Writing a Program to Evaluate Lambda Expressions

Yola Jones

August 8, 2019

Chapter 1

Design and Implementation

1.1 Design

The overall program is split into a number of distinct elements. A grammar is used to define the syntax of a lambda term. Antlr is used to turn this grammar into an abstract syntax tree, which is used by an expression evaluator to traverse the tree and determine the result of the input lambda term.

A web interface is used to house this program, providing the user with a convenient and simple way to interact with the program.

Since Antlr is used to turn the grammar into an abstract syntax tree, this will not be documented in the design section, as this process has already been well-defined and will not be deviated from [12]. However, the grammar, expression evaluator and the web interface are key blocks which need to be implemented with key factors in mind, and so have been discussed below.

1.1.1 Grammar

The key aim in creating this grammar is 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 [8]. Applied lambda calculus adds functions and constants to this definition [17]. Therefore the grammar of a lambda calculus term becomes:

| Application (of form `[term] [term]`

| Abstraction (of form `[abstraction_term].[term]` where `[abstraction_-term]` is of the form λ`[variable]`)

| Function (of form `[term] [operation] [term]`)

| 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 [8] [5], and will allow each student to input a lambda term directly from the lecture slides with minimal adjustment.

1.1.2 Expression Evaluator

The fundamental idea behind this evaluator is tree traversal, which navigates through the token nodes and performs different operations depending on the type of token 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 the incoming value and substitute it into its own function. This will allow an evaluated expression to be built up, and a result evaluated.

Antlr provides two mechanisms for traversing an abstract syntax tree: listeners and visitors.

A listener is a passive way of evaluating a syntax tree, an antlr Walker object is declared alongside the desired listener class, the walker traverses the tree using a depth-first approach, triggering methods from the listener as it enters and exits each token [12]. These listener methods can't return values, so expressions and evaluations are 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 [20].

The key difference between a listener and a visitor is that a visitor controls its own traversal of the tree. By visiting the children of the current node explicitly, the path they take around the tree can be controlled [12], for example some children not being visited until their parents are evaluated, or a right-hand child being visited and evaluated before the left.

Visitors also allow custom return types, meaning rather than having to rely on separate objects for expression value return, nodes can return their resultant expressions directly to their parent node [20].

With beta reduction, different methods evaluate terms in different ways. 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 being more appropriate for this task than a listener. This is supported by the fact that when evaluating expressions, the evaluation will be happening as the tree is visited, and therefore there will be a great deal of return values needed. 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 will be the main code written in this project. A parser will be passed to a custom visitor interface, with different visitors being defined for each of the beta-reduction methods being implemented, since these methods will each traverse the tree in a different way.

The visitor should return tree things upon returning to the topmost node: the value of the expression, whether or not the term is typable, and what type the expression will be. It will also return details of any errors, for example syntax or “normal form does not exist” errors where applicable.

1.1.3 Web Interface

The web interface will allow the user to input a lambda term, along with the types of any terms. 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.

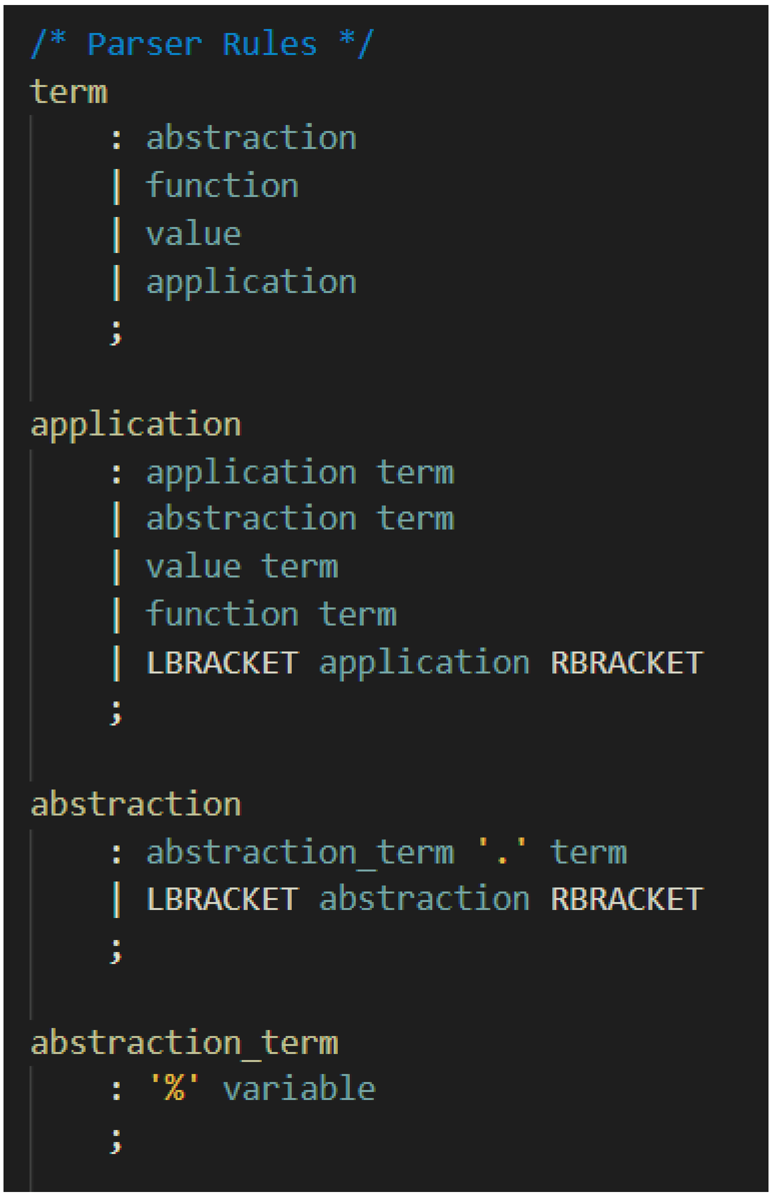
After the user has entered the data, a HTTP POST request will be used to send the data through the back-end code, which will process the term and return the result, type validity and the type of the final term from the visitor. This will then get displayed back to the user.

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

1.2 Implementation

1.2.1 Grammar

The lambda grammar is contained in a .g4 file which defines the parser and lexer rules for lambda calculus, as required by Antlr. A section of the grammar is shown in 1.1, and indicates the parser rules for the term, application and abstraction token nodes, alongside the lexer rules in 1.2.



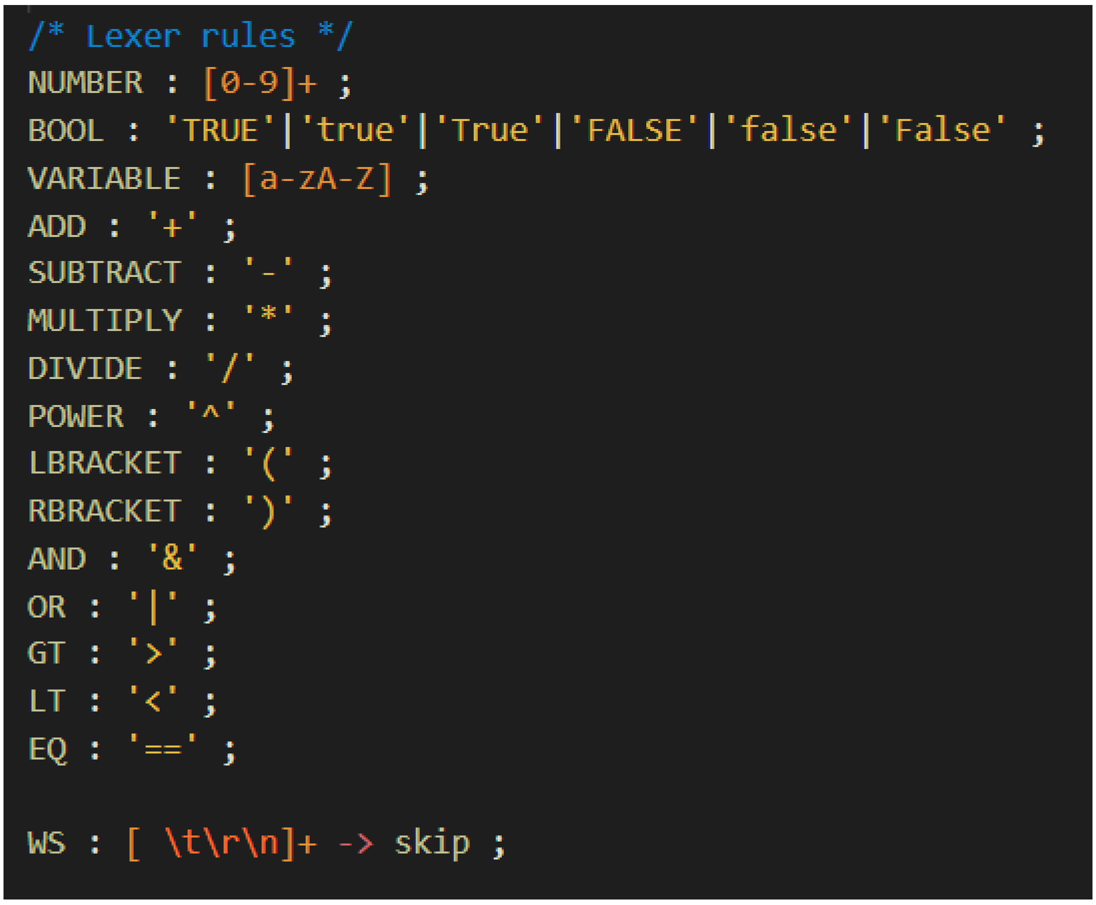
```
/* Parser Rules */
term
        : abstraction
        | function
        | value
        | application
        ;

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

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

abstraction_term
        : '%' variable
        ;
```

Figure 1.1: 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 1.2: Lexer Rules

1.2.2 Abstract Syntax Tree

Having defined a grammar, antlr can then be used to create an abstract syntax tree. This is very simple to do, and involves passing the input term through an Input Stream class which converts this to a string of characters, then feeding this 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 which can be explored [22]. This process is well documented and therefore will not be deviated from.

1.2.3 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. This code can be split into three distinct sections as discussed in the Background chapter of this report: 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 previously. These rules are as follows:

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

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

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

The final rule is the rule to be focused on, and contains 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 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 the situation where a clash in free variables between the two expressions are found. This produces an alpha-converted term, and regular substitution happens using a string.replace() python method, as abiding by the Barendregt Convention.

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, but the key difference is in when terms are evaluated in an application MN, whether N is evaluated before being passed into M, or whether it is substituted first and then evaluated inside M.

Because of these differences in evaluation strategy, two separate visitors are used. However, since they share a lot of the same common functionality (when alpha conversion happens, typing rules, what happens inside functions), 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 to nothing other than typing, since the type of a term happens during the evaluation of that term as will be discussed in more detail below. 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.

Hello! [7] [21] [1] [2] [3] [4] [5] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [22] [20]

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] 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.

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

[5] Ornela Dardha and Simon Gay. *The Theory of Computation*. 2019.

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

[7] 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).

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

[9] 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).

[10] 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).

[11] 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).

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

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

[14] 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).

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

[16] 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).

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

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

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

[20] 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).

[21] 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).

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