Creating a Lambda Calculus Compiler as Research into a Core Language for Type Analysis

Thomas K. Butterwith
AC40001 Honours Project
BSc (Hons) Applied Computing
University of Dundee, 2015
Supervisor: Dr. M. Gaboardi

Abstract – *Lambda calculus is a simple way of expressing functions as formulas. This project seeks to create a compiler able to simplify lambda calculus expressions using the call-by-name method. The development process consisted of learning OCaml and functional programming, creating an arithmetic compiler, before finally creating a lambda calucus compiler.*
The project is written in OCaml, using OCamllex and OCamlyacc to create the lexical analyser and parser respectively.
The compiler uses alpha equivalence and beta simplification to simplify, via substitution, any lambda expression entered by the user, using church encoding to interpret integers entered by the user.

1 Introduction

The aim of this project was to create a compiler capable of implementing arithmetic as well as lambda calculus (from here on referred to as λ calculus). The compiler allows a user to input a text file containing λ calculus expressions and outputs a simplified and calculated solution.
The project was designed to give an understanding of compiler design, parsing, lexing and a coherent understanding of λ calculus. This report covers background research in this area, a clear specification of the problem, descriptions of the design, implementation and testing, followed by an evaluation of the project as a whole and recommendations for any furture work.

2 Background

2.1 Lambda Calculus

λ calculus is a way of expressing functions as formulas. It consists of a single conversion rule, variable substitution, allowing it to be easily learnt and understood but still powerful enough to create complex sequences of functions[1, 2, 3]. Each λ expression is defined in the following way:

<expression> := *<id>* | *<function>* | *<application>*
<function> := λ *<id>* . *<expression>*
<application> := *<expression>* *<expression>*

Figure 1. Summary of lambda expressions

An expression can be enclosed in rounded brackets for clarity but retains its original meaning e.g. expression B is identical to (B). Figure 1 shows the definition of λ calculus, an example of a function is λx.x (the identity function) where the first x is the id and the x following the dot is a bound variable. Bound variables occur anywhere a variable occurs in both the id and the expression of that λ function. Other variables not included in the id but found in the expression are considered to be free. Arguments can be passed to the function in the following manner:

(λx.x)y => [y/x]x => y

Figure 2. Substitution of x with y using standard notation. [y/x] shows every instance of x is replaced by y

The variable y can be replaced with any value, including another λ expression, as it is purely a placeholder.
When writing λ calculus it is important to remember the variable names carry no meaning and can be easily replaced with others to increase understanding, known as alpha equivalence. Consider the following function:

(λx.x(λy.y))y

Figure 3. y is passed as an argument into the function

The free y being passed into the function as an argument would create possible confusion between the bound and free variables in the expression. This problem can easily be resolved by replacing the y in the second λ expression with t so the expression reads as:

(λx.x (λt.t)) y
⇨ [y/x] x (λt.t)
⇨ y (λt.t)

Figure 3.1 Simplification of the function showing the renaming of a λ expression

2.2 Church Numerals

In λ calculus even natural numbers can be represented as functions. By using Church Encoding [3] it is possible to represent numbers greater than or equal to zero and apply them to functions. Numbers can be derived from the number of times a function is applied to its argument.

$0 = \lambda s.\ \lambda z.z$
$1 = \lambda s.\ \lambda z.s(z)$
$2 = \lambda s.\ \lambda z.s(s(z)$

Figure 4. Examples of Church Numerals

Using the successor function (λn. λs. λz.s(n s z)) as an example (the function does the equivalent of the ++ operator in Java and C++) it is clear how church numerals can be utilised.

$(\lambda n.\ \lambda s.\ \lambda z.s(n\ s\ z)\ )(\lambda s.\ \lambda z.\ s(z))$
⇨ $\lambda s.\ \lambda z.s((\lambda s.\ \lambda z.\ s(z))\ s\ z\ )$
⇨ $\lambda s.\ \lambda z.s((\lambda z.\ s(z))\ z\ )$
⇨ $\lambda s.\ \lambda z.s(s(z))$

Figure 5. Simplification of the successor function with 1 passed as an argument

Here the successor function has taken the number 1 (λs. λz.s(z)) and has simply increased the value by one. There are a number of predefined functions dealing with arithmetic, a some of which are included in the compiler as key words to be utilised alongside the church numerals.

2.3 Beta Simplification

Beta simplification allows λ expressions to be applied to one another using left most simplification. When an argument is passed to a λ expression, the every instance of the first id in the expression is replaced by the argument. As shown in fig 5. the second stage shows every instance of n is replaced by the church numeral for one. This replacement can be represented using the notation [λs. λz.s(z)/n]. Steps three and four show the further simplification of the λ expression by applying the argument **s** to the church numeral, followed by the application of **z**.

2.4 Compiler Design

Due to the relative simplicity of λ calculus and church numerals it makes for an ideal learning tool for compiler design. Compilers consist of a lexical analyser (lexer) and a parser, used to turn a stream of characters and strings into useable data structures utilised by the rest of the compiler [8].

The lexer is responsible for turning the incoming string of characters into a stream of tokens, each one identifying the type of character or string. Tokens can be anything from key words to punctuation.

The parser takes the token stream and turns it into usable data structures for the rest of the compiler to handle. This involves pattern matching the token stream with different scenarios.

Identifying the key concepts of a language is key to designing a lexer and identifying tokens. Lexers are used to identify the key words and data types the language will utilise, for λ calculus the λ symbol, the dot, single characters and parenthesis are all required to be identified by the lexer.

3 Specification

3.1 Aim

The aim of this project is to create a λ calculus compiler capable of parsing and simplifying λ calculus expressions.

3.2 Lambda Calculus as a Learning Tool

Due to the simplicity of λ calculus as a programming language it is perfect as a tool to investigate compiler design. The limited number of key words and token combinations makes it a straightforward language to parse. As outlined in figure 1. λ calculus expressions can be defined as one of three forms: a value, a function or an application of two expressions. This is simple to model and easy to parse data into this structure. This moves the main focus of the project from learning λ calculus to developing a lexer and parser, and developing a compiler to simplify λ expressions using beta simplification where possible.

3.3 OCaml

OCaml is a strong choice of language to build the compiler in as it has built in libraries for lexical analysis and parsing built on top of Lex and Yacc (Yet Another Compiler Compiler). The built in libraries allow the creation of .mll and .mly files, written in OCaml, that will be compiled into a fully formed lexer and parser.

Lex is an ideal solution for creating a lexer as it is well documented and well used. The simple, but powerful, syntax of Lex adds to its appeal when completing this project. The syntax of Ocamllex [4] is very similar to that of Lex and perfect for the scope of this report.

Similarly to Lex, Yacc is well documented and widely used when creating parsers. OCamlyacc [4] has similar syntax to Yacc and can be used to convert a stream of tokens and pass them into an OCaml program to manipulate the data.

Both OCamllex and OCamlyacc can be compiled by the default OCaml compiler making them excellent choices to combine with the logic of the λ calculus compiler.

Alongside this OCaml has a number of other features making it a suitable language for compiler construction. The functional programming style allows for easier creation of a compiler, relying heavily on recursion and

pass-by-value functions over the object orientated style of programming. By using functional programming, the compiler is able to work with the flow of data rather than manipulating objects.

3.4 Time Frame

The initial part of this project was spent gaining an understanding of the project and OCaml as a language. Moving away from object orientated programming and into a functional language is a large step so a number of simple programming tasks, including the *99 problems* listed on the OCaml website [5], were undertaken to help gain a solid foundation of understanding before embarking on the main project work. Gaining a working knowledge of OCaml and having the confidence to progress with the project constituted the first few months of the project, reading about the language and completing the programming challenges before mentioned programming tasks.

Once a number of these programming tasks were completed, the next step was then to create a simple arithmetic compiler capable of addition, subtraction, multiplication and division. This small compiler would create a template structure for the project ahead with the implementation of a lexer, a parser, and a program to read a file from the command line to the compiler and output the result.

From there, the λ calculus portion of the compiler could be added along with the necessary logic for beta reduction and handling of church numerals. This portion of the work would form the core of the project, requiring a wider understanding of compilers, parsing, and lexing as well as a solid grasp of λ calculus. The majority of the time set aside for this section was used to properly implement the beta reduction and simplification of the λ expressions.

3.5 Resources

As part of this project, a number of resources have been utilised. The main reading leading to the creation of this project has been *Real World OCaml* [6]. This text outlines the syntax and usage of OCaml as a language as well, in its later stages, the constructs of a very simple lexer and parser, something that has been key in the delivery of this project.

Early in the development process this book was solely used to gain an understanding of OCaml, with the OCaml manual proving to be too detailed in parts. The tutorials and walkthroughs provided a steady but detailed introduction into the language, enough that no other texts were required during the learning portion of this project.

To aid in the understanding of λ calculus, a number of papers and tutorials were utilised. Mainly instructional material from other universities [1,2,3], these tutorials provided a much-needed explanation of λ calculus and beta simplification. A number of these texts are sited throughout this report as some of the information and λ expressions have been taken verbatim.

3.6 Program Requirements

At the beginning of the project design, following on from the learning stage, it was important to create a set of requirements for the finished product providing a clear set of goals to work towards. The requirements are almost all functional, reflecting the research focus of this project. Due to a lack of a ‘client’ in the project, the requirements gathered were designed to drive the project to the goal as specified in the aim. They were used to separate the task into smaller sections that could be worked through using the iterative methodology.

4 Design

4.1 Methodologies

Due to the research intensive nature of this project an iterative design approach was chosen over an agile or waterfall approach. The iterative design approach consists of a constant cycle of planning, implementation, testing, and evaluation; the cycle restarting once the loop has been completed. As this project is separated up into a number of smaller goals that consist of these steps, it is clear this methodology is the most appropriate. The advantages an iterative approach has over an agile approach for this project are as follows:

1) The inability to split some tasks into smaller user stories renders sprints hard to plan,
2) Most tasks require the previous task to be complete before moving on.

Often tasks throughout the project would require prior research and reading, potentially up to two or three weeks before programming starts, which would have lead to sprints involving no output, only research. Using the iterative design methodology allowed time before implementation to research the upcoming task, as well as the completion of testing at the end of the task. Often once a task has been completed or a goal has been met, the evaluation portion of the loop allowed for a look back at what worked well and could be added into the later stages of the project. This process also allowed for any knowledge gained while completing the current cycle to be used to update the plan/deliverables for the next cycle.

4.2 Preprograming Learning

As mentioned in section 3.5, before starting any of the work covered in this report an understanding of OCaml and its workings had to be obtained. The structure of this learning consisted of working through *Real World OCaml* [5] and the tutorials included within the book. A focus was placed on developing a knowledge of manipulating data types using recursive functions, predominantly using OCaml’s pattern matching ability. It was identified early on in the project that this would compose the main body of the work when building the λ calculus compiler.

Once this understanding had been established it was clear the next step would be to work through several λ calculus examples by hand, exploring functions capable of arithmetic calculations and the expansion of church numerals. Before any code for the compiler could be written, it was key to understand beta simplification fully as any misconceptions in the early stages of the project would be carried through the iterative design process, resulting in costly fixes (in terms of time) in the later stages of the project.

During this stage it was also important to identify a development environment that would fit the aims of this project. After experimenting with a number of IDEs (integrated development environments) it was decided upon that a simple text editor with syntax highlighting was the preferred tool, compiling the project using the command line. Using the command line to compile the project provided a deeper insight into the workings of the OCaml compiler and added more flexibility in how the project was constructed. It also fits in with the requirements of the project to be a command line tool, outputting its simplified λ expressions to the command line. Compiling in the command line has the added benefit of being able to compile to any platform OCaml can be installed on.

4.3 Arithmetic Compiler

As a midpoint to the λ calculus compiler, the arithmetic compiler was designed to provide an introduction to compiler creation. The arithmetic compiler would still consist of a lexer and parser but in a much simpler form. At this point in the project the lexer was the more complex part of the program, having to identify all of the tokens required to do simple arithmetic. The parser was able to use OCaml's built in arithmetic functions to manipulate the tokens and provide output. The full code for the arithmetic compiler can be found in the appendix.

4.4 Lambda Calculus Compiler

Being the main body of the project, the λ calculus compiler took the most time to design. Any errors in the design at this stage could percolate through into the implementation later on, causing serious problems. Using the arithmetic compiler as a starting point it was easy to identify the additional tokens that would be required to parse λ expressions (λ and .) and add them to the lexer.

Using the parser from the arithmetic compiler was useful but not entirely necessary as many of the token patterns being matched were no longer valueable. Going back to the definition of a λ expression in figure 1. it is clear the parser had to identify each of these instances in the incoming token stream and pass them into the appropriate data structure. For this task a custom data type had to be created to match the structure of the λ expression. Once this was implemented, the parser could then simply parse the tokens straight into the custom type and then simplify the expression once an end of line token was received. The only exception to this was if the token received was an integer. At this point in the process any integers read by the parser were converted into church numerals to allow for easier beta simplification.

4.5 λ Type

In order to easily process the incoming token stream, a custom λ expression data type had to be designed. To make this as simple but functional as possible it was important to model this data type on the λ expression format detailed in figure 1. using a character to represent the identifier of each expression.

Originally this data type included a string type instead of a character to represent the id of a function, to allow the user to create more complex λ expressions using shorthand expressions but this proved infeasible as the project went on.

4.6 Beta Simplification

As part of the parsing process, once a λ expression has been read and an end of line token has been received, the compiler should begin the beta simplification process. Following the steps outlined in section 2.3, the original design would take the expression, separate it into a list of individual λ expressions and then attempt to evaluate these functions in relation to each other. This initial idea would have worked for the first set of simplifications but for more than one the compiler would lose track of the order of expressions.

After some research on the topic and reading into the De Bruijn index it was clear the original method would result in having to reparse the string each time, a very time expensive and complex process. The De Bruijn index is a notation method to aid in the substitution and simplification of λ expressions, removing the names of the variables and replacing them with the position at which the corresponding statement occurs. This idea can be used to implement a system of substitutions using two lists, one for the variable and one for the values that will replace the variable. This call-by-name method is much more suited to the recursive nature of OCaml as well as providing a cleaner solution to the problem.

The beta simplification process was designed to be a recursive system, working through applications of λ expressions. The process should start with the second expression, the one being applied to the first, and attempt to simplify it before attempting to substitute it into the first expression. This design meant the system would start with the very last variable or expression in the λ expression and work its way back to the left most expression simplifying it as it went before outputting the end result to the console.

4.7 Output

It was decided early on that this compiler for simplifying λ expressions would be a research tool only, to be used by those who have knowledge of programming/the command line already. Unlike compilers for other languages that output to a file once completed, the λ calculus compiler

simply outputs the result to the command line itself, allowing the user to read the result from there. Should the need to compile and output to a file arise, in the event an expression or series of expressions are so complex that the output is a long statement, it would be straightforward to add a command line argument to pipe the result into a file. Details of how to do this on a UNIX based system are included in the user document.

4.8 User Interface

As this tool was designed primarily to investigate compiler design and the implementation of a λ calculus compiler, the user interface was one of the last parts of the project to be considered. The main purpose of the project was to gain an understanding of the theory behind compiler design and the λ calculus compiler was merely a tool to facilitate that. With that in mind, the original design for the user interface consisted of a command line program that the user could open and type their λ calculus expression into, pressing enter to see the simplified version. This simplistic idea was modelled from interpreted languages such as php or JavaScript, where the input is compiled at run time. Although this design was simple to use, it made the tool tedious to use when inputting longer statements or reusing λ expressions in multiple functions. It also left the user with viewing their simplified expression in the command line as the only output method.

In order to make the tool more usable it had to be modelled around a compiled language, taking in an imputing file and either outputting to the command line or to a specified output file. As the project was built in a UNIX environment it was simple to pass in a text file as an argument to the command and allow the user to pipe the output to a text file if necessary. This negated all of the negatives from the interpreted version, allowing the user to edit an input file in plain text in any manner they wished, utilising copy, paste and other functions not easily available in command line programs, and allowing the user to control how the output of the program is handled.

5 Implementation and Testing

5.1 From Design to Implementation

As specified in section 4.1 the project followed the iterative design approach utilising the constant cycle of planning, implementation, testing, and evaluation. This section of the report will cover the implementation of both the arithmetic compiler and the λ calculus compiler before progressing on to testing. Before beginning any programming, at this stage of the iterative cycle it was important to review the design to be sure any assumptions or mistakes were removed and begin research into the coming iteration.

5.2 Implementation

5.2.1 Reading in a File

One of the more complex parts of the initial implementation was creating a main function that when called from the command line, would take in a file as an argument and pass it into the lexer for the first part of the compilation. Combining learning from *Real World OCaml*, along with samples from the OCaml official documentation, a method was created to take the first argument from the command line, create an input channel, then pass the input from that channel into the lexer for analysis.

This section of the project provided a starting function for passing in files to the compiler, used in both the arithmetic and λ calculus compiler.

5.2.2 Arithmetic Compiler

As this was the first step towards creating the λ calculus compiler, the implementation of the arithmetic compiler required a large amount of trial and error, both to perfect the syntax for identifying tokens as well as understanding OCamlex and OCamlyacc as languages. Having looked at a small set of simple mathematical functions such as addition, subtraction, multiplication, and division it was an easy task to identify the necessary tokens for basic functionality. From there the less obvious tokens such as end of file (EOF) had to be added for smoother operation of the compiler. To ascertain which tokens were needed and which tokens were missing from the lexer, it was important to look at the language of the file the compiler would have to parse. This was done by writing a test file using the syntax and many of the functions the language should support. By running this test file through the compiler and analysing the errors presented, it became clear which tokens were missing and any syntax that would cause the compiler issues.

This method of design and implementation created the testing methodology for the rest of the project. By creating a test file with each of the potential conditions the compiler may come across and running it through the compiler, any errors were easily identified.

The process of creating the arithmetic compiler led to the discovery of a number of tokens/syntactic objects that programmers are usually oblivious of and take for granted. As previously mentioned, the end of file marker is something that is rarely thought about but has to be identified by the compiler in order to prevent errors. The compiler cannot parse anything past the end of file marker as nothing exists, any attempt to do so will result in an error. Alongside this, simple but usually overlooked tokens include blank or white spaces and new lines, added to the code to make it more readable by humans but in the case of the language used by this compiler unnecessary. A token was added for each of these and the compiler was instructed to ignore all white space, with new lines being used to finish a statement and signify the compiler should evaluate everything in the line up to that point.

Once the input stream had been read by the lexer, the resulting token stream was passed into the parser. The process of implementing the parser was as much trial and error as the creation of the lexer, understanding the syntax of OCamlyacc and the positioning of certain sections within the file. The parser consisted of two sections, token declaration at the head of the file, and pattern matching for the token stream in the body.
The token declaration section is used to set a data type for the incoming tokens. In the case of the arithmetic compiler the token type was an integer as any token that was not of the type integer was simply a command for manipulating the other tokens. Due to the simplicity of this compiler using the standard integer class in OCaml was sufficient instead of creating a custom data type. Also in the head was a line to tell the main program which set of pattern matching clauses held in the body was the entry point of the program. This was done using the syntax,

```
%type<int option> main
%start main
```

Figure 6. Type declaration in OCamlyacc

where **`type`** declares the token type and **`start`** is the entry point of the parser pointing towards the main function.
The body of the parser, separated using **`%%`**, contained two sets of pattern matching. The first set, and entry point of the program contains the decision making for tokens dealing with the end of the file or the end of the line. If the token stream consisted of one of these two markers and no further expression, the parser would return an OCaml option for None, telling the main program to move to the next token stream to avoid errors. If an expression was found, it was then passed to the second set of patterns to match, to decide on how to proceed. The second set of pattern matching was used to simply identify the mathematical function being applied to the integers and to extract the expression from within parenthesis. To complete the evaluation, the parser uses OCaml's built in mathematical functions (`+, -, *, /`) to act upon the tokens, with the result returned back to the main function to be printed to the command line. Initially the two sets of pattern matching were combined but the separation was added to aid readability and to make clear the separation of functions.

5.2.3 λ Calculus Compiler

Using the arithmetic compiler as a starting point the lexer was adjusted to recognise the new tokens required to identify and lex λ expressions. By adding in tokens for the λ symbol, in this case simplified to \ as a representation, and the dot . used to signify a λ function, the lexer was almost at the stage where it was ready to be used. As the lexer reads in strings it was important to add in a regular expression to identify individual characters, those used as variables in the λ expressions, rather than take a whole string.
The parser however could not simply use built in functions as it could when dealing with arithmetic. Using the λ type specified in section 4.5 the parser was adjusted to parse the tokens into sections of that type where the identifiers are characters, an id and expression a λ function, and any two expressions following each other an application. Adjusting the head of the file to export a λ expression rather than an integer was a simple conversion process, changing the type declaration to,

```
%type<Lambda.expr option> main
```

Figure 7. Adjusted type declaration from integer to λ expression

Each line of the input file is analysed and parsed until all that remains is a single λ type consisting of the various λ expressions and applications thereof ready to be simplified. If a user enters an integer, a utility function was created to convert it to church numerals so it could be simplified in the beta simplification process.
Once the expressions have been parsed, the λ type is passed into a function to undergo beta simplification using the process outlined in section 4.6. The recursive nature of this function allows the program to progress into the λ type through each of the applications of expressions until it reaches the end, and is then able to start simplifying and substituting functions.

5.2.4 Beta Simplification

Using OCaml's pattern matching, the system to simplify λ expressions is based on the λ calculus compiler created by Neil Jones [9] using a tuple of two lists to store the variables and values for substitution.
The beta simplification function takes in an expression, in this case anything of the λ type, and the stack, the pair of lists. If the expression is simply a character it can be assumed this is either the id in a λ function or a bound variable in need of substitution. This variable is passed to a lookup function to ascertain if it has a replacement and if so, swap the variable out for its replacement and attempt to simplify again. If the expression is a λ function, the function is placed into a special 'closed' type in the λ type, with the expression of the λ function going through the simplification process again to identify any pending substitutions.
The complex part of this function is when dealing with applications of expressions. Due to the nature of an application being any two expressions applied to one another, this part of the function required a second, nested pattern-matching statement to separate the different cases. Some of the patterns could be generalised, in the case of the application of any expression on a character, but others, like any applications involving λ functions had to be very specific. The breakdown of this function can be seen in the appendix [appendix section] where each pattern is explained via notation.
By adding the 'closed' type to the λ type the functions avoid becoming stuck in an infinite loop of simplification,

where a λ expression is already in its simplest form. This 'closed' type can be utilised to tell the function the expression has finished simplification and can simply be returned. The main function of the program was also adjusted from the arithmetic compiler to pass the parsed λ type into the simplification process rather than just print it to the command line.
As the λ type is a custom type, a print function had to be written to convert the λ type into a string, again utilising OCaml's pattern matching.

5.3 Testing and Debugging

Using the command line to compile OCaml makes the process more customisable but it does come with its own challenges. Without any built in debugging tools, such as the debug mode when debugging java or a stack trace, it can be quite difficult at times to debug a program. When compiling the program using `ocamlbuild`, clear positions of the problem were given but often a description of the compilation issue was lacking thanks to OCaml's very generic error messages. One of the few scenarios that were easily identified was a misunderstanding with OCaml's type inference. These problems were caused by a lack of clarity when declaring variables, leading OCaml to assign it the incorrect type. To avoid this problem, the definitions and use of variables had to be made clear and explicit throughout the program.
As there was no built in debugging tool to utilise during testing, `print_endline()` statements were placed throughout the functions, usually at the start so the current expression could be seen at each stage of the simplification process. By adding clear labels to each type of expression (character, λ function or application) it was easier to follow the expression through the pattern matching cases and discover where problems lay. Debugging this way required a very solid understanding of the program's functionality as well as the expected outcome, constantly reinforcing an understanding in λ calculus and beta simplification.

As mentioned in section 5.2.2, testing of the λ calculus compiler consisted of creating a test file containing all of the language conditions the compiler would have to deal with. This file was then run through the compiler and as errors appeared they could be corrected. The test file for the arithmetic compiler was of a simple design, having to test each of the mathematical operations, the use of brackets, and division by zero. It could be assumed that in any case where a mathematic operation gave the correct output, it would work for all other scenarios.
This method is similar to the unit testing methodology, with each test case consisting of an expression in the test file. In order to create this test file, a number of tutorials on λ calculus were employed to be sure all areas of beta simplification were covered. To be sure the output from each of these cases were correct, a pre created λ calculator [10] was used, comparing the output as well as the simplification steps if necessary. Cases covered in the testing document include conversion of integers to church numerals, applications of each type of λ expression upon every other type, as well as unbound variables that have not been assigned a value.
Although this method of testing is not completely exhaustive, as there could be potential bugs in very specific cases of simplification, it does ensure a high level of accuracy over all cases.

Another form of testing utilised within this project was unit testing, using a series of mock λ expressions to test in the correct output from various functions. To test the parsing and lexing, the original test file was used, having the compiler simply read from the text file and output to the command line. Each data type was annotated with its name, ensuring the data was being read in and parsed into the correct data type.
To test the simplification methods, a unit testing class was created containing a function to test each method used during the beta simplification process. These unit tests were useful for tracking down the exact locations of bugs, as well as consistency testing. This meant if any of the code was changed during development, the unit tests would fail and the problem could be dealt with, rather than letting it manifest later in the implementation.

5.4 Difficulties

Throughout the implementation stage of the project, a number of difficulties arose when creating each of the compilers. For the arithmetic compiler, the main issue was in dealing with end of line and end of file markers.
Originally the design specified the parser would return an integer every time, causing the program to fail should an end of line/file marker be present without any mathematic expressions. The initial solution to the issue was to have the parser skip to the next expression in the file, but obviously this solution was only appropriate to the end of line marker and not for the end of file. After reading into the topic, a solution was found in Real World OCaml [5] using OCaml's option type, useful for returning `Some int` if there was an expression found or `None` if only a marker was present. The main function of the program could then be instructed to print the result to the console if a result was present, or if `None` was returned simply ignore it and move to the next token in the stream.
During the implementation of the λ calculus compiler a number of small but time consuming errors arose. In the beginning of the design phase λ functions could be expressed using a string for the id, rather than a character, allowing the user to build complex λ functions quicker with less effort.

λsz.s(s(z))
⇨ λs. λz.s(s(z))

Figure 8. Adjustment from id represented as a string, to id as characters

The issue with this notation is during the simplification process, the lookup function would attempt to find

variables declared as ‘sz’ rather than variables declared ‘s’ then ‘z’. Although this is a minor issue with the program it can easily be resolved by requiring the user to enter characters as a λ id rather than a string.

When implementing the beta simplification process, there was an issue of becoming stuck in an infinite loop once the expression had been simplified as far as possible. The original design mimicked the definition of a λ expression from figure 1. using only an id, a function, and an application to define the λ type. This resulted in an issue of having no way to declare an expression as simplified. After consulting the simplistic λ calculus compiler by Neil Jones [9] and analysing the ‘closure’ type, it was clear an extra type definition was required to mark the expression as closed. The resulting changes to the simplification function were minimal, adding in an extra pattern-matching clause for any expression of the closed type to be returned, avoid the infinite loop.

Further along in the beta simplification process it became clear that if a λ expression contained three or more applications of λ functions, the function would mark these as closed too soon and not simplify the expression fully. This issue was difficult to find as the expressions were becoming longer and more complex. The root cause of the issue was only discovered after printing out each stage of the simplification process as well as the data types of each part of the expression. In order to solve this problem, the function had to be adapted for the case of an expression being applied to a closed expression. The first expression had to have its closed tag removed, and then be passed back into the simplification process.
This solution again caused the problem of infinite loops, with closed tags being reopened and the beta simplification function being called repetitively. Using the knowledge gained in debugging previous problems, it was important to work the problem by hand, making a note of each of the scenarios the loop was occurring and adjusting the pattern-matching clause to be more specific. A function to reorganise applications of expressions within λ functions was required at this point, due to the number of substitutions moving applications into the wrong order, further stabilising the compiler.

Due to the huge number of possible combinations of λ expressions possible it was important to test a wide variety of cases, in case any syntax caused the compiler difficulties. One of these cases not identified till the late stages of the project was the λ expression known as Ω. In the case of:

(λx.x x) (λx.x x)

Figure 9. Ω

the simplified result is identical to the original expression no matter how many times the substitution is applied. To avoid this problem a loop counter has been applied to the simplification function, causing the function to halt once an infinite loop has been identified.

6 Evaluation

6.1 Usability

From the beginning, the λ calculus compiler was designed to be a tool accessed from the command line. The original strategy was to create a compiler that the user would type their λ expressions into directly, in the same way the OCaml shell works. The user would enter their λ expression, hit enter, and the simplified version would be displayed below. One advantage to this method was the user would only have to open the tool to compile their λ expressions, rather than saving them to a file and inputting that file into the compiler. Unfortunately this method of input was only functional for small λ expressions or expressions that did not have repeating parts, as copy and paste is restrictive in the command line. To overcome this issue the compiler was redesigned to take a file as an input and get rid of having to open the tool separately.

This redesign, allowing the user to pass in a text file containing all of their λ expressions, created a much more useable tool allowing the user to write their λ expressions in any text editor they preferred and have the compiler simplify them in bulk. Although this tool may slow down the simplification of single, simple expressions it makes the tool much more useful for users looking to compile multiple expressions at once, or expressions that require repeating elements. Another advantage to this system over the old one is if the user discovers a problem with their λ expression, e.g. a miss placed identified, the user no longer has to retype their whole expression. Instead they can simply change the input file and recompile.

6.2 Test Cases

As mentioned previously, the compiler was built around a number of test cases. Each test case was designed to assess how the complier would deal with certain scenarios within λ expressions. For each test the compiler prints to the console the input, the expected output, how it was read by the compiler, the state of the expression after it has been put through the alpha equivalence function, and the expression in its most simplified form ready to be compared with the expected output.
The tests include individual characters, and λ functions as well as various combinations of applications to be sure all varieties of expression are covered. The built in expressions for addition and finding the successor of a number are also tested at this point.

Due to these tests being present during the creation of the compiler, the finished project passes all of these tests.

7 Description of the final product

The final project is a compiler accessed through the command line to simplify λ calculus expressions, taking in a text file containing any number of λ calculus expressions, separated by a semicolon, the compiler utilises alpha-equivalence and beta simplification to output the λ expression in the simplest form possible. The program is accessed by calling the command `./lambda_compiler` with the text file passed in as the first argument after the call.
The program will then simplify the expression as far as possible and print the original statement followed by the simplified version to the command line. Any errors in syntax or unrecognised tokens within the λ expression file will be displayed in the command line.
As this is a command line tool it is possible to pipe the output of the program into a text file by appending `> output_file_name.txt` to the end of the command should the user wish to save the output for later use.

8 Summary and Conclusions

8.1 Appraisal

Throughout the development of this project a number of criticisms became clear. The first being a much more structured form of time management would have benefited the project, especially in the earlier stages of the project. The λ calculus compiler had the potential to be much more expansive, especially the research into type checking, had the project progressed faster early on. Although this project had a good time plan set out, with clear goals at the end of each section, the time plan could have been separated up further into smaller, more manageable sections. This would have led to research into λ calculus being started earlier, allowing the compiler to progress faster. A similar error in the project was, the amount of time required to gain an understanding of λ calculus and beta simplification was underestimated. Research into λ calculus should have been undertaken at the same time as learning OCaml, so that when the project was ready to move on to creating the first iteration of the compiler, an understanding of λ calculus was already present.

Another issue with the project was a lack of clear requirements early on in the project. Had an official set of requirements been drawn up in November, rather than February, the project would have been clearer in its goals and outcomes. This lack of clear goals resulted in slow progress at times and potential underestimation of time required to complete later sections of the project. As a result, the λ compiler was unable to handle simply typed λ calculus, only untyped.
Development of the project could have been assisted by utilising a more multi-tasking approach to research rather than simply tackling one topic at a time. As previously mentioned, while learning OCaml, an understanding of λ calculus could have been developed, and likewise while implementing the λ calculus section of the compiler, the research into type checking could have been started.

Choosing to use the iterative methodology proved to be a good decision as other methodologies would have compounded the issue of time management further. Using the waterfall approach would have meant all of the learning would have been placed at the start of the project and development could have started very late, resulting in a much less functional compiler. The agile methodology would have been more fitting than waterfall but still would have fallen down when tasks took longer than one sprint. The iterative methodology allowed the project to finish one portion of the project, including testing, before progressing on the next section but with more flexibility over time than agile.

One point of the project that worked well was gaining a clear understanding of OCaml before starting any of the work on the compiler. Starting the project without a good knowledge of the language would have been detrimental to the project, resulting in code that would have had to be changed as the compiler become more complex. As mentioned in section 5.4, the original data type for λ expressions used a string as the identifier instead of a character. This was implemented before understanding the beta simplification and resulted in the loss of a number of workdays while trying to adapt the simplification process to work with strings over characters.

8.2 Future Work

Given more time the λ calculus compiler could be developed to be used as a teaching tool for λ calculus, explaining to students how to simplify and evaluate expressions. Each stage of the beta simplification process could be printed to the screen, showing the user a step-by-step evaluation of a λ expression. This could also be utilised as a ‘fill in the blank’ system, requiring the user to enter missing information in order to complete an expression. In its current state, the λ calculus compiler is very useful in checking the results of hand simplified expressions as well as expressions involving church numerals that may be too long to comfortably write out by hand.

As it stands, the λ calculus compiler created as part of this project is a good starting point for an investigation into type checking and typing systems.
In addition to this, the learning undertaken in this project could be applied in the creation of a more complex programming language with a broader syntax than λ calculus. The understanding of parsing and lexing, creation of custom data types, as well as a knowledge of type checking systems could easily be put to use in creating a functional programming language, perhaps a simplified version of Haskell.

8.3 Knowledge Gained

Throughout this project a number of valuable skills have been learnt, most notably the processes a compiler utilises to convert a plain text file to something readable by a machine. The parsing and lexing stages of the λ calculus compiler have developed skills to deal with an incoming data stream, converting it into something that is manageable and easily manipulated.

Working in OCaml has also brought an understanding in functional programming and has developed a way of thought based around recursion, rather than loops usually employed in object-orientated programming. The use of functional programming in this project has provided a completely different way of looking and problems and the way they are broken down, something that will aid development of any future projects undertaken, including those built with object orientated techniques.

Acknowledgments

The author would like to thank Marco Gaboardi for his support and guidance as supervisor to this project. His input was vital to the completion of the project. Thanks are also due to the School of Computing staff, both administration and lecturers, for providing the knowledge and support throughout the project.

References

[1] Barendregt, H. and Barendsen, E. (1984) Introduction to Lambda Calculus. Nieuw archief voor wisenkunde 4.2 p.337-372.

[2] Rojas, R. A Tutorial Introduction to the Lambda Calculus. [Online] University of Texas Dallas. Available from : http://www. utdallas. edu/~ gupta/courses/apl/lambda. pdf. [Accessed: December 2014]

[3] Selinger, P. (2008) Lecture Notes on the Lambda Calculus. [Online] Department of Mathematics and Statistics Dalhousie University, Halifax, Canada. Available from : http://cs.simons-rock.edu/cmpt320/selinger.pdf [Accessed: December 2014]

[4] Smith, J. B. (2007) Practical Ocaml. Apress.

[5] Ocaml (Unknown) 99 Problems (solved) in Ocaml. [Online] Ocaml.org. Available from: https://ocaml.org/learn/tutorials/99problems.html [Accessed: October 2014]

[6] Minsky, Y., Madhavapeddy, A., and Hickey, J. (2013). Real World OCaml: functional programming for the masses. O'Reilly Media, Inc..

[7] SooHyoung, O. (2004) Ocamlyacc Tutorial. [Online] Programming Languages Laboratory, Korea Advanced Institute of Science and Technology. Available from: http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/ [Accessed: 20th January 2015]

[8] Mogensen, T. Æ. (2010) Basics of Compiler Design. Anniversary Ed. Department of Computer Science, University of Copenhagen

[9] Jones, N. (1993) A Lambda Interpreter in ML. [Online] Department of Compuer Science, Dartmouth. Available from: http://www.cs.dartmouth.edu/~mckeeman/cs118/lectures/14.html#anchor2 [Accessed: 26th March 2015]

[10] Burch, C. (2012) Lambda Calulator [Online] Hendrix College. Available from: http://www.cburch.com/lambda/ [Accessed: 28th March 2015]

Appendices

1. Meeting Minutes
2. User Manual
3. User Guide
4. Source Code