1. Meeting Minutes

Date: **10/10/14**
Previous Meeting: **N/A**
Discussion:

Introduction to OCaml. Should look over the OCaml manual and begin to become aquianted with the language. Read around functional programming. Discussed the very basics of the project and what is to come.

Date: **17/10/14**
Previous Meeting: **10/10/14**
Discussion:
More discussion about OCaml as a language. Brief discussion about Marco's research paper *Fuzz*. Discussed potential direction of the project, maybe adapting *Fuzz* to do something different – could be quite complex. Should carry on working on OCaml, developing a knowledge of the language, and read the *Fuzz* research paper as best I can.

Date: **24/10/14**
Previous Meeting: **17/10/14**
Discussion:
Short meeting about my progress with OCaml. I should carry on working through the examples in Real World Ocaml that I have found. Rather than completing the work in the OCaml shell I should try to compile files into byte code or native applications to be run from the command line.

Date: **04/11/14**
Previous Meeting: **24/10/14**
Discussion:
Discussion mostly involving parsing and lexical analysis. I should read Chapter 16 of Real World OCaml to get an idea of how parsing works in OCaml now that I have an understanding of the language.
The goal for the next meeting is to create an arithmetic compiler that will be able to handle the set n | plus | minus | multiply | divide

Date: **18/11/14**
Previous Meeting: **04/11/14**
Discussion:
Demonstration of my arithmetic compiler. Discussed issues with compilation of modules as well as reading in files into the compiler. As it stands the compiler allows the user to enter arithmetic expressions into the OCaml shell but does not read in a file, something that will be importand down the line.
Next steps involve dealing with the compilation errors and also the errors due to missing token combinations as the compiler throws errors when too many empty lines follow each other.

Date: **20/11/14**
Previous Meeting: **18/11/14**
Discussion:
Looked at my arithmetic compiler again. I had been able to fix the errors surrounding the tokens, just required an extra rule in the parser to ignore extra empty lines. This will be the last meeting with Marco before he goes to the USA and I start exams etc.
The next step in the process is to start looking at lambda calculus and expanding the arithmetic compiler to parse and lex lambda expressions

Date: **26/01/15**
Previous Meeting: **20/11/14**
Discussion:
Discussed project goal, editing *Fuzz* seems overly complex and would be much better to create my own compiler to handle arithmetics and lambda calculus with simple type checking. Time permitting it would be good to compare my work with Marco's compiler mentioned in the previous weeks (http://staff.computing.dundee.ac.uk/marcogaboardi/publication/Gaboardietal13submitted.pdf)

Date: **06/03/15**
Previous Meeting: **26/01/15**
Discussion:
Spoke about the project so far, progressing with the Lambda calculus compiler. Now that it reads from a text file it should begin to simpify lambda expressions. Look at beta simplification.

Date: **26/03/15**
Previous Meeting: **06/03/15**
Discussion:
Discussed problems with beta simplification. I had been approaching the problem incorrectly, thinking of the lambda expressions as a list rather than applications of one another. Given information about the De Bruijn index which should help with the simplification process along with two examples of lambda calculus compilers written in ml. I need to make sure my lexer and parser are tranfering the data into the appropriate data types.

http://www.cs.dartmouth.edu/~mckeeman/cs118/lectures/14.html#anchor2
http://iml.univ-mrs.fr/~regnier/taylor/lambda.ml.html

Date: **17/04/15**
Previous Meeting: **26/03/15**
Discussion:
Discussed the report and what should be submitted. Not really time to implement type checking anymore, just need to make sure the lambda compiler is properly tested and the report is well thought out. Sent a draft of the report to Marco for him to read through and check to make sure it is well structured and covers the correct material.

2. User Manual

Installation
Requirements
In order to compile the lambda calculus compiler you must have OCaml version 4.01.0 or higher installed. The recommended install method can be found in the documentation on the OCaml website (https://ocaml.org/docs/install.html)

Compiling
To compile navigate to the source code directory and run the command:

```
ocamlbuild -tag thread -use-ocamlfind -pkg core lambda_compiler.native
```

Running
Running the compiler is as simple as using the command where the file name is your lambda calculus:

```
./lambda_compiler.native *file_name*.txt
```

Syntax
The compiler uses regular lambda calculus notation replacing λ with \ . Lambda expressions are expressed as follows :

```
(\x.x x)(\y.y);
```

which will be simplified down to:

```
(\y.y)
```

Lambda functions should be enclosed within parenthesis with the end of an expression being marked with a semicolon.

Key Words
The compiler contains a number of prewritten functions accessed using key words

Key Word	Function
successor	λnfx.f(nfx)
addition	λmnfx.mf(nfx)

Examples

```
        (\x.x)y;
=>      y

        (\n.\f.\x.f(nfx))(\s.\z.s(s(z)));
=>      \s.\z.s(s(s(z)))

        addition 2 1;
=>      \s.\z.s(s(s(z)))
```

Output
On completion the compiler will automatically print your output to the command line window you have open. Should you wish to the compiler to output to a file use the command:

```
./lambda_compiler.native *file_name*.txt > *output_file_name*.txt
```

3. User Guide

Aim
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.

Background
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.
Expressions are defined as follows:

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

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.

Church Numerals
In λ calculus even natural numbers can be represented as functions. By using Church Encoding 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 = λs. λz.z
1 = λs. λz.s(z)
2 = λs. λz.s(s(z)
```

Lambda Calculus Compiler
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.

Usage

See Usage Guide

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

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

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]

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

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

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

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]

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]

Smith, J. B. (2007) Practical Ocaml. Apress.

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]

4. Source Code

lambda_type.ml

```
type expr =
        | Int of int
        | Char of char
        | Lambda of char * expr
        | App of expr * expr
        | Close of expr
        | Error of string
```

lib.ml

```
open Lambda_type

let succ       = Lambda ( 'n', Lambda ('s', Lambda('z', App( Char 's', App(Char 'n',
App (Char 's', Char 'z'))))))
let add        = Lambda ( 'm', Lambda ('n', Lambda ('f', Lambda ('x', App(Char 'm',
App(Char 'f', App(Char 'n', App(Char 'f', Char 'x'))))))))
let mult       = Lambda ( 'p', Lambda ('q', App( Char 'q', App(add, App( Char 'p',
Lambda ('s', Lambda ('z', Char 'z')))))))

let alpha_list =
['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p';'q';'r';'s';'t';'u';'
v';'w';'x';'y';'z';]
```

simplification.ml

```
open Core.Std
open Lib
open Lambda_type


let rec lambda_to_string expr =
        match expr with
        | Int i                         -> string_of_int i
        | Char e                        -> Char.to_string e
        | Lambda (id, e1)      -> "\\" ^ Char.to_string id ^ "." ^ (lambda_to_string
e1)
        | App (e1, e2)         -> lambda_to_string e1 ^ "(" ^ lambda_to_string e2 ^ ")"
        | Close e                       -> lambda_to_string e
        | Error e                       -> e

let rec lambda_to_string_annotate expr =
        match expr with
        | Char e                        -> Char.to_string e
        | Lambda (id, e1)      -> "\\" ^ Char.to_string id ^ "." ^ (lambda_to_string
e1)
        | App (e1, e2)         -> "a {" ^ lambda_to_string e1 ^ "}{(" ^
lambda_to_string e2 ^ ")}"
        | Close e                       -> "c [" ^ lambda_to_string e ^ "]"
        | Error e                       -> e

let rec expand_church expr =
        match expr with
        | 0 -> Char 'z'
        | _ -> App (Char 's', (expand_church (expr - 1)) )

let int_to_church expr =
        match expr with
        | 0     ->      Lambda('s', Lambda ('z', Char 'z'))
```

```
        | 1            ->      Lambda('s', Lambda ('z', App (Char 's', Char 'z')))
        | _            ->      Lambda('s', Lambda ('z', (expand_church expr)))

let rec remove_closed expr =
        match expr with
        | Char c                        -> Char c
        | Lambda(id, e)                 -> Lambda (id, remove_closed e)
        | App(e1, e2)           -> let exp1 = remove_closed e1 in
                                                        let exp2 = remove_closed e2 in
                                                         App (exp1, exp2)
        | Close e                       -> remove_closed e

let rec combine_apps expr =
        match expr with
        | App (e1, e2)                  ->
                (match e1, e2 with
                        | App (ex1, ex2), App (ex3, ex4)      -> App(combine_apps ex1,
combine_apps (App (ex2, App(ex3, ex4))))
                        | App (ex1, ex2), _                             ->
App(combine_apps ex1, combine_apps (App (ex2, e2)))
                        | _ , App (ex1, ex2)                            -> App(e1,
combine_apps e2)
                        | _, _                                                                  ->
App (e1, e2))
        | Lambda (id, e)        -> Lambda (id, combine_apps e)
        | Char c                        -> expr

let rec lookup_id id used =
        match used with
        | []            -> false
        | x::xt         -> if x = id then true else (lookup_id id xt)

let rec replace_id expr id replacement =
        match expr with
        | Char c                        -> if c = id then Char replacement else Char c
        | Lambda (i, e)                 -> if i = id then Lambda (replacement,
(replace_id e id replacement)) else Lambda (i, (replace_id e id replacement))
        | App (e1, e2)          -> App( replace_id e1 id replacement, replace_id e2 id
replacement)

let rec find_replacement_id used alpha =
        match alpha with
        | []    -> 'a'
        | x::xt -> if (lookup_id x used) = false then x else find_replacement_id used
xt

let rec get_used_ids expr used =
        match expr with
        | Char c                -> c::used
        | Lambda(id, e) -> id::(get_used_ids e used)
        | App (e1, e2)          -> (get_used_ids e1 used)@(get_used_ids e2 used)

let rec alpha_equiv expr taken =
        match expr with
        | Int i                         -> Int i
        | Char c                        -> Char c
        | Lambda (id, e)        -> if (lookup_id id taken) = false then Lambda (id,
(alpha_equiv e (id::taken))) else
                                                                let new_id =
find_replacement_id taken alpha_list in
                                                                  let new_lambda = Lambda
(new_id, (replace_id e id new_id)) in
                                                                   alpha_equiv new_lambda
taken
        | App(e1, e2)           -> let alpha_e1 = alpha_equiv e1 taken in
```

```
                                                let alpha_e2 = alpha_equiv e2
(get_used_ids alpha_e1 taken) in
                                                 App (alpha_e1, alpha_e2)

let rec lookup x (variables,values) =
        match variables, values with
        | y::yt, z::zt -> if y = x then z else lookup x (yt,zt)
        | [], []              -> Char x

let rec beta_simp expr stack steps=
        match steps with
        | 200   -> expr
        | _            ->
                match expr with
                | Int i                       -> Int i
                | Char e                      -> lookup e stack
                | Lambda (id, e)      -> let Lambda(id2, e2) = combine_apps expr in
                                                 Close (Lambda (id2,
beta_simp e2 stack (steps+1)))
                | App (e1, e2)        ->
                                (match e1, e2 with
                                | Char c, _                   ->
                                        let new_c = beta_simp e1 stack (steps+1) in
                                         let new_e2 = beta_simp e2 stack (steps+1) in

                                                (match new_c with
                                                | Char c       -> Close (App (Char c,
new_e2))
                                                | _                    -> beta_simp
(App(new_c, e2)) stack (steps+1))
                                | Lambda (id, e), Char c      ->
                                        (match stack with
                                                | variables, values -> beta_simp e
(id::variables, Char c::values) (steps+1))
                                | Lambda (id1, expr1), Lambda (id2, expr2)   ->
                                        (match stack with
                                        | variables, values    ->
                                                let simp_lamb = beta_simp expr1
(id1::variables, e2::values) (steps+1) in
                                                 beta_simp (remove_closed simp_lamb)
(id1::variables, e2::values) (steps+1))
                                | Lambda (id1, exp1), App (exp2, exp3)       ->
                                                (match stack with
                                                | variables, values    ->
                                                        let simp_e1 = beta_simp exp1
(id1::variables, exp2::values) (steps+1) in
                                                        beta_simp (App(simp_e1, exp3))
(id1::variables, exp2::values) (steps+1))
                                | Lambda (id, exp), Close e ->
                                        (match stack with
                                        | variables, values    ->
                                                beta_simp exp (id::variables, e::values)
(steps+1))
                                | App (expr1, expr2), App (expr3, expr4) ->
                                        let simp_e1 = beta_simp (App (expr1, expr2))
stack (steps+1) in
                                                let simp_e2 = beta_simp (App (expr3,
expr4)) stack (steps+1) in
                                                beta_simp (App (simp_e1, simp_e2)) stack
(steps+1)
                                | App (expr1, expr2), _ ->
                                        let simp_e1 = beta_simp e1 stack (steps+1) in
                                         beta_simp (App (simp_e1, e2)) stack (steps+1)
                                | Close e, Close f -> Close( App( Close e, Close f))
                                | _ , Close e ->
```

```
                                    let simp_app = beta_simp e1 stack (steps+1) in
                                            beta_simp (App (simp_app, Close e)) stack
(steps+1)
                                | Close e, _ -> let rem_clo = remove_closed e in
                                                                        beta_simp (App
(rem_clo, e2)) stack (steps+1))
                | Close e              -> Close e
```

parser.mly

```
%{
        open Simplification
        open Lambda_type
        open Lib
        open Core.Std
        exception Unrecognised_syntax of string
%}


%token <int> INT
%token <char> CHAR
%token PLUS MINUS MULT DIV OPEN CLOSE EOL EOF LAMBDA DOT SUCC ADDITION MULTIPLY

%type <Lambda_type.expr option> main
%start main

%%

main:
        | EOF                                   { None }
        | EOL                                   { None }
        | EOL expr                              { Some $2 }
        | expr EOL                              { Some $1 }
        | expr EOF                              { Some $1 }
;

expr:
        | INT                                   { int_to_church $1 }
        | CHAR                                  { Char $1 }
        | expr expr                             { App ($1, $2) }
        | LAMBDA CHAR DOT expr                  { Lambda ($2, $4) }
        | OPEN expr CLOSE                       { $2 }
        | SUCC                                  { succ }
        | ADDITION                              { add }
;
```

lexer.mll

```
{
        open Core.Std
        open Parser
        exception Unrecognised_token of string
        exception LexError
}

let charList = ['a'-'z''A'-'Z']
let punc = ['!''"''#''$''%''&''\'''('')''*''+'','-
''/'':''<''=''>''?''@''['']''^''_''`''{''|''}''~']

rule read = parse
        | [' ' '\t' '\n']       { read lexbuf }
        | ['0' - '9']+ as s     { INT(int_of_string s) }
        | '\\'                  { LAMBDA }
```

```
        | '.'                 { DOT }
        | "successor"         { SUCC }
        | "addition"          { ADDITION }
        | '('                 { OPEN }
        | ')'                 { CLOSE }
        | eof                 { EOF }
        | ';'                 { EOL }
        | charList as s       { CHAR s}
        | punc as s           { raise (Unrecognised_token ("unrecognised syntax " ^
Char.to_string s) )}
        | _                   { raise LexError }
```

lambda_compiler.ml

```
open Core.Std
open Lexer
open Printf
open Simplification
open Lambda_type

let rec parse_channel lexbuf =
        let output = Parser.main Lexer.read lexbuf in
        match output with
        | Some c        ->      print_endline( "" );
                                print_endline( "   " ^ lambda_to_string c );
                                let alpha = alpha_equiv c [] in
                                print_endline( "=> " ^ lambda_to_string (beta_simp alpha
([],[]) 0));
                                print_endline( "" );
                                parse_channel lexbuf
        | None          ->      ()



let () =
        let filename = Sys.argv.(1) in
        let inx = In_channel.create filename in
        let lexbuf = Lexing.from_channel inx in
        lexbuf.lex_curr_p <- { lexbuf.lex_curr_p with pos_fname = filename };
        parse_channel lexbuf;
        In_channel.close inx
```

tests.ml

```
open Core.Std
open Lexer
open Printf
open Lambda_type
open Simplification

type test =
{       name    : string ;
        input   : string ;
        output  : string ;
}

let test_list = [
        { name = "numerals";
          input = "1";
          output = "\\s.\\z.s(z)";};

        { name = "single lambda";
```

```
          input = "\\x.x";
          output = "\\x.x";};

        { name = "lambda application";
          input = "(\\x.x)(\\y.y)";
          output = "\\y.y";};

        { name = "char on char";
          input = "c d";
          output = "c d";};

        { name = "char on lambda";
          input = "(\\x.x) c";
          output = "c";};

        { name = "lambda on char";
          input = "c (\\x.x)";
          output = "c(\\x.x)";};

        { name = "lambda on lambda";
          input = "(\\x.x)(\\y.y)";
          output = "\\y.y";};

        { name = "successor";
          input = "successor";
          output = "\\n.\\s.\\z.s(n(s(z)))";};

        { name = "addition";
          input = "addition";
          output = "\\m.\\n.\\f.\\x.m(f(n(f(x))))";};

        { name = "addition application";
          input = "addition 2 3";
          output = "\\f.\\x.f(f(f(f(f(x)))))"; };
]

let rec parse_channel lexbuf =
        let output = Parser.main Lexer.read lexbuf in
        match output with
        | Some c       ->      print_endline( "Read as:          " ^ lambda_to_string c
);
                                       let alpha = (alpha_equiv c []) in
                                       let value = print_endline( "Alpha equivalence:"
^ (lambda_to_string alpha)) in
                                       print_endline( "Simplified to:    " ^
lambda_to_string (beta_simp alpha ([],[]) 0));
                                       print_endline( "" );
                                       parse_channel lexbuf
        | None         ->      ()

let rec parse_list tests =
        match tests with
        | []    -> ()
        | x::xl -> let test = Lexing.from_string x.input in
                               let print_name = print_endline("Test Name:        " ^
x.name ) in
                                let print_input = print_endline ("Input:            "
^ x.input) in
                                 let print_output = print_endline ("Output should be: "
^ x.output) in
                                  parse_channel test;
                                  parse_list xl

let () =
        parse_list test_list
```