Masaryk University
Faculty of Informatics

JAKUB KADLECAJ

Online Lambda Calculus Evaluator

BACHELOR'S THESIS

Brno, Spring 2018

Declaration of (authorial) independence
Acknowledgement,
Abstract: intro, len kratsie?
hashtags: #lambda calculus, #web application, #functional programming, #type theory, #JavaScritpt

Contents

Chapter 1

Introduction

Lambda calculus is a formal system of logic invented by Alonzo Church in the early 1930s that deals with functions described as substitution-based evaluation rules. It plays an important role in computer science and mathematics, with applications in other fields, such as linguistics and philosophy.[**cardoneHindley**]

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 already existing 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. Lastly, a thorough description of the evaluator web application itself will be given. In the appendices, one can find examples of some of the possible inputs to the application, along with an enumeration of predefined aliases and constants, potentially serving as a reference manual.

1.1 Historical background

"Wir müssen wissen – wir werden wissen!" [a] were the well-known words of David Hilbert, countering the Emil du Bois-Reymond's *"Ignoramus et ignorabimus"* [b] aimed towards the natural sciences, and expressing his belief that any mathematical problem posed in an appropriate formal language can be solved, preferably by mechanical means. In 1928 Hilbert stated the challenge, entitled *Entscheidungsproblem*[c], 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.[**hilbert**] In order to prove or disprove existence of such an algorithm, there was a need to formalize the intuitive notion of an algorithm itself first.

[a] German for "We must know—we will know!"
[b] Latin for "We do not know and we will not know"
[c] German for "Decision problem"

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 formalizes 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.[**turingPaper**]

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 it has become a successful model of computation with a multitude of contributions to the theory of programming languages.

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.[**turingDefin**] Assuming that functions on natural numbers computable by a human with limitless resources are the same as functions definable by Turing machine (widely accepted *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—independently by both Turing and Church in 1936.[**turingPaper**, **churchPaper**]

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 (ˆ), then already used as a class abstraction operator in Russel's and Whitehead's *Principia Mathematica*, with lambda, which is similar in appearance.[**barenImpact**] Dana Scott, a PhD student of Church's and a Turing Award laureate, opposes this hypothesis and claims the lambda letter was chosen arbitrarily.[**scottLecture**]

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, historically more popular approach of the imperative programming, based on a consecutive modification of a program's state.[d] 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.[**baren94**]

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[e] 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:

[d] This dichotomy 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.

[e] For instance, lambda expressions were introduced in 2011 version of C++ and 2014 version of Java.[**cppRef**][**oracleRef**]

λ-calculus	$\lambda xy.\text{TIMES}\,(\text{PLUS}\,x\,y)\,y$
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 a 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.

λ-calculus	$\lambda fgx.f(gx)$
Haskell	`f g x = f (g x)`
JavaScript	`function compose(f, g) { return x => f(g(x)); }`

The three previous examples show a definition of a function that returns another as an output. However, in imperative languages, a more common occurrence is using functions as parameters of another function, rather returning function as its result, as in the following example, calculating a product of an array v:

Haskell
```
foldl v (*) 1
```
C++
```
auto times = [](int a, int b){ return a * b; };
std::accumulate(v.begin(), v.end(), 1, times);
```

In this example, the multiplication function has been passed as the input of another. In the C++ example, a lambda expression was used to define a multiplication function.

Partial application

Type derivation

Chapter 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, or 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 syntactic abbreviations, for example $\{\lambda x[\{x\}(\{y\}(z))]\}(w)$,[churchPaper] $((\lambda x((x\ (y\ z))))\ w)$,[zlatuska] or $((\lambda x.(x\ (y\ z)))\ w)$,[hudak] 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[slides] will be adopted, along with the common abbreviation methods (Section 2.1.2).

In the following definitions, the symbols M and 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 denoted by 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 alphabetical 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 $\Lambda = (\Lambda\Lambda) \mid (\lambda \mathrm{VAR}.\Lambda) \mid \mathrm{VAR}$, where VAR = lowercase letter | lowercase letter$_{n\in\mathbb{N}}$. The set of terminals consists of the elements of VAR, dot and 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 a 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 lambda-abstracted variable, and a *function body* for the abstraction's term.

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

2.1.2 Syntactic conventions

To make terms more succinct and convenient to read and write, the following syntactic conventions, describing omission of redundant parentheses, dots and lambda symbols, 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)*.[**pierce**] 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. An example of an alias is SUCC, which stands for the term $\lambda n f x.f\ (n\ f\ x)$, which is a successor function, as will be shown later, along the means of assigning terms to numeral names, described in Section 2.1.6 – Church encoding. It is important to understand the names to be only a syntactic, meta-λ-calculus shorthand, especially with regard to self-reference of terms.

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 inductively defined function $\mathrm{FV} : \Lambda \to \{\,\mathrm{VAR}\,\}$, mapping terms to the respective sets of free variables in the following manner:[**zlatuska**]

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

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 a *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, inductively defined as follows:

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

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

β-reduction 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, albeit this treatment of the terms is not universal.[**pierce**] 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:[**slides**]

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

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

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*. An important property of β-reduction is that if a β-normal form does exist, it is, up to α-conversion, unique. This property is is known as *Church-Rosser property*, as proven by Church and Rosser in 1936.[**churchRosser**]

η-reduction Another possible operation on terms is called η-reduction (or η-conversion), formalizing the idea of simplifying redundant variable abstractions, for instance, the term $(\lambda x.NOT\ x)\ FALSE$ will β-reduce to $NOT\ FALSE$, with the function being mostly useless, as the term NOT would be a sufficient substitute for such a function. Thus,

the η-reduction is defined, enabling contraction of intuitively superfluous abstractions, $(\lambda x.M\ x) \to M$, if the x is not free in M, formally:

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

For example, $\lambda xy.PLUS\ x\ y \to_\eta \lambda x.PLUS\ x \to_\eta PLUS$. The necessity of the condition on the abstracted variable not being free in M is apparent on this example: $(\lambda x.x\ x)\ M \to_\beta M\ M$, but if the function was η-converted first, the result would be $(\lambda x.x\ x)\ M \to_\eta x\ M$, which is clearly a different term.

A term that is in β-normal form and cannot be η-reduced is said to be in $\beta\eta$-normal form. The Church-Rosser property also holds for η-reduction and $\beta\eta$-reduction.

α-conversion The α-conversion captures the notion of intuitive equality of terms which are effectively identical, except with different variables. The α-equivalence is given by the following rules,[**slides**] and the α-conversion is the act of renaming variables, where the resulting term is α-equivalent ($=_\alpha$) to the

$$\frac{}{x =_\alpha x} \qquad \frac{M_1[x := z] =_\alpha M_2[y := z] \qquad z \notin \mathrm{FV}(M_1 M_2)}{\lambda x.M_1 =_\alpha \lambda y.M_2}$$

$$\frac{M_1 =_\alpha M_2 \qquad N_1 =_\alpha N_2}{M_1\ N =_\alpha M_2\ N} \qquad \frac{N_1 \to_\beta N_2}{M\ N_1 \to_\beta M\ N_2}$$

One more operation, the δ-reduction, will be presented later in Section 2.1.8.

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*, is a strategy evaluating arguments only if they are used in the function body.[**pierce**] 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 referential transparency[a] 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

[a] Referentially transparent 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`.

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:
$(\lambda x.w)\,\underline{((\lambda r.r\,r)\,(\lambda r.r\,r))} \to_\beta (\lambda x.w)\,\underline{((\lambda r.r\,r)\,(\lambda r.r\,r))} \to_\beta (\lambda x.w)\,\underline{((\lambda r.r\,r)\,(\lambda r.r\,r))} \to_\beta \ldots$, 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 the few languages that use a non-strict evaluation strategy, e.g. Haskell or (slightly older) Miranda. A drawback of non-strict evaluation strategy is a possible duplication of work, as the arguments have to be evaluated each time they occur in function. For instance, $(\lambda m.m\ m\ m)\,(\textit{complex problem}) \to_\beta (\textit{complex problem})\,(\textit{complex problem})\,(\textit{complex problem})$, whereas employing a strict evaluation strategy, the *complex problem* would have to be solved only once, before passing it as an argument.

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

Normal order Normal order is a non-strict evaluation strategy, selecting the leftmost, outermost of redexes for evaluation. This strategy is *complete*, or *normalizing*—hence the name, that is, if a normal form does exist, it will be reached.[baren94] Example evaluation:

$$\begin{aligned} & \underline{(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda w.w)\,z))} \\ \to_\beta\ & \underline{(\lambda x.x)\,(\lambda z.(\lambda w.w)\,z)} \\ \to_\beta\ & \lambda z.\underline{(\lambda w.w)\,z} \\ \to_\beta\ & \lambda z.z \quad \not\to \end{aligned}$$

Call by name Call by name is a non-strict evaluation strategy,

$$\begin{aligned} & \underline{(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda w.w)\,z))} \\ \to_\beta\ & \underline{(\lambda x.x)\,(\lambda z.(\lambda w.w)\,z)} \\ \to_\beta\ & \lambda z.(\lambda w.w)\,z \quad \not\to \end{aligned}$$

Applicative order stricc

$$\begin{aligned} & (\lambda x.x)\,((\lambda x.x)\,(\lambda z.\underline{(\lambda w.w)\,z})) \\ \to_\beta\ & (\lambda x.x)\,\underline{((\lambda x.x)\,(\lambda z.z))} \\ \to_\beta\ & \underline{(\lambda x.x)\,(\lambda z.z)} \\ \to_\beta\ & \lambda z.z \quad \not\to \end{aligned}$$

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 instances

of the *abstraction* rule, later on, however, the set of values will be extended by primitive values, such as numeric or logical constants, or functions on terms (Section 2.1.8). The set of values is written as V, and the evaluation rules are described as follows:

$$\frac{}{(\lambda.M)\,V \to M[x := V]} \qquad \frac{M_1 \to M_2}{M_1 N \to M_2 N} \qquad \frac{N_1 \to N_2}{V\,N_1 \to V\,N_2}$$

The evaluation below is an example of using the call by value strategy. The term on last line cannot be reduced any further, as it is already value.

$$\begin{aligned} & (\lambda x.x)\,(\underline{(\lambda x.x)\,(\lambda z.(\lambda w.w)\,z)}) \\ \to_\beta \quad & \underline{(\lambda x.x)\,(\lambda z.(\lambda w.w)\,z)} \\ \to_\beta \quad & \lambda z.(\lambda w.w)\,z \quad \not\to \end{aligned}$$

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.

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, $\mathbf{TRUE} \equiv \lambda xy.x$, and $\mathbf{FALSE} \equiv \lambda xy.y$. After the Boolean values are established, the definition of the fundamental logic operators is desired. It is required that $\mathrm{OR}\ M\ N$ *(logical disjunction written in prefix notation)* will reduce to $\mathbf{TRUE}$ 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 $\mathbf{TRUE}$. 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:

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

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:

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

This term makes use of the property of $\mathbf{TRUE}$ and $\mathbf{FALSE}$ returning the first, respectively the second, of the two arguments.

Church numerals and arithmetic operators

2.1.7 Recursion

Recursion is one of the fundamental principles of computer science, allowing for solving a problem by combining the solutions of the smaller instances of the same problem. In

programming languages, or mathematical notation, this is realized by enabling functions to refer to themselves in a subsequent function call. Notwithstanding the practical aspects, recursion is able to completely substitute for iterative control structures, such as loops, which are sometimes entirely absent in functional programming languages, and clearly, absent in lambda calculus.

However, because the λ-calculus terms are not assigned any language-level identifiers, or memory addresses, the self-referential functions cannot be defined directly, but it is possible to arrange the terms in such a way, that an expression will be self-applied, receiving itself as an argument, for instance $(\lambda w.w\ w)\ M \rightarrow_\beta M\ M$.

The solution comes from using a fixed point of a function, applying a function

for example, $\mathbf{fix}\ \sin = \sin(\mathbf{fix}\ \sin) = \sin(\sin(\sin(\ldots(\mathbf{fix}\ \sin)\ldots)))$, as $0 = \sin 0$.

Using the canonical example of recursively-defined factorial function, $\mathrm{fac}\ x = \text{if}\ x = 0\ \text{then}\ 1\ \text{else}\ x \cdot \mathrm{fac}(x-1)$, the recursion using fixed point combinator is demonstrated:

$\lambda f.(\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1))$

infinite application

$$(\lambda f.(\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1)))\ \Big((\lambda f.(\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1)))\ \Big((\lambda f.(\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1)))\ \ldots\Big)\Big)$$

$\downarrow_\beta$

$$R_0 : \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (R_1\ (x-1))$$
$$R_1 : \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (R_2\ (x-1))$$
$$R_2 : \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot (R_3\ (x-1))$$

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 $\mathbb{C}$, can be 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; functional constants are written using the prefix notation. Operational semantics is defined via a set of contraction rules, called δ-rules.[**baren94**] The left-hand side of a δ-rule is called δ-redex.

In literature, functional symbols are usually not allowed to occur as stand-alone terms, and they have to be supplied with appropriate arguments.[**pierce**] In this work, however, to provide more flexibility, such as passing functional arguments, the functional symbols are considered as proper as any other terms, enabling the partial application.

The set of *values*, V, is extended with constants, and with partial applications of them, collectively denoted by $C^\flat$, which is either a constant C, or an application of terms initiated with a constant, $CM_1M_2\ \ldots\ M_n$, if there exist terms $N_1N_2\ \ldots\ N_m$, such that $CM_1M_2\ \ldots\ M_nN_1N_2\ \ldots\ N_m$ is a δ-redex.

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

In this work, the constants are written as strings of uppercase letters set in italic type. The exhaustive list of predefined constants can be found in the appendix A. An example of

δ-rules for logical disjunction defined for constants:

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

***Let* expressions** In untyped λ-calculus, the *Let* expression is a syntactic shorthand, enabling more convenient definitions of functions that appear to be named, and use them in a clearly bounded scope.

$$\mathit{Let}\ x = M\ \mathit{In}\ N \equiv (\lambda x.N)\,M$$

To simplify definitions of recursive function definitions, an extended, *LetRec* expression can be used, as the attempt to self-reference using just the plain *Let* would not produce the expected result.

$$\mathit{LetRec}\ x = M\ \mathit{In}\ N \equiv (\lambda x.N)\,(\mathrm{Y}\,\lambda x.M)$$

To make the notation even more convenient, the λ-abstracted parameters can be put to the other side of the equality sign, closely resembling definition of a function using the mathematical notation:

$$\mathit{Let}\ f\,x_1 x_2 \ldots x_n = M\ \mathit{In}\ N \equiv \mathit{Let}\ f = \lambda x_1 x_2 \ldots x_n.M\ \mathit{In}\ N$$

An example of a recursive function definition, computing factorial of five:

$$\begin{aligned} & \mathit{LetRec}\ f\,x = \mathit{ITE}\,(\mathit{EQ}\,x\,0)\,1\,(\mathit{TIMES}\,x\,(f\,(\mathit{PRED}\,x)))\,\mathit{In}\ f\ 5 \\ = \quad & (\lambda f.f\ 5)\,(\mathrm{Y}\,(\lambda f x.\mathit{ITE}\,(\mathit{EQ}\,x\,0)\,1\,(\mathit{TIMES}\,x\,(f\,(\mathit{PRED}\,x)))\,)) \\ \rightarrow^* \quad & 120 \end{aligned}$$

While usually not capitalized in literature, in this work, the "Let" and "In" keywords are capitalized, as to prevent needless ambiguity arising from the established syntactic conventions, particularly the omission of parentheses in the application of variables, and furthermore, for the same reasons, the "LetRec" is written as a single word, although usually appearing in literature as "let rec".

2.2 Typed lambda calculi

2.2.1 Simply typed lambda calculus

The simply typed λ-calculus, often abbreviated as $\lambda^{\rightarrow}$, is a system introduced by Alonzo Church to address the paradoxes arising from lack of restrictions of pure λ-calculus, namely the possibility of self-application of functions, generally inadmissible when dealing with conventional, set-theoretic understanding of functions, associating each element of domain to some element of the codomain, as

An introduction of types to λ-calculus deals with, but

Types The system operates with a single, binary type constructor, arrow $\rightarrow$, taking two types α and β, and constructing a new function type—a type of a function mapping input of type α onto the result of type β. As a base case, a finite number of atomic *base types* is considered, in the case of this work, these are Bool and Int, which are the the types of Boolean values, and numerical values (integers), respectively.

$$\begin{array}{rcl} \sigma & ::= & (\sigma \rightarrow \sigma) \\ & | & \text{Int} \\ & | & \text{Bool} \end{array}$$

For instance, the logical negation function is of a type (Bool $\rightarrow$ Bool), as the expected input is a Boolean value, and so is the output. The arrow type constructor is not able to construct tuple types (or any other product types), so the types of functions accepting multiple arguments, i.e. functions of arity $n > 1$, have to be deconstructed into types of n successive unary functions. Thus, the addition function is of a type (Int $\rightarrow$ (Int $\rightarrow$ Int)), i.e. a type of an unary function accepting a number, and returning another function. When understood as sets, the type Int is occupied by the numerical constants defined in Section 2.1.8, the type Bool by Boolean constants, and so forth for functional constants and their types.

Analogous to syntactic conventions of terms, a convention to simplify the type notation is given by the following rule, asserting the right associativity of the arrow:

$$\sigma_1 \rightarrow \sigma_2 \rightarrow ... \sigma_{n-1} \rightarrow \sigma_n \equiv (\sigma_1 \rightarrow (\sigma_2 \rightarrow (\,... (\sigma_{n-1} \rightarrow \sigma_n) ...)))$$

Adopting the convention, the type (Int $\rightarrow$ (Int $\rightarrow$ Int)) can be written as Int $\rightarrow$ Int $\rightarrow$ Int, and (Bool $\rightarrow$ ((Int $\rightarrow$ Int) $\rightarrow$ Int)) as Bool $\rightarrow$ (Int $\rightarrow$ Int) $\rightarrow$ Int.

Term syntax The term syntax is largely unchanged, with an exception of the *abstraction* rule, adding an explicit type annotation of the parameter.

$$\begin{array}{rcll} M & ::= & x & \textit{Variable} \\ & | & (M\,M) & \textit{Application} \\ & | & (\lambda x{:}\sigma.M) & \textit{Abstraction} \\ & | & C \in \mathbb{C} & \textit{Constant} \end{array} \qquad \begin{array}{rcl} V & ::= & (\lambda x{:}\sigma.M) \\ & | & C^b \end{array}$$

To clearly separate types and abstractions, the "lambda-dropping" syntactic convention will be relaxed and terms of form $\lambda x_1 x_2 ... x_n.M$ will be written as $\lambda x_1.\lambda x_2. \,... .\lambda x_n$, but with type annotations.

Typing

Evaluation The evaluation rules for the simply typed λ-calculus are identical to that of untyped, call-by-value lambda calculus, however are constrained by the type restriction imposed by the system: if the type of argument is not equal to the type of parameter, the β-reduction is undefined.[**zlatuska**]

Often, a term alone does not contain sufficient information to determine its type, because the type of free variables may be assigned in an outer level by λ-abstraction. $\lambda x{:}\text{Bool}.fx$

$\lambda f{:}\text{Int}.(\lambda x{:}\text{Bool}.fx)$

$$\frac{}{(\lambda{:}\sigma.M)\,V \rightarrow M[x := V]} \qquad \frac{M_1 \rightarrow M_2}{M_1 N \rightarrow M_2 N} \qquad \frac{N_1 \rightarrow N_2}{V\,N_1 \rightarrow V\,N_2}$$

Strong normalization The simply typed lambda calculus has a *strong normalization property*, that is, any well-typed term has a normal form, and this normal form can be reached by an arbitrary sequence of reductions. This also means that the simply typed λ-calculus is, compared to the untyped, no longer Turing complete, as it, for instance, cannot express an infinite loop.

Recursion Recursion, in the pure lambda calculus, works by self-application of functions, for example $(\lambda x.x\ x)\ M \rightarrow M\ M$. If a concrete type was assigned as a parameter of function enabling such self-application, which is, for example, the Y combinator, the parameter would have to be a function of some type $\sigma \rightarrow \tau$ as the left-hand side of the application, and type σ of the righ-hand side of the application, forcing an expression to have both of these types at a same time, which is not possible. Because looping terms cannot be assigned a type, the recursion has to be achieved by the introduction of an external constant, ***FIX***, supplied with suitable typing- and evaluation rules.

$$\frac{\Gamma \vdash M : \sigma \rightarrow \sigma}{\Gamma \vdash FIX\ \ M : \sigma}$$

$$\frac{}{FIX\ (\lambda x{:}\sigma.M) \rightarrow M[x := FIX\ (\lambda x{:}\sigma.M)]} \qquad \frac{M_1 \rightarrow M_2}{FIX\ \ M_1 \rightarrow FIX\ \ M_2}$$

***Let* expressions**

Preservation theorem as

2.2.2 Hindley-Milner type system

Some functions are inherently polymorphic—they intuitively allow for having many concrete types and

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

Chapter 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.

The application is publicly available at `https://kdlcj.gitlab.io/lambda/`, with the repository containing the source files accessible via the application.

3.1 Features and the corresponding user interfaces

The application interface is structured, besides the *Help* page, as two screens: *initial screen*— the screen used to accept the user's input and to set preliminary options (fig. 3.2), e.g. a type system, and the *main screen* (fig. 3.1) for evaluation of the already input terms, together with relevant options and information presentation. The entry page, or the *initial screen*, (fig. 3.2) is also equipped with links potentially useful to newcome users, particularly one to the *Help* page, and one specifically to the *Examples* section of the said help page. On the initial screen, there is also an option to load a previously saved file. A closer look on the available import and export options is given in Section 3.1.6.

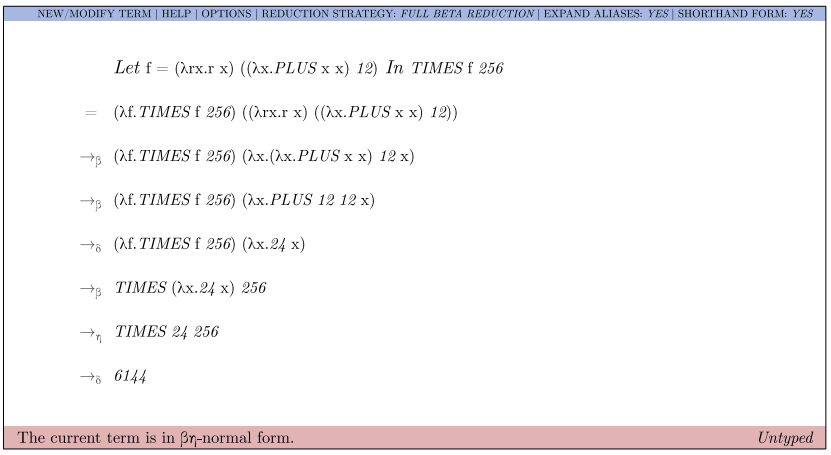

Figure 3.1: *The main screen. Size of the elements is increased for print.*

Head navigation bar On the main screen, the head navigation bar accommodating useful options is always present. The bar contains an option to return to the initial screen to modify the currently evaluated term, or enter a new one, a link to the help page, and display and evaluation strategy settings. Accessible from the *Options* drop-down menu, the alias addition, import and export, and a conversion to SKI combinators options are available.

Evaluation space This area is the essential component of the application, taking up the majority of the screen estate. The user-input terms are displayed on this surface, with the evaluation taking place there. The topmost term is the user-input one, and the terms below are products user-evaluation of the topmost. On the left-hand side of a term there is an arrow, indicating what kind of redex was used to produce the term. The evaluation itself is described in Section 3.1.2.

Bottom panel The bottom panel is a surface designed to show the information relevant to the evaluation of a particular term. The information about a number of each kind of redex present in the term on the last line is displayed, if a type system is selected, a type of the term or a notification of a failure to type the term, or when loading from a file, a number of the loaded terms or aliases. The information is written in a clear and a complete sentence, communicating the message in a natural manner. The content of the panel is dependent on the current type system, display settings, and reduction strategy. As an example, some terms and the corresponding bottom panel messages are given:

$\lambda f.sf$	The current term is in β-normal form, and it can be η-reduced.
s	The current term is in $\beta\eta$-normal form.
$(\lambda x.x)\,(PLUS\ 2\ 2)$	The current term contains 1 beta-redex & 1 delta-redex.
$\lambda fgx.f(gx)$	The current term is in β-normal form and its type is $(e \to d) \to (c \to e) \to c \to d$.
$\lambda a.b$	The current term has no redexes and it is not typeable.

3.1.1 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 syntactic 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 most natural way to write an arrow using a standard keyboard.

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*.

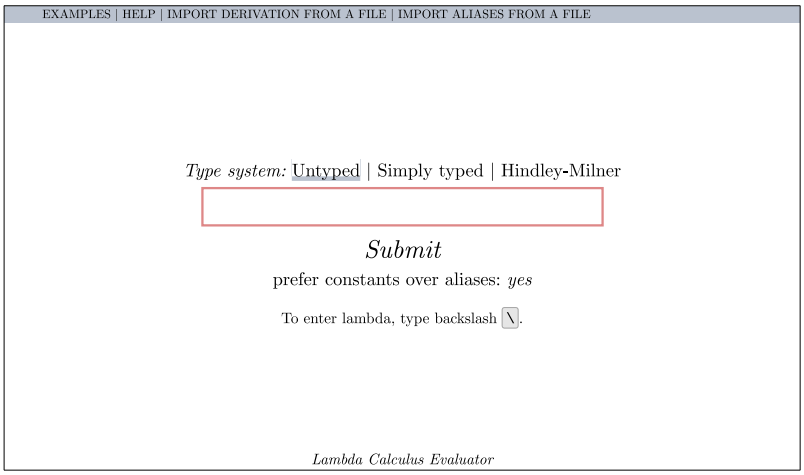

Figure 3.2: *The initial screen. Size of the elements is increased for print.*

Aliases and constants Aliases and constants can be used for input, including Church-encoded numerals up to three digits, except in the case of employed simply typed λ-calculus type system which does not accept Church-encoded data and functions. 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.

3.1.2 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 full beta-reduction is selected, it is possible to select at most one redex, as accorded by the chosen evaluation strategy. When evaluating untyped terms, the default strategy is full beta-reduction, otherwise, as formally defined, call by value is used, and the option to change the strategy is disabled. Presence of a redex inside a term is indicated by an argument- and function color outline, shown on mouse-hover, as illustrated in Figure 3.3. Different kinds of redexes are indicated by an outline of these different colors, written as hexadecimal triplets, using the RGB color model:

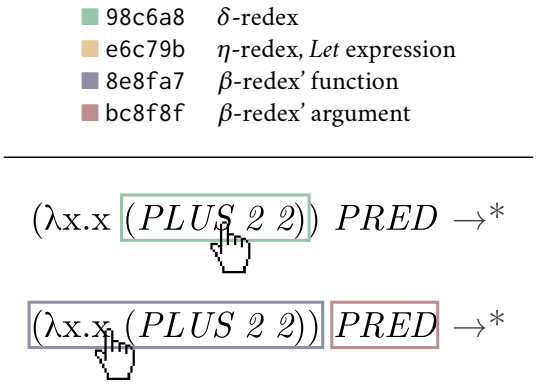

Figure 3.3: *A color outline indicating a reducible expression appearing on mouse-hover. The asterisk-arrow for performing an immediate evaluation is visible following the term.*

As the user would anticipate, 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. It is not possible to perform an η-conversion if some other kind of reduction is available.

Immediate evaluation 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 3.3 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,[zlatuska] 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 in succession is employed. Eta-redexes are not reduced automatically during the immediate evaluation and they have to be reduced only manually.

3.1.3 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. 3.4), 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, and an already existing name cannot be assigned to another term. An option to enter the term on the last (*current*) line automatically into the input box is available. The color of the input boxes, which are both stylized as line, is changing based on the syntactic correctness of the input term, or the admissibility of the intended name, in the same manner as that described in Section 3.1.1 – 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.

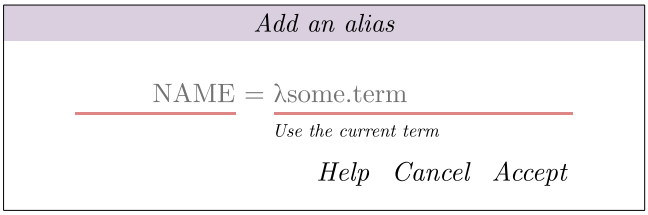

Figure 3.4: *The alias addition dialog box with placeholders describing the expected values.*

3.1.4 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. 3.5). On mouseover, the term will be shown without the alias expansion, enabling a user to select potentially aliased or nested reducible expressions. The suitability of 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.

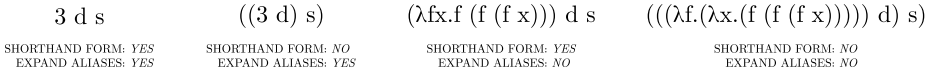

Figure 3.5: *A collage of application's term renditions under different settings.*

By default, aliases are expanded, and terms and types 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.

3.1.5 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 predefined aliases and constants.

3.1.6 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. The import and export options dialog (fig. 3.6) is accessible via the *Options* menu from the head navigation bar.

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.

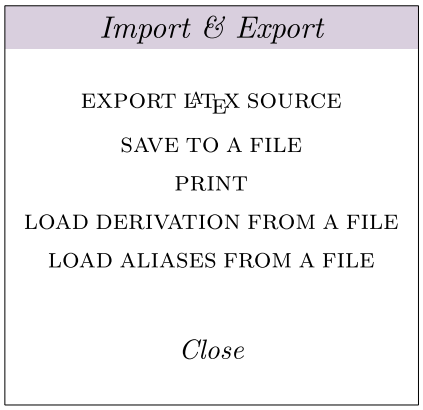

Figure 3.6: *The "Import & Export" options dialog box.*

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 3.7). 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
```

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

Figure 3.7: *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. 3.8.*

The *File* adheres to an intelligible structure, where each line is initiated with a keyword specifying the content of the line, followed by the actual data. In the resulting derivation, the terms are ordered the same way they appear in the file, otherwise, an order of the elements is not relevant. Formally, the syntax of the file is given by the following grammar, where the *Name*, as defined previously, is a string of uppercase letters, with terminals written using a monospaced typeface, where `\n` represents the newline character:

$$
\begin{aligned}
\textit{Line} &::= \texttt{alias}\ \textit{Name}\ \textit{M} \\
&\mid \texttt{term}\ \textit{Arrow}\ \textit{M} \\
&\mid \texttt{discipline}\ \textit{TypeSystem} \\
\textit{Arrow} &::= \texttt{NO} \mid \texttt{BETA} \mid \texttt{ETA} \mid \texttt{DELTA} \mid \texttt{EQ} \\
\textit{TypeSystem} &::= \texttt{UNTYPED} \mid \texttt{HINDLEY\_MILNER} \mid \texttt{SIMPLY\_TYPED} \\
\textit{File} &::= \varepsilon \mid \textit{Line}\texttt{\textbackslash n}\textit{File}
\end{aligned}
$$

LaTeX source code generation The exported evaluation chain is typeset using the `align*` environment, available through the popular `amsmath` package, which is required for any LaTeX distribution[a]—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 as 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.

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

```
% 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 3.8: *An example of a generated LaTeX code, setting the derivation shown in fig. 3.7.*

Printing Derivation printing is realized via the usual system-provided prompt, using a customized style sheet, describing a page layout for the printing. Notable changes, compared to the application's screen, are hiding the user interface elements, particularly head navigation bar and the bottom panel, modifying text overflow and page margin settings, and decoloring visible elements, e.g. the background, which has a slightly grey tint. The system printing dialog can be invoked via the *Import & Export* window, or using the web browser interface, and it is commonly accessible via the keyboard shortcut `ctrl + p`.

3.2 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:

	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
[**Interpreter1**]	no	normal order	no, displays intermediate results	no options available	no options available	lambda
[**Interpreter2**]	no	call by value	no	no options available	no options available	^
[**Interpreter3**]	yes, input and output	lazy evaluation, eager evaluation, normal order	no, displays intermediate results	option to export user-defined aliases	no options available	\
[**Interpreter4**]	yes, input only	*not listed*	no	no options available	no options available	\ *and* λ
[**Interpreter5**]	yes, via let expressions, input only	*not listed*	no	no options available	no options availabe	\
[**Interpreter6**]	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 3.1: *A comparison with existing web λ-calculus evaluators.*

Besides often missing useful features and an ease of use, some of the applications were unable to carry out even the more basic tasks, for instance accepting shorthand input, evaluating inputs of a still reasonably large size, or lacking the desired input-output idempotence.

Chapter 4

Implementation

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 static—it is executed exclusively on the client-side, in case of an actual deployment alleviating the server load on one hand, allowing for downloading and subsequent offline usage by a user on the other. The application has no external dependencies in terms of frameworks, libraries, or fonts, and is entirely self-contained.

The application is written using a modern JavaScript standard—the 6$^{\text{th}}$ edition of ECMAScript, providing many useful features, for instance the new syntax for object definition ("classes"), or lambda expressions ("arrow functions").[a] Would the JavaScript be disabled in the user's browser, an appropriate notification will be displayed.

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

The authorial source code is licensed under a public domain CC0 license,[c] allowing anybody to reuse and improve the work for any purpose and with no restrictions whatsoever.

4.1 Operation

The application works by programmatically modifying elements of the *Document Object Model* (DOM)—a tree structure with nodes representing parts of a document—based on the user's actions and the application's state. The application's evaluation operates in a working cycle initiated by a user action: first, after the user action has occurred (e.g. clicking on a textually represented redex), the application identifies and fetches the corresponding internal term-representation object, then, using the logic system's rules, a new term is

[a] `http://es6-features.org/`
[b] `http://scripts.sil.org/OFL/`.
[c] `https://creativecommons.org/publicdomain/zero/1.0/`

produced and the internal representation is updated, and finally, the changes are reflected on the external (screen) representation of the derivation chain.

HTML structure and DOM Both of the application's screens are described in a single hypertext markup `index.html` file (fig. 4.1). By programmatic modification of the tree structure, these screens, and also the dialog windows, are being displayed or hidden using the setting of an appropriate CSS attributes to the nodes. The presentation of the currently evaluated terms is realized by inserting a procedurally generated HTML code into the designated node, `evaluationFrame`, using the `Element.innerHTML` utility provided by the DOM API.[d]

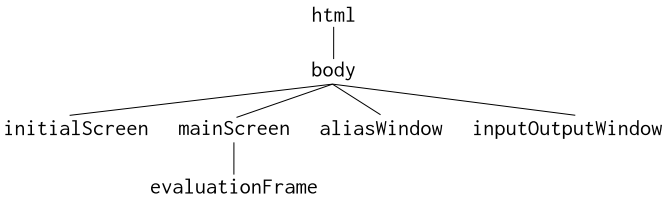

Figure 4.1: *An abridged structure of the application's HTML markup.*

Text input processing A top-down recursive descent parser, implemented using a single `Parser` object, 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.[**compilers**] 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.

Mouse redex highlighting processing As it is described in Section 3.1.2, mouse-hover events on terms on the screen invoke a color outline around a deepest pointed-at reducible expression, as portrayed in Figure 3.3. Although visual change on mouse-hover event is natively supported by CSS, in this context, using only the CSS would not be sufficient, as reducible expressions can be nested, which would instigate coloring not only of a one reducible expression, but also of all its parent elements—this problem is solved programmatically using JavaScript. When a HTML code of a term is generated, `class` markers are used to indicate a reducible expression inside a term, invisible at the time. On mouse-hover event, a function is called with the pointed-at DOM object passed as an argument. If the object is marked as a redex, the function swaps the `class` marker for another, triggering a CSS outline appearance, and terminating, otherwise, it is called for the parent element until it succeeds, or it is being applied outside of a reasonable scope.

Mouse redex clicking processing A mouse click processing is a bit more involved, as the particular term on screen, which has been clicked on, has to be identified within the internal representation, in order to be evaluated, for a term can contain multiple redexes. This identification is achieved by making each terminal symbol a button during the HTML generation, signalling what term is being clicked on, using a simple binary tree prefix

[d] `https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML/`

encoding, where an empty string does denote the root of a tree, and `0` and `1` do denote the left, respectively the right subtree (fig. 4.2). A part of the term interface, the `fromCode` method is used to fetch the requested subterm within the internal representation, optionally returning the pointed-at redex instead of the exact subterm.

The encoding of a term's position within the derivation chain is straightforward; the code of a term is prefixed with a line number (indexed by zero), followed by a point character (dot). For instance, the code `2.1` would represent the right subtree of a term on the third line of the derivation chain.

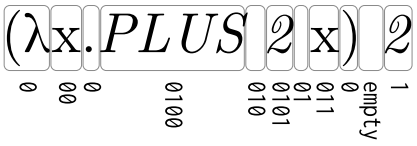

Figure 4.2: *An example of a subterm encoding, enabling an identification of the clicked term.*

Evaluation The evaluation itself is realized via the `Evaluator` object, which contains a library of static[e] methods designated to term evaluation and manipulation, containing redex identification, redex fetching, optionally taking into consideration the evaluation strategy or a particular redex kind, for instance `Evaluator.getRedex.eta`, or more general-purpose functions, such as `Evaluator.isValue`, or the ubiquitous `Evaluator.substitute`.

The `Evaluator` employs the static methods of the `Type` class when dealing with typed terms, and utilizes the application's state—for example the current type system settings—by accessing the main object's `OLCE.Settings` properties.

4.2 Structure

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 syntactic support for such a construct in the language itself. Moreover, the language lacks a conventional *class* semantics, and it is rather centered around *objects*. The `class` syntax, present in newer standards of JavaScript, is an syntactic sugar over the language's existing prototype-based inheritance model.[f]

The names of source code files do approximately match the object definitions they contain, and are briefly described in the Appendix C.

[e] Static method is a member function not bound to a specific instance, but rather the whole class. Because the `Evaluator` is a single object that is not an instance of any class, these methods are, in fact, bound to it's instance, thus are not static in a conventional sense.

[f] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`

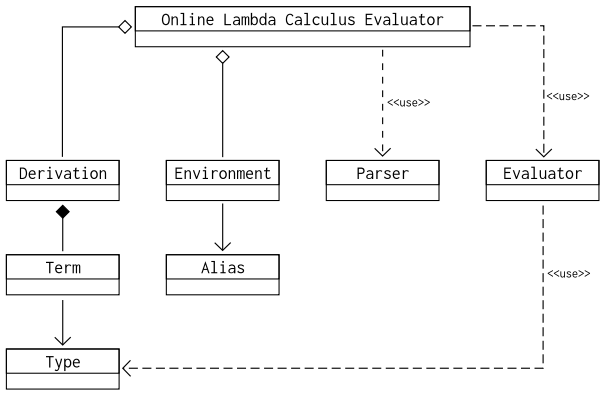

Figure 4.3: *A simplified class diagram of the project.*

The entry point The application itself is structured as a single object (named `OLCE`), containing a number of methods for direct DOM manipulation, and methods which are triggered by a user action—mouse or keyboard events. Generally, every control element of the application, e.g. a button, does have a corresponding `OLCE.UI` method, which gets invoked on the element's interaction, for instance on a button click. The data members of the `OLCE` object include a collection of the application's settings (e.g. how to display terms), DOM object references (e.g. where to render and display terms), the user- and predefined aliases, and the currently evaluated derivation chain (what terms to display). On a page load, the `OLCE.DOM.prepare()` associates the page elements with references for a rapid and a systematic further access by methods.

Internal representation The evaluated terms, as presented on the screen, are contained in the `Derivation` class, which is a singly linked list, extended with a functionality to generate a HTML code of the derivation chain, a LaTeX code, or to generate a file text for a subsequent downloading. Each instance of the `Derivation`—each node of the list—does contain a single `Term` object, and a kind of reduction that produced the term (i.e. an arrow).

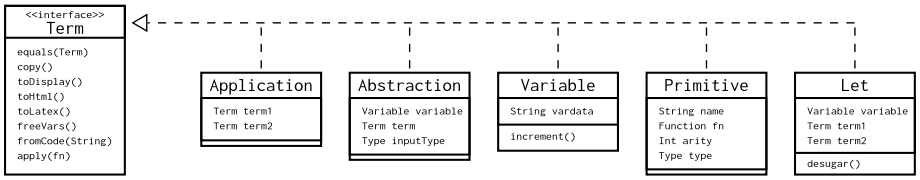

Figure 4.4: *Object term representation using an implicit interface.*

Because JavaScript is a dynamically typed language, a suitability of an object for a method call is determined by a presence of the method in the object, rather than by a type of the object. For this, the subtype polymorphism is no longer needed, the object just needs to be checked for an implementation of a desired call—consequently, the language lacks an explicit interface support present in more conventional, statically typed object-oriented programming languages, such as Java, or C++ (as abstract classes). For a better understanding, this interface is imagined, as to express the need for an implementation of particular methods by some other application component.

As can be seen in Figure 4.4, the object structure is closely matching the logic system's syntactical structure (cf. Section 2.2.2's term grammar)—an abstraction enabling a clear reflection upon the internals of the evaluator system.

4.3 Compatibility and performance

The application is intended for use on personal computers equipped with a mouse or a trackpad, for it relies on mouse events, such as mouse-over for redex highlighting, and due to this fact, an experience on devices with smaller screens, such as tablets or smartphones, may be impractical and less satisfactory, although technically attainable.

The support of ES6 features is in modern web browsers practically complete,[g] and the application does not use any non-standard features or APIs, resulting in a product that is compatible with a wide range of target systems. The apparently only possibly occurring, cosmetic issues seem to be connected to support for the CSS' `box-shadow` property in older Microsoft and Apple browsers, which are nowadays seldom used.

A large section of the application's functionality is based on recursive procedures, as these are closely resembling the logic system's rules (for instance, the inductive definition of the substitution presented in Section 2.1.3), which are prone to stack overflow issues on very large inputs. Furthermore, due to the extensive feature set of the application—for example counting all the reducible expressions within a term in order to display such an information—the evaluator does carry out a numerous computational tasks on each step of the evaluation, which may be, in some cases, slowing-down the application. A decision to trade off some efficiency and "premature optimizations" for code simplicity and readability has been made.

[g] `https://kangax.github.io/compat-table/es6/`

Chapter 5

Conclusion

A lambda calculus evaluator application has been designed and implemented, providing rich feature set containing an optional step-by-step evaluation, multiple reduction strategies to choose from, term aliasing support with numerous predefined such aliases, besides the untyped system, *simple* and Hindley-Milner type system, various import and export options, including printing, flexible LaTeX source code generation, URL link generation, file input and output, a convenient help page, and several display preferences—an application that could hopefully be considered user-friendly and visually engaging, and it is, indeed, online.

Possible enhancements of the application would include some code optimizations, an implementation of automatically α-convertible aliases, that could be especially useful when dealing with Church encoding, with necessary variable renaming occurring in the course of an evaluation, an addition of a support for lists and tuples along with an appropriate syntactic sugar, enabling users to enjoy the infix notation, or perhaps an implementation of Girard's System F.

Appendix A

Predefined constants and aliases

	Alias	Constant	
name	*description and term*	*arity*	*description*
n	Church encoded natural numbers $n \in \mathbb{N}$, input up to 3 digits only $\lambda f.\lambda x.f^n\,x$	0	integer $n \in \mathbb{Z}$ primitives. JavaScript implementation-defined range
SUCC	successor function on Chuch numerals $\lambda nfx.f\ (n\ f\ x)$	1	successor function on constants
PRED	predecessor function, the least result is zero $\lambda nfx.n\ (\lambda gh.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)$	1	predecessor function, negative results allowed
PLUS	addition function on Church numerals $\lambda mnfx.m\ f\ (n\ f\ x)$	2	addition function on constants
MINUS	substraction on Church numerals, the least result is zero $\lambda mn.n$ PRED m	2	substraction on constants, negative results allowed
TIMES	multiplication on Church numerals $\lambda mnf.m\ (n\ f)$	2	multiplication function on constants
ISZERO	predicate testing if a Church numeral is zero, returning Church Boolean $\lambda n.n\ (\lambda x.\text{FALSE})\ \text{TRUE}$	1	predicate testing is a numeral constant is zero
LEQ	predicate testing if a Church numeral is less than, or equal to another $\lambda mn.\text{ISZERO}\ (\text{MINUS}\ m\ n)$	2	predicate testing if a numeral constant is less than, or equal to another
DIV	division on Church numerals *recursive definition below*	2	division function on integer constants
EQ	predicate testing if two Church numerals are equal $\lambda mn.\text{AND}\ (\text{LEQ}\ m\ n)\ (\text{LEQ}\ n\ m)$	2	equality function on numerical constants, returns Boolean constant
TRUE	Church Boolean true $\lambda xy.x$	0	Boolean true constant
FALSE	Church Boolean false $\lambda xy.y$	0	Boolean false constant
ITE	conditional, the first argument is Church Boolean $\lambda xyz.x\ y\ z$	3	conditional, the first argument Boolean constant, type system dependent semantics
OR	logical conjunction on Church Booleans $\lambda yx.y\ y\ x$	2	logical conjunction on Boolean constants
AND	logical disjunction on Church Booleans $\lambda yx.y\ x\ y$	2	logical disjunction on Boolean constants
NOT	logical negation on Church Booleans $\lambda x.x\ \text{FALSE}\ \text{TRUE}$	1	logical negation on Boolean constants
FIX	*undefined*	1	fixed point of the abstraction input
Y	fixed point combinator $\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$	*undefined*	
THETA	fixed point combinator, displayed as Θ $(\lambda xf.f\ (x\ x\ f))\ (\lambda xf.f\ (x\ x\ f))$	*undefined*	
OMEGA	Omega combinator, displayed as Ω $(\lambda x.x\ x)\ (\lambda x.x\ x)$	*undefined*	
S	pure λ-calculus S combinator $\lambda xyz.x\ z\ (y\ z)$	3	S combinator constant
K	pure λ-calculus K combinator $\lambda ij.i$	2	K combinator constant
I	pure λ-calculus I combinator $\lambda i.i$	1	I combinator constant

DIV $= \lambda n.Y\ (\lambda cnmfx.(\lambda d.\text{ISZERO}\ d(0\ f\ x)\ (f\ (cdmfx)))\ (\text{MINUS}\ n\ m))\ (\text{SUCC}\ n)$

Table A.1: *An enumeration of 21 non-numeral predefined aliases and 19 non-numeral constants.*

Appendix B

Input examples

A minimal term of a given property is one of possibly many minimal terms with such property, and it is minimal by character count.

Table B.1: My caption

Query	Description
Untyped λ-calculus	
p	minimal well-formed term
$(\lambda w.w)\,o$	minimal β-reducible term
$(\lambda k2.k9\;c4\;f22)\;w3$	input of a term with variable numerical indices
$(\lambda f x.f)\;x$	term with the variable renaming occuring
$\lambda f.x\;f$	minimal η-reducible term
$\lambda rasp.b(err)y$	input of abbreviated form
$(\lambda r.(\lambda a.(\lambda s.(\lambda p.((b((er)r))y)))))$	input of full syntactical form (the same term as above)
!PLUS !16 !32	addition using Church encoding
TIMES 256 256	multiplication using constants
LetRec f x = (ITE (EQ x 0) 1 (TIMES x (f (PRED x)))) In (f 5)	factorial of five using constants
LetRec e x = (ITE (EQ x 1) FALSE (ITE (EQ x 0) TRUE (e(PRED(PRED x))))) In (e 11)	recursive function deciding if the input (11) is even
Simply typed λ-calculus	
$\lambda x : \mathrm{Int}.x$	minimal well-typed $\lambda^{\rightarrow}$ term
q	minimal maltyped $\lambda^{\rightarrow}$ term
$(\lambda x : \mathrm{Bool}.ITE\;x\;2\;5)\;TIMES$	maltyped $\lambda^{\rightarrow}$ term
$((\lambda f : (\mathrm{Int} \rightarrow \mathrm{Int}).(f\;4))\;(PLUS\;2))$	well-typed $\lambda^{\rightarrow}$ term
Hindley-Milner type system	
$\lambda r.r$	minimal well-typed HM term
o	minimal maltyped HM term
$(\lambda ab.a)\;FALSE\;2$	well-typed HM term

Appendix C

Attached files

At the root directory, `lambda`, there are two `*.html` files located: `help`, which is a hypertext document containing the help page, and the main application frame, `index`, which is the evaluator access point itself, and three CSS files containing the visual layouts of the help page, the application, and a style used for print. A copy of the CC0 license is included in the root directory.

The `src` directory contains all of the JavaScript source code files. The files of capitalized names do contain definitions of objects and classes under the loosely matching names, and the `definitions.js` file does contain a predefined aliases and constants, which are loaded into the application by the `OLCE` object on page load.

The `resources` directory contains a web page `favicon`, which is a small image, often displayed in the web browser's tabs, and the Computer Modern typeface in two variants: italic and roman, along with an appropriate license in a textual format.

```
lambda/
├── index.html
├── help.html
├── help_style.css
├── print_style.css
├── app_style.css
├── LICENSE.txt
├── src/
│   ├── OLCE.js
│   ├── definitions.js
│   ├── Derivation.js
│   ├── Environment.js
│   ├── Evaluator.js
│   ├── Parser.js
│   ├── Term.js
│   └── Type.js
└── resources/
    ├── favicon.png
    └── fonts/
        ├── cmunrm.ttf
        ├── cmunti.ttf
        └── LICENSE.txt
```