A Pamphlet against R

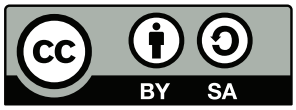
Computational Intelligence in Guile Scheme

Panicz Maciej Godek

edited by: Lawrence Bottorff and Rick Hanson

March 12, 2016

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



I dedicate this booklet to my brother, who excels at Excel programming.

Contents

1 Introduction **7**

1.1 Epistemic role of programming 7

1.1.1 Expression-based programming 8

1.2 Introduction to Scheme 11

1.2.1 Constructions 12

1.2.2 How do we know 15

1.2.3 Destructuring 16

1.2.4 Getting your hands dirty 19

1.2.5 Looking down 20

1.2.6 Building structures 25

1.2.7 Conclusion 26

2 Genetic Algorithms **27**

2.1 Biological inspiration 27

2.1.1 Chromosomal crossover 28

2.1.2 The ceremony of procreation 28

2.1.3 Evolution 31

2.2 Solving the SAT 32

2.2.1 Parsing DIMACS CNF files 32

2.2.2 Applying the genetic strategy 36

3 Fuzzy Logic **39**

3.1 Basic concepts 39

3.2 Criticism 40

3.3 Exposition 40

4 Matrix Operations **51**

4.0.1 Matrix addition 51

4.0.2 Variadic functions 52

4.0.3 Transpose 53

4.0.4 Matrix multiplication 54

5 Classifiers **55**
5.1 Introduction 55
5.2 Naive Bayes and Probability 55
5.2.1 What is the universe 56
5.3 Decision Trees and Information 60
5.3.1 Representing trees in Scheme 61
5.3.2 Constructing a decision tree 64
5.3.3 Deciding upon a decision tree 67
5.4 k Nearest Neighbours 68
5.5 Quantization of numerical data 70
5.6 Evaluating classifiers 71
5.7 Clusterization 74

Preface

"I hate all those stupid quotes at the beginning of each chapter."
– Anyone who hates all the stupid quotes at the beginning of each chapter

Behind every text, however technical, there is a personal story of its author[1] and the author's motivations.

Having worked for a couple of months in an engineering company that manufactured electronic equipment, I decided that perhaps it was the best time to finish my education. So, I enrolled at the department of Computer Science at the University of Gdańsk. Among other topics, one lab was devoted to data processing heuristics jointly named "computational intelligence".

Our first meeting was spent getting acquainted with the programming language R that we were supposed to use throughout the semester for the assignments. But I balked.

On the one hand, I've used many programming languages throughout my career: JavaScript, Python, PHP, Perl, Bash, Matlab, SQL, C, C++, Assembly, Haskell, Erlang, Prolog, Pascal, BASIC, VisualBasic, MiniKanren, and a few others. Some of those languages were really mind-expanding in their own unique ways.

On the other hand, being a rather seasoned Schemer, I knew that most of these languages were only inferior incarnations of Lisp, and I could find little value in learning a new syntactical disguise for those well-established concepts.

I was very soon to find out that my skepticism was right, and that learning R would require internalizing a lot of unfortunate idiosyncrasies that the authors of the language had chosen for their own idiosyncratic reasons, and that I would be constantly stumbling over those incomprehensible limitations and discrepancies between common practices and the language developers' decisions. And for that main reason I didn't want to waste my time.

[1] Even if the text was generated by a machine, and the machine was designed by a computer program, and so on, this causal chain eventually has its author.

Bjarne Stroustrup said that there are only two kinds of languages: the ones people complain about and the ones nobody uses[2]. The advocates of R sometimes admit that perhaps the language isn't perfect, but it contains a lot of convenient libraries to make the whole thing worthwhile.

Being a nobody, I was in a gray area where I could complain about the language or not. I asked my TA whether he would allow me to choose another language for doing the assignments, and he agreed.

My language of choice was of course Scheme, or – more precisely – Guile Scheme. I chose Guile Scheme because it has a wonderful trait: rather than helplessly complain about features I don't like, I can replace them with whatever I find more suitable. If a feature is missing from the language, I can add it, rather than swear and curse its developers.

The title of this pamphlet may suggest that it is written against the R programming language, but it is actually written against the majority of programming languages in use today[3]. The ultimate goal of this text is to violently eliminate them all.

Recognizing that the aforementioned goal is rather quixotic, I will be satisfied instead if anyone finds any value in this pamphlet, whether it be education, entertainment or personal hygiene[4].

Structure of this pamphlet

This pamphlet assumes no prior knowledge of programming. If you think you already possess programming knowledge, you'll probably see much that is new here.

Throughout the subsequent chapters, a large subset of the Scheme programming language will be laid out, later chapters building upon the earlier. The first chapter introduces all the elementary notions needed for programming in Scheme. The next two chapters show how to use that knowledge in practice. The fourth chapter introduces a few more features of the language that are used in the fifth chapter.

Each chapter will introduce new definitions to be used in later applications. For example, the third chapter presents the notion of `equivalence--classes` used extensively throughout the fifth chapter.

The vision of programming presented here departs significantly from the mainstream view, where a program is perceived as a sequence of steps that lead us to a desired goal.

[2]He probably said that, because he was himself a creator of a popular language that everybody complains about.

[3] However, the case of the R programming language is particularly interesting, because initially it was just a harmless implementation of Scheme; but then as a result of the irresponsible experiments of mad scientists it mutated into a monster[3].

[4]Concerns the printed copies

On the contrary, I intended to explain programs as intellectual constructs of increasing complexity; hence, reading the chapters in their proper sequence is strongly recommended.

How to read this pamphlet

The most essential part of this pamphlet are the definitions. Everything else is just scaffolding whose purpose is to facilitate their comprehension. The ultimate goal is to make those definitions dance in the reader's head; hence each definition should be studied with care. If there are no examples given, it is up to the reader to come up with proper examples, because if a definition is the dance, then examples are the dancers.

The whole point of a definition is to create or express a notion that can be further used to think about phenomena in the real world. Internalizing a definition is not always a pleasant task, because it requires an intellectual effort of focusing and remembering.

Reading programs isn't like reading novels. Because of their familiar narrative generality, we typically absorb stories almost effortlessly. However, comprehending logical abstractions requires a very different approach. Well written programs are a bit like mathematical textbooks. The latter usually consist mainly of definitions, examples, theorems and proofs (and exercises).

The role of theorems is usually diminished in the realm of computer programs: essential are the constructions, the definitions. Examples (sometimes called *unit tests* by software folks) serve mainly as a means of ensuring a program behaves as expected, which gains importance as the program is subjected to changes. Theorems manifest themselves in the weak form of *assertions* and *type signatures*, while proofs are usually performed by specific tools such as type checkers or model checkers. Furthermore, false assertions can typically be refuted during the execution of a program.

In the case of math textbooks, exercises and examples are a way of facilitating comprehension. It is usually much easier to understand an abstract definition if we consider examples that are concrete instances of the abstract definition. Similarly, exercises are like examples but where some or all of the reasoning is left out; as such, they can be thought of as "partial examples" where we are challenged to come up with the reasoning for ourselves.

Reading a math textbook without doing the exercises is also possible, but it is up to the reader to come up with the deeper and broader understanding[5].

[5] Programmers usually read computer programs in order to change them, which is an ultimate exercise in the comprehension of complex systems. On the other hand, programs are often written with the purpose of "doing their job", rather than being comprehended and modified.

From my experience, this approach dominates, not only in industry (which is understandable), but also in education (which is harmful to industry).

Who should read this pamphlet

The title of this pamphlet may be misleading since readers might think we are only concerned with computational methods. Although there are certainly computational methods involved, this work is really about thinking and communicating thoughts.

While it is difficult to imagine anyone who would not benefit from the knowledge presented here, there are some groups who are particularly encouraged to do so. For instance, this text may be of interest to philosophers and linguists, as it presents a linguistic perspective on communicating thoughts in a precise manner.

It may also be interesting to evolutionary biologists, because it contains a quote from Richard Dawkins and stuff like that.

It should be of particular interest to computer scientists and programming language designers, to convince them to *stop designing new languages immediately!*, because – as the history teaches us – they often enough succeed, thereby forcing generations of programmers to suffer due to the ignorance of their creators.

However, as previously indicated, this book is about programming *and* clear thinking in equal measure, and it is difficult to imagine someone who could not benefit from this combination.

And so perhaps the shortest answer to the question posited in the title of this section is: *you*.

Acknowledgements

Most of all, I would like to thank Grzegorz Madejski, the teaching assistant who allowed me to go my own way, and encouraged me to devote time to this project. His classes were the source of all the examples presented in this text.

I am also truly grateful to my family and my girlfriend, as they all encouraged me to go back to university. I certainly couldn't have done it without their support.

I deeply appreciate the interest, support, feedback and encouragement of my friends who allowed me to bother them with the early versions of this text, and with whom I have had many valuable and inspiring conversations, including Ścisław Dercz, Hubert Melchert, Michał Janke, Marcin Jędruch, Jan Makulec and Aleksander Mohuczy-Dominiak.

I owe thanks to David Hashe, Robert Różański, Martin Clausen and Dave Creelman for pointing out some errors and suggesting improvements.

Reporting bugs

I realize that the quality of this text may be questionable. After all, this is a pamphlet. However, if you find any parts of the text obscure or difficult to understand, or if you find any mistakes in the explanations presented here, or simply want to talk about how miserable the world is, feel free to write an e-mail to godek.maciek+pamphlet@gmail.com.

Also make sure that you are reading the latest version of this pamphlet, always available from

`https://panicz.github.io/pamphlet`

as many of the bugs may already have been fixed.

Finally, if you would like to become a co-author and add your own chapters or examples, it will be my pleasure to merge your pull-requests.

1

Introduction

> *"The acts of the mind [...] are chiefly these three: 1. Combining several simple ideas into one, and thus all complex ideas are made. 2. [...] bringing two ideas, whether simple or complex, together, and setting them by one so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. [...] separating them from all the other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made."*
>
> *John Locke, An Essay Concerning Human Understanding (1690)*

[...]

A powerful programming language [...] serves as a framework within which we organize our ideas [...]. Every programming language has three mechanisms for accomplishing this:

- ***primitive expressions*** *[...]*
- ***means of combination*** *[...]*
- ***means of abstraction*** *[...]*

– Hal Abelson and Gerald Sussman, *Structure and Interpretation of Computer programs*

1.1 Epistemic role of programming

This text was written with the belief that programming is not only a way of automating common repetitive tasks, but – most importantly – it is a means (probably the most efficient) of representing knowledge and understanding the world and its phenomena, as well as expressing original ideas.

This view is frequently obscured by some bad practices that are widespread among programmers. Apart from expressing knowledge and ideas, programming languages allow programmers to do virtually everything they like – it is therefore possible that they write code which gets in the way to understanding, rather than facilitate it[1].

The claim that many programmers write code in "the wrong way", and that the author of this pamphlet knows which way is "the right way" may seem pretentious. It must therefore inevitably be left to the reader to judge whether the author's claims are true.

1.1.1 Expression-based programming

Throughout this pamphlet, we will be perceiving computer programs as constructions of compound objects. This view has recently been gaining more popularity, but it is still far from the mainstream view, which claims that a program is a set of instructions that ought to be performed in a given order.

The latter view is often called *imperative programming*, and has traditionally been contrasted with *declarative programming*. Within the realm of declarative programming, there is a particular subset called *functional programming*, where the programmer creates programs by defining functions (in mathematical sense).

However, since the notion of functional programming has recently been somewhat appropriated by Haskell, and for some reasons that shall become clear later, I will prefer to use the term *expression-based programming* or *constructive programming*.

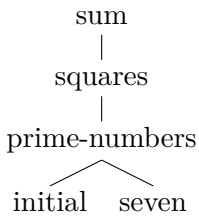
For example, let's say that someone wishes to write a program that computes **a sum of squares of initial seven prime numbers**. The imperative program that does this could look as follows:

```
counter := 7
number := 0
sum := 0
while(counter > 0):
    if is_prime(number):
        sum := sum + number^2
        counter := counter - 1
    number := number + 1
```

[1] Apparently even excellent programmers and great erudites have been failing at this task. One of the most extreme examples is the code from Donald Knuth's *The Art of Programming*. Written in the `MIX` assembly, the code only hides the conceptual structure of presented solutions from the reader.

After its execution, the `sum` variable will contain the desired value.

However, if we take a look at the specification of the problem, we will notice that it can be itself perceived as a part of the solution: the expression "sum of squares of initial seven prime numbers" has a certain linguistic (or syntactic) structure, and a corresponding conceptual (or semantic) structure. The syntactic structure may not be immediately apparent, but simple analysis shows that the outer form of the expression is "sum of X", where X is "squares of initial seven prime numbers". We can further decompose the latter, extracting the form "squares of Y", where Y is "initial seven prime numbers". Making yet another step, we obtain the expression "initial N prime numbers", where N is "seven".



This isn't the only way in which we can perform decomposition. That's because the natural language that we use in our everyday communication is naturally ambiguous. For instance, by "initial seven prime numbers", do we mean "a set of initial prime numbers", or "a sequence of initial prime numbers"?

This distinction is of course insignificant from the point of view of our task, because addition is commutative, and hence the order in which we compute this sum is arbitrary. On the other hand, if we asked someone to name the initial seven prime numbers, it would be likely that he or she would enumerate them in the ascending order. If we investigated further into that phenomenon, it would probably turn out that the order is a consequence of the order in which the numbers are computed by the speaker. Indeed, what a shocking experience it would be to hear those numbers being uttered in a random or descending order!

In order to avoid such shocks, people apply the so-called *cooperative principles* to their conversations[2]: they try to adopt their utterances to the expectations of their interlocutors (to the extent in which they can recognize them), or – in the parlance of programmers – they "try to design clean and simple APIs"[3].

[2] The theory of Cooperative Principles was formulated by Paul Grice. He explicated four maxims that ought to be applied in effective communication. Probably the most relevant to programming and the one that is least respected by programmers is the Maxim of Manner, which recommends to avoid obscurity of expression, avoid ambiguity, be brief and be orderly.

[3] Seriously.

Note however, that even if there is no natural order of elements in a set, whenever we wish to enumerate elements from that set, imposing *some* order is inevitable.

Therefore, unless specified otherwise, whenever we speak about some multitude of elements, we shall mean a sequence of those elements, rather than a set (even if the order is irrelevant).

Returning to our example, we have a function "initial N prime numbers", where N is a natural number. The value of a function, for a given N, is an ascending sequence of N initial prime numbers. For instance, the value (or meaning [4], or denotation) of expression "initial 7 prime numbers" is a sequence `(2 3 5 7 11 13 17)` [5].

This is as far as we can go with the linguistic analysis. Obviously, the expressions under consideration consist of words, but their meanings do not (at least not in this particular case)[6]: the sequence of numbers consists of numbers and their order. We could therefore ask: what is the structure of a sequence?

We can note that sequences can be empty, as it is the case with the denotation of the expression "initial 0 prime numbers", or "empty sequence". Non-empty sequences can be decomposed into their first element, and the remainder of its elements (which may be empty). This prompts us with a recipe to construct sequences recursively. First, let's note that the phrase "initial N prime numbers" is synonymous with "N least prime numbers greater than 0". Therefore we have function of two arguments: "N least prime numbers greater than M".

We can define this function as follows:

> The meaning of "0 least prime numbers greater than M" is an empty sequence.
>
> The meaning of "$N+1$ prime numbers greater than M" is:
>
> - a sequence whose first element is $M+1$, and whose remaining elements are the N prime numbers greater than $M+1$, **if** $M+1$ is a prime number;
> - otherwise it's just the same as $N+1$ prime numbers greater than $M+1$.

[4] As it was shown by Frege, the word "meaning" is also ambiguous, and can either mean *intension/connotation* or *extension/denotation*. Intension of a semantically compound concept is its formulation in simpler terms.

[5] Of course, we cannot write down a sequence, and so we are forced to write down a *textual representation* of a sequence. In fact, we can only write down two distinct textual expressions, hoping that the reader will establish in her or his mind, that those distinct two expressions are meant to refer to the same abstract object.

[6] It is possible to build linguistic expressions whose meanings consist of words. A fine example is a linguistic expression "This sentence", which refers to a sequence of two words: `(This sentence)`.

Although the above formulation is a legitimate description that explains how to construct a sequence of subsequent prime numbers, it disconcertingly brings legal gibberish to mind. It also makes it difficult to distinguish between the strict definitional part of the text from its more explanatory fragments. Moreover, as it was noted before, the syntactical structure of the text isn't immediately clear even for the simplest examples.

In order to handle these issues, it is customary to introduce a formal notation. It is a common practice in mathematics, where mathematicians use special symbols to denote certain abstract concepts, and in logic, where logicians try to make certain reasonings more explicit.

1.2 Introduction to Scheme

In the quest for our perfect notation, we shall stick to the rule of parsimony: we want to make as few assumptions and conventions as possible, but we also want to make sure that they are universal. Firstly, let's embrace the compound syntactical units in parentheses. Then, the expression "sum of squares of initial seven prime numbers" becomes:

```
(sum (squares (initial seven prime numbers)))
```

While at first the parentheses may seem difficult to match, they allow to avoid confusions and ambiguities that are typical for the natural language. Note also, that we got rid of the "of" prepositions, because they provide no information: we can simply read `(f x)` as "f of x" (provided that f has only one argument).

Furthermore, let's stick to a convention, that a single concept is expressed using a single word, where by words we mean sequences of letters separated with parentheses or white spaces. Also, let's make sure that the ruling word of the expressions appears at the first position. We can make compound words using hyphens:

```
(sum (squares (prime-numbers initial seven)))
```

This formulation expresses the structure of our problem well enough using so-called *fully parenthesized Polish notation* – certainly we can no longer be confused with lawyers. Now the only thing that's left is to explain in basic terms what we mean by `sum`, `squares` and `prime-numbers`. The question which terms are to be regarded as basic depends on the context. In this particular case, we also happen to have written (a fragment of) a program in a programming language called Scheme, which enumerates a whole set of notions that are regarded as simple[7], and the notion of `prime-numbers` certainly does not belong to that set. It does, however, contain the notions of numbers, addition, multiplication, division, comparison and it also provides

[7] The exact specification of the basic terms of Scheme is provided in the *Revised^n Report on the Algorithmic Language Scheme* that can be found at `http://www.schemers.org/Documents/Standards`.

some means to construct and deconstruct sequences. Most importantly, it provides means for defining new notions.

1.2.1 Constructions

Paraphrasing our construction of "least N prime numbers greater than M" from previous section, we can write:

```
(define (prime-numbers amount from)
  (if (= amount 0)
    '()
    (if (prime? from)
      (cons from (prime-numbers (- amount 1) (+ from 1)))
      (prime-numbers amount (+ from 1)))))
```

Although conceptually we present nothing new here compared to the previous section, there are a few formal elements that need to be explained. First of all, we use a special `define` form from the Scheme language. This way, we give a name to the new concept that we are defining[8]. We also observe that there are quite a few new concepts used in the definition. The body (or *definiens*, if you please) consists of an `if` expression, which – in general – takes the following form:

`(if <condition> <then> <else>)`

The value of the `if` expression depends on the value of `<condition>`. If it is true in a given context, then the value of the whole expression becomes the value of the `<then>` expression. Otherwise, it is the value of the `<else>` expression. (This is actually quite straightforward, if you think about it.)

The expression that appears as the `<condition>` has a form `(= amount 0)`. The meaning of the word `=` is the *numerical comparison*, commonly found in mathematics. It can be surprising to see it used as `(= x y)` rather than `(x = y)`. Recall however our convention that "the ruling word of the expression appears at the first position". We could depart from that rule for mathematical expressions (and whoever knows where else), but this would only make our rule set more complex. However, it actually turns out that mathematical expressions do not appear so frequently in computer programs to make it worth the trouble, and having a simple language turns out much greater an advantage.

As one can expect, the `(= amount 0)` expression evaluates to the logical truth, if the meaning of `amount` is 0 in that context, and otherwise it evaluates to the logical falsehood.

In the first case, the value of the `if` expression becomes `'()`, that is – an empty sequence. In the second case, the expression becomes:

[8] Although the concept might not be so new for us, because we have defined it informally in the previous section, it certainly is new for the Scheme interpreter.

```
(if (prime? from)
  (cons from (prime-numbers (- amount 1) (+ from 1)))
  (prime-numbers amount (+ from 1)))
```

This, again, is an `if` expression. Its `<condition>` clause takes form `(prime? from)`. The intent is, that `prime?` is a function that evaluates to the logical truth if its argument is a prime number, and to the logical falsehood if its argument is a non-prime number. We will explain later what we exactly mean by that.

It may be surprising for someone to see a punctuation mark being a part of a name of a concept. There is however a common convention among Scheme programmers to end the names of *predicates*[9] with question marks[10].

The `<then>` clause of the considered `if` expression is interesting. It has a form:

```
(cons from (prime-numbers (- amount 1) (+ from 1)))
```

There are four things to be noted here: first, the expression refers to the `cons` function, which is used for *constructing* sequences. Second, the second argument to `cons` refers to the notion of `prime-numbers`, which is currently being defined. This form of definitions is often called *recursive*. Third, we use the elementary mathematical operations of *addition* and *subtraction* in the same way that we used the numerical comparison earlier, i.e. using the prefix notation. Lastly, the fact that we pass `amount` decreased by one to the recursive call, in conjunction with the fact that the sequence of 0 prime numbers is empty, guarantees that our construction will terminate, provided there is enough prime numbers.

The `<else>` clause of the second `if` expression has a form:

```
(prime-numbers amount (+ from 1))
```

In other words, it establishes the fact that if n isn't a prime number, then the prime numbers starting from n are the same as the prime numbers starting from $n + 1$.

So far we have defined `prime-numbers` in terms of primitive notions, such as addition, subtraction, construction, conditionals and recursion. The only thing that's left is to explain what it means that a number is `prime?`.

We know from the mathematics that a prime number is a number whose only natural divisors are 1 and the number itself. We also know that a number cannot be divisible by a number greater than itself. Hence, in order to find a list of numbers, we can try all numbers starting from 1.

[9] i.e. functions whose values can either be logical truth or logical falsehood.

[10] A cautious reader probably noticed that the `=` function used before is also a predicate, although it's name doesn't end with a question mark. This lack of consequence isn't bothersome in practice though, because naming conventions are not as essential to the structure of a language as the rules given earlier.

```
(define (divisors n from)
  (if (> from n)
    '()
    (if (= (remainder n from) 0)
      (cons from (divisors n (+ from 1)))
      (divisors n (+ from 1)))))
```

This definition has a very similar structure to the previous one. We used some new elementary functions, namely the comparison predicate `>` (greater than) and the `remainder` operation.

Perhaps the program could be made clearer if we had defined what it means for a number to divide another number. However, if we decided to define a predicate `(divides? a b)`, it wouldn't be clear whether we mean that "`a` divides `b`" or that "`b` divides `a`" – this is an apparent limitation of the prefix notation.

This problem could be solved by adding another pair of parentheses in our definition:

```
(define ((divides? a) b)
  (= (remainder a b) 0))
```

Now it should be clearer that we ask whether `b` divides `a`. Although the core Scheme doesn't allow such notation, one of the modules contained in this pamphlet's repository legalizes it. From the semantic point of view, it defines `divides?` as a function of `a` whose value is a function of `b` that compares the remainder of the division of `a` by `b` with `0`.

Since we have means to construct the sequence of divisors of a given number, we can check whether the number is a prime simply by comparing its list of divisors with a list containing only the number 1 and the considered number itself:

```
(define (prime? number)
  (equal? (divisors number 1) (list 1 number)))
```

We have used here two new primitive functions: `equal?` and `list`. They are not entirely primitive, as they could be defined rather easily in terms of other operations (we will see how to do that shortly), but every Scheme implementation certainly provides those, so for now we can consider them as such.

The `list` function simply returns a list of values of its arguments. In this particular example, we could as well have written `(cons 1 (cons number '()))` instead of using `list`.

The `equal?` function is a predicate which says that two or more compound objects are identical.

In Scheme, there are a few distinct concepts of equality. The first that we have seen is called *numerical equality* and is expressed using the `=` symbol. The second that we've just seen above is called *identity*, and is expressed using the `equal?` symbol. The third is called *sameness* and is expressed using the `eq?` symbol.

It is guaranteed that two symbols having the same shape are `eq?` and that all empty sequences are `eq?`. However for various reasons it is not guaranteed that two instances of the same number are `eq?`. Two lists probably won't be `eq?` even if they have all the same elements in the same order – the reason for that is that usually each usage of `cons` or `list` creates a new object in the computer memory.

It is noteworthy that if two things are `eq?` or `=`, they will certainly be `equal?`, so the latter predicate is most general and can be used in most situations.

1.2.2 How do we know

So far, we have defined a few notions that were used in the formulation of the original problem. We defined them by means of recursion, so we had to worry only about two things: what would be the shape of the simplest instance of a given notion, and how do we build a more complex instance out of a simpler one.

It would be helpful to have some means of verifying whether our definitions actually work as expected. It turns out, that those means are also very simple: those are *substitution* and *reduction*.

Let's see how it works for the `divisors` function. To begin with something simple, let's see what is the denotation of `(divisors 4 1)`. We obtain the *connotation* of the expression by substituting the complex notion of `divisors` with its definition, replacing any formal parameters with their values. Thus we obtain:

```
(if (> 1 4)
  '()
  (if (= (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors n (+ 1 1))))
```

We see that the first condition is clearly false, because 1 is smaller than 4. We can therefore substitute the conditional with its `<else>` branch:

```
(if (= (remainder 4 1) 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors n (+ 1 1))))
```

If we substitute the expression `(remainder 4 1)` with its value, i.e. the number 0, it is apparent that the condition `(= 0 0)` is true, so we can replace the conditional with its `<then>` branch:

```
(cons 1 (divisors 4 (+ 1 1)))
```

It is also clear that `(+ 1 1)` is 2. We therefore go back to the initial situation, with the difference that our expression is now an argument to `cons` function, and its `from` argument has value 2 instead of 1.

By expanding the expression with its connotation and performing appropriate reductions, we obtain the following form:

```
(cons 1 (cons 2 (divisors 4 3)))
```

In next iteration of substitutions and reductions it will turn out that the remainder of 4 divided by 3 is non-zero, so the above expression can be rewritten as:

```
(cons 1 (cons 2 (divisors 4 4)))
```

As the remainder of the division of 4 by 4 is 0, we rewrite the above as:

```
(cons 1 (cons 2 (cons 4 (divisors 4 5))))
```

Now the `<condition>` of the outer `if` form is satisfied, so the `(divisors 4 5)` expression evaluate to empty sequence to yield:

```
(cons 1 (cons 2 (cons 4 '())))
```

which eventually evaluates to the sequence `(1 2 4)`. The assumptions under which we performed the above inference are called *the substitutional model of computation.*

1.2.3 Destructuring

So far, we managed to explain what we mean by the expression "initial N prime numbers" from our formulation of the original problem. The formulation was a bit more complex, because it asked to sum the squares of those numbers.

So in order to accomplish that task, we still need to explain what we mean by squares and their sum. Let's begin with the denotation of the expression "squares of Y". From our previous considerations we know that Y is a list of numbers. Intuitively, if Y were a list `(1 2 3)`, then the meaning of the outer expression would be a list `(1 4 9)`, that is, each element of the denotation of the expression "squares of Y" is a square of the corresponding element of Y.

A square of a number is – by definition – that number multiplied by itself:

```
(define (square number)
  (* number number))
```

To make it clear what the squares of a sequence of numbers are, or rather to explain how they are constructed, we need to notice that the squares of an empty sequence of numbers are an empty sequence.

As for a non-empty sequence, we know that it can be decomposed into its first element and the sequence of its remaining elements. We could therefore define squares of a non-empty sequence as a sequence, whose first element is a square of the first element of the input sequence, and whose remaining elements are the squares of the remaining elements of the input sequence (this statement sounds very tautological indeed, but we will see shortly how to deal with that)

```
(define (squares numbers)
  (if (eq? numbers '())
    '()
    (cons (square (car numbers)) (squares (cdr numbers)))))
```

The most important thing about this definition is that it reveals the means for destructuring lists. For historical reasons, the function whose value is the first item of a list is called `car`, and the function that yields the remaining of the list is called `cdr`.

It should be clear that for any values `a` and `b`, `(car (cons a b))` is `a` and `(cdr (cons a b))` is `b`.

The names `car` and `cdr` used to be popular, because they prompted contractions like `(caddr x)`, which was an abbreviation of `(car (cdr (cdr x)))`. Although some examples from this chapter will be written using those functions, a more powerful (and readable) technique will be presented in one of the later sections.

Looking back at our definition of `squares`, we notice that there's something essentially wrong with it. Linguistically, the word *squares* is just a plural form of *square*. This relation isn't reflected in our usage.

Suppose that we wanted to solve a problem of finding the cubes of a list of numbers, rather than squares. Following the above pattern, we would need to define the function `cube` that would explain what a cube of a number is. Then we would define a function `cubes` that would apply the function `cube` to each elements of a given list.

Such a practice would be very unhandy. We would rather wish to have a function, let's call it `plural`, that would take a function of one argument, and return a function of list of arguments:

```
(define ((plural concept) list)
  (if (eq? list '())
    '()
    (cons (concept (car list)) ((plural concept) (cdr list)))))
```

While it would be completely fine to have and use such function, historically things have gone differently, and the grammatical plural form is achieved using the `map` function[11]:

```
(define (map f l)
  (if (eq? l '())
    '()
    (cons (f (car l)) (map f (cdr l)))))
```

Therefore, "the squares of initial seven primes" should be written as `(map square (prime-numbers 7 1))`.

The only thing that's left is to explain what the sum of a list is. We should consider the boundary condition first: what would be the sum of an empty list? We know that no elements are zero elements, so we could expect that a sum of zero elements will be zero. Otherwise it's the value of the first element added to the value of the remaining elements[12]:

```
(define (sum numbers)
  (if (eq? numbers '())
    0
    (+ (car numbers) (sum (cdr numbers)))))
```

We can now present the solution to the original problem: "the sum of squares of initial 7 primes" is

```
(sum (map square (prime-numbers 7 1)))
```

Note the similarity between the formulation and the solution. This sort of similarity isn't reserved for the Scheme programming language, and can be achieved in virtually every programming language that allows to create functions[13]. The advantage of Scheme however is its simplicity: we already know almost everything about the syntax of our language, and our knowledge of its semantics is sufficient to construct many advanced programs.

The most important thing is that, unlike the imperative code given earlier, this code explains exactly what it is about: the reader doesn't need to recognize any implicit patterns, as everything has been laid out explicitly.

The only thing that may seem mysterious at first sight is the role of arguments to `prime-numbers`. We could add a comment that would explain that role.

Scheme has three types of comments. There are comments that begin with a semicolon (`;`) and end with the end of the line, block comments that

[11] Note that `map` is a built-in function, so there's no actual need to define it. Moreover, the built-in version is in fact better than our variant, so redefining it is strongly discouraged.

[12] We could also define `sum` without resorting to zero, by specifying it for lists whose length is at least one, but that would actually make the code more complicated.

[13] Some additional practical considerations may regard languages that do not automatically reclaim storage, such as C.

begin with `#|` and end with `|#`, and very useful comments that begin with `#;` and comment out exactly one expression that follows them. We could therefore enrich our solution with additional information:

```
(sum (map square (prime-numbers 7 #;greater-than-or-equal 1)))
```

1.2.4 Getting your hands dirty

So far, we have been using a formal notation to express certain mathematical notions in a precise manner – everything was going on on the paper and in our heads. There's yet another important advantage of our notation: it can be "comprehended" by computers, so on one hand, we have a practical instance of verification whether our beliefs regarding the meanings of certain expressions are right, and on the other – we can employ the computer to do some parts of reasoning for us. Of course, if you do not wish an inferior machine to perform the noble task of reasoning for you, you can still benefit from reading the pamphlet and exercising the strict notation.

There are many free practical implementations of Scheme. Probably the most popular and accessible is Racket, which comes with a very intuitive editor. Another nice implementation is MIT/GNU Scheme, that comes with a slightly less intuitive editor. There are implementations that focus on speed, such as Gambit or Stalin, and those that compile to some popular platforms like Java or JavaScript. There's also an implementation called Biwa Scheme, that can be run directly in a web browser.

The implementation that is going to be used in this text is called Guile. It is not so easy to install, but it is very convenient for practical interactions and incremental development. It integrates particularly well with the Emacs editor through the Geiser package. The only bad thing about Emacs is that it requires getting used to.

The detailed information regarding setting up Emacs, Geiser and Guile is platform-dependent and as such is beyond the scope of this pamphlet. An interested reader will easily find the required information on the web.

Assuming that everything is up and running, you can evaluate a single expression in a Scheme interpreter. The most common way is to enter an expression into a so-called *command line* or REPL (which stands for "read-eval-print loop"), however having a proper environment like Emacs with Geiser, it is possible to enter an expression in a text file and evaluate it using a magic keystroke[14].

Before we get anything done, we need to legalize the syntactic extensions that are used throughout this pamphlet: first, the additional modules need to be downloaded from the pamphlet's repository, available at `https://github.com/panicz/pamphlet` (the `libraries` directory), and placed

[14] By default, it's `C-x C-e` in Geiser.

somewhere in Guile's `%load-path`, and then we need to type in the expression:

```
(use-modules (ice-9 nice-9) (srfi srfi-1) (pamphlet))
```

Finally, we can feed the interpreter with all the necessary definitions: we need to explain what we mean by `divisors`, what it means for a number to be `prime?`, how to construct a sequence of `prime-numbers`, what is a `square` of a number and how to `sum` a list of numbers.

Then, we can ask the computer to give us the value of the expression `(sum (map square (prime-numbers 7 1)))`.

1.2.5 Looking down

So far, as we were referring to certain expressions of the English language whose meanings we were trying to paraphrase, we were usually quoting them. For example, we could say: "The meaning of the expression «first three natural numbers» is a sequence `(0 1 2)`". It is therefore typical, especially when talking about the meanings of linguistic expressions, to use the *quotation* operator.

In Scheme, the quotation operator is expressed using `quote` form. For example, the value of the expression `(quote (sum (map square (prime-numbers 7 1))))` is a sequence of two elements, whose first element is a symbol `sum`, and whose second element is a sequence of three elements, whose first element is a symbol `map`, second element is a symbol `square`, and third element is a sequence of three elements: a symbol `prime-numbers`, a number 7 and a number 0.

We've actually already seen an example of quotations in our definitions: it turns out that the operation of quotation is so useful that it has a notational shorthand: `'x` means the same as `(quote x)`, so our notation for the empty sequence – `'()` – could equivalently be written as `(quote ())`. Note that the `quote` operator is very different from the quotation marks that are often used in programming languages for constructing strings, because it allows us to quote structures, rather than create flat sequences of characters.

As you might have already noticed, the compound expressions in Scheme are also sequences[15]. This property, called *homoiconicity*, is one of the most valuable properties of Scheme, because it allows to treat programs as syntactic objects and write programs that transform other programs in Scheme rather easily.

[15]Note that we need to quote the empty sequence in order for it to mean the empty sequence, because the meaning of an empty expression is unspecified. Some dialects of Lisp blur this distinction and assume that the meaning of an empty expression is the empty sequence. This may be confusing though, and the Schemers are proud that their language is conservative in this regard

It also allows to create new languages with a common syntax. Or, as Alfred Tarski would put it, it allows to use Scheme as a *meta-language* for some other formal languages[16].

To illustrate that idea, we will describe the semantics of a simple formal system called *propositional logic*, which captures how compound sentences are built from simpler ones. The simplest units of propositional logic are called *atomic formulas* – they stand for propositions that aren't analyzed in simpler terms, and that can only be *asserted* or *rejected*.

The expressions of propositional logic are either propositional variables $p, q, r, ...$ standing for atomic formulas, or junctions of expressions: $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \Rightarrow \psi$, $\phi \equiv \psi$, $\neg\phi$. These compound expressions mean disjunction ("ϕ or ψ"), conjunction ("ϕ and ψ"), implication ("if ϕ then ψ"), equivalence ("ϕ if and only if ψ"), and negation ("it is not the case that ϕ"), accordingly. For the remainder of this example we shall be concerned only with conjunction, disjunction and negation.

The semantics of propositional logic determines the logical value (i.e. truth or falsehood) of each formula with respect to some given *valuation*, i.e. a mapping from propositional variables to logical values.

If a formula is atomic, then its value is simply specified by the valuation. Otherwise it is a compound formula: a negation, a disjunction or a conjunction. If it is a negation, then it is true only if the negated formula is false; if it is a disjunction, then it is true only if at least one of its sub-formulas is true; finally, if it is a conjunction, then it is true only if all its sub-formulas are true.

A formula is satisfiable if there exists a valuation under which it is true. A formula is logically valid if it is true for any valuation. For example, $p \vee \neg p$ is a valid formula, because if p is true, then the whole disjunction is true, and if p is false, then $\neg p$ is true and the disjunction is true as well. It is easy to see that if a formula is valid then its negation is not satisfiable.

Testing whether a given valuation satisfies a given formula can be a tedious task that can be performed more efficiently on a computer.

We will use our fully parenthesized Polish notation to represent formulas of propositional calculus. We will write `(and` ϕ ψ`)` to mean conjunction, `(or` ϕ ψ`)` to mean disjunction and `(not` ϕ`)` to mean negation.

Note that conjunction and disjunction both possess an algebraic property of *associativity*, i.e. $(\phi \circ \psi) \circ \chi = \phi \circ (\psi \circ \chi)$ for $\circ \in \{\wedge, \vee\}$. We can therefore interpret `(and` ϕ_1 ϕ_2 `...` ϕ_n`)` as $\phi_1 \wedge \phi_2 \wedge ... \wedge \phi_n$ with no ambiguity. The same applies to disjunction.

Having a fixed representation, we can explicate the conditions under

[16] It is also possible to use Scheme as a meta-language for itself – this technique, called *meta-circular evaluation*, has been explored in the grand book *Structure and Interpretation of Computer Programs*[1]. This idea was at the heart of the seminal paper *Recursive Functions of Symbolic Expressions and their Computation by Machine* by John McCarthy[4], which gave birth to the predecessor of Scheme called Lisp.

which a formula is `satisfied?` under a given valuation. However, we need to find some way to represent a valuation. One obvious representation of a mapping is by enumerating the name-value pairs. For example, if there are three distinct atomic formulas p, q, r in ϕ, one of the eight possible valuations could be written as `'((p . #t)(q . #f)(r . #t))`. The `#t` and `#f` inscriptions represent the logical truth and falsehood in Scheme.

Notice the strange dot that appears between each key and its corresponding value. This is because we said that we will represent the mapping using a key-value *pairs*, rather than two-element lists. We could as well have chosen the latter representation, but we didn't, so that the strange dot could appear.

We said earlier that `cons` is a primitive function that constructs lists. However, it would be more accurate to say that `(cons a b)` creates (or *allocates*) a new pair, whose *left value* (or `car`) is its first argument, `a`, and whose *right value* (or `cdr`) is its second argument, `b`.

A list is therefore either an empty sequence, or a pair whose right value is a list[17].

Now it turns out that the list `(1 2 4)` could equivalently be written as `(1 . (2 . (4 . ())))`[18], and similarly, the `(cons a b)` expression could be written as `(cons . (a . (b . ())))`. The list notation is therefore a shorthand for a pair notation[19].

So under this representation, if we wish to obtain a value of a given atomic formula from a given valuation, we may need to use the `lookup` procedure:

```
(define (lookup key #;in mapping)
  (let* ((this (car mapping))
         (remaining (cdr mapping))
         (name (car this))
         (value (cdr this)))
    (if (eq? name key)
      value
      (lookup key remaining))))
```

We see a new construct here – the `let*` form. The form is useful for naming the intermediate components of the object that we are trying to obtain. We could equivalently have written:

[17] A nesting of pairs whose last right value is not the empty list is called *improper list.*

[18] Note the spaces between the symbols and the dot: they are important, because it is fine to use the dot as a part of a symbol or a number.

[19] Note that the distinction of the left element of the pair as a first element of the list is only a matter of convention – for example, if Lisp was invented in the Arabic countries, it could have been the other way around.

```
(define (lookup key #;in mapping)
  (if (eq? (car (car mapping)) key)
    (cdr (car mapping))
    (lookup key (cdr mapping))))
```

but this only increases the nesting level of expressions and makes the code more difficult to read. We could also have introduced intermediate names using local `define` forms:

```
(define (lookup key #;in mapping)
  (define this (car mapping))
  (define remaining (cdr mapping))
  (define name (car this))
  (define value (cdr this))
  (if (eq? name key)
    value
    (lookup key remaining)))
```

The extent of the inner `define` forms is limited to the outer `define` form. However, the structure of the code is more difficult to follow, as it contains many obscure symbols.

Finally, I promised before that we won't be using the `car` and `cdr` functions, because our code doesn't deal with vehicles nor compact disc recording. Instead we can use the marvelous feature of *pattern matching*:

```
(define (lookup key #;in mapping)
  (let* ((((name . value) . remaining) mapping))
    (if (eq? name key)
      value
      (lookup key remaining))))
```

This definition is much more compact than the original one, but may be slightly more difficult to follow. It simply decomposes the structure of `mapping`, i.e. that it is a pair whose left value is a pair, and names its parts accordingly: the left value of the left value of `mapping` is named `name`, the right value of the left value of `mapping` is named `value`, and the right value of `mapping` is named `remaining`.

Note that all the above variants of the `lookup` function assume that there has to be an element whose left value is `key` somewhere in the mapping, and it is an error if that condition isn't satisfied (because it is an error to talk about the left or right value of something that isn't a pair).

Equipped with the `lookup`, we can go back to the original problem and explain what it means for a proposition to be satisfied:

```
(define (satisfied? formula #;under valuation)
  (match formula
    (('and . clauses)
     (every (lambda (clause)
              (satisfied? clause #;under valuation))
            clauses))
    (('or . clauses)
     (any (lambda (clause)
            (satisfied? clause #;under valuation))
          clauses))
    (('not clause)
     (not (satisfied? clause #;under valuation)))
    ((? symbol?)
     (lookup formula #;in valuation))))
```

There are many new things going on in here: the main part of the definition is the `match` expression, which – in general – takes the following form:

```
(match <expression>
  (<pattern-1> <value-1>)
  (<pattern-2> <value-2>)
  ...)
```

The first `<pattern>` is `('and . clauses)`. It matches against all pairs whose left element is the symbol `and`, and if the match succeeds, the right element of the pair gets bound with the name `clauses`. So this part of code could be rewritten as[20]:

```
(if (and (pair? formula) (eq? (car formula) 'and))
  (let* ((clauses (cdr formula)))
    <value-1>)
  (match formula
    ((<pattern-2> <value-2>)
     ...)))
```

The `<value>` part from the scope of the first `<pattern>` is `(every (lambda (clause) (satisfied? clause valuation)) clauses)`. The function `every` takes a predicate and a list, and evaluates to the logical truth if the predicate is satisfied for every element in the list.

[20] Technically speaking, the code is equivalent only if the `formula` always evaluates to the same value. In other cases, we would need to capture the value of `formula` before expansions.

The first argument to `every` is a `lambda`[21] expression. The `lambda` expression takes the form:

```
(lambda <arguments> <body>)
```

It creates a new anonymous function. Actually we've already used `lambda` expressions before, albeit implicitly: the `(define (f . <args>) <body>)` is equivalent to `(define f (lambda <args> <body>))`.

The second `(<pattern> <value>)` is similar, except that it uses the `any` function. The only unobvious part that is left is the `(? symbol?)` pattern. This pattern matches only if the `formula` satisfies the predicate `symbol?`, or in other words, the pattern is matched if it is the case that `(symbol? formula)`.

1.2.6 Building structures

The `match` expression isn't a native Scheme form. It is a "specialized language" for destructuring complex objects, and it was added to Scheme through its powerful syntax extension mechanism.

So far we have been using the `cons` or `list` functions for creating complex structures. I promised that in the remainder of this text we won't be using `car` and `cdr`, because `match` allows to do the same task better.

We usually won't be using `cons` as well. There's yet another "specialized language" that is meant for building complex objects. It makes use of the fact, that – just as the notation `'x` is a shorthand for `(quote x)` – the notation `` `x `` is a shorthand for `(quasiquote x)`, `,x` is a shorthand for `(unquote x)`, and `,@x` is a shorthand for `(unquote-splicing x)`.

Those words alone, `quasiquote`, `unquote` and `unquote-splicing`, mean nothing. As with every other word in Scheme, you can make it mean whatever you want. However, the `quasiquote` by default is a syntactic extension that gives a special meaning to the other two words, and allows to quote only a part of a list. For example, the expression `` `((+ 2 3) ,(+ 2 3)) `` evaluates to a list `((+ 2 3) 5)`, and the expression `` `(0 ,@(divisors 4) 5) `` evaluates to `(0 1 2 4 5)`.

So instead of writing `(cons a b)`, we shall write `` `(,a . ,b) `` from now on.

[21] One has to admit that `lambda` is a rather strange name. The reason for this weirdness is that the semantics of Scheme is based on a very simple powerful model of computation known as λ-calculus. Allegedly, its name is a result of editor's mistake, and dates back to the logical *opus magnum* of Bertrand Russell and Alfred North Whitehead, *Principia Mathematica*, where they used the notation $f(\hat{x})$ to mean a function of x, rather that the value of f in the point x. Alonzo Church modified the notation by taking the $\hat{x}$ before the function application, i.e. $\hat{x}.f(x)$. A typesetter wrote that as $\Lambda x.f(x)$, someone read that as the Greek letter *lambda*[2].

1.2.7 Conclusion

At this point, we know enough about Scheme to use it to represent advanced notions and to solve some practical problems.

2

Genetic Algorithms

> *"It is absolutely safe to say that if you meet somebody who claims not to believe in [what Richard Dawkins claims to believe], that person is ignorant, stupid or insane."*
>
> – Richard Dawkins

In the previous chapter, we defined what it means for a proposition to be satisfied under a given valuation. We also said informally, what it means for a propositional logic formula to be satisfiable – namely, that there exists a valuation such that the formula is satisfied.

This formulation is rather easy to comprehend, but in practice the amount of checks that we may need to perform to see if a formula is satisfiable may grow exponentially with the number of atomic propositions occurring in the formula: adding another atomic proposition doubles the number of possibilities, so the problem quickly becomes intractable.

The problem isn't just theoretical. Although propositional calculus is extremely simple, it has some useful practical applications in software verification.

Besides, despite the fact that the general solutions fail for more complex formulas, one can frequently find some *heuristics* that perform very well for the most common cases.

2.1 Biological inspiration

One of such heuristics is called *genetic algorithms*. It is inspired by the process of natural selection. Most generally, the idea is to encode the solution as a sequence of *chromosomes* that behave as binary switches, where each switch splits the search space in half.

The solutions are searched in larger groups called *populations*. Each specimen of a population is evaluated using a so-called *fitness function*, which measures the quality of a solution. Rather than saying that a given specimen is good or bad, the fitness function assigns it a number.

2.1.1 Chromosomal crossover

During the evolution process, the best solutions are combined with each other, and the worst ones are rejected. The hope is that if we combine a few individuals into a new one, it has a chance to inherit the good traits of its originals. In practice, the chromosomes of couples of the best specimens from the population are recombined in a *crossing-over* process: therefore the new individuals can be regarded as an offspring of the old ones.

The crossing-over can be defined as a function that takes two chromosomes, splits them at a random point and returns a new one composed of the prefix of the first one and the suffix of the second:

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))
```

Of the things that deserve attention, the first one is the presence of an *assertion*. An assertion essentially does nothing more than informing the reader about the assumptions that the author made about the intended usage of a function – in this particular case, he assumed that the two arguments – `mommy` and `daddy` – are lists of equal length.

Then we choose a random[1] position from the range between 0 and *the length of chromosome* minus 1. We use the prefix of the `daddy` chromosome (for example, if `daddy` is `(a b c d e)`, then `(take daddy 3)` evaluates to `(a b c)`) and the suffix of the `mommy` chromosome (similarly, if `mommy` is `(u v w x y)`, then `(drop mommy 3)` evaluates to `(x y)`).

Finally, we construct a new list from the prefix and the suffix.

Now that we've seen the intimate details of copulation, we can have a look at the process from the social perspective.

2.1.2 The ceremony of procreation

Whether a given specimen becomes a daddy or a mommy is a matter of luck, and it may happen that the same specimen serves as a mommy during an intercourse with one partner, and as a daddy with another. It is important however, that the specimen with the high social status (i.e. better fitness function) procreate more often than the ragtag.

[1] Each occurrence of the `(random n)` expression can be replaced with a value randomly chosen from the range 0 to $n - 1$. This *indeterminism* is the main reason why I prefer the term `expression-based programming` to `functional programming`. Disallowing such non-deterministic expressions would make many simple things complex.

Also, contrary to what many catholic priests say, it is perfectly fine when a specimen copulates with itself: when it happens, its child is an exact copy of its sole parent (however in practice this privilege is reserved to the elite).

The ceremony proceeds as follows. First, a census is prepared, where each member of the population is assigned a value of a fitness function. Then, the census is sorted according to that value. Subsequently, the lottery drawing occurs, where each member of the population draws an opportunity to copulate. A single specimen can appear on the list more than once, and the probability of being enrolled to the list is higher for the specimens that are higher on the list. The length of the list equals the population.

Finally, each one of those lucky beggars is assigned a partner from another list generated in the same way (so, as it was said before, it can happen that the he-and-she spends the night with a *very* familiar company), and then the new generation comes to the fore.

This cycle can repeat for a few hundreds of times or more, until the best man from the population is satisfactory to its transcendent creator.

It can also happen, say, once in a generation on average, that a chromosome of a specimen is mutated.

The social perspective on the ceremony of copulation can be expressed in Scheme in the following way:

```
(define (procreate population social-status)
  (let* ((census (map (lambda (specimen)
                        `(,(social-status specimen) . ,specimen))
                      population))
         (social-ladder (sort census (lambda ((a . _) (b . _))
                                       (> a b))))
         (population (map (lambda ((status . specimen)) specimen)
                          social-ladder))
         (size (length population))
         (males (biased-random-indices size))
         (females (shuffle (biased-random-indices size)))
         (offspring (map (lambda (man woman)
                           (cross-over (list-ref population man)
                                       (list-ref population woman)))
                         males females)))
    (map (on-average-once-in size (mutate #;using not)) offspring)))
```

Note that the arguments to `lambda` provided to the `sort` function are destructured (pattern-matched) in place. This is a convenient syntactic extension to Scheme that replaces the built-in `lambda`.

Also, we provided three arguments to `map`, rather than two – one function of two arguments and two lists. The value of the expression will be a list whose elements are values of the function for the first element from the first

list and the first element of the second list, the second element of the first list and the second element of the second list, and so on. It is assumed that the length of both lists is the same – and that will also be the length of the result.

We will learn how to define functions of variable number of arguments later. For now it is sufficient to know that it is possible to do so, which can often be convenient.

It is apparent that the `cross-over` function is given two arguments that result from the invocation of the built-in `list-ref` function, which takes a list of length $n > 0$ and an index i (starting from 0) and returns the i-th element of the list.

The procedure `biased-random-indices` generates a list of indices with the desired distribution:

```
(define (biased-random-indices size)
  (if (= size 0)
      '()
      `(,(random size) . ,(biased-random-indices (- size 1)))))
```

The length of the generated list will be `size`. Furthermore, the index 0 is guaranteed to appear on the list as the last element, has a 50% chance to appear as the penultimate element, one third to appear one before, and so on, so it will almost certainly appear on the list more than once. On the other hand, the last index can appear only as the first element of the list, but this opportunity is shared evenly by all the other indices, so it is rather unlikely that it will ever happen. This strategy is called "equal opportunity policy" by the social ideologies.

Two separate lists are created for `males` and `females`. In order to increase the variation of the population, the list of `females` is shuffled, which gives them a chance to advance to the upper class.

The shuffling proceeds as follows: if the list has more than one element, then a random pivot point is chosen, and the (recursively shuffled) sub-lists are swapped with a 50% probability. Otherwise the shuffled list is identical with the original one.

```
(define (shuffle l)
  (match (length l)
    (0 '())
    (1 l)
    (n (let ((left right (split-at l (random n))))
         (if (= (random 2) 1)
             `(,@(shuffle right) ,@(shuffle left))
             `(,@(shuffle left) ,@(shuffle right)))))))
```

The last thing that requires an explanation is the mutation. We wish it to occur on average once per generation, therefore we define a function that applies a given function with a probability $\frac{1}{n}$ for a given n, where n is a positive integer:

```
(define ((on-average-once-in n action) arg)
  (assert (and (integer? n) (> n 0)))
  (if (= (random n) 0)
      (action arg)
      arg))
```

The `action` of `mutation` that we wish to apply inverts the random chromosome:

```
(define ((mutate how) specimen)
  (let* ((n (random (length specimen)))
         (mutation (how (list-ref specimen n))))
    (alter #;element-number n #;in specimen #;with mutation)))
```

The meaning `(alter n l value)` is a list that contains all the same elements as `l`, except its `n`-th element, which has a new `value`.

2.1.3 Evolution

The struggles of a sole generation in the quest for the perfect society are unlikely to bring satisfactory results (especially if these struggles boil down to copulation). Only within the span of many lifetimes can the true value arise. It is entirely up to the creator to decide many cycles of lives and deaths will the world witness, how complex can its inhabitants be, and how many of them will be brought to existence.

```
(define (evolve population #;towards criterion #;for iterations)
  (assert (and (integer? iterations) (>= iterations 0)))
  (if (<= iterations 0)
      population
      (evolve (procreate population criterion)
              #;towards criterion
              #;for (- iterations 1))))

(define (generate-specimen dimension)
  (generate-list dimension (lambda () (= (random 2) 0))))

(define (generate-population size dimension)
  (generate-list size (lambda () (generate-specimen dimension))))
```

```
(define (optimize dimension population-size iterations criterion)
  (let* ((population (generate-population population-size dimension))
         (modern-society (evolve population #;towards criterion
                                 #;for iterations)))
    (argmax criterion modern-society)))
```

The `generate-list` procedure takes a `lambda` expression of no arguments that evaluates nondeterministically to some value (in our case, it is either logical truth or logical falsehood) and generates a list containing a specified number of results of such evaluation:

```
(define (generate-list n generator)
  (assert (and (integer? size) (>= size 0)))
  (if (= n 0)
      '()
      `(,(generator) . ,(generate-list (- n 1) generator))))
```

Lastly, the `argmax` is a library function that takes a measure function and a list and evaluates to the element of the list that has the greatest measure.

2.2 Solving the SAT

In the previous section, we have presented a heuristic for optimizing certain problems with reasonable (linear) computational means. In this section, we will apply that heuristic to the problem of satisfiability in propositional logic.

2.2.1 Parsing DIMACS CNF files

We are not going to operate in vacuum – there are many resources on the Web that provide the information and databases of examples. However, the format in which the examples are encoded may vary. One such format, proposed by the Center for Discrete Mathematics and Theoretical Computer Science, is called DIMACS CNF (for *Conjunction Normal Form*), and is encoded in text files.

A DIMACS CNF file is line-based. If a line begins with the `c` character, then it is a comment and shall be ignored. A first line of the DIMACS CNF file that is not a comment should take the form:

`p cnf <variables> <clauses>`

where `<variables>` specifies the maximum number of atomic propositions that appear in the formula, and `<clauses>` specifies the number of disjunctive clauses in the formula, or the number of meaningful lines that follow the given line.

After the header, there appear `<clauses>` lines that contain sequences of integers (possibly negative) separated with white space and terminated with zero. A number n means that an atomic formula appears in a given disjunctive clause, or – if it is negative – that it negation appears in the disjunctive clause.

This may sound complicated, so it can be instructive to see an example. The file

```
c simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

would correspond to the formula

`(and (or x1 (not x3)) (or x2 x3 (not x1)))`

As we can see, the whole expression is a conjunction (this is why it's called *conjunctive* normal form), and its sub-expressions are disjunctions. There are two of them, as there are lines below the DIMACS CNF header. The first line contains numbers `1 -3 0`, which correspond to the expression `(or x1 (not x3))`, and the second line contains numbers `2 3 -1 0` that correspond to the expression `(or x2 x3 (not x1))`.

It can be puzzling why someone decided to be able to represent only conjunctions of disjunctions of formulas and their negations, but it turns out that for every formula of classical logic one can find an equivalent conjunctive normal form. The details are beyond the scope of this pamphlet.

We therefore need to find a way to transform the numbers given in the lines of the DIMACS CNF file into S-expressions.

We will be using the `(ice-9 regex)` library bundled with Guile and its ability to process *regular expressions*. The regular expressions will be exposed here rather than explained, as the more detailed explanation can be found rather easily among the vast resources of the Worldwide Web.

Also, we are going to process the file line by line. The Scheme itself doesn't provide facilities for doing that, but Guile comes with the `(ice-9 rdelim)` library which provides a function that returns a next available line from a given `input-port`. We will fix the `input-port` to mean `(current-input-port)`, which is a notion similar to *standard input* known from UNIX and its descendants.

We shall gather the input lines from the list. The function terminates when it approaches an *end-of-file object*:

```
(define (input-lines)
  (let* ((line (read-line (current-input-port))))
    (if (eof-object? line)
        '()
        `(,line . ,(input-lines)))))
```

We shall ignore all the lines that contain anything else than (possibly negative) numbers separated with white-spaces and terminated with 0. For that, we will use the `filter` function which takes a predicate and a list, and returns a new list that contains only those elements from the old list that satisfy the predicate (preserving the order). Thus, the expression

```
(filter (lambda (line)
          (string-match "^\\s*(-?[0-9]+\\s+)+0\\s*$" line))
        (input-lines))
```

will evaluate to a list that contains only the meaningful lines. Note that we used the `string-match` procedure here, that takes a regular expression and a string (I didn't mention it before, but Scheme also provides strings as its elementary data type) and evaluates to truth-ish value if the string matches the regular expression[2]. The meaning of the regular expression is as follows. The `^` character anchors the regular expression at the beginning of the string. Then, `"\\s*"` allows a sequence of white spaces to appear at the beginning. The group `"(-?[0-9]+\\s+)+"` requires that there will appear (one or more times) a non-empty sequence of digits followed by a non-empty sequence of white-spaces, and that it can be prepended with the minus sign. Lastly, the expression `"0\\s*$"` says that `0` has to appear at the end of the sequence, and that it can optionally be followed with a sequence of white characters.

We now have a sequence of valid lines that need to be converted to propositional logic expressions. If a line has a form of a string `"2 3 -1 0"`, then if we split the line by spaces, we obtain a list of strings `("2" "3" "-1" "0")`. We can then further convert each of those strings (but the last one) to number using the built-in `string->number` function. We used the plural form, so it will probably require the use of `map`.

Lastly, we need to convert the number to an expression, that is – either a symbol or a negation of a symbol. We can therefore define:

```
(define (number->expression number)
  (if (negative? number)
    `(not ,(number->expression (- number)))
     ((numbered-symbol 'x) number)))
```

[2] We said that the logical truth is expressed using the `#t` syntax, but in fact any value other than `#f` is regarded as true in the context of the `if` form and its derivatives.

where `numbered-symbol` could be defined in terms of the primitive Scheme functions:

```
(define ((numbered-symbol symbol) number)
  (symbol-append symbol (string->symbol (number->string number))))
```

Putting it all together, we can define our DIMACS-CNF processor:

```
(define (process-dimacs-cnf)
  (define (disjunction line)
    (let* (((strings ... "0") (filter (lambda (s)
                                        (not (equal? s "")))
                                      (string-split line #\space)))
           (numbers (map string->number strings))
           (expressions (map number->expression numbers)))
      `(or . ,expressions)))
  (let* ((data-lines (filter (lambda (line)
                               (string-match
                                "^\\s*(-?[0-9]+\\s+)+0$"
                                line))
                             (input-lines))))
    `(and . ,(map disjunction data-lines))))
```

The only thing that can be unobvious is binding the value of the expression `(string-split line #\space)` to the pattern `(strings ... "0")`. What it actually does is that it skips the last element (that is assumed to be `"0"`) from the result of `string-split`. The `...` operator for the pattern matcher behaves in a complementary way to the `unquote-splicing` operator, so for example, in the expression

```
(match '(1 2 3 4 5 6)
  ((a b c ... z))
  `(,a ,b ,@c ,z))
```

`a`,`b` and `z` will be bound with `1`, `2` and `6`, respectively, and the `c` name will be bound with the list `(3 4 5)`, so the value of the above expression will simply be the list `(1 2 3 4 5 6)`. The `...` operator cannot appear more than once in a list, so for example patterns like `(a ... k ... z)` are illegal, as they would result in an ambiguous match. The `...` operator has another magical property – in the pattern

```
(match '((a . 1) (b . 2) (c . 3))
  (((keys . values) ...)
   `(,keys ,values)))
```

the `keys` symbol will be bound with the list `(a b c)`, and the `values` symbol – with `(1 2 3)`, so the list subjected to pattern matching will be *unzipped*. We will be making a use of this feature later.

Back to our parser, note that we used the value `#\space` as the second argument to `string-split`. It is a Scheme name for the "space" character (characters are also a data type in Scheme). Note also that we had to apply another filter on the result of the `string-split`, in order to remove empty strings from it, that appeared if there were two or more consecutive spaces in a given line.

2.2.2 Applying the genetic strategy

Now that we know how to convert the DIMACS CNF files to Scheme, we can test whether the provided formulas are satisfiable – or, to put it in another way – seek for a valuation under which they are satisfied.

It is rather obvious that our chromosomes will denote the values of subsequent atomic propositions for the valuations (therefore, they will be rather straightforward to decode). We need to know how many distinct atomic formulas are there in the main formula:

```
(define (atomic-formulas proposition)
  (match proposition
    (('not clause)
     (atomic-formulas clause))
    ((operator . clauses)
     (delete-duplicates (append-map atomic-formulas clauses)))
    ((? symbol?)
     `(,proposition))))
```

The code uses two library functions, `delete-duplicates` (that does exactly what it says[3]) and `append-map`, which is like `map` except that its functional argument is expected to return a list, and that list is *appended* to the resulting list, rather than *inserted* as a single element.

The structure of the formulas – the conjunctive normal form – also prompts us with the fitness function: that will be the number of the satisfied disjunctive sub-clauses.

[3] It is very important in programming to choose appropriate names for concepts, whether ultimate or intermediate, because knowing that we can trust the names, we don't need to resort to documentation.

```
(define (number-of-satisfied-subformulas #;of cnf #;for chromosome)
  (let* ((variables (generate-variables #;from 1 #;to (length chromosome)))
         (valuation (map cons variables chromosome))
         ((’and . or-clauses) cnf))
    (count (lambda (subformula)
             (satisfied? subformula #;under valuation))
           or-clauses)))

;; where
(define (generate-variables #;from first to #;last)
  (if (> first last)
    ’()
    ‘(,((numbered-symbol ’x) first)
      . ,(generate-variables #;from (+ first 1) #;to last))))
```

Here, we use the library function `count` that takes a predicate and a list and returns the number of elements of the list that satisfy the predicate.

The only thing that's left is to put the pieces together and enjoy the show[4]:

```
(let* ((formula (with-input-from-file "dubois20.cnf"
                  process-dimacs-cnf))
       (dimension (length (atomic-formulas formula)))
       (measure (lambda (chromosome)
                  (number-of-satisfied-subformulas #;of formula
                                                   #;for chromosome))))
  (optimize dimension 160 100 measure))
```

[4]The file "dubois20.cnf", among others, was obtained from the website `http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`.

3

Fuzzy Logic

"A lie told often enough becomes the truth."

– Vladimir Ilyich Ulyanov

3.1 Basic concepts

In the previous two chapters, we have been inquiring into the issue of satisfiability of propositional calculus. We expressed the notion of satisfiability by means of the notions of truth and falsehood, that were represented by the Scheme special values `#t` and `#f`.

It is customary however to represent these notions using numerical values: truth is usually represented using 1, and falsehood – using 0. If we view it that way, the conjunction can be perceived as a particular case of the *minimum* function, and disjunction – as an instance of the *maximum* function. Furthermore, the negation can be interpreted as the function $1-x$, where x is the logical value of the negated formula.

This observation could prompt someone to allow propositional formulas to take any real value between 0 and 1, for example 0.785398, because then the basic junctions would generalize naturally to support those values.

However unrealistic it may sound, this actually happened: in the 1960. Lotfi Zadeh proposed a theory of *fuzzy sets*, where the aforementioned observation found its application. Unlike in the traditional set theory, where a given element either belongs to a given set, or it does not, Zadeh proposed a theory where an element can belong to a set with a certain *degree*, and each set is characterized with a *membership function* that describes how certain elements (usually being real numbers) belong – or not – to that set.

3.2 Criticism

Having formulated a theory, Zadeh tried to find an application for it. Therefore he announced that his theory models the way in which *vague expressions* behave in the natural language, and how people perform *approximate reasoning*.

Worst of all, Zadeh really believed it. In the justification of his work, he claimed that "a natural language is basically a system for describing perceptions"[6], not even caring how is the sentence "a natural language is basically a system for describing perceptions" a description of a perception. Reading Zadeh's work, one gets the sense that, trying to prove the value of his idea, he indeed started to reason in a vague and imprecise way.

Among the examples that Zadeh gives is the notion of "tallness". For example, most people will admit that a person that is two-meters tall could be called a tall person. On the other hand, if a person has 1.5 meter, he will unlikely be called a tall person. It is however impossible to set the limit between the height to be considered tall and non-tall: we are more likely to admit that the shift between tallness and non-tallness occurs gradually.

Zadeh claims that the accurate description is to provide a membership function that would describe the interpolation between tallness and its negation.

The problem is that it is that there could be many competing membership functions that differ only in some tiny details, but there is no criterion to prefer any one of these functions over another.

A deeper problem is that the whole "thought experiment" makes absolutely no sense: those seemingly "vague" descriptions usually serve the practical purpose of designating an object of interest. We have no problem to understand an expression "the tall midget" when presented a group of midgets of varying height. But we are rarely faced with the task of demarcating between being tall and being non–tall.

Although the notion of fuzzy logic is very susceptible to criticism that it has been receiving ever since it was formulated, for some reason it also received a lot of attention among people who dealt with Artificial Intelligence, and some practical tools were built based on that theory (It may be impressive to some that this actually succeeded, but in practice the results were usually much less efficient than they could be if the classical control theory was applied). Because of that, it will also be presented here, albeit in a rather flaccid way.

3.3 Exposition

The remainder of this chapter will assume some basic intuitions concerning real-valued functions of real variables. The humanity won't suffer if you

decide to skip it. (Make however sure to read the section regarding the notion of `equivalence-classes`.)

Also, if you wish to get a detailed explanation of the presented method, it's best if you look elsewhere.

We will use fuzzy logic to implement a virtual *underwriter*, whose purpose is to give a rating of potential clients for an insurance agency based on a set of parameters, such as *Body-Mass Index*, the level of *glycated hemoglobin* in organism or the *blood pressure*.

The underwriter performs the inference based on a set of *fuzzy rules*, like "if BMI is *underweight* or BMI is *obese* or the glycated hemoglobin level is *low*, then the rating is *refuse*".

We could define the set of rules for the considered problem in the following way.

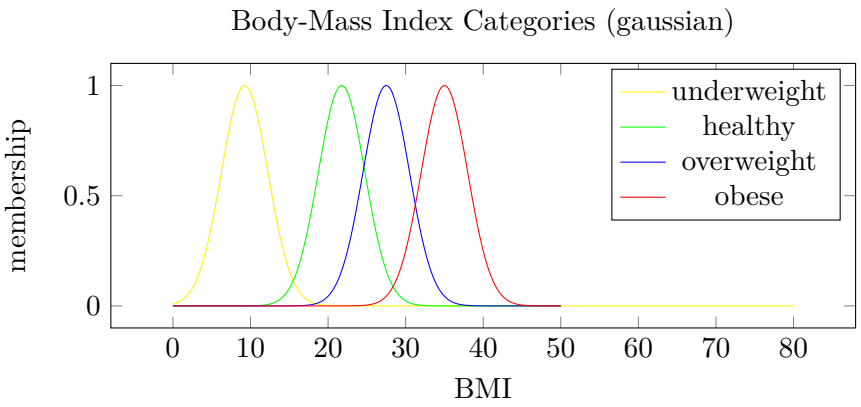
```
(define underwriter-rules
  '((if (or (is bmi underweight) (is bmi obese)
            (is glycated-hemoglobin low))
        (is rating refuse))
    (if (or (is bmi overweight) (is glycated-hemoglobin low)
            (is blood-pressure slight-overpressure))
        (is rating standard))
    (if (and (is bmi healthy) (is glycated-hemoglobin normal)
             (is blood-pressure normal))
        (is rating preferred))))
```

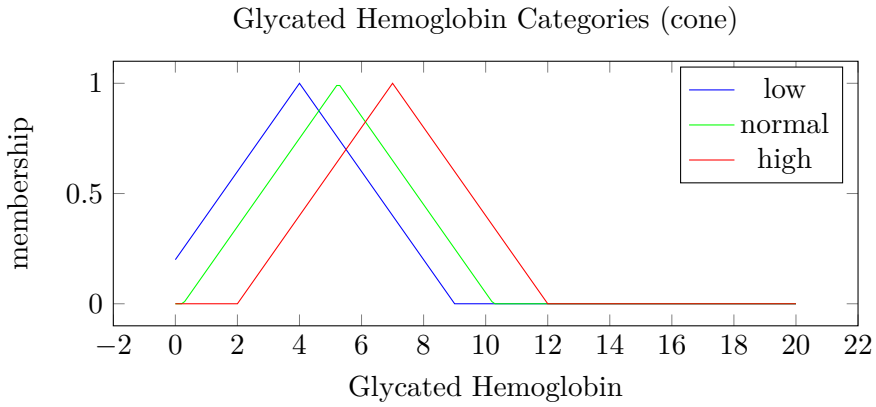
An inquisitive reader could ask an uncomfortable question: why should we consider this set of rules as valid, rather than some other one? The slack-off answer is that this particular set was specified by an experienced professional who encoded his or her intuitions acquired throughout the years of practice.

Now we need to define what it means for the glycated hemoglobin to be normal or low:

```
(define health-categories
  `((bmi (underweight ,(gaussian 9.25 3.0))
         (healthy ,(gaussian 21.75 3.0))
         (overweight ,(gaussian 27.5 3.0))
         (obese ,(gaussian 35.0 3.0)))
    (glycated-hemoglobin (low ,(cone 4.0 5.0))
                         (normal ,(cone 5.25 5.0))
                         (high ,(cone 7.0 5.0)))
    (blood-pressure (normal ,(gaussian 0.0 2.5))
                    (slight-overpressure ,(gaussian 10.0 2.5))
                    (overpressure ,(gaussian 20.0 2.5))
                    (high-overpressure ,(gaussian 30.0 2.5)))
    (rating (refuse ,(cone 10.0 5.0))
            (standard ,(cone 5.0 5.0))
            (preferred ,(cone 1.0 5.0)))))
```

An inquisitive reader could ask an uncomfortable question: what makes us choose these shapes of membership functions rather than some other ones? The slack-off answer would be that the data was prepared and evaluated by a team of highly educated medical experts who spent years studying the complicated machinery of human body and now are able to present their knowledge and linguistic intuitions in a highly digestive form of a few functions of a single variable. Even if you didn't trust that guy from the insurance, it would be insane not to trust the medicals, or wouldn't it?





It should be instructive to see the plots of at least some of the membership functions, to appeal to our visual intuitions. Also, for the practical purposes it is inevitable to define the formulas for the cone and the gaussian curve:

```
(define ((gaussian center deviation) x)
  (exp (- (/ (square (- x center))
             (* 2 (square deviation))))))

(define ((cone center radius) x)
  (cond ((<= x (- center radius))
         0)
        ((<= x center)
         (/ (+ x (- radius center)) radius))
        ((< x (+ center radius))
         (/ (+ (- x) center radius) radius))
        (else
         0)))
```

In the definition of `gaussian` we refer to the built-in exponential function `(exp x)`, or e^x.

The definition of the `cone` function contains a `cond` clause that we haven't seen before. It is actually a variant of the `if` instruction that we're already familiar with, and the definition could as well have been written using the latter form:

```
(define ((cone center radius) x)
  (if (<= x (- center radius))
    0
    (if (<= x center)
      (/ (+ x (- radius center)) radius)
      (if (< x (+ center radius))
        (/ (+ (- x) center radius) radius)
        0))))
```

It is apparent that the form using `if` has a more complex structure because of the higher nesting level.

As all the rules and words' meanings are already known, the only thing that's left is to actually conduct the inference: given a tuple of labeled parameters describing a `patient`, say, `((bmi 29) (glycated-hemoglobin 5) (blood-pressure 20))`, we wish to provide a desired rating for it. In other words, want to `(infer #;from underwriter-rules #;about patient #;within health-categories)`.

First, we need to say to what `extent` does a `patient` belong to a given classes: for example, according to the membership functions defined above, the BMI parameter of 29 may be considered *overweight* with the degree 0.88, *obese* with the degree 0.13, *healthy* with the degree 0.05 and *underweight* with a degree very close to 0.

So it would be convenient to have a function that, for a given labeled tuple, returns the extent in which the tuple's values belong to certain classes defined within specified categories. It could also be handy if the classes were sorted, so that the classes to which a given values "belongs more" would appear earlier on the list.

```
(define (extent #;in-which entity #;belongs-to categories)
  (map (lambda ((property value))
         (let* ((category (lookup property categories))
                (classes (map (lambda ((name extent))
                                `(,name ,(extent value)))
                              category)))
           `(,property . ,(sort classes
                                (lambda ((_ degree-1) (_ degree-2))
                                  (> degree-1 degree-2))))))
       entity))

(e.g.
 (extent '((bmi 29) (glycated-hemoglobin 5) (blood-pressure 20))
         health-categories)
 ===> ((bmi (overweight 0.8824969025845955)
            (obese 0.1353352832366127)
            (healthy 0.053926197030622854)
            (underweight 3.879522352578383e-10))
       (glycated-hemoglobin (normal 0.95) (low 0.8) (high 0.6))
       (blood-pressure (overpressure 1.0)
                       (slight-overpressure 3.3546262790251185e-4)
                       (high-overpressure 3.3546262790251185e-4)
                       (normal 1.2664165549094176e-14))))
```

Probably the most surprising is the use of the `e.g.` form. This is a syntactic extension that serves as a lightweight *unit test* framework: it

evaluates the function before the `===>` sign and raises an error condition it its value isn't `equal?` to the value on the right. It therefore enhances the reliability of the system, because we have a confirmation that our function works as expected.

More importantly, it enriches the code with the information that is more concrete and easier to digest than the sole definition, because it shows exactly, which outputs can be expected for given inputs.

The example could not be comprehended in the absence of the definition of `health-categories`, but it shows us what kind of output from the function we should expect, which in turn prompts us with the idea of how this output should be processed further.

We ought to take another look at our `underwriter-rules`. They all take the form

`‘(if ,condition (is ,property ,classification))`

where `condition` either has a form

`‘(is ,property ,classification)`

or is a junction (i.e. conjunction, disjunction or negation) of simpler formulas. In the first case, say, `(is bmi overweight)` we evaluate it to the extent in which the BMI belongs to the set “overweight”. In the particular case of BMI being equal 29, it will be the value of approx. 0.82.

In the case of the junction of formulas, we evaluate it according to the rules given above: conjunction is the minimum function, disjunction – the maximum function, and negation – the $1 - x$ function. Note the resemblance between the below `satisfaction-degree` function and the definition `satisfied?` from the earlier chapters.

```
(define (satisfaction-degree formula interpretation)
  (match formula
    (('and . clauses)
     (minimum (map (lambda (clause)
                     (satisfaction-degree clause interpretation))
                   clauses)))
    (('or . clauses)
     (maximum (map (lambda (clause)
                     (satisfaction-degree clause interpretation))
                   clauses)))
    (('not clause)
     (- 1.0 (satisfaction-degree clause interpretation)))
    (('is property classification)
     (let* ((classifications (lookup property interpretation))
            ((extent) (lookup classification classifications)))
       extent))))
```

```
;; where
(define (maximum list)
  (match list
    ((last)
     last)
    ((first . rest)
     (max first (maximum rest)))))

(define (minimum list)
  (match list
    ((last)
     last)
    ((first . rest)
     (min first (minimum rest)))))
```

We defined two auxiliary functions that designate the `minimum` and `maximum` of the list, using the built-in `min` and `max` functions that return the largest of its arguments. As we will see in the following chapters, that definition wasn't strictly necessary.

Now that we can determine the `satisfaction-degree` of a given predicative formula, we have a set of rules and the preconditions of those rules, the only thing that is left is to conduct the inference.

This is actually the trickiest part, as it is concerned with a notion (or rather pseudo-notion) of *defuzzification*. Let me remind that the `underwriter--rules` consist of three rules, corresponding to three possible classifications of `rating`: `standard`, `perfect` and `refuse`. For example, the first rule said:

```
(if (or (is bmi underweight) (is bmi obese)
        (is glycated-hemoglobin low))
    (is rating refuse))
```

We need to blend those three rules, based on the extents of their preconditions, in order to choose the "right" classification.

One idea would be to choose the rule whose precondition is the highest `satisfaction-degree`. However, according to Wikipedia, "this approach loses information". Therefore another method is used: first we clip the membership functions for the various classes of the category `rating`, so that their values are limited to the the satisfaction-degrees, then we create a new function that gives the maximum of each of the clipped functions, and finally we apply some numerical method like calculation of the *center of gravity* of the function (Wikipedia lists 20 other defuzzification functions, each of them being equally arbitrary, none of them having any apparent advantages over the another).

```
(define (infer #;from rules #;about entity #;within categories)
  (let* ((judgment (extent entity categories))
         (conclusions
           (map (lambda (('if condition ('is property classification)))
                  (let* ((degree (satisfaction-degree condition judgment))
                         (categories (lookup property categories))
                         ((membership) (lookup classification categories))
                         (conclusion (clip membership 0 degree)))
                    `(,property ,conclusion)))
                rules))
         (common-subject (equivalence-classes
                          (lambda ((property-1 _) (property-2 _))
                            (eq? property-1 property-2))
                          conclusions)))
    (map (lambda (((properties functions) ...))
           (let* (((property . _) properties)
                  (composition (lambda (x)
                                 (maximum (map (lambda (f)
                                                 (f x))
                                               functions)))))
             `(,property ,composition)))
         common-subject)))
```

The code above is somewhat terse, and requires a bit of explanation. First we calculate the `judgment` using the `condition` function. The `judgment` therefore contains a list of the form `'((bmi (overweight 0.9) (normal 0.1) ...) (glycated-hemoglobin (normal 0.9) ...) ...)`. Then for each rule we decompose it into `condition` and the consequent of the form `('is property classification)`, so for example, the name `condition` can be bound to the sequence `'(or (is bmi underweight) (is bmi obese) (is glycated-hemoglobin low))`, `property` is bound to `'rating`, and `classification` is bound to `'refuse`. Then we take the `membership` function for the given `property` and `classification` pair. In the case of `rating`, `refuse` it is the `(cone 10.0 5.0)` function. Eventually we return a pair with the name of the classification and the clipped membership function.

Note that although it isn't the case with our example, we could in general have more properties than just `rating` contained in the consequents of our rules. Therefor we would need to split the rules that refer to the same property. This is what the `equivalence-classes` concept is for. In general, it takes a set and the so-called *equivalence relation*[1] and returns a list of lists

[1] Intuitively, an equivalence relation expresses some sort of common property, like "X is of the same religion as Y". This particular relation divides humans into equivalence classes such as Buddhists, Christians, Jews, Muslims, and so on, which leads to many

whose elements all belong to the same equivalence class.

For example, if we had a list of natural numbers, say, `(1 2 3 4 5 6 7 8 9)`, then its equivalence classes for the equivalence relation "x and y have the same remainder of the division by 3" would be a list of three lists: `((1 4 7) (2 5 8) (3 6 9))`.

Although the notion of `equivalence-classes` isn't essential for this particular task, it will be used later, so I will present the code here without any comments, except the note that it preserves the original order among the elements of the classes, and the observation that it uses the so-called *named-let* construct that won't be covered in this pamphlet. The explanation isn't difficult to find with Google.

```
(define (equivalence-classes equivalent? set)
  (let next-item ((set set)(result '()))
    (match set
      (()
       (reverse (map reverse result)))
      ((item . set)
       (match result
         (()
          (next-item set `((,item) . ,result)))
         ((this . next)
          (let next-class ((past '()) (present this) (future next))
            (match present
              ((paradigm . _)
               (if (equivalent? item paradigm)
                   (next-item set `((,item . ,present)
                                    . (,@past ,@future)))
                   (match future
                     (()
                      (next-item set `((,item) ,@result)))
                     ((this . next)
                      (next-class `(,present . ,past) this next)))
                    )))
             )))))))
```

Having the conclusions split into `equivalence-classes` of conclusions that regard the same `property`, we create a `composition` of the clipped functions of `membership` to given `classification`s, which are constructed by selecting the `maximum` value of each of the component functions.

The only thing that may be unknown at this point is what it means to `clip` a function. This is rather straightforward:

---

pointless wars.

```
(define ((clip function bottom top) x)
  (assert (<= bottom top))
  (max bottom (min top (function x))))
```

The `infer` function returns a list of form `((property function) ...)`. In order to get the classifications of the `property` values, we need to defuzzify each `function`.

We are going to use the aforementioned method of computing the `center-of-gravity`. This method is based on *numerical integration* that will not be covered here.

```
(with-default ((bottom 0)
               (top 100)
               (step 0.1))
  (define (center-of-gravity function)
    (let* ((step (specific step))
           (domain (range #;from (specific bottom)
                          #;to (specific top) #;by step)))
      (/ (sum (map (lambda (x) (* x (function x) step)) domain))
         (sum (map (lambda (x) (* (function x) step)) domain))))))
;; where
(define (range #;from bottom #;to top #;by step)
  (if (> bottom top)
    '()
    `(,bottom . ,(range #;from (+ bottom step) #;to top #;by step))))
```

The surprising thing is the use of the `with-default` derived form. It is used to give some default values to certain names, but without committing to those values. The form will be explained in one of the later chapters.

Finally, we can evaluate the expression

```
(let* ((conclusions (infer #;from underwriter-rules
                           #;about '((bmi 29)
                                     (glycated-hemoglobin 5)
                                     (blood-pressure 20))
                           #;within health-categories)))
  (map (lambda ((property function))
         `(,property ,(center-of-gravity function)))
       conclusions))
```

to find out that the `rating` received by our candidate is 7.46. In other words, we passed along some numbers to receive yet another number.

4

Matrix Operations

"Code is like poetry; most of it shouldn't have been written."
– David Jacobs

In the previous chapters we've been dealing with certain methods of *soft computing*. The numerical packages (or "programming languages", as some people call them) like R, SPSS or Matlab have often been praised for their "native" support for matrix operations and statistical notions.

In this chapter, we will show how easily these concepts can be implemented in Scheme using the means that we already know. We will also learn to define and use *variadic* functions.

It is common in mathematics and science to organize the computations in the form of *matrices*, because this allows to process large amounts of data in a systematic and uniform manner.

A *matrix* is a rectangular array of numbers, for example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 5 & 8 \\ 0 & 0 & 7 & 6 \end{pmatrix}$$

We can use lists of lists of equal length to represent matrices, for example

```
'((1 2 3 4)
  (0 4 5 8)
  (0 0 7 6))
```

4.0.1 Matrix addition

Given a matrix, we can determine its *dimensions*. For example, the above matrix is three-by-four.

```
(define (dim x)
  (match x
    ((first . rest)
     `(,(length x) . ,(dim first)))
    (_
     '())))
```

Probably the simplest algebraic operation that one can conduct on a matrix is *matrix addition*: given matrices of the same dimensions, we add each element of one matrix to the corresponding element of the second matrix:

```
(define (M+2 A B)
  (assert (equal? (dim A) (dim B)))
  (map (lambda (a b)
         (map + a b))
       A B))

(e.g.
 (M+2 '((1 2 3)
        (4 5 6)) '((1 2 3)
                   (4 5 6))) ===> ((2  4  6)
                                   (8 10 12)))
```

4.0.2 Variadic functions

I called the function `M+2` to emphasize that it takes two arguments. However, because matrix addition is associative (like conjunction and disjunction in propositional logic), so it might be more convenient to write `(M+ A B C)` instead of `(M+ (M+ A B) C)` or `(M+ A (M+ B C))`. Actually you can check that this is already the case for scalar addition: the expression `(+ 1 2 3)` evaluates to 6. This is also true for multiplication.

We can use the `M+2` function to perform the generalization to the arbitrary number of arguments, but no less than one.

```
(define (M+ . MM)
  (match MM
    ((M)
     M)
    ((A B . X)
     (apply M+ `(,(M+2 A B) . X)))))
```

We have used an *improper list* of arguments, or a *dotted tail notation*. An improper list is a list whose last `cdr` is not the empty list. If the list of arguments is improper, then the symbol in the dotted tail will be bound the list of remaining arguments, so for example if we had no `list` function defined, we could use that feature:

```
(define (list . items)
  items)

;; or equivalently:

(define list (lambda items items))
```

Note the use of the `apply` function. It takes a function and a list of arguments, and *applies* the function to arguments. For example, if `l` is a list containing three elements, then `(apply f l)` is equivalent to `(f (first l) (second l) (third l))`. However, if the number of arguments is unknown, then the use of `apply` is inevitable if we want to preserve generality.

Note also, that since the `max` and `min` functions can take the arbitrary number of arguments (and return the greatest and the smallest of them, respectively), instead of defining the `maximum` and `minimum` functions, we could have written `(apply max ...)` and `(apply min ...)`.

The `apply` function can also take a variable number of arguments – all arguments between the first and the last are simply `cons`ed to the last, so we could have written `(apply M+ (M+2 A B) X)` in the previous definition, yielding the same effect.

4.0.3 Transpose

If we swap columns and rows of a matrix, we obtain its *transpose*. For example, the transpose of the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

is the matrix

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

The code for constructing the transpose of a matrix is a bit tricky:

```
(define (transpose M)
  (apply map list M))
```

To see why it works, let's try to evaluate

```
(transpose '((1 2 3)
             (4 5 6)))
```

According to our definition, it can be rewritten as

```
(apply map list '((1 2 3) (4 5 6)))
```

which – according to what we said earlier – can be rewritten as

```
(map list '(1 2 3) '(4 5 6))
```

which yields the desired result.

4.0.4 Matrix multiplication

Multiplying two matrices is much trickier than adding them and is best explained by a math tutor pointing fingers on the blackboard.

If A and B are matrices, the product AB is defined only if the number of columns of matrix A equals the number of rows of the matrix B. Then, the number of rows of AB is the same as the number of rows of A, and the number of columns of AB is the same as the number of columns of B.

```
(define (M*2 A B)
  (assert (let* (((A-rows A-cols) (dim A))
                 ((B-rows B-cols) (dim B)))
            (= A-cols B-rows)))
  (let* ((B^T (transpose B)))
    (map (lambda (rA)
           (map (lambda (cB)
                  (sum (map * rA cB)))
                B^T))
         A)))
```

We can extend the `M*2` to support arbitrary number of arguments analogically as we did for `M+2`.

There are some other important operations that can be defined for matrices, like the determinant or rank. Their exposition is beyond the scope of this pamphlet, but some of them can be found in this pamphlet's repository.

5

Classifiers

"Truth is a mobile army of metaphors."

– Friedrich Nietzsche

5.1 Introduction

In this chapter, we will be dealing with the task of *classifying* data – given a tuple, we need to classify it to one of given categories, based on some set of existing classifications (called the *training set*). We do not know what the underlying rules of classification are – our system should infer that for us.

For the remainder of this chapter, we will be assuming that a database will be a list of lists, and that the first list of the database will be a header (a list of symbols) and all the remaining lists will contain data, whether it be *numerical* or *nominal*.

5.2 Naive Bayes and Probability

First of the classifiers that we are going to consider is based on the notion of *conditional probability* and makes an indirect use of the so-called *Bayes theorem*. The underlying idea is very simple: first we compute the *absolute* probability that the record belongs to a given class (based on the training set), then for each possible class we compute the conditional probabilities of an item of this class having the specified values as the certain fields of the tuple, multiply it altogether and choose whichever product of probabilities turns out to be the greatest. For example, if our data set has the form

```
(define computer-purchase
  '((age    income student credit-rating buys)
    (31..40 high   no      fair          yes)
    (>40    medium no      fair          yes)
```

```
  (>40    high   yes     excellent      yes)
  (>40    low    yes     excellent       no)
  (31..40 low    no      excellent      yes)
  (<=30   medium no      fair            no)
  (<=30   low    yes     fair            no)
))
```

and we are to predict the class of a new record `(>40 low no fair ?)`, then we first compute the probabilities $P(\texttt{buys} = \texttt{yes})$ and $P(\texttt{buys} = \texttt{no})$, then we compute the conditional probabilities $P(\texttt{age} = > 40|\texttt{buys} = \texttt{yes})$ and $P(\texttt{age} = > 40|\texttt{buys} = \texttt{no})$, the conditional probabilities $P(\texttt{income} = \texttt{low}|\texttt{buys} = \texttt{yes})$ and $P(\texttt{income} = \texttt{low}|\texttt{buys} = \texttt{no})$, and so on (for all the other data). For example, for this particular database, $P(\texttt{buys} = \texttt{yes})$ is $\frac{4}{7}$, because there are 7 entries, and 4 of them have the `buys` field equal to `yes`. Similarly, $P(\texttt{age} = > 40|\texttt{buys} = \texttt{yes})$ equals $\frac{2}{4}$, because there are 4 entries that have the `buys` field equal to `yes`, and 2 of them have the `age` field equal to `>40`.

Note that, unlike in the case of fuzzy logic, the values of probabilities have a well-defined meaning, so we shouldn't be disgusted with this approach, at least not in principle.

Having computed the conditional probabilities for both `buys=yes` and `buys=no`, we multiply them by the corresponding probabilities of belonging to given `buys` classes and choose the class whose probability is higher.

5.2.1 What is the universe

We have used the notation that is commonly used in the mathematics: $P(\phi)$ is a probability that the proposition ϕ is satisfied, and $P(\phi|\psi_1, ..., psi_n)$ is a probability that the proposition ϕ is satisfied given that $\psi_1, ..., psi_n$ are satisfied.

The exact meaning of the proposition is usually relative to some context or situation (in this case – our database). In the remainder of this pamphlet, we will be referring to this situation as `universe`.

In our case, the structure of a `universe` is rather simple and can be divided into two realms: the realm of *ideas* and the realm of *entities*. The realm of ideas names the possible properties of each single entity, and each entity consists of values whose meaning is specified in the realm of ideas. The variable `computer-purchase` is an example of a universe: the first element of the list is the realm of ideas, and contains the names of 5 properties: `(age income student credit-rating buys)`. The rest of the lists contains 7 individual entities, each of them obviously having 5 properties.

We may wish to refer to the property of an entity from the specific universe, for example *the age of a person*. For this purpose, we will define the function `the`:

```
(with-default ((universe '(()())))
  (define (the property #;of item)
    (let* (((names . things) (specific universe))
           (index (list-index (lambda (label)
                                (eq? label property))
                              names)))
      (if (not index)
          (throw 'not-found property names))
      (list-ref item index))))
```

We have used the `with-default` derived form that we have also used in the definition of the `center-of-gravity` function that was used in the defuzzification process. The default universe consists of an empty realm of ideas and of a single entity with no properties.

Unlike the `center-of-gravity` function, the default context of the `the` function isn't particularly helpful, so we will likely need to override the desired value somehow. For this purpose, we will use another derived form called `specify`.

For example, the expression

```
(specify ((universe '((name age))))
  (the 'age #;of '(George 48)))
```

evaluates to `48`.

If we wanted to express the naive Bayes classifier, we'd need to use the notion of `probability`. The proposition in the form $P(\phi)$ can be expressed using regular functions (i.e. lambdas) ranging the objects in the universe, for example $P(\texttt{buys} = \texttt{yes})$ could be paraphrased as

```
(probability
  (lambda (item)
    (eq? (the 'buys item) 'yes)))
```

Likewise, we could define `probability` as a variadic function, treating all the additional arguments as the *given* propositions:

```
(with-default ((universe '(()())))
  (define (probability proposition #;given . circumstances)
    (let* (((names . things) (specific universe))
           (known-world (filter (apply compose circumstances)
                                things)))
      (/ (count proposition known-world)
         (length known-world)))))
```

where `compose` is a function that takes arbitrary number of functions and returns their *composition* (if there are no arguments, the *identity* function is returned), i.e. `((compose f g) x)` means the same as `(f (g x))`.

Back to the Naive Bayes classifier, we can stick to the convention that the unknown property (whose value is supposed to be used) will be represented using the question mark symbol. Therefore we expect that the

```
(specify ((universe computer-purchase))
  (naive-bayes '(>40 low no fair ?)))
```

will try to classify the last entry in the tuple, i.e. the `buys` property. Therefore we may need to be able to retrieve the name of the unknown property, as well as the names of the known properties (note that we assume that there's only one unknown per tuple):

```
(without-default ((universe '(()()))
                  (unknown? (lambda (x) (eq? x '?))))
  (define (unknown-label+rest tuple)
    (let* (((header . data) (specific universe))
           (unknown/index (list-index (specific unknown?) tuple))
           (unknown-property (list-ref header unknown/index))
           (known-properties (filter (lambda (label)
                                       (not (eq? label unknown-property)))
                                     header)))
      (values unknown-property known-properties))))
```

The novel thing is the use of the function `values`. One of the most controversial features of Scheme is the functions' ability to have more than one value. It is useful on some occasions though, because it allows to extend a function to give additional information without breaking any existing code. It may be considered inelegant though, and perhaps it would be better to define two functions rather than one.

In addition to retrieving the name of the unknown, it would also be helpful if we were able to get all the possible values of a given property within a universe, in order to compute the appropriate conditional and unconditional probabilities:

```
(with-default ((universe '(()())))
  (define (column name)
    (let* (((header . data) (specific universe))
           (index (list-index (lambda (x) (eq? x name)) header)))
      (map (lambda (row)
             (list-ref row index))
           data)))

  (define (possible-values attribute)
    (delete-duplicates (column attribute))))
```

```
(e.g.
 (specify ((universe computer-purchase))
   (possible-values 'age)) ===> (<=30 31..40 >40))
```

Note that we also defined a `column` auxiliary function that allows us to project the universe onto one of its dimensions. That function will also be used later.

We can now define the Naive Bayes of a record in the following way:

```
(with-default ((universe '(()())))
  (define (naive-bayes tuple)
    (let* ((unknown-property known-properties
                             (unknown-label+rest tuple))
           (classes (possible-values unknown-property)))

      (define (unconditional-probability class)
        (probability (lambda (x)
                       (eq? (the unknown-property x) class))))

      (define (partial-conditional-probabilities #;for class)
        (map (lambda (property)
               (probability
                 (lambda (item)
                   (eq? (the property item) (the property tuple)))
                 #;given
                 (lambda (item)
                   (eq? (the unknown-property item) class))))
             known-properties))

      (apply argmax (lambda (class)
                      (apply * (unconditional-probability class)
                               (partial-conditional-probabilities
                                class)))
             classes))))
```

In the definition, we introduced two auxiliary definitions of `unconditional--probability` of belonging to a `class` and of `partial-conditional-probabilities` of having certain properties given that we belong to given `class`.

We have also used the `argmax` library function that takes a measure function and arbitrary number of arguments and returns the argument whose measure is the greatest.

The code above expresses the idea of the naive Bayesian classifier in a manner that is probably a lot more precise and concise than the textual description given at the beginning of this section. It may take some time

to be able to both read and write such code, but once it is mastered, it becomes as natural as listening and talking, allowing to express more and more advanced concepts.

5.3 Decision Trees and Information

The naive Bayes classifier is structurally very simple. In this section we are going to learn about a slightly more sophisticated classifier known as the *decision tree*. It is interesting, because in addition to being a classifier, it can also be seen as an algorithm for compressing knowledge contained in a data set.

In general, a decision tree is an acyclic directed graph whose nodes alternately represent questions and possible answers to those questions. Usually there are many ways in which a decision tree can be constructed for a given data set, but some of those graphs are more compact than others (i.e. contain less nodes). In the worst case the number of leaves in a tree can be equal to the number of database entries. This can depend on both the structure of data and on the order in which we decide to ask questions, so it is important to ask them in a way that allows to reject as many alternative options as possible.

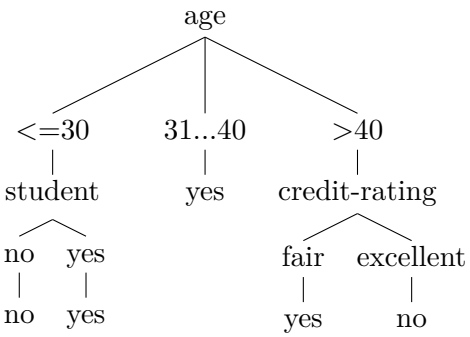
For example, let's take a look at this slightly larger version of the `computer-purchase` database:

```
(define computer-purchase
  '((age     income student credit-rating buys)
    (<=30    high   no      fair           no)
    (<=30    high   no      excellent      no)
    (31...40 high   no      fair          yes)
    (>40     medium no      fair          yes)
    (>40     low    yes     fair          yes)
    (>40     low    yes     excellent      no)
    (31...40 low    yes     excellent     yes)
    (<=30    medium no      fair           no)
    (<=30    low    yes     fair          yes)
    (>40     medium yes     fair          yes)
    (<=30    medium yes     excellent     yes)
    (31...40 medium no      excellent     yes)
    (31...40 high   yes     fair          yes)
    (>40     medium no      excellent      no)
  ))
```

We can notice, that if a person's `age` is `<=30`, then that person buys a computer only if she or he is a student – thus we can get the answer after two questions. People whose `age` is `31...40` always buy a computer. If a person's

`age` is `>40`, then he or she buys a computer only if the `credit-rating` is `fair`.

The above *decision process* can be represented in the form of a tree:



5.3.1 Representing trees in Scheme

One could ask how can we represent such a tree in Scheme. The truth is however that we already have, by using nested lists. The most straightforward way of representing a binary tree with labeled nodes is to use a list of the form `(label branches ...)`, where `branches` are also trees. The only exception is that we represent leaves as atomic objects. So for example, the above tree could be written as the following Scheme expression:

```
(age (<=30 (student (no no)
                    (yes yes)))
     (31...40 yes)
     (>40 (credit-rating (fair yes)
                         (excellent no))))
```

It may not be immediately apparent that those two objects are *isomorphic*, and besides the former is much more pleasant to watch, so it would be helpful if we could convert Scheme expressions to pictures in a systematic way. This shouldn't be particularly difficult – the above tree was generated in LaTeX using the *qtree* package from the following code:

```
\Tree [.age
  [.<=30
    [.student
      [.no no ]
      [.yes yes ]  ]  ]
  [.31...40 yes ]
  [.>40
    [.credit-rating
      [.fair yes ]
      [.excellent no ]  ]  ]  ]
```

(Note the resemblance). Before we create our converter, however, we should note one thing: although for simple examples such as the one above we could decide, for a given path on the tree, which label should be assigned to a given data tuple, our database could contain a few tuples with the same values but different labels, like:

```
...
(<=30 high no fair no)
...
(<=30 high no fair yes)
...
```

In such situations, we would rather that the leaf contained the *probability distribution* of belonging to certain classes, than a single value. How do we do that, if we already used pairs to represent trees?

One way would be to change our representation of a tree, so that the nodes, branches or leaves were tagged somehow. However that would add complexity (and perhaps ambiguity) to our representation.

Another way would be to *freeze* compound data structures so that they could be seen as atomic objects in the context of a tree, but could also be *defrost* for further processing.

Scheme does provide some means for freezing the compound objects: in addition to lists or pairs, one can also use containers called *vectors*, and convert vectors to lists and lists to vectors using `vector->list` and `list->vector` built-in procedures.

We can therefore assume that a leaf of a tree is either a symbol or a vector, and in the latter case we would like to generate a table containing the information from that vector. The probability table will have the form

```
#((class-1 probability-1)
  (class-2 probability-2)
  ...)
```

where `#(elements ...)` is a legitimate Scheme notation for vectors.

The code for converting Scheme objects to LaTeX trees and tabulars looks as follows:

```
(define (latex/tabular rows)
  (let* ((columns (length (first rows))))
    (string-append
     "\\begin{tabular}{"
     (string-join (make-list columns "c") "")
     "} "
     (string-join
      (map (lambda (row)
             (string-join (map ->string row) " "))
           rows)
      " \\\\ \\hline ")
     " \\end{tabular}")))

(define (latex/qbranch tree)
  (match tree
    ((node . branches)
     (string-append
      "[." (->string node) " "
      (string-join (map latex/qbranch branches) " ")
      " ] "))
    ((? vector?)
     (string-append
      "\n[.{"
      (latex/tabular (vector->list tree))
      "} ] "))
    (leaf
     (->string leaf))))

(define (latex/qtree tree)
 (string-append
  "\\begin{figure}"
  "\\Tree "
  (latex/qbranch tree)
  "\\end{figure}"))
```

The `latex/qtree` interface is a simple wrapper on the `latex/qbranch` function that "does all the work", i.e. that converts a tree into a string containing LaTeX code for displaying a tree. The `latex/tabular` function does nothing beyond the obvious.

The defined functions make an extensive use of built-in functions `string--append` and `string-join` as well as a library function `->string` that converts arbitrary Scheme objects to strings.

Note that we had to escape the slash symbols for LaTeX.

5.3.2 Constructing a decision tree

Now that we are equipped with proper tools for visualizing the tree structures, we can try to construct a tree for a given data set. In order to do so, let's recall the informal construction of the tree that we conducted at the beginning of this section. The first question that we chose to divide our set was about the `age`, because it seemed intuitive that this parameter was most *informative* with regard to the question whether a given person `buys` a computer or not.

Regardless of the underlying intuitions, we can show how to construct a decision tree, provided that we know how to specify what is an `information-gain` of an `attribute` with regard to a certain `label` (which, in the above case, is the `buys` attribute).

The construction for a given database will proceed by splitting the database rows into equivalence classes according to the values of the considered attributes, and removing the columns containing the attributes that have already been considered. The smaller databases created in this way will be called `slices`:

```
(define (drop-column database label)
  (let* (((header . data) database)
         (index (list-index (lambda (name) (eq? name label)) header)))
    (map (lambda (row)
           (skip index row))
         database)))

(define (slices #;of database #;on attribute)
  (specify ((universe database))
    (let* (((names . things) database)
           (classes (possible-values attribute)))
      (values
       (map (lambda (class)
              (let* ((peers (filter (lambda (item)
                                     (eq? (the attribute item)
                                          class))
                                   things)))
                (drop-column `(,names . ,peers) attribute)))
            classes)
       classes))))
```

The `(skip i l)` is a library function that returns a copy of list `l` without its `i`-th element.

The construction stops (for a given slice) either when every record of that slice has the same value of the `label` attribute (e.g. `buys`=`yes`), or when the only column left in the database is the `label` – in the former case we

know what the node value should be, but in the latter we need to construct the probability distribution table:

```
(define (probability-table ((label) . column))
  (specify ((universe `((,label) . ,column)))
    (let* ((classes (possible-values label)))
      (map (lambda (class)
             `(,class ,(probability (lambda (item)
                                      (eq? (the label item)
                                           class)))))
           classes))))

(define (decision-tree #;of data #;for label)
  (match data
    (((last) . _)
     (assert (eq? label last))
     (list->vector (probability-table data)))
    ((names . things)
     (let* ((attributes (filter (lambda (name)
                                  (not (eq? name label)))
                                names))
            (attribute (apply argmax
                              (lambda (attribute)
                                (specify ((universe data))
                                  (information-gain attribute label)))
                              attributes))
            (sections classes (slices #;of data #;on attribute))
            (branches (map (lambda (class section)
                             (specify ((universe section))
                               (match (possible-values label)
                                 ((result)
                                  `(,class ,result))
                                 (_
                                  `(,class ,(decision-tree section
                                                           label))))))
                           classes sections)))
       `(,attribute . ,branches)))))
```

What's left before our code can actually work is to define the `information-gain` somehow. There are many ways in which it can be defined. I will present the definition based on Claude Shannon's notion of *information entropy*, although I need to note that I do not comprehend it fully.

We start off by defining the measure of information contribution of given proposition:

```
(define (information proposition #;given . circumstances)
  (let ((p (apply probability proposition circumstances)))
    (if (zero? p)
        0.0
        (- (* p (log p))))))
```

As we can see, the notion of *information* is defined in terms of `probability`, so it needs a specific `universe` to operate on. It also allows to specify "conditional information", which corresponds exactly to the conditional probability.

We had to check whether the probability of given event is zero, because if it were, then the logarithm of that probability would be $-\infty$, and in the Scheme arithmetic zero times ∞ is not a number.

The notion of `information` is used in the definition of `information-gain`:

```
(define (information-gain attribute target-class)
  (let* ((classes (possible-values target-class)))
    (- (sum (map (lambda (class)
                   (information (lambda (item)
                                  (eq? (the target-class item)
                                       class))))
                 classes))
       (attribute-entropy attribute target-class))))

;; where
(define (attribute-entropy attribute target-class)
  (let* ((classes (possible-values target-class))
         (options (possible-values attribute)))
    (sum (map (lambda (option)
                (sum (map (lambda (class)
                            (* (information
                                (lambda (item)
                                  (eq? (the target-class item)
                                       class))
                                #;given
                                (lambda (item)
                                  (eq? (the attribute item)
                                     option)))
                               (probability
                                (lambda (item)
                                  (eq? (the attribute item)
                                       option)))))
                          classes)))
              options))))
```

The definitions are rather simple structurally, and the only problem with their comprehension may stem from the lack of understanding of the underlying theory.

5.3.3 Deciding upon a decision tree

Now that we know how to build trees, we may wish to use them as classifiers: we shall follow the tree according to the path specified by the properties of the data that we wish to classify. If the leaf that we reached is a single value, we simply assign it to our tuple. Otherwise it is a probability distribution table, and we should draw lots according to that distribution:

```
(define (draw-lots probability-table)
  (let* ((((classes probabilities) ...) probability-table)
         (cumulative-distribution (scan + 0 probabilities))
         (lot (random 1.0))
         ((class upper-limit) (find (lambda ((class upper-limit))
                                      (<= lot upper-limit))
                                    (zip classes
                                         cumulative-distribution))))
    class))
```

we used here the `scan` library function for calculating *prefix sum*. Assuming that `(random 1.0)` returns an evenly distributed random variable, this method is known as *inverse transform sampling*.

The code for actually making a decision based on the tree isn't particularly complicated.

```
(with-default ((universe '(()())))

  (define (decide/tree tuple)
    (let* ((unknown-label (unknown-label+rest tuple))
           (tree (decision-tree (specific universe) unknown-label)))
      (tree-decide tree tuple)))

   (define (tree-decide tree tuple)
     (match tree
       ((property . selections)
        (let* ((class (the property tuple))
               ((_ next) (find (lambda ((value next))
                                 (eq? value class))
                               selections)))
          (tree-decide next tuple)))
       ((? vector? probability-table)
        (draw-lots probability-table))
       (_
        tree))))
```

The `decide/tree` procedure constructs a tree for the unknown label, and then refers to the `tree-decide` function that actually traverses the path on the tree by recursively calling itself. The code follows the established convention of operating within the context of `universe`.

5.4 k Nearest Neighbours

So far we have been dealing with a *nominal* database, i.e. such that its entries belonged to a finite set of unordered values. In this section, we are going to learn about the *k Nearest Neighbours* classifier that is intended to handle the *numerical* data, i.e. data consisting of real numbers.

For this purpose, we will use the database acquired from *The Data-mining Project* called "iris.csv", which – as its name suggests – is stored in a CSV file.

Although the format is rather simple (it is a simple line-based text file), there is a readily available library for Guile (surprisingly called "Guile-CSV") that we will use for reading the data. The patched version of the library (with a simplified interface) is available in the pamphlet's repository, as well as the database itself.

In order to load the "iris.csv" database, having loaded the `(guile csv)` module, we shall write

```
(define iris (read-csv "iris.csv"))
```

The database is compatible with previously described structure of the `universe`. All columns but the last contain some numerical data (dimen-

sions of certain parts of flowers), while the last column, labeled as `class`, contains the name of the flower species: `Iris-setosa`, `Iris-versicolor` or `Iris-virginica`.

The numerical parameters of database entries can be seen as points in a multi-dimensional space. The idea behind the *k Nearest Neighbours* classifier is simple: given a point that we wish to classify, we take a look at the k nearest existing points, and classify the new point to the class to which the majority of those nearest points belongs.

We will be assuming, that in the given data set the classification is always the last field. While this may not be the most fortunate idea in general, it turned out to be sufficiently good in practice (it is always OK to generalize the code when needed).

In order to build the desired classifier, we actually only need only three functions: one that would allow us to compute the *distance* between two points, another for finding the nearest neighbours and one for taking the majority of them.

With regard to the distance, we will be using the `euclid-distance` function:

```
(define (euclid-distance a b)
  (sqrt (sum (map square (map - a b)))))
```

Note that – in addition to the built-in `sqrt` function for computing the square root of a number, we have used two functions defined in the first chapter. How surprising the life can be!

Also note, that the function makes no assumptions on the dimensionality of the space that we investigate.

In the case of the `nearest-neighbours` procedure, we will simply sort the data according to the distance between its points and the point being classified, and then we take the k first points from the list. For the practical reasons, we will compute the distance and attach that information to database before sorting – that way we will avoid computing the value of the distance function in each comparison.

```
(define ((nearest-neighbours k #;in data) #;of sample)
  (let* ((measured (map (lambda ((values ... label))
                          `(,(euclid-distance values sample)
                            ,@values ,label))
                        data))
         (sorted+ (sort measured
                        (lambda ((distance-1 . _) (distance-2 . _))
                          (< distance-1 distance-2))))
         (sorted (map (lambda ((distance . data)) data) sorted+)))
    (take sorted k)))
```

Lastly, we need to choose the label such that the majority of the points bear it. We will use the `equivalence-classes` operation to facilitate that:

```
(define (dominant-label labeled-points)
  (let* ((classes (equivalence-classes
                   (lambda ((_ ... label-1) (_ ... label-2))
                     (eq? label-1 label-2))
                   labeled-points))
         (biggest (apply argmax length classes))
         (((_ ... label) . _) biggest))
    label))
```

The only thing that's left is to bring those ideas together:

```
(define (((classify/knn k) universe) sample)
  (let* (((header . data) universe))
    (dominant-label ((nearest-neighbours k data) sample))))
```

5.5 Quantization of numerical data

Suppose that we want to classify numerical data using our decision trees or naive Bayes classifier. Applying them to numerical data could be rather inefficient, because we would likely obtain very low probabilities in the naive Bayes classifier, and very complex and inaccurate decision trees (and most likely no path on the decision tree would correspond to our data).

A possible solution would be to split the numerical data into intervals, and construct trees or compute probabilities for those intervals, rather than exact numbers. This method is called *quantization*. In particular, if all the intervals (except, perhaps, the first and the last one) are of the same length, then we deal with *uniform quantization*[1].

This section presents briefly how the numerical data can be quantized.

The procedure is rather straightforward – for each dimension, we need to take the extreme values (the lowest and the highest). They will span over the range that will be used for quantization. Additional categories will be created for points lying beyond those extremes.

In order to classify a point, we search all the intervals and return the one the given point belongs to.

We need to classify each dimension separately, so we shall create a list of *quantizers* (i.e. functions that assign a point the name of its range).

This is how this works in practice:

[1]Perhaps a more fruitful idea would be to quantize the data so that the number of elements that belong to various intervals is equal, rather than the length of the intervals.

```
(with-default ((intervals 5)
               (universe '(()())))
  (define ((quantizer label) value)
    (let* ((data (column label))
           (low high (apply min+max data))
           (unit (/ (- high low) (specific intervals)))
           (range (iota (specific intervals) low unit))
           (upper-bound (find (lambda (upper-bound)
                                (< value upper-bound))
                              range)))
      (if upper-bound
          (symbol-append '< (number->symbol upper-bound))
          (symbol-append '>= (number->symbol high)))))

  (define (quantize-item)
    (let* ((((header ... class) . _) (specific universe))
           (quantizers (map quantizer header)))
      (lambda ((values ... label))
        `(,@(map (lambda (quantize value)
                   (quantize value))
                 quantizers values) ,label)))))

(define (quantize universe)
  (let (((header . data) universe))
    (specify ((universe universe))
      `(,header . ,(map (quantize-item) data)))))
```

For creating the names of the labels, we have used library functions `symbol-append` and `number->symbol`, whose names are rather self-explanatory. As you can see, we only use the upper-bounds to name the intervals, so for example if we want to classify number 21.5 to one of intervals $(-\infty, 18.0]$, $(20.0, 22.0]$, $(22.0, 24.0]$, $(24.0, +\infty)$, it will be labeled as `<22.0`.

Of course, we need to create a quantized version of a universe in order to further operate on it, for example:

```
(define iris/quantized (quantize iris))
```

5.6 Evaluating classifiers

Although we have presented a few methods for classifying data, it would be interesting to ask whether they actually work, or to what extent we can rely on them. This section will present a small framework for evaluating the classifiers.

The idea is to divide our data set into two parts: the *training* part (that would be the base for the construction of decision trees, calculating Bayesian probabilities, or determining the set of neighbours that we use for classification) and the *verification* part that would allow us to determine how many predictions were right and in how many cases our classifier was mistaken, and what sort of errors it made.

In order to do so, we shall write a procedure that takes a given strategy (understood as a function that takes the training set and produces a function that actually performs classification of samples), a training/verification set and a fraction that expresses how big should be the training set compared to verification set, and returns some detailed information regarding the confusions made.

```
(define (evaluate-strategy strategy training-proportion data)
  (assert (< 0 training-proportion 1))
  (let* (((header . data) data)
         (classes (equivalence-classes
                   (lambda ((_ ... label-1) (_ ... label-2))
                     (eq? label-1 label-2))
                   data))
         (((training . verify) ...)
          (map (lambda (class)
                 (let* ((total-size (length class))
                        (training-size (inexact->exact
                                        (floor (* training-proportion
                                                  total-size))))
                        (training verify (split-at class
                                                   training-size)))
                   `(,training . ,verify)))
               classes))
         (training verify (values `(,header . ,(concatenate training))
                                  (concatenate verify)))
         (classify (strategy training))
         (classified (map (lambda ((sample ... label))
                            `(,(classify sample) . ,label))
                          verify)))
    (values
     (confusion-matrix classified)
     (exact->inexact (/ (count (lambda ((guess . real))
                                 (not (eq? guess real)))
                               classified)
                        (length classified))))))
```

Despite its length, the code is rather simple. The only new thing here is the use of the library function `concatenate` that takes a list of lists and

returns a list of elements of those lists, for example `(concatenate '((1 2) '(3) '(4 5)))` would return a list `(1 2 3 4 5)`, and the use of the `inexact->exact` procedure, that requires a bit of explanation. The Scheme programming language has two distinct types of numbers: exact, which include integers and rationals, and inexact, that correspond to the notion of *floating point* numbers[2]. The exact numbers are called so because they can be arbitrarily large (with regard to integers) or arbitrarily small (with regard to fractions), while the inexact numbers are always represented with limited precision. The above use of the `inexact->exact` function stems from the fact that for some reason the `floor` function (which returns the largest integer part of its argument) returns an inexact number, while the `split-at` function takes an exact.

The most important thing that the `evaluate-strategy` function does is producing a pair of labels `(guess . real)` based on the verification set and gathering them in the `classified` list. Then the list is passed on to the `confusion-matrix` procedure that makes a summary of errors:

```
(define (confusion-matrix classified-data)
  (let* ((rows (equivalence-classes (lambda ((a . _)(b . _))
                                      (eq? a b))
                                    classified-data)))
    (map (lambda (row)
           (let* ((classes (equivalence-classes
                            (lambda ((_ . a) (_ . b))
                              (eq? a b))
                            row))
                  (total (length row)))
             (map (lambda (class)
                    (let* ((((labeled . guessed) . _) class)
                           (classified (length class)))
                      `(,labeled ,guessed ,classified ,total
                                 ,(exact->inexact
                                   (/ classified total)))))
                  classes)))
         rows)))
```

As we said earlier, the use of the evaluator requires us to stick to certain conventions – the strategy needs to be a function that takes data and returns a classifier for a single sample. The `classify/knn` function already conforms to that convention (once the `k` is settled), but the other two classifiers do not, and therefore we need to adopt them:

[2]Actually, the notion of inexact numbers also embraces *complex numbers* that are represented as pairs of floating point numbers.

```
(define ((classify/naive-bayes data) sample)
  (specify ((universe data))
    (naive-bayes `(,@sample ?))))

(define ((classify/decision-tree data) sample)
  (specify ((universe data))
    (let* ((tree (decision-tree data 'class)))
      (tree-decide tree sample))))
```

Note that the `classify/decision-tree` will misbehave under the substitutional model of computation, because it will require to reconstruct the decision tree for each sample. This can be solved easily using the technique of *memoization* that we will not cover here.

Finally, we can use our evaluator to verify the quality of our classifiers:

```
(evaluate-strategy classify/decision-tree 7/10 iris/quantized)
```

5.7 Clusterization

In the previous sections we've been dealing with a task of assigning known labels to the elements in a set. This section presents a method called *k means* that allows to establish distinct categories within an unlabeled set of numerical data. This technique is in general called *clusterization*.

Although it may sound fancy, the method of k means is rather simple, but also limited, because it requires its user to provide the maximum number of clusters that are to be found in the set.

Once it is done, we choose random points in a given space and for each of those points, we assign it the samples that are the nearest. Then we replace the random points with mean values of those samples, and repeat that process until we reach a *fixed point*, i.e. until the points remain the mean values of the attracted samples.

In order to pick the random values, we first need to specify the ranges among which we ought to choose, so that we can select a value from within that range. Then, we simply generate a list of points of the desired length:

```
(define (random-centers k data)
  (let* ((upper (apply map max data))
         (lower (apply map min data)))
    (generate-list k (lambda ()
                       (map (lambda (lower upper)
                              (+ lower (random (- upper lower))))
                            lower
                            upper)))))
```

Once you get used to using `map`, `apply` and `lambda`, the code is very straightforward and needs no comment. Before this happens, you probably need to practice a lot, though.

The assignment of samples to the alleged cluster centers is also very simple, once you get familiar with the notion of `equivalence-classes`:

```
(define (classify data centroids)
  (let* ((nearest-centroid (lambda (point)
                             (apply argmin
                                    (lambda  (c)
                                      (euclid-distance point c))
                                    centroids)))
         (clusters (equivalence-classes
                     (lambda (a b)
                       (equal? (nearest-centroid a)
                               (nearest-centroid b)))
                     data)))
    clusters))
```

When we put those two above together, we get the clusterization routine:

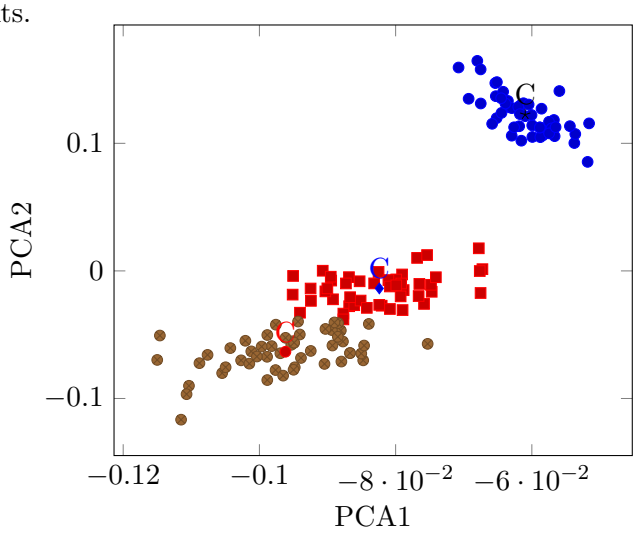
```
(define (clusterize data k)
  (let* ((centroids (random-centers k data))
         (clusters (classify data centroids))
         (new-centroids (map (lambda (cluster)
                               (apply map (lambda elements
                                            (mean elements)) cluster))
                             (filter pair? clusters))))
    (if (equal? centroids new-centroids)
        (values clusters new-centroids)
        (clusterize data new-centroids))))
```

Probably the most surprising part is the `(filter pair? clusters)` expression – its purpose is to filter out empty clusters, if they happen to appear, because for them, calculating the `mean` wouldn't make sense (so effectively the final number of clusters can be smaller than the initially assumed). The `mean` function is obviously defined as

```
(define (mean list)
  (/ (sum list) (length list)))
```

We can observe the result of clusterization on the following picture. Compared to the original labeling, only one label was classified improperly. The

clusterization was performed on the data preprocessed by Principal Component Analysis[3], and the plot shows two most significant principal components.



[3] Principal Component Analysis is a technique for reducing the number of parameters in a numerical data set while preserving the most significant information. The Scheme code for PCA that I used is based on Singular Value Decomposition routine extracted from the Scheme numerical package `scmutils`. More information about PCA can be found in [5].

What to do next?

> *"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want [...]"*
>
> – Donald Knuth

Barry Rowlingson, one of the contributors to the R project, wrote: "This is all documented in TFM. Those who WTFM don't want to have to WTFM again on the mailing list. RTFM".

My hope is that after reading this pamphlet at least some readers will see that there is another way – that instead of being divided into groups of "software package creators" and "software package users", we can all use participate in the joint movement of software literacy: that we will R&WTFSC and FTM.

There's plenty of ingenious ideas that have found their expression in the Scheme programming language, although they made it so without the "rock-star status", going against the flow.

I encourage the interested readers to take a look at the papers available at `http://readscheme.org`.

```
$ chmod -R 666 /
```

Bibliography

[1] Abelson, Harold and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs* MIT Press, 1996
https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-1.html

[2] Harrison, John, *Introduction to Functional Programming*
http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf

[3] Ihaka, Ross, *R: Lessons Learned, Directions for the Future*
https://www.stat.auckland.ac.nz/~ihaka/downloads/JSM-2010.pdf

[4] McCarthy, John *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*
http://www-formal.stanford.edu/jmc/recursive.ps

[5] Shlens, Jon *Tutorial on Principal Component Analysis* https://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition_jp.pdf

[6] Lotfi Zadeh, *Is there a need for fuzzy logic?* Information Sciences, Vol. 178, No. 13, 2751-2779, 2008
http://www.cs.berkeley.edu/~zadeh/papers/Isthereaneedforfuzzylogic.pdf