Writing a Program to Evaluate Lambda Expressions

Yola Jones

August 15, 2019

Chapter 1

Background

1.1 Lambda Calculus

Lambda calculus is a formal syntax for expressing computation statements as mathematical expressions [24] created by Alonzo Church in 1936 [2]. It is an extremely simple model of computation, while still being Turing complete [22] and forms the basis for functional programming languages like Haskell [2]. Pure lambda calculus terms are comprised of three elements, variables (such as x or y), abstractions (with one input and one output, in the form λx.M) and applications (in the form MN) [12].

Applied lambda calculus expands this notation, adding constants which act as values and operations [25]. Typed lambda calculus extends this further, adding types to each term, allowing us to restrict the operations available to different types of objects and ensuring that our expression is runnable [17]. The combination of applied and typed lambda calculus gives us a grammar which can be used for type checking and arithmetically evaluating lambda calculus terms.

The Theory of Computation course teaches simply typed lambda calculus (applied lambda calculus with types), and includes lectures on alpha conversion and beta reduction, alongside the definition of normal form and typing rules. As a result, these will be covered in the following sections in an attempt to use lecture material to define requirements of the final system.

1.1.1 Basic Principles

As discussed above, the lambda grammar is made up of three elements: variables, abstractions, and applications. The lambda calculus is Turing complete, meaning any computable function can be expressed in this grammar [22].

Abstractions are representations of single input functions, which take one input, and use this input to compute one output. An example lambda abstraction is $(\lambda x.x+1)$, which is a representation of the function $f(x) = x + 1$, taking one input x, and returning the value of x incremented by 1. Multi-input functions can be expressed in nested abstraction terms, for example $f(x, y) = x + y$ in strict lambda notation becomes $\lambda x.\lambda y.x + y$.

Applications take abstractions and apply other lambda terms to them using substitution. In a term $(\lambda x.M)N$, substitution takes all the instances of the bound variable x and replaces them with the incoming variable N, or expressed mathematically as $(\lambda x.M)N = [N/x]M$. For example, the application $(\lambda x.x{+}1)3$ takes a function $\lambda x.x{+}1$, (or $f(x) = x{+}1$) and applies the number 3 to it, to become $3 + 1$. Substitution happens at the abstraction level, with the incoming value occurring as the result of an application of a term to an abstraction.

With abstractions, applications and variables, any computable function can be expressed. Functions and constants in applied lambda calculus simplify this slightly, allowing the number one to be expressed as the integer 1 rather than the Church Numeral encoding $\lambda f.\lambda x.fx$ in lambda calculus [29].

1.1.2 Alpha Conversion

Complexity is increased when having applications of variables to abstractions. The lambda term $\lambda x.\lambda y.yx$ translates to "take the first input to the function, and multiply it by the second input to the function". The point here is that x and y are distinct, different terms.

Applying y to this function would result in the application $(\lambda x.\lambda y.yx)y$ which would reduce to $\lambda y.yy$, the mathematical representation of taking the value of y and multiplying it by itself. This is incorrect, the meaning of the function has been changed because of the letter chosen to act as the input, and is an example of variable capture.

Alpha equivalence is the notion in lambda calculus that the choice of variable names does not matter, $\lambda y.y$ is alpha-equivalent to $\lambda a.a$. This notion is used to prevent variable capture through a process called alpha conversion, the process of comparing the incoming variable to the terms inside the abstraction and changing variable names to eliminate capture.

De Bruijn notation is an alternative method of handling this issue, eliminating the need for variables altogether and instead expressing abstraction terms as an index representing which abstraction scope they refer to [15].

The Barendregt Convention assumes bound variables are suitably chosen to avoid capture, a convention which is taught in the Theory of Computation course. From this and alpha conversion, a set of rules for substitution can be created which rename terms to avoid variable capture as substitution is taking place, defined as follows [1]:

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \tag{1.1}$$

$$[t/x](t_1 t_2) = [t/x]t_1[t/x]t_2 \tag{1.2}$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda z.[t'/x][z/y]t & \text{if } x \neq y \wedge z \notin FV(t) \cup FV(t') \end{cases} \tag{1.3}$$

The last rule changes the bound variable y to a variable z that doesn't belong to the set of free variables in the incoming term or the existing function t, simultaneous to the substitution $[t'/x]$ being performed.

1.1.3 Reduction

A formal method for evaluating lambda terms is known as a reduction, of which there are different levels which each perform different operations. Alpha conversion (or alpha reduction) [14], and is the process of renaming variables to avoid variable capture. Beta reduction focuses on finding the result of applications, discussed below. Delta reduction applies functions to constants arithmetically. Finally, eta reduction takes the output of delta- or beta-reduction and removes any leftover abstraction term in an attempt to simplify the final result [26].

Not all of these reduction steps are necessary, but each simplifies the term in some way, terminating with a final result.

Normal Form

After alpha conversion is performed, beta reduction is ready to be performed following the Barendregt Convention. Another rule is needed which defines when a lambda term has reached its simplest form and can be evaluated no further. This is known as the Normal Form of a lambda term [2], and is defined as the following [12]:

A term is in normal form if it has no subterms of the form $(\lambda x.M)N$

To account for applied lambda calculus, one more formalism is needed, δ reduction. Delta reduction is simply the evaluation of operations applied to constants, for example 4+2 delta reduces to 6 [25] [23]. The definition of normal form for applied lambda calculus then becomes [12] [25]:

A term is in normal form if it has no subterms of the form $(\lambda x.M)N$ and there are no delta rules which can be applied

Beta Reduction

Having defined the termination condition for evaluation, beta reduction can now be discussed. As previously stated, beta reduction is the process of finding the result from the application of a function, and can be performed by following a number of rules [8].

1. $(\lambda x.M)N = [N/x]M$

   $(\lambda x.M)N$ is equal to M with all instances of x replaced by N (substitution)

2. $M = M$

   A term is equal to itself

3. $\frac{M=N}{N=M}$

   If $M = N$ then $N = M$ through symmetric equality

4. $\frac{M=N \quad N=L}{M=L}$

   If $M = N$ and $N = L$ then $M = L$ through transitivity

5. $\frac{M=N}{MZ=NZ}$ and $\frac{M=N}{ZM=ZN}$

   If $M = N$, then Z applied to M is the same as Z applied to N. Similarly if $M = N$ then M applied to Z is the same as N applied to Z

6. $\frac{M=N}{\lambda x.M=\lambda x.N}$

   If $M = N$, then $(\lambda x.M) = (\lambda x.N)$

Different choices made in the application of these rules can generate different results. A fundamental theorem in the world of lambda calculus is the Church-Rosser theorem, and leads to a useful axiom: a term in the lambda calculus has at most one normal form [6] meaning different reduction strategies cannot produce different normal forms. However, if a lambda term does have a normal form, not all methods of reduction reach it [14].

The Theory of Computation course teaches two approaches to beta-reduction, call-by-value and call-by-need. Call-by-value is a leftmost innermost strategy, [8], meaning in a term MN, M gets reduced to its normal form before N (the left term is reduced first). In a term $(\lambda x.M)N$, N is evaluated before it gets substituted into $(\lambda x.M)$. For this reason, call-by-value is sometimes referred to as applicative order reduction, as the arguments are reduced before they are applied [27].

Call-by-need is similar but with one key difference, rather than reducing N and then substituting the result into M, the original N is substituted into M before $[N/x]M$ is reduced together as a whole. This is a subtle but very important difference, call-by-need can be proven to always turn a lambda term into its normal form if one exists [14], unlike call-by-value. Because of this property, call-by-need is often referred to as normal-order-reduction.

1.1.4 Typing Rules

A desirable property in software engineering is static checking at compile time. This can be done with lambda calculus by introducing types, which can be used to verify a lambda term is performing valid computations before being evaluated with delta reduction. For example, the lambda term $(\lambda x.x + 1)y$ is a valid term, but is not feasible if y is a boolean value such as TRUE or FALSE (since addition is not a valid operation with boolean values).

To use this, a type definition is first needed. A type T can be of ground type A (boolean or integer), or function type T→T (taking a type T, and converting it into another type T through a function) [8]. In order to type-check the validity of a lambda term, a typing context Γ is expressed which declares a set of variables to be associated with specific types. From this, three typing rules can be expressed [8]:

- For variables, if x of type T belongs to the typing context Γ, in Γ, x is of type T:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} TVar \tag{1.4}$$

- For abstractions, if in a typing context Γ where x has type T, if M has type U, then λx.M has type T → U:

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U} TAbs \tag{1.5}$$

- In an application MN, if M has type T→U and N has type T, then MN has type U

$$\frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U} TApp \tag{1.6}$$

This is all that is needed to statically analyse a lambda term, and determine whether or not a lambda term is typable, or feasible when considering typing and typing rules, andw hat type the lambda term should return, provided information about the types of some of the variables.

Type Inference

In a lambda expression, the type of a term can often be inferred without any explicit type declaration. For example, in an expression $\lambda x.x + 1$, x can be assumed to have type int, as $+$ is an in integer operator. This is type inference, and is the process of giving a variable a generic type (or supertype) and then refining this type based on gathered information until a minimal type is reached. [8] [3] [31]

This gathering of information can be split into two categories, structural and syntax [3]. Structural is done using lambda typing rules, for example using the fact that abstractions are function types of form $T \to T$ to find the final type of the term $\lambda x.x + 5$. The syntax of the term can also be used to gather information, for example using arithmetic or boolean operators to determine the type of a variable used in that expression.

Chapter 2

Analysis

As defined in the project description (defined somewhere else in this report), the focus of this tool is to help students taking the Theory of Computation course gain a better grasp lambda calculus. The point is not to teach them lambda calculus, but to allow them to evaluate lambda terms in an interface aimed at supporting their learning.

2.1 Lambda Calculus

The Theory of Computation course details [11] state three key intended learning outcomes relating specifically to Lambda Calculus: evaluate expressions in lambda calculus according to the definition of reduction relation, determine whether or not expressions in lambda calculus are typable in the simple type system, and implement lambda calculus expressions in a functional language such as Haskell or ML, or the functional fragment of Python.

From the lecture notes of the most recent academic year, the following topics are covered, most of which have been defined in the background section:

- Subterms
- Contexts
- Applied lambda calculus
- Barendregt convention and the definition of free and bound variables

  This is in contrast to the De Bruijn notation as discussed previously
- Beta reduction and normal form, including different reduction strategies (call-by-value and call-by-need)
- Typing rules and derivations

Since the goal of this project is to aid students understanding of lambda calculus, the learning outcomes and what is taught on the course will be focused on. Because of this, the following objectives can be selected:

- The Barendregt Convention is being chosen as a means of avoiding variable capture over De Bruijn notation, since it aligns more closely with what is taught on the course and therefore will better support student learning
- Simply typed applied lambda calculus will be implemented
- Subterms and nested terms will be allowed
- Contexts are not a large part of the course so will not be included in order to put focus on more prominent sections of lambda calculus such as reduction
- Beta reduction will be included, with options to evaluate terms using both call-by-value and call-by-need reduction strategies
- Typing derivations will be implemented in order to type check the lambda term and evaluate the final type of the reduced function

Further to what is taught in the course, there are other options which could be included in such an interface.

- Delta reduction has been discussed throughout the background section. While this is not taught explicitly, lecture slides do include constants being added together arithmetically, performing delta reduction implicitly.

  Because of this, the resultant lambda term will have arithmetic constants evaluated but delta reduction will not be explicit
- Eta reduction has also been discussed, but is not an integral step to lambda calculus, and most resources teaching lambda calculus miss it out [1] [2] [8] [14] [22] [12]

  Because of this, and because manipulating the final result in a way students have not been taught could cause further confusion, eta reduction will not be included in the final program

2.2 Abstract Syntax

To take a set of rules which define how a lambda calculus term should be processed and turn it into a program which can evaluate lambda terms, a way of representing the structure of a lambda term is needed. Once this has been completed, the resultant program can use the defined rules to evaluate terms.

The idea behind this is to turn a sequence of character strings (the surface syntax of an expression) into an abstract syntax (a representation of the meaning of the expression). A lexer is used to do this, which converts a character sequence into a series of identifiers called tokens, as defined by the grammar of a language (in this case lambda calculus, the grammar of which is defined above). A parser is then used which turns this sequence of tokens into a tree known as an Abstract Syntax Tree, which can then be traversed and explored to create compilers and interpreters of a language [21]. The applications of this are wide, it can be used to evaluate arithmetic expressions 2.1, in syntactic analysis used for natural language processing [4] 2.2, or to evaluate lambda calculus terms, as done in this project 2.3.

2.3 Antlr

One such tool which can help represent the structure of a term based on a particular grammar is Antlr. Antlr is a parsing tool which is used to build and explore syntax trees from character strings. It has a lexer to generate tokens, and has a parser to turn this stream of tokens into an abstract syntax tree which can then be evaluated using either a listener or a visitor [19]. It has been used in a wide variety of contexts, including Twitter (for query parsing) and the NetBeans IDE (for parsing C++).

2.4 Interface

Once a program has been created which can parse and evaluate lambda terms, an interface is needed to allow this to be used by students. The interface should be simple to understand, and as a minimum requirement should include a place for user input which will allow students to input a lambda expression alongside an output showing the resultant reduced expression. It should also include an input which will allow the user to select different types of beta-reduction, and an output which will show the typability and evaluated type of the returned expression.

The interface will not be connected to any central database containing personal information, nor will it require a login or store any personal data at all, so does not need to be protected from SQL injection.

However, it will allow users to type information into the interface so should be protected from HTML injection, cross site scripting and buffer overflow attacks.

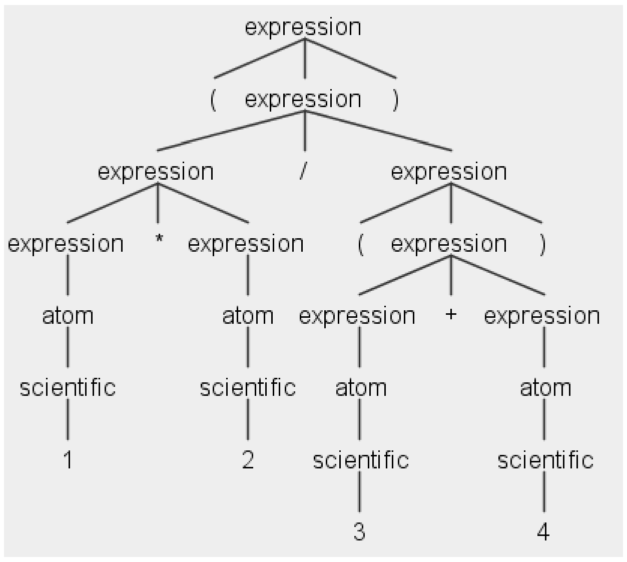


Figure 2.1: Syntax Tree of an Arithmetic Term (generated using code from [9])

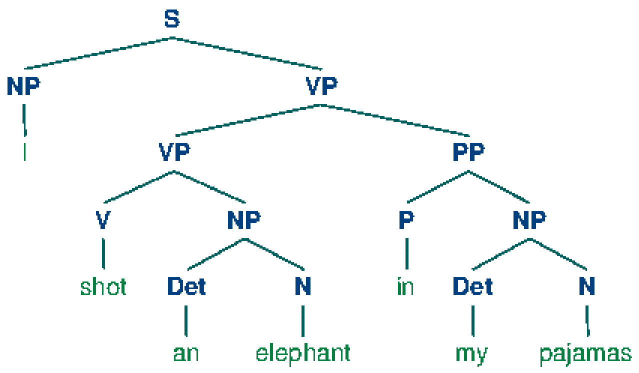


Figure 2.2: Syntax Tree for Natural Language Processing [4]

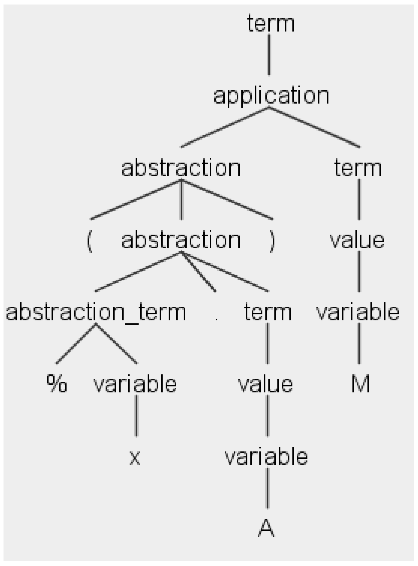


Figure 2.3: Lambda Calculus term $(\lambda x.A)M$ abstract syntax tree

Chapter 3

Aims and Objectives

From the background and analysis discussed above, a set of key aims and objectives can be defined. These eight objectives have been defined below, and have been labelled with the MoSCoW system, (must-have, should-have, could-have and would-like-to-have) [7]. The program will include:

- Must Have: The ability to evaluate the validity of lambda calculus terms, so students can enter terms and check whether or not they are valid. This will be done by turning the user input lambda term into an Abstract Syntax Tree using Antlr
- Must have: Applied lambda calculus, to allow the input of expressions
- Must have: The ability to alpha convert these terms, to help students understand alpha conversion and equivalence. This will be done by navigating the abstract syntax tree generated by Antlr
- Must have: Which will be contained in a web interface to allow users to use the tool easily
- Should have: Using beta reduction to evaluate terms to normal form to help students understand beta reduction. This will also be done by navigating the abstract syntax tree generated by Antlr
- Should have: Which includes types, to help students understand types and typing rules
- Could have: Which offers different reduction strategies, to give students the option to understand call by value/call by need evaluation
- Would like to have: Which performs type inference to determine the output type of a lambda term
- Would like to have: And arithmetically evaluates these terms using delta reduction

Chapter 4

Design and Implementation

4.1 Design

The overall program is split into a number of distinct elements, all of which are based on the fundamental rules of Lambda Calculus as defined in previous sections.

Firstly, a grammar is written which defines the syntax of lambda terms and allows a term to be broken down into individual tokens. Antlr uses this grammar to process lambda terms, creating an Abstract Syntax Tree for each incoming expression[19], a process which has been well documented and therefore will not need to be rewritten. A tree traversal mechanism is the main section of code which navigates the abstract syntax tree and performs operations to determine the resultant expression. This is the section of code which will perform the evaluation, and therefore needs to be designed effectively, with the rules of beta reduction and lambda calculus in mind. A web interface is used to house this program, giving the user a simple and convenient way to interact with the underlying system.

The relevant sections have been discussed below, detailing design and key decisions which needed to be made before implementation began.

4.1.1 Grammar

The key aim in creating this grammar was creating a syntax which sticks as closely as possible to the rules of lambda calculus. Lambda Calculus grammar is already clearly defined, with a lambda term being either a variable, an abstraction or an application [12]. Applied lambda calculus adds functions and constants to this definition [25], resulting in the following lambda grammar. A lambda term is one of the following:

| An application (of form `[term] [term]`)

| An abstraction (of form `[abstraction_term].[term]` where `[abstraction_term]` is defined by λ`[variable]`)

| A function (of form `[term] [operation] [term]`)

| A value (of form `[variable]` (the letters a-z) or `[number]` (constant))

With the addition of types, the grammar adds the option of a `:[type]` term to each variable, with each type being either a ground type (bool, int or none), or in the form `[type]` → `[type]`. This follows the standard syntax for typing used in the lecture material [12] [8], and consistency with this syntax allows students on the Theory of Computation course to input a lambda term directly from the lecture slides with minimal adjustment.

4.1.2 Expression Evaluator

The fundamental component of this evaluator is a tree traversal, which navigates through the token nodes and performs different operations depending on the type of node encountered. For example, application tokens in the form MN will pass the right-hand term N to the left-hand term M. Abstractions will take an incoming value if one exists, and substitute it into the body of the abstraction. This allows an evaluated expression to be built up, and a result determined.

Antlr provides two mechanisms for traversing an abstract syntax tree: listeners and visitors. A listener is a passive way of evaluating a syntax tree, used within an antlr Walker class which traverses the tree using a depth-first approach, triggering methods from the listener as it enters and exits each token [19]. These listener methods are unable to return values, so expressions and evaluations have to be handled using separate objects within the listener class. As the walker traverses the tree, the listener builds up a running evaluation of the term, returning the result when it exits the topmost node [28].

Unlike listeners, visitors control their own traversal of the tree. By visiting the children of each node encountered explicitly, the path they take around the tree can be controlled [19], allowing some children to not being visited until their parents are evaluated, or a right-hand child of a node being visited and evaluated before its left.

Visitors also allow custom return types, meaning nodes can return their resultant expressions directly to their parent node and do not have to rely on separate objects [28].

With beta reduction, different methods take different approaches to evaluating terms. In an application MN using a call-by-value approach, N is evaluated before M. In a call-by-need approach, N is passed into M before being evaluated. This means that depending on the type of reduction selected, the evaluator will have to traverse the tree in a different order, suggesting visitor over listener is more appropriate for this task. The fact that evaluation happens as the tree is being visited supports this further, since there will be a large amount of data being passed around the tree. Having a separate object for storing these values could get complex, and so the visitor methods being able to return values directly to their parents will be more convenient for this task.

The visitor therefore was the main code written in this project. An Antlr generated parser is passed to a custom visitor interface. Different visitors are defined for each of the beta-reduction strategies being implemented, since each method takes a different approach to evaluating expressions.

The visitor should return three separate items upon returning to the topmost node: the evaluated value of the input lambda term, whether or not the term is typable, and what type the expression will be. It will also return details of any errors where applicable, for example syntax errors which Antlr is unable to parse, or cases where the normal form of a lambda term does not exist.

4.1.3 Web Interface

The web interface will allow the user to input a lambda term along with the type of any variable. It will also allow the user to select which reduction strategy they would like to have the term evaluated by, with call-by-need (or normal order reduction) being selected as the default.

The code running the interface and the underlying evaluation code will be kept as separate entities, communicating through input parameters and return statements. This is to ensure the code is kept as modular as possible, allowing the evaluator code to be run using multiple different mechanisms, for example through a web interface and a command line for testing purposes. Any data the user enters will be passed to the underlying code, which will process the term and return any output via a return statement, being read and displayed to the user.

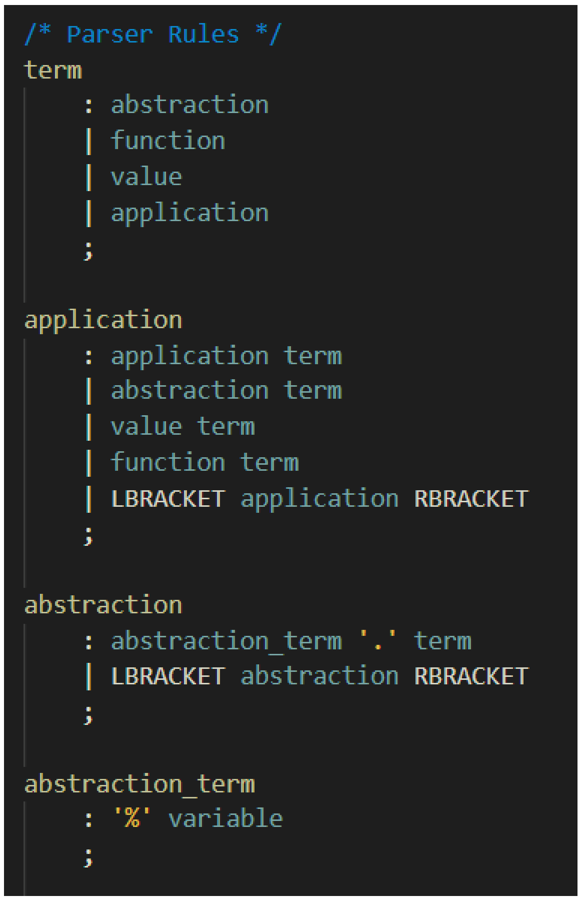
The layout of the interface should be simple and uncluttered, and should be suitable for those with a visual impairment.

4.2 Implementation

Despite the examples in the Antlr Reference Guide [19] all being written in Java, Python was chosen as the main language for this project. Antlr offers support for Java and Python, both of which are regularly listed in the top three programming languages being used in industry from various sources [13][20][5]. Since the MSc Software Development course which this dissertation is being written for teaches only Java, Python was chosen in order to expand the number of programming languages encountered throughout the course. Therefore the majority of the code for this project will be written in Python, with HTML being used for the web interface.

4.2.1 Grammar

Antlr requires that the grammar of a language is contained in a .g4 file which defines the parser and lexer rules for lambda calculus in this case. A section of the grammar is shown in Figure 4.1, and indicates the parser rules for some of the token nodes, alongside the lexer rules which are shown in Figure 4.2. This grammar is consistent with the syntax defined in the background section of this report, with slight modifications such as the addition of parentheses and an explicit definition for boolean and integer values.



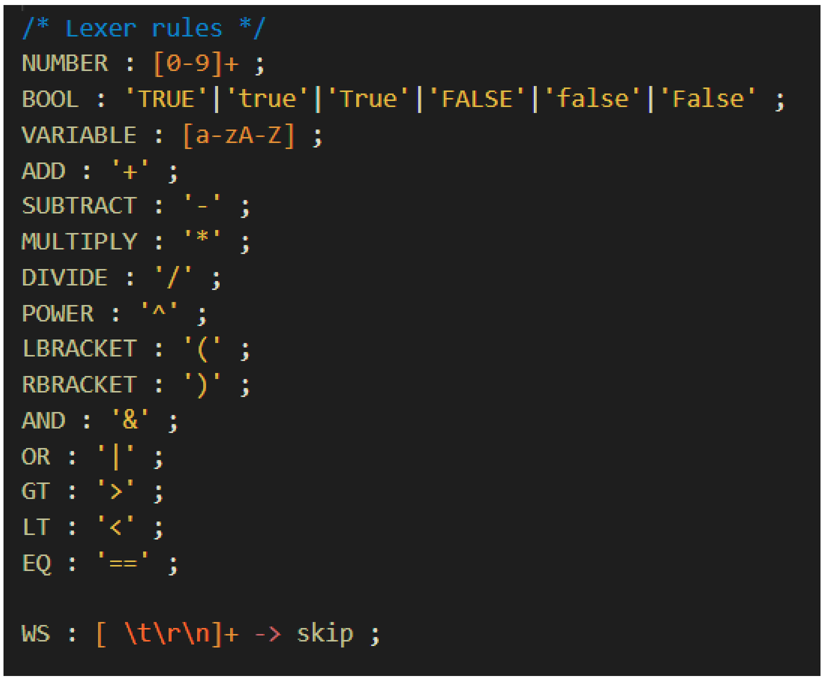
```
/* Parser Rules */
term
        : abstraction
        | function
        | value
        | application
        ;

application
        : application term
        | abstraction term
        | value term
        | function term
        | LBRACKET application RBRACKET
        ;

abstraction
        : abstraction_term '.' term
        | LBRACKET abstraction RBRACKET
        ;

abstraction_term
        : '%' variable
        ;
```

Figure 4.1: Parser Rules



```
/* Lexer rules */
NUMBER : [0-9]+ ;
BOOL : 'TRUE'|'true'|'True'|'FALSE'|'false'|'False' ;
VARIABLE : [a-zA-Z] ;
ADD : '+' ;
SUBTRACT : '-' ;
MULTIPLY : '*' ;
DIVIDE : '/' ;
POWER : '^' ;
LBRACKET : '(' ;
RBRACKET : ')' ;
AND : '&' ;
OR : '|' ;
GT : '>' ;
LT : '<' ;
EQ : '==' ;

WS : [ \t\r\n]+ -> skip ;
```

Figure 4.2: Lexer Rules

4.2.2 Abstract Syntax Tree

Having defined a grammar, Antlr can then be used to create an abstract syntax tree, which can be seen for an example term in Figure 4.3. This is very simple to do using Antlr libraries, and involves passing the input term through an Input Stream class, feeding the resultant string of characters into a lexer (which outputs a set of tokens), a class which converts the tokens into an indexable list, and finally passing this through a parser which turns the tokens into an abstract syntax tree[30]. The lexer and parser are created by Antlr when the grammar file is compiled, resulting in a syntax tree which processes terms based only on the syntax of the defined language.

The process of creating a tree from the grammar and an incoming term is well documented, and so no modifications need to be made, with the exception of attaching a custom Error Listener to both the lexer and parser, as expanded on later on in this Implementation section.

4.2.3 Expression Evaluator

This is the largest block of code in the program, and defines how the incoming lambda term should be evaluated depending on the input given by the user. For the purposes of simplicity this code can be split into three distinct sections, alpha conversion, beta reduction and typing rules.

Alpha Conversion

The alpha conversion is based on the rules for explicit alpha conversion with substitution, as defined by [1], and as discussed in detail in a previous chapter but which are repeated below for clarity:

$$[t/x]y = \begin{cases} t & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$[t/x](t_1 t_2) = [t/x]t_1[t/x]t_2$$

$$[t'/x](\lambda y.t) = \begin{cases} \lambda y.t & \text{if } x = y \\ \lambda z.[t'/x][z/y]t & \text{if } x \neq y \wedge z \notin FV(t) \cup FV(t') \end{cases}$$

The final rule is the rule to be focused on, and defines the process of finding a variable z that does not appear in the free variables of the incoming term or the existing term, replacing all bound variables y with this new term z, and then substituting in t' as normal.

The alpha conversion code written for this project follows this process. First, the set of free variables in the term are determined, by taking the set of all alphabetic characters in the term and eliminating the bound variables. These variables are then replaced with letters that are not in the list of free variables, in situations where a clash in free variables between the two expressions are found. This produces an alpha-converted term, into which substitution can be performed using a simple `String.replace()` method in Python.

Beta Reduction

Two beta reduction strategies are taught on the Theory of Computation course, call-by-value and call-by-name. The differences between these two strategies have been discussed in detail, but the key difference is in the order in which substitution and evaluation occurs in an application MN.

Because of these differences in evaluation strategy, two separate visitors are used to limit the use of control coupling [16]. However, since their differences are only in evaluation order with the rest of the functionality being the same, a BaseVisitor was defined which contains all common code between these two strategies. The two call-by-value and call-by-need visitors are subclassed from this base visitor, and define their unique behaviour for the application and abstraction terms.

The abstraction term differs between the two methods due only to typing, since determining the type of a term happens during the evaluation of that term. In call-by-value, the type of N is known before substitution, so can be carried throughout the function. In call-by-need, the whole term needs to be type checked after substitution has happened to determine the type of M with N incorporated. Two separate abstraction methods were therefore needed, with common code moved into a `process_abstraction()` method defined within the base visitor.

Typing Rules

Lambda Calculus has clearly defined typing rules which are taught in the Theory of Computation course. These rules have been discussed in more detail previously, but are repeated below for clarity:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} TVar$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U} TAbs$$

$$\frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash MN : U} TApp$$

Each type can either be of ground type T, or function type T→U [12].

Since the visitNode methods already evaluate their children nodes, these typing rules can be integrated directly into these nodes, with each visitNode method returning a value and a type. Type checking happens throughout the code in each method where evaluation occurs, and is implemented as follows:

Variable
Variables simply return the type given to them by the user, as defined by Rule 1.4. Any number is given type `int`, and any boolean values TRUE or FALSE are given type `bool`.

Abstraction

Users enter types in the form `x:type`, which can be applied to any variable included in the term. The abstraction method takes the input type T, and joins it with the output type with `->`, to become of function type `input->output`. The input and output types are either specified by the user, or determined through type inference when the body of the abstraction is a function.

Application

For an application MN, application typing involves iteratively removing the first ground type of M and the first ground type of N until the type of N is None. For example, if M is of type `int->bool->int` and N is of type `int->bool` the typing of application MN is of type `int` through the following process:

$$M : int \to bool \to int \qquad N : int \to bool$$

$$M : bool \to int \qquad N : bool$$

$$M : int \qquad N : None$$

If at any point the first ground type of N does not match the first ground type of M, the typing is invalid (for example if N was of type `bool->bool` in this case). This is implemented using string manipulation, splitting a type T into its individual ground type using the `split()` method in Python, then comparing and removing the types of M and N iteratively.

Type inference uses the typing rules defined above to determine the type of an output term where possible. Abstractions are initially given function type `None`→`None`, and variables are given ground type `None`. This is inference using structural information.

In the visitFunction node, syntactical information can be used to determine the input and output types of terms by examining the operation term. The following is declared in code:

- The operations {&,|} take two boolean values and return a boolean. Therefore the type of the incoming and outgoing term can be inferred to be of type boolean
- The operations {+,-,*} take two integer values and return an integer. Similar to above, the incoming and outgoing term can be inferred to be of type integer
- The operations {==,>,<} take two integer values and return a boolean, it can therefore be inferred that the input term is of type integer, and the output is of type boolean.

The output type of a function is used to determine the output type of its parent node. In an abstraction, this type inference can be used to determine the type of its input term, for example the lambda term $\lambda x.x + 1$ can be inferred to have type `int->int` despite no input type given by the user.

There is a limit to this type inference, the function will infer the input type when there is only one operation term. For example, the term $\lambda x.(x == 1)\&b$ contains two operations, $==$ and $\&$. In this case, working out the input type is more complicated, since the typing has to be broken down in to sub-functions, which need to determine what they think the input type should be, which then needs to be examined collectively as a complete term. The type of this output is returned by the program therefore as `None->bool`. This is correct, just not as minimal as its alternative type `int->bool`.

While narrowing type inference to its most minimal state is definitely possible, it is not the key goal in the project, and is a small edge-case when it comes to improving student understanding. Because of this, and due to the finite nature of this project, this limitation has been deemed acceptable, with other tasks which more greatly contribute to students understanding of lambda calculus being prioritised.

Error Handling

Each visit method in the visitor returns an evaluated result and a type, which is used to build the evaluated term and determine the final result, type and type validity of the input expression.

However, during this process, there are a number of occurrences which could stop the program from being able to return a final result. These are broken down into two key issues: syntax errors and occurrences where a term doesn't have a normal form (and therefore cannot be evaluated to termination). These have been summarised below.

Syntax Errors
Lambda terms often include nested parentheses. It is easy for a user to mismatch brackets, such as forgetting to close an open bracket. Because of this, before the term is passed to antlr, the code checks to ensure brackets are matched correctly, and if not, returns an error to the user informing them of the issue and asking them to re-enter.

Aside from this, syntax errors specific to lambda calculus are likely to occur which cannot be picked up until the term is passed through a lexer. To handle this, Antlr's ErrorListener class is overridden, instead throwing a custom SyntaxTokenError exception which can be caught by the program and passed back to the user.

Recursion Errors
Recursion errors are thrown in Python when a maximum stack depth is reached in order to protect Python from crashing [10]. This is the exception which is thrown when a lambda term has no normal form, as the code keeps trying to evaluate the term before eventually throwing a RecursionError.

To handle this, the recursion limit for the code is set to 200, smaller than the python default to prevent excessive time waiting for the program to fail, but large enough to ensure that any reasonable lambda term can be processed. A recursion limit for the code equates to a lambda term with approximately 50 nested abstractions (since each abstraction could potentially be comprised of the following nodes: its parent term, the abstraction contained within parentheses, the actual abstraction and its inner terms). While this is a limitation, it's is deemed to be a reasonable one, as limiting students to 50 nested abstractions as opposed to 100 is unlikely to impact student understanding.

If a recursion error is thrown, it is caught by the code in a `try/except` block, and an error message informing the user that a normal form cannot be found for this term is given.

Web Interface

Flask is a web framework designed for use with Python, and allows python scripts to be controlled from a web interface [18]. The websites' front-end is written in HTML, and is connected to the python file using Flask's render_template() function.

Users enter lambda terms in the input dialogue box, entering a % sign as a lambda symbol. Using a HTML onkeydown event, the interface automatically changes this symbol to a λ, allowing users to easily enter lambda terms. The complete web interface can be seen in Figure 4.4, Figure 4.5 and Figure 4.6.

Clicking the *Check Expression* button on the web interface sends a HTTP post request to the Flask server, which takes the value held in the user input box along with the selected reduction strategy, and sends this to the main lambda program. The output is returned to the HTML code through Flask, and the webpage is re-rendered to display the results.

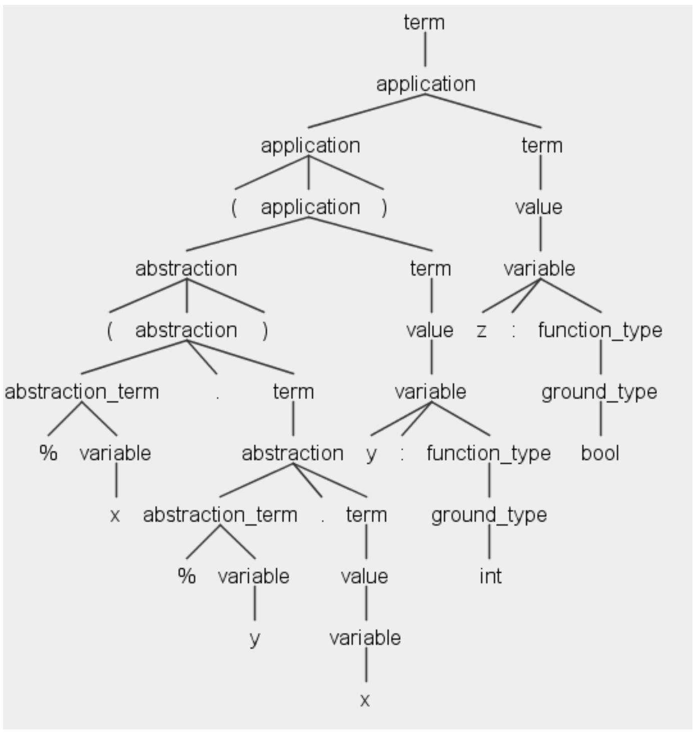
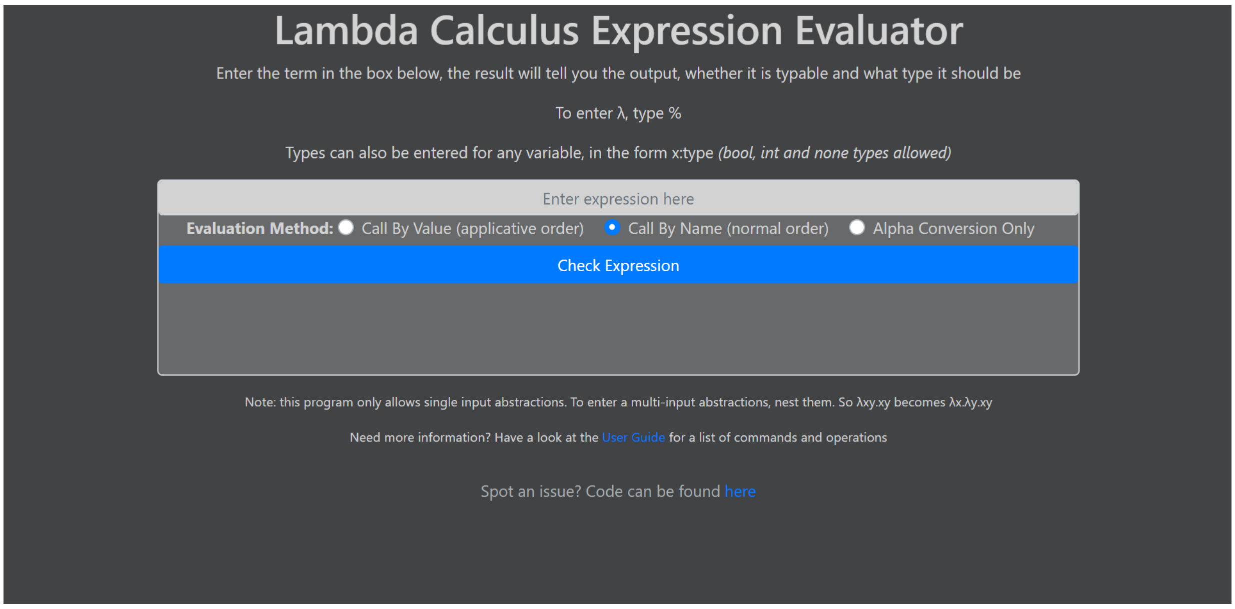


Figure 4.3: Abstract Syntax Tree Example for ((λx.λy.x)y:int)z:bool



Lambda Calculus Expression Evaluator

Enter the term in the box below, the result will tell you the output, whether it is typable and what type it should be

To enter λ, type %

Types can also be entered for any variable, in the form x:type *(bool, int and none types allowed)*

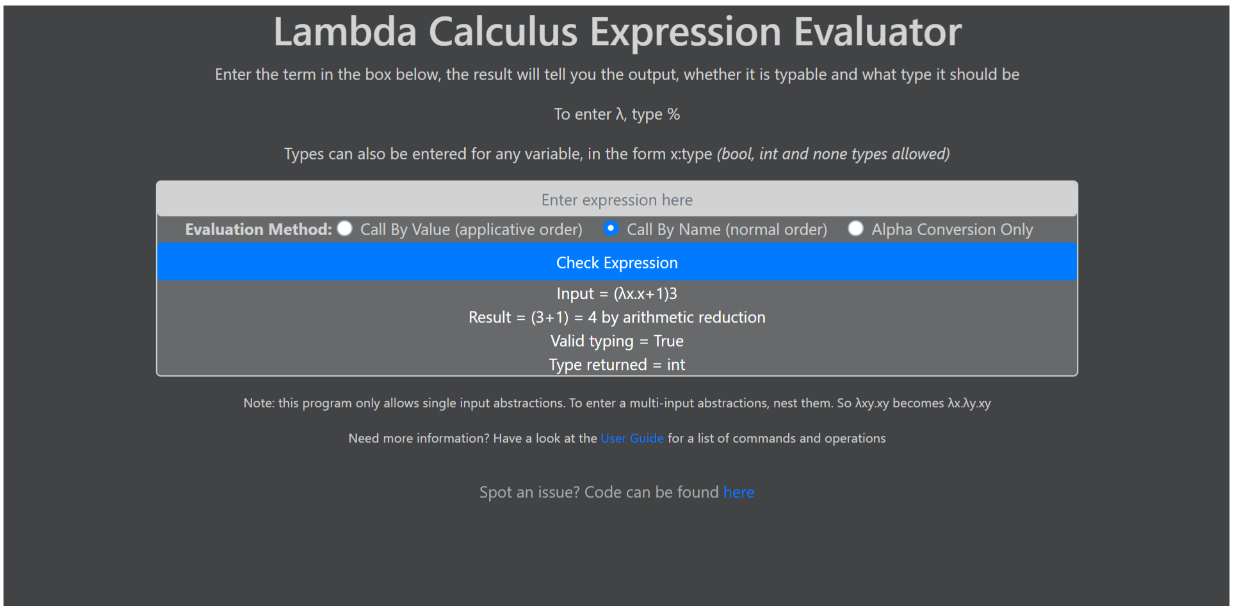
Evaluation Method: Call By Value (applicative order) [✓ Call By Name (normal order)] Alpha Conversion Only

Note: this program only allows single input abstractions. To enter a multi-input abstractions, nest them. So λxy.xy becomes λx.λy.xy

Need more information? Have a look at the User Guide for a list of commands and operations

Spot an issue? Code can be found here

Figure 4.4: Web Interface


Lambda Calculus Expression Evaluator

Enter the term in the box below, the result will tell you the output, whether it is typable and what type it should be

To enter λ, type %

Types can also be entered for any variable, in the form x:type *(bool, int and none types allowed)*

Enter expression here

Evaluation Method: ○ Call By Value (applicative order) ● Call By Name (normal order) ○ Alpha Conversion Only

Check Expression

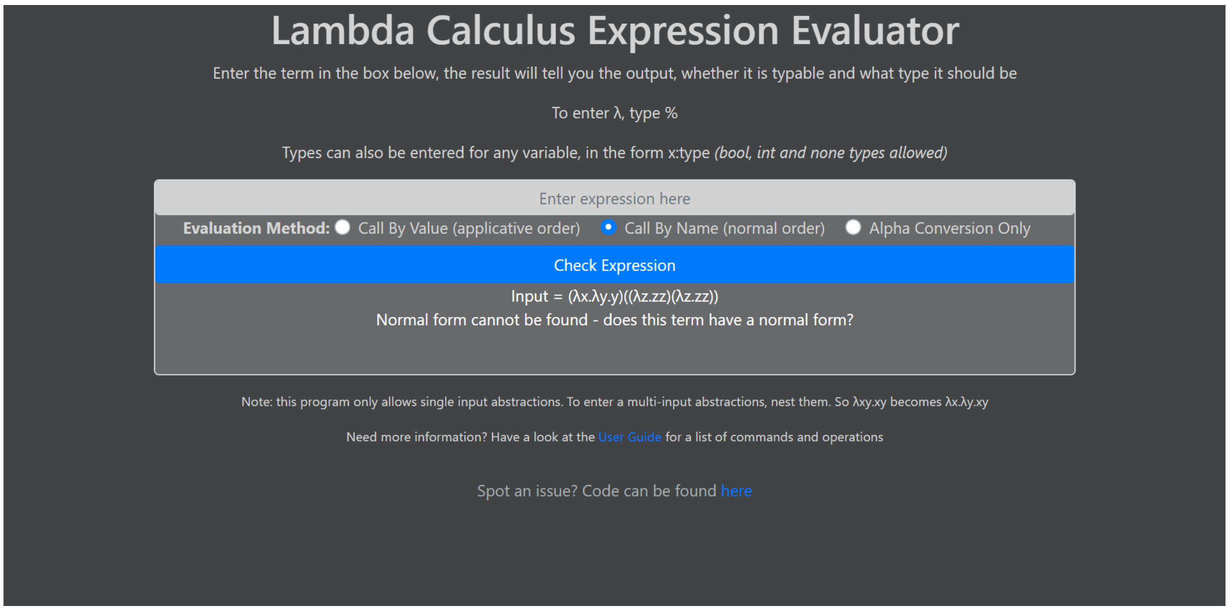
Input = (λx.x+1)3
Result = (3+1) = 4 by arithmetic reduction
Valid typing = True
Type returned = int

Note: this program only allows single input abstractions. To enter a multi-input abstractions, nest them. So λxy.xy becomes λx.λy.xy

Need more information? Have a look at the User Guide for a list of commands and operations

Spot an issue? Code can be found here

Figure 4.5: Web Interface With Returned Result


Lambda Calculus Expression Evaluator

Enter the term in the box below, the result will tell you the output, whether it is typable and what type it should be

To enter λ, type %

Types can also be entered for any variable, in the form x:type *(bool, int and none types allowed)*

Enter expression here

Evaluation Method: ○ Call By Value (applicative order) ● Call By Name (normal order) ○ Alpha Conversion Only

Check Expression

Input = (λx.λy.y)((λz.zz)(λz.zz))
Normal form cannot be found - does this term have a normal form?

Note: this program only allows single input abstractions. To enter a multi-input abstractions, nest them. So λxy.xy becomes λx.λy.xy

Need more information? Have a look at the User Guide for a list of commands and operations

Spot an issue? Code can be found here

Figure 4.6: Web Interface With Error

Chapter 5

Testing

The aim of this project is to build a web application which will verify and evaluate lambda calculus terms input by the user, with the goal of improving student understanding.

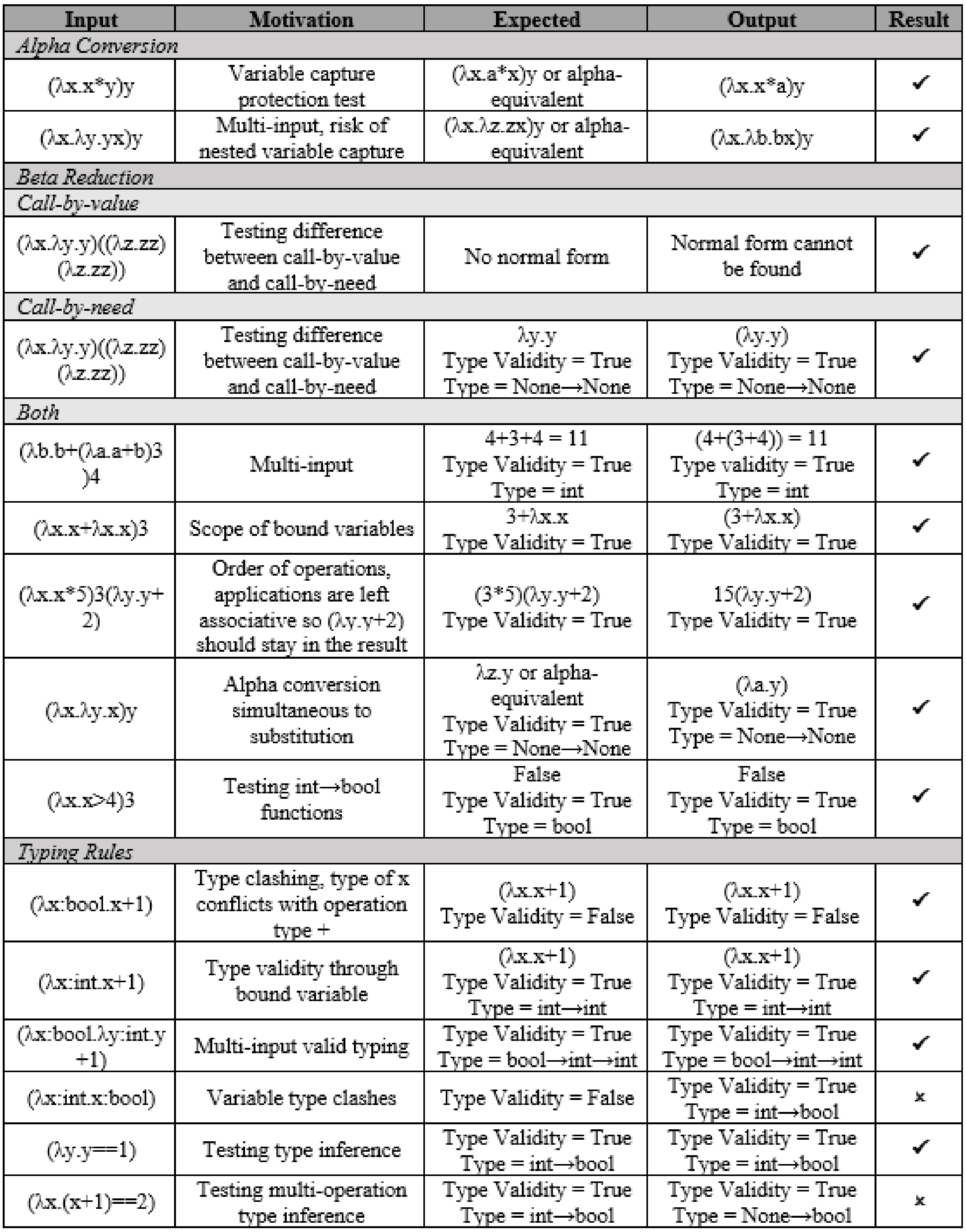
This can be broken down into two key goals for the created tool, the interface should evaluate lambda terms accurately and should be useful for students learning functional programming.

Because of this, the testing will be broken into sections in accordance with these two goals. The first set of tests will evaluate the accuracy of the lambda term evaluation. The second set of tests will test the usability of the interface and whether or not it meets the goal of improving student understanding, by performing a user test with students who have an understanding of computer science but who have little to no knowledge of lambda calculus.

5.1 Accuracy of Lambda Term Evaluation

In order to test the accuracy of the evaluation, the tests were split into three categories: alpha-conversion, beta-reduction and typing rules. For each category, key points of potential failure were identified, such as multi-input lambda terms, bound variable naming clashes (λx being used twice in the same term testing the scope of the lambda term), order of operations issues (since applications are left-associative), and terms with no normal form.

All of the above were tested for, the results shown in Figure 5.1 below. The table includes the input to test, the motivation for testing that input, the expected and actual results, and an indicator showing whether or not the test was successful.



Input	Motivation	Expected	Output	Result
Alpha Conversion				
(λx.x*y)y	Variable capture protection test	(λx.a*x)y or alpha-equivalent	(λx.x*a)y	✓
(λx.λy.yx)y	Multi-input, risk of nested variable capture	(λx.λz.zx)y or alpha-equivalent	(λx.λb.bx)y	✓
Beta Reduction				
Call-by-value				
(λx.λy.y)((λz.zz)(λz.zz))	Testing difference between call-by-value and call-by-need	No normal form	Normal form cannot be found	✓
Call-by-need				
(λx.λy.y)((λz.zz)(λz.zz))	Testing difference between call-by-value and call-by-need	λy.y Type Validity = True Type = None→None	(λy.y) Type Validity = True Type = None→None	✓
Both				
(λb.b+(λa.a+b)3)4	Multi-input	4+3+4 = 11 Type Validity = True Type = int	(4+(3+4)) = 11 Type validity = True Type = int	✓
(λx.x+λx.x)3	Scope of bound variables	3+λx.x Type Validity = True	(3+λx.x) Type Validity = True	✓
(λx.x*5)3(λy.y+2)	Order of operations, applications are left associative so (λy.y+2) should stay in the result	(3*5)(λy.y+2) Type Validity = True	15(λy.y+2) Type Validity = True	✓
(λx.λy.x)y	Alpha conversion simultaneous to substitution	λz.y or alpha-equivalent Type Validity = True Type = None→None	(λa.y) Type Validity = True Type = None→None	✓
(λx.x>4)3	Testing int→bool functions	False Type Validity = True Type = bool	False Type Validity = True Type = bool	✓
Typing Rules				
(λx:bool.x+1)	Type clashing, type of x conflicts with operation type +	(λx.x+1) Type Validity = False	(λx.x+1) Type Validity = False	✓
(λx:int.x+1)	Type validity through bound variable	(λx.x+1) Type Validity = True Type = int→int	(λx.x+1) Type Validity = True Type = int→int	✓
(λx:bool.λy:int.y+1)	Multi-input valid typing	Type Validity = True Type = bool→int→int	Type Validity = True Type = bool→int→int	✓
(λx:int.x:bool)	Variable type clashes	Type Validity = False	Type Validity = True Type = int→bool	✗
(λy.y==1)	Testing type inference	Type Validity = True Type = int→bool	Type Validity = True Type = int→bool	✓
(λx.(x+1)==2)	Testing multi-operation type inference	Type Validity = True Type = int→bool	Type Validity = True Type = None→bool	✗

Figure 5.1: Evaluating the Accuracy of Lambda Term Evaluation

Figure 5.1 demonstrates that in the majority of cases which have been tested, the evaluation is accurate and does yield expected results. There are a few instances where the output is not as expected, which have been discussed in greater detail in the Evaluation section of this report.

5.2 Interface Usefulness and Usability

5.2.1 Testing

In order to test the usability and usefulness of the interface, a user test was conducted using participants who had a computer science background but who had little to no understanding of lambda calculus. This was done in order to simulate students on the Theory of Computation course who have just been introduced to Lambda Calculus for the first time.

Participants were first given a Participant Information sheet and to sign a Participant Consent Form, which stated they were happy with the results of their user test to be included in this dissertation, provided their answers will be kept anonymous as will be done here.

They were then asked to read a document giving them a brief introduction to lambda calculus, which was written for this test and was based on the Theory of Computation lecture slides [8]. This was done to simulate students who have been introduced to lambda calculus, but are still very new to the subject.

Having read a brief summary, participants were asked to rate their understanding of lambda calculus on a scale of 1-10, before being given the interface to test. They were provided with a few sample lambda terms to test if they wanted to, but were not obligated to use them. This was to simulate students who may be using the tool to evaluate a particular lambda expression they have found in the lecture slides or online.

Finally, users were given a questionnaire which again asked them to rate their understanding of lambda calculus having used the interface for a short period of time, so their understanding could be compared before and after using the tool.

The questionnaire also asked the following questions:

- Would you use a tool like this if you were studying the Theory of Computation Course?

  Yes/no/other (please detail...)

- Did you find it useful in improving your understanding?

  Yes/no/other (please detail...)

- What did you think of the interface?
- Do you have suggestions for any improvements that could be made?

- Any further comments?

5.2.2 Results

Although the test was not conducted with a time limit, each participant spent approximately five minutes using the interface before stating they were ready to move on. As can be seen from Figure 5.2, in the five minutes participants interacted with the interface their understanding improved by 27% on average. 100% of participants said they would use the tool created if they were on the Theory of Computation course, and 100% said they found it useful in improving their understanding.

The qualitative questions on the interface and tool revealed that participants found the system simple and easy to use, with some suggestions for improvement. Suggested interface changes included making the colour of the input box darker to differentiate between input boxes and menus, and changing the background as some participants found it distracting. Participants also suggested adding additional features such as tooltips to give the user more information on what particular lambda terms meant, and adding a workbook section which would allow users to save the results to a file in order to create a study resource they could go back to later.

Overall the feedback was incredibly positive, supporting the conjecture that the tool created could support student learning for those starting out with lambda calculus. Participants commented that it helped improve understanding by adding a practical element to the much more theoretical-based lecture slides.

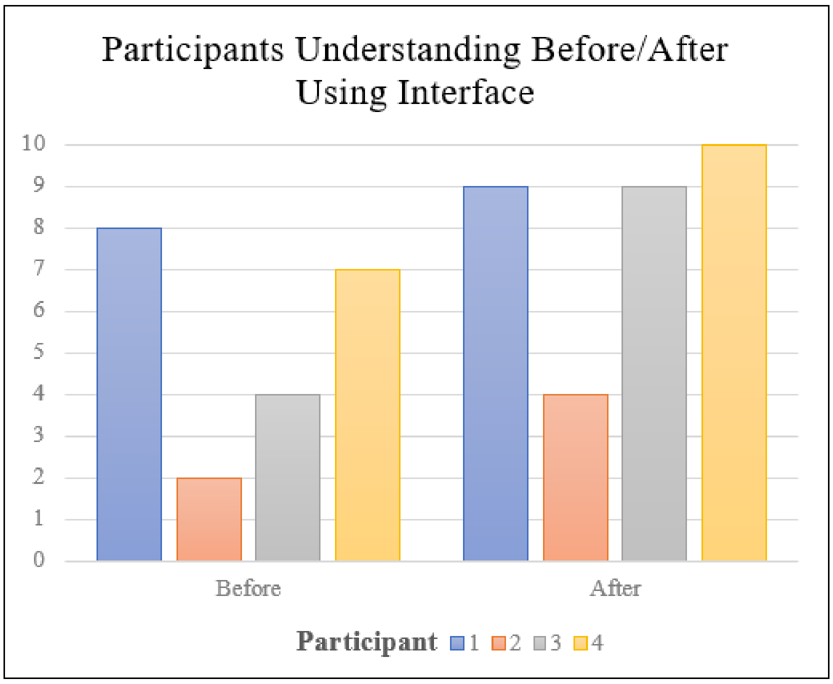


Figure 5.2: Participant Understanding Results

Chapter 6

Evaluation

As stated in the beginning of this report, the project summary is as follows:

> *The aim of this project is to build a web application which will verify and evaluate lambda calculus terms input by the user. This tool will be created with the goal of helping future students learning functional programming to gain a better understanding of lambda calculus.*

This chapter will discuss whether or not the project description has been met, and whether or not the tool created could be used as a supplementary resource for students learning lambda calculus on the Theory of Computation course taught at the University of Glasgow.

6.1 Accurately Evaluating Lambda Terms

As evidenced by section 5.1, the tool created does accurately evaluate lambda terms, returning the normal form, type and type validity of lambda terms for both call-by-value and call-by-need evaluation strategies. Applied lambda calculus with types is supported, as taught on the Theory of Computation Course.

[list of known limitations and bugs] [limitations: mutual left recursion issue (discuss in Future Work), type inference]

6.2 Improving Student Understanding

The user tests conducted and summarised in section 5.2 suggest that students with a computer science background learning lambda calculus for the first time would likely find the tool useful in supporting their learning. Possible improvements such as including tooltips as suggested in user feedback are valid and would be implemented given more time.

The user feedback also suggested that the simplicity of the interface is a positive, and care must be taken to prevent the interface from becoming too complicated. The interface is to be used as support for the Theory of Computation lecture slides or other lambda calculus guidance as opposed to teaching those using the interface lambda calculus, and this should be maintained in any future work done.

With a goal of supporting learning by providing an interface which accurately evaluates lambda terms, it can be said that these goals have been met.

6.3 Future Work

Due to the time limited nature of this project, not all features or corrections have been implemented. Given more time, the following work would be done:

1. Fix the bugs discussed previously
2. Add in multi-input abstraction terms to allow users to enter $\lambda xy.M$ as appears in the lecture slides [8] as opposed to $\lambda x.\lambda y.M$
3. Improve the type inference to allow input types of multi-operation functions to be inferred
4. Include tooltips linked to various aspects of the web interface to give users more information
5. Do further user testing for the web interface to get a wider understanding of any style/usability issues that exist
6. Allow users to enter lambda terms in the De Bruijn notation, for those unfamiliar with the Barendregt convention

Bibliography

[1] Umut Acar and Amal Ahmed. *Lecture 3: Lambda Calculus (Syntax, Substitution, Beta Reduction)*. Jan. 17, 2018. URL: `https://ttic.uchicago.edu/~pl/classes/CMSC336-Winter08/lectures/lec3.pdf` (visited on 08/01/2019).

[2] Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus*. Mar. 2000. URL: `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf` (visited on 07/27/2019).

[3] Marc Bezem and Jan F. Groote. *Typed Lambda Calculi and Applications*. Springer-Verlag, June 13, 2008, p. 356.

[4] Steven Bird, Ewan Klein, and Edward Loper. *Natural Languge Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, 2009, pp. 291–322.

[5] Rosalie Chan. *The 10 most popular programming languages, according to the 'Facebook for programmers'*. Aug. 10, 2019. URL: `https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10?r=US&IR=T` (visited on 01/22/2019).

[6] Alonzo Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 38 (3 1936), pp. 472–482.

[7] Agile Business Consortium. *The DSDM Agile Project Framework*. Jan. 2014.

[8] Ornela Dardha and Simon Gay. *The Theory of Computation (H), course code COMPSCI4072 Lecture Slides*. 2019. URL: `https://www.gla.ac.uk/coursecatalogue/course/?code=COMPSCI4072`.

[9] Stacy Everett. *Grammar*. July 11, 2019. URL: `https://github.com/antlr/grammars-v4/tree/master/arithmetic` (visited on 07/28/2019).

[10] Python Software Foundation. *Built-in Exceptions*. Aug. 8, 2019. URL: `https://docs.python.org/3/library/exceptions.html` (visited on 08/08/2019).

[11] University of Glasgow. *Theory of Computation (H) Course Specification*. 2019. URL: `https://www.gla.ac.uk/coursecatalogue/document?type=sp&courseCode=COMPSCI4072` (visited on 08/02/2019).

[12] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. 2nd ed. King's College Publication, 2004.

[13] Mehedia Hasan. *The 20 Most Popular Programming Languages To Learn For Your Open Source Project*. Aug. 10, 2019. URL: `https://www.ubuntupit.com/top-20-most-popular-programming-languages-to-learn-for-your-open-source-project/` (visited on 2019).

[14] Susan B. Horowitz. *Lambda Calculus (Part 1).* 2013. URL: `http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html#normal` (visited on 08/01/2019).

[15] Fairouz Kamareddine. *Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems.* Oct. 4, 2000. URL: `https://pdfs.semanticscholar.org/2b65/df6c8c0cc14db2afcacbca0f0423588ec6f0.pdf?` (visited on 08/02/2019).

[16] Timothy C. Lethbridge and Robert Laganière. *Object-Oriented Software Engineering.* McGraw Hill Book Company, Dec. 1, 2004, p. 323.

[17] Ralph Loader. *Notes on Simply Typed Lambda Calculus.* Feb. 1998. URL: `http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/ECS-LFCS-98-381.pdf` (visited on 07/28/2019).

[18] Matt Makai. *Flask.* Aug. 8, 2019. URL: `https://www.fullstackpython.com/flask.html` (visited on 2019).

[19] Terrance Parr. *The Definitive Antlr4 Reference.* 1st ed. The Pragmatic Programmers, 2012.

[20] Alison DeNisco Payome. *Forget the most popular programming languages, here's what developers actually use.* Aug. 10, 2019. URL: `https://www.techrepublic.com/article/forget-the-most-popular-programming-languages-heres-what-developers-actually-use/` (visited on 10/22/2018).

[21] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[22] Raúl Rojas. *A Tutorial Introduction to the Lambda Calculus.* 1998. URL: `https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf` (visited on 07/28/2019).

[23] Michael L. Scott. *Programming Language Pragmatics.* 4th ed. Morgan Kaufmann Publishers, 2016.

[24] Peter Selinger. *Lecture Notes on the Lambda Calculus.* 2013. URL: `https://www.irif.fr/~mellies/mpri/mpri-ens/biblio/Selinger-Lambda-Calculus-Notes.pdf` (visited on 07/27/2019).

[25] Ken Slonneger and Barry Kurtz. *Chapter 5: The Lambda Calculus.* Pearson, 1995, pp. 139–166.

[26] Kevin Sookocheff. *Eta Reduction.* Sept. 27, 2018. URL: `https://sookocheff.com/post/fp/eta-conversion/` (visited on 08/02/2019).

[27] Kevin Sookocheff. *Normal, Applicative and Lazy Evaluation.* Sept. 18, 2018. URL: `https://sookocheff.com/post/fp/evaluating-lambda-expressions/` (visited on 08/01/2019).

[28] Saumitra Srivastav. *Antlr4 - Visitor vs Listener Pattern.* Oct. 19, 2017. URL: `https://saumitra.me/blog/antlr4-visitor-vs-listener-pattern/` (visited on 08/06/2019).

[29] Paul Tarau. *Church Encoding.* Feb. 17, 2017. URL: `http://www.cse.unt.edu/~tarau/teaching/PL/docs/Church%20encoding.pdf` (visited on 08/02/2019).

[30] Gabriele Tomassetti. *The ANTLR Mega Tutorial.* Mar. 8, 2007. URL: `https://tomassetti.me/antlr-mega-tutorial/` (visited on 08/06/2019).

[31] Sebastian Wiesner. *Type Inference - Main seminar, summer term 2011*. Aug. 14, 2019. URL: `http://www2.in.tum.de/hp/file?fid=880` (visited on 06/22/2011).