Thesis

Miskolci Egyetem

Parser and Interpreter Development for the Fuzzy Behavior Description Language in MATLAB/Octave

Author:

Balázs Nagy

Computer Science

Advisor:

Dr. Tamás Tompa

Miskolc, 2024

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Nagy Balázs (EIO1RQ) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: fuzzy logika, fuzzy viselkedés, fbdl, fuzzy viselkedésleíró nyelv, fbdl fordító

A szakdolgozat címe: Értelmező Fejlesztése a Fuzzy Viselkedésleíró Nyelvhez MATLAB és GNU/Octave Környezetben

A feladat részletezése:

Egy rendszer működésének a leírása különféle módokon valósulhat meg. A dolgozat ennek egy deklaratív leírási módjával, egy fuzzy szabályinterpolációs módszerre épülő automatával és annak leírónyelvével foglalkozik. A cél az, hogy MATLAB, illetve GNU/Octave nyelven is elérhető legyen egy olyan függvénykönyvtár, amely segítségével az FBDL nevű fuzzy viselkedésleírónyelvben megfogalmazott működés egyszerűen implementálhatóvá válik. A dolgozat egy rövid áttekintéssel bemutatja a fuzzy logikával kapcsolatos elméleti és a korábban elért eredményeket, majd szemlélteti az ezkre épülő leírónyelvnek a sajátosságait, annak feldolgozási módját és az automata működését. Összeveti a MATLAB és GNU/Octave implementáció módját. Konkrét példákon és szimulációkon keresztül bemutatja annak használatát.

Témavezető: Dr. Tompa Tamás

Konzulens(ek): Piller Imre

A feladat kiadásának ideje:

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Nagy Balázs**; Neptun-kód: `EIO1RQ` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Értelmező Fejlesztése a Fuzzy Viselkedésleíró Nyelvhez MATLAB és GNU/Octave Környezetben* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Contents

1 Introduction **1**

2 Core Concepts and Previous Research **3**
2.1 Fuzzy Logic . . . 3
2.1.1 Fuzzy Set . . . 3
2.1.2 Various Membership Functions . . . 5
2.1.3 Dealing with Multiple Fuzzy Sets . . . 5
2.2 Applications of Fuzzy Logic . . . 6
2.2.1 Healthcare . . . 6
2.2.2 Chemical Science . . . 7
2.2.3 Optimization Problems (Operations Research) . . . 7
2.2.4 Behaviour Control . . . 7
2.3 Fuzzy Rules . . . 7
2.3.1 Fuzzy Inference . . . 8
2.4 Fuzzy Automaton and Behavior Control . . . 8
2.5 Fuzzy Behavior Description Language . . . 9
2.5.1 Motivation . . . 10
2.5.2 Language Specifications . . . 10

3 Design **12**
3.1 Requirements and Technical Considerations . . . 12
3.2 Structure of The Interpreter . . . 13
3.3 Data Structures . . . 14
3.3.1 Lexer . . . 15
3.3.2 Token . . . 16
3.3.3 Syntax Tree . . . 17

4 Implementation **18**
4.1 Lexical Analyzer . . . 18
4.1.1 Identifier (reserved) . . . 19
4.1.2 Terminal (reserved) . . . 19
4.1.3 String . . . 19
4.1.4 Number . . . 20
4.2 Parser . . . 20
4.3 Engine . . . 24
4.4 Error Handling . . . 26
4.5 Future Extensions . . . 27

5 Testing **28**
5.1 Unit Tests 28
5.2 Performance 29
5.3 Usage 29
5.4 Rule Surfaces 30

6 Summary **32**

Irodalomjegyzék **33**

Chapter 1

Introduction

This thesis work presents an interpreter library/package for the Fuzzy Behavior Description Language (FBDL) implemented in the Octave programming language.

This initial description of the problem at hand can either be informative or utterly confusing for anyone reading it for the first time, simply because it entails many concepts perhaps still unknown to the reader. But it is quintessential to state the main topic being tackled as to not lose sight of it amidst the following discussions that will eventually lead up to the task itself. Anyone with a moderate to advanced knowledge in the field of programming can easily comprehend the workings of an interpreter if described properly, however simply mentioning that without any prior introduction to the underlying principles would still leave a fairly large gap in the reader's mind concerning the motivation behind such a language and also its use of a myriad of fuzzy logic related ideas. Therefore it is necessary to treat the subject as a whole and describe not only the technical implementations and results, but the theory as well, on which all of it is built.

With this in mind, the work is split into two main sections along with this preceding foreword to allow for some clarifications and provide a greater description of the whole subject matter. The first half exposes the reader to core concepts related to fuzzy logic and incrementally builds them up into its more complex and intricate applications. Also in this part the idea of behavior control and fuzzy state machines are presented along with various mathematical models to help with formalization; it also goes into detail about the specifications of the aforementioned Fuzzy Behavior Description Language. The second half of the work delves into the implementation and inner workings of the interpreter. In the beginning, decisions regarding language specific implementation and other architectural considerations are discussed; possible alternatives are slightly touched upon. Following a general overview of the process of interpreting a language, each stage and their operations are shown separately in detail along with possible corner cases that may require special attention and samples of unit tests to check the integrity and correct operation of the program. Finally the reader is provided with working examples of source code written in FBDL and also a demonstration of said code, where the output of the interpreter can be verified.

The main difficulty lies in connecting the different parts of the overarching subject, so as to allow the reader to indulge in this work without getting lost consider the following short explanation as a guide to the various topics about to be presented.

The motivation behind creating a programming language, be it any kind, is always attributed to the existence of a specific problem it is trying to solve. This could range

from low-level hardware management, such as those found in embedded systems, all the way to server-side applications and numerical analysis.

The language (FBDL) appearing in this work has been constructed to serve a particular application of fuzzy logic, namely that of behavior control. For example when a system, due to some event, reacts or behaves in a certain way based on predefined rules dictating its appropriate response. A fundamental property of fuzzy logic, essentially the fact that it is continuous, make it a prime candidate for such a use, since natural systems are hard, sometimes nearly impossible to accurately model with Boolean-logic.

Describing how such a system would operate, the rules it would follow and the actions it would take can be tricky to model with ordinary programming languages. Therefore, an easier alternative was designed with the primary aim of facilitating the ease of use, particularly even if the user happens to lack any kind of previous formal experience in the field of programming. To further simplify the task of using this language it takes another useful property of fuzzy logic that arises from its continuity: we are able to describe the state of a fuzzy variable or in other words the degree to which it satisfies a certain statement with the help of natural language in contrast to using concrete numerical values.

There exits many applications for fuzzy logic and its extensions, some requiring complex calculations and methods that are quite conveniently present in scientific programming languages such as MATLAB and Octave. For this reason an interpreter library in these languages would provide great utility for programs already using fuzzy logic and open new opportunities for those seeking to venture into such areas.

These ideas constitute the majority of the work, so they shall be further examined at length in the following chapters.

Chapter 2

Core Concepts and Previous Research

Before diving into the operations of the FBDL interpreter some prerequisite knowledge is essential, namely that of fuzzy logic and the many related concepts that stem from it. Furthermore, examining some research previously conducted and demonstrating various real world applications of these concepts. These ideas constitute the mathematical models that serve as the foundation and building blocks for the Fuzzy Behavior Description Language. And in turn we must consider these important aspects when building the interpreter itself as it uses the very same principles.

2.1 Fuzzy Logic

In order to gain a sound understanding of the idea of *fuzziness* we must first familiarize ourselves with the notion of fuzzy sets. The concept was first introduced and described by mathematician Lotfi A. Zadeh in 1965 as an extension to classical sets [14]. The key difference between ordinary sets and fuzzy ones is simple: In the case of the former all elements are either a part of a set or not, where as in the world of fuzzy sets an element may belong to multiple sets. The measure of how much an element is part of a given set is referred to as its *degree of membership* and is calculated with the aid of the *membership function*.

2.1.1 Fuzzy Set

As opposed to classical sets, every element in a fuzzy set has an additional property beside its value that being the degree to which that given element is the set. These aspects are more formally defined in the following section.

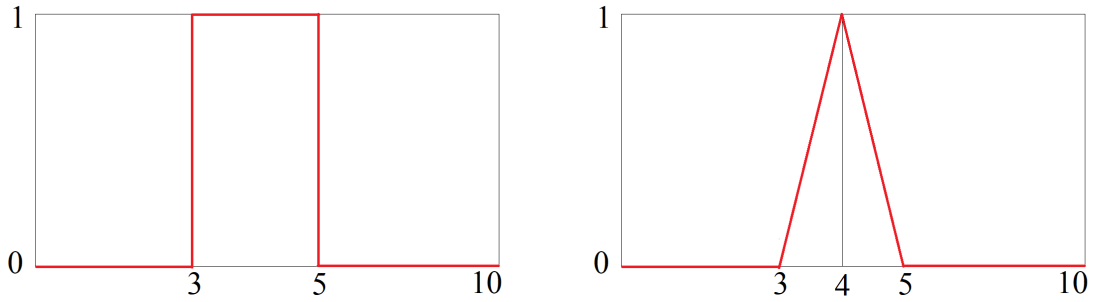
2.1. definition. Let U, referred to as the *universe of discourse*, be a set containing all the elements we wish to describe and define $m : U \to [0, 1]$ as a membership function. The pair (U, m) forms a fuzzy set A in which $\forall x \in U$ the value given by $m(x)$ is called the degree of membership of x. The function $m(x)$ is equivalent to $\mu_a(x)$.

Taking the example from Claudio Moraga's *Introduction to fuzzy logic* (2005) [10]: given the interval $[0, 10]$ of the real line as our universe of discourse and the statement "x is between 3 and 5", we may represent it with the function $\mu_{3-5} : [0, 10] \to [0, 1]$. Where $\mu_{3-5}(x) = 1$ if $3 \leq x \geq 5$, and $\mu_{3-5}(x) = 0$ otherwise as seen on 2.1a below. This function describes the classical set $[3, 5]$. Consider now the statement "x is near 4". The proximity, or nearness to the number 4 can be represented as $4 - \varepsilon$, given the

assumption that ε is a sufficiently small positive real number. Values obtained by the continued subtraction of ε will have a decreasing *"degree of nearness"* to 4 until the value, and subsequently those smaller then itself, is no longer considered to be "near" the number 4. Repeating this experiment with $4 + \varepsilon$ and the continued addition of ε will yield symmetric results. If we take the function $\mu_{near4} : [0, 10] \to [0, 1]$ to represent this statement just as previously, it becomes apparent that it cannot be of the same kind as μ_{3-5} (that lead to a classical set). If we assume that 3 and 5 are acceptable limit points for "near 4" and marking these as , $\alpha_{\min} = 3, \alpha_{\max} = 5, \beta = 4$, then

$$\mu_{near4}(x) = \begin{cases} 0, & x \leq \alpha_{\min} \text{ or } x \geq \alpha_{\max}, \\ 1, & x = \beta, \\ \frac{x-\alpha_{\min}}{\beta-\alpha_{\min}}, & \alpha_{\min} < x < \beta, \\ \frac{\alpha_{\max}-x}{\alpha_{\max}-\beta}, & \beta < x < \alpha_{\max}. \end{cases}$$

The function will be continuous and increasing for $3 < x < 4$ and will be continuous and decreasing for$4 < x < 5$. Without further information, linear transitions will be chosen as shown in 2.1b. μ_{near4} represents a **fuzzy set**.



(a) Classical set (b) Fuzzy set

Figure 2.1: Difference in steepness during the transition from 0 to 1.

Other than assigning values linearly to elements not fully contained inside a fuzzy set any kind of membership function can be utilized, but by far the most common is the previously mentioned linear way, which produces a trapeziod shape during visualization (Triangles arise, when the upper side of the trapezoid is a point). As mentioned by [4], in terms of terminology the following expressions are defined regarding any given fuzzy set:

Core (Elements where the membership function is 1):

$$\text{core}(A) = \{x \in U | \mu_\alpha(x) \geq \alpha\}.$$

Support (Elements where the membership function is greater than 0):

$$\text{support}(A) = \{x \in U | \mu_\alpha(x) > 0\}.$$

Boundary (Elements where the membership function is between 0 and 1):

$$\text{boundary}(A) = \{x \in U | 0 < \mu_\alpha(x) < 1\}.$$

Height (The height of the fuzzy set A is the maximum value taken on by the membership function):

$$\text{height}(A) = \{x \in U | \max(\mu_\alpha(x))\}.$$

2.1.2 Various Membership Functions

An example of the different shapes that a membership function may take include the following cases and their respective definitions appearing in [4]:

Triangular (Same as in the above example, a special case of the trapezoid):

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{\min} \text{ or } x \geq \alpha_{\max}, \\ 1, & x = \beta, \\ \frac{x-\alpha_{\min}}{\beta-\alpha_{\min}}, & \alpha_{\min} < x < \beta, \\ \frac{\alpha_{\max}-x}{\alpha_{\max}-\beta}, & \beta < x < \alpha_{\max}. \end{cases}$$

Trapezoidal (With the upper side (core) taking values from $[\beta_1, \beta_2]$):

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{\min} \text{ or } x \geq \alpha_{\max}, \\ \frac{x-\alpha_{\min}}{\beta_1-\alpha_{\min}}, & \alpha_{\min} < x < \beta_1, \\ \frac{\alpha_{\max}-x}{\alpha_{\max}-\beta_2}, & \beta_2 < x < \alpha_{\max}. \end{cases}$$

Γ-membership function:

$$\mu_A(x) = \begin{cases} 0, & x < \alpha, \\ 1 - e^{\gamma(x-a)^2}, & \alpha_{\min} < x < \beta_1. \end{cases}$$

S-membership function:

$$\mu_A(x) = \begin{cases} 0, & x \leq \alpha_{\min} \text{ or } x \geq \alpha_{\max}, \\ 2\left(\frac{x-\alpha_{\min}}{\alpha_{\max}-\alpha_{\min}}\right)^2, & \alpha_{\min} < x < \beta, \\ 1 - 2\left(\frac{x-\alpha_{\min}}{\alpha_{\max}-\alpha_{\min}}\right)^2, & \beta < x < \alpha_{\max}. \end{cases}$$

Logistic function:

$$\mu_A(x) = \frac{1}{1 + e^{-\gamma x}}.$$

Exponential-like function:

$$\mu_A(x) = \frac{1}{1 + \gamma(x-\beta)^2},$$

where $\gamma > 1$.

Gaussian function:

$$\mu_A(x) = e^{-\alpha(x-\beta)^2}.$$

Besides these, any function that fits the intended purpose of characterizing a certain fuzzy set is acceptable and it is left up to the decision of experts in the given field to decide which one is most appropriate.

2.1.3 Dealing with Multiple Fuzzy Sets

There are some cases, where precise numerical measurements might not be required or even be detrimental, for example stating someone's age as being 17 years 32 days and 8 hours old, does not necessarily demand such accuracy. It is much more sensible to

describe that person simply as young. This notion of the use of words in our statements instead of numerical values was introduced by Zadeh in 1975 and is called a **linguistic variable**. The varying values taken by such a variable can be described by **linguistic terms**, such as *low, middle, high, very small, average, large*, meaning we are able to take advantage of natural language, thus making it easier to work with.

For a simple example consider one's age as a variable, and the two sets: young and old. A person who is 5 years of age is considered very young and not at all old, similarly someone in their twenties may be called young, but slightly old as well, however a middle aged individual of 43 years is neither very young nor very old, but rather an even mix of both. Representing the two fuzzy sets, young and old, on figure 2.2 we can see they overlap. Any value at this given interval of overlay is a linear combination of, in this particular case, both of these fuzzy sets. The number of sets can, of course, be extended and then the values at these intersection would be a linear combination of all the defined sets, given that they overlap. From giving proper definitions of how to operate on these linguistic variables arises the notion of **fuzzy logic** and subsequently it serves as the basis for many advanced concepts such as: inference, fuzzy decision making and fuzzy control.

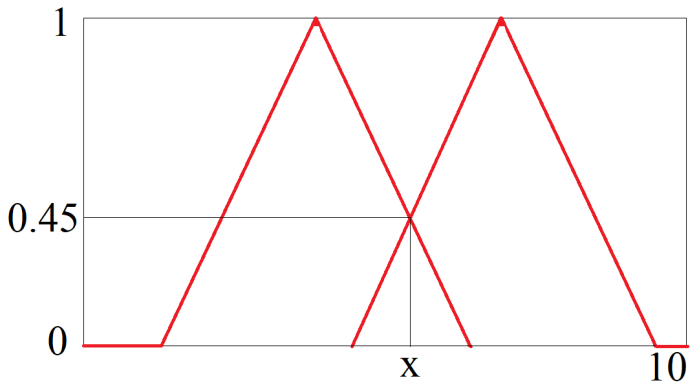


Figure 2.2: Overlapping sets

2.2 Applications of Fuzzy Logic

Many fields make use of fuzzy logic and all the differing unique characteristics from ordinary sets that it has to offer. Fuzzy logic is best suited for problems that may be hard to define or model precisely with Boolean-logic. A collection of some notable examples of already tried and implemented solutions explored by [12] are discussed in this section.

2.2.1 Healthcare

Due to the intrinsic non-linearity of biomedical systems it is difficult to accurately model various processes. Regulation of blood pressure in the case of medical patients has been tested with the help of a real time drug delivery system that used an integrated fuzzy controller. Separately, it has also been shown that test reports yield estimates of likelihood rather than confirmation of presence or absence of a disease, hence these

empirical estimates can be treated as the output of a membership function and used as such in fuzzy inference modeling [5].

2.2.2 Chemical Science

A fuzzy control system was used to both apply current to a series of anodes protecting an underground pipeline and to minimize the system's power consumption. For comparison the system used 126 *fuzzy rules* (further discussed in the following chapter) and an empirically adjusted membership function to optimize the model. Another study conducted in relation to pH measurement in waste water adopted fuzzy logic to calculate errors and acceptable levels of pH in the data [5].

2.2.3 Optimization Problems (Operations Research)

Pappis and Mamdani (1977) [11] applied fuzzy logic to control the flow of traffic at an intersection of two one-way streets and minimize traffic obstruction. Teodorovic and Kalic (1996) [9] experimented with fuzzy logic based decisions for choosing the mode of transportation in order to minimize both the cost of traveling and the travel time. Jarkko and Esko (2003) [8] had applied fuzzy logic to minimize the waiting time and risk of collisions during the operation of traffic signals.

2.2.4 Behaviour Control

Perhaps the most intriguing application found in the field of fuzzy logic is the modeling of certain behaviors of systems; most peculiar of them being ethological ones. Of course the topic being the bulk of this work, its details and methods of operation will we further elaborated, but here we examine the interesting possibilities fuzzy logic and fuzzy control can offer us in the form of behavior control. The main difficulty of simulation or prediction of evolution regarding such systems comes from its dependence on a large number of variables and combinations of possible outcomes making it extremely difficult to model with great accuracy. Fuzzy logic allows, in a sense, to approximate these processes and provide a reasonably close solution to the problem. It closely resembles the natural ways of decision making as well, given the fact that there are no sharp boundaries needing to be crossed while considering a decision as it follows, by definition, a continuous range of values.

2.3 Fuzzy Rules

Following [4], the behavior of a system can be represented by a simple model if we consider only its relevant aspects. Such a construct makes use of a set of rules in the "if - then" form. Fuzzy rules are categorized by two major types, Mamdani fuzzy rules and Takagi-Sueno fuzzy rules [13]. In the general form of a fuzzy rule a list of antecedents is followed by a number of consequents such that:

$$\begin{aligned} &\text{if } \mathit{antecedent}_1 \text{ and } \ldots \text{ and } \mathit{antecedent}_n, \\ &\text{then } \mathit{consequent}_1 \text{ and } \ldots \text{ and } \mathit{consequent}_n, \end{aligned}$$

where the *antecedent* is of the form v_1 is S_1 and the consequent z_1 is W_1 respectively. v_i is an input variable belonging to the input fuzzy set S_i and z_j is an output variable

of the output fuzzy set W_j. In the case of Takagi-Sueno fuzzy rules the consequents are replaced with functions of the input variables so that $z_j = f_p(v_1, \ldots, v_i)$, where f_p is any real function.

2.3.1 Fuzzy Inference

In order to use fuzzy logic for any sort of application we must first consider how to integrate it with existing Boolean-logic. More precisely, we are interested in a solution that operates on linguistic variables and an outcome that relies solely on fuzzy rules along with linguistic terms. Since the input variables to any given system are usually not fuzzy ones, they must be converted to satisfy this requirement in order to then later be used within the fuzzy application. This first step is called **fuzzification** and as the name implies we make sure to supply our further calculations with variables of the correct form. By taking the desired element $x \in U$ from our universe of discourse and some fuzzy set A we convert x to a membership function value, given by $\mu_A(x)$. Repeating this procedure for every element we wish to utilize yields a degree of membership for each one, therefore translating all discrete inputs to fuzzy ones.

Now, we have a number of fuzzy variables to work with the next step is to apply predefined rules of the form described in the previous section. This step is referred to as **inference** and mathematically it is a mapping of the antecedents (input variables) to the consequents (output variables), resulting in an output fuzzy set. A degree of membership of any variable in this resultant set is depended upon the degree of membership of values in the input set or sets that have been defined by the given rule. Let A and B denote the fuzzy set of the antecedent dimensions and C of the consequent respectively, and X, Y, Z linguistic terms. Then, according to the rule

$$\text{if } A \text{ is } X \text{ and } B \text{ is } Y \text{ then } C \text{ is } Z$$

the inference process will calculate the output fuzzy set C based on the known values of $\mu_A(X)$ and $\mu_B(Y)$.

In a similar manner to the calculation of the membership functions, there are a multitude of methods which are applicable in determining the output, these are functions that do the actual mapping between the two sets of antecedent and consequent. This process of **fuzzy rule interpolation** (FRI) entails a great number of ways in which different aspects and characteristics of the membership functions are considered and thereafter the calculations made, each having their advantages and difficulties. Numerous techniques are listed in contemporary research with regards to FRI methods [2].

The result, or in other words the consequent, of each rule is obtained and then all these functions are aggregated to produce a final, combined output, which is then finally converted back into crisp, non-fuzzy values in the process of **defuzzification**. The most common form of procedure employed to regain these crisp values is by taking the aforementioned combined function and finding its centroid.

2.4 Fuzzy Automaton and Behavior Control

Behavior is, in the most simple sense, a series of states, where a transition between two states occurs in response to some events. The closest and most accurate mathematical model to this notion are state machines, which follow an almost identical definition.

Integrating fuzzy logic into a state machine will result in a Fuzzy Finite-state Automaton (FFA); this work uses the same model as in [6], being depicted on figure 2.3; where the definition is given in the following manner:

$$F = (S, X\delta, P, O, Y, \sigma, \omega),$$

where

- S is a finite set of fuzzy states, $S = \{\mu_{s1}, \mu_{s2}, \ldots, \mu_{sn}\}$.
- X is a finite dimensional input vector, $X = \{x_1, x_2, \ldots, x_m\}$.
- $P \in S$ is the fuzzy start state of F.
- O is a finite dimensional observation vector, $O = \{o_1, o_2, \ldots, o_p\}$.
- Y is a finite dimensional output vector, $Y = \{y_1, y_2, \ldots, y_l\}$.
- $\delta : S \times X \to S$ is the fuzzy state-transition function which is used to map the current fuzzy state to the next fuzzy state based on the input value.
- $\sigma : O \to X$ is the input function which is used to map the observation to the input value.
- $\omega : S \times X \times Y$ is the output function which is used to map the fuzzy state and input to the output value.

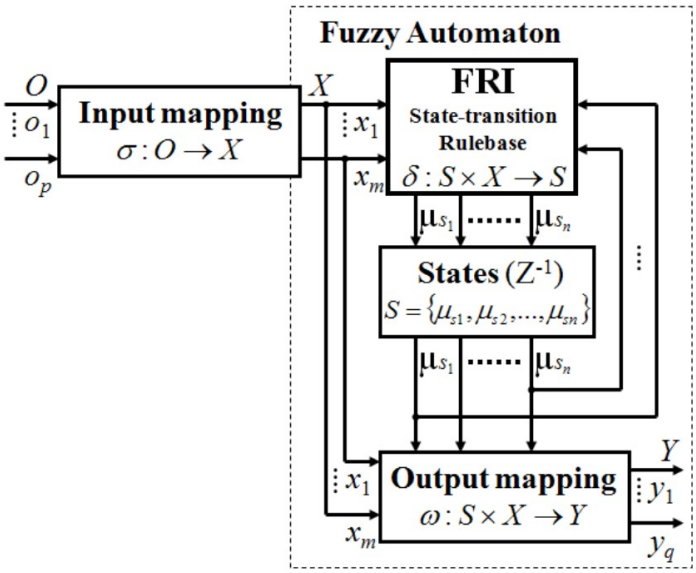


Figure 2.3: Fuzzy Automaton

2.5 Fuzzy Behavior Description Language

After having discussed in detail the needed prerequisites for greater understanding we can finally turn our attention to the main topic of this work, namely the language with which the aforementioned fuzzy behavior can be described.

2.5.1 Motivation

The language aims to provide an environment that enables the creation of programs utilizing fuzzy logic while requiring minimal to almost no prior knowledge in the field of programming. The target applications mainly entail those that delve into behavior control, as its name implies, and in order to facilitate the efficient development of such programs a higher level of abstraction is required leaving the specific implementations with regards to hardware constraints and fuzzy calculations to their respective layers of operation.

2.5.2 Language Specifications

An easy to use language encompasses not just logical abstraction, but is also simple in terms of syntax. An SQL-like syntax for verbosity and the lack of special characters or any complicated notation schemes makes it appeal to a wider audience than it would otherwise. The specifications for the grammar is provided below using the extended Backus-Naur form. Note that the interpreter discussed in this work is based on a JavaScript implementation from the same paper [6].

⟨*behavior*⟩ ::= universe+ rulebase+ [init]

⟨*universe*⟩ ::= '`universe`' string ['`description`' string] symbol+ '`end`'

⟨*symbol*⟩ ::= string number number

⟨*rulebase*⟩ ::= '`rulebase`' string ['`description`' string] rules '`end`'

⟨*rules*⟩ ::= rule+

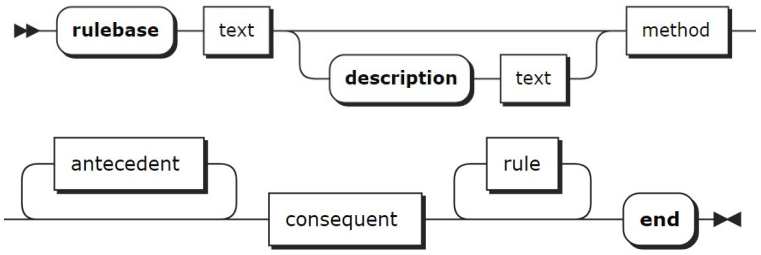
⟨*rule*⟩ ::= '`rule`' ['`description`' string] ['`use`'] string ['`when`' predicates] '`end`'

⟨*predicates*⟩ ::= predicate ('`and`' predicate)*

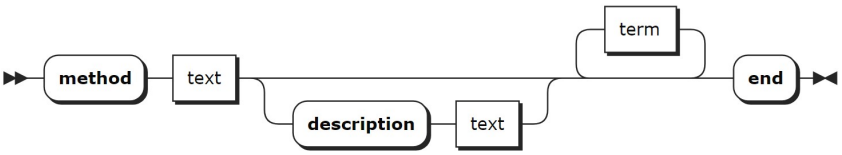
⟨*predicate*⟩ ::= string '`is`' string

⟨*init*⟩ ::= '`init`' ['`description`' string] (string (string | number))+ '`end`'

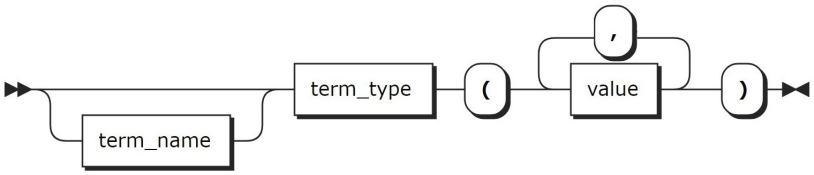
As a graphical representation, a railroad diagram of each element is also included [6].



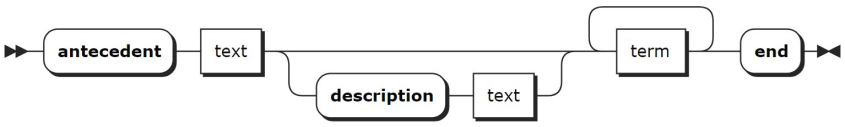
(a) Rulebase definition



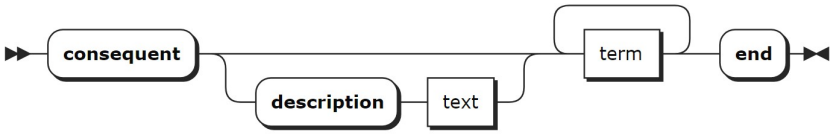
(b) Method definition



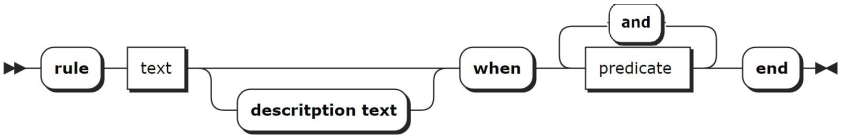
(c) Term definition



(d) Antecedent definition



(e) Consequent definition



(f) Rule definition

Figure 2.4: Syntax diagram of language elements

Chapter 3

Design

Now, after having discussed the necessary concepts and ideas required to fully understand the mathematical model of the interpreter we can finally start inspecting its design requirements, what aspects must be taken into consideration, the language of choice, compatibility related questions etc. Regarding the task of interpretation, these types of programs conform to a specific pipeline that largely determines their architecture. With that said there is still leeway for flexibility and that allows us to adept and choose adequate data structures along with various techniques and algorithms for controlling data flow.

3.1 Requirements and Technical Considerations

Due to past research around fuzzy logic and its application in behavior control the need for having an adequate language along with a library or interpreter to make calculations easier also arose. For avoiding ambiguity, in the following discussions the terms library, program and interpreter are used interchangeably and should be treated as referring to the same concept, namely the FBDL interpreter. The entire operation of the program is based on the preceding works and research in fuzzy behavior control and behavior-based systems [7].

Many of the applied examples were produced in the MATLAB language, therefore facilitating the need for developing a framework capable of operation in that environment. The similarity between the MATLAB and Octave programming languages presented a great opportunity; implementing the interpreter in such a way as to conform to both languages would make it more accessible and allow for wider usage. Therefore the decision was made to produce a program that uses in its operation only minimal parts that are found in both languages, thus making it compatible and interoperable.

To further elaborate, the usage of language specific components and features should be kept at a minimum or avoided altogether if possible, for example the definition of classes or application of several built in functions, function definitions, unit tests etc. are such areas where the two target languages tend to differ quite majorly. It also does not help that at each iteration and new versions these differences grow ever larger since new features are added and perhaps old ones modified or removed, hence implementation with only basic and fundamental features of each language should be prioritized to the utmost extent.

3.2 Structure of The Interpreter

The difference in function definitions, the most noticeable one being that in MATLAB scripts cannot contain any local function definitions before version R2016b, and also other incompatibilities related to the syntax of this operation are the main reason that the program is organized in a way so that every function definition occupies its own *.m* file.

Since the interpreter is not a standalone program, rather a library, its entry point is a function that gets called from the user of this library. From there it goes through each stage of interpreting the input; it is comprised of 3 major parts and a lot of smaller, auxiliary functions that help in completing the task, along with making the program code more readable and modular as depicted on figure 3.1.

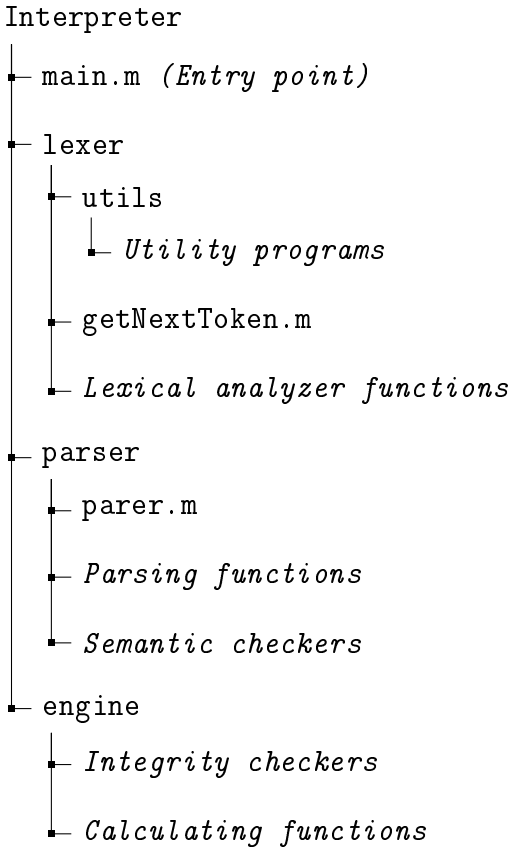


Figure 3.1: Interpreter file hierarchy

The program's entry point is a callable function that takes either a string or a file path as input and returns the solution vectors resulting from the fuzzy calculations. In both cases the content supplied must be a valid piece code written in FBDL.

```
function engine = simulator(input, type)
  retval = 0;
  addpath("lexer");
  addpath("lexer/utils");
  addpath("parser");
  addpath("engine");
```

```
  content =  "";
  if strcmp(type, "f")
    content = fileread(input);
  elseif strcmp(type, "s")
    content = input;
  else
    print_usage();
  end

  behavior = parser(content);
  engine = createEngine(behavior);
  engine = init(engine);
end
```

Functions that facilitate the usage of the program must first be "included" with path definitions treating the above entry point as root, since they are stored in their separate files. Supplying the function with either type of valid input will initiate its operation, anything other than that will result in an error and the user will be provided a message on basic usage.

The function *parser*() being called first might be a bit counter intuitive, but the lexer and parser operate simultaneously, not in a procedural manner, where the output of the lexer is the input to the parser. Of course doing it that way is also possible, returning a vector of tokens and then parsing that, however it is not just less efficient, but at the same time takes away the ability to report errors with correct positional messages, since that information is carried by the lexer and not the tokens.

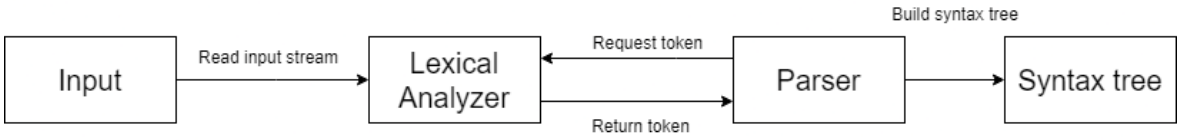


Figure 3.2: Pipeline up to the parser

As seen on figure 3.2, the parser continuously requests tokens from the lexer and at the same time it builds the syntax tree for the entire program, after which this completed structure is passed to the engine for calculations.

3.3 Data Structures

Regarding the data structures used in the program, the most compatible with both languages were found to be the *struct* and *cell*, hence all complex objects are stored in such a manner. For clarity, the *cell* data type is not used as frequently, but in some parts it is more appropriate than other solutions. The most important of these objects is the *lexer*, but others include *token* and the *syntax tree* along with other minor internal data structures that store and manage the information read during or after lexical analysis.

3.3.1 Lexer

This structure holds the input and several key information about the position of the cursor currently analyzing the text. Due to organizing every function into its own file the lexer can only be either a global structure that gets modified by any given function or it is created locally in the top most function in terms of calling hierarchy and is passed around by every other function that needs either access to the input stream or the metadata stored in it. A separate function for creating a lexer is provided in foresight to unit testing, so as to avoid having to create it in every single test case.

```
function lexer = createLexer(content)
  keywords = {
    "universe", "rulebase", "end", "description", "rule",...
    "when", "and", "is", "init",  "use"
  };

  lexer = struct(
    "content", content,
    "content_len", length(content),
    "cursor", 1,
    "line", 1,
    "beginning_of_line", 1,
    "token_begins", 1
  );

  lexer.keywords = keywords;
end
```

The input stream, or as *content* in the lexer structure, is processed by one character at a time and the *cursor* represents how many characters we have read so far, in other words our current position withing the supplied text. Lines are incremented with each encounter of a $\backslash n$ *(newline)* character. The staring position of the given token is also stored to allow for copying the value and report error messages, indicating the position of the incorrect token.

The more complex structures seen on figures 3.3, 3.4a and 3.4b are defined as composites, where the rectangles with rounded corners indicate some sort of data, either primitive or a structure, and ellipses denote some "methods" these structures access. Though these figures resemble UML diagrams it is important to note that the structures presented are not classes; this is simply a logical view to better understand the overall building blocks of the interpreter.

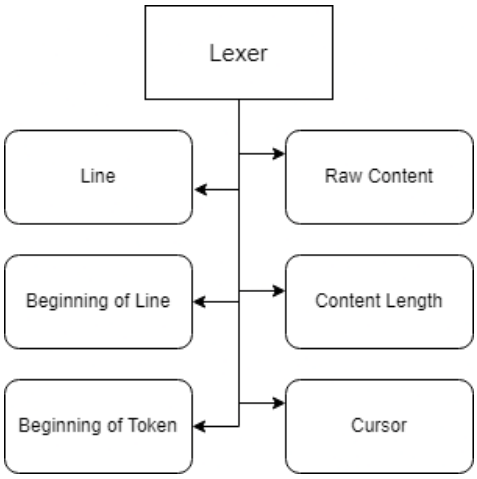
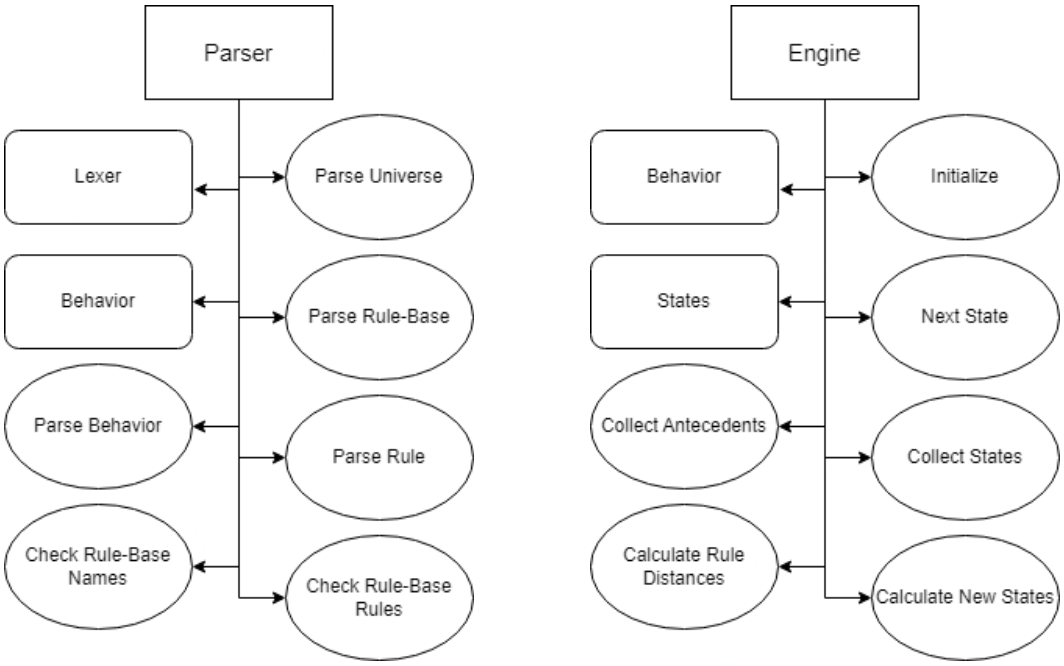


Figure 3.3: Lexer Structure



(a) Parser (b) Engine

Figure 3.4: Parser and Engine Structures

3.3.2 Token

Defined as having the fields: type and value, where the former can be any of the valid token types described by the grammar rules of the FBDL and the latter takes on the actual value read from the input stream. Below are the types accepted and the possible values a token may take:

- `keyword`: *universe, rulebase, init, end, description, rule, when, and, is, use,*
- `string`: *Any character sequence between two " (double quote) symbols,*
- `number`: *Any number, be it decimal or whole with . (dot) for separator in case of the former,*

- `identifier`: **RESERVED** *Any ASCII alphanumeric sequence starting with either a letter or _ (underscore),*
- `terminal`: **RESERVED** *Special symbols such as (), ::, {}, [].*

*Please note that tokens designated as **reserved** are either already in the program code, but not actively used or there are room for them should the need arise for future extension.*

Tokens are "produced" or emitted by the *emitToken* function that constructs a token structure with the correct type and value.

```
function token = emitToken (type, value)
  token = struct(
    "type", type,
    "value", value);
end
```

3.3.3 Syntax Tree

During the process of parsing, every element of the language must be stored for later processing and calculations by the engine. The final structure resembles a tree, hence the name, and mimics the grammar of the language; it is a composite of all the objects defined within the code. Parts that may appear multiple times or none at all are stored in structure/cell fields with their names in order to achieve access similar to that of a hash table; in another case, where the given objects must be iterable they are stored as lists and all these list elements contain the name of the object along with other properties, such as values or further embedded structures. Both the hash table and the list provide sufficient storage and access capabilities, however each have their drawbacks regarding the method by which elements can be reached. The information and data used by the engine for calculations does most of the access and therefore the nature of these structures must be carefully considered beforehand to facilitate ease of use and smooth operations.

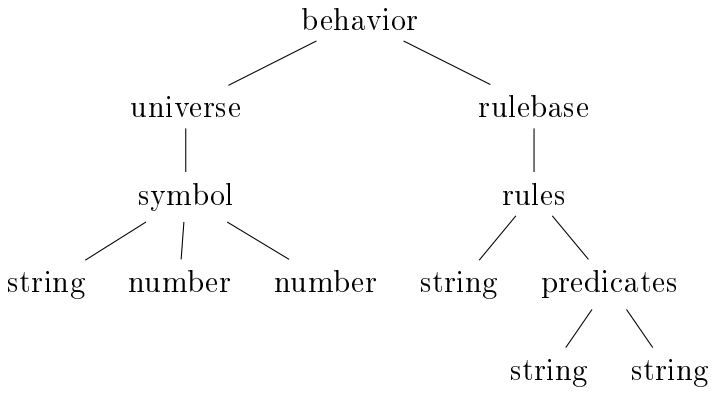


Figure 3.5: Syntax tree

The representation is only logical and not entirely accurate as it would take too much space to represent it in its whole from, but nonetheless it offers another clear depiction of the grammar besides BNF, EBNF and railroad diagrams.

Chapter 4

Implementation

Taking into consideration the language specific elements, we must construct the three main parts, namely the lexer, parser and engine, in such a way to make their operations and interactions smooth. The different algorithms and processes to analyze text, parse tokens and calculate the states is explained via pseudo code and, in some cases, with the help of code snippets from the interpreter.

4.1 Lexical Analyzer

The first stage in the program is the lexical analysis, also called tokenization, a process which dissects and "categorizes" the input based on some predefined rule and extracts a stream of tokens, or in this case the *getNextToken* function returns them one by one at each call. Everything else that is not needed or does not carry meaningful information is discarded such as white spaces, tabs, newline character, or any type of control character and only the allowed tokens are processed.

Upon receiving a call the function must first *trim* the input, simply skip the characters which are of no use to us such as all the control characters and those outside the alphanumeric range. Note that whenever we iterate over the input extensive checks are necessary in order to mitigate any indexing related issues.

```
function lexer = trim (lexer)
  while lexer.cursor <= lexer.content_len &&
  ((lexer.content(lexer.cursor) < 33) ||
  (lexer.content(lexer.cursor) == 127))
    if lexer.content(lexer.cursor) == "\n"
      lexer.line++;
      lexer.beginning_of_line = lexer.cursor;
    end
    lexer.cursor++;
  end
end
```

Then we must determine if we have run into a comment line, marked by a # *(hashtag)* symbol or reached the end of the file. In the case of the former, we treat it in a similar manner to trimming, but only going until a
n (newline character is reached, whilst regarding the latter, a token is emitted with type *EOF* and no value.

Now we can finally start examining whether the stream of characters read from the input are part of an accepted token. Since we read individual characters from the input it is not possible to tell if it is going to be a valid token until we have read the whole word, however just from the first character we can categorize it as a possible token and then hand the procedure for checking to the appropriate function. In a case where an invalid character sequence is encountered the program raises a syntax error with the appropriate message and also displays the line and column numbers where the fault occurred, then exits.

4.1.1 Identifier (reserved)

An identifier may start with an underscore or any letter from the English alphabet found in the ASCII character set; case sensitivity is not considered. Characters apart from the first one can include numbers as well. Since no functionality for handling identifiers are currently implemented in the latest version of the interpreter, and neither in the grammar of the FBDL, only keywords are permitted in the supplied FBDL source code. However given the similarity of lexical analysis in both cases some room has been left for possible accommodation of this feature at a later time.

After a token is read as an identifier its value is compared against the list of available keywords, if a match is found the token type is changed to keyword and returned.

```
...
while lexer.cursor <= lexer.content_len &&
                isIdent(lexer.content(lexer.cursor))
  lexer.cursor++;
end
token.value = substr(lexer.content, lexer.token_begins,
                lexer.cursor - lexer.token_begins);
if any(strcmp(lexer.keywords, token.value))
  token.type = "keyword";
end
...
```

The list of keywords may be extended or have entries removed, granted the modification is permitted by the grammar of the language. This is the case with the *dominates* keyword as hierarchical rule dominance is not implemented in the program, but is found in the grammar as an optional language element.

4.1.2 Terminal (reserved)

Similarly to identifiers it is not yet available in the FBDL grammar, but some consideration has been taken to allow for future extensions that include these elements.

4.1.3 String

Encountering a " (double quote) symbol implies that a string will follow and accordingly every character is skipped until the closing pair is not reached. The lexer holds every token's starting position, in this particular case that happens to coincide with the position of the double quote at the beginning, which will later be used as an index to

copy the contents of the string and store it in a token. This procedure is used in case of every token that has a value.

The length of a given string is unknown when iterating over it, the only way to see if it is invalid, or in other words the closing double quotes are missing, is to see if we have reached the end-of-file beyond which there cannot be any more characters. Empty strings are permitted and no value copying happens in such a case.

Escape characters are not yet allowed in strings and there are no implementations to process them, they are simple copied as literal characters just as the rest of the string.

With extended Backus-Naur form :

⟨*string*⟩ ::= " character* "

4.1.4 Number

The first thing to consider when checking for numerical constants is the presence of the optional negative sign, since it is not itself a numerical value, most often being denoted with a - *(dash)*. The absence of such a sign implicitly implies a positive number, therefore the need for a + *(plus)* sign is eliminated and is not processed. Then a series of numbers, digits, must follow until the end of the token; the very first digit cannot be zero. Every number may contain a single negative sign before the first digit and a single decimal point between two adjacent digits marked by a . *(dot)* character. Failing to meet any of these conditions will result in a syntactical error being raised and the program exiting. Although floating point numbers are accepted, their syntactical version of scientific notation is not; furthermore the language does not support arbitrary-precision arithmetic.

After passing these checks the series of digits is copied from the input text and is converted to a double precision floating point type, which is then returned in the token. This step allows direct referencing of the token's value during calculations in the engine without needing to do the conversion there.

With extended Backus-Naur form:

⟨*number*⟩ ::= [-] (integer | fraction)

⟨*integer*⟩ ::= digit-z+ digit*

⟨*fraction*⟩ ::= (integer | 0) '.' digit*

⟨*digit*⟩ ::= 0 | digit-z

⟨*digit-z*⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.2 Parser

Every language conforms to some form of grammatical rule base from which it cannot deviate, otherwise it would not be valid. The grammar employed by the FBDL has been mentioned before and now the part that is responsible for the correct interpretation of code that uses said grammar will be explained. The interpreter makes use of a technique called *recursive descent top-down parsing* [3][1], where every non-terminal in the grammar is handled by a dedicated function. In this section each rule within the

grammar will be compared against the code of the parser to see its method of operation and also note some problems which often makes the process more complicated than necessary. Since the actual implementation of each function is quite lengthy, it is most adequate to demonstrate only the underlying logic via pseudo code.

The very first rule is straightforward, stating that there must be at least 1 occurrence with regards to the *universe* and *rulebase*. The *init* optional part is not implemented in the program, therefore it will be skipped.

⟨*behavior*⟩ ::= universe+ rulebase+ [init]

⟨*universe*⟩ ::= '`universe`' string ['`description`' string] symbol+ '`end`'

⟨*symbol*⟩ ::= string number number

Every language construct besides terminals and the first rule is enclosed between two keywords, namely the one indicating what object it is defining and an *end* keyword. The *universe* requires a name of type string and a series of symbols, which will define the variables and their range of values for later usage. Despite an optional description being available in the grammar, this and all such options in the following rules are yet to be implemented and therefore no algorithm is given for the procedure.

```
Result: Universe:
if token is literal then
    universe.name = token.value;
    check optional description;
    next token;
    while token is literal do
        symbol.name = token.value;
        next token;
        if token is number then
            symbol.position = token.value;
        end
        next token;
        if token is number then
            symbol.value = token.value;
        end
        next token;
    end
    if keyword is 'end' then
        universe += symbol;
        return universe;
    end
end
```

Algorithm 1: Parsing the *universe*

Implementation and language specific elements have been eliminated or reduced as much as possible for clarity in figure 1, furthermore all checks and error messages have been omitted in order to provide the minimal code required while preserving its semantic integrity. Similar simplifications will be used for all demonstrations to keep the focus on understanding the algorithm rather than being lost in obscure syntax.

With that said the next rule is concerned with the *rulebase*, which is almost identical to the *universe* only differing in the containment of a series *rule* elements as opposed to *symbols*.

⟨*rulebase*⟩ ::= '`rulebase`' string ['`description`' string] rules '`end`'

⟨*rules*⟩ ::= rule+

However, these rules require an algorithm much more complicated than in the case of the previous grammar rules.

⟨*rule*⟩ ::= '`rule`' ['`description`' string] ['`use`'] string ['`when`' predicates] '`end`'

⟨*predicates*⟩ ::= predicate ('`and`' predicate)*

⟨*predicate*⟩ ::= string '`is`' string

Once again, as observed on figure **??**, discarding the first optional argument, the *use* keyword references the value of a previously calculated variable and is further used in calculating behavior fusion resulting from all other behavior components, i.e. rule-bases.

Result: Rule:
if *token is literal or keyword* **then**
  read optional arguments;
  next token;
  if *token is 'when' keyword* **then**
    read predicates;
  end
else if *token is not 'end' keyword* **then**
  error;
else
  error;
end

Algorithm 2: Parsing a *rule*

These two algorithms occupy the same function, but to avoid any ambiguity they have been presented as separate. In the case of every grammar rule the checking of enclosing keywords is paramount along with other type checks as well. Encountering a problem, as denoted by *error*, at any point in these functions will result in the program exiting and returning an appropriate error message.

Result: Predicates:
while *token is not 'end'* **do**
  next token;
  if *token is literal* **then**
    antecedent = token.value;
  end
  next token;
  if *token is not 'is' keyword* **then**
    error;
  end
  next token;
  if *token is literal* **then**
    value = token.value;
  end
  next token;
  if *antecedent already in predicates* **then**
    error;
  end
  next token;
  if *token is neither 'and' nor 'end' keyword* **then**
    error;
  end
end

Algorithm 3: Parsing the *predicates*

The series of predicates is processed in the same manner is the *symbols* from before, albeit with a longer algorithm, seen in figure 3. Furthermore in case any antecedent is already present in the list of predicates, in other words, it is a duplicate predicate, an error is raised.

After the parsing of the input stream is completed additional semantic checks are required to ensure that the variables used within rules conform to previous definitions regarding names and values; the same applies to the rules themselves, including the predicates.

```
function checkRulebaseNames(behavior)
  len = length(behavior.rulebases);
  if len == 0
    error("Parse error! At least one rulebase must exist!\n")
  end

  for i = 1:len
    if !isfield(behavior.universes, behavior.rulebases(i).name)
      error("Parse error! Missing universe definition for rulebase\n");
    end
  end
end
```

If any part of a given rule, regardless of being a consequent or an antecedent, is not presented in the universe, then it is reported as an error, since it has not been

previously defined.

```
function checkRulebaseRules(behavior)
  for i = 1:length(behavior.rulebases)
    rulebase = behavior.rulebases(i);
    for j = 1:length(rulebase.rules)
      rule = rulebase.rules(j);
      checkRule(rulebase.name, rule, behavior.universes);
    end
  end
end
```

4.3 Engine

At the last stage of the operation is a direct implementation of the previously mentioned Fuzzy Automaton mathematical model, that performs all the necessary calculations required for simulating fuzzy behaviour. The entire system is a state machine, meaning that the initial values of the variables inside the universe will be used in the first and then subsequent transitions, and are subject to change as the automaton moves from one state to another.

Similarly to the lexical analysis stage an *engine* object is created at the beginning, which holds the syntax tree and also contains additional information such as the state of each variable, distance values and other elements as well. This structure is passed around by functions that use these data to calculate various intermediate variables for later determining the state transition of each variable. For easier referencing during processing the states and collection of antecedents are gathered and stored in a list. In the next step, the integrity of these initial variables are checked and they must conform to the following conditions:

- All universes listed must be defined.
- The symbol values of a given universe must be bounded by the universe.
- All input values must be defined.

As these are initial values the user of the interpreter can modify any of their parameters, name the values of variables, to fit their need or simply experiment.

Next, we must evaluate the consequents, but before that, it is required to calculate the distance of observations from the symbols on every antecedent universe. As listed by the example from [7], considering the following universes and initial values defined:

```
universe "distance"
  "zero" 0 0
  "close" 1 0.1
  "far" 5 1
  "max" 10 1
end
```

```
universe "curiosity"
  "low" 0 0
  "high" 1 1
end
```

```
universe "speed"
  "low" 0 0
  "high" 1 100
end
```

```
init
  "distance" 3
  "curiosity" 0.4
end
```

In the distance universe example above the consequent value lies on the $[0, 1]$ interval. Distance values above 5 are treated the same; anything larger up to 10 is considered "far". We also define a set of rules:

```
rulebase "speed"
  rule "high" when "distance" is "far" and "curiosity" is "high" end
  rule "low" when "distance" is "close" end
  rule "low" when "curiosity" is "low" end
end
```

Considering the first rule, we must calculate the distance of 3 from "far" in the "distance" universe. The distance is determined from the difference of the cumulative scaling function values of the two points. That is, the value of the cumulative scaling function at 3 is 0.55 and at "far" it takes the value 1, therefore their cumulative scaled distance is 0.45. Similarly we can calculate the distance of 0.4 and "high" on the "curiosity" interval resulting in the value 6. The distance of the rule is given by the Euclidean norm of the distances by dimensions divided by the square of the number of antecedents.

The rule distance for the first rule can be calculated in the following manner:

$$d_{rule1} = \frac{\sqrt{0.45^2 + 0.6^2}}{\sqrt{2}} \approx 0.5303.$$

The other two rules are computed in the same way:

$$d_{rule2} = \frac{\sqrt{0.45^2}}{\sqrt{2}} \approx 0.3182, d_{rule3} = \frac{\sqrt{0.4^2}}{\sqrt{2}} \approx 0.2828.$$

If the distance from some rules is 0, then the consequent is calculated as the *mean value* of the corresponding consequent symbol values, or variable values. The weights are the reciprocals of the distances raised to the Shepard-power p, so that $w_i = \frac{1}{d_i^p}$.
If $p = 1$, then the weights of the rules are respectively:

$$w = 1.8856, 3.1427, 3.5355.$$

The consequent values of the rules are:

$$c = 1, 0, 0.$$

The consequence can be obtained as the sum of weighted rule consequents, using the rule weights:

$$C = \frac{\sum_i w_i c_i}{\sum_i w_i} \approx \frac{1.8856}{8.5638} \approx 0.2202.$$

The consequent value of the “speed” rule-base is approximately 22.0183.

Seen on figure 4.1 is a model of the interactions between universes, rules and consequents.

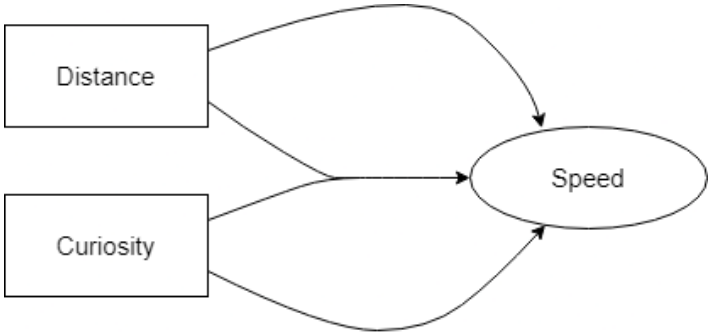


Figure 4.1: Evaluation of rule-bases

4.4 Error Handling

Upon encountering an error, the program should terminate its operations immediately, report the fault with informative messages and then exit gracefully. A “token" based method was considered before to check the integrity of operations; in response to catching an error a special token would be emitted and since the top-most function is the one receiving these tokens, specifically the parser, it should be the one to terminate the program. However the probability of errors occurring is numerous and creating tokens at every one of these places is neither a space efficient nor a logically sound approach, leading to much confusion and clutter. The preferred method chosen was using the *error(msg)* function both found in Octave and MATLAB, where a message *msg* would also be displayed to the user. Calling the function directly is not appropriate, since not only the fault needs to be reported, but its position as well. Furthermore to facilitate the process of locating the problematic section of code resulting in the error, the whole line where the program failed is printed to the screen.

```
function raiseError (lexer, type, msg)
  while (lexer.cursor <= lexer.content_len &&
       lexer.content(lexer.cursor) != "\n")
    lexer.cursor++;
  end

  snippet = substr(lexer.content,
```

```
  lexer.beginning_of_line, lexer.cursor - lexer.beginning_of_line);
  error("%s! At line %d, column %d!\n%s\n%s\n",
    type, lexer.line, lexer.cursor - lexer.beginning_of_line,
    msg, snippet);
end
```

4.5 Future Extensions

With time programming languages usually evolve, and the FBDL is no exception, therefore, it is quite sensible to employ an architecture that is capable of adapting to changes in code and also leaves room for extensions. Various elements in the language such as strings, numbers, terminals and keywords or even grammar are susceptible to change. Regarding the first in the list, strings might contain escape characters and as such the *lexer* must store a list of characters that are accepted as valid escape sequences and during the parsing of strings correctly map them to the given control character. Number definitions could also include scientific notation or perhaps even complex numbers, although the benefits of the latter are not obvious to me. Implementing functions would require the modification of the grammar and also introducing new elements to parsing, namely identifiers and terminals, hence the main reason they are considered reserved.

In the case of the parser, the processing of optional arguments should be implemented at some point in the future along with warning messages, when they are missing. In the original paper [6], a hierarchical implementation of rules is also defined with the help of the *dominates* keyword, which also ought to be interpreted correctly to work with multiple, highly interconnected rules. With regards to numbers once more, currently only numerical constants are accepted as valid values, but should the need arise for them to be replaced with mathematical expressions, then new parsing techniques would also be required, namely precedence climbing, in order to handle these cases.

More advanced techniques may be integrated into the engine to process these rules more efficiently. Since each rules if independent of one another this presents a massively parallel environment, which is beneficial during the scaling of the model. Other types of interpolation methods could be employed and examined to see if the produce different results.

Although these possible extensions mentioned might evoke optimism for broadening the language, a fundamental property would be lost in the process, that being its simplicity. Therefore one should carefully consider before any significant alteration to the language what is to be gained with these additions of “tools" and what is to be lost.

Chapter 5

Testing

In order to make sure that all the modules and different parts of the interpreter function correctly the program must be subject to thorough testing. Evaluating the critical components and ensuring that they operate within the defined bounds; and even if they were to fail, each fatal case should be handled properly and produce correct, informative error messages.

5.1 Unit Tests

At the end of each major function of the interpreter, namely the lexical analyzer, parser and engine there are a series of tests, which examine specific edge and corner cases. These are not yet complete and should be extended to include more obscure input patterns; the engine is particularly tricky to examine, since its operation is highly integrated with other functions and also produces many different outputs, whose integrity is difficult to verify.

A minor obstacle in testing the code base in a bilingual environment is that native unit tests vary, making uniform integrity checks impossible. In the case of Octave every such test is located at the bottom of each function; as an example this part is from the lexer and is a case of correct negative number usage:

```
%!test
%! content = "-4.5";
%! [lex, token] = getNextToken(createLexer(content));
%! assert(token.type, "number");
%! assert(token.value, "-4.5");
```

Every Octave unit test comes after a %! pair and can either be a test block or a single line command such as:

```
%!error <incorrect use of decimal point>getNextToken(createLexer("4."));
%!error <incorrect use of decimal point>getNextToken(createLexer("4.a"));
%!error <unrecognized character>getNextToken(createLexer(".4"));
%!error <non-numeric>getNextToken(createLexer("--4"));
%!error <non-numeric>getNextToken(createLexer("-b"));
%!error <non-numeric>getNextToken(createLexer("-.4"));
```

Where each of these inputs specified at the end will result in an error, which is then successfully caught and handled by the function.

Similarly, in MATLAB we can also define test function at the bottom those we need to test, however the incompatibility arises from the different syntax of these unit tests. Fortunately, MATLAB supports unit tests that are written in separate script files and then run via a command. Although storing multiple copies of test cases is redundant, at the same time it allows us to run the same unit tests on the same code base by using two different languages. Overall, tests for an Octave environment are located the the bottom of each function, and test provided for a MATALB environment are in their own files.

5.2 Performance

In both MATLAB and Octave the ability to pass variables by reference is missing, this feature being available only to functions, therefore they can only be passed by value, meaning that each time a function call is made the inputs are copied for use inside the function. This is the main reason for passing around data structures in the interpreter; it's the only way to modify them without making them global or without the use of classes. However, this comes at the cost of efficiency, since values and entire complicated data structures are constantly being copied, they slow down the program significantly and consuming more memory as well.

Some measurement data gives a reference to the performance of the interpreter. The FBDL code used in the test is located in the root directory as *test.txt*. Reading the complete input data, about 100 lines of code with 9 universes, 5 rule bases and a total of 15 rules, parsing it and then initializing the engine takes about *0.5473s* on average. Each advancement of the state machine, with its list of calculations, is performed in about *0.01435s*. Significantly higher execution speeds could be achieved with the help of parallel computing, implementing it on the parsing and evaluation of rules.

5.3 Usage

The user first has to call the *simulator* function and provide the source alongside a character indicating what type of source is supplied. This can be either a string and *s* or a file path and *f*, in both cases the content must hold valid FBDL code. For example, when using code from a file:

```
e = simulator('test.txt', 'f');
```

An engine structure is returned, that holds the parsed code and the current states of the fuzzy state machine, which, in this case, is simply the initial state. Displaying the current states can be done by either directly accessing them or using the *getStates* function that returns all states as a vector:

```
e.states % as fields
e.states.speed % 'speed' state
getStates(e) % as a vector
```

For altering these states use the *setStates* function:

```
states = [1, 2, 3, 4];
e = setStates(e, states);
```

Where the engine and a vector of states, containing the new ones, is passed. In case the dimensions of the actual states and the ones provided does not match, an error is produced.

Advancing the fuzzy state machine is done via the *step* function:

```
e = step(e);
```

Calculating the next state takes into account the previously parsed rules and also the current states. In case the original state of the system is required the *resetStates* function should be used.

```
e = resetStates(e);
```

5.4 Rule Surfaces

From the previous example in chapter 4, section 4.3, we can visualize the relationship between rules by displaying the rule base "speed" as a function of its two antecedents as seen in 5.1.

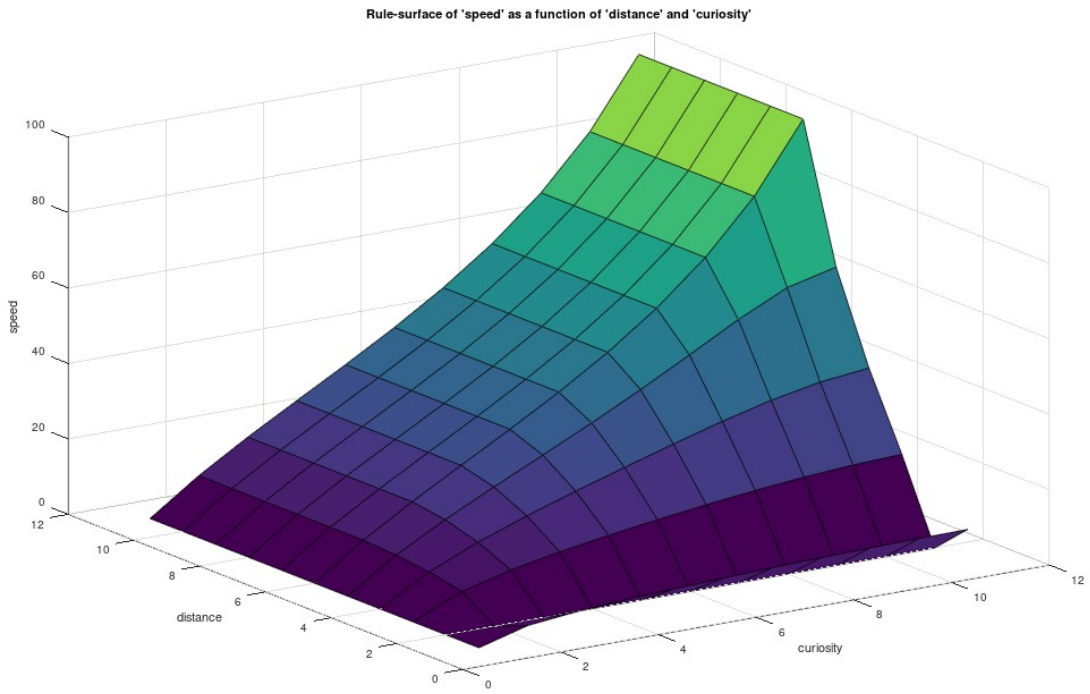


Figure 5.1: Rule Surface

Examining these rules separately we see how the "speed" value changes: figure 5.2 in case of curiosity and figure 5.3 for distance.

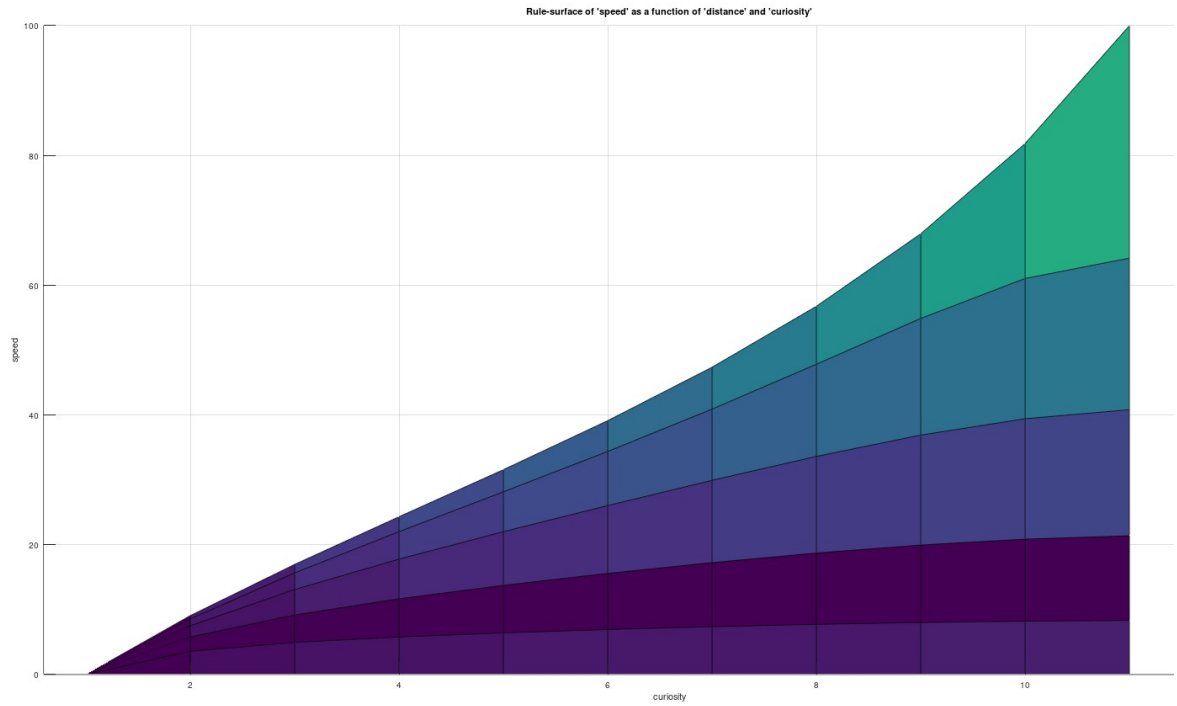


Figure 5.2: Curiosity

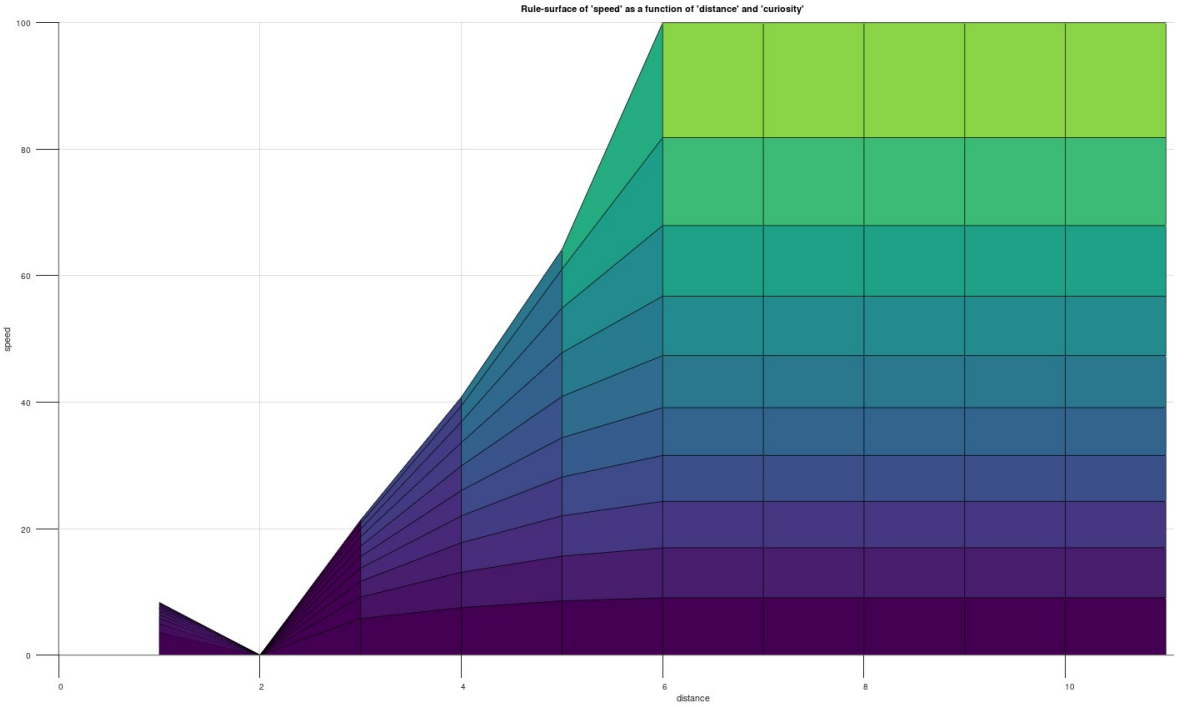


Figure 5.3: Distance

Chapter 6

Summary

The interpreter has been written and implemented in Octave and is yet to be tested in the MATLAB environment, but since it has been designed with consideration for both languages, at most, only a minimal or at best, no amount of modification should be preformed. The complete source code is present on the physical copy provided alongside this work available and also available on github, which can be downloaded from https://github.com/baliking01/FBDL_Interpreter.

With the functions provided by the interpreter, the user can manipulate the fuzzy state machine and change or analyze its current state. In subsequent version and iteration of the interpreter, more functions should be added that further facilitate this process, Furthermore, it is evident that passing around the engine is very cumbersome and classes could offer significant improvement over the current implementation, however this way it remains compatible with various versions of both languages. Despite it being a usable solution, more user friendly approaches and implementations should be developed in the future along with a focus on performance and memory constraints.

A major advantage offered by these languages is the many scientific functions they provide, which can be used to interact with the fuzzy state machine; displaying rule surfaces for example. Since the original purpose of creating the language was to allow easy programming of ethological simulation models, complicated calculations and data structures are not necessary due to the abstractions that both MATLAB and Octave offer, thus the user can fully focus on the FBDL source code.

Although the interpreter described here is fully functional, there are still many things left to improve and some even to be extended. With constantly ongoing research in the field of fuzzy logic, I hope, that this piece of software can be of use and aid those looking for practical applications by providing an adequate working environment for new discoveries.

Bibliography

[1] Jeffrey Ullman Alfred V. Aho, Ravi Sethi. *Compilers: Principles, Techniques and Tools (First Edition)*. 1986.

[2] Szilveszter Kovács Alzubi M., Johanyák Zsolt Csaba. Fuzzy rule interpolation methods and fri toolbox. *Journal of Theoretical and Applied Information Technology*, 96(21):–, 2018.

[3] William H. Burge. *Recursive programming techniques*. 1975.

[4] Sabri N. et al. Fuzzy inference system: Short review and design. *International Review of Automatic Control*, 6(4):441–449, 2013.

[5] Valerie Davidson Gordon Hayward. Fuzzy logic applications. *Analyst*, 128:1304–1306, 2003.

[6] Dávid Vincze Imre Piller, Szilveszter Kovács. Declarative language for behaviour description. *Acta Polytechnica Hungarica*, 316:103–112, 2015.

[7] Szilveszter Kovacs Imre Piller. Fuzzy behavior description language: A declarative language for interpolative behavior modeling. *Acta Polytechnica Hungarica*, 16:47–72, 2019.

[8] Esko Turunen Jarkko Niittymaki. Traffic signal control on total fuzzy similarity based reasoning. *International Journal of Soft Computing*, 133:109–131, 2003.

[9] Dusan Teodorovic Milica Kalic. Solving the trip distribution problem by fuzzy rules generated by learning from examples. *Proceedings of the XXIII Yugoslav Symposium on Operations Research*, pages 777–780, 1996.

[10] Claudio Moraga. Introduction to fuzzy logic. *Facta Universitatis. Series: Electronics and Energetics*, (18):319–328, 2005.

[11] Mamdani E. Pappis C. A fuzzy logic controller for a traffic junction. *IEEE Transactions on Systems, Man and Cybernetics*, 7(10):707–717, 1977.

[12] Makkar R. Application of fuzzy logic: A literature review. *International Journal of Statistics and Applied Mathematics*, 3(1):357–359, 2018.

[13] Sugeno T. Takagi, M. Fuzzy identification of system and its applications to modeling and control. *IEEE Transaction on Systems, Man and Cybernetics*, 15:116–132, 1985.

[14] Lotfi Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).