Fuzzy Logic in the Functional Paradigm

P15194468

Sam Fuller

2019

# Objective

The main research objective of this project is to determine if it is possible for fuzzy inference systems to be developed within the functional paradigm, and how suitable it is to do so.

# Abstract

In order to answer the main research objective, a new functional language was designed, and its compiler written, a fuzzy library was then written in the new language. The creation of the language is a focus of this paper, however; the fuzzy library is also a large factor in answering the research objective.

# Acknowledgements

John North – De Montfort University – Project Supervisor

David Elizondo – De Montfort University – Tutor for AI Programming MSc Module

Francisco Chiclana – De Montfort University – Tutor for Fuzzy Logic MSc Module

# Dictionary

|  |  |
| --- | --- |
| Term | Definition |
| Element | The program is separated into elements, an element has an identifier an assignment operator, and an output. |
| Identifier | The name of a function, it is used when defining and calling a function. |
| Function | All elements with the same identifier represent one function. |
| Argument | A value passed to a function; all arguments are immutable. |

Contents

[Objective 1](#_Toc19889897)

[Abstract 1](#_Toc19889898)

[Acknowledgements 1](#_Toc19889899)

[Dictionary 1](#_Toc19889900)

[1 Functional Language 4](#_Toc19889901)

[1.1 Background 4](#_Toc19889902)

[1.2 Functionality 5](#_Toc19889903)

[1.2.1 Types 5](#_Toc19889904)

[1.2.2 Operators 5](#_Toc19889905)

[1.2.3 Function Syntax 6](#_Toc19889906)

[1.2.4 Argument Syntax 7](#_Toc19889907)

[1.2.5 Caching 7](#_Toc19889908)

[1.2.6 Mapping 8](#_Toc19889909)

[1.3 Examples 8](#_Toc19889910)

[1.3.1 The Fibonacci Sequence 8](#_Toc19889911)

[1.3.2 The Fizz Buzz Game 8](#_Toc19889912)

[2 The Compiler 10](#_Toc19889913)

[2.1 Functionality 10](#_Toc19889914)

[2.1.1 Compiler.pl 10](#_Toc19889915)

[2.1.2 LexicalParser.pl 10](#_Toc19889916)

[2.1.3 SyntaxParser.pl 10](#_Toc19889917)

[2.1.4 JSGenerator.pl 11](#_Toc19889918)

[2.1.5 function.js 11](#_Toc19889919)

[2.2 Testing 0](#_Toc19889920)

[3 The Fuzzy Library 0](#_Toc19889921)

[3.1 Background 0](#_Toc19889922)

[3.2 Functionality 0](#_Toc19889923)

[3.2.1 Input functions 0](#_Toc19889924)

[3.2.2 Output functions 1](#_Toc19889925)

[3.2.3 Fuzzy Membership Functions 1](#_Toc19889926)

[3.2.4 Fuzzy Logic Rules 2](#_Toc19889927)

[3.2.5 Fuzzy Logic Analysis 3](#_Toc19889928)

[4 Conclusion 4](#_Toc19889929)

[4.1 The Functional Language 4](#_Toc19889930)

[4.2 The Fuzzy Library 4](#_Toc19889931)

[5 References 5](#_Toc19889932)

# Functional Language

## Background

The functional paradigm abstracts computation to the assessment of mathematical functions (Hudak, 1989). Within the functional paradigm there is no state changing or mutable data, replacing iteration with recursion. A functional program is written as a series of mathematical expressions describing relationships not the control flow of the program. This method of defining a program results in a function, that when passed identical inputs, will always result in the same output (Payton Jones, 1987).

The lack of mutability of a functional language can limit the ability of the language. Although state change can me simulated by passing around immutable states to functions, other mutable concepts cannot be implemented. Random number generators and I/O interactions are not pure functional concepts, as they cannot be expected to remain unchanged between function calls. Within Haskell monads are used to encapsulate non-deterministic computation. Monads allow for I/O and updatable state while still maintaining Haskell’s pure functional environment (Jacobs, 2015).

The language made for this project addresses this concern by implementing with JavaScript. Purely functional code can be developed and coded in the functional language, compiled, and then state changes and I/O can be implemented within JavaScript. Allowing for the programmer to program only the parts that work within a purely functional environment to be programmed in the functional language.

As part of this project, a functional language was defined, and a compiler was created. The compiler is an executable prolog script that turns a file coded in the functional language into JavaScript. Prolog was chosen for its ability to process the input file with a set of given rules pertaining to the language structure.

The language compiles to JavaScript, this has many advantages. JavaScript has a compiler with built in garbage collection and type prediction. However; the most important reason JavaScript was chosen was for its compatibility with HTML and CSS, a very capable method of input and output. In addition, programming in JavaScript will allow the language to be compiled to run on any device that has supports a modern internet browser. The ability to compile to JavaScript should not be mistaken that the purpose of the language is to replace JavaScript’s use as a web scripting language, but as a functional addition to JavaScript. This is made clear by the languages ability to integrate with JavaScript, functions within the language can be called from within JavaScript, and JavaScript functions call be called from within the code.

There are already many languages that compile to JavaScript because of its monopoly as the web programming language. Elm is a purely functional language that can compiles to JavaScript, HTML and CSS. Elm is made for web development with its ability to be used to make a complete site while JavaScript requires the presence of at least HTML. In addition, Elm boats faster benchmark times and lower asset size than many of its competitors, including reach and angular (Elm, 2019). The language made for this project is made for general purpose programming, not specifically web development, however; Elm proves how successful a functional language that compiles to JavaScript can be.

## Functionality

### Types

The language is dynamically typed using JavaScript’s type definition to allow arguments to be passed between the languages. These types include: Numbers, Strings, Boolean, null, undefined, Arrays. In the functional language there is a focus on arrays, with many operators and functionality for them. There is not currently any ability to code for objects, these must be passed individually or using an array.

### Operators

There are many operators in this language, many of them use JavaScript equivalents, some of them use pre-written functions. Here is a list of all the operators in the language:

#### Assignment Operators

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| = | Equals | Lazy assignment, calculated results are stored in memory. |
| <- | Gets | Lazy assignment, calculated results are not stored in memory. |

Assignment operators are used in element to separate the input and output and define if the output will be cached in memory. Every element will have exactly one assignment operator. These operators, or an equivalent, do not exist in JavaScript, instead a Boolean is used to store if the results will be cached along with both the inputs and output of the function.

#### Mathematical Operators

|  |  |
| --- | --- |
| Operator | Name |
| a+b | Addition |
| a-b | Subtraction |
| a\*b | Multiplication |
| a/b | Division |
| a%b | Modulus |
| a\*\*b | Exponentiation |

Mathematical operators are the same as JavaScript’s mathematical operators, this does not make compiling them easier and the operators could be changed to compile differently, however; it does make the language more consistent with JavaScript. The operators here require two number inputs with the operator positioned between them, they always return a number.

#### Logical Operators

|  |  |
| --- | --- |
| Operator | Name |
| a&&b | And |
| a||b | Or |
| !a | Not |

Logical operators are also equal to JavaScript’s Logical operators, this has the same reasoning as the mathematical operators. These operators accept one or two Booleans and always output one Boolean.

#### Relational Operators

|  |  |
| --- | --- |
| Operator | Name |
| a>b | Greater than |
| a<b | Less than |
| a>=b | Greater than or equal to |
| a<=b | Less than or equal to |
| a==b | Equal to |
| a!=b | Not equal to |

Relational operators, like both mathematical and logical operators, are the same as JavaScript’s relational operators. These operators accept two values with the operator positioned between them and always return a Boolean.

#### Array Operators

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Input | Output |
| [a..b] | Generates a List of all integers between a and b inclusive | a and b are numbers | Array |
| #a | Length of an array | a is an array | Number |
| a:b | Concatenation of two arrays | a and b are arrays | Array |
| ::a | Concatenation of an array | a is an array | String |
| :+a | Sum of an array | a is an array | Number |
| :\*a | Product of an array | a is an array | Number |
| :&a | Returns true element if ALL elements are true | a is an array | Boolean |
| :|a | Returns true element if ANY element is true | a is an array | Boolean |

Array operators do not have a JavaScript equivalent, and are instead replaced with pre-written JavaScript functions, these are stored in function.js.

#### Layout

|  |  |
| --- | --- |
| Operator | Description |
| /\*\* | Function documentation comment opener |
| /\* | Generic comment opener |
| \*/ | Generic comment closer |
| , | Separates arguments |
| | | Where, used to define arguments in inputs |
| ; | End of element |
| ( |  |
| ) |  |

The layout tokens are used for comments, and syntax. Comments and documentation are moved and left in the compiled document to allow for compiled code that is easy to read and understand. The other operators are used differently in JavaScript, so they are removed changed during syntax parsing.

### Function Syntax

Identifier(input1, input2) = input1 + input2;

A function is split into many elements, each element has an identifier, possible inputs, an assignment operator and an output. The identifier is the name of the function, many elements can have the same identifier this means they are in the same function. An identifier can have multiple or no arguments separated by commas the next section contains more information on the syntax of arguments. The assignment operator can either be an = or an <- the choice between these affects the caching, elements of the same function can have different assignment operators. The assignment operator marks the start of the output. The output contains the code, this can vary dramatically depending on the purpose of the function, however it must always evaluate to a returnable value.

An identifier must start with only the English characters A-Z and a-z but may contain a digit after the first character, only English letters and digits can be used in an identifier. The same is true of argument names.

### Argument Syntax

There are many ways of defining an argument, including Boolean logic, which must all evaluate to true for the element to be activated, the code will iterate over all elements and will activate only the first element where the inputs evaluate to true.

Identifier(input1, input2) = input1 + input2;

An element can have multiple arguments separated by commas. These arguments can be used to calculate the output.

Identifier = 3;

An element can have no arguments, in this case brackets are not needed.

Identifier(5) = 50;

An element can have an argument that is expected to be a value, in this case the element will only trigger if the first argument is equal to 5. This is the same as writing x=5, however; in this case there is no named argument that can be used within the output. This is not limited to numbers, the same can be achieved with strings.

Identifier(x>10) <- x^3;

An element can have an argument with Boolean logic, this assumes the argument is the first lexeme to appear, the argument can then be used to calculate the output.

Identifier(x| even(x)) <- x/2;

The | operator allows for an argument to appear embedded in the Boolean logic. The argument name appears before the vertical bar and the Boolean logic appears after. This allows for more complex input conditions.

### Caching

Caching is a feature that allows for operations to be completed faster, instead of calculating the result multiple times, caching allows for the result to be stored in memory. Each record of cache contains the arguments that triggered it and the result, if on another function call the arguments match an element of cache, the result in cache will be returned instead of analysing the elements. This result can only be achieved because of the nature of functional languages. It is certain that given a set of inputs a function will always return the same output, there is no stochasticity. This improves times it takes to calculate recursion in programs such as the Fibonacci sequence and also functions that are called many times, such as stringout in the Fizz Buzz game.

### Mapping

Arrays are a focus in this language, as the only data type that can store multiple arguments. Mapping functions over data has been added to allow for a single function to be applied to all elements in an array. One of the advantages of doing this in a functional language means that this can be easily delegated to many threads, allowing for simple multithreading of a program. Mapping a function can easily be done by adding the keyword ‘map’ to the start of a function, for example mapfunction(array) will apply function to every element of array. If more arguments are needed to be passed they can be passed in addition to the array. Mapfunction(array, arg1) will map over each element of array, where each thread will have access to the same arg1. This allows for functions such as mapadd(array, number) to be made where each element of array increased by number.

## Examples

### The Fibonacci Sequence

/\*\*Fibonacci sequence is declared recursively\*/

/\*declaring known fib numbers\*/

fib(0) <- 0;

fib(1) <- 1;

/\*declaring unknown fib numbers recursively\*/

fib(x) = fib(x-1) + fib(x-2);

The Fibonacci sequence is the sequence of numbers where each element is defined as the sum of the previous two. The second to fifth Fibonacci numbers are 1, 2, 3, 5, 8. This example shows recursion within the program, the final element has caching enabled to significantly speed this recursion up. The output of this program is the element of the Fibonacci sequence in x position.

### The Fizz Buzz Game

/\*\*defining fizz recursively\*/

fizz(x<3) <- "";

fizz(3) <- "fizz";

fizz(x) <- fizz(x-3);

/\*\*defining buzz recursively\*/

buzz(x<5) <- "";

buzz(5) <- "buzz";

buzz(x) <- buzz(x-5);

/\*\*the output is the concatenation of both fizz and buzz\*/

stringout(x) = fizz(x)+buzz(x);

/\*\*if the cardinality of out for x is 0, return x, else return the concatenation of out\*/

singleout(x|stringout(x) == "") = x;

singleout(x) = stringout(x);

fizzbuzz(x>0) <- mapsingleout([0..x]);

The Fizz Buzz game is a sequence, where each multiple of 3 is replaced with fizz, each multiple of 5 is replaced with buzz and multiples of both are replaced with fizzbuzz. The first 20 values of the sequence are: 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz, 16, 17, fizz ,19, buzz. This example shows string and list manipulation within the language.

# The Compiler

## Functionality

The compiler has many stages, each is broken down into a different file.

### Compiler.pl

Compiler.pl contains the main function that calls all the other functions. To use this function the Compile(‘input file’, ’output file’) command is used. This function first opens the input file and passes it to the lexical parser. The lexemes are then passed to the syntax parser. The syntax tree and output file are finally passed to the code generator.

### LexicalParser.pl

The lexical compiler takes the file input and turns it into a list of lexemes. This file contains content from an efficient tokenizer designed by M Covington (Covington, 2012). All code made by M. Covington has a comment above to show which parts have been developed by him.

These lexemes represent small segments of code, the lexical compiler does not order or change these lexemes, it simply extracts it from the file. Each character is assigned a type: white space, operator, string operator, bracket, letter. The characters are combined into lexemes based on type and arrangement, there are many different types of lexeme:

* Name: a series of letters or digits where the first character must be a letter. They are used for both function names and arguments.
* Operator: a series of one or more operator characters. These are used by the language as the operators.
* String: A string is anything surrounded by quotation marks, a string may contain a quotation mark if it is preceded by a backslash.
* Number: A number is a series of digits; a number may contain a decimal place if it is succeeded by a digit. Note that numbers do not contain a negative sign, this is classified as an operator.
* Bracket: Any bracket character, is kept separate from operators so the compiler can easily identify them.
* Special: An undefined character.
* Documentation: A comment in the format /\*\* comment \*/. Documentation is kept separate from comments so that it can be placed before the main function, this allows it to work with JavaScript’s Javadoc.
* Comment: A comment in the format /\* comment \*/. Comments are placed above their element, and will be compiled so that they stay with their element allowing for readable JavaScript compiled code.

### SyntaxParser.pl

The syntax parser contains four functions called by Compiler.pl, one to check imports, one to parse the lexemes, one to change operators, and one to fix internal function calls.

The check imports function ensures all inputs are compiled into the same output file. All inputs are strings at the start of the file. The function checks for string lexemes at the start of the file, if one is detected the file is loaded and passed to the lexical parser. The lexemes are then also checked for imports recursively, this allows for a chain of imports with no maximum. The import lexemes are combined at the end of the main file, this allows elements in the main file priority over imported function elements.

The syntax parser collects all lexemes into a list of functions. A function contains an identifier and content, the content consists of documentation, comments and elements. Documentation is defined as belonging to the function of the next element declared. Comments are defined as belonging to the next element declared. Elements are split up into the assignment operator, inputs, and output. The inputs contain multiple arguments the names of which are changed to numbers, both the inputs and output are updated to reflect the new numerical arguments.

The check operator function updates operators so that they respect the elements around them and surrounds their inputs in brackets.

The check function function updates functions so that all arguments are properly bracketed. The code is compiled so that arguments are passed in an array, therefore function calls need to be confined within square brackets. Only internal function calls are handled this way,

### JSGenerator.pl

The JSGenerator file prints JavaScript code to the output file, most of the functions in this file are for formatting the code properly. For each function in the syntax tree there are five JavaScript components:

* The elements: An array of the elements of the function, it contains caching information, the input predicates and output function. These are stored within JavaScript as lambda functions. Comments are placed above the relevant element.
* The cache: Initialised as an empty array, this will be filled with arrays of values. A single cached element will have an output and several inputs. At runtime, before the element array is considered, the cache is searched to find an element which matches the inputs already.
* The function: The method used to call the function directly, function documentation is placed here this allows it to be picked up by Javadoc.
* The map function: The method used to map the function to a list, this function splits up the array and calls the main function on multiple threads.
* The html function: This is a convenience method for assigning the result to an html element, it has no use within the language, however; makes it easier for HTML users to access the function without having to know JavaScript.

### function.js

This file is not part of the compiler, it is pre-written code, it must be present and imported for compiled code to work, it contains essential functions for caching and operators. Salfunction is the function that handles cache, it is called from every call within the functional language. The cache is checked before the elements to find a valid input, the cache stores the output values directly, whereas the elements store a lambda function, if no valid input is found null is returned. Other functions also exist for the array operators.

## Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Tested Feature | Test | Input | Result |
| = Assignment Operator | Ensure the = operator enables caching | From fib.sal:  /\*declaring unknown fib numbers recursively\*/  fib(x) = fib(x-1) + fib(x-2); | From fib.js  /\*declaring unknown fib numbers recursively\*/     [true,  True flag means caching is on |
| <- Assignment Operator | Ensure the <- operator disables caching | From fib.sal:  /\*declaring known fib numbers\*/  fib(0) <- 0;  fib(1) <- 1; | From fib.js:  /\*declaring known fib numbers\*/     [false,     [false,  False flags mean caching is off |
| + Mathematical Operator | Test mathematical operators, all operators were tested only + and - are shown this is because they are all compiled in the same way | From fib.sal:  fib(x) = fib(x-1) + fib(x-2); | From fib.js:  (*vars*)*=>*{return fib([vars[0]-1])+fib([vars[0]-2])}  Mathematical operators are the same as JavaScript, so they are maintained. |
| && Logical Operator | Test logical operators, all operators were tested, only && is shown because they are all compiled identically | From fuzzyInMF.sal, fuzzyInMFTriangular:  input>leftFoot && input<=shoulder | From fuzzyexample.js:  (*vars*)*=>*{return vars[3]>vars[0]&&vars[3]<=vars[1]}  Input predicate contains only the Boolean logic from the required inputs, logic is correct. |
| > Relational Operator | Test relational operators, all operators were tested, only > and <= are shown because they are all compiled identically. | From fuzzyInMF.sal, fuzzyInMFTriangular:  input>leftFoot && input<=shoulder | From fuzzyexample.js:  (*vars*)*=>*{return vars[3]>vars[0]&&vars[3]<=vars[1]}  Input predicate contains only the Boolean logic from the required inputs, logic is correct. |
| [a..b] Array Operator | Test the array constructor method | From tests.sal:  arrayConstructor <- [1..10]; | From tests.js:  salListGenerator(1,10)  Code is generated correctly |
| #a Array Operator | Test the array length operator | From tests.sal:  arrayLength <- #arrayConstructor; | From tests.js:  salLength(arrayConstructor([])) |
| a:b Array Operator | Test the concatenate arrays operator | From tests.sal:  concat2Array <- arrayConstructor:arrayConstructor; | From tests.js:  arrayConstructor([]).concat(arrayConstructor([])) |
| ::a Array Operator | Test the concatenate array operator | From tests.sal:  concat1Array <- ::arrayConstructor; | From tests.js:  salJoin(arrayConstructor([])) |
| :+a Array Operator | Test the sum operator | From tests.sal:  addArray <- :+arrayConstructor; | From tests.js:  salSum(arrayConstructor([])) |
| :\*a Array Operator | Test the product operator | From tests.sal:  prodArray <- :\*arrayConstructor; | From tests.js:  salProduct(arrayConstructor([])) |
| :&a Array Operator | Test the all operator | From tests.sal:  allArrayFalse <- :&mapsomePred(arrayConstructor); | From tests.js:  salAll(mapsomePred([arrayConstructor([])])) |
| :|a Array Operator | Test the any operator | From tests.sal:  anyArrayFalse <- :|mapnoPred(arrayConstructor); | From tests.js:  salAny(mapnoPred([arrayConstructor([])])) |
| Documentation | Test documentation is placed above the main function | From fib.sal:  /\*\*fibonacci sequence is declared recursively\*/ | From fib.js:  /\*\*fibonacci sequence is declared recursively\*/  *function* fib(*variables*){  Documentation remains unchanged and in the correct place, as expected. |
| Comment | Test comments are unchanged and placed above their element | From fib.sal:  /\*declaring known fib numbers\*/  fib(0) <- 0; | From fib.js:  /\*declaring known fib numbers\*/     [false, |
| No Arguments | Test when there are no arguments on an element | From tests.sal:  arrayConstructor <- [1..10]; | From tests.js  (*vars*)*=>*{return true} |
| Single Argument | Test when there is one argument to an element | From fizzbuzz.sal:  fizz(3) <- "fizz"; | From fizzbuzz.js  (*vars*)*=>*{return vars[0]==3} |
| Multiple Arguments | Test when there are multiple arguments to an element | From fuzzyexample.sal:  fuzzyInputs("any", inputs) <- null; | From fuzzyexample.js  (*vars*)*=>*{return vars[0]=="any"&&true} |
| Boolean Logic | Test that Boolean logic compiles | From fizzbuzz.sal:  fizz(x<3) <- ""; | From fizzbuzz.js:  (*vars*)*=>*{return vars[0]<3} |
| | Operator and Boolean Logic | Test that the where operator works | From fizzbuzz.sal:  singleout(x|stringout(x) == "") = x; | From fizzbuzz.js:  (*vars*)*=>*{return stringout([vars[0]])==""} |
| Cache | Run test to check the cache works | Fib.sal is checked where fib(x) is and is not cached, it is tested with fib(1000). | Without caching the command fails after several minutes of processing, with caching the result is found almost instantly. |
| Map Single Argument | Run test to check map works | From tests.sal:  allArrayFalse <- :&mapsomePred(arrayConstructor); | From tests.js:  salAll(mapsomePred([arrayConstructor([])]))  code has been run and the map function works as expected. |
| Map Multiple Arguments | Run test to check map works with multiple arguments | From fuzzylibrary.sal:  fuzzyTotalMembership(rule, input)  <- :+mapfuzzyInputs(rule[0], input)/ #input; | Code has been run and the map function works as expected. |

# The Fuzzy Library

## Background

Set logic attempts to classify items as either in a set or not, fuzzy logic introduces a new dimension: membership. This allows fuzzy logic to define an item as having a membership to a set, for example: 25 degrees could be considered cold for food, or warm, this all depends on context. The membership is defined on a scale of 0 to 1, an object with a membership of 0 not being contained within the set, and an object with a membership of 1 belonging fully to that set.

A fuzzy inference system uses fuzzy logic to aid a computer gain a human level of understanding of language. Fuzzy inference systems can be used to classify items, simulate processes, decision support tools and process control (Guillaume, 2001) (Das, et al., 2016). A fuzzy inference system allows for many inputs to be understood and an output produced with all logic in a fuzzy state, allowing for human readable rules. The similarity of the rules to human though enables fuzzy systems that solve complex problems to be coded and understood more easily than a traditional mathematical approach. Within this project, a focus should be made on making the logic of the system more human readable. (Zadeh, 1973) describes how the English language can be used to describe fuzzy rules, using linguistic variables. This further increases how easy it is to understand the fuzzy inference system.

For a fuzzy inference system to work, the input data (a real-world analogue value) must be fuzzified to show membership to a set. The values membership to different sets (cold, warm, hot) can be used to determine an output. To do this the membership values are used with different rules, the rules define which input memberships must be high for which outputs to also be high. The output of this stage is then analysed to give an analogue value. It is important to note that the input and output are both analogue values, it is simply the logic that happens within a fuzzy membership environment (Torra, 2010).

## Functionality

### Input functions

fuzzyInputs("any", inputs) <- null;

fuzzyInputs("temp-cold", inputs) <- fuzzyInMFTriangular(0,5,30,inputs[0]);

fuzzyInputs("temp-warm", inputs) <- fuzzyInMFGaussian(40,20,inputs[0]);

fuzzyInputs("temp-hot", inputs)

<- fuzzyInMFTrapezoidal(50,85,95,100,inputs[0]);

The user must format the input functions, the fuzzy Input function is used within the library to determine the membership of an input to the function in accordance with the rules. In the example: the input is the ideal temperature of the food, and the output is the length of time needed in a microwave. The first argument is the name of the name of the membership function, here it has been formatted as “input name-function name”. This allows for a second set of inputs to be easily added, for example to add microwave power an additional set of functions could be added:

fuzzyInputs(“power-low”, inputs) <- fuzzyMFGaussian(300,300,inputs[0]);

fuzzyInputs(“power-high”, inputs) <- fuzzyMFGaussian(600,200,inputs[0]);

### Output functions

fuzzyOutputs("any", membership) <- null;

fuzzyOutputs("time-short", membership)

<- fuzzyOutMFTriangular(0,5,10,membership);

fuzzyOutputs("time-long", membership)

<- fuzzyOutMFTrapezoidal(5,30,40,membership);

The user must format the output functions, the fuzzy output function is used within the library to determine the output of a membership to a rule. A limitation of the library is that there can only be one output.

### Fuzzy Membership Functions

Within the library several membership functions have been implemented, these are triangular, trapezoidal, and gaussian. Each of these membership functions has an input function that takes a real-life value and returns a fuzzy set membership value between 0 and 1. In addition to these the inverse has also been created, the inverse is calculated using the middle of the maximum and turns a membership value into a real-live.

#### Triangular Membership

The triangular membership function resembles a triangle, it consists of two lines, one positive in gradient, the other negative. The left foot (lf) is where the positive gradient line meets y = 0. The right foot (rf) is where the negative gradient line meets y = 0. The shoulder (s) is where the two functions intersect. It is represented by the formula:

This formula can easily be coded into the language and appears in the fuzzyInMF.sal library.

The inverse of the triangular membership takes the middle of the maximum, to achieve this the average of the two points at which the membership intersects the two lines lf-s and s-rt is taken.

#### Trapezoidal Membership

The trapezoidal membership function resembles a trapezoid, it consists of three lines, one positive in gradient, one negative and one where y = 1. The left foot (lf) is where the positive gradient line meets y = 0. The right foot is where the negative gradient line meets y = 0. The left shoulder (ls) is where the positive gradient line meets y = 1. The right shoulder (rs) is where the negative gradient line meets y=1. It is represented by the formula:

The inverse of the trapezoidal membership takes the middle of the maximum, to achieve this the average of the two points at which the membership intersects the two lines lf-ls and rs-rt is taken.

#### Gaussian Membership

The gaussian membership function consists of a gaussian bell curve. It can be determined by the formula:

The middle of maximum is easy to calculate, because the formula is always symmetrical, the centre is always the middle.

### Fuzzy Logic Rules

/\*if temperature is cold, then time will be short\*/

rule1 <- [["temp-cold"], "time-short"];

/\*if temperature is warm, then time is both short and long\*/

rule2 <- [["temp-warm"], "time-short"];

rule3 <- [["temp-warm"], "time-long"];

/\*if temperature is hot, then time is long\*/

rule4 <- [["temp-hot"], "time-long"];

/\*\*define the rule list as the concatenation of each rule\*/

ruleList <- [rule1, rule2, rule3, rule4];

The user must define the rules. Each rule is defined within an array, this must be two items long, the input array and then the output. The input array can contain as many inputs as necessary and the average membership is calculated. The string inputs make for a very readable set of rules. The rules must then be concatenated into an array, this is so they can be passed to the fuzzy logic library. The fuzzy inference system made to show the uses of the library is not accurate, but nor does it need to be, its purpose is only as an example of capability. The rules, input and output functions are not reliable and cannot be used to accurately predict the time to temperature functionality of a microwave.

### Fuzzy Logic Analysis

/\*\*Returns the average membership of a single rule\*/

fuzzyTotalMembership(rule, input)

<- :+mapfuzzyInputs(rule[0], input)/ #input;

/\*\*Returns the fuzzy output of a single rule\*/

fuzzyAssessAllRules(rule, input)

<- fuzzyOutputs(rule[1], fuzzyTotalMembership(rule, input));

/\*\*Assesses an input with the given rules, fuzzyInputs and fuzzyOutputs must be declared\*/

fuzzyAssessInput(rules, input)

<- :+mapfuzzyAssessAllRules(rules, input) /

:+mapfuzzyTotalMembership(rules, input);

The fuzzy library also contains functions for analysing an input, within the analysis the average input membership of each rule is calculated. The input is then passed through the correct output membership function to calculate the output of a rule, a weighted output is then calculated using the memberships and outputs of each rule, this provides the output.

# Conclusion

## The Functional Language

Many conclusions can be drawn from the implementation of a functional language that compiles to JavaScript.

HTML + CSS is an excellent output for any language, it can be used to very quickly create a better interface than a console. Although the functional language is not good at controlling the HTML page, because of its lack of ability to handle with changing state and timing, an important part of modern web design; HTML is still an excellent output for the language. The conclusion here is that the language should be thought of more as a general-purpose language, that outputs to HTML, unlike JavaScript which is a web programming language that controls an HTML page.

The language works exceptionally well with JavaScript, with JavaScript code able to call functions within the language and JavaScript functions able to be called by the language. Even considering the previous conclusion that this functional language is not great at web development, this fact allows for parts of a program to be done functionally and other parts within JavaScript. This compatibility between the languages allows for the best of both languages to be used at any time within the same program.

## The Fuzzy Library

Fuzzy logic works exceptionally well within the functional language. Fuzzy inference systems do not require any loops or recursion, meaning that the immutability of the functional paradigm means the code is easier to understand. The map function contained within the language allows for all membership functions of all rules to be assessed in parallel. This allows for faster code to be created with no thought to the process behind the code, a common trait across all higher level languages.

In addition to this, the human readability of the rules is excellent. Despite not being as clear as has been proposed by other programmers (Zadeh, 1973), the rules are very easily readable with an if x then y layout. With all of this being considered, it can be concluded that not only is fuzzy logic possible in the functional paradigm, but that fuzzy logic favours the functional paradigm.

# References

Covington, M. A., 2012. *ET: an Efficient Tokenizer in ISO Prolog,* s.l.: s.n.

Das, S., Guha, D. & Bapi, D., 2016. Medical diagnosis with the aid of using fuzzy logic and intuitionistic fuzzy logic. *Applied Intelligence,* 45(3), pp. 850-867.

Elm, 2019. *Elm - A delightful language for relaiable webapps.* [Online]   
Available at: https://elm-lang.org  
[Accessed 28 8 2019].

Guillaume, S., 2001. Designing fuzzy inference systems from data: An interpretability-oriented review. *IEEE Transactions on Fuzzy Systems,* 9( 3), pp. 426-443.

Hudak, P., 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys,* 21(3), pp. 359-411.

Jacobs, B., 2015. Dijkstra and Hoare monads in monadic computation. *Theoretical Computer Science,* Volume 604, pp. 30-45.

Payton Jones, S., 1987. *The Implementation of Functional Programming Languages.* s.l.:Prentice Hall.

Torra, V., 2010. Hesitant fuzzy sets. *International Journal of Intelligent Systems.*

Zadeh, L. A., 1973. Outline of a New Approach to the Analysis of Complex Systems and Decision Processes. *IEEE Transactions on Systems, Man, and Cybernetics,* 3(1), pp. 28-44.