a redex can appear anywhere inside a term, a definition of a non-deterministic *full beta reduction*, enabling reduction of reducible expressions occurring deeper inside a term, and in any order, is given by the following rules, written using common notation for inference rules with premises above the line, and a conclusion below the line:

$$\frac{}{(\lambda x.M)\, N \ \rightarrow_\beta \ M[x := N]} \qquad \frac{M_1 \ \rightarrow_\beta \ M_2}{\lambda x.M_1 \ \rightarrow_\beta \ \lambda x.M_2}$$

$$\frac{M_1 \ \rightarrow_\beta \ M_2}{M_1\, N \ \rightarrow_\beta \ M_2\, N} \qquad \frac{N_1 \ \rightarrow_\beta \ N_2}{M\, N_1 \ \rightarrow_\beta \ M\, N_2}$$

Simply put, employing the full β-reduction, a redex can be reduced (top-left), redex can be evaluated inside an application (bottom), and finally, it can be evaluated inside an abstraction (top-right). A term that cannot be β-reduced any further is said to be in *β-normal form*.

called η-reduction (or η-conversion), enabling a contraction of terms

$$\frac{x \notin \mathrm{FV}(M)}{(\lambda x.M\, x) \ \rightarrow_\eta \ M}$$

church rosser, normal forns

2.1.5 Reduction strategies

Reduction strategy is a way of choosing which reducible expression, possibly out of many, to reduce. The two classes of reduction strategies are considered: a *strict evaluation strategy* (or *eager* evaluation strategy) does always perform an evaluation

of arguments, regardless whether actually used inside the function's body. On the other hand, a ***non-strict evaluation strategy***, or sometimes, *lazy* evaluation strategy, is a strategy evaluating arguments only if they are used in the function body.[15] Most programming languages employ some kind of strict evaluation strategy—this is especially true with regard to imperative languages, freely manipulating side-effects. A prominent example of a language that is employing a non-strict evaluation strategy is Haskell, as its purity[6] allows for such a usage.

These two `C` functions are presented as an example:

```
int hello(int parameter)
{
   printf("Hello World!");
   return 41;
}
```

```
int fac(int n)
{
   if (n == 0) return 1;
   else return n * fac(n - 1);
}
```

In the function call `hello(fac(13))`, the argument of the function `hello` is not used inside the function's body, in this case rendering the computation of `fac(13)` futile. Non-strict valuation strategy would replace all of the occurrences of `parameter` inside the body of `hello`, however, because there are none, the call would immediately produce a result of `41`. The unsuitability of a non-strict evaluation in imperative languages is clear in this very example: `hello(hello(0))` would (in this case) yield the anticipated result `41`, but only one of the two expected `Hello World!` greetings would be printed. Nonetheless, non-strict evaluation strategies can be greatly beneficial, improving performance (by possible minimization the number of reductions), and allowing for an admission of non-terminating arguments.

Using a strict evaluation, a computation of the following term would never terminate, as the argument itself is not terminating:
$(\lambda x.w)\ \underline{((\lambda r.r\ r)(\lambda r.r\ r))} \to_\beta (\lambda x.w)\ \underline{((\lambda r.r\ r)(\lambda r.r\ r))} \to_\beta (\lambda x.w)\ \underline{((\lambda r.r\ r)(\lambda r.r\ r))} \to \ldots$, but conversely, a non-strict evaluation reaches normal form of this term immediately:
$\underline{(\lambda x.w)\ ((\lambda r.r\ r)(\lambda r.r\ r))} \to_\beta w$. This behavior is often taken advantage of when programming in languages that use a non-strict evaluation strategy, e.g. Haskell or slightly older Miranda.

Besides the already defined ***full β-reduction***, non-deterministically selecting any of the reducible expression, these four other, deterministic evaluation strategies are implemented:

Normal order Normal order is a non-strict evaluation strategy, selecting the left-most, outermost of redexes for evaluation. This strategy is *complete*, that is, if a normal form does exist, it will be reached.

Call by name non stricc

Applicative order stricc

Call by value This strict evaluation strategy requires an introduction of the notion of *values*, that is, a subset of terms considered to be irreducible any further, with the computation on them being finished. In the pure λ-calculus, the only values are the

[6] Purely functional programming languages ensure that the output of functions is dependent only on their input, in contrast with results being also dependent on mutable data or global state of a program. For more information about Haskell's evaluation method see `https://wiki.haskell.org/Lazy_evaluation`.

instances of the *abstraction* rule, later on, however, the set of values will be extended by primitive values, such as numeric or logical constants, etc. (section 2.1.8). The set of values is written as V.

$$V \quad ::= \quad (\lambda x.M) \qquad \textit{Abstraction}$$

2.1.6 Church encoding

Church encoding is a means of representing data, and operation on them, using only pure lambda calculus. Because of the simplicity of the evaluation rules, the powerfulness and expressiveness, with which the data and operations are encoded, may be unexpected. not good - rewrite

Church Booleans and logic operators Foremost, the definitions of the Boolean values can be understood as binary functions, returning the first of parameters in a case of true, and the second in a case of false value, TRUE $\equiv \lambda xy.x$, and FALSE $\equiv \lambda xy.y$. After the Boolean values are established, the definition of the fundamental logic operators is desired. It is required that OR $M\,N$ *(logical disjunction written in prefix notation)* will reduce to TRUE if M and N are Church-encoded Boolean values, or could be reduced into them, and at least one of the terms M and N can be reduced to TRUE. Because, in pure lambda calculus, there is no guarantee of validity of the operands, there is no way of predicting the result of an application to an incorrect argument. (Later, the problem will be solved by typing the expressions in section 2.2 – Typed lambda calculi). A term adhering to such requirements is:

$$\text{OR} \equiv \lambda yx.y\ y\ x$$
$$\text{and similarly, AND} \equiv \lambda yx.y\ x\ y$$

Another useful construct is a conditional, here expressed as a ternary function of which the first argument is the Boolean condition itself, returning one of the remaining arguments based on the condition. The definition of ITE *(if—then—else)* is as follows:

$$\text{ITE} \equiv \lambda xyz.xyz$$

This term makes use of the property of TRUE and FALSE returning the first, respectively the second, of the two arguments.

Church numerals and arithmetic operators

2.1.7 Recursion

2.1.8 Implemented extensions and syntactic sugar

Constants For the sake of efficiency, convenience, and clarity, the constants, the set of which is written as $\mathbb{C}$, are introduced as an extension of the Λ. Constants can represent primitive values, such as numbers or Booleans, as well as functions on terms (including constants), for instance, the arithmetic multiplication or the logical disjunction. Operational semantics is defined via a set of reduction rules, called δ-rules.[2] The set of *values*, V, is extended with constants.

M	::=	x	*Variable*	V	::=	$(\lambda x.M)$
	|	$(M\ M)$	*Application*		|	$C \in \mathbb{C}$
	|	$(\lambda x.M)$	*Abstraction*			
	|	$C \in \mathbb{C}$	*Constant*			

An example of δ-rules for logical disjunction defined for constants:

$$\begin{aligned} \mathit{OR\ FALSE\ TRUE} &\rightarrow_\delta \mathit{FALSE} \\ \mathit{OR\ FALSE\ FALSE} &\rightarrow_\delta \mathit{FALSE} \\ \mathit{OR\ TRUE\ TRUE} &\rightarrow_\delta \mathit{FALSE} \\ \mathit{OR\ TRUE\ FALSE} &\rightarrow_\delta \mathit{FALSE} \end{aligned}$$

In this work, the constants are written as strings of uppercase letters set in italic type. The exhaustive list of pre-defined constants can be found in the appendix.

***Let* expressions** In pure λ-calculus, and later also in simply typed λ-calculus, the *let* expression is a syntactical shorthand, enabling more convenient definitions of functions.

2.2 Typed lambda calculi

progress theorem

2.2.1 Simply-typed lambda calculus

2.2.2 Hindley-Milner type system

3 Evaluator

The purpose of the implemented Lambda Calculus Evaluator, further simply *the application*, is to evaluate λ-calculus terms input by a user. The application provides functionality to choose an evaluation strategy, toggle between different modes of term-display, including rendering aliased terms as the alias name, or the term it represents, and between full-parenthesized terms over the shorthand form, as presented in section 2.1.2 – *Syntactic conventions*. An effort was made to develop an ergonomic, easy-to-use application with simple yet appealing visuals.

Besides the pure, untyped λ-calculus, a user can also select one of the two type systems, *Simply-Typed lambda calculus* and *Hindley-Milner type system*, described in section 2.2, which halts the execution if the term is typed incorrectly or the type cannot be deduced, effectively stopping an evaluation of inconsistent terms.

3.1 Technical aspects

Using the three core web development technologies, HTML *(Hypertext Markup Language)*, CSS *(Cascading Style Sheets)*, and JavaScript, the evaluator is built as a single-page application, i.e. an application using solely dynamic modifications of the content of the current page, as opposed to redirecting, reloading, or downloading a page with a new content. Using a web browser as an application platform contributes to the application's compatibility and availability, as there is no need for an installation on the target system. The application is executed exclusively on the client-side, in case of an actual deployment alleviating the server load on one hand, allowing the downloading and subsequent offline usage by a user on the other.

To tightly control the application's typography across many different systems, the *Computer Modern* typeface, licensed under free and open-source SIL Open Font License,[7] is distributed along the application, allowing for consistent user experience and arguably æsthetically pleasing visual appearance of the application.

Object-oriented programming paradigm, supported by JavaScript, is employed to model the data, as *objects* can closely match the term structure, e.g. a *term* can be represented by an interface, an *abstraction* can be an implementation thereof, closely matching the lambda calculus' grammar. Due to the JavaScript's dynamic nature, the mentioned interface is only implicit, as there is no syntactical support for such a construct in the language itself.

libraries & frameworks

3.2 Features and respective user interfaces

The application interface is structured, besides the *Help* page, as two screens: *initial screen*—the screen used to accept user input and to set preliminary options, e.g. a type system, and the *main screen* for evaluation of the input terms, together with the relevant options and information presentation.

info bar:sentence, help page figs.

3.2.1 User's input

Upon entering the application, the user can immediately begin to type the expression to be evaluated. The outline of the input box is colored based on the syntactical

[7] Available at `http://scripts.sil.org/OFL`.

correctness of the input term; ■ red on syntactically incorrect, and ■ green on syntactically correct terms, providing a useful and immediate feedback to the user after each keystroke. As per the information shown on the entry page, in order to enter the lambda character, the user simply enters the backslash key, which in turn gets automatically converted into the lambda character, written into the input box. The type constructor arrow for the simply typed λ-calculus is input in a similar fashion, using a succession of a hyphen and a greater-than sign.

The entry page (the *initial screen*) is also equipped with links potentially useful to newcome users, specifically one to the *Help* page, and one specifically to the *Examples* section. On the initial screen, there is also an option to load a previously saved file. A closer look on available import and export options is given in section 3.2.6.

Aliases and constants can be used for input, including Church-encoded numerals up to three digits. An option to disambiguate between conflicting alias- and constant names is available, delivering interpretation of the names accordingly to the preference set by a user. The default behavior is to interpret all conflicting names as constants. For instance, the user inputs a string "21": such a string could represent Church-encoded numeral as well as the primitive value—a constant defined outside the λ-calculus. Considering the user did not change the default setting, this string will be parsed as a constant of twenty-one. Occasionally, one may wish to enter both an alias and a constant. In such case, one can use an exclamation mark just before the conflicting name to indicate the exception to the preference setting. Again assuming the default setting, a string "!FALSE 12 TRUE" would be interpreted as $(\lambda xy.y)$ *12 TRUE*, the constants being written in italic type.

Because of the memory-intensive generation of Church-encoded numerals, a user can input such numerals only up to three digits. Displaying the terms on the screen is done by stack-based recursive procedure; rendering large enough numerals, and generally, any terms, will cause a stack overflow.

A top-down recursive descent parser is used to translate the user's input text into an internal data structure: for each of the production rules of the grammar there exists a procedure deciding whether the currently parsed text is generated by the respective production rule. The expected input is short (dozens of characters), which allows use of an implementation with an arbitrary look-ahead, straightforwardly applying trial-and-error method for each of the rules, trading off efficiency for simplicity.[1] To enable a use of the shorthand notation, a preprocessor, working on a textual level, is employed to fill in the missing parentheses, bypassing problems connected with parsing the shorthand form's left-recursive grammar.

grammar

To submit a syntactically correct term for an evaluation, a user clicks on the *Submit* button, or presses the Enter key, entering the *main screen.*

3.2.2 Evaluation

After a syntactically valid term has been submitted, one can commence and repeat the manual, step-by-step evaluation by mouse-clicking the desired redex inside the expression until a normal form is reached. The new, reduced expression will be rendered below the clicked one, together with an arrow indicator of the redex' kind. As described in section 2.1.4, under full β-reduction, redexes inside the term may be evaluated in an arbitrary order. If an evaluation strategy other than the default full beta reduction is selected, it is possible to select at most one redex, as accorded by the chosen evaluation strategy. Presence of a redex inside a term is indicated

by an argument- and function color outline, shown on mouse-hover, as illustrated in figure 1. Different kinds of redexes are indicated by an outline of these different colors, written as hexadecimal triplets, using the RGB color model:

`98c6a8`	δ-redex
`e6c79b`	η-redex, *Let* expression
`8e8fa7`	β-redex' function
`bc8f8f`	β-redex' argument

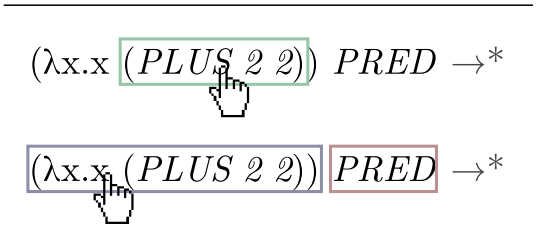


Figure 1: *A color outline indicating a reducible expression appearing on mouse-hover. The asterisk-arrow for performing an immediate evaluation is visible following the term.*

Expectedly, on mouse click, the highlighted expression will be reduced. The reason behind highlighting the redexes only on mouse-hover, instead of highlighting them permanently, is twofold: for visual clarity, as a permanent color outline would create bothersome visual clutter on the screen, and secondly, redexes are often nested, which would result in confusing outlines bearing no useful information. In a case of nested redexes, mouse selection is analyzed bottom-up—the deepest redex that is pointed at will be chosen.

As some of the evaluation chains, or *derivations*, can be rather long, in some cases using only step-by-step evaluation to reach the normal form would be tiring, or even altogether unfeasible. To directly evaluate a term at once, one can simply click on an asterisk-arrow following a reducible term, as visible in figure 1 towards the right margin. If a derivation is short enough, precisely, less than 50 evaluation steps, each evaluation step will be rendered on a new line, as if evaluated manually, otherwise, only the computed normal form will be appended to the already rendered derivation, and the information about the number of taken evaluation steps needed to reach the normal form will be displayed on the bottom panel. The user-selected reduction strategy will be used for an immediate evaluation, except in the case of *full beta reduction*, where the *normal order* will be used, as it is complete—if the normal form does exist, it will eventually be reached. As the problem of recognizing the non-normalizing terms is generally undecidable,[21] the computation may continue indefinitely, without any means to circumvent or detect such non-terminating computation beforehand. A simple heuristic terminating execution of already computed terms is employed. clarify, eta reduciton

A summary information about the number of possible redexes on the current line, their kind, and, would be selected a type system, type of the evaluated term is presented on the bottom panel.

3.2.3 Alias management

Sometimes, it may be useful to assign a name to a commonly used terms. Addition of an alias is done via the *Add an alias* dialog box (fig. 2), accessible from the head

navigation bar's *Options*. The name can be composed of only uppercase alphabetical characters, and this condition is reflected in the behavior of the name's input box: firstly, any alphabetical character will be immediately converted to upper case, and secondly, as non-alphabetical characters are not admissible for names, it is not possible to enter them into the box at all.

The term to be named is entered in a similar fashion to that of the initial screen; expectedly, the already defined aliases and constants can be used to define a new alias. An option to enter the term on the last (*current*) line automatically into the input box is available. The color of the input box (stylized as a line) is changing based on the syntactical correctness of the term, or if using an already defined name, in a same manner as that described in § 3.2.1 – User's input.

An alias is assigned only to a particular term; this means that generally the α-equivalent terms do not share the same assigned name, the name is applied only to a syntactically identical terms.

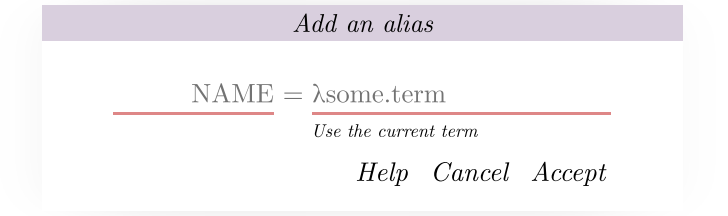


Figure 2: *The alias addition dialog box.*

3.2.4 Display settings

At any point, it is possible to change the active display setting via the head navigation bar options *Expand aliases* and *Shorthand form*, both ranging over binary values. If the alias expansion option is active, terms will be rendered as the name they carry (fig. 3). On mouseover, the term will be shown without the alias expansion, enabling a user to select potentially aliased reducible expressions. The suitability for the alias expansion is checked top-to-bottom: would there be a named term M containing another named term N, the name of the term M will be preferred.

The shorthand form option toggles between rendition of terms (and types, if any present) either as fully parenthesized, or abbreviated, with unambiguously omitted parentheses, as described in section 2.1.2.

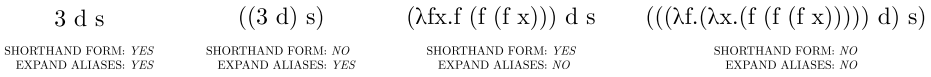


Figure 3: *A collage of application's term renditions under different settings.*

By default, aliases are expanded, and terms are shown in the shorthand form. Changing any of the two settings will cause an immediate redraw of the terms currently presented on the screen.

3.2.5 Help page

The *Help* page does serve as a readily available reference, accessible from the applications' head navigation bar at all times. Besides the general information about

the application and its use, the help page also contains a number of illustrative λ-calculus expressions, possible to be evaluated using the application. To load the mentioned examples to the application, the user simply selects the term by clicking on it. Additionally, on the help page, one can find an enumeration of all available pre-defined aliases and constants.

3.2.6 Import and export

At times, one may wish to store an evaluation chain for later retrieval, print the derivation, or export it as a LaTeX source. The printing and LaTeX source generation functionality utilizes the active display settings, namely the alias expansion and shorthand form, thus making the exporting facility as flexible as the web application's term-displaying abilities themselves.

A quick and convenient way to share or save a term on the current line is to copy the URL from the web browser's address bar, as the application does encode the current term in the URL's optional *fragment identifier* part, initiated with a hash symbol (#), typically used to point to a particular portion of a document.

The import and export options dialog (fig. 4) is accessible via the *Options* menu from the head navigation bar.

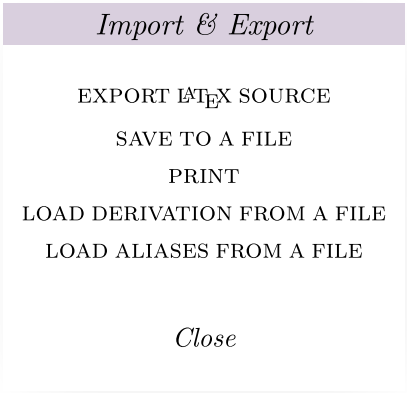


Figure 4: *The "Import & Export" options dialog box.*

File input and output By user request, a file containing the derivation chain is generated and downloaded. The default name of the file is `save.lambda`, and in a human-readable format, the file holds the information about term on every line, what kind of reduction did produce the term *(the arrow)*, and the type system used (figure 5). All of the user-defined aliases are also stored inside the file, as the aliases may be used to display the saved terms, or simply to enable users to conveniently reuse the already defined aliases in the future; this is especially useful with a combination of the initial screen feature allowing loading only the aliases from the file, as these can be used for an input of a new term right away. When loading a file into the evaluator, the system prompt looks only for `*.lambda` files by default.

```
discipline UNTYPED
alias ADDFIVE (PLUS 5)
term NO ((λx1.((PLUS 5) x1)) 4)
term BETA ((PLUS 5) 4)
term DELTA 9
```

$$
\begin{aligned}
& (\lambda x_1.\text{ADDFIVE}\, x_1)\, 4 \\
\to_\beta \quad & \text{ADDFIVE}\, 4 \\
\to_\delta \quad & 9
\end{aligned}
$$

Figure 5: *Human-readable file text (left) and the corresponding resulting derivation after a loading (right). The LaTeX source code of the resulting derivation is shown in fig. 6.*

LaTeX source code generation The exported evaluation chain is typeset using the `align*` environment, available through the popular `amsmath` package (required for any LaTeX distribution[8]); the user has to include the `\usepackage{amsmath}` declaration in the preamble of their document. On export, a new browser window will be opened, containing the LaTeX source code of the derivation in plain text. Before exporting a derivation, one should make sure that the desired displaying options are in place, as these will be used to generate the code to make the compiled result look essentially the same as the derivation shown on the application screen. In the same manner as on the application screen, constants are set in italic type, and aliases are set in roman type.

```
% don't forget to \usepackage{amsmath}
\begin{align*}
&(\lambda x_{1}.\mathrm{ADDFIVE}\;x_{1})\;\mathit{4}\\
\to_\beta\quad&\mathrm{ADDFIVE}\;\mathit{4}\\
\to_\delta\quad&\mathit{9}
\end{align*}
```

Figure 6: *An example of a generated LaTeX code, setting the derivation shown in fig. 5.*

Printing

3.3 Web browser compatibility

mouse and on phones

3.4 Comparison with existing solutions

There are several existing applications for evaluating lambda calculus expressions available online. In order to justify the existence of yet another, a qualitative comparison of multiple features of six of the existing solutions to this work's is given in the following table:

Although an evaluation of user interfaces is

[8] `https://ctan.org/pkg/required`

	aliases	reduction strategies	step-by-step evaluation	import & export	type systems	lambda input
this work	yes, input and output, programmatic Church numerals	normal order, applicative order, call by value, call by name	yes	file, URL, LaTeX	simply typed λ-calculus, Hindley-Milner type system	\ *and* λ
[9]	no	normal order	no, displays intermediate results	no options available	no options available	lambda
[6]	no	call by value	no	no options available	no options available	^
[4]	yes, input and output	lazy evaluation, eager evaluation, normal order	no, displays intermediate results	option to export user-defined aliases	no options available	\
[13]	yes, input only	*not listed*	no	no options available	no options available	\ *and* λ
[16]	yes, via let expressions, input only	*not listed*	no	no options available	no options availabe	\
[18]	yes, input and output	normal order, call by name, head spine reduction, call by value, applicative order, hybrid normal order, hybrid applicative order	yes	no options available	no options available	\

Table 1: *A comparison with existing web λ-calculus evaluators.*

4 Conclusion

Finito

References

[1] Alfred Aho. *Compilers: principles, techniques, & tools*. Boston: Pearson/Addison Wesley, 2007, p. 61. isbn: 0-321-48681-1.

[2] Henk Barendregt. *Introduction to Lambda Calculus*. 1994. url: http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf.

[3] Henk Barendregt. "The Impact of the Lambda Calculus in Logic and Computer Science". In: *The Bulletin of Symbolic Logic* 3.2 (1997), pp. 181–215. issn: 10798986. url: http://www.jstor.org/stable/421013.

[4] Carl Burch. *Lambda Calculator*. 2012. url: http://www.cburch.com/lambda/ (visited on 04/06/2018).

[5] Felice Cardone. "Lambda-Calculus and Combinators in the 20th Century". In: *Logic from Russell to Church*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Handbook of the History of Logic. North-Holland, 2009, pp. 723–817. doi: https://doi.org/10.1016/S1874-5857(09)70018-4. url: http://www.sciencedirect.com/science/article/pii/S1874585709700184.

[6] Zach Carter. *Lambda Calculus Evaluator*. 2010. url: http://zaa.ch/lambdacalc/ (visited on 04/06/2018).

[7] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. issn: 00029327, 10806377. url: http://www.jstor.org/stable/2371045.

[8] Oracle Corp. *Java Programming Language Enhancements*. Apr. 20, 2018. url: https://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8.

[9] Liang Gong. *λ Calculus Interpreter*. University of California, Berkley. 2014. url: https://people.eecs.berkeley.edu/~gongliang13/lambda/ (visited on 04/06/2018).

[10] David Hilbert. *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

[11] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Comput. Surv.* 21.3 (1989), pp. 359–411. issn: 0360-0300. url: http://doi.acm.org/10.1145/72551.72554.

[12] *Lambda expressions*. 2018. url: http://en.cppreference.com/w/cpp/language/lambda (visited on 04/20/2018).

[13] Ben Lynn. *Lambda Calculus*. url: https://crypto.stanford.edu/~blynn/lambda/ (visited on 04/06/2018).

[14] Jan Obdržálek. *IA014 – Advanced Functional Programming*. lecture notes. Faculty of Informatics, Masaryk University, 2017. url: https://is.muni.cz/el/1433/jaro2017/IA014/um.

[15] Benjamin Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. isbn: 0-262-16209-1.

[16] Jim Pryor. *Lambda Evaluator*. 2015. url: http://lambda.jimpryor.net/code/lambda_evaluator/ (visited on 04/06/2018).

[17] Dana Scott. *Lambda Calculus Then and Now*. [lecture recording]. Association for Computing Machinery. 2013. url: https://www.youtube.com/watch?v=SphBW9ILVPU (visited on 04/15/2018).

[18] Peter Sestoft. *Lambda calculus reduction workbench*. 2002. url: https://www.itu.dk/~sestoft/lamreduce/ (visited on 04/06/2018).

[19] Alan Mathison Turing. "Computability and λ-Definability". In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163. issn: 00224812. url: http://www.jstor.org/stable/2268280.

[20] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. url: http://dx.doi.org/10.1112/plms/s2-42.1.230.

[21] Jiří Zlatuška. *Lambda-kalkul*. Masaryk university, 1993. isbn: 80-210-0826-1.

A Predefined constants and aliases

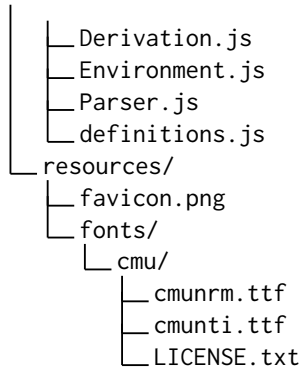
B Input examples

C Attached files

```
lambda/
├── index.html
├── help.html
├── help_style.css
├── print.css
├── style.css
└── src/
```

Table 2: I hate LaTeX tables

	Aliases		Constants	
	description and term		arity	description
n	Church encoded natural numbers $\mathbb{N}$, input up to 3 digits only		0	integer $\mathbb{Z}$ primitives. JavaScript implementation-defined range
	$\lambda f.\lambda x.f.nx$			
SUCC	successor function on Chuch numerals		1	successor function on constants
	$\lambda nfx.f\ (n\ f\ x)$			
PRED		predecessor function, the least result is zero	1	predecessor function, negative results allowed
PLUS		addition function on Church numerals	2	addition function on constants
MINUS	lambda m n.n PRED m	church encoded substraction, the least result is zero.	2	substraction on constants, negative results allowed
TIMES		multiplication on Church numerals	2	multiplication function on constants
DIV			2	division function on integer constants
EQ			2	equality function on numerical constants, returns Boolean constant
TRUE		Church Boolean true	0	Boolean true constant
FALSE		Church Boolean false	0	Boolean false constant
ITE	lambda xyz.xyz	conditional, the first argument is Church Boolean	3	conditional, the first argument is Boolean constant, type system dependent semantics
OR		logical conjunction on Church Booleans	2	logical conjunction on Boolean constants
AND		logical disjunction on Church Booleans	2	logical disjunction on Boolean constants
FIX	[undefined]		1	fixed point of the input
Y		fixed point combinator	[undefined]	
THETA		fixed point combinator, displayed on screen as THETA	[undefined]	
OMEGA		Omega combinator, displayed on screen as OMEGA	[undefined]	
S			3	
K			2	
I	lambda x.x	Identity combinator	1	


Derivation.js
Environment.js
Parser.js
definitions.js
resources/
favicon.png
fonts/
cmu/
cmunrm.ttf
cmunti.ttf
LICENSE.txt