Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

Programa de Pós-Graduação em Engenharia Elétrica

The implementation of a theorem prover in functional language and the use of the Rasch Model combined with Condorcet-List Theorem

Junia Magalhães Rocha

Setembro, 2015

Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

The implementation of a theorem prover in functional language and the use of the Rasch Model combined with Condorcet-List Theorem

Tese apresentada por Junia Magalhães Rocha à Universidade Federal de Uberlândia (UFU) como parte dos requisitos para obtenção do título de doutor.

Banca Examinadora:

Luciano Vieira Lima, Dr. UFU. Orientador (UFU)

Marcelo Rodrigues Sousa, Dr. (UFU)

Antônio Eduardo Costa Pereira, Dr. (UFU)

José Lopes de Siqueira Neto, Dr. (UFMG)

Reny Cury Filho, Dr. (PMU)

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

R672i
2015

Rocha, Júnia Magalhães, 1985-
The implementation of a theorem prover in functional language and the use of the Rasch Model combined with Condorcet-List Theorem / Júnia Magalhães Rocha. - 2015.
122 f. : il.

Orientador: Luciano Vieira Lima.
Tese (doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.

1. Engenharia Elétrica - Teses. 2. Inteligência artificial - Teses. 3. Processamento de linguagem natural (Computação) - Teses. 4. Prolog (Linguagem de programação de computador) - Teses. I. Lima, Luciano Vieira, 1960-. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 621.3

The implementation of a theorem prover in functional language and the use of the Rasch Model combined with Condorcet-List Theorem

Junia Magalhães Rocha

Tese apresentada por Junia Magalhães Rocha à Universidade Federal de Uberlândia (UFU) como parte dos requisitos para obtenção do título de doutor.

Luciano Vieira Lima, Dr.	Alexandre Cardoso, Dr
Orientador	Coordenador do
	Curso de Pós-Graduação

Tudo é do Pai

Dedico esse trabalho aos meus pais, José Antônio e Maria das Graças, a Vovó Nini, a minha irmã Juliane, ao meu esposo Fábio e a minha filha que está a caminho, Ísis.

Agradecimentos

Em primeiro lugar agradeço a Deus pela força que me concedeu para a realização desse doutorado. Aos meus pais, José Antônio e Maria das Graças, minha irmã Juliane, minha avó Nini e ao meu esposo Fábio pelo apoio incondicional. Agradeço a minha família pela compreensão nos momentos que estive ausente.

Aos professores Luciano Lima e Eduardo Costa pelo gigantesco conhecimento ao longo desses anos de Mestrado e Doutorado.

Aos meus amigos Rubens Barbosa e Will Roger que tanto me apoiaram nos trabalho do laboratório.

Agradeço aos membros da banca pelas contribuições para melhoria desse trabalho.

Aos funcionários da Universidade Federal de Uberlândia e ao Departamento de Engenharia Elétrica, em especial a Sra. Cinara Matos, secretária da pós-graduação pelo auxílio, esclarecimentos e presteza.

Contents

List of Figures vi

1 Introduction **2**

1.1 Objective 5

1.2 Work Layout 5

2 Condorcet-List Theorem and Rasch Method **7**

2.1 The Jury Theorem 7

2.2 Rasch Model 10

2.3 Scales 10

2.4 Specific objectivity 12

2.5 Origin 13

2.6 Rasch Logistic Distribution 15

2.7 Calibration Algorithm 18

3 Lisp and Mathematics **24**

3.1 Tool for natural language processing 27

3.2 Emacs 28

3.3 Quicklisp 30

3.4 Emacs commands 34

3.5 Arithmetic operations 35

3.6 The let-form 37

3.7 Lists 38

3.8 Packages 41

3.9 The cond-form 42

3.10 format . . . 43
3.11 Loops . . . 44

4 An space efficient implementation of WAM **46**
4.1 Concepts . . . 52
4.2 Enunciate the thesis . . . 52
4.3 Performing Tests . . . 53
4.4 Warren Abstract Machine - WAM . . . 55
4.4.1 Registers . . . 56
4.4.2 Unification . . . 58
4.4.3 The Cut . . . 59
4.4.4 Deterministic Predicates . . . 59
4.4.5 Infix Syntax . . . 60
4.4.6 Consult function . . . 63
4.4.7 Generated Lisp code . . . 63
4.5 Tests performed with WAM . . . 64

5 Future Work **78**
5.1 Web pollution . . . 78
5.1.1 Text mining . . . 80

6 Conclusion **82**

7 Appendix I - WAM **85**

List of Figures

2.1 Condorcet's Jury Probabilities 8
2.2 Hit probability of the jury 9
2.3 C x V 11
2.4 Success and failure probabilities for problem R 11
2.5 Specific Objectivity 12
2.6 Origin of the Ability Scale 16
2.7 Origin of the Difficulty Scale 17

Abstract

This thesis proposes the implementation of a theorem prover using a functional programming language. The implementation was based on the Warren Abstract Machine (WAM). The objective behind implementing the WAM is to achieve robustness and while running remain constant in both time and space. The inference engine was implemented using Common Lisp, due to its mathematical roots. The Theorem prover syntax approaches that of the language Prolog. Besides the theorem prover, it was demonstrated that the Rasch Model and the Condorcet-List can be brought together and used for the processing of natural language.

Keyword

Natural Language Processing, Warren Abstract Machine, Rasch Model, Condorcet-List Theorem

Chapter 1

Introduction

Natural language processing (nlp) is a complex task, due to the fact that it requires analyzing underlying relationships, grammatical rules, meanings, logic and shared experiences.

One finds multiple meanings frequently among individual words and sentences. Besides this, one can express a concept in many different forms. Therefore, handling the ambiguity that arises from these two aspects of language poses a significant challenge to linguists.

It is a well known fact that people resolve ambiguity through the surroundig text, knowledge of the world and shared experience.

Surrounding text: Compared to other issues involving natural language processing, it is relatively simple task to analyse the surrounding text automatically.

Knowledge about the world: Procedural semantics can gather knowledge about the world.

Shared experience: The implementation of surrounding text analysis, and of a system that gathers knowledge about the world, leaves the question: How can a machine share experience with a human being? Before trying to answer such a question, let us see what procedural semantics pertain.

Procedural semanticists agree that the meaning is equal to a program. A person understands another if he/she acts in accordance with the other. This view is bound to language understanding, since a computer can translate a program into actions: it can operate a robot in the execution of physical work, change the internal states in memory, store information, etc.

Procedural semantics contrasts with logical semantics, where it is assumed that facts are available in a database. In this case, the listener must match queries against this database. Critics of logic semantics claim that there is a very large amount of information that a speaker of a language must gather about the world and so the data base management system will never reach a complete set of facts. For instance, to speak at the level of a four year old child, a computer needs to know a lot of things about water alone: it freezes, it takes the shape of a glass, it boils, one can use it to take a bath, etc. People do not consult a database to obtain information about water. It is retrieved directly from observations about the world, and stored persistently in human memory. Therefore, the key is data persistence.

Instead of gathering facts about the world, procedural semantics actively constraints which facts are collected from the environment or from an interlocutor.

In order to create a model of the world and of its interaction with human beings and other machines, the computer must build a model of various objects. Of course, object oriented programming can be the basis for building a representation of the world.

Objects have attributes, like color, size, position, weight, integrity, etc. Attributes have values. For instance, colors can be white, blue, red, etc. A set of values for the attributes determines a state for an object.

Of course, objects belong to classes: birds, cities, stars, chemical elements, furniture, employers, etc. There are actions that can be performed by a system of classes. For instance, to kill is an action that can be realized by almost any kind of object. However, the consequence of killing can fall only upon animals, microorganisms or plants. Fighting requires two animals. Any act that one can perform on a set of objects is called a method. Choosing a method that will be applied to a set of objects is called dispatch.

Of course there are many methods that the computer can apply to a given set of objects. The computer can choose one of these methods or more than one. When the computer applies methods one after the other, each method adds constraints to the solution. In the end, it is necessary to put together the result of the many applicable methods. This operation is called method combination, and builds an effective method.

One arrives at the effective method by carrying out three steps: Initially, a generic function builds a list of applicable methods based on the class of arguments it was given.

Then, the computer sorts the list of applicable methods by placing the more specific method in front of the more general method. Finally, the machine takes the methods in the same order as found in the sorted list, and combines their code to produce the effective method. Of course there are many strategies for combining methods.

One cannot leave out the point that many computer languages that claim to be object oriented are in fact not. They fail in the first feature of object orientation: they do not have multiple dispatch. In consequence, there is no automatic method combination. The programmer must hardwire the effective method inside each class. This makes implementing procedural semantics very difficult towards the impossible.

It is useless to have shared an experience with somebody else if one does not remember it. People build a common context with other people because they remember facts, events, and actions. They remember solutions to problems and the procedures for arriving at those solutions. They learn and store what they have learned as skills and behaviors. Two words sums up the building of shared experience: persistence and memoization.

Persistence refers to a state that outlives the process that created it. Without this capability, the state is lost at a computer shutdown. Let us consider a computer C that is helping a child K to write a prose composition. The computer has many methods for detecting errors and to produce appropriate advice and assessment. When a teacher corrects a mistake made by K, the computer can add a new method or constraint to the object concerned, or generate states to existing objects. However, when K turns off C, all states are lost. The common experience that C and K had together remains only in K's memory. Of course one can add a logic rule in the C database, but this is tantamount to building a logic semantic. In the case of C and K, the child will remember his experience with the computer and teacher, but the computer will be left with a dead database.

Memoization stores the results of expensive function calls and returns the cache when the same inputs occur again. In few words, what persistence does for object states, memoization does for procedures and algorithms. In natural language processing, memoization is specially useful to accomodate ambiguity and left recursion in polynomial time and space.

This thesis recycles the contribution of previous works. This means that grammars, libraries, packages, repositories, dictionaries and ideas were liberally used by the author.

The author believes that it is impossible to build a complex natural language processing application without targeted contributions of thousand of programmers during decades.

1.1 Objective

1. Objective 1: Implement a theorem prover based on Warren Abstract Machine. It is necessary to use a computer language that can accomodate any new technological advance, such as parallelism, persistence, memoization, etc. Its syntax and semantics must be based on mathematics, so it will not suffer any substantial change from one decade to another. Finally, it must have a powerful scheme of encapsulation to prevent the work of a contributer having deleterious effects on the efforts of another researcher. The only language with all these attributes is Common Lisp.

2. Objective 2: The syntax of the proposed inference engine should be as close as possible to the tradicional syntax of Prolog.

3. Objective 3: The interface should accept declarations of deterministic predicates as well as non-deterministic predicates.

4. Objective 4: The inference engine should optimize the use of time and space.

5. Objective 5: It is also necessary to create a repository management system so that a researcher can have instant access to all previous work. The technology for such a repository also exists, and it is called quicklisp. Therefore, the project presented herein will be available in quicklisp.

1.2 Work Layout

This study is divided into seven chapters. Chapter 1 presents an introduction to natural language processing, the incumbent difficulties encountered carrying out this process become the essence of the presented thesis.

Chapter 2 describes the Condorcet-List theorem and Rasch Model, both of which can be used for natural language processing.

Chapter 3 functions as a foundation for understanding the programming of the codes presented in this thesis.

Chapter 4 provides details of the efficienty implementation of the Warren Abstract Machine (WAM) as proposed in this thesis.

Chapter 5 discusses the difficulties one finds when searching for information on web, this difficulty presents itself through the excessive content available. also discussions based on textmining techniques are used in order to facilitate the extraction of such information.

The conclusion for this thesis is presentes in chapter 6.

The source code for the implemented wam is available in appendix i.

Chapter 2

Condorcet-List Theorem and Rasch Method

The first motivating problem of this thesis was to apply Condorcet-List Theorem in a panel of experts. Deeper into the theme, the necessity of knowing the probability of success of each specialist who participated in the panel arose. At this moment, the second motivating problem emerged. How to discover the probability of success of each specialist? Would there be a measurement to each skill? Would there be a measurement for knowledge? Studying Theory of Measurement, we ascertained that the Model proposed by Rasch, Lord and Lazarsfeld may be applied to this problem.

This chapter aims at presenting how the Model of Rasch combined with Condorcet-list Theorem may be applied. The content of this chapter may be applied in several areas, such as data processing and engineering and in the problem that will be presented in the chapter 5 of this thesis, the process through which voting functions as a decantation filter, or be it produces through the majority vote a sufficient filter for finding the best content.

2.1 The Jury Theorem

The Marquis of Condorcet was the first person to address the problem of assessing a collective decision. In its original form, his jury theorem requires that decisions be taken through a majority vote. The method permits the probability calculation of a correct answer to a given yes/no question, where N is the number of voting experts, p is the

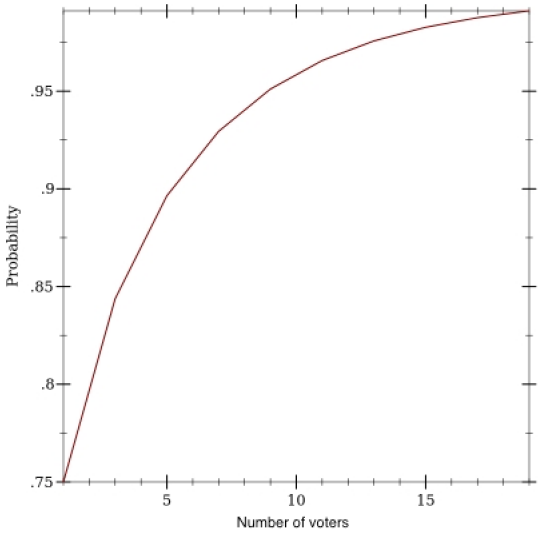


Figure 2.1: Condorcet's Jury Probabilities

probability of an individual expert being right, and m the number of votes necessary for a majority. In this form, one can use the cumulative binomial distribution to calculate the probability of a majority rule decision being correct.

$$P_N = \sum_{i=m}^{N} \left(\frac{N!}{(N-i)!i!}\right)(p)^i(1-p)^{N-i} \tag{2.1}$$

Figure 2.1 shows that the probability of the majority vote producing the correct result tends to 1 as the number of voters increase, provided that all voters have a probability $p > 0.5$ of producing the correct classification. One can take inspiration from the fact that although this result does not solve the problem, it suggests that a solution to the problem of accepting or refusing a piece of information exists. In fact, a publisher often needs a filter that reduces noise and eliminates spurious readings. This kind of filter is based on the knowledge of the probability distribution of the random variable that one needs to measure, and also on the probability of getting the right result from a set of referees.

The first problem one faces here is that not all referees from the set have the same probability of hitting a given error interval.

Although Condorcet's theorem suggests that adding voters with different hitting probabilities may increase the combined probability, this fact was not proven by Condorcet

Figure 2.2: Hit probability of the jury

himself, since his proof assumes that all jury members have the same hitting probability. However, Christian List and Robert Godin generalized the Condorcet's theorem to many voters with different probabilities. Basically, they proved two propositions:

Proposition 1. There are k options, and each referee has independent probabilities $p_1, p_2, p_3 \ldots p_k$ of choosing options $1, 2, ..., k$. Besides this, the probability p_i of voting for the correct outcome i exceeds each of the probabilities p_j of voting for any of the wrong outcomes, $j \neq i$. In this case, the correct option is more likely than any other option to be the plurality winner.

Proposition 2. As the number of voters/jurors tends to infinity, the probability of the correct option being the plurality winner converges to 1.

The reader will find the proof of these two propositions in [20].

In the present work the author will show how the Rasch model can be used to measure the value of a random variable, and assign a probability to the random variable that corresponds to a given measurement. In particular, the work describes how one can use the Rasch model in the identification of the information contents. Wright and Mok [21] provides a good tutorial on the Rasch model.

A voter is an adaptative referee that learns how to classify from a group of examples. For instance, a video camera that learns to recognize faces or Chinese characters is an

example of a classifier. Learning machines achieve classification goals through Artificial Intelligence schemes, like neural networks, deep learning, energy based learning, symbolic computation and genetic programming.

The author of the present work have suggested taggers coupled with the Condorcet-List theorem, Rasch method to solve the inverse problem of determining the grammatical classes of words, necessary to designing a natural language processing system. Since other researchers have placed a great number of well tested classifiers in libraries (for example, the rasp3os system) easily available to any engineer with access to the Internet and a working knowledge of Lisp, this work will concentrate on the Rasch method and the Condorcet-List theorem.

2.2 Rasch Model

In a measurement, there is a variable that one wants to evaluate. Variables like weight, temperature or height can be measured directly with scales, measuring tapes, etc. Unobservable variables like skill, efficiency, initiative, or soil electric properties are not so easy to measure. One can describe such latent variables, but cannot compare them to a standard meter, since they lack physical dimensions. However, one needs to assess them to appraise the quality of health care services, to design a grounding system or students in a particular subject.

In order to estimate the value of a latent variable, Rasch, Lord and Lazarsfeld developed independently a branch of statiscs known as Measurement Theory.

Since the Rasch model proved so useful in measuring latent traits and attributes in human sciences, the author of this work used this Model for example to evaluate the grammatical classes of English words.

2.3 Scales

When one decides to measure something, a choice of measurement scales must be made. In statistics, these scales can be nominal, ordinal, interval and ratio. The present work uses ratio scales. Influenced by previous research, the author talk about learning entities

as being individuals.

Let us consider two persons c and v, and an item r that one wants to classify. What is the probability of c hitting item r? What is the probability of v hitting item r? The first problem is to compare the skills of c and v based on their performance in resolving item r. In this context, an item is a member of an object class. The two persons need to classify many items to arrive at a raw score.

Figure 2.3: C x V

The difference between raw scores is not a good basis for comparison. If both scores are very high, a large difference may not be meaningful. Therefore, statisticians prefer ratios. Let us ignore those results where both c and v hit or both of them fail together.

The probability of c succeeding in solving the problem r is given by P_{cr} and the probability of c failing to solve the same problem can be calculated by $(1 - P_{cr})$. On the other hand, the probabilities of v succeeding and failing at solving the same problem are given by P_{vr} and $(1 - P_{vr})$ respectively.

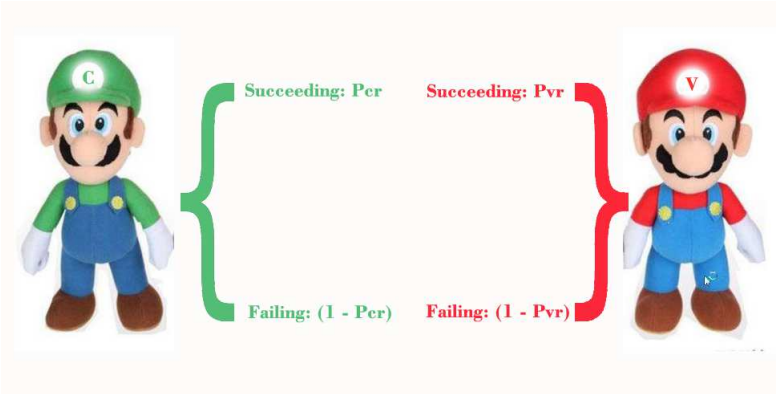


Figure 2.4: Success and failure probabilities for problem R

Let N_{10} be the notation of how many items c hits and v misses by the number of tries. On the same token, let N_{01} denote the number of c misses and v hits by the number of tries. The ratio between N_{10} and N_{01} is given below

$$\frac{N_{10}}{N_{01}} = \frac{P_{cr} \times (1 - P_{vr})}{(1 - P_{cr}) \times P_{vr}}$$

Probability is estimated as the relative ratio of success to the number of trials, where the number of trials tend to infinity.

2.4 Specific objectivity

One can say that c is better than v if its superior results do not depend on the problem. This property is called specific objectivity.

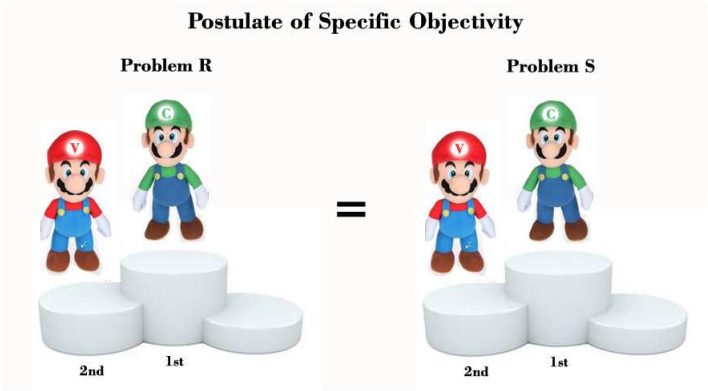


Figure 2.5: Specific Objectivity

Let us consider two variations r and s of a given problem. One has hand a good statistical comparison between c and v when the ratio N_{10}/N_{01} does not change when one changes the item. In this case, one has the equality given below.

$$\frac{P_{cr} \times (1 - P_{vr})}{(1 - P_{cr}) \times P_{vr}} = \frac{P_{cs} \times (1 - P_{vs})}{(1 - P_{cs}) \times P_{vs}}$$

Odds can be described as the ratio of the probability of an event occurring to the probability of it not occurring. One can rewrite this equality in order to obtain the odds of success for c solving item r

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{cs}}{(1-P_{cs})} \times \frac{(1-P_{vs})}{P_{vs}} \times \frac{P_{vr}}{(1-P_{vr})}$$

2.5 Origin

The next step is to choose the origin for the measurement scales that one intends to introduce. Let us consider a classifier o whose ability matches the difficulty of an item o. In this case, the classifier will solve the item in half of the trials, and the item will defeat the classifier for the other half. This classifier is said to be at the origin of the ability scale, and the item is at the origin of the difficulty scale. Since the classifier solves the item half of the times, the probability of success is $P_{oo} = 0.5$.

Let us compare c with the classifier of the origin. The ability of a classifier at the origin ties with the difficulty of the problem at the origin.

For the problem discussed in this work, when the grammatical classes of English words is correct the problem is overcome, on the contrary, the problem defeats the method.

When the two are tied in strength, the probability of the method at the origin solving the problem at the origin is 0.5, and the probability of the problem persisting is also 0.5.

Substituting individual v for individual o and problem s for problem o, one can rewrite the formula to calculate the odds of c finding the solution to the item r.

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{co}}{(1-P_{co})} \times \frac{(1-P_{oo})}{P_{oo}} \times \frac{P_{or}}{(1-P_{or})}$$

Since P_{oo} is 0.5 one has that $(1-P_{oo})/P_{oo} = 1$. Therefore

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{co}}{(1-P_{co})} \times \frac{P_{or}}{(1-P_{or})}$$

Let's take the logarithm from both sides of this equation

$$\ln(\frac{P_{cr}}{(1-P_{cr})}) = \ln(\frac{P_{co}}{(1-P_{co})} \times \frac{P_{or}}{(1-P_{or})})$$

$$\ln(\frac{P_{cr}}{(1-P_{cr})}) = \ln(\frac{P_{co}}{(1-P_{co})}) + \ln(\frac{P_{or}}{(1-P_{or})})$$

$$\ln(\frac{P_{cr}}{(1-P_{cr})}) = \ln(\frac{P_{co}}{(1-P_{co})}) - \ln(\frac{(1-P_{or})}{P_{or}})$$

If one defines

$$A_c = \ln(\frac{P_{co}}{(1-P_{co})})$$

$$D_r = \ln(\frac{(1-P_{or})}{P_{or}})$$

The equation becomes:

$$\ln(\frac{P_{cr}}{(1-P_{cr})}) = A_c - D_r$$

Notice that A_c does not depend on the problem r and D_r does not depend on the classifier c. This finding is the greatest contribution made by Georg Rasch to the Measurement Theory.

Therefore, A is a measurement of the classifier, and does not depend on any problem in particular; D is the measurement of the problem, and does not depend on any classifier.

The definition of the logarithm yields the following expression for the ratio between the probability of success and the probability of failure for a given item and classifier.

$$\frac{P_{cr}}{(1-P_{cr})} = e^{A_c - D_r}$$

$$P_{cr} = e^{A_c - D_r} - P_{cr} \times e^{A_c - D_r}$$

$$P_{cr} \times (1 + e^{A_c - D_r}) = e^{A_c - D_r}$$

$$P_{cr} = \frac{e^{A_c - D_r}}{1 + e^{A_c - D_r}} \quad \textbf{Logistic equation} \tag{2.2}$$

One often refers to parameters A_c and D_r as the ability of the individual/classifier and item difficulty respectively.

An assumption of the model is that the ratio N_{10}/N_{01} that compares two classifiers remains invariant when one changes the item.

The logistic equation 2.2 was deduced from the postulate of Specific Objectivity, which therefore is the measurement model that will be used for the problem under analysis in this work.

2.6 Rasch Logistic Distribution

The Danish psychologist Georg Rasch proposed a measurement theory where the probability distribution is essentially logistic, id est, the probability distribution is given by the following expression:

$$P_{cr} = \frac{e^{A_c - D_r}}{1 + e^{A_c - D_r}} \qquad \textbf{Logistic equation}$$

The model for testing will be represented by a list of tuples. Each tuple represents a test item. The first element of the tuple is the parameter u, with the value 1 for a hit or 0 for a miss. In order to estimate the skill of an examinee, one starts with a first approximation of the skill, and obtains better estimations through successive approximations.

Table 2.1: Raw Scores

Individual/Item	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Person C	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	0.0
Person V	1.0	0.0	1.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0
Person 3	1.0	1.0	1.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0
Person 4	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
Person 5	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0

Many people not conversant with measurement theory think that one does not need a model for measuring a latent variable. For instance, why do I need probability distribution to calculate a classifier's probability of hits or a student's score? One often argues that one can simply divide the number of correct answers by the total number of questions, in order to obtain the score.

According to the Classic Theory the average arithmetic of the rows of the array, given by Table 2.1 represents the ability of the individual as one can see from table 2.2.

Table 2.2: Ability of Individuals - First approximation

Individual	**Ability**
Person C	0.8
Person V	0.6
Person 3	0.7
Person 4	0.2
Person 5	0.5

According to the Rasch model, the problem presented herein follows the logistic curve. Thus the ability of the individual is presented in Table 2.3. By considering the individual `O` as the origin of the ability scale, one has:

```
|_____________________________|_____________________________|
      P4                    P5   O       V       P3      C
   -1.919                -0.054       0.548    1.194  1.936
```

Figure 2.6: Origin of the Ability Scale

Table 2.3: Ability of Individuals - Rasch Model

Individual	**Ability**
Person C	1.936
Person V	0.548
Person 3	1.194
Person 4	-1.919
Person 5	-0.054

According to the Classic Theory the average arithmetic of the column of the array, given by Table 2.1 represents the first approximation of difficulty for the item/problem is presented on Table 2.4.

Table 2.4: Difficulty of Problem - First approximation

Item	1	2	3	4	5	6	7	8	9	10
Difficulty	0.8	0.8	0.8	0.8	0.6	0.4	0.6	0.2	0.2	0.4

Table 2.5: Difficulty of Problem

Item	1	2	3	4	5	6	7	8	9	10
Difficulty	-1.507	-1.507	-1.507	-1.507	-0.124	0.968	-0.124	2.169	2.169	0.968

Table 2.5 presents the difficulty of each item, in accordance with the Rasch model. By considering the problem `0` as the origin of the difficulty scale, one has:

```
|_______________________________|_______________________________|
        1                 5     0           6                 8
    -1.507          -0.124              0.968           2.169
```

Figure 2.7: Origin of the Difficulty Scale

The results obtained in tables 2.3 and 2.5 will be explained in section 2.7.

This section shows that models are necessary in any measurement specially in measurements of latent variables. Imagine that a pilot wishes to measure the distance between the cities of Paris and New York. Through the use of Riemann Geometry the shortest distance between two points lies on a great circle; in this case, the distance between Paris and New York is around 5,884,000 meters. Below, one calculates the same distance according to the Euclidean Geometry.

```
CL-USER(1): (defparameter rieD 5884000)
RIED
CL-USER(2): (defparameter R #i(40000000.0/2.0/pi))
R
CL-USER(3): (defparameter eucD  #i(2*R*sin(alpha/2)))
EUCD
CL-USER(4): #i(rieD-eucD)
207208.64909257367d0
```

By comparing the two measurements and using the Earth as a model for the Riemann

Geometry, a difference of about 200 km is encountered. Conclusion: data only makes sense when inserted into a model.

2.7 Calibration Algorithm

In this section, the author presents the steps for calibrating the Rasch Model. The facilitator builds a table where each column contains hits or misses for a given item. Each row shows hits or misses for a grammatical class (vide Table 2.6).

Table 2.6: Raw Scores

	You+is	**She+give**	**They+is**	**It+stay**	**She+have**	**He+do**
Person 1	0	1	1	0	1	1
Person 2	1	0	1	1	0	0
Person 3	0	1	0	1	1	1
Person 4	0	0	1	0	0	0
Person 5	0	0	0	1	0	0
Person 6	0	1	1	1	0	1

Let us understand what calibration is. If we determine that probability depends on parameters like difficulty and ability, calibration therefore is the process of determining these parameters.

In general, raw measurements do not correlate well with the model. However, Joint Likelihood Estimation can force raw data into the model, thus discovering the ability and difficulty parameters.

On the facilitator's matrix, the cells receive value 1 if grammatical class is correct or 0 otherwise (vide Table 2.6).

```
(defparameter Xn
  #2A((0.0 1.0 1.0 0.0 1.0 1.0) ; 1
      (1.0 0.0 1.0 1.0 0.0 0.0) ; 2
      (0.0 1.0 0.0 1.0 1.0 1.0) ; 3
      (0.0 0.0 1.0 0.0 0.0 0.0) ; 4
      (0.0 0.0 0.0 1.0 0.0 0.0) ; 5
      (0.0 1.0 1.0 1.0 0.0 1.0) ; 6
))
(defparameter d0 (array-dimension Xn 0))
(defparameter d1 (array-dimension Xn 1))
```

Using this initial data one proceeds to determine the origin of the ability and difficulty scales. To meet this goal, an iterative algorithm must force raw data onto the logistic curve. The first step of the iteration calculates row and column averages to estimate initial difficulty and ability vectors for the data matrix Xn.

The `hits` function returns a list with the average arithmetic for each line of the array `Xn`. The average will be the starting point for the ability calculation. The function `hab_logit` calculates the initial ability vector.

```
(defun hits(m)
 (make-array (list d0) :initial-contents
  (loop for i from 0 below d0
    collect (loop
     for j from 0 below d1
      when #i(m[i,j]==1) sum 1.0 into s1
        finally (return #i(s1 /d1))) )))

(defun hab_logit (vet &optional
   (v_logit (make-array d0)))
   (loop for i from 0 below d0 do
      #i(v_logit[i]=log(vet[i]/(1-vet[i]))))
   v_logit)
```

The `misses` function returns a list with the arithemetic average of each column in the array `Xn`. The average will be the starting point for the difficulty calculation. The function `dif_logit` produces the initial difficulty vector.

```
(defun misses(m)
 (make-array (list d1) :initial-contents
  (loop for j from 0 below d1
    collect (loop
     for i from 0 below d0
      when #i(m[i,j]==1) sum 1.0 into s1
        finally (return #i(s1 / d0))) )))

(defun dif_logit (vet &optional
   (v_logit (make-array d1)))
   (loop for i from 0 below d1 do
     #i(v_logit[i]=log((1-vet[i])/vet[i])))
   v_logit)
```

The next step is to adjust the difficulty vector by subtracting the average from each

element. The function `probability` calculates the odds given by equation 2.2. The odds for each difficulty/ability pair is stored in a two dimensional array.

```
(defun avg-vector (vet)
   (loop for x across vet
      sum x into s count x into c
      finally (return (/ s c))))
(defun adj_dif_logit (vet avg)
   (map 'vector (lambda(x) (- x avg)) vet))

(defun probability (A Dadj &optional
        (m (make-array (list d0 d1))))
 (loop for i from 0 below d0 do
   (loop for j from 0 below d1 do
       #i(m[i,j] :=  (exp (A[i] - Dadj[j])) /
                (1.0 + (exp (A[i] - Dadj[j])))
)) m)
```

In order to update the ability and difficulty vectors, one must calculate the residual between the current and the previous probability matrix.

```
(defun residual(mi me &optional
 (rs (make-array (list d0 d1))))
    (loop for i from 0 below d0 do
 (loop for j from 0 below d1 do
     #i(rs[i,j] := mi[i,j] - me[i,j]) ))
    rs)
(defun residual_sum(m &optional
 (sum (make-array d0)))
    (loop for i from 0 below d0 do
 (loop for j from 0 below d1 do
       #i(sum[i] := sum[i] + m[i,j]) ))
    sum)
```

Besides the residual matrix, one needs also to calculate the variance of the probability matrix.

```
(defun variance (m &optional
         (mv (make-array (list d0 d1))))
 (loop for i from 0 below d0 do
   (loop for j from 0 below d1 do
      #i(mv[i,j] := m[i,j] * (1 - m[i,j])) ))
    mv)

(defun sum_row_mat(m)
   (make-array (list d0) :initial-contents
   (loop for i from 0 below d0 collect
      (loop for j from 0 below d1
           sum #i(-m[i,j])))) )

(defun sum_col_mat(m)
  (make-array (list d1) :initial-contents
    (loop for j from 0 below d1 collect
       (loop for i from 0 below d0
            sum #i(-m[i,j])))) )
```

After summing up the residual and variance for each of the two dimensional arrays along each row, one is ready to update the difficulty and ability vectors.

```
(defun newA(A rs vrc &optional
            (nAbil (make-array d0)))
  (loop for i from 0 below d0 do
      #i(nAbil[i] := A[i]-rs[i]/vrc[i]))
  nAbil)

(defun newD (D rs vrc &optional
              (nDif (make-array d1)))
  (loop for i from 0 below d1 do
     #i(nDif[i] := D[i]-rs[i]/vrc[i]) )
  nDif)
```

These steps must be repeated until the sum of the squares of the residuals becomes sufficiently small.

The calibration algorithm produces a table containing the probabilities of each classifier solving a given item, see table 2.7. In this table the lower the value present in the column *Inability* lower will be ability that the person possesses, on the other hand, the value for ability of the person will be higher. Therefore, the lower the value presented on line *Easiness* the solution to the item will be easier, on the contrary, it solution will be more difficult.

Table 2.7: Probabilities

	You+is	**She+give**	**They+is**	**It+stay**	**She+have**	**He+do**	**Inability**
Person 1	0.2683	0.7255	0.8771	0.8771	0.5153	0.7255	0.8326
Person 2	0.1343	0.5280	0.7512	0.7512	0.3103	0.5280	-0.0405
Person 3	0.2683	0.7255	0.8771	0.8771	0.5153	0.7255	0.8326
Person 4	0.0234	0.1472	0.3179	0.3179	0.0649	0.1472	-1.9306
Person 5	0.0234	0.1472	0.3179	0.3179	0.0649	0.1472	-1.9306
Person 6	0.2683	0.7255	0.8771	0.8771	0.5153	0.7255	0.8326
Easiness	1.8066	-0.1509	-1.1238	-1.1238	0.7479	-0.1509	

Using this initial data one proceeds to determine the origin of the ability and difficulty scales. To meet this goal, an iterative algorithm must force raw data onto the logistic curve.

The facilitator adds a row for the Condorcet-List classifier to the table. The Condorcet-List classifier selects all classifiers with probabilities above 0.5 and promotes a votation among them, choosing the plurality winner class.

Table 2.8: Probabilites of Voters

	You+is	**She+give**	**They+is**	**It+stay**	**She+have**	**He+do**
Person 1		0.7255	0.8771	0.8771	0.5153	0.7255
Person 2		0.5280	0.7512	0.7512		0.5280
Person 3		0.7255	0.8771	0.8771	0.5153	0.7255
Person 4						
Person 5						
Person 6		0.7255	0.8771	0.8771	0.5153	0.7255

In table 2.8 all voters with a hitting probability less than 0.5 were removed. After this filtering process the majority vote was realized. The majority vote was obtained through a consideration of the vote of each voter, presented in table 2.6. In the case of a draw or that there does not exist a single vote for the item, the value of zero is attributed for the Condorcet-List vote. Therefore, the Condorcet-List vote is presented in table 2.9.

Following this, the facilitator repeats the calibration process, in order to recalculate the abilities and difficulties again, this time including the Condorcet-List classifier. Table 2.10 shows that the Condorcet-List classifier produces hit probabilities better than others

Table 2.9: Condorcet-List Voter

	You+is	**She+give**	**They+is**	**It+stay**	**She+have**	**He+do**
Condorcet-List	0.0	1.0	1.0	1.0	1.0	1.0

voters.

Table 2.10: Hit Probabilities

	You+is	**She+give**	**They+is**	**It+stay**	**She+have**	**He+do**
Condorcet-List	0.3999	0.9117	0.9644	0.9645	0.8168	0.9117

The size of the sample in the Rasch Model, does not need to be large. As with a small amount of data it is possible to trace the logistic curve. In the example presented herein, a small sample of individuals was used, as the aim was to illustrate the model. However, when the goal is to analyse grammatical structures of English words a larger sample base is necessary.

Chapter 3

Lisp and Mathematics

In this chapter, the author will provide the content necessary for an adequate understanding of all following chapters.

Which programming methodology should I use to prevent my programs from becoming obsolete? The author presents the reasons why Lisp was chosen as the language for the development of this work.

Computer languages have, in general, a very complex syntax. A Python programmer, for instance, must learn many syntactical variants for calling up a function: The function that calculates the logarithm has a prefix notation, while the functions that perform multiplications and additions obey infix syntax rules. Even the four basic arithmetic operations do not obey the same syntax rules, since they have different precedence and associativity.

```
>>> import math
>>> math.log(3,4)
0.7924812503605781
>>> (3+4)*(5+6+7)*8
1008
```

Since this complicated syntax is beyond the reach of most programmers, the design of the forms that one can use in source code is left to a team of compiler writers. Therefore, software has two components, a compiler for a general purpose language and an application written with forms that the compiler accepts. The professionals who develop the

application cannot adapt the compiler to fit the problem they are dealing with.

Instead of accepting an arbitrary syntax, the Lisp community adopted a kind of symbolic expressions, which evolved from a mathematical notation proposed by Łukasiewicz in 1920. In mathematics, basic constructions and transformation rules are kept to a minimum. This LEX PARSIMONIÆ has many consequences. The first one is that few rules can classify all symbolic expressions:

1. Numerals have traditional notation: 3, 3.1416, -8, etc.
2. Symbols are sequences of characters that cannot be interpreted as numbers and do not contain brackets: `sin`, `x`, `y`, `*ops*`, etc.
3. Quoted lists such as `'(a b e)` represent sequences of objects.
4. Unquoted lists such as `(log 8 2)` denote function calls.
5. Abstractions such as `(lambda(x y) (log (abs x) y))` define functions. The sequence of symbols `(x y)` is called a binding list, and the expression `(log (abs x) y)` is the body of the abstraction.

Any variable that appears in the binding list is said to be bound. Free variables occur in the body of the abstraction but not in the binding list.

Another consequence of its mathematical foundations is that Lisp has a clear and simple semantics. The rules of computation are based on a Mathematical system called the Lambda Calculus.

α-conversion Changing the name of a bound variable produces equivalent expressions. Therefore, $(\lambda(x)x)$ and $(\lambda(y)y)$ are equivalent.

β-reduction $((\lambda(x)E)v)$ can be reduced to $E[x := v]$.

Lisp has a Read Eval Print Loop (REPL) that performs β-reduction over symbolic expressions in order to simplify their forms. In the β-reduction process, the first element of a list can be considered as a function or a macro.

In Lisp, all functions follow the list syntax, without a single exception: The first element of the list is the function identifier, and the other elements are arguments.

```
CL-USER(1): (log 3 4)
0.79248124
CL-USER(2): (* (+ 3 4) (+ 5 6 7) 8)
1008
```

One immediate advantage of its mathematical foundations is that Lisp is unlikely to become obsolete. This means that one may expect that a language like Python or Fortran to suffer changes without backward compatibility. One can even expect that Python or Fortran would be phased out. However, Lisp code written many decades ago can be easily run in a modern computer. Besides this, since Lisp does not change, computer scientists can work on Lisp compilers for many decades, which results in fast and robust code.

Let `(lambda(x) (- (/ (* (+ x 40) 9) 5.0) 40))` represent the function $\lambda(x)(x + 40) \times 9/5 + 40$ that converts Celsius degrees to Fahrenheit. One can store this function in a functional symbol, as one can see below.

```
(setf  (symbol-function 'c2f)
 (lambda(x)
   " Celsius to Fahrenheit"
   (- (/ (* (+ x 40) 9) 5.0) 40)))

#| From REPL:
CL-USER(1): (c2f 100)
212.0
|#
```

There are three ways to make comments. Everything from a semicolon to the corresponding end of line is considered a comment by the compiler. Besides this, one can put a comment block between `#|` and `|#` as exemplified in the above snippet. A documentation string can appear after the argument list of a function definition. From the REPL, the documentation string can be retrieved through the `(documentation 'c2f 'function)` command.

3.1 Tool for natural language processing

One tool used for natural language processing is rasp3os[2]. The system is written in Lisp, with a few low level components in C. The distribution, according to the README, includes *unix shell scripts for running the whole analysis system, or just the parser.* Let us see how to install the system, and test it using the provided shell scripts.

One must operate the rasp3os through the command-line interface. A shell command language interpreter (CLI) is a tool for interacting with the operating system where the client issues commands in the form of successive lines of text. A search on the Internet shows that there are plenty of tutorials on the CLI[3], but a person who is not fluent in shell script will be better off asking help from a computer science major.

In order to study and use rasp3os, the interested reader must download the installer from the distribution page[2]. After expanding the archive, all one needs to do is type `make` into the shell. The next step is to test the scripts.

```
~/rasp3os/scripts$ echo "Helen, thy beauty is to me\
 like those Nicean barks of yore" | ./rasp.sh -p'-os -u'
(|Helen_NP1| |,_,| |thy_APP$| |beauty_NN1|
 |be+s_VBZ| |to_II| I+_PPIO1 |like_II|
 |those_DD2| |Nicean_NP1| |bark+s_NN2|
 |of_IO| |yore_NN1|) 1 ; (-25.427)
sparkle: 1
("S" ("NP" "Helen") "," ("NP" "thy" "beauty")
 ("VP" "be+s" ("PP" "to" "I+")
  ("PP" "like"
   ("NP" "those" "Nicean" "bark+s"
      ("PP" "of" ("N1" "yore"))))))
```

From the above output, one can readily see that rasp performs two sorts of text analysis, tagging and syntax.

- **Tagging.** This consists of classifying each input word according to its grammatical function. The download page of the Constituent Likelihood Automatic Word-tagging System (CLAWS) states that it has been in continual developement since

the early 1980s. The latest version of the tagger, CLAWS4, was used to perform Part-of-speech tagging of circa 100 million words of the British National Corpus. This very near complete and robust tagger is part-and-parcel of the rasp3os package.

- **Syntax.** After tagging, the rasp system builds the text syntax tree. The system can produce many syntactic representations for a given tagged text. The representation shown above is convenient as it can be read back into a Lisp system for further processing.

The author will look closer at Tagging and Syntax in chapter 4.

3.2 Emacs

In order to simplify the communication style, let us assume that Nia, a programmer, has already checked the functionality of rasp, and decided to try her hand at natural language processing. In order to work with Lisp, Nia should install the following packages:

1. A Lisp compiler, such as sbcl[4] or clozure.
2. Emacs, a customizable, self-documenting display editor.
3. Slime[5], the superior Lisp interaction mode for Emacs.
4. Quicklisp[6], a tool for installing Lisp applications.

A linguist that is not conversant with Lisp should ask for help from a computer science major when installing and configuring the above systems.

An environment used frequently for developing Lisp programs is Emacs, a text editor that one can configure with macros written in a dialect of Lisp designed for processing text. Emacs has a Superior Lisp Interaction Mode, also known as *slime*. Make sure to ask the computer science major that is helping you to check whether *slime* is configured correctly.

In order to configure Emacs, Nia will start the editor, and open a unix shell with the command `M-x eshell`. Any tutorial about emacs explains how to issue this command: Nia must maintain the Alt key down, and press the x key. After this, she types `eshell`

into the emacs minibuffer (a one line buffer at the bottom of the page). From the eshell, she uses the command line interpreter to check whether slime is correctly installed.

```
~/wrk/ $ cd ~  # Comments start with the hash symbol
~ $ ls quicklisp/ # checks for quicklisp
asdf.lisp  setup.lisp  cache
dists      quicklisp   tmp
~ $ cd .emacs.d/  # visits the emacs directory
~/.emacs.d $ ls # make sure that slime is installed
auto-save-list  eshell  lisp  slime
~/.emacs.d $ ls lisp # OS-x needs this application
exec-path-from-shell.el
```

The command `C-x C-f` reads the `~/.emacs` name from the minibuffer, creating a new buffer for the `~/.emacs` configuration file.

```
Find file: ~/.emacs
```

Just like any other program, Emacs can benefit from a configuration file in order to work correctly. Therefore, Nia types the following script into the `.emacs` configuration file:

```
;; .emacs for scripting in sbcl
(global-visual-line-mode t) ; wrap word at the end of line
(setq inhibit-splash-screen t)
(setq inhibit-startup-message t)
(set-face-attribute 'default nil :height 160)

;; Necessary to make emacs PATH equal to the shell PATH
(add-to-list 'load-path "~/.emacs.d/lisp")
(require 'exec-path-from-shell)
(exec-path-from-shell-initialize)

(global-set-key (kbd "C-c C-s") 'enlarge-window)
```

```
(setq linum-format "%4d ")
(global-linum-mode 1)

(setq inferior-lisp-program "/usr/local/bin/sbcl")
(add-to-list 'load-path "~/.emacs.d/slime/")
(require 'slime)
```

Computational systems that perform complex tasks such as natural language processing are large and have dependencies scattered all over the web. Richard Stallman suggests that learners of such a system should start their studies by testing open source applications, then proceed to modify them to fit the situation at hand.

3.3 Quicklisp

The quicklisp system provides the tools to download and install packages written in Common Lisp. Therefore, the interested reader should visit the distribution page[6] and install quicklisp before any attempt at experimenting with natural language processing.

Let us examine how Nia performed the installation and configuration of quicklisp. She downloads the `quicklisp.lisp` installer and places it in her home directory.

```
~$ sbcl   # Nia has already installed sbcl
This is SBCL 1.2.11, an implementation of ANSI Common Lisp.

* (load "quicklisp.lisp")
T
* (quicklisp-quickstart:install)
NIL
* (ql:add-to-init-file)
```

After checking Emacs, Nia will configure the compiler for Common Lisp. There are two open souce compilers sbcl and clozure. In this work the author will use sbcl. Nia downloaded the binaries for her operating system, and performed the installation according to

the instructions given in the readme file.

From Emacs, Nia press `C-x C-f` to open the `~/.sbclrc` configuration file, and finds it in the following situation:

```
;;; The following lines added by ql:add-to-init-file:
#-quicklisp
(let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                       (user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))
```

The only command present in the `.sbclrc` configuration file was introduced there by quicklisp. Then Nia adds a few Lisp instructions to this nearly empty `.sbclrc` file, in order to profit from a better environment for experimenting with Common Lisp.

Lisp has a read eval print loop (REPL). This means that the Lisp compiler reads an expression, evaluates it and prints the result. After installing sbcl and quicklisp, Nia can start the REPL from a unix shell and test the whole system as shown below. She also installs the `tagger` package to get acquainted with the normal Lisp installation process.

```
~/tg$ sbcl
* (require :sb-aclrepl)  ;; use semicolon for comments
("SB-ACLREPL")
CL-USER(2): (ql:quickload :tagger)
(:TAGGER)


CL-USER(3): (tag-analysis:tag-string
     "The white cat also catches mice.")
The white cat also catches mice.
at  jj/2  nn  rb   vbz/2   nns
```

```
CL-USER(3): (ql:quickload :infix)
To load "infix":
  Load 1 ASDF system:
    infix
; Loading "infix"

;;; ******************************************
;;;   Infix notation for Common Lisp.
;;;   Written by Mark Kantrowitz.
;;; ******************************************

(:INFIX)
CL-USER(4): #i(1.8 * 38+32)

100.4
CL-USER(5): (exit)  ;; quit Lisp
```

Nia starts sbcl from the unix shell. The default prompt is an asterix, which is far from ideal, since it does not provide any information concerning the environment. The command

```
(require :sb-aclrepl)
```

introduces a better prompt. Note that Lisp expressions always have the form: open parenthesis, operation, arguments, close parenthesis. In the present case, the operation is `require`, and the sole argument is the `:sb-aclrepl` keyword. If Nia adds this improved prompt into the `.sbclrc` configuration file, Lisp activates it at each startup.

```
;;; The following lines added by ql:add-to-init-file:
#-quicklisp
(let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                       (user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))
```

```
;; Infix notation for Lisp
(ql:quickload :infix)

;; A better Read Eval Print Loop
(require 'sb-aclrepl)

;; Let us turn the debugger off
(defun debug-ignore (c h)
   (declare (ignore h))
   (format t "~a" c)
   (abort))

(setf *debugger-hook* #'debug-ignore)
```

Infix notation. Lisp arithmetic operations have prefix notation, like any other expression in the language. Instead of writing `9*(38+40)/5.0-40` for converting 38 Celsius degrees to Fahrenheit, a Lisp programmer would type `(- (/ (* 9 (38+40)) 5.0) 40)` to reach the same result. By adding

```
;; Infix notation for Lisp
(ql:quickload :infix)
```

to the `.sbclrc` configuration file, Nia installs infix notation in Lisp. Then, she can write the temperature conversion formula thus:

```
#i(9*(x+40)/5.0-40)
```

Debugger. When one makes a mistake, Common Lisp enters a debugger, and presents a list of options to the programmer. Most people consider the debugger a nagging tool, and so they just press the number in front of the *ABORT* option to exit. Since Nia defined a `debug-ignore` function for the `*debugger-hook*`, Lisp will print error messages, but will not enter the debugger.

3.4 Emacs commands

One can use emacs through menus. However, advanced users prefer shortcut key strokes. In the following cheat sheet, κ can be any key, `Spc` denotes the space bar, `C-` is the `Ctrl` prefix, and `M-` represents the `Alt` prefix.

- `C-`κ – Press and release `Ctrl` and κ simultaneously

  `C-s` – search a text

  `C-k` – kill a line

  `C-Spc` then move the cursor – select a region

  `M-w` save selected region into a ring

  `C-w` kills region

  `C-y` insert killed or saved region

  `C-g` cancel minibuffer reading

- `C-x C-`κ – Keep `Ctrl` down and press `x` and κ

  `C-x C-f` – open a file into a new buffer

  `C-x C-s` – save file

  `C-x C-c` – exit emacs

- `C-x` κ – Press and release `Ctrl` and `x` together, then press κ

  `C-x b` – switch buffers

  `C-x 2` – split window into cursor window and other window

  `C-x o` – jump to the other window

  `C-x 1` – close the other window

  `C-x 0` – close the cursor window

When text is needed for carrying out a command, it is read from the minibuffer. For instance, if Nia presses `C-s` for searching a text, the text is read from the minibuffer. For giving up on a command, she may press `C-g`.

To transport a region to another place, Nia presses `C-Spc` to start selection and move the cursor to select the region. Then she presses `C-w` to kill the selection. Finally, she

moves the cursor to the insertion place, and presses the `C-y` shortcut. To copy a region, Nia presses `C-Spc` and moves the cursor to select the region. Then she presses `M-w` to save the selection into the kill ring. Finally, she takes the cursor to the destination where the copy is to be inserted and issues the `C-y` command.

3.5 Arithmetic operations

In order to get acquainted with Emacs, Nia entered the editor and pressed `C-x C-f` to create the `fahrenheit.lisp` buffer. She typed the program below into the buffer.

```
;; File: fahrenheit.lisp


(defun c2f(c)
  "Celsius to Fahrenheit" ;optional documentation
  (- (/ (* (+ x 40) 9) 5.0) 40))


(defun f2c(x)
  #i(5.0*(x+40)/9-40))
```

Function c2f converts Celsius readings to Fahrenheit. The `defun` macro, that defines a function, has four components, the function id, a parameter list, an optional documentation, and a body that contains expressions and commands to carry out the desired calculation. Comments are prefixed with a semicolon. In the case of `c2f`, the id is `c2f`, the parameter list is `(x)`, and the body is `(- (/ (* (+ x 40) 9) 5.0) 40)`.

Lisp programmers prefer the prefix notation: open parentheses, operation, arguments, close parentheses. Therefore, in `c2f`, `(+ x 40)` adds `x` to 40, and `(* (+ x 40) 9)` multiplies $x + 40$ by 9.

It is possible to use macros for automatic conversion from any syntax to the prefix form. For instance, the definition of `f2c`, that converts Fahrenheit readings to Celsius, the body is in the infix notation[7] of highschool precalculus.

In order to perform a few tests, Nia must initially save the program with `C-x C-f`. Then she goes to the emacs menu, and chooses the *Lisp/Run inferior Lisp* option. From

the Lisp REPL, Nia can load the program and call any application that comes with Lisp, or she defined herself.

```
CL-USER(1): (load "fahrenheit.lisp")
T
CL-USER(2): (c2f -40)
-40.0
CL-USER(3): (documentation 'c2f 'function)
"Celsius readings to Fahrenheit"
CL-USER(4): (f2c 100)
37.77778
```

Nia is working with two buffers, one running Lisp and the other containing a file. There are two ways for switching from one buffer to the other. Nia can go to the `Buffers > myfile` menu and choose the destination from a list of options. A better method is the `C-x b` command (press and release `Ctrl x`, then type `b`. The minibuffer shows the first option. Nia can scroll through the options by pressing the up arrow key. Once she finds the destination buffer by this process, she presses `Enter` to go to that point.

There is also a third option for switching between buffers. Initially, Nia splits the window with `C-x 2` and, afterwards, uses the `C-x b` to visit the `fahrenheit.lisp` buffer on one of the two windows.

```
    ;; CtoF.lisp


(defun c2f(x)
   #i(9*(x+40)/5.0-40))
```

```
CL-USER(1): (load "CtoF.lisp")
T
CL-USER(2): (c2f 37.77778)
100.0
```

Now she has the two buffers in front of her, each with its own window. Nia can use the mouse for jumping from one window to the other.

```
(defun wday( y m mday &optional
               (yfix  (floor (- 14 m) 12))
               (mm  #i(m+ 12*yfix -2))
               ( yy  (- y yfix))
               (century  (- (floor yy 100))))
     (mod (+ yy  (floor #i(13*mm - 1) 5)
                 (floor yy 4) century
          (floor yy 400) mday) 7))
```

The above program calculates the day of the week through Zeller's congruence. The `&optional` keyword introduces pairs of variable/values. For instance, the pair `(yy (- year yfix))` establishes that the variable `yy` may be initialized with the result of the `(- y yfix)` calculation. The `floor` function performs the integer divisions required by the algorithm.

3.6 The let-form

The let-form introduces local variables through a list of `(id value)` pairs.

```
(defun zr(y m d &optional
               (ax  (floor (- 14 m) 12))
               (mm  (+ m (* 12 ax) -2))
               (yy  (- y ax)) )
  (let ( ;open list of local variables
           (c  (- (floor yy 100))) ;first pair of (id value)
           (ly (floor yy 4))       ;second pair of (id value)
      ) ;close list of local variables
    (mod (+ yy  (floor (- (* 13 mm) 1) 5)
                ly c
         (floor yy 400) d) 7)))
```

In the basic let-form, a variable cannot depend on previous variables that appear in the local list. In the let-star form, one can use previous variables to calculate the value of a variable, as noted in the definition below.

```
(defun zlr(y m d)
  (let* (   (ax  (floor (- 14 m) 12)) ;id= ax
            (mm  (+ m (* 12 ax) -2))  ;id= mm
            (yy  (- y ax))            ;id= yy
            (c  (- (floor yy 100)))   ;id= c
            (ly (floor yy 4))         ;id= ly
        ) ;close list of local variables
     (mod (+ yy
             (floor (- (* 13 mm) 1) 5)
             ly (floor yy 400) d) 7)))
```

When using a let-form, the programmer must bear in mind that it needs parentheses for grouping together the set of `(id value)` pairs, and also parentheses for each pair. Therefore, one must open two parentheses in front of the first pair, one for the list of pairs and the other for the first pair.

A common error when dealing with the let-form is to forget the parentheses that group local variables together. Besides this, learners often leave out the parentheses that associate each variable with its value. Therefore, one is advised to study the syntax of the let-form carefully, to avoid these two types of error.

3.7 Lists

In Lisp, there is a data structure called list, that uses parentheses to represent nested sequences of objects.

```
;;File: probability.lisp


(setf  (get 'noun 'rs)
       '( ((n 0.5  cat)/)  ((n 0.5 mouse)/)) )
```

```
(setf  (get 'det 'rs)
       '( ((def 0.5 the)/)
          ((indef 0.5 a)/)
          ((indef 0.5 an)/))  )
(setf  (get 'np 'rs)
       '( ((np 0.5) = (n))
          ((np 0.5) = (det) (n))))
```

In the above file, the `setf` macro associates lists to the `'rs` indicator of the symbols `noun`, `det` and `np`. When these lists represent data, and not programs, they must be prefixed with a quotation mark.

From its very outset, Lisp connects indicators to every symbol. The symbols together with their indicators can be seen as a tabular data base called a property list. A property list has a number of entries, each one associated to an indicator. For instance, `(get 'np 'rs)` associates a list of clauses with an "`rs`" indicator appended to the symbol `np`. The command

```
(setf  (get 'np 'rs)
       '( ((np 0.5) = (n))
          ((np 0.5) = (det) (n))))
```

saves the quoted list into the `rs` storage location associated with symbol `np` in such a way that it can be retrieved with the `(get 'np 'rs)` function.

The functions `(car s)` and `(cdr s)` are called selectors, and are used to access the parts of a list: `(car s)` produces the first element of the `s` storage, while `(cdr s)` is the result of removing the first element from `s`. Combinations of `(car s)` and `(cdr s)` can fetch any element from a list. For instance, `(car (cdr ns))` produces the second element.

```
CL-USER(1): (load "probability")
T
CL-USER(2): (get 'np 'rules)
(((NP 0.5) = (N)) ((NP 0.5) = (DET) (N)))
```

```
CL-USER(3): (car (get 'np 'rules))
((NP 0.5) = (N))
CL-USER(4): (car (cdr (get 'np 'rules)))
((NP 0.5) = (DET) (N))
```

Besides the two selectors, lists have a constructor: `(cons x xs)` builds a list whose first element is `x`, and the remaining elements are grouped in `xs`.

```
CL-USER(5): (cons '((n 0.5 mouse device)/) (get 'n 'rules))
(((N 0.5 MOUSE DEVICE) /) ((N 0.5 CAT) /) ((N 0.5 MOUSE) /))
CL-USER(6): (get 'n 'rules)
(((N 0.5 CAT) /) ((N 0.5 MOUSE) /))
```

One has learned previously that lists must be prefixed by the special operator quote, in order to differentiate them from programs. To make a long story short, a quote prevents the evaluation of a list or symbol.

A backquote, not to be confused with quote, also prevents evaluation, but the backquote transforms the list into a template. When there appears a comma in the template, Lisp evaluates the expression following the comma and replaces it with the result. If a comma is immediately followed by `@,` (at-sign), then the expression following the at-sign is evaluated to produce a list. This list is spliced into place in the template. The examples below will make things clearer.

```
CL-USER(8): `(np ,(car (get 'det 'rules)) plus noun)
(NP (DET 1.0 THE) PLUS NOUN)
CL-USER(9): `(np ,@(car (get 'det 'rules)) plus noun)
(NP DET 1.0 THE PLUS NOUN)
```

Macros are programs that port a more convenient notation to a standard Lisp form. The syntax of Lisp, that unifies data and programs, makes it possible to create powerful macros that implement Domain Specific Languages (DSLs), speed up code or create new software paradigms. In fact, the macro system is one of the two features that places Lisp asunder from other languages, the other being its Mathematical Foundations. For learning macros in depth, the reader is referred to Dough Hoyte's book on the subject[8].

```
(defmacro def(category &rest rs)
  `(setf (get ',category 'rules) ',rs))

(def n ((n 0.5  cat)/) ((n 0.5 mice)/) )
(def det ((det 1.0 the)/) )
(def vbz ((vbz 1.0 chases)/) )
(def vp   ((vp 1.0) = (vbz) (np)) )
(def np  ((np 0.5) = (n)) ((np 0.5) = (det) (n)) )
(def sentence ((sentence 1.0) = (np) (vp)) )
```

In the above macro definition, the keyword `&rest` groups all remaining parameters into a list that the body of the macro can refer to through the variable that follows the keyword, in this case the variable is `rs`.

3.8 Packages

Nia wants to transform a string of characters into a list of words. This can be done through the `ppcre` library.

```
(ql:quickload :cl-ppcre)
(defpackage :xr (:use :cl :cl-ppcre))
(in-package :xr)
(defparameter ws "[A-Za-z][a-z]*")
(defparameter nums "-?[0-9]+\.?[0-9]*")
(defun tokenize(x) (all-matches-as-strings ws x))
```

```
CL-USER(1): (load "pkg.lisp")
T
CL-USER(2): (in-package :xr)
#<PACKAGE "XR">
XR(3): (tokenize "the black cat chases mice.")
("the" "black" "cat" "chases" "mice")
XR(4): (cl-user::exit)
```

Name clash is the first problem that one needs to solve when dealing with large projects. The system may have many collaborators, often working in places distant from one another. Even a lone ranger in programming needs to use libraries created by other people, as for example those libraries made available on quicklisp. Packages avoid that a person use an id that is already taken.

In the above example, Nia defined the package `xr`, that uses definitions from the `cl` and `cl-ppcre` packages. She declared that both her program and the REPL are inside the `xr` package. Since the REPL is inside the `xr` package, which does not know how to `exit`, Nia prefixed the `cl-user::exit` procedure with the `cl-user` package id.

Initially, the `xr` package starts out with only three names: the `ws` and `nums` parameters and the `tokenize` function.

The `all-matches-as-strings` function tokenizes a string, producing a list of words. A *regular expression*, regexp for short, determines what a word is. The regexp `"[A-Za-z][a-z]*"` states that a word starts with a letter from `A` to `Z` or from `a` to `z` (`"[A-Za-z]"`) followed by zero or more `[a-z]` letters. The asterisk mark that appears in the definition of `numbers` makes the previous character optional. The plus sign in `[0-9]+` means that numbers have at least one digit.

3.9 The cond-form

Now that Nia knows how to tokenize a text with `cl-ppcre`, she needs a function that calculates the average size of the words of the text.

```
(defun avg(ws &optional (acc 0.0) (sz 0))
    (cond
      ( (and (null ws) (> sz 0))    ;;1st clause condition
        (/ acc sz)                  ;;action
      );;end of 1st clause
      ( (null ws)                   ;;2nd clause condition
            0.0)                    ;;action
      ( (consp ws)                  ;; 3rd clause condition
        (avg (cdr ws)               ;; action
```

```
          (+ (length (car ws)) acc) ;;
          (+ sz 1)) )
     );;end cond
);;end defun
```

The cond-form is used to control conditional execution, based on a set of clauses. Each clause has a condition followed by a sequence of actions. Lisp starts with the top clause, and proceeds in decending order. It executes the first clause whose condition produces a value different from NIL. For instance, the first clause condition is the `(and (null ws) (> sz 0))` predicate. Since the predicate `(null ws)` returns `T` (true) for an empty `ws`, this condition tests whether the `ws` list is empty `and sz` is greater than zero. Immediately after a true condition, the cond form executes the corresponding action, which is `(/ acc sz)` when dealing with the first clause.

The symbol `nil` represents both the empty list and the false boolean value. To make a long story short, the Lisp word for false is nil, which also represents the empty list. To better understand the above snippet, let us follow the execution of the `(avg '("a" "bb" "ccc"))` call.

```
user call: (avg ws= ("a" "bb" "ccc") acc= 0.0 sz= 0.0)
  1st and 2nd clauses fail, becase ws is not nil
3rd clause call: (avg ws= ("bb" "ccc") acc= 1.0 sz= 1.0)
  1st and 2nd clauses fail because ws is not nil
3rd clause call: (avg ws= ("ccc") acc= 3.0 sz= 2.0)
  1st and 2nd clauses fail together
3rd clause call: (avg ws= nil acc= 6.0 sz=3.0
    1st clause succeeds because ws= nil and sz>3
1st clause executes (/ acc sz) producing the result.
```

3.10 format

In order to write objects onto a data stream, Lisp programmers use the following macro:

`(format` *stream pattern data...*`)`

Since space is limited, let us restrict the study of output to three situations, writing on the screen, superseding an existing file, and appending text to an existing file.

- Writing on the screen.

```
(defun greeting(name)
     (format t "Hello, ~a~%" name))
```

- Create a new file, supersede if it exists.

```
(defun greeting(name file)
     (with-open-file
          (out file :direction :output
            :if-exists :supersede)
     (format out "Hello, ~a~%" name)))
```

- Append to an existing file.

```
(defun greeting(name file)
     (with-open-file
          (out file :direction :output
            :if-exists :append)
     (format out "Hello, ~a~%" name)))
```

3.11 Loops

A common method of simplifying a complex problem is to divide it into parts that share the same structure or the same type. Then a program calls up each part to be resolved, and the results from each part are combined to form the solution to the original problem. This method of problem solving is called recursion. Let us consider the mathematical definition of factorial.

$$n! \text{ is defined as } \begin{cases} n = 0 \rightarrow & 1 \\ n > 0 \rightarrow & n * (n-1)! \end{cases}$$

It is straightforward to translate this mathematical definition to Lisp.

```
(defun !(n)
  (cond ( (= n 0) 1)
        ( (> n 0) (*  (! (- n 1)) n))
  )
)
```

The above function divides the problem of finding the factorial of n into two parts: calculating the factorial of (n-1) and multiplying the result by n. The successive decrementation of the input n will lead the calculation to the base case, where n=0 and the solution is 1. After the recursive call, the computer must return to the second clause of the cond form, in order to perform the multiplication. This is inefficient, and should be avoided whenever possible. Here is a new definition, that performs the multiplication before calling the factorial function on the simplified problem:

```
(defun factorial(n &optional (acc 1))
  (cond ( (= n 0) acc)
        ( (> n 0) (factorial (- n 1) (* n acc)))
  )
)
```

When the recursive call does not leave any task behind, the Lisp compiler can perform what is called tail call optimization and thus avoid the saving of information for completing the calculation.

The loop form is another way of repeating an operation. The interested reader who wishes to study Loop and other Lisp structures, should consult the book Practical Common Lisp[9].

Chapter 4

An space efficient implementation of WAM

One of the principal problems through which this thesis is motivated is the technique of textmining. Textmining can be seen as the analysis of text, whereby information is obtained in high quality from texts by use of this technique. As will be presented in chapter 5.1 the Web finds itself overloaded with information, some of which is relevant and essential for research, whereas other information can be considered inadequate or useless for any kind of research. The extraction of necessary information is by any standard an arduous task. Thereby, the established techiques should be used as an aid to web users in order that they perform more efficient searches.

One particular technique that can be used is textmining. In order to perform this task two steps are necessary:

Tagger: The system that identifies grammatical and semantic categories of evaluated sentences, through the addition of a tag. Normally, the understanding of text should be founded upon data extract from large corpora, such as the Brown Corpora.

Syntactic Analysis/Parsing: This is the process use for analysing an input sequence in order to determine its grammatical structure in accordance with a particular formal grammar.

In this thesis the author used the library tagger made available on Quicklisp for applying tags onto each word in the setence. Instructions as to the installation of the library

can be found in section 3.3.

In the following one finds an example of the program that uses a library `tagger`.

```
;; File: tag.lisp

(ql:quickload :tagger)
(use-package :tdb)
(use-package :tag-analysis)

(defun tkn(token-stream)
  (multiple-value-bind
    (token tag)
    (next-token token-stream)
    (if token
      (cons (string-downcase (format nil "~a" tag)) token)
      nil)))

(defun my-tagger (str)
  (with-input-from-string (char-stream str)
   (loop with token-stream =
      (make-ts char-stream (make-instance 'tag-analysis))
        for xt = (tkn token-stream)
        then (tkn token-stream)
        until (not xt)
        collect xt)))

(defun vocabulary(str)
  (let* ((s1 (my-tagger str))
         (s2 (remove-duplicates s1 :test #'equal))
         (s (sort s2 (lambda(x y)(string>= (car x)(car y))))))
     (loop for (x . y) in s
       do (format t "~a([~a|S], S).~%" x y)) ))
```

Let us run the above program in Lisp.

```
CL-USER(1): (load "tag.lisp")
To load "tagger":
  Load 1 ASDF system:
    tagger
; Loading "tagger"
.
T
CL-USER(2): (vocabulary "I saw the man on the hill with the telescope.")
;;; Reading /home/junia/quicklisp/dists/quicklisp/software/
tagger-20140713-git/data/brown/suffix.trie ... Done.
;;; Reading lexicon from /home/junia/quicklisp/dists/quicklisp/software/
tagger-20140713-git/data/brown/lexicon.txt ...
;;; Done reading lexicon.
;;; Reading HMM from /home/junia/quicklisp/dists/quicklisp/software/
tagger-20140713-git/data/brown/brown.hmm
vbd([saw|S], S).
ppss([I|S], S).
nn([telescope|S], S).
nn([hill|S], S).
nn([man|S], S).
in([with|S], S).
in([on|S], S).
at([the|S], S).
NIL
CL-USER(3):
```

One observes that given the sentence *"I saw the man on the hill with the telescope."* the tags were attributed based on the Brown Corpora.

After the addition of the tags, the next step is to perform syntactic analysis. In order to perform the task, the author proposed the development of an inference engine

implemented in a functional language based on Prolog. In the following sections some of the implemented inference engine parts will be described. The complete code is presented in Appendix I.

Let us consider the program `synbench`, that uses the implemented inference engine, for performing the syntactic analysis.

```
;; File: synbench
$ n(animal(cat,s), [cat|S], S).
$ n(animal(mouse,p), [mice|S], S).
$ det(def(the), [the|S], S).
$ vbz(vt(chase), [chases|S], S).

$ vp(vb(V, N), S0, S2) :- vbz(V, S0, S1), np(N, S1, S2).
$ np(np(none,N), S0, S1) :- n(N, S0, S1).
$ np(np(D,N), S0, S2) :- det(D, S0, S1), n(N, S1, S2).
$ sentence(s(N, V), S0, S2) :- np(N, S0, S1), vp(V, S1, S2).

$ len([], L, L) :- !.
$ len([_|Resto], L, Resp) :- plus(L1, L, 0.1), len(Resto, L1, Resp).

$ wrt(I, FF) :- mod(0, I, 100000), !, write(FF), nl.
$ wrt(I, FF).

$ parse(Text, 0, S, S) :- !, sentence(Sentence, Text, _),
                          write(Sentence), nl.
$ parse(Text, I, S, Res) :- one,
                        sentence(FF, Text, RR), nt,
                        I1 is I-1,
                        wrt(I, FF),
                        len(RR, 0, L),
                        S1 is S+L,
                        parse(Text, I1, S1, Res).
```

```
$ teste(N) :- cputime(X1),
              parse([the, cat, chases, mice, sentence], N, 0, F),
              cputime(X2),
              minus(Time, X2, X1),
              write('Time '),
              write(Time), nl.
```

In `synbench` the dictionary tags were added by the library `tagger`. For example, the word `cat` is a noun and can be classified as `animal(cat,s)`.

```
$ n(animal(cat,s), [cat|S], S).
$ n(animal(mouse,p), [mice|S], S).
$ det(def(the), [the|S], S).
$ vbz(vt(chase), [chases|S], S).
```

In the following, some grammar rules are proposed.

Verb Phrases: verb + noun phrases

Noun Phrases: noun

Noun Phrases: determiner + noun

Sentence: noun phrases + verb phrases

```
$ vp(vb(V, N), S0, S2) :- vbz(V, S0, S1), np(N, S1, S2).
$ np(np(none,N), S0, S1) :- n(N, S0, S1).
$ np(np(D,N), S0, S2) :- det(D, S0, S1), n(N, S1, S2).
$ sentence(s(N, V), S0, S2) :- np(N, S0, S1), vp(V, S1, S2).
```

In `synbench` given a quantity `N` of iterations `teste(N)` one adds the parser in loop. The act of performing tests in a loop is important, due to the fact that when one evaluates texts extracted from the Web a larger volumn of sentence have to be analysed. Therefore, the program needs to support a larger quantity of input.

In order to test the program it is necessary to load the WAM code source and run it.

```
CL-USER(1): (load "wam17.lisp")
T
CL-USER(2): (mini:wam)
mini-Wam
```

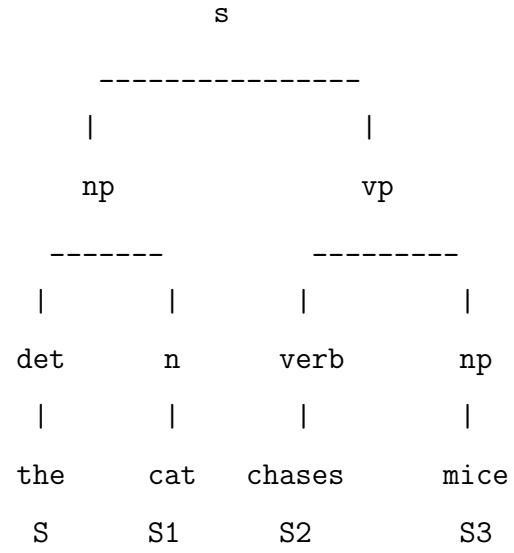
For the loading of `synbench` let us use `sult`.

```
| ?- sult(synbench).
T
Yes
no More
```

Now, let us execute `teste(N)`:

```
| ?- teste(1000).
s(np(def(the),animal(cat,s)),vb(vt(chase),np(none,animal(mouse,p))))
Time 0.015999973
Yes
More : d
no More
```

The output for `teste(N)` is a syntactic tree of the evaluated sentence.



If the reader wishes to exit WAM, the command is:

```
| ?- halt.
Yes
no More
Bye!
#<PACKAGE "COMMON-LISP-USER">
CL-USER(3): (exit)
```

4.1 Concepts

Two concepts were essential in the development of this thesis, those being: function and relation.

Function: map an element from the domain in one and only one element of the counter domain.

Relation: is a map that can relate an element from the domain with one or more elements from the counter domain.

4.2 Enunciate the thesis

This thesis proposed the unification of the function and relation concepts as programming foundations.

Currently, there exist a implemenations which produce this union, as for example: Gambol [10], Paiprolog [11], Racklog [12], Schemelog [13], Kanren [14] among others. However, these implementations are not complete or do not function for large programs, or be it they do not optimize the use of time and space.

There exists those implementations that were written in non-mathematical languages. When the subject becomes a thesis these implementations presented two problems.

1. The implementations became obsolete very quickly, due to the fact that the language becomes obsolete.

2. Mathematics imposes the condition that the axioms set along with the inference rules need to be minimal, that is with the least cardinality possible. Besides this,

an axiom can not be deduced from another. The non-mathematical languages are not submitted to the constraints.

In this manner, this thesis proposes the development of a system for logic programming that possesses the following features, which are not presented in the complete sense in currently available systems.

- The system should be implement in a language that does not become obsolete, that is it needs to guarantee that the user can work for decades on a compiler and still be functional.
- The execution of program will be performed in space complexity, whenever possible.
- The system will not be a toy system, in the sense that it is possible to write practical programs with it.
- The system will be complete, that is it will have all the recourses found in logical languages.
- Try to reach a velocity close to that of logic program systems implemented in non-mathematical languages.

4.3 Performing Tests

For comparison purposes, a program similar to `synbench` was written in Paiprolog. The objective behind this comparison is to verify the performance of both programs.

```
;; File: tpai.lisp

(ql:quickload :paiprolog)
(defpackage :teupai
(:use :cl :paiprolog))
(in-package :teupai)

(<-- (len ?s ?n) (length ?s 0 ?n))
```

```
(<-- (length () ?n ?n) !)
(<-  (length (?x . ?y) ?n ?s)
     (is (+ ?n 1) ?m)
     (length ?y ?m ?s))

(<-- (n (animal cat s) (cat . ?x) ?x))
(<-  (n (animal mouse p)(mice . ?x) ?x))

(<-- (det (def the) (the . ?x) ?x))

(<-- (vbz (vt chase) (chases . ?x) ?x))

(<-- (np (np none ?nm) ?x ?y) (n  ?nm ?x ?y))
(<-  (np (np ?d ?nm) ?x ?y) (det ?d ?x ?s1) (n ?nm ?s1 ?y))

(<-- (vp (vb ?v ?nm) ?x ?y) (vbz ?v ?x ?s1) (np ?nm ?s1 ?y))

(<-- (s (s ?nm ?vp) ?x ?y) (np ?nm ?x ?s1) (vp ?vp ?s1 ?y))

(<-- (zim 0 ?s ?s) ! (s ?sen (the cat chases mice)() )
     (lisp (format t "~a~%" ?sen)))
(<- (zim ?n ?s ?r)
    (is (- ?n 1) ?n1)
    (s ?sen (the cat chases mice) ()) !
    (length ?sen 0 ?len)
    (is (+ ?s ?len) ?s1)
    (zim ?n1 ?s1 ?r))
```

Executing the program `tpai.lisp`.

```
CL-USER(1): (load "tpai.lisp")
To load "paiprolog":
```

```
  Load 1 ASDF system: paiprolog
; Loading "paiprolog"
T
CL-USER(2): (in-package :teupai)

#<PACKAGE "TEUPAI">
TEUPAI(3): (prolog-collect (?x) (s ?x (the cat chases mice)()))
((S (NP (DEF THE) (ANIMAL CAT S)) (VB (VT CHASE) (NP NONE (ANIMAL MOUSE P)))))
```

Let's put the parser that evaluates the sentence into a loop.

```
TEUPAI(4): (prolog-collect (?x) (zim 1000 0.0 ?x))
(S (NP (DEF THE) (ANIMAL CAT S))
            (VB (VT CHASE) (NP NONE (ANIMAL MOUSE P))))
```

In Table 4.1 the time spent in seconds is presented, for performing a syntactic analysis of the sentence in a loop with 1,000, 3,000, 5,000, 7,000, 50,000, 500,000 and 1,000,000 iterations. In Paiprolog it was possible to perform a test with up to 5,000 iterations. However, above this rate overflow occurred.

Table 4.1: Benchmark

	1,000	**3,000**	**5,000**	**7,000**	**50,000**	**500,000**	**1,000,000**
Paiprolog	0.022	0.051	0.078	Overflow	Overflow	Overflow	Overflow
WAM	0.016	0.036	0.080	0.056	0.464	3.920	7.852

As previously presented, Paiprolog executes a quick syntactic analysis. However, as the number of inputs increase, overflow occurs.

4.4 Warren Abstract Machine - WAM

In 1983 David Warren developed an abstract machine for executing Prolog programs, see [15]. The proposed machine was called the Warren Abstract Machine (WAM) and is used as standard for the development of Prolog compilers.

As presented in chapter 1this thesis proposes:

1. To write an interface engine which is robust based on the Warren Abstract Machine. This engine, besides the traditional Prolog cut, should also accept the deterministic predicates declaration.

2. Implement an interface between relations and functions.

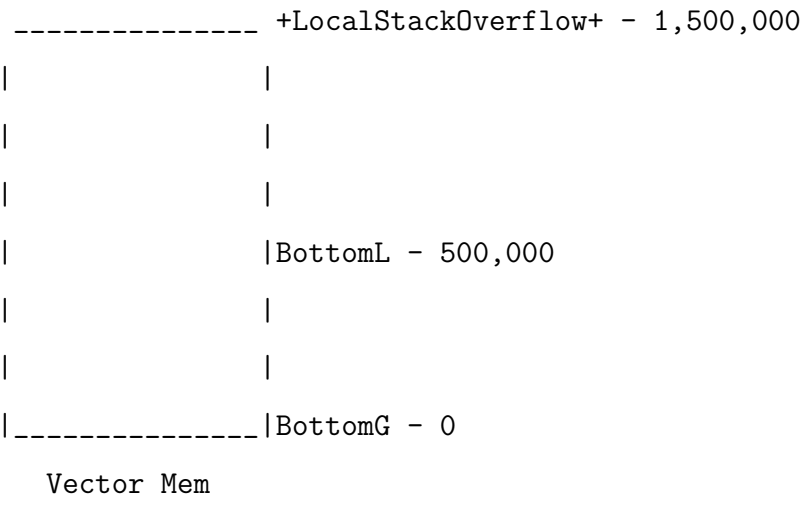
3. To write a grammar in such a manner that the syntax of the proposed engine interface remains as close as possible to the traditional Prolog syntax.

The engine interface was implemented in Common Lisp, with its starting point centered on the books of [15], [16] e [17].

4.4.1 Registers

The WAM has the following memory management:

- The local stack or control stack, updated upon return from determinist calls and upon backtracking;
- The global stack or copy stack, updated upon backtracking;
- The trail updated upon backtracking



The local stack and the copy stack are both implemented in the `Mem` vector. Since `Mem` is a vector that can accept any kind of data, its content must be retrieved

with `(svref Mem i)`. In order to store data on Mem, one should use the command `(setf (svref Mem i) Val)`.

The copy stack goes from `BottomG` to `BottomL` in `Mem`. `BottonL` indicates the bottom of the local stack in `Mem`, and `+LocalStackOverflow+` delimits its top.

```
(defconstant BottomG (the fixnum 0))
(defconstant BottomL (the fixnum 500000))
(defconstant +LocalStackOverflow+ (the fixnum 1500000))
(defvar Mem (make-array +LocalStackOverflow+ :initial-element 0 ))
```

The trail stack is implemented in a separate vector, `trailMem`. In Lisp, a vector containing only fixnums is vastly more efficient than a general vector, that may receive any data structure. Therefore, one has defined the trail stack `trailMem` as a vector of fixnums.

```
(defconstant BottomTR (the fixnum 0))
(defconstant +TrailOverflow+ 1000000)
(defvar trailMem
  (make-array +TrailOverflow+ :initial-element 0
      :element-type 'fixnum))
```

Definition of the variables:

```
(defvar TR (the fixnum 0))              ;;top of the trail stack
(defvar *LocalPointer* (the fixnum 0)) ;;top of the local stack
(defvar *GPointer* (the fixnum 0))      ;;top of the copy stack
(defvar CP (the fixnum 0))              ;;current continuation
(defvar CL (the fixnum 0))              ;;mother block
(defvar Cut_pt (the fixnum 0))          ;;specific cut
(defvar BL (the fixnum 0))              ;;previous choice in the local stack
(defvar BG (the fixnum 0))              ;;top of the copy stack before
                                        ;;allocation of the top stack.
(defvar PC (the fixnum 0))              ;;current goal
(defvar PCE (the fixnum 0))             ;;current environment
```

A variable `X` is stored as `(V . 0)`. A functor `fp(a,b)` is stored as `((|fp|) |a| |b|)`. A copy of this functor is stored as `((|fp| . t) |a| |b|)`.

```
(defmacro functorCopy (des largs) `(cons (cons (car ,des) t) ,largs))
```

4.4.2 Unification

The unification algorithm verifies if there exists a union between the terms of each predicate. To perform this task it is necessary to inform the two predicates and check for one of the following conditions:

1. The two terms are equal constants.
2. The two terms are variable.
3. The first term is a variable.

Should unification not occur it is necessary to undo the links made during unification. The unification function should be well designed, due to the great number of calls made through it.

```
(defun unif (x y)
   (cond
     ((eql x y) t)
     ((var? y)
      (if (var? x)
        (if (= (cdr x) (cdr y)) t (bindv y x))
        (bindte (cdr y) x)))
     ((var? x) (bindte (cdr x) y))
     ((or (atom x) (atom y)) (throw 'impossible 'fail))
     ((let ((b (copy? y)) (dx (pop x)) (dy (pop y)))
       (if (eq (functor dx) (functor dy))
         (do* ((ax x (cdr ax))
               (vx (mval (car ax))(mval (car ax)))
               (vy (pop y) (pop y)))
```

```
            ((null ax))
            (unif vx (if b (mval vy)
              (ultimate vy (Environment *LocalPointer*))) ))
          (throw 'impossible 'fail))))))
```

4.4.3 The Cut

The cut is a predefined predicate in that when found, it forces the next rewind point. In general it is used to minimize the time used in the search of sub-objectives that do not lead to the principal objective. Besides this, the cut economizes on memory, as it avoids unnecessary rewind points being stored on the stack. The cut syntax is represented by the character `!`.

In the following example we use `$` to introduce the Prolog syntax from Lisp.

```
$ gcd(X, Y, Y) :- =(X, Y), !.
$ gcd(X, Y, G) :- <(X,Y), !,
                  minus(Y1, Y, X),
                  gcd(X, Y1, G).
$ gcd(X, Y, G) :- >(X,Y),
                  gcd(Y, X, G).
```

4.4.4 Deterministic Predicates

The implemented WAM accepts the deterministic predicates declaration. Thus, it is easier to say that the predicate gcd has only one solution, than to introduce cuts to the predicate. One observes in the following example a new definition of the predicate.

```
$ gd(x, y, ?g).
$ gd(x, y, g) :- =(x, y), univ(g, x).
$ gd(x, y, g) :- <(x,y),
                 -(y1, y, x),
                 gd(x, y1, g).
$ gd(x, y, g) :- >(x,y),
                 gd(y, x, g).
```

In order to reach a suitable comparison, the author tested two predicates: `gcd` and `gd`. `looptest(N)` was set to perform `N` calls to `gcd` and `gd`. In Table 4.2 a time comparison in seconds is presented, from the execution of `gcd`, with the traditional Prolog cut, and the `gd`, defined as a deterministic predicate. One observes that when the predicate is declared as deterministic its execution time is optimized.

Table 4.2: Cut x Determinist Predicate

	100	**1,000**	**10,000**
gcd	0.004	0.084	4.344
gd	0.0	0.004	0.104

4.4.5 Infix Syntax

The implemented WAM accepts prefixed notation, this is normally the notation Lisp programmers most use. The infix notation is also accepted, this notation is commonly adopted by Prolog programmers.

Note the definition of infix operators.

```
(defparameter *infix-ops*
  '((([ list match ]) ({ elts match }) (|(| nil match |)|))
    ((not not unary) (~ |neg| unary) )
    ((*) (/))
    ((+) (-))
    ((<) (>) (<=) (>=) (=) (/=))
    ((and) (& and) (^ and))
    ((or) (\| or))
    ((=>))
    ((<=>))
    ((|,|)))
  "A list of lists of operators, highest precedence first.")
```

The conversion of the infix expression for the Lisp notation is made in `->prefix`.

```
(defun ->prefix (infix)
  "Convert an infix expression to prefix."
  (loop
     (when (not (length>1 infix))
       (RETURN (first infix)))
     (setf infix (reduce-infix infix))))
```

In Prolog negation is represented by -. Consider a predicate defined as `taco(X) :- X * -1.` the compiler when finding the symbol - did not know if it should be treated as an operation of negation or of subtraction. In this manner, the function `change_infix` was defined in order to change the syntax, every time it encounters a - preceded by a (or a mathematical operator.

```
(defun change_infix (s)
  (cond ((null s) s)
        ((null (cdr s)) s)
        ((and (eq (car s) '|(|)
              (eq (cadr s) '-)))
         (cons '|(| (cons '\~ (change_infix (cddr s))))))
        ((and (eq (car s) '*)
              (eq (cadr s) '-))
         (cons '* (cons '\~ (change_infix (cddr s)))))
        ((and (eq (car s) '/)
              (eq (cadr s) '-))
         (cons '/ (cons '\~ (change_infix (cddr s)))))
        (t (cons (car s) (change_infix (cdr s ))))))

(defun prefx(s) (list
     (->prefix (change_infix
         (if (and (consp s)
                  (eq (car s) '-))
            (cons '~ (cdr s))
```

```
            s))) ))

(defun reduce-infix (infix)
  "Find and reduce the highest-precedence operator."
  (dolist (ops *infix-ops*
     (error "Bad syntax for infix expression: ~S" infix))
     (let* ((pos (position-if
                    #'(lambda (i) (assoc i ops)) infix
                    :from-end (eq (op-type (first ops))
                    'MATCH)))
            (op (when pos (assoc (elt infix pos) ops))))
       (when pos
        (RETURN
          (case (op-type op)
           (MATCH (reduce-matching-op op pos infix))
           (UNARY (replace-subseq infix pos 2
                    (list (cons (op-name op) 1)
                          (elt infix (+ pos 1)))))
           (BINARY (replace-subseq infix (- pos 1) 3
                     (list (cons (op-name op) 2)
                           (elt infix (- pos 1))
                           (elt infix (+ pos 1)))))))))))
```

With WAM accepting the infix notation, the predicates now do not need to be defined preceded by the symbol $.

```
fat(0,1) :- !.
fat(N, F) :- X1 is N-1,
             fat(X1,F1),
             F is N*F1.
```

Performing a test:

```
CL-USER(1): (load "wam17")
T
CL-USER(2): (mini:wam)
mini-Wam

| ?- consult(mim2).
Yes
no More

| ?- fat(5,X).
X = 120
no More
```

4.4.6 Consult function

In loading the archive with the predicates by use of `consult` the program performs a reading of the archive content and will call functions to generate efficient Lisp code from the predefined predicates.

```
(defun |consult| (f)
  (let* ((filename (format nil "~a" (value f)))
         (code (rd_file filename)))
     (loop for c in code do (addit c))))
```

In the case where the user prefers to use prefix notation, one only needs to use the function `sult` in order to load the predicates.

4.4.7 Generated Lisp code

Consider the Fibonacci definition.

For the definition:

(DEF (FB N X) (< X 2) (UNIV N 1))

The generated Lisp code was:

```
(LAMBDA (X)
             (OR
              (BLOCK ET
                (PROGN
                 (WHEN (NOT (< X 2)) (RETURN-FROM ET NIL))
                 (LET ((N 1))
                   N)))))
```

For the definition:

(DEF (FB N X) (- X1 X 1) (- X2 X 2) (FB N1 X1) (FB N2 X2) (+ N N1 N2))

The generated code was:

```
(LAMBDA (X)
        (OR
         (BLOCK ET
           (PROGN
            (WHEN (NOT (< X 2)) (RETURN-FROM ET NIL))
            (LET ((N 1))
              N)))
         (BLOCK ET
           (LET ((X1 (- X 1)))
             (LET ((X2 (- X 2)))
               (LET ((N1 (FB X1)))
                 (LET ((N2 (FB X2)))
                   (LET ((N (+ N1 N2)))
                     N)))))))))
```

4.5 Tests performed with WAM

The performed tests were extracted from P-99: Ninety-Nine Prolog Problems.

P01. Find the last element of a list

```
% my_last(X,L) :- X is the last element of the list L

my_last(X,[X]).
my_last(X,[_|L]) :- my_last(X,L).

| ?- my_last(X,[1,2,3,4,5]).
X = 5
```

P02. Find the last but one element of a list

```
% last_but_one(X,L) :- X is the last but one element of the list L

last_but_one(X,[X,_]).
last_but_one(X,[_,Y|Ys]) :- last_but_one(X,[Y|Ys]).

| ?- last_but_one(X, [1,2,3,4]).
X = 3
More : d
no More
```

P03. Find the K'th element of a list.

```
% element_at(X,L,K) :- X is the K'th element of the list L

element_at(X,[X|_],1).
element_at(X,[_|L],K) :- >(K,1), K1 is K - 1, element_at(X,L,K1).

| ?- element_at(X, [a,b,c,d], 3).
X = c
More : d
no More
```

P04. Find the number of elements of a list.

```
% my_length(L,N) :- the list L contains N elements

my_length([],0).
my_length([_|L],N) :- my_length(L,N1), N is N1 + 1.

| ?- my_length([a,b,c,d], X).
X = 4.0
More : d
no More
```

P05. Reverse a list.

```
% my_reverse(L1,L2) :- L2 is the list obtained from L1 by reversing
%    the order of the elements.

my_reverse(L1,L2) :- my_rev(L1,L2,[]).
my_rev([],L2,L2) :- !.
my_rev([X|Xs],L2,Acc) :- my_rev(Xs,L2,[X|Acc]).

| ?- my_reverse([a,b,c,d,e], X).
X = [e,d,c,b,a]
no More
```

P06. Find out whether a list is a palindrome

```
palin(List1):- my_reverse(List1,List2),
               compare(List1,List2).
compare([],[]):- !.
compare([X|List1],[X|List2]):- compare(List1,List2).
compare([X|List1],[Y|List2]):- !, fail.
```

```
| ?- palin([a,m,a]).
Yes
More : d
no More
| ?- palin([a,b,c]).
no More
```

P07. Flatten a nested list structure.

```
% flatten(L1,L2) :- the list L2 is obtained from the list L1 by flattening

flatten([], []) :- !.
flatten([L|Ls], FlatL) :- !,
                          flatten(L, NewL),
                          flatten(Ls, NewLs),
                          app(NewL, NewLs, FlatL).
                          flatten(L, [L]).

| ?- flatten([a,[b,c],[[d],[],[e]]],Xs).
Xs = [a,b,c,d,e]
no More
```

P08. Eliminate consecutive duplicates of list elements.

```
% compress(L1,L2) :- the list L2 is obtained from the list L1 by
% compressing repeated occurrences of elements into a single
% copy of the element.

compress([],[]).
compress([X],[X]).
compress([X,X|Xs],Zs) :- compress([X|Xs],Zs).
```

```
compress([X,Y|Ys],[X|Zs]) :- neq(X,Y),
                             compress([Y|Ys],Zs).

| ?- compress([a,a,b,c,d,e,f,f], L).
L = [a,b,c,d,e,f]
More : d
no More
```

P09. Pack consecutive duplicates of list elements into sublists.

```
% pack(L1,L2) :- the list L2 is obtained from the list L1 by packing
% repeated occurrences of elements into separate sublists.
% transfer(X,Xs,Ys,Z) Ys is the list that remains from the list Xs
% when all leading copies of X are removed and transfered to Z

pack([],[]).
pack([X|Xs],[Z|Zs]) :- transfer(X,Xs,Ys,Z), pack(Ys,Zs).

transfer(X,[],[],[X]).
transfer(X,[Y|Ys],[Y|Ys],[X]) :- neq(X,Y).
transfer(X,[X|Xs],Ys,[X|Zs]) :- transfer(X,Xs,Ys,Zs).

| ?- pack([a,a,b,c,d,e,f,f], L).
L = [[a,a],[b],[c],[d],[e],[f,f]]
More :d
no More
```

P10. Run-length encoding of a list

```
% encode(L1,L2) :- the list L2 is obtained from the list L1 by run-length
% encoding. Consecutive duplicates of elements are encoded as terms [N,E],
% where N is the number of duplicates of the element E.
```

```
encode(L1,L2) :- pack(L1,L),
                 transform(L,L2), !.
transform([],[]) :- !.
transform([[X|Xs]|Ys],[[N,X]|Zs]) :- my_length([X|Xs],N),
                                     transform(Ys,Zs).

| ?- encode([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4.0,a],[1.0,b],[2.0,c],[2.0,a],[1.0,d],[4.0,e]]
More : d
no More
```

P11. Modified run-length encoding

```
% encode_modified(L1,L2) :- the list L2 is obtained from the list L1 by
% run-length encoding. Consecutive duplicates of elements are encoded
% as terms [N,E], where N is the number of duplicates of the element E.
% However, if N equals 1 then the element is simply copied into
% the output list.

encode_modified(L1,L2) :- encode(L1,L),
                          strip(L,L2).
strip([],[]).
strip([[1.0,X]|Ys],[X|Zs]) :- strip(Ys,Zs).
strip([[N,X]|Ys],[[N,X]|Zs]) :- >(N,1.0),
                                 strip(Ys,Zs).
| ?- encode_modified([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4.0,a],b,[2.0,c],[2.0,a],d,[4.0,e]]
More : d
no More
```

P12. Decode a run-length compressed list.

```
% decode(L1,L2) :- L2 is the uncompressed version of the
% run-length encoded list L1.

decode([],[]).
decode([X|Ys],[X|Zs]) :-  not_is_list(X),
                          decode(Ys,Zs).
decode([[1,X]|Ys],[X|Zs]) :- decode(Ys,Zs).
decode([[N,X]|Ys],[X|Zs]) :- >(N,1),
                             N1 is N - 1,
     decode([[N1,X]|Ys],Zs).

| ?- decode([[4,a],[1,b],[2,c],[2,a],[1,d],[4,e]],L).
L = [a,a,a,a,b,c,c,a,a,d,e,e,e,e]
More : d
no More
```

P13. Run-length encoding of a list (direct solution)

```
% encode_direct(L1,L2) :- the list L2 is obtained from the list L1 by
% run-length encoding. Consecutive duplicates of elements are encoded
% as terms [N,E], where N is the number of duplicates of the element E.
% However, if N equals 1 then the element is simply copied into the
% output list.
% count(X,Xs,Ys,K,T) Ys is the list that remains from the list Xs
% when all leading copies of X are removed. T is the term [N,X],
% where N is K plus the number of X's that can be removed from Xs.
% In the case of N=1, T is X, instead of the term [1,X].

encode_direct([],[]).
encode_direct([X|Xs],[Z|Zs]) :- count(X,Xs,Ys,1,Z), encode_direct(Ys,Zs).

count(X,[],[],1,X).
```

```
count(X,[],[],N,[N,X]) :-  >(N,1).
count(X,[Y|Ys],[Y|Ys],1,X) :- neq(X,Y).
count(X,[Y|Ys],[Y|Ys],N,[N,X]) :- >(N,1),
                                  neq(X,Y).
count(X,[X|Xs],Ys,K,T) :- K1 is K + 1,
                          count(X,Xs,Ys,K1,T).

| ?- encode_direct([a,a,a,a,b,c,c,a,a,d,e,e,e,e],X).
X = [[4,a],b,[2,c],[2,a],d,[4,e]]
More : d
no More
```

P14. Duplicate the elements of a list

```
% dupli(L1,L2) :- L2 is obtained from L1 by duplicating all elements.

dupli([],[]):-!.
dupli([X|Xs],[X,X|Ys]) :- dupli(Xs,Ys).

| ?- dupli([a,b,c,c,d],X).
X = [a,a,b,b,c,c,c,c,d,d]
More : d
no More
```

P15. Duplicate the elements of a list agiven number of times

```
% duplic(L1,N,L2) :- L2 is obtained from L1 by duplicating all
% elements N times.

duplica(L1,N,L2) :- duplic(L1,N,L2,N).

duplic([],_,[],_).
```

```
duplic([_|Xs],N,Ys,0) :- duplic(Xs,N,Ys,N).
duplic([X|Xs],N,[X|Ys],K) :- >(K,0),
                             K1 is K - 1,
     duplic([X|Xs],N,Ys,K1).

| ?- duplica([a,b,c,c,d],4,X).
X = [a,a,a,a,b,b,b,b,c,c,c,c,c,c,c,c,d,d,d,d]
More : d
no More
```

P16. Drop every N'th element from a list

```
% drp(L1,N,L2) :- L2 is obtained from L1 by dropping every N'th element.

drop(L1,N,L2) :- drp(L1,N,L2,N).

drp([],_,[],_).
drp([_|Xs],N,Ys,1) :- drp(Xs,N,Ys,N).
drp([X|Xs],N,[X|Ys],K) :-  >(K,1),
                           K1 is K - 1,
    drp(Xs,N,Ys,K1).

| ?- drop([a,b,c,d,e,f,g,h,i,k],2,X).
X = [a,c,e,g,i]
More : d
```

P17. Split a list into two parts

```
% split(L,N,L1,L2) :- the list L1 contains the first N elements
% of the list L, the list L2 contains the remaining elements.

split(L,0,[],L).
```

```
split([X|Xs],N,[X|Ys],Zs) :- T is N > 0,
                             N1 is N - 1,
     split(Xs,N1,Ys,Zs).

| ?- split([a,b,c,d,e,f,g,h,i,k],3,L1,L2).
L1 = [a,b,c]
L2 = [d,e,f,g,h,i,k]
More : d
no More
```

P18. Extract a slice from a list

```
% slice(L1,I,K,L2) :- L2 is the list of the elements of L1 between
% index I and index K (both included).

slice([X|_],1,1,[X]).
slice([X|Xs],1,K,[X|Ys]) :- T is K > 1,
                            K1 is K - 1,
    slice(Xs,1,K1,Ys).
slice([_|Xs],I,K,Ys) :- T is I > 1,
                        I1 is I - 1,
K1 is K - 1,
slice(Xs,I1,K1,Ys).

| ?- slice([a,b,c,d,e,f,g,h,i,k],3,7,L).
L = [c,d,e,f,g]
More : d
no More
```

P19. Rotate a list N places to the left

```
% rotate(L1,N,L2) :- the list L2 is obtained from the list L1 by
```

```
% rotating the elements of L1 N places to the left.

rotate(L1,N,L2) :-  >(N,0),
                    my_length(L1,NL1),
    mod(N1, N, NL1),
    rotate_left(L1,N1,L2).

rotate(L1,N,L2) :-  =(N,0),
                    my_length(L1,NL1),
    mod(N1, N, NL1),
    rotate_left(L1,N1,L2).

rotate(L1,N,L2) :-  <(N,0),
                    my_length(L1,NL1),
    mod(NN1, N, NL1),
    N1 is NL1 + NN1,
    rotate_left(L1,N1,L2).

rotate_left(L,0,L).
rotate_left(L1,N,L2) :- >(N,0),
                        split(L1,N,S1,S2),
app(S2,S1,L2).
```

P20. Remove the K'th element from a list.

```
% remove_at(X,L,K,R) :- X is the K'th element of the list L; R is the
% list that remains when the K'th element is removed from L.

remove_at(X,[X|Xs],1.0,Xs).
remove_at(X,[Y|Xs],K,[Y|Ys]) :- >(K,1.0),
                                 K1 is K - 1.0,
        remove_at(X,Xs,K1,Ys).
```

```
| ?- remove_at(X, [a,b,c,d,e], 2, R).
X = b
R = [a,c,d,e]
More : d
no More
```

P21. Insert an element at a given position into a list

```
% insert_at(X,L,K,R) :- X is inserted into the list L such that it
% occupies position K. The result is the list R.

insert_at(X,L,K,R) :- remove_at(X,R,K,L).

| ?- insert_at(j, [a,b,c,d,e], 2, R).
R = [a,j,b,c,d,e]
More : d
no More
```

P22. Create a list containing all integers within a given range.

```
% range(I,K,L) :- I <= K, and L is the list containing all
% consecutive integers from I to K.

range(I,I,[I]).
range(I,K,[I|L]) :-  <(I, K),
                     I1 is I + 1,
     range(I1,K,L).

| ?- range(4, 9, L).
L = [4,5,6,7,8,9]
More : d
no More
```

Extra The N Queens Problem

```
queen(N,R) :- ranges(1,N,Ns),
              queens(Ns,[],R).

ranges(N,N,[N]) :- !.
ranges(M,N,[M|Ns]) :- M1 is M + 1,
                      ranges(M1,N,Ns).

queens([],R,R).
queens(Unplaced,Safe,R) :- del(Q,Unplaced,U1),
                           not_attack(Q,Safe),
                           queens(U1,[Q|Safe],R).

not_attack(Q,Safe) :-  attack(Q,1,Safe), !, fail.
not_attack(_1,_2).

attack(X,N,[Y|_]) :-  X is Y + N, !.
attack(X,N,[Y|_]) :-  Y is X + N, !.
attack(X,N,[_|Ys]) :- N1 is N + 1,
                      attack(X,N1,Ys).

tqueen(X, R) :- queen(X, R), fail.
tqueen(X, R) :- queen(X, R), !.

bqueen :- cputime(X1), tqueen(11, Q), cputime(X2),
          write(Q), nl,
          minus(Time, X2, X1),
          write(Time), nl.

| ?- bqueen.
```

```
[10,8,6,4,2,11,9,7,5,3,1]
4.908
Yes
no More
```

Chapter 5

Future Work

5.1 Web pollution

In the not so distant past, knowledge, data and segments of information were stored mainly in written documents, such as books, journals, reports, etc. Since publication was an expensive and difficult endeavor, people would strive to make a careful selection of that material considered to be worth printing. This selection was realized through the discretion of referees, reader opinions, editorial policy, boards of education, and the like.

The advent of the Internet, World Wide Web and Social Networks eliminated all this quality filtering that was put in place due to the limited resources imposed on the publication of low value information. Therefore, in contemporary society, all individuals and organizations are able to publish ideas, data, news, theories, media, etc. This brings two problems to readers and other consumers of information. The first one is focus: It is difficult to find information related to the topic of interest in the vast volumes of text available on the Internet. The second problem is assessment: The person who browses the web is unable to distinguish valuable information from gossip, quackery or hype.

There are many factors that create difficulties concerning data retrieval from the web, along with the selection of useful information and noiseless transmission of knowledge. One can use many metaphors to classify these factors:

- Pollution. Here the web is compared to an open body of water that one uses both as supply and sewage disposal.

- Garbage, junk and debris. The term "junk" is largely used when refering to information posted on the various bulletin boards of the Internet. Confer: Junk mail.
- Noise. The Internet is compared to a communication channel, and useless information is associated with noise. This metaphor is often used to introduce many of the information theory concepts, such as entropy.

The Internet has become a library of infinit content, which has modify society. The *web* directs one towards a torrent of information and communication channels that integrate millions a people to thousands to an enourmous content base.

A patient goes to the doctor and he is told: *You should not eat eggs because they are bad for your health.* The patient quickly answers: *But I read that this is not true. A current study shows that eggs are beneficial.* In order to justify this ficticious patient, the reader of this work will find on the Internet statements of the following degree:

> Recent studies have grouped the fats into different types, there are those that not only are good for the organism, but are essential for it to function well and bring health benefits, besides not harming colesterol. This is the case of egg fat.

In summary, the Internet has become a resorvoir where one takes information for consumption, and at the same time, flushes cultural sewage. It is necessary to treat the sewage and filter the water.

Jackob Nielsen [1] compares useless information on the web to water pollution. If this metaphor is in any way of value, one should use information treatment methods that are analogous to the processes that make water fit for end-users.

The treatment of sewage, better still the sewage water, can be summarized in the following steps:

1. Removal of large solids and send for the protection of the treatment units.
2. Decantation and Sedimentation
3. Remove the toxic pollutants or non-biodigradable
4. Desinfection: Elimination of microorganisms.

The first thing that comes to mind when one speaks of filtering cultural information is *Reddit*. Reddit is a decantation filter, in other words, it carries out decantation and sedimentation, in order to filter out useless information. Simply put, the Reddit user votes to decide upon those articles which will go to landfill of oblivion, against those of interest to the common reader. After all, Reddit is open to any registered user voting on the promotion of an article. The problem with Reddit is that every user can vote. In chapter 2, the author will discuss the voting process, in order that it truly acts as a decantation filter.

5.1.1 Text mining

Text mining is the process of automatically deriving information and possibly knowledge from text. To achieve this goal, one typically needs to devise similarities and trends through means such as statistical pattern learning.

A traditional search engine does little more than look for the presence of query terms in a text under scrutiny. A small improvement is to deal with synonyms and context. However, it is obvious that an ideal text mining tool should be able to understand what is written. Therefore, it is surprising that natural language processing is rarely found in search engines. There are two reasons for this fact. The first one is that Artificial Intelligence is still not a mature technology. The second one is that natural language processing algorithms consume a lot of processing power. Even if these reasons are enough for limiting nlp to simpler tasks like determining the context of a query, they should not prevent an engineer or a physician from using deeper analysis for determining whether a text contains the information that he or she needs.

At this point, it is necessary to define the meaning of the verb "understand" in the context of natural language processing. Linguists agree that language understanding has four phases.

1. Tokenizing. Here the text is broken into words and other tokens. Besides separating one word from another, the tokenizer makes the first attempt at classifying each token into grammatical and semantic classes. This process is called tagging.

2. Parser. The natural language processor builds structures that represent the position

of one word in relation to another. It also builds the so called syntactic tree, that shows the relationship between groups of words.

3. Semantic. The natural language processor assigns meanings to words and syntactic trees. These meanings can be expressed as logic clauses, a knowledge representation language, or even a simple form of a human language, like Esperanto.

4. Pragmatic. Here the text miner tries to discover the intention of the person posing the query.

Artificial Intelligence is not even close to executing these four stages of language comprehension with the proficiency of a human being. However, a modern natural language processor may be good enough to be useful, as one will be seen in this work.

Chapter 6

Conclusion

According to the objectives outlined for this thesis and agreed upon by the thesis committee, that being namely to conclude upon the following:

1. Objective 1: Implement a theorem prover based on the Warren Abstract Machine. It is necessary to use a computer language that can accomodate any new technological advance, such as parallelism, persistence, memoization, etc. Its syntax and semantics must be based on mathematics, so it will not suffer any substantial change from one decade to another. Finally, it must have a powerful scheme of encapsulation to prevent the work of a contributer having deleterious effects on the efforts of another researcher. The only language with all these attributes is Common Lisp.

   Conclusion: This objective was reached in chapter 4, with the implementation of a theorem prover based on the Warren Abstract Machine. The bringing to fruition of the objective laid out herein was a collective effort at the Artificial Intelligence Laboratory of Faculty of Electrical Engineering, Federal University of Uberlândia. The author of this thesis was responsible for implementing the interface between relations and functions, for the unification and prover's inherent infix notation. The thesis collaborator Mauro Honorato, was responsible for mapping the interface engine in functional language as well as develop an escape function for relation control. In the appendix of this thesis, the snippets of code produced by Mauro Honorato are indicated by a comment.

2. Objective 2: The syntax of the proposed inference engine should be as close as

possible to the tradicional syntax of Prolog.

Conclusion: This work's second objective was to bring the language in the prover closer to the traditional syntax of Prolog. This objective was reached in section 4.4.5 in line with the examples given. It is necessary to finalize this item by adding more Prolog predicates.

3. Objective 3: The interface should accept declarations of deterministic predicates as well as non-deterministic predicates.

   Conclusion: As presented in the examples of section 4.4.4 the interface accepts declarations of deterministic and non-deterministic predicates.

4. Objective 4: The inference engine should optimize the use of time and space.

   Conclusion: In section 4.3 by performing trial runs, it was noted that the inference engine optimized the use of space and time. Tests were performed with other inference engines (PaiProlog), which obtained good results when tested with a large volume of data.

5. Objective 5: It is also necessary to create a repository management system so that a researcher can have instant access to all previous work. The technology for such a repository also exists, and it is called quicklisp. Therefore, the project presented herein will be available in quicklisp.

   Conclusion: The task of providing WAM implemented in quicklisp is underway. The author entered in contact with Zach Beane, responsible for quicklisp, in order to upload the archives and create the distribution licences.

With the above objectives concluded, other works will look to make additions over the short to medium term, with contributions, such as research from the author as well as research guidance, namely:

- Accomplish the junction of the Condorcet-List Theorem as in the Rasch model. An implementation of the Rasch model, as presented in this work was implemented by the author. As future works the author will continue on this line of research applying the Rasch model combined with Condorcet theorem in order to process

natural language of texts extracted from the Web. The underlying objective of such work is, when concluded, to assist Internet users in filtering out relevant content, which is personalized, with a lower error rate along with a lower return of information outside of the search focus, at the moment of carrying out an Internet search.

Chapter 7

Appendix I - WAM

```
;; File: wam21.lisp

(defpackage :mini (:use :cl)
    (:export :wam :exec))

(in-package :mini)

(declaim (optimize (speed 3) (debug 0) (safety 0) (space 0))
 (ftype (function (* *) t) bindte)
         (ftype (function (*) t) reduce-infix)
         (ftype (function (* * *) t) reduce-matching-op)
         (ftype (function () t) read_prompt)
 (ftype (function () t) banner)
         (ftype (function (&optional *) t) rchnsep)
         (ftype (function (&optional *) t) read_code_tail)
 (ftype (function (*) t) readProvePrintLoop)
 (ftype (function (* &optional *) t) read_args)
 (ftype (function (* &optional *) t) xread_args)
 (ftype (function (* &optional *) t) read_term)
 (ftype (function (* &optional *) t) xread_term)
 (ftype (function (* &optional *) t) read_pred)
 (ftype (function (* *) t) ultimate)
 (ftype (function (*) t) vvarp)
 (ftype (function (* *) t) cop)
 (ftype (function (*) t) pr_det)
 (ftype (function (*) t) genvar)
 (ftype (function (*) t) ult)
 (ftype (function () t) load_pc)
 (ftype (function () t) cont_eval)
 (ftype (function () t) load_A2)
```

```
 (ftype (function (*) t) shallow_backtrack)
 (ftype (function () t) backtrack)
 (ftype (function (*) t) pr_choice)
 (ftype (function (*) t) pr)
 (ftype (function (*) t) pr2)
 (ftype (function (*) t) lisploop)
 (ftype (function (* *) t) unif)
        (ftype (function (*) t) impl)
        (ftype (function (*) t) expl)
 (ftype (function () t) printvar)
 (ftype (function (* *) t) writesl)
 (ftype (function (* *) t) writesf)
        (ftype (function (*) t) write1)
 (ftype (function () t) read_terme)
 (ftype (function (*) t) write1))

;; A solution to prefix conversion:
(defparameter *infix-ops*
  '((([ list match ]) ({ elts match }) (|(| nil match |)|))
    ((not not unary) (~ |neg| unary) )
    ((*) (/))
    ((+) (-))
    ((<) (>) (<=) (>=) (=) (/=))
    ((and) (& and) (^ and))
    ((or) (\| or))
    ((=>))
    ((<=>))
    ((|,|)))
  "A list of lists of operators, highest precedence first.")

(defun length>1(x) (> (length x) 1))

(defun replace-subseq (sequence start length new)
  (nconc (subseq sequence 0 start) (list new)
         (subseq sequence (+ start length))))

(defun op-token (op) (first op))
(defun op-name (op) (or (second op) (first op)))
(defun op-type (op) (or (third op) 'BINARY))
(defun op-match (op) (fourth op))
(defun length=1(s) (= (length s) 1))

(defun ->prefix (infix)
  "Convert an infix expression to prefix."
  (loop
     (when (not (length>1 infix))
```

```
       (RETURN (first infix)))
     (setf infix (reduce-infix infix))))

(defun change_infix (s)
  (cond ((null s) s)
((null (cdr s)) s)
((and (eq (car s) '|(|)
      (eq (cadr s) '-))
(cons '|(| (cons '\~ (change_infix (cddr s)))))
((and (eq (car s) '*)
      (eq (cadr s) '-))
 (cons '* (cons '\~ (change_infix (cddr s)))))
       ((and (eq (car s) '/)
      (eq (cadr s) '-))
 (cons '/ (cons '\~ (change_infix (cddr s)))))
(t (cons (car s) (change_infix (cdr s ))))  ))

(defun prefx(s) (list (->prefix
       (change_infix
(if (and (consp s)
(eq (car s) '-))
   (cons '~ (cdr s))
   s))) ))

(defun reduce-infix (infix)
  "Find and reduce the highest-precedence operator."
  (dolist (ops *infix-ops*
     (error "Bad syntax for infix expression: ~S" infix))
     (let* ((pos (position-if
              #'(lambda (i) (assoc i ops)) infix
                :from-end (eq (op-type (first ops))
                'MATCH)))
              (op (when pos (assoc (elt infix pos) ops))))
        (when pos
         (RETURN (case (op-type op)
           (MATCH (reduce-matching-op op pos infix))
           (UNARY (replace-subseq infix pos 2
                   (list (cons (op-name op) 1)
                         (elt infix (+ pos 1)))))
           (BINARY (replace-subseq infix (- pos 1) 3
                   (list (cons (op-name op) 2)
                         (elt infix (- pos 1))
                         (elt infix (+ pos 1)))))))))))
(defun op(s) (car s))
(defun arg1(s) (cadr s))
(defun arg2(s) (caddr s))
```

```
(defun remove-commas (exp)
  "Convert (|,| a b) to (a b)."
  (cond ((eq (op exp) '|,|)
         (nconc (remove-commas (arg1 exp) )
                (remove-commas (arg2 exp))))
        (t (list exp))))

(defun handle-quantifiers(x) x)
(defun function-symbol? (x)
  (and (symbolp x) (not (member x '(and or not ||)))
       (alphanumericp (char (string x) 0))))

(defun reduce-matching-op (op pos infix)
  "Find the matching op (paren or bracket) and reduce."
  ;; Note we don't worry about nested parens because we search :from-end
  (let* ((end (position (op-match op) infix :start pos))
         (len (+ 1 (- end pos)))
         (inside-parens (remove-commas
 (->prefix (subseq infix (+ pos 1) end)))))
    (cond ((not (eq (op-name op) '|(|)) ;; handle {a,b} or [a,b]
           (replace-subseq infix pos len
                           (cons (op-name op) inside-parens))) ; {set}
          ((and (> pos 0)  ;; handle f(a,b)
                (function-symbol? (elt infix (- pos 1))))
           (handle-quantifiers
            (replace-subseq infix (- pos 1) (+ len 1)
    (cons (elt infix (- pos 1))
  inside-parens))))
          (t (replace-subseq infix pos len
   (first inside-parens))))))

;; mini-Wam
;; boot
;;

(defparameter *keep-going* nil)

(defun exec(str)
  (setf *keep-going* nil)
  (in-package :mini)
  (let ( (oldin *standard-input*)
(oldout *standard-output*)
        (fstr(make-array '(0) :element-type 'base-char
 :fill-pointer 0 :adjustable t)))
    (with-input-from-string (s (format nil "~a " str))
```

```
      (setf *standard-input* s)
      (with-output-to-string
  (z fstr)
(setf *standard-output* z)
      (handler-case (lisploop (read_code_tail))
  (error(e) (format t "toplevel error: ~a~%" e)  ))))
      (setf *standard-input* oldin
    *standard-output* oldout)
       (in-package :cl-user)
      fstr))

(defun wam ()
  (setf *keep-going* t)
  (banner)
  (in-package :mini)
  (readProvePrintLoop (read_prompt))
  (in-package :cl-user))

(defun read_prompt ()
  (terpri)
  (format t "| ?- ")
  (finish-output)
  (read_code_tail))

(defun banner ()
  (dotimes (i 2) (terpri))
  (format t "mini-Wam~%")
  (dotimes (i 2) (terpri)))

(defun l ()
  (format t "Back to mini-Wam top-level~%")
  (readProvePrintLoop (read_prompt)))


;; mini-wam
;; reader

(defvar *lvar nil)
(set-macro-character #\% (get-macro-character #\;))

(defun rch (&optional (stream *standard-input*))
  (do ((ch (read-char stream) (read-char stream)))
      ((char/= ch #\Newline) ch)))

(defun ignore-to-eol(xch &optional (stream *standard-input*))
  (do ((ch xch (read-char stream)))
```

```
      ((char= ch #\Newline) (rchnsep stream))))

(defun commts(ch &optional (stream *standard-input*))
  (if (char= #\% ch) (ignore-to-eol (read-char  stream) stream)
      ch))

(defun rchnsep (&optional (stream *standard-input*))
  (do ((ch (rch stream) (rch stream)))
      ( (and (char/= ch #\space)
     (char/= ch #\tab)) (commts ch stream)  ) ))

(defun spcial (ch) (char= ch #\_))
(defun alphanum (ch) (or (alphanumericp ch)
 (spcial ch)))
(defun valdigit (ch) (digit-char-p ch))


(defun read_frac(ch x acc &optional (stream *standard-input*))
  (cond((digit-char-p (peek-char nil stream))
(read_frac (read-char stream)
   (/ x 10.0)
   (+ acc (* (valdigit ch) x)  )
   stream))
       (t (+ acc (* (valdigit ch) x))  )))

(defun read_fr(ch x acc &optional (stream *standard-input*))
  (declare (ignorable ch))
  (if (not (digit-char-p (peek-char nil stream)))
      (progn (unread-char  #\. stream)  0.0)
      (read_frac (read-char stream) x acc stream)))

;; Improvements: Add double precision to numbers
(defun read_number (sign ch &optional (stream *standard-input*))
  (do ((v (valdigit ch) (+ (* v 10) (valdigit (read-char stream)))))
      ((not (digit-char-p (peek-char nil stream)))
       (if (char= (peek-char nil stream) #\.)
   (* sign (+ v  (read_fr (read-char stream)
  0.1 0.0 stream) ))
          (* v sign))) ))

(defun implode (lch) (intern (map 'string #'identity lch)))

(defun read_atom (ch &optional (stream *standard-input*))
  (cond  ((char= ch #\-)  '-)
         ((char= ch #\*)  '*)
         ((char= ch #\/)  '/)
```

```
        ((char= ch #\+)  '+)
 ((and (char= ch #\<)
       (char= (peek-char nil stream) #\()) '|lt|)
 ((and (char= ch #\>)
       (char= (peek-char nil stream) #\()) '|gt|)
 ((and (char= ch #\=)
       (char= (peek-char nil stream) #\()) '|eqn|)
 ((and (char= ch #\<)
       (char= (peek-char nil stream) #\=))
  (read-char stream) '|<=|)
        ((and (char= ch #\>)
       (char= (peek-char nil stream) #\=))
     (read-char stream) '|>=|)
        (t (do ((lch (list ch) (push (read-char stream) lch)))
((not (alphanum (peek-char nil stream)))
 (implode (reverse lch)))  ))))

(defun read_at (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((char= (peek-char nil stream) #\') (read-char stream)
       (implode (reverse lch)))))

(defun do_l (x) (if (atom x) x
    (list '(\. . 2)
  (car x)
  (do_l (cdr x)))))

(defun read_string (ch &optional (stream *standard-input*))
  (do ((lch (list (char-int ch))
            (push (char-int (read-char stream)) lch)))
      ((char= (peek-char nil stream) #\") (read-char stream)
       (do_l (reverse lch)))))

(defun read_var (type_var ch &optional (stream *standard-input*))
  (let ((v (read_atom ch stream)))
(cons type_var
  (position v
            (if (member v *lvar)
                *lvar
                 (setq *lvar (append *lvar (list v)))))) ))

(defun read_simple (ch &optional (stream *standard-input*))
  (cond
    ((and (eq ch #\n) (upper-case-p (peek-char nil stream)))
     (read_var 'N (read-char stream) stream))
   ((or (spcial ch) (upper-case-p ch)) (read_var 'V ch stream))
```

```
   ((digit-char-p ch) (read_number 1 ch stream))
   ((and (char= ch #\-) (digit-char-p (peek-char nil stream)))
    (read_number -1 (read-char stream) stream ))
   ((char= ch #\") (read_string (read-char stream) stream))
   ((char= ch #\') (read_at (read-char stream) stream))
   ((char= ch #\#) (unread-char ch stream) (read stream))
   (t (read_atom ch stream))))

(defun read_fct (ch &optional (stream *standard-input*))
(let ((fct (read_simple ch stream))
      (c (rchnsep stream)))
    (if (char= c #\()
(let ((la (read_args (rchnsep stream) stream)))
  (cons (list fct ) la))
      (progn (unread-char c stream) fct))))

(defun read_args (ch &optional (stream *standard-input*))
  (let ((arg (read_term ch stream)))
    (if (char= (rchnsep stream) #\,)
(cons arg (read_args (rchnsep stream) stream))
      (list arg))))

(defun read_factor (ch &optional (stream *standard-input*))
  (cond
   ((or (spcial ch) (upper-case-p ch)) (read_var 'V ch stream))
   ((digit-char-p ch) (read_number 1 ch stream))
   ((char= ch #\") (read_string  (read-char stream) stream))
   ((char= ch #\') (read_at (read-char stream) stream))
   ((char= ch #\#) (unread-char ch stream) (read stream))
   ((char= ch #\() '|(|)
   ((char= ch #\)) '|)|)
   (t (read_atom ch stream))))

(defun read_expr (ch &optional (stream *standard-input*))
  (let ((arg (read_factor ch stream)))
    (let ( (next-ch (rchnsep stream)))
    (cond ( (eql next-ch #\,)
    (list arg) )
          ( (eql next-ch #\.)
   (unread-char next-ch stream)
    (list arg))
  (t (unread-char next-ch stream)
     (cons arg (read_expr (rchnsep stream) stream)) ))) ))

(defparameter *cns* '(\.)) ;;(the fixnum 2)))
```

```
(defun read_list (ch &optional (stream *standard-input*))
  (if (char= ch #\])
      ()
    (let ((te (read_term ch stream)))
      (case (rchnsep stream)
    (#\, (list *cns* te
               (read_list (rchnsep stream) stream)))
    (#\| (prog1 (list *cns* te
                 (read_term (rchnsep stream) stream))
   (rchnsep stream)))
    (#\] (list *cns* te nil))))))

(defun read_term (ch &optional (stream *standard-input*))
  (if (char= ch #\[)
      (read_list (rchnsep stream) stream)
      (read_fct ch stream)))

(defun read_tail (ch &optional (stream *standard-input*))
  (let ((tete (read_pred ch stream)))
    (if (equal tete '(|one|))
(let* ((solvendum (read_pred (rchnsep stream) stream))
       (chr (rchnsep stream)))
  (cond ( (char= chr #\.) (list tete solvendum '(|nt|)))
( (char= chr #\,)
 (cons tete (cons solvendum
  (cons '(|nt|)
(read_tail
 (rchnsep stream) stream)))))
       (t (unread-char chr stream)
          (cons tete (read_tail (rchnsep stream) stream)))))
      (let ((chr (rchnsep stream)))
        (cond ( (char= chr #\.) (list tete))
    ( (char= chr #\,)
     (cons tete (read_tail (rchnsep stream) stream)))
    (t (unread-char chr stream)
       (cons tete (read_tail (rchnsep stream) stream)))))))))

(defun vname(v)
  (cond ((not (vvarp v)) v)
        (t (let ((x (symbol-name v)))
     (intern (subseq x 1 (length x)))))))

;; Lisp uses uppercase ids, and Prolog uses lowercase.
;; (mkLispID '|fib|) converts |fib| to FIB, where
;; |fib| may be replaced by any Prolog id.
(defun mkLispID(n)
```

```
  (intern (string-upcase
      (symbol-name  n  ))))

;; Read Prolog Deterministica
(defun ximplode (lch) (intern (string-upcase
       (map 'string #'identity lch) )))
(defun xread_atom (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((not (alphanum (peek-char nil stream)))
       (ximplode (reverse lch)))))

(defun xread_at (ch &optional (stream *standard-input*))
  (do ((lch (list ch) (push (read-char stream) lch)))
      ((char= (peek-char nil stream) #\')
       (read-char stream) (implode (reverse lch)))))

(defun xread_string (ch &optional (stream *standard-input*))
  (do ((lch (list (char-int ch)) (push (char-int (read-char stream)) lch)))
      ((char= (peek-char nil stream) #\") (read-char stream)
       (coerce (reverse lch) 'string))))

(defun xread_simple (ch &optional (stream *standard-input*))
  (cond
   ((digit-char-p ch) (read_number 1 ch stream ))
   ((char= ch #\-) (read_number -1 (read-char stream) stream ))
   ((char= ch #\") (xread_string (read-char stream) stream))
   ((char= ch #\') (xread_at (read-char stream) stream))
   ((char= ch #\#) (unread-char ch stream) (read stream))
   (t (xread_atom ch stream))))

(defun xread_fct (ch &optional (stream *standard-input*))
  (let ((fct (xread_simple ch stream)) (c (rchnsep stream)))
    (if (char= c #\()
(let ((la (xread_args (rchnsep stream) stream)))
  (cons (list fct (length la)) la))
      (progn (unread-char c stream) fct))))

(defun xread_args (ch &optional (stream *standard-input*))
  (let ((arg (xread_term ch stream)))
    (if (char= (rchnsep stream) #\,)
(cons arg (xread_args (rchnsep stream) stream))
      (list arg))))

(defun xread_list (ch &optional (stream *standard-input*))
  (if (char= ch #\])
      ()
```

```
    (let ((te (xread_term ch stream)))
      (case (rchnsep stream)
    (#\, (cons te (xread_list (rchnsep stream) stream)))
    (#\| (prog1 (cons te (read_term (rchnsep stream) stream))
   (rchnsep stream)))
    (#\] (cons te nil))))))

(defun xread_term (ch &optional (stream *standard-input*))
  (if (char= ch #\[)
      (xread_list (rchnsep stream) stream)
      (xread_fct ch stream)))

(defun xread_pred (ch &optional (stream *standard-input*))
  (let ((nom (xread_atom ch stream)) (c (rchnsep stream)))
    (if (char= c #\()
(cons nom (xread_args (rchnsep stream) stream))
      (progn (unread-char c stream) (list nom)))))

(defun xread_tail (ch &optional (stream *standard-input*))
  (let ((tete (xread_pred ch stream)))
    (if (char= (rchnsep stream) #\.)
(list tete)
      (cons tete (xread_tail (rchnsep stream) stream)))))
;;end read Prolog deterministica

;; Tools for Lisp-Prolog communication:

;; (fix xs) converts all ids in a predicate to
;; Lisp ids. For instance, if xs= (|fib| |n| |?f|)
;; to (FIB N ?F).
(defun fix(xs)
  (mapcar #'mkLispID xs ))

(defun sfx(s) (car (last s)))
(defun rmsfx(s) (butlast s))
(defun pfx(s) (car (cdr s)))
(defun rmpfx(s) (cons (car s) (cddr s)))

(defun vvarp(v)
  (and  (symbolp v)
    (eql (aref (symbol-name v) 0) #\?)))

(defun modedeclarationp(s)
  (cond ((null s) nil)
((and (consp s) (vvarp (car s))) t)
(t (modedeclarationp (cdr s))) ))
```

```
(defun notStructure(x)
  (not (and (consp x)
    (not (eq (car x) 'V)))))

(defun hasNoStructure(pred)
  (or (atom pred)
      (eq (car pred) '|is|)
      (every #'notStructure pred)))

(defun onlyLocals(cl)
  (every #'hasNoStructure cl))

(defun changePred(p)
  (cond ((null p) p)
        ((atom p) p)
((eq (car p) 'V)
 (cons 'L (changePred (cdr p))))
(t (cons (changePred (car p)) (changePred (cdr p)))) ))

(defun changeClause(cl)
  (mapcar #'changePred cl))

(defun fixVar(cl)
  (if (onlyLocals cl)
      (changeClause cl)
      cl))

(defun read_clause (ch &optional (stream *standard-input*))
  (let ((tete (read_pred ch stream))
(neck (rchnsep stream)))
    (if (char= neck #\.)
(cond (  (modedeclarationp tete)
         (list 'mdef (fix  tete  )))
      (t (list tete)))
(let ((nneck (read-char stream)))
  (cond (  (and (char= neck #\:) (char= nneck #\-)
(get (mkLispId (car tete)) 'mod) )
   (let ((tail (xread_tail (rchnsep stream) stream)))
     (cons 'def  (cons (fix tete) tail))))
(t (cons tete
 (read_tail (rchnsep stream)
    stream))))  )) ))

(defun processIs(l)
   (cond
```

```
    ( (and l   (consp (cdr l))
       (consp (cadr l))
       (equal (car (cadr l)) '|is|))
 (cons (cons '|is|
      (cons (car l) (cdr (cadr l))))
       (processIs (cddr l))))
    ( l (cons  (car l) (processIs (cdr l))))))

(defun processCut (l)
  (when l (cons (if (or (eq (caar l) '!)
(eq (caar l) '|nt|)
   )
  `(,(caar l) ,(length *lvar) ,@(cdar l))
  (car l))
      (processCut (cdr l)))))

(defun read_code_cl (&optional (stream *standard-input*))
  (let ((*lvar ()))
    (let ((x (read_clause (rchnsep stream) stream)))
      (cond ((member (car x)  '(pdef! pdef sdef! sdef mdef def))  x)
            (t  (cons (length *lvar)
      (processCut (processIs x)))))))

(defun read_code_tail (&optional (stream *standard-input*))
  (setq *lvar ())
  (let ((x (read_tail (rchnsep stream) stream)))
     (cons (length *lvar) (append (processCut (processIs x))
 (list '(|true|))))))

(defun listNom(X)
  (cond ( (and (consp X)  (eq (car X) 'V) ) X)
( (member X '(- + * /)) X)
(t (list X))))

(defun read_pred (ch &optional (stream *standard-input*))
  (let ((nom (read_simple ch stream))
(c (rchnsep stream)))
    (cond ( (equal nom '|is|)
    (cons nom (prefx (read_expr c stream) ) ))
          ( (char= c #\()
            (cons nom (read_args (rchnsep stream) stream)))
  ( (char= c #\,) (listNom nom))
          (t (unread-char c stream) (listNom nom)))))

(defun rdc(stream)
  (handler-case (read_code_cl stream)
```

```
    (error (e) (declare (ignorable e))  nil)))

(defun rd_file(fileName)
  (with-open-file (s fileName)
    (loop for l = (rdc s) then
 (rdc s)
       while l
       collect l)))

;; Begin Mauro Honorato

;; Aqui comecam os macros que convertem de Prolog deterministica
;; para Lisp. Utilizando a capacidade da Lisp de retornar multiple
;; values para simular as os predicados da Prolog.
;; Cada clausula eh colocada dentro de um block et. Se algum
;; predicado falha, temos um return-from et.
;; Um block et, portanto, faz o papel de um (and p1 p2 p3 ...).

;; Testa se o predicado eh um operador aritmetico
(defun arithmeticp(p)
  (and (consp p)
    (member (car p) '(+ - * /))))

;; Identificadores para constantes.
(defun cnsname(c)
  (cond ( (symbolp c) c)
( (and (consp c) (equal (car c) 'quote))
  (cadr c))
((numberp c)
 (intern (format nil "~a" c))) ))

;; Testa se temos um operador logico
(defun ispred(p)
  (and (consp p)
       (member (car p) '(> < >= <= = eq eql equal))))

(defun apply-mode(xs ys)
  (cond ((and (null xs) (null ys)) nil)
((null xs) (error "wrong arity: ~a~%" ys))
((null ys) (error "wrong arity: ~a~%" xs))
((equal (car xs) (car ys))
 (cons (car xs) (apply-mode (cdr xs) (cdr ys))))
((equal (car xs) '-)
 (cons (car ys) (apply-mode (cdr xs) (cdr ys))))
((and (equal (car xs) '+)
      (symbolp (car ys)))
```

```
 (cons (list 'quote (car ys))
       (apply-mode (cdr xs) (cdr ys))))))


;; chain converte Prolog para Lisp.
(defun chain(hd tail)
  (cond ((null tail) `(values ,@(cdr hd)   ))
((arithmeticp (car tail))
 `(multiple-value-bind
      ,(mapcar #'cnsname (cdr (car tail)))
      (values (,(car (car tail))
,@(cddr (car tail)))
      ,@(cddr (car tail)))
    (declare (ignorable ,@(mapcar #'cnsname
       (cdr (car tail)))))
    ,(chain hd (cdr tail))))
((ispred (car tail))
`(progn (when (not ,(car tail)) (return-from et nil))
   (return-from vel ,(chain hd (cdr tail))   )))
( (and (consp (car tail))
       (equal (car (car tail)) 'univ))
 `(let (( ,(cadr (car tail)) ,(caddr (car tail))))
    ,(chain hd (cdr tail))))
((and (consp (car tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
 `(multiple-value-bind
  ,(mapcar #'cnsname (cdr (car tail)))
      ,(apply-mode (get (car (car tail)) 'mod)
   (car tail))
      (declare (ignorable ,@(mapcar #'cnsname
    (cdr (car tail)))))
       ,(chain hd (cdr tail))) ))  )

(defun funChain(hd tail)
  (cond ((null tail)  (cadr hd))
((arithmeticp (car tail))
 `(let (( ,(cadr (car tail))
  ,(cons (car (car tail)) (cddr (car tail)) )))
          ,(funChain hd (cdr tail))))
((ispred (car tail))
 `(progn (when (not ,(car tail)) (return-from et nil))
 ,(funChain hd (cdr tail))   ))
( (and (consp (car tail))
       (equal (car (car tail)) 'univ))
 `(let (( ,(cadr (car tail)) ,(caddr (car tail))))
```

```
    ,(funChain hd (cdr tail))))
((and (consp (car tail))
      (null (cdr tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
  (cons (car (car tail)) (cddr (car tail))))
((and (consp (car tail))
      (symbolp (car (car tail)))
      (get (car (car tail)) 'mod))
 `(let (( ,(cadr (car tail))
 ,(cons (car (car tail)) (cddr (car tail)) )))
        ,(funChain hd (cdr tail))))  ))

(defun mkFun(clause)
  `(block et ,(funChain (car clause) (cdr clause))))

(defun mkAND(clause)
  `(block et ,(chain (car clause) (cdr clause))))

(defun shw(x)
  (format t "Code= ~a~%" x) x)

;; Funcao check que verifica a ocorrencia
;; de apenas um mais na clausula
(defmacro clauseSetFunctor(s) `(caar ,s))

(defun checkFunMode(m)
  (let (( md (get (clauseSetFunctor m) 'mod)))
  (and (eql (cadr md) '+)
       (every (lambda(x) (eql x '-)) (cddr md)) )))

(defun mkdef(args clauses &optional (funMode (checkFunMode (car clauses))))
   (if funMode
   (list 'lambda (cdr args)
 (cons 'or
      (loop for x in clauses collect
   (mkFun x))))
   (list 'lambda args
 `(declare (ignorable ,@args))
 (cons 'block (cons 'vel
      (loop for x in clauses collect
   (mkAnd x))))))  )

(defun ck(nm args p clauses)
  (if (and (consp p) (equal args p)
   (every #'symbolp p))
```

```
      clauses
      (error "(def ~a ~a|~a...)?" nm args p)))

(defun getfun(s) (car s))
(defun getargs(s)  (cdr s))

;; macro para converter Prolog deterministica para Lisp.
(defmacro def (&rest clause)
  (let* ((hd (car clause))
 (fun (getfun hd))
 (args (getargs hd)))
  (setf (get fun 'clauses)
     `(,@(get fun 'clauses)
 ,(ck fun (get fun 'vs) args clause )))
  `(setf (symbol-function ',fun)
 ,(mkdef args (get fun 'clauses)) )))

(defun vmode(v)
  (cond ((vvarp v) '+)
(t '-)))

(defun argnames(vs)
  (loop for v in vs collect (vname v)))

(defun declaremodes(vs)
  (loop for v in vs collect (vmode v)))

;; Inicializa predicados da Prolog deterministica.
;; Armazena os modos, argumentos e as clausulas.

(defmacro mdef(hd)
  (let ((fun (getfun hd))
(args (getargs hd)))
    `(progn (setf (get ',fun 'clauses) nil)
         (setf (get ',fun 'vs) (argnames ',args))
         (setf (get ',fun 'mod) (cons ',fun (declaremodes ',args))))))
;; End Mauro Honorato

;; mini-wam
;; Machine
;; I. Registers
;;

(defconstant BottomG (the fixnum 0))
(defconstant BottomL (the fixnum 500000))
(defconstant +LocalStackOverflow+ (the fixnum 1500000))
```

```
;; Mem implements the local stack and the heap.
(defvar Mem (make-array +LocalStackOverflow+
:initial-element 0 ))

;; The trail stack is implemented in a separate vector.
(defconstant BottomTR (the fixnum 0))
(defconstant +TrailOverflow+ 1000000)
(defvar trailMem
  (make-array +TrailOverflow+ :initial-element 0
      :element-type 'fixnum))

;; The arguments of a predicate are stored in xArgs.
(defvar xArgs (make-array 50 :initial-element 0))

(defvar TR (the fixnum 0)) ;;top of the trail stack
(defvar *LocalPointer*     ;;top of the local stack
  (the fixnum 0))
(defvar *GPointer*         ;;top of the copy stack
  (the fixnum 0))
(defvar CP (the fixnum 0))     ;;current continuation
(defvar CL (the fixnum 0))     ;;mother block
(defvar Cut_pt (the fixnum 0)) ;;specific cut
(defvar BL (the fixnum 0))     ;;last choice point
(defvar BG (the fixnum 0))
(defvar PC (the fixnum 0))     ;;current goal
(defvar PCE (the fixnum 0))    ;;current environment
(defvar Duboulot)

;; Environment part of the action block allocation
;; in the local stack:
;; BL field -- previous choice in the local stack
;; BP field -- set of clauses in the remaining choices
;; TR field -- top of the trail
;; BG field -- top of the copy stack before allocation
;;             of the top stack. In the pair (BL, BG),
;;             BG designanes the top of the copy stack
;;             associated with the previous choice BL.

(defmacro vset (v i x) `(setf (svref ,v ,i) ,x))
(defmacro trailset(v i x) `(setf (aref ,v ,i) ,x))

(defmacro functorCopy (des largs)
  `(cons (cons (car ,des) t) ,largs))

(defmacro functorDescription (te) `(car ,te))
```

```
(defmacro functor (description) `(car ,description))

(defmacro largs (x) `(cdr ,x))
(defmacro fargs (x) `(cdr ,x))
(defmacro var? (x)
  `(and (consp ,x) (numberp (cdr ,x))))

(defmacro list? (x)
  `(eq (functor (functorDescription ,x)) '\.))

;; II. Local Stack
;;

;; The WAM environment contains [CL CP Cut E]
;;
(defmacro CL (b) `(svref Mem ,b))
(defmacro CP (b) `(svref Mem (1+ ,b)))
(defmacro Cut (b) `(svref Mem (+ ,b 2)))
(defmacro Environment (b) `(+ ,b 3))

(defmacro push_continuation ()
  `(progn (vset Mem *LocalPointer* CL)
  (vset Mem (1+ *LocalPointer*) CP)))

(defmacro push_Environment (n)
  `(let ((top (+ *LocalPointer* 3 ,n)))
     (if (>= top +LocalStackOverflow+)
 (throw 'debord (print "Local Stack Overflow")))
     (vset Mem (+ *LocalPointer* 2) Cut_pt)
     (dotimes (i ,n top) (vset Mem (decf top) (cons 'V top)))))

(defmacro max_Local (nl) `(incf *LocalPointer* (+ 3 ,nl)))

(defmacro TR (b) `(svref Mem (1- ,b)))
(defmacro BP (b) `(svref Mem (- ,b 2)))
(defmacro BL (b) `(svref Mem (- ,b 3)))
(defmacro BG (b) `(svref Mem (- ,b 4)))
(defmacro BCL (b) `(svref Mem (- ,b 5)))
(defmacro BCP (b) `(svref Mem (- ,b 6)))
(defmacro AChoice (b) `(svref Mem (- ,b 7)))

(defun save_args ()
  (dotimes (i (svref xArgs 0)
      (vset Mem (incf *LocalPointer* i) i))
    (declare (fixnum i))
    (vset Mem (+ *LocalPointer* i)
```

```
  (svref xArgs (+ i 1)))))

(defun push_choice ()
  (save_args)
  (vset Mem (incf *LocalPointer*) CP)
  (vset Mem (incf *LocalPointer*) CL)
  (vset Mem (incf *LocalPointer*) *GPointer*)
  (vset Mem (incf *LocalPointer*) BL)
  (vset Mem (incf *LocalPointer* 2) TR)
  (setq BL (incf *LocalPointer*) BG *GPointer*))

(defun push_bpr (resto) (vset Mem (- BL 2) resto))
(defmacro size_C (b) `(+ 7 (AChoice ,b)))

(defun pop_choice ()
  (setq *LocalPointer* (- BL (size_C BL))
BL (BL BL)
BG (if (zerop BL)
       BottomG (BG BL))))

;; III. Copy Stack
;;

(defmacro push_Global (x)
  `(if (>= (incf *GPointer*) BottomL)
       (throw 'debord (print "Heap Overflow"))
     (vset Mem *GPointer* ,x)))

(defmacro adr (v e) `(+ (cdr ,v) ,e))

(defun copy (x e)
  (cond
   ((var? x) (let ((te (ult (adr x e))))
       (if (var? te) (genvar (cdr te)) te)))
   ((atom x) x)
   ((functorCopy (functorDescription x)
  (mapcar (lambda(x) (copy x e)) (fargs x))) )))


(defmacro recopy (x e) `(push_Global (copy ,x ,e)))
(defmacro copy? (te)
  `(cdr (functorDescription ,te)))

;;IV. Trail
;;
```

```
(defmacro pushtrail (x te)
  (declare (ignorable te))
  `(cond ((>= TR +TrailOverflow+)
  (throw 'debord (print "Trail Overflow")))
 ( (trailset trailMem TR ,x) (incf TR) )))

(defmacro poptrail (top)
  `(do () ((= TR ,top))
     (let ((v (aref trailMem (decf TR)) ))
       (vset Mem v (cons 'V v)))))

;; mini-wam
;; utilities

(defmacro nvar (c) `(car ,c))
(defmacro head (c) `(cadr ,c))
(defmacro tail (c) `(cddr ,c))
(defmacro pred (g) `(car ,g))

(defmacro partial? (g) `(get (pred ,g) 'partial))
(defmacro user? (g) `(get (pred ,g) 'def))
(defmacro builtin? (g) `(get (pred ,g) 'evaluable))

(defmacro def_of (g)
  `(get (pred ,g)
      (if (largs ,g)
       (nature (ultimate (car (largs ,g)) PCE)) 'def)))

(defmacro def_part (g)
  `(get (pred ,g) 'partial))

(defun nature (te)
  (cond
   ((var? te) 'def)
   ((null te) 'empty)
   ((atom te) 'atom)
   ((list? te) 'list)
   (t 'fonct)))

(defun getOutputVariables(c)
  (let* (  (fn (car c))
  (mods (cdr (get fn 'mod)))
  (args (cdr c)))
    (loop for x in args
  for m in mods
 when  (equal m '+) collect x)))
```

```
;; Begin Mauro Honorato

(defun mkCall(application)
  (let* (  (fn (car application))
   (mods (cdr (get fn  'mod)))
   (args (cdr application)))
    (cons fn
       (loop for x in args
  for m in mods
  for qx = (if (equal m '+)
       (list 'quote x)
       (list 'value x))
 collect qx))))

(defun mkFunCall(application)
  (let* (  (fn (car application))
   (mods (cdr (get fn  'mod)))
   (args (cdr application)))
    (cons fn
       (loop for x in args
  for m in mods
  when (equal m '-)
    collect (list 'value x)))))

(defun mkprologside(application)
    (let* (  (fn (car application))
   (mods (cdr (get fn  'mod))))
      (loop for x in (cdr application)
 for m in mods
 collect
   (if (equal m '+)
       (list (gensym (symbol-name x)))
       x))))

(defun denudeGlobalVars(s)
  (loop for x in s
     collect (if (consp x) (car x) x)))

(defun theOutputVariables(s)
  (loop for x in s when (consp x) collect (car x)))

(defun uniglobals(gs vs)
  (loop for g in gs
     for v in vs
     collect (list 'uni g v)))
```

```
;; End Mauro Honorato

(defparameter *defs* nil)

(defmacro notsafe? (x)
  `(and (not CP) (>= (cdr ,x) CL)))

(defun unifnum (x y)
    (bindte (cdr y) x))

(defun generate_specialized_unification(i arg clauses)
  (declare (ignorable clauses))
  (if  (and (consp arg)
    (eq (car (cadr arg)) 'N))
       `(unifnum (svref xArgs ,i)
     (ultimate ,arg env))
     `(unif (svref xArgs ,i)
 (ultimate ,arg env))))

(defun generate_arg_unifications (nv args clauses)
  ;;(format t "nv= ~a, args= ~a~%" nv args)
  (if (eq nv 0) t
  (list 'catch (list 'quote 'impossible)
`(let ((env (push_Environment ,nv)))
           ,@(loop for i from 1 to nv
           for x in args
                collect
  (generate_specialized_unification
   i (if (numberp x) x (list 'quote x))
   clauses))) )))

(defun specialized_choice (unifs paq)
  (let* ((resu  (shallow_backtrack paq)  )
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
            (if (funcall unifs) ;(eq (unify_with (largs (head c))
    ; (push_Environment (nvar c))) 'fail)
             (backtrack)
             (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
```

```
  (max_Local (nvar c)))))))

(defun generate_choice (unifs paq)
  `(let* ((resu  (shallow_backtrack ,paq)  )
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
            (if ,unifs ;(eq (unify_with (largs (head c))
    ; (push_Environment (nvar c))) 'fail)
             (backtrack)
             (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c)))))))

(defun provit (paq)
   (if (cdr paq)
     (let*((caput (car (cadr paq)))
           (args (largs (head caput)))
           (nv (length args))
           (unifs `(lambda()
            ,(generate_arg_unifications nv args paq))))
           (declare (ignorable caput args nv unifs))
         `(progn (push_choice)
            (let* ((resu (shallow_backtrack ,paq))
           (c (car resu))
                  (r (cdr resu)))
              (cond ((null r)
                     (pop_choice)
                     (if (eq
                          (let ((unargs (largs (head c)))
                               (e (push_Environment (nvar c))))
                            (declare (ignorable e unargs))
                            (catch 'impossible
                              ,@(loop for i from 1 to nv collect
                                (list 'unif
                                  (list 'svref 'xArgs i)
                                  '(ultimate (pop unargs) e)))) )
   'fail)
                           (backtrack)
                           (when (tail c)
                             (push_continuation)
```

```
                              (setq CP (tail c)
                                    CL *LocalPointer*)
                              (max_Local (nvar c)))))
                        ((push_bpr r)
                         (when (tail c)
                         (push_continuation)
                         (setq CP (tail c) CL *LocalPointer*)
                         (max_Local (nvar c))) )
                    )) ))
                   (let* ((c (car paq))
                          (tc (when (cdr paq)(tail paq)))
                          (args (largs (head c)))
                          (nv (length args))
                          (unifs (generate_arg_unifications
                                   nv args paq)) )
                    `(if  (eq ,unifs 'fail)
                        (if (zerop BL) ;; backtrack
                          (setq Duboulot nil)
                          (progn (setq *LocalPointer* BL
                                 *GPointer* BG Cut_pt (BL BL)
                                 CP (BCP *LocalPointer*)
                                 CL (BCL *LocalPointer*))
                                 (load_A2)
                                 (poptrail (TR BL))
                                 (pr_choice
                                   (BP *LocalPointer*))))
                      (when ,tc (push_continuation)
                          (setq CP ,tc CL *LocalPointer*)
                          (max_Local ,nv)))  )))

(defun compile-args(pred ind)
  (let* ((nargs (length (cdr (cadar (get pred ind)))))
         (definition (list 'quote (get pred ind)))
         (vs (loop for i from 1 to nargs collect
              `(vset xArgs ,i
                 (let ((te (ultimate (pop largs) PCE)))
                    (cond
                      ((atom te) te)
                      ((var? te)
                       (if (notsafe? te)
                          (genvar (cdr te))
                          te))
                      ((copy? te) te)
                      ((recopy te PCE)))) ))))
        `(lambda()
           (let ((largs (largs PC)))
```

```
                (declare (ignorable largs))
                (vset xArgs 0 ,nargs)
                ,@vs)
                (if CP ,(provit definition)
                  (progn
                    (if (<= BL CL)
                      (setq *LocalPointer* CL))
                    (setq CP (CP CL) CL (CL CL))
                    ,(provit definition))  ))))

(defun add_cl (pred c ind)
  (cond ((and (equal pred 'def))
         (let* ((nm (intern (string-downcase
                   (symbol-name (caadr  c)))))
                (args (cdr (cadr c)))
                (prologside (mkprologside (cadr c)))
                (fn (if (checkFunMode (cdr c))
                     `(lambda(,@(denudeGlobalVars prologside))
                         (uni ,(car (car prologside))
                              ,(mkFunCall (car (cdr c)))))
                      `(lambda(,@(denudeGlobalVars prologside))
                         (multiple-value-bind ,args
                            ,(mkCall  (car (cdr c))  ))
                            (declare (ignorable ,@args))
                            ,@(uniglobals
                              (theOutputVariables prologside)
                              (getOutputVariables (cadr c))))) )))
              (eval c)
              (setf (symbol-function nm) (eval fn))
              (setf (get nm 'evaluable) t)))
         ((equal pred 'mdef) (eval c)  )
         ((equal ind 'def)
          (when (not (member pred *defs*))
                (push pred *defs*))
          (setf (get pred ind) (append (get pred ind) (list c)))
          (setf (get pred 'partial) t)
          (setf (symbol-function pred)
                (eval (compile-args pred ind))))
         (t (when (not (member pred *defs*))
              (push pred *defs*))
              (setf (get pred ind)
                 (append (get pred ind) (list c))))))

(set-macro-character
 #\$
 #'(lambda (stream char)
```

```
     (declare (ignorable char))
     (let* ( (*standard-input* stream) (c (read_code_cl)))

       (add_cl (if (symbolp (car c)) (car c)
   (pred (head c ))) c 'def)
       (if (largs (head c))
   (let ((b (nature (car (largs (head c))))))
     (if (eq b 'def)
 (mapc
  #' (lambda (x) (add_cl (pred (head c)) c x))
  '(atom empty list fonct))
       (add_cl (pred (head c)) c b)))))
     (values)))

(defun yes/no()
  (princ "More : ")
  (finish-output)
  (member (rchnsep) '(#\o #\y #\s #\d #\j #\;)))

(defun answer ()
  (printvar)
  (if (zerop BL)
      (setq Duboulot nil)
    (if (yes/no)
(backtrack)
      (setq Duboulot nil))))

(defun printvar ()
  (if (null *lvar)
      (format t "Yes ~%")
    (let ((n -1))
      (mapc
       #' (lambda (x)
    (format t "~A = " x)
    (write1 (ult (+ (incf n)
    (Environment BottomL)))) (terpri))
       *lvar))))


;; mini-wam
;; unification

(defun ult (m)
  (declare (fixnum m))
  (do* ( (n m (cdr te))  (te (svref Mem n) (svref Mem n)))
       ( (not (and (var? te) (/= (cdr te) n))) te)
```

```
      (declare (fixnum n))))

(defun ultimate (x e) (if (var? x) (ult (adr x e)) x))
(defun val (x) (if (var? x) (ult (cdr x)) x))

(defmacro bind (x te)
  `(progn
     (if (or (and (> ,x BottomL) (< ,x BL))
     (<= ,x BG))
 (pushtrail ,x ,te))
     (vset Mem ,x ,te)))

(defun bindte (xadr y)
  (declare (fixnum xadr))
  (if (or (atom y) (copy? y))
      (bind xadr y)
      (bind xadr (recopy y (Environment *LocalPointer*)))))

(defun genvar (x) (declare (fixnum x))
       (bind x (push_Global (cons 'V *GPointer*))))

(defmacro bindv (x y)
  `(if (< (cdr ,x) (cdr ,y))
       (bind (cdr ,y) ,x)
       (bind (cdr ,x) ,y)))

(defun unify_with (largs e)
  (catch 'impossible
    (do ((i 1 (1+ i)))
((null largs))
       (declare (fixnum i))
      (unif (svref xArgs   i)
    (ultimate (pop largs) e)))))

(defmacro mval(x) `(if (var? ,x) (ult (cdr ,x)) ,x))

(defun unif (x y)
    (cond
     ((eql x y) t)
     ((var? y)
   (if (var? x)
 (if (= (cdr x) (cdr y)) t (bindv y x))
       (bindte (cdr y) x)))
     ((var? x)
      (bindte (cdr x) y))
   ((or (atom x) (atom y)) (throw 'impossible 'fail))
```

```
   ((let ((b (copy? y)) (dx (pop x)) (dy (pop y)))
      (if (eq (functor dx) (functor dy))
  (do* ( (ax x (cdr ax))
 (vx (mval (car ax)) (mval (car ax)) )
 (vy (pop y) (pop y)))
       ((null ax))
      (unif vx
    (if b (mval vy)
(ultimate vy (Environment *LocalPointer*)))  ))
(throw 'impossible 'fail))))))


;; mini-wam
;; resolution

(defun lispforward ()
  (do () ((null Duboulot))
      (cond ((null CP) (answer))
    ( (load_PC)
      (cond
       ((user? PC)
(let ((d (def_of PC)))
  (if d (pr2 d) (backtrack))))
       ((builtin? PC)
(if (eq (apply (car PC) (cdr PC)) 'fail)
    (backtrack)
  (cont_eval)))
       ((backtrack)))))))

(defun forward ()
  (do () ((null Duboulot) (format t "no More ~%"))
      (cond ((null CP) (answer))
    ( (load_PC)
              ;; PC contains the goal
     (cond
       ((partial? PC)
(funcall (car PC)))
       ((user? PC)
(let ((d (def_of PC)))
  (if d (pr2 d) (backtrack))))
       ((builtin? PC)
(if (eq (apply (car PC) (cdr PC)) 'fail)
    (backtrack)
  (cont_eval)))
       ((backtrack)))))))
```

```
(defun load_PC ()
  (setq PC (pop CP) PCE (Environment CL) Cut_pt BL))

(defmacro notsafe? (x)
  `(and (not CP) (>= (cdr ,x) CL)))

(defun cont_eval ()
  (unless CP
    (if (<= BL CL) (setq *LocalPointer* CL))
         (setq CP (CP CL) CL (CL CL))))

(defun pr_choice (paq)
  (let* ((resu  (shallow_backtrack paq)  )
 (c (car resu)) (r (cdr resu)))
    (cond ( (null r)
    (pop_choice)
            (if (eq (unify_with (largs (head c))
     (push_Environment (nvar c))) 'fail)
             (backtrack)
             (when (tail c) (push_continuation)
     (setq CP (tail c) CL *LocalPointer*)
     (max_Local (nvar c)))))
  ( (push_bpr r)
    (when (tail c)
  (push_continuation)
  (setq CP (tail c) CL *LocalPointer*)
  (max_Local (nvar c)))))))

(defun shallow_backtrack (paq)
  (if (and (cdr paq)
   (eq (unify_with
(largs (head (car paq)))
(push_Environment (nvar (car paq))))
       'fail))
      (progn
(poptrail (TR BL))
(setq *GPointer* BG)
(shallow_backtrack (cdr paq)))
    paq)  )

(defun backtrack ()
  (if (zerop BL)
      (setq Duboulot nil)
      (progn (setq *LocalPointer* BL *GPointer* BG Cut_pt (BL BL)
   CP (BCP *LocalPointer*) CL (BCL *LocalPointer*))
   (load_A2)
```

```
   (poptrail (TR BL))
   (pr_choice (BP *LocalPointer*)))))

(defun load_A2 ()
  (let ((deb (- *LocalPointer* (size_C *LocalPointer*))))
    (dotimes (i (AChoice *LocalPointer*) (vset xArgs 0 i))
      (declare (fixnum i))
(vset xArgs (+ i 1)
      (svref Mem (+ deb i))))))

(defun lisploop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
        CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (lispforward)))

(defun readProvePrintLoop (c)
  (setq *GPointer* BottomG *LocalPointer* BottomL TR BottomTR
        CP nil CL 0 BL 0 BG BottomG Duboulot t Cut_pt 0)
  (push_continuation)
  (push_Environment (nvar c))
  (setq CP (cdr c) CL *LocalPointer*)
  (max_Local (nvar c))  (read-char)
  (catch 'debord (forward))
  (cond ( *keep-going*
 (handler-case (readProvePrintLoop (read_prompt))
   (error(e) (format t "Error: ~a" e) (readProvePrintLoop (read_prompt)))))
(t (format t "Bye!~%")
   (setf *keep-going* t))))

;; mini-wam
;; predicados predefinidos.
;;

(defvar Ob_Micro_Log
  '(|debugvar| |one| |nt| |write| |nl| |tab| |read| |get| |get0|
    |var| |nonvar| |atomic| |atom| |number| |clear|  |is|
    ! |fail| |true| |halt| |lisp| |div| |eqn| |gt| |ge|
    |divi| |mod| |plus| |minus| |times| |le| |lt|
    |name| |consult| |sult| |abolish| |cputime| |statistics|))
(mapc #'(lambda (x) (setf (get x 'evaluable) t)) Ob_Micro_Log)

(defmacro value (x)
```

```
  `(if (or (var? ,x) (atom ,x))
       (ultimate ,x PCE)
       (copy ,x PCE)))
(defun uni (x y) (catch 'impossible (unif (value x) y)))

(defun |debugvar|(x)
  (format t " dvar ~a " x))

;;lisp/2
(defun |lisp|(res fn)
  (uni res
       (eval (cons (intern
    (string-upcase
       (symbol-name
(functor (functorDescription fn) ))))
         (mapcar (lambda(x) (value x)) (largs fn))) ) )

(defun evalfp(x)
  (cond ((numberp x) x)
( (and (consp x) (eq (car (functor x)) '|neg|))
 (- (evalfp (car (fargs x)))))
((functor x)
 (funcall (symbol-function (car (functor x)))
  (evalfp (car (fargs x)))
  (evalfp (cadr (fargs x))))) ))

;;write/1 (?term)
 (defun |write| (x) (write1 (value x)))
 (defun write1 (x)
      (cond
       ((null x) (format t "[]"))
       ((atom x) (format t "~A" x))
       ((var? x) (format t "X~A" (cdr x)))
       ((list? x) (format t "[")
(writesl (val (cadr x)) (val (caddr x)))
(format t "]"))
       ((writesf (functor (functorDescription x))
 (largs x)))))

(defun writesl (tete q)
      (write1 tete)
      (cond
       ((null q))
       ((var? q) (format t "|X~A" (cdr q)))
       (t (format t ",") (writesl (val (cadr q))
  (val (caddr q))))))
```

```
(defun writesf (fct largs)
      (format t "~A(" fct)
      (write1 (val (car largs)))
      (mapc #'(lambda (x) (format t ",")
      (write1 (val x))) (cdr largs))
      (format t ")"))

;;nl/0
(defun |nl| () (terpri))
;;tab/1 (+int)
(defun |tab| (x)
  (dotimes (i (value x)) (format t " ")))
;;read/1 (?term)
(defun |read| (x)
   (let ((te (read_terme)))
      (catch 'impossible
      (unif (value x)
      (recopy (cdr te)
      (push_Environment (car te)))))))

(defun read_terme ()
   (let ((*lvar nil))
     (let ((te (read_term (rchnsep))))
       (rchnsep) (cons (length *lvar) te))))
;;get/1 (?car)
(defun |get| (x) (uni x (char-int (rchnsep))))
;;get0/1 (?car)
(defun |get0| (x) (uni x (char-int (read-char))))
;;var/1 (?term)
(defun |var| (x) (unless (var? (value x)) 'fail))
;;nonvar/1 (?term)
(defun |nonvar| (x) (if (var? (value x)) 'fail))
;;atomic/1 (?term)
(defun |atomic| (x) (if (listp (value x)) 'fail))
;;atom/1 (?term)
(defun |atom| (x) (unless (symbolp (value x)) 'fail))
;;number/1 (?term)
(defun |number| (x) (unless (numberp (value x)) 'fail))
;;fail/0
(defun |fail| () 'fail)
;;true/0
(defun |true| ())
;;divi/3 (+int,+int,?int)
(defun |divi| (z x y) (uni z (floor (value x) (value y))))
;;div/3
```

```
(defun |div| (z x y) (uni z (/ (value x) (value y))))
;;mod/3 (+int,+int,?int)
(defun |mod| (z x y) (uni z (rem (value x) (value y))))
;;plus/3 (+int,+int,?int)
(defun |plus| (z x y) (uni z (+ (value x) (value y))))
;;minus/3 (+int,+int,?int)
(defun |minus| (z x y) (uni z (- (value x) (value y))))
;;mult/3 (+int,+int,?int)
(defun |times| (z x y) (uni z (* (value x) (value y))))
;;le/2 (+int,+int)
(defun |le| (x y) (if (> (value x) (value y)) 'fail))
;;lt/2 (+int,+int)
(defun |lt| (x y) (if (>= (value x) (value y)) 'fail))
;; eqn/2
(defun |eqn| (x y) (if (not (= (value x) (value y))) 'fail))
;;gt/2
(defun |gt| (x y) (if  (<= (value x) (value y)) 'fail))
;;ge/2
(defun |ge| (x y) (if (< (value x) (value y)) 'fail))
;;name/2 (?atom,?list)
(defun undo_l (x)
  (if (atom x) x
      (cons (undo_l (val (cadr x))) (undo_l (val (caddr x))))))

(defun |name| (x y)
  (let ((b (value x)))
     (if (var? b)
         (uni x (impl (undo_l (value y))))
         (uni y (do_l (expl b))))))

(defun impl (l) (intern (map 'string #'code-char l)))
(defun expl (at) (map 'list #'char-int (string at)))

(defun addit(c)
    (add_cl (if (symbolp (car c))
                (car c)
                (pred (head c ))) c 'def)
    (if (largs (head c))
       (let ((b (nature (car (largs (head c))))))
           (if (eq b 'def)
               (mapc
                 #' (lambda (x) (add_cl (pred (head c)) c x))
                  '(atom empty list fonct))
                 (add_cl (pred (head c)) c b)))))

(defun |tolisp| (f)
```

```
  (let ((nm (value f)))
    (setf (get nm 'partial) (get nm 'def))
    (setf (get nm 'def) nil)))

(defun |consult| (f)
  (let* ((filename (format nil "~a" (value f)))
 (code (rd_file filename)))
    (loop for c in code
       do (addit c))))

;;consult/1 (+atom)
(defun |sult| (f) (format t "~A~%"
(load (format nil "~A" (value f))  )))

;; abolish/1
(defun |abolish| (p)
  (mapc  #'(lambda (x) (setf (get p x) nil))
 '(atom empty list fonct def)))

;; clear/0
(defun |clear| ()
  (mapc #'(lambda(x) (|abolish| x)) *defs*))

;; cputime/1
(defun |cputime| (x)
  (uni x (float (/ (get-internal-run-time)
   internal-time-units-per-second))))

;; statistics/0
(defun |statistics| ()
  (format t " local stack : ~A (~A used)~%"
  (- +LocalStackOverflow+ BottomL)
          (- *LocalPointer* BottomL))
  (format t " global stack : ~A (~A used)~%"
  BottomL (- *GPointer* BottomG))
  (format t " trail : ~A (~A used)~%"
  (- (array-dimension trailMem 0) BottomTR)
  (- TR BottomTR)))

(defun |halt| ()
  (setf *keep-going* nil))

(defvar *G* (the fixnum 0))
(defvar *L* (the fixnum 0))
(defvar *TR* (the fixnum 0))
(defvar *BG* (the fixnum 0))
```

```
(defun |one| ()
  (setq *G* *GPointer* *L* *LocalPointer* *TR* TR *BG* BG))

(defun |is|(x fx)
  (uni x (evalfp  (value fx ))))

(defun |nt|(n)
    (setq BL (Cut CL) BG (if (zerop BL) BottomG (BG BL))
  *LocalPointer* (+ CL 3 n))
    (when *TR*
       (setq TR *TR* *GPointer* *G* *G* nil *TR* nil)) )

;;cut/0
(defun ! (n)
  (setq TR (if (zerop BL) BottomTR (TR BL))
 BL (Cut CL)
BG (if (zerop BL) BottomG (BG BL))
       *LocalPointer* (+ CL 3 n)) )

(defun bye()
  (cl-user::exit))
```

Bibliography

[1] Nielsen, Jakob Information Pollution. Available in `http://www.nngroup.com/articles/information-pollution/` on August 11, 2003. Accessed at July 03, 2015.

[2] Briscoe, T., Buttery, P., Carroll, J., Medlock, B., Watson, R. Open Source RASP 3.1 – Robust Parsing System for English. http://ilexir.co.uk

[3] Stonebank, M. UNIX Tutorial for Beginners. http://www.ee.surrey.ac.uk/Teaching/Unix/

[4] Steel Bank Common Lisp. Available in `www.sbcl.org`.

[5] SLIME: The Superior Lisp Interaction Mode for Emacs. Available from `https://common-lisp.net/project/slime/`

[6] Beane, Zach. quicklisp Beta. Available in `www.quicklisp.org`

[7] Mark Kantrowitz. Infix. http://www.cliki.net/infix

[8] Doug Hoyte. Let Over Lambda, 50 years of Lisp. Lulu.com Publisher. First edition, 2008. ISBN 978-1-4357-1275-1 M. G. F. Fuortes, ed., Handbook of Sensory Physiology, Springer-Verlag, Berlin, 1972.

[9] Peter Seibel Practical Common Lisp. Apress, First edition, 2005, ISBN-10: 1590592395.

[10] Gambol Gambol is basic logic programming. https://code.google.com/p/cl-gambol/wiki/API

[11] Peter Norvig PAIProlog that an update of Peter Norvig's "Prolog in Common Lisp". https://github.com/quek/paiprolog

[12] Dorai Sitaram Racklog: Prolog-Style Logic Programming. http://docs.racket-lang.org/racklog/

[13] Daniel P. Friedman and R. Kent Dybvig. Schemelog - a logic programming language. ftp://ftp.cis.upenn.edu/pub/milevin/schemelog.html

[14] Oleg Kiselyov Kanren: A declarative logic programming system beta. http://sourceforge.net/projects/kanren/files/

[15] David H. D. Warren An Abstract Prolog Instruction Set. SRI International, 1983.

[16] Series Editor: J. A. Campbell. Implementations of Prolog. Ellis Horwood Limited, 1984.

[17] Hassan Ait-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press Version, 1999.

[18] Tarantola, Albert Popper, Bayes and the inverse problem. Nature Physics, vol. 2 pp.492-494, 2006.

[19] KOZA, John. Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, 1992.

[20] LIST, Christian, GODIN, Robert E. Epistemic Democracy: Generalizing the Condorcet Jury Theorem. Journal of Political Philosophy, vol. 9, jan. 2001.

[21] WRIGHT, Benjamin D., MOK, Magdalena M. C.. An Overview of the Family of Rasch Measurement Models. In Introduction to Rasch Measurement, pp.1-24, 2004.

[22] James Allen Natural language understanding. 2nd ed., 1994.

[23] SUNDE, E. D., Earth conduction effects in transmission systems. MacMillan & Company, London, 1968.

[24] WENNER, F. A.. Method of measuring earth resistivity. Bureau Standards, 1916.

[25] Francisco J. Moral, Francisco J. Rebollo and José M. Terrón. "Analysis of soil fertility and its anomalies using an objective model". Journal of Plant Nutrition and Soil Science, vol. 175, no.6, pp.912-919, December, 2012.

[26] Louisa Lam "Theory and application of majority vote - from Condorcet Jury Theorem to pattern recognition". Proceedings of the International Conference on "Mathematics for Living". Jordan: The Mathematics Education into the 21st Century Project, 2000.

[27] Svingen, B. "When Lisp is Faster than C". Genetic and Evolutionary Conference, pp.957-958, 2006. ISBN: 1-59593-187-2.