Universidad de Cantabria

Facultad de Ciencias

Implementación de un Intérprete de Programación Lógica en Scheme y Estrategias de Resolución de Cláusulas de Horn

Implementation of a Logic Programming Interpreter in Scheme and Horn Clause Resolution Strategies

Trabajo de fin de grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Fernando Frías García-Lago

Director: Domingo Gómez Pérez

Febrero de 2024

Implementación de un Intérprete de Programación Lógica en Scheme y Estrategias de Resolución de Cláusulas de Horn

Resumen

palabras clave: intérprete, programación, lógica, Scheme, cláusulas de Horn

En el presente Trabajo de Fin de Grado (TFG) se ha explorado la equivalencia entre los paradigmas de programación imperativa y funcional, usando como lenguaje conductor Scheme, y el paradigma de la programación lógica, tomando como referencia el lenguaje PROLOG.

Como objetivo final del mismo, se ha implementado un intérprete de programación lógica en Scheme, incluyendo la programación de algoritmos para trabajar con la resolución SLD de cláusulas de Horn, un método que ha sido ampliamente examinado.

Por lo tanto, la parte fundamental del trabajo desarrollado ha sido el estudio y la comprensión en profundidad de los conceptos y bases de la programación lógica, así como la resolución de una amplia variedad de problemas de programación funcional e imperativa para poder implementar el intérprete final.

Acompañando a esta memoria se adjunta un archivo con el código que genera el intérprete junto a una base de conocimiento que recoge una variedad de ejercicios de programación lógica resueltos mediante dicho intérprete. Además, se indica el enlace a un directorio de GitHub con toda la colección de problemas resueltos: https://github.com/ffriaslago/lambda.

Implementation of a Logic Programming Interpreter in Scheme and Horn Clause Resolution Strategies

Abstract

keywords: interpreter, programming, logic, Scheme, Horn clauses

In the present Bachelor's Thesis, it has been explored the equivalence between imperative and functional programming paradigms, using Scheme as the guiding language, and the logic programming paradigm, using as reference the PROLOG language.

The ultimate objective was the implementation in Scheme of a logic programming interpreter, including the development of algorithms to work with SLD resolution of Horn clauses. a method that has been examined in detail.

Therefore, the fundamental part of the work developed has been the in-depth study and understanding of logical programming concepts and foundations, as well as the resolution of a wide variety of functional and imperative programming problems in order to implement the final interpreter.

Accompanying this report, a file containing the code that generates the interpreter is attached along with a database that includes a variety of logic programming exercises solved using the mentioned interpreter. Additionally, the link to a GitHub directory with the entire collection of solved problems is provided: https://github.com/ffriaslago/lambda.

Contents

1 Introduction

Artificial Intelligence (AI) is in the epicentre of one of the most intense debates about technology. It is being questioned how it should be used and what it is behind it. Although this discussion is recent, AI is not a new concept, as its formal foundations can be considered in the work of Alan Turing during the 40s and 50s of the last century. However, over the last years, the fast deployment of AI has become essential for many applications in industry and entertainment, raising a question about the quality of it and how to audit the results.

There is a broad shared ground between AI and logical programming, being the latter known for its ability to efficiently handle logical inference and knowledge representation. Some common key concepts are their declarative nature and how both are designed for solving problems. In the next section, Logical Programming is reviewed, focusing on the advantages, components and theoretical background of logical programming.

1.1 Logical Programming

Logical programming is a programming paradigm that is characterised by being directed to solve problems from a declarative perspective instead of an imperative one. In other words, stating the objective that has to be achieved is the main focus in logical programming, rather than giving the computer precise instructions of how to approach a problem, typical from an imperative view, which involves a strong planning and requires a deep understanding of knowing how to transmit the solving method to the computer. Hence, in logical programming, the problem is expressed as a set of facts and rules, defining logical relationship. For this reason, it is said to be based on First-order logic, including all its helpful features.

its declarative nature is one of the key advantages of logical programming. This often means having smaller and easier to understand codes because they are better defined. Therefore, it improves their readability. From the user's point of view, it allows more flexibility, as one of the most important characteristics is dynamic typing, a concept it will be expounded later in the section.

Because of the main focus of logical programming being the description of the problem, it is necessary to define certain methods that will be used during the resolution of problem instances. The two most important ones are SLD resolution and backward chaining. Therefore, when doing the implementation of an interpreter for a logical programming language, a user has to be aware of how these methods work in order to describe meticulously the problem to solve. All this structure gives an advantage for the user: by implementing these pieces the focus of logical programming shifts to the expressive power and flexibility. Also, as it uses First-order logic, it can be easier to discern the reasoning behind some solutions. In summary, logical programming is a modern topic nowadays as it is one of the foundations of data analysis or, as it has already been mentioned, AI, where the comprehension of the solution given by this tool is crucial.

Going now into technical details for implementing First-order logic, its components can be classified in: statements, predicates, variables, quantifiers ($\forall$, $\exists$) and logic operators AND ($\wedge$), OR ($\vee$) and

NOT ($\neg$). The idea is to combine these elements to form sentences (complex statements), which can also be expressed as a fact (other used names for this concept are assertion or statement) and rules (or formulas). After establishing this set of facts and rules, other facts can be inferred from the defined ones in different ways. This can be achieved when giving an input to an interpreter, that through and automated reasoning returns an answer. These inputs can be referred as queries and, depending on the form and structure of the query, the kind of answers are very different, starting from just checking if the given statement is true or not. This is because of the mentioned flexibility, that it is highly present in logical programming.

Next, it will be used an elementary example to see how it can be represented the evaluation of an statement in First-order logic. Using the logical operators, it is built the formula given by expression 1. Besides, it is assembled the truth table that is defined by it, reconstructing all the sub-formulas involved in the final one. Finally, in Figure 1 it is represented a tree diagram that is related to the formula 1.

$$(A \vee B) \wedge (A \vee (A \wedge B)) \tag{1}$$

A	B	$A \vee B$	$A \wedge B$	$A \vee (A \wedge B)$	$(A \vee B) \wedge (A \vee (A \wedge B))$
1	1	1	1	1	1
1	0	1	0	1	1
0	1	1	0	0	0
0	0	0	0	0	0

When evaluating the formula in equation (1), certain paths of the tree diagram of Figure 1 are travelled through as it is indicated by the arrows. This type of diagrams are very useful when wanting to track the evaluation process of an interpreter for a given program.

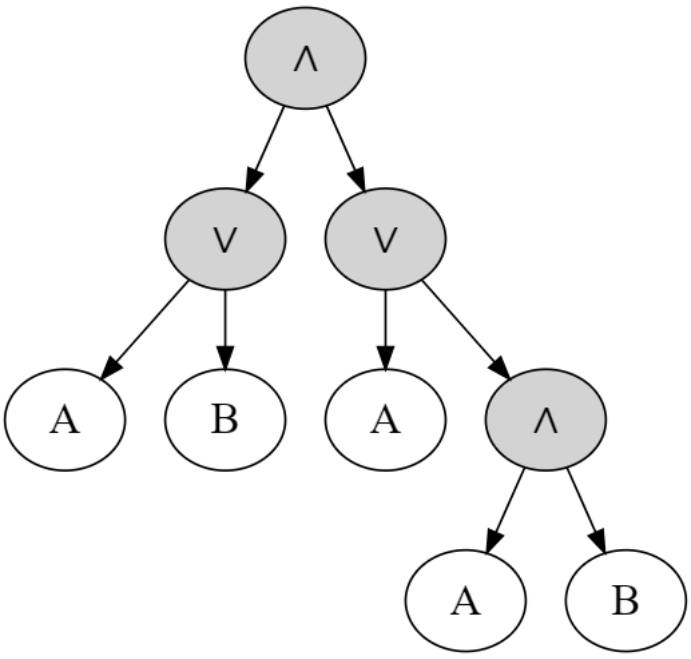


Figure 1: Tree diagram of $(A \vee B) \wedge (A \vee (A \wedge B))$

1.1.1 First programming language

From the idea of using logic as a programming language, Prolog was originated in the early 70s of the 20th century by Alain Colmerauer in Marseille as it is featured in Colmerauer and Roussel [1996]. It

was initially conceived for natural language processing and symbolic reasoning, although it has experienced a continuous evolution until reaching areas such as expert systems (Section 1.1.5) or formal verification: a field where the focus is on verifying the correct behaviour of logic circuits, protocols or software programs. Other areas where Prolog can be found are bioinformatics or data analysis.

In despite of its multiple and different applications and its progression, Prolog has kept its name during the last 50 years. For this reason, Körner et al. [2022] distinguishes a Prolog implementation from other systems that have enough differences to be considered a separate language. One of this common properties is the capacity to support SLD resolution and Horn Clauses, which will be key concepts to be discussed in subsection 1.1.2.

There is an important test to check whether a logic solver can be considered as a Prolog system or not. This test is based on the *append* function, which is used to concatenate lists, and it is created with two rules:

1. For any list `X`, the empty list and `X` *append* to form the list `X`.
2. For any elements `H`, `X`, `Y` and `Z`. The list formed by concatenating `H` and `X` *append* with `Y` to form a list formed by concatenating `H` and `Z` is `X` and `Y` *append* to form `Z`.

Now, it is displayed how these two rules look in Prolog, first block in Code 1.1, and Scheme, second block in Code 1.2. The differences are notorious, specially when implementing the recursion in the second rule and breaking the first list in two parts. In Prolog, these parts are usually named *head* and *tail*, while in Scheme it is used the concept of *car* and *cdr*, two primitive procedures that refer to the first element of a list (*car*) and to the rest of them (*cdr*), as it will be explained in Section 3.1, since none of them are necessary to implement the *append* predicate.

```
append([], X, X).

append([H | X], Y, [H | Z]) :-
   append(X, Y, Z).
```

Code 1.1: *append* predicate in Prolog

```
(rule (append () ?x ?x))

(rule (append (?h . ?x) ?y (?h . ?z))
      (append ?x ?y ?z))
```

Code 1.2: *append* predicate in Scheme

This just serves as a glimpse of the differences between Scheme, which is a dialect of Lisp, and Prolog, that, at the same time, has a relation with Lisp. By Körner et al. [2022], this connection comes from the fact that the motivation of Prolog comes from three different areas of research: AI programming, automatic theorem proving and language processing. The first of those three had been present for about 15 years before the appearing of Prolog and it had already lead to the creation of Lisp, having some important features such as abstraction.

Before entering the next subsection with the explanations of the two most important concepts behind the implementation of a logical programming language and using the *append* test, it is worth mentioning here a good illustration of the flexibility present in logical programming. For almost every language, there is an *append* function that manipulates lists adding elements or other lists to them. It is very useful and widely used. From an imperative perspective, this normally translates into giving two arguments, being the first one usually a list and the second one a list or element and the function

returns one list integrating them. Usually, there are not more options. However, the flexibility in logical programming allows the user in, either the Prolog or Scheme implementation of the function *append*, answer more questions. For example, if one gives an interpreter the query `(append (1 2 3) ?y (1 2 3 4 5))`, the result will be `(append (1 2 3) (4 5) (1 2 3 4 5))`, as it has enough information and capacity for knowing which list has to be *appended* to the first argument to give the third one. Even more, if given `(append ?x ?y (1 2 3 4 5))`, it would produce all the possible combinations of `?x` and `?y` that engender the list of the third argument.

1.1.2 Horn Clauses and SLD Resolution

Introduced by the mathematician Alfred Horn in 1951 [Horn, 1951], Horn clauses were designed to limit the language in First-order logic. One way to define a Horn Clause is: a clause where, at most, one of its literals is positive. From this first definition, it comes a classification of the Horn Clauses into two types: positive (or definite), when there is one positive literal, and negative, when there is none. A first general expression of these kinds can be found at 2, where the symbols $\neg p_i$ or q represent the literals, the basic building blocks of Horn clauses, which can be understood as an atomic proposition in logic, i.e., either a predicate (q) or the negation of a predicate ($\neg p_i$).

$$\begin{array}{ll} \neg p_1 \vee \cdots \vee \neg p_n \vee \quad q & (A\ positive\ Horn\ Clause) \\ \neg p_1 \vee \cdots \vee \neg p_n \vee \neg q & (A\ negative\ Horn\ Clause) \end{array} \tag{2}$$

However, when defining rules, the most practical way is to do it following the next structure: *if* p_1 *and* p_2 *and ... and* p_n*, then it implies* q. This can be satisfied easily using the logical equivalence $\neg A \vee B \equiv A \Longrightarrow B$ [1]. Consequently, the two kinds of Horn Clauses can also be written as they appear in equation (3). This notation is more helpful to navigate through the examples that will be explained later, although both ways of writing the clauses will be used along the text.

$$\begin{array}{ll} p_1 \wedge \cdots \wedge p_n \rightarrow \quad q & (positive) \\ p_1 \wedge \cdots \wedge p_n \rightarrow \neg q & (negative). \end{array} \tag{3}$$

In First-order logic, there is a subset called *quantifier-free Horn Clause Logic*, where the quantifiers, such as $\exists$ (existential) or $\forall$ (universal) are treated in a particular way. As the name hints, there are not any quantifiers. The universal ones are assumed to be implicit in the logic and for the existential ones it can be performed a process called *Skolemization* [Beyene et al., 2013].

Once all the objects in the data base are expressed in the form of a Horn clause, it can be performed the process called Horn Clause resolution, which is the basis for a logic programming language as Prolog. Two of these clauses can generate a resolvent clause, with the condition that one of them has to be positive. The type of the resolvent one will be determined by the other clause. Examples of the two different cases can be found at 4 and 5 respectively.

$$\begin{array}{rl} \neg p_1 \vee \cdots \vee \neg p_n \vee \quad q & + \\ \neg p_1 \vee \cdots \vee \neg p_n \vee \neg q & - \\ \hline \neg p_1 \vee \cdots \vee \neg p_n & - \end{array} \tag{4}$$

$$\begin{array}{rl} \neg p_1 \vee \cdots \vee \neg p_n \vee \quad q & + \\ \neg q \vee \quad s & + \\ \hline \neg p_1 \vee \cdots \vee \neg p_n \vee \quad s & + \end{array} \tag{5}$$

This resolution method is called SLD (*selected literals, linear pattern, over definite clauses*). It was first described by Robert Kowalski in 1974 [Kowalski, 1974] and then named that way in 1982 by van

[1] It is a straightforward equivalence, it can be seen by building the following truth table:

A	B	$\neg A$	$\neg A \vee B$	$A \Longrightarrow B$
1	1	0	1	1
1	0	0	0	0
0	1	1	1	1
0	0	1	1	1

Emden and Apt [Apt and van Emden, 1982]. Then, if a problem can be enunciated in way that all the clauses within its database are Horn Clauses, then SLD resolution is a valid method to solve it.

It is worth mentioning that there exists a general resolution method, which is more extensively explained in chapter 4 by Brachman and Levesque [2004]. Similarly to the SLD resolution, there is one rule of inference, stating what formulas can be inferred from other formulas. This single rule can be synthesised in:

> Given two clauses, the first one of the form $p_1 \vee q$ and the second one containing the complement of q, $p_1 \vee \neg q$. The clause $p_1 \vee p_2$ is inferred, including the literals other than q of the first clause and the ones other than its complement of the second one. The clause $p_1 \vee p_2$ is called a *resolvent* of the two input clauses with respect to the literal q.

Hence, a set of clauses can have more than one different *resolvent*. The clauses $r \vee s$ and $\neg r \vee \neg s$ have two *resolvents*, one with respect to r, $s \vee \neg s$ and other with respect to s, $r \vee \neg r$. The *resolvent* of two complementary unit clauses is the empty clause. The process where a clause p is *derivated* from a set of clauses $\mathcal{S}$ through a sequence of clauses which are *resolvents* of two earlier clauses in the process is called *Resolution derivation*.

Now, some examples will be used to show the explained concepts, but before that, it is important to remark that, although this bachelor thesis is outlined from a logical programming point of view and the student, as it will be shown during the document, got familiarised with the former concepts and learn the principles of logic programming, all the code included in these examples will be written in Scheme, a Lisp dialect, that is, indeed, a language created for functional programming. Nonetheless, as the title indicates, the ultimate goal is to understand the basis of logic programming and, at the same time, learning how to implement this structures in Scheme, ending in the implementation of a logic programming interpreter in this language. Additionally, as it will be indicated in the final part of the Introduction, two folders with exercises written in Prolog from the book by Bratko [2001] can be found in the working directory, demonstrating that in order to better understand the basis of logical programming is also important to work with a suited programming language.

1.1.3 Examples

Example 1

Let the first data base contain the following facts (or assertions) and rules. It can be highlighted as the rule found in the database Code 1.3 could be understood as, $\forall x$, if x is a man, then x is mortal, it is one way to check why the universal quantifier is assumed to be implicit in the logic.

```
; 1. Fact
(man Socrates)
; 2. Rule
(rule (mortal ?person)
      (man ?person))
```

Code 1.3: Example 1 database

The query, that is the question it has to be asked, i.e., a way to formulate the problem, is if Socrates is mortal. In the Scheme interpreter, this would be done using the following order as an input to the interpreter: (`mortal Socrates`). If this would be done in the interpreter that has been implemented, the output would be (`mortal Socrates`), confirming that the fact could be deduced from the database and that it is true. If it were not true, it would no return anything.

Using the SLD resolution explained above, it is going to be shown the deductive process that it is performed in the diagram of Figure 2. We start from the negated goal, that it is a negative clause expressing the opposite to the goal, (`not` (`mortal Socrates`)). The rule present in the database is a positive Horn Clause, then, it can be written (in Scheme), as (`or (not (man ?person)) (mortal ?person)`). Together with the fact that Socrates is a man, it can be deducted that Socrates is mortal[2].

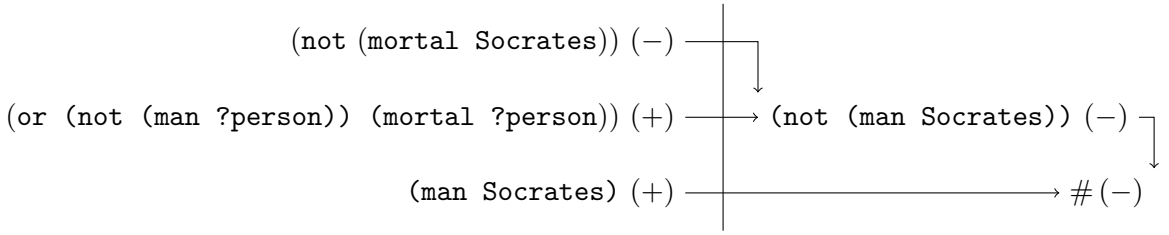


Figure 2: SLD resolution process detailed for Example 1

Example 2

The following example is to show that First-order logic is semi-decidable. This can be seen finding a case where the SLD resolution does not guarantee an ending. Hence, there will be an infinite loop. In this case, the database Code 1.4 only consists on one rule[3].

```
(rule (lastname ?person ?name)
      (lastname (father ?person ?father) ?name))
```

Code 1.4: Example 2 database

If it is tried to guess if a person called *Jim* has as lastname *Halpert*, the following query would be submitted to the interpreter: `(lastname Jim Halpert)`. If it was true, the same as in the previous example would happen, it would be returned the same fact. If not, nothing would be returned. Applying the SLD resolution once, this would be the result:

```
(or (not (lastname (father ?person ?name) ?name)) (lastname ?person ?name))
(not (lastname Jim Halpert))
-------------------------------------------------------------------------
(not (lastname (father Jim Halpert) Halpert))
```

It leaves as a resolvent a negative clause in which it can be applied the same SLD resolution again, giving:

```
(or (not (lastname (father ?person ?name) ?name)) (lastname ?person ?name))
(not (lastname (father Jim Halpert) Halpert))
-------------------------------------------------------------------------
(not (lastname (father (father Jim Halpert) Halpert) Halpert))
```

The method has not been able to end yet, and, as it can be already inferred, it could go on and on infinite times, in an endless recursion. The next iteration would return as a resolvent the clause:

```
(not (lastname (father (father (father Jim Halpert) Halpert) Halpert) Halpert))
```

[2]The null clause, that is the consequence of the presence of two clauses opposite to each other, lies within the negative clauses. It will be denoted with the symbol #, although it can be found also as ⊥ in the related literature.

[3]As it can be noted, the `father` relation is not a predefined rule, although it will be assumed it is. From how it is used. In [Abelson and Sussman, 1996, exercise 4.63], that can be found in the file *ExercisesLogicProgramming.rkt*, it is built a father rule using the condition that one person is the father of other person if the latter has been defined as the son of the other person or if there exists a wife of the former who has a son.

This example shows that it is important to define a good database in order to work with it. It does not mean that a database can not be useful with only one rule because our interpreter contains more rules that are added implicitly. For example, there is a rule to define the relation of two objects being the same, which is very useful in cases where it is wanted to distinguish between two different objects that have the same properties, but are different. The rule is defined by `(rule (same ?x ?x))`. If there was a query of the form `(same Fernando Fernando)`, the output would be `(same Fernando Fernando)`, as it is true and no other fact or assertion is necessary.

Example 3

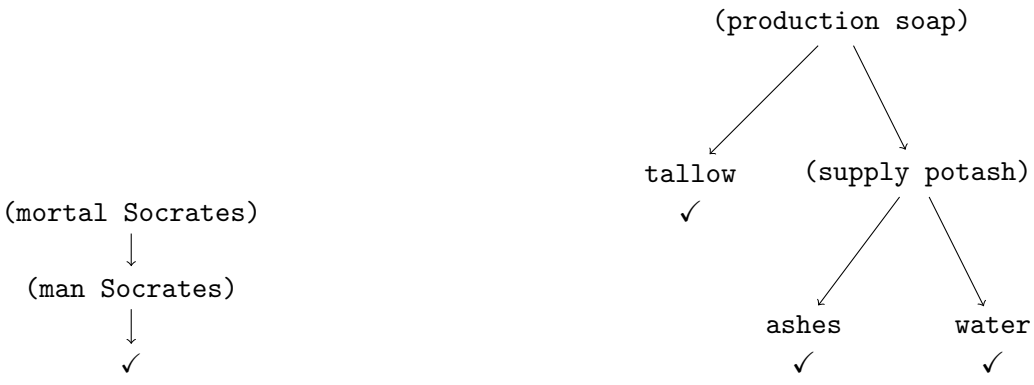
In this example it will be introduced the concept of *goal tree*. A *goal tree* is just another way of representing the process of the SLD resolution showing the objectives of each step. For the Example 1 (1.1.3), the *goal tree* is very simple. Firstly, it was wanted to check is Socrates was mortal, then, the next aim was to verify is Socrates was a man, and as it is was a fact in the original database, the process was terminated successfully. Its correspondent diagram can be found at Figure 3a.

Now, it will be analysed a more complex problem. The idea is to produce soap and to achieve it, it will be needed a first ingredient, which is tallow and a supply of potash. At the same time, potash is not directly available, it needs two ingredients: water and ashes. It is displayed the *goal tree* that describes the list of objectives for the example at Figure 3b.

Commenting on a first version of the database Code 1.5, the facts are different to the one in Example 1, as they describe a relation between two objects instead of a property of one. This was made with the aim of specifying that those three ingredients combine give soap. This is more clearly explained with an alternative database Code 1.6.

One would be tempted to think that both databases are equivalent, as they generate the same output to the query `(production soap)`, which is `(production soap)`, as it can be produced soap. However, what would be the output in the case the query was `(production steel)`?

The first database Code 1.5 would not return anything, as it is defined that tallow is an ingredient only for soap and that water and ashes are only for making potash. Nonetheless, the output for the second database Code 1.6 would be `(production steel)`! This is because the argument `?soap` in the second rule (so does `?potash` in the first one) does not represent anything, although it has been written in that way for a clearer understanding of the database. Hence, as it is defined the second database Code 1.6, any product can be made, as the three ingredients have been created as generic



(a) Example 1 (b) Example 3

Figure 3: Goal Trees for Examples 1 (left) and 3 (right)

ingredients, instead of ones related to certain product. This reasoning was the one behind the election of the first database Code 1.5 as the definite one, where the process of making soap is a concrete one. A final remark would be that, if this database was to be expanded and include other products to be manufactured, it could be the case where, for example water, will be used for other thing. Then, two options arise: one is to create one fact for each product that needs water; a second one would be to declare water as a generic product, as it is in the second database. The election is up to the programmer, as the key is to elaborate a non confusing database.

```
; 1. Facts
(ingredient tallow soap)
(ingredient water potash)
(ingredient ashes potash)
; 2. Rules
(rule (supply ?potash)
      (and (ingredient water ?potash)
           (ingredient ashes ?potash)))

(rule (production ?soap)
      (and (ingredient tallow ?soap)
           (supply potash)))
```

Code 1.5: Example 3 first database

```
; 1. Facts
(ingredient tallow)
(ingredient water)
(ingredient ashes)
; 2. Rules
(rule (supply ?potash)
      (and (ingredient water)
           (ingredient ashes)))

(rule (production ?soap)
      (and (ingredient tallow)
           (supply potash)))
```

Code 1.6: Example 3 second database

Example 4

A last brief example will be used to illustrate another case and stress how implementing a database is highly user dependent. The fictional situation is that if a person is able to arrive home or not, with up to four means of transportation: bus, taxi, foot or bike. The *goal tree* diagram, in Figure 4, is very simple, as each form of transportation is independent from the others and only one is sufficient to arrive to the destination desired. Then, having one available is enough. In the example, only the bus is a valid option. This is represented in the database with the fact `(transport bus home)`. As discussed in the previous example, it is obvious that by bus a person can reach multiple destinations, yet the interest of this database 1.7 is to guarantee to the user if he or she can arrive home. The rest of the options are coded, but commented.[4]

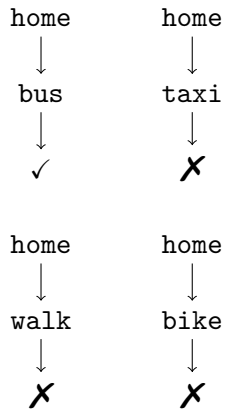


Figure 4: Goal Tree for Example 4

```
; 1. Facts
(transport bus home)
;(transport taxi home)
;(transport walk home)
;(transport bike home)
; 2. Rule
(rule (arrive ?home)
      (or (transport bus ?home)
          (transport taxi ?home)
          (transport bike ?home)
          (transport walk ?home)))

))
```

Code 1.7: Example 4 database

[4] In Scheme, the semicolon serves as the comment symbol, as it could be inferred from previous databases.

1.1.4 Improvements over the algorithm

Two possibilities are contemplated to improve the SLD resolution: forward chaining, which is considered to be data-directed and backward chaining, goal-directed. One typical example to distinguish between these two concepts is a situation where a supermarket's owner wants to find out some patterns based on the behaviour of its clients, whose purchasing data is available. If forward chaining is applied, the patterns are extracted from the data itself, without making any assumptions, the data is analysed and conclusions are drawn. Backward chaining is useful when the goal is to know who is more likely to buy frozen food, that is when we have a defined goal. After the goal is set, data is studied to solve the problem.

Prolog is a language that it is based on SLD resolution and backward chaining. Example 1 can be an easy example where backward chaining is used, as a query can be understood as a goal. Then, in that case, knowing if Socrates was mortal can is the goal. From here, it is applied the method it was explained. It is reasoned backwards until, in this case, found a fact from the database that generates the null clause. Forward chaining is not considered a resolution method, although, as it can be found in [Bratko, 2001, section 15.3.2], it can be implemented an interpreter in Prolog that uses a forward chaining reasoning.

Indeed, there is a tool called CLIPS (C Language Integrated Production Building), which is very powerful for building and developing of expert systems, that uses forward chaining to solve the problems. CLIPS syntax is based on Lisp, which makes it easier to understand from the functional programming reference. As it relies on forward chaining for reasoning, it looks for facts that match the conditions of the rules given, expanding its knowledge base. If the Example 1 is tried to replicate using CLIPS, the code from Code 1.8 would be needed.

```
; A rule is defined in a database that will be called 'rules'
(defrule mortality
  (man ?x)
  =>
  (assert (mortal ?x)))

; Working now in the console, the database is loaded
(load <rules>)
; A fact is defined in the working memory (temporary)
(assert (man Socrates))
; Now it is checked the facts in the current working memory
(facts)
f-0 (initial-fact)
f-1 (man Socrates)
; (run) command for using the rules from the database
(facts)
f-0 (initial-fact)
f-1 (man Socrates)
f-2 (mortal Socrates)
```

Code 1.8: Example 1 implemented in CLIPS

In Code 1.8, it is presented how, using forward chaining, the facts are extracted from the database without the need of using queries or assumptions. One remark is that the database is only composed by rules, the facts are stored in a temporary working memory. Some facts are defined by the user and then more facts are triggered by the loaded rules. More about CLIPS can be found in the reference manual T.X.N. [1993].

1.1.5 Expert Systems

In the context of logical programming it is necessary to define what is an Expert System. The idea is to create a program that, in a specific domain of application, is able to behave like an expert, i.e., solving problems that demand specialised knowledge in an area of expertise. It uses logical rules and facts as the ones defined before.

There are three fundamental components an Expert System should have: database (or knowledge base); inference engine, that is the one who know how to use the facts and rules of the database and a user interface. This last one is vital, as an Expert System must have the ability of explaining its decision and transmit the reasoning behind them, especially necessary in some areas as medical diagnosis. This is a common use, and this implies that it is usually requires that an Expert System has to be able to handle incomplete or uncertain data. Other examples of application of Expert Systems apart from medical diagnosis can be financial analysis or environmental monitoring.

1.1.6 Conflict Resolution

In modern systems, the databases (or knowledge bases) are usually composed of many facts and rules. This leads to some redundancies that generate inefficiencies. For this reason, the solutions to some problems or answers given to some questions, although correct, could be absurd. A ridiculous example could be that, when it rains, some system who tracks the environment of a garden could inform us that there are four leaves of a certain tree that are getting wet instead of giving relevant information about the ambient humidity of other important factors for the well-being of the plants.

This kind of conflict can be resolved too, paying special attention to repeated rules that try to express the same relations in a different way of, in a more basic level, how the statements or rules are ordered within the database or within a rule itself. In order to clarify this situation, a simple case will be described in where it is tried to calculate the greatest common divisor of two numbers. This procedure (Code 1.9) is decomposed in two rules. The first one states that, if the second number is 0, the *gcd* is the absolute value of the first one. The second one implements Euclid's algorithm. Seeing Code 1.9 one could think that the goal is achieved by this implementation.

However, the order of the two calls inside the second rule provokes an error, as it will be warned that the arguments are not sufficiently instantiated because `R` is has not been instantiated when calling `gcd(B, R, D)`. Thus, the correct order would be having `R is mod(A,B).` before doing the recursion on the *gcd* function. Still, this is not enough, as it is necessary to include the control predicate *cut (!)* in the first rule before of `D is abs(A).`. If not, Prolog would continue searching for alternative solutions even is one is found. This would lead to an error, as when dividing by zero in the second rule would not be possible. In other words, the control predicate *cut (!)* limits backtracking and end it when it is no longer necessary. The final version of the correct implementation of the *gcd* predicate would be the one in Code 1.10.

```
gcd(A, 0, D) :-
    D is abs(A).

gcd(A, B, D) :-
    gcd(B, R, D),
    R is mod(A, B).
```

Code 1.9: First try of implementing *gcd* predicate in Prolog.

```
gcd(A, 0, D) :-
    !,
    D is abs(A).

gcd(A, B, D) :-
    R is mod(A, B),
    gcd(B, R, D).
```

Code 1.10: Second try of implementing *gcd* predicate in Prolog.

Now, defining the same *gcd* predicate in an Scheme syntax offers some differences with respect to the Prolog one. The code is shown at Code 1.11.

```
(rule (gcd ?a 0 ?c)
      (cut)
      (gcd ?a 0 abs(?a)))

(rule (gcd ?a ?b ?c)
      (gcd ?b (remainder ?a ?b) ?c))
```

Code 1.11: Implementation of *gcd* predicate in Scheme.

Using Scheme over Prolog for this particular case implies one advantage and one disadvantage. The beneficial aspect is that what in Prolog was separated in two lines that had to be ordered in a precise way, here it is integrated. In the next chapter it will be explained how evaluation works in Scheme in a more extensive way, but because of how it is done, `(remainder ?a ?b)` is a subexpression of `(gcd ?b (remainder ?a ?b) ?c)` and it will be evaluated earlier. The detrimental part comes from the fact that, as it will be also cover later, it needs the implementation and importation of a library containing the information of how to deal with the primitive procedure `remainder` in the logical programming interpreter. In the same way, it would be needed to incorporate in the same library, or a different one, the operator *cut (!)*, defined here as a procedure with no arguments and reproduce the same process as it is done in Prolog.

1.2 Lisp language

This bachelor thesis was decided to be done using the programming language Lisp, that stands as an acronym for LISt Processing. It was conceived by John McCarthy, based on his paper McCarthy [1960]. It has various dialects and, in particular, Scheme is the selected one for the present work, which is popular because of being a powerful educational tool.

Visually, the most recognisable characteristic is its syntax being parenthesis-based. It reminds to the Polish notation (also known as prefix notation), where operators precede operands, instead of between them, typical of the infix notation. For example, in Polish notation, the addition of 1 and 2 is expressed as $+\ 1\ 2$. As in Lisp, primitive procedures do not have a fixed number of arguments, it cannot be achieved the full simplification of Polish notation in more complex combinations. For example, when wanting to calculate, in infix notation, $(3 - 8) * 5$, if assumed a given arity of two, it could be expressed as $* - 3\,8\,5$. However, parentheses are used in Lisp for this combinations, as it would be expressed $(*\ (-\ 3\ 8)\ 5)$, as the arity for the $-$ operator is undefined.

Another important feature is dynamic typing. In languages like C++, variables have to be defined with a certain type and in the compilation process this is an important factor. However, in Lisp, the dynamic typing generates a lot of flexibility for the user, as everything is treated as data. A different advantage of Lisp is that functions can be understood as values too, meaning that they can be passed as arguments and be returned. This is very valuable for abstraction and it is one of the basis of the functional programming paradigm, where it can be highlighted the ease to implement interpreters. Thus, as the name illustrates, in functional programming, functions are what is called a *first-class citizen* entities, meaning that support all the operations that can be done to other entities such as the one already mentioned: being passed as an argument, being the output of a function and being assigned to a variable. Here it lies the difference with respect to object-oriented programming, where the focus is on objects, which have data (atributes) and functions (methods).

1.3 Contents of the bachelor thesis

1.3.1 Objectives

The main goal of this bachelor thesis was the implementation of a logic programming interpreter as a basis for a way to understand logic programming. Having established that, it was decided that the best approach to achieve it was to follow the book *Structure and Interpretation of Computer Programs*, by Harold Abelson and Gerald Jay Sussman with Julie Sussman, usually named just *SICP*, Abelson and Sussman [1996]. This book is considered one of the bibles of computer science and it was used at MIT from 1984 until 2007 within the introductory programming course 6.001. This book covers a lot of advanced topics related to computer science, a lot of them necessary to gain a better insight about this project. Other of them are included in the studied exercises, although they are not relevant enough with respect to the implementation of the interpreter to be commented in this report.

Due to the set goal, it was required to become familiarised with the fundamentals of computer science such as recursion of abstraction. Hence, the most vital phase of the bachelor thesis was to read through the book and do as many proposed exercises as possible in order to understand the basis behind computer program in order to make our interpreter efficient. That is why the language selected in the book is Scheme, as it has been explained, is very simple and it requires to build almost everything from scratch, allowing the user to understand better the programs.

At the same time, as it was noted before, when facing the second phase of the thesis, that is, when learning about the elements of logic programming in order to implement the interpreter and the exercises derived from it, it was used the book *Prolog. Programming for Artificial Intelligence* by Bratko [2001].

Lastly, from a mathematical point of view, this bachelor thesis has the humble goal of exploring one of the ideas behind the thesis of Church-Turing, that is, all the computing paradigms are equivalents. This objective has been achieved through the implementation of a logical programming interpreter using programs defined from a imperative paradigm. Of course it does not mean that the user has the same experience or that the same efficiency is reached. Nevertheless, it proves how the same problems can be solved in different paradigms, involving less or more effort.

1.3.2 Results

The main effort of this thesis is reflected in the exercises, especially those from the Abelson and Sussman [1996], in order to programme a full production system. All of them can be found at `https://github.com/ffriaslago/lambda`. The reader is more than welcome to look through them. Here it will be given a little guide of the content of each folder to navigate through them[5].

The difference between the extension *rkt* and *bak* files is that the former ones contain the proper exercise while the latter contain a recovery version of it, usually not a full one, so there is no point of paying attention to them, as they normally contain an incomplete or more simple version of the same exercises. It can be noticed too that, although it was tried to do most of them with the *DrRacket* editor, some of the exercises are in a PDF format for a better visualisation.

- Interpreter: it is composed by the two main files of the thesis.
  - The one named *ExercisesLogicProgramming.rkt* contains a set of problems of [Abelson and Sussman, 1996, section 4.4]. It serves as a preparation for composing a generous database to test the interpreter and to learn how to implement the Horn Clauses in Scheme.

[5]This same guide is the content of the README file of the repository.

  - The one named *LogicProgramming_Interpreter.rkt* contains the main goal of the bachelor thesis, it can be separated into three parts. The first one is all the definitions of the procedures necessary for the interpreter. The second one is the proper interpreter and the third one is a big database containing all the facts and rules derived from the exercises and the examples showed in the introduction.

- Bratko. Chapter 1: All the exercises from the first chapter of Bratko [2001], separated by sections. The format is *.txt*, although they were made using the platform `https://swish.swi-Prolog.org/` that creates files with extension *.pl* and can be used to run the code given.

- Bratko. Chapter 2: Same as previous folder, but for the second chapter of the book.

- SICP. Chapter 1: It consists of all the exercises from the first chapter of Abelson and Sussman [1996]. As it is an introductory one thought to learn the basis of Lisp, it was done completely.

- SICP. Chapter 2: It does not include all the exercises from chapter 2 of Abelson and Sussman [1996], albeit the majority of them. The ones from the last section of the chapter were discarded because it needed implementations explained in chapter 3 and it seemed ineffective to solve exercises that could not be run.

- SICP. Chapter 3: roughly 80% of the exercises were made. [Abelson and Sussman, 1996, section 3.4] was skipped, as concurrency was not a relevant topic for the interpreter. Finally, the last two subsections of Streams were omitted too.

- SICP. Chapter 4: It encloses most of the exercises from [Abelson and Sussman, 1996, section 4.1], where the book shows how to build the evaluator that the user had been using until there. It serves as a good exercise of decomposition and abstraction. Besides, it generates the basis to define a programming language from scratch and helps to understand how to implement an interpreter.

Lastly, it is worth mentioning that, because of how the exercises are proposed in the book, not all of them can be run correctly. This does not mean that they are wrong. However, with the intention of a clearer presentation of the main structure, some necessary procedures were omitted. The book introduces plenty of procedures that later are required for some exercises. In some of them, those procedures are present. It was not followed a certain criteria, but some of the reasons behind writing a complete script or not were finding the equilibrium between presenting the contribution made to the exercise and the reader being able to interact with the proposed solution. For this reason, some of the exercises can be difficult to understand if a reader has not gone through some sections of the original book.

1.3.3 Document Organization

It is important to emphasise again that, although being the ultimate goal of this bachelor thesis was the implementation of the interpreter, the main part was focusing on abstraction and extensibility, researching on the best possible techniques to deliver an extensible functional interpreter. This implies that a significant part of the work was the resolution of the exercises that lead to that objective. This document serves as a report of it and, at the same time, it tries to synthesise the process developed to reach that target. At the same time, it can set a starting point for other bachelor theses or future works. It can serve as a benchmark, The implemented interpreter has some basic features and it is enough to reproduce some simple examples. However, there is enough room to explore the explained concepts of logical programming in a more profound way.

To that end, this extensive introduction explains the logical basis behind the interpreter and illustrates with several simple examples how it works depending on different scenarios. The following

chapters are related to the chapters of Abelson and Sussman [1996] in the way that they try to summarise the most important concepts seen in each of them that are crucial to the implementation of the interpreter.

In Chapter 2, it is covered the linear recursion and its differences with respect to iteration. Chapter 2 of Abelson and Sussman [1996] involves a wide range of concepts, as it introduces how to build abstractions with data. A couple of complex exercises such as the 8 queens problem will be explained and it will be compared the distinction between data directed and message passing programming. Chapter 3 will address streams, a very useful tool in terms of efficiency.

Finally, last chapter will detail the logic programming interpreter, paying special attention to the highlighted components of the previous chapters, explaining the database that was implemented and offering the user a quick guide on how to use it and what to expect from it.

2 Procedures with Lisp language: a first level of abstraction

2.1 A computational model: λ-calculus

One of the fundamental question in Computer Science, is how to define a computation. One of the first computation models is λ-calculus, proposed by Alonzo Church, an American mathematician and logician that was a collaborator of Alan Turing. His idea was to provide meticulous basis for analysing the concepts of a function and its application. Indeed, λ-calculus is a programming language that was created as a mean to represent all possible computations. It has only three different objects: variables, functions and applications.

A notable observation is the absence of integer numbers within the components of λ-calculus. However, as seen in *Exercise2.6.rkt*, they can be defined through *Church numerals*, that can be implemented in Scheme because λ-calculus had a big influence in designing Lisp, and the same structures are present nowadays. For this reason, it will be showed how a computation in λ-calculus using the Scheme notation, as examples in the introduction were written. The biggest difference is that, as it was firstly defined, λ-calculus did only support single parameter functions, although this was improved later by Haskell Curry with a composing technique named *currying* after him. In Scheme, this is not a problem, as lambda expressions can have as many parameters as required. Nonetheless, for this explanation, although using Scheme language, single parameter functions will be used.

The general expression for a function[1] is `(lambda (⟨parameter(s)⟩) ⟨body⟩)`, then, the most basic one, the identity, will be expressed as `(lambda (x) x)`. For evaluating this function, the correct syntax is `((lambda (x) x) 1)`, that gives `1`. The process of evaluation of a lambda abstraction is named *β-reduction* and it is just substitution. If it is wanted to define a higher order function, i.e., one that would be able to work with more than one parameter, it is possible, the input variable of a function can be another function, as in `(lambda (⟨par. 1⟩) (lambda (⟨par. 2⟩) ⟨body⟩))`. An easy example of how to sum two numbers is `(((lambda (x) (lambda (y) (+ x y))) 2) 1)`, that returns `3`, assigning `2` to `x` and `1` to `y`. Functions may be arguments too, one example of this can be `((lambda (f) (f 3)) (lambda (x) (+ x 1)))`, which gives `4`. In the Code 2.1 it is given a glimpse of the first Church Numerals next to a function to add 1 (this was the followed process to generate the definitions of `one, two` and `three` from `zero`. For a general addition procedure, consult *Exercise2.6.rkt*).

[1] The name of λ-calculus comes from the definition of the function

```
(define zero (lambda (f) (lambda (x) x)))

(define one (lambda (f) (lambda (x) (f x))))

(define two (lambda (f) (lambda (x) (f (f x)))))

(define three (lambda (f) (lambda (x) (f (f (f x))))))

(define (add-1 n) (lambda (f) (lambda (x) (f ((n f) x)))))
```

Code 2.1: Implementation of the first Church numerals and the `add-1` function.

2.1.1 Turing completeness of λ-calculus

λ-calculus is a language of theoretical interest, not practical, although it has a great influence in the creation of most of common languages such as Scheme, that will be covered in the next section. Apart from being important for the foundation of computer science, a major advantage in λ-calculus lies in its ease of expressing algorithms and, at the same time, implementing a machine-level interpreter. These features were ported into Lisp (Scheme is a dialect of Lisp). Before discussing in detail the technical features, it is going to be briefly explained what makes a programming language being *Turing complete* and showed why λ-calculus fulfills those requirements. Actually, it was Turing himself who proved that the two models were equivalent.

The most straightforward way of defining the property of being *Turing complete* for a programming language is the capacity of representing any computation possible on a Turing machine. It is important to remark that Turing completeness is a theoretical concept, as an ideal Turing machine would have an infinite tape and, in reality, memory is finite. Focusing on three concepts, it can be shown why λ-calculus is *Turing complete*:

1. Controlling the flow of execution based on conditions. In a Turing machine, the read-write head has the ability to move to any part of the tape depending on the instructions, that is, certain conditions that also depend on the read value. λ-calculus can replicate conditional behaviour. Although there are not true or false boolean values, they can be represented by the following definitions

   - True by `(lambda (x) (lambda (y) x))`, two arguments are passed (one to each lambda function) and the first one is selected.
   - False by `(lambda (x) (lambda (y) x))`, same as above, but the second one is selected.

   Generating the capacity of expressing *if* statements,i.e., conditional behaviour, such as: `((((lambda (cond) cond) (lambda (x) (lambda (y) x))) 1) 0)`, that will return `1` and `((((lambda (cond) cond) (lambda (x) (lambda (y) y))) 1) 0)`, that will return `0`. Recursion can be achieved too Church [1941].

2. Manipulating data. In a Turing machine, the read-write head was able to read data, like 1 and 0. It has been showed two different of data than can be built inside λ-calculus, like boolean values or numbers (Church numerals).

3. Expressing computations as data. This has already seen and explained, as functions can be the arguments of other functions and outputs too. This property is the one behind the construction of higher order procedures, enabling more powerful and complex techniques.

As a final commentary, there is not the possibility to 'storage' a function, in the sense a function cannot have a name or tag in order to use it later. Fortunately, Scheme implements a naming function, being much friendlier for the user.

2.2 The practical implementation: Scheme

Scheme was developed in the 1970s and it is conceived as a cleaner, more elegant Lisp dialect. It has key improvements like the performance for recursive functions. The main use of Scheme is educational because of its clarity and expressiveness, which made it a easy choice for the development of this bachelor's thesis, as a Lisp program usually consists of a large number of relatively simple procedures.

Some of the code displayed so far in this document was written in Scheme even though it has not been explained yet the basis of this programming language, a reader familiarised with reading code has already checked that is a simple language and easy to understand. The most elementary expressions are a *combination* of parenthesis, numbers (in base 10) and symbols such as `+` or `*` that represent primitive procedures. This combinations can be nested, i.e, combined operations and there is not limit to the depth of the nesting. Indexing (*pretty-printing* format) is used to clarify the hierarchy of the code, aligning subexpressions that are operated using the same operand[2]. An easy example is the one from *Exercise1.2.rkt*, represented in this document in the Expression 6 and in the Code 2.2.

$$\frac{5+4+\left(2-\left(3-\left(6+\frac{4}{5}\right)\right)\right)}{3(6-2)(2-7)} \qquad (6)$$

```
(/ (+ 5
      4
      (- 2
         (- 3
            (+ 6
               (/ 4 5)))))
   (* 3
      (- 6 2)
      (- 2 7)))
```

Code 2.2: Expression 6 in Scheme.

Having presented this, before explaining how the evaluation of a combination is done, in contrast to λ-calculus, Scheme allows the user to name variables (⟨*body*⟩ is empty) and functions in order to refer to them using their names. *Define* is Scheme's simplest means of abstraction and the general form of a procedure definition is `(define (`⟨*name*⟩ ⟨*formal parameters*⟩`)` ⟨*body*⟩`)`. Along with this, Scheme has some *special forms* for conditional expressions and logical operators. The general form of a conditional expression, which is often more useful than a *if* statement (it can also be implemented in Scheme with the structure `(if` ⟨*predicate*⟩⟨*consequent*⟩⟨*alternative*⟩`)` because it allows to have ⟨*consequent*⟩ clauses of more than one line is `(cond (`⟨p_1⟩⟨c_1⟩`) ... (`⟨p_n⟩⟨c_n⟩`))`. A more detailed description can be found at [Abelson and Sussman, 1996, section 1.1.6].

It is now crucial to explain how the evaluation is done, as there are some options. The one used is *applicative-order evaluation*, although in some exercises it has been compared to the *normal-order evaluation*. The interpreter that is implemented by default in the software used, *DrRacket* uses *applicative-order evaluation* and follows this rules when evaluating a combination such as the one in the above code (*Exercise1.2.rkt*):

1. Evaluate the subexpressions of the combination. This means starting in the deepest nested expressions. In the example, if it is wanted to calculate the innermost parentheses of the numerator, before summing 6 and the fraction, the operation `(/ 4 5)` has to be performed before, as it is a subexpression of that addition. Here, the subexpressions are simple combinations (the numbers do not need to be evaluated), in more complex procedures, the subexpressions can be calls to other procedures and a proper evaluation is required. This will be seen in more detail when explaining linear recursion with the `factorial` procedure.

[2] Revising the examples from the introduction, an example of this can be found in the third one. In both rules, it is used the `and` logic operator, and the two clauses that have to be checked to know the value of the operator are aligned.

2. Apply the procedure, that is the value of the leftmost subexpression to the arguments. In the example, all the procedures are primitive procedures, as they are arithmetic operations.

From these two rules, it is straightforwardly concluded that the evaluation process is recursive, as when evaluating the subexpressions, the same steps have to be applied again. An alternative evaluation process is the *normal-order evaluation*, where the subexpressions are not evaluated until needed. This process would reduce the evaluation into more basic procedures until reaching primitive operators and then it would perform the operation. In *Exercise1.5.rkt*, whose code is shown in Code 2.3, there is a very simple example that shows how different the two approaches are.

When using *applicative-order evaluation*, it evaluates the two subexpressions present in the call to the `test` procedure. `0` is a number, so it does not require evaluation. Conversely `(p)` is a procedure whose body is `(p)` too, the same procedure. This implies the evaluation results in an infinite loop as `(p)` gives `(p)` again and again. On the other hand, when *normal-order evaluation* is applied, the `test` procedure is expanded before evaluating the subexpressions. As the first argument is `0`, when using it in the predicate of the *if* statement, it returns the *consequent* clause, which is the value `0`, ending the process without evaluating the subexpression `(p)`, not entering an infinite loop.

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

(test 0 (p))
```

Code 2.3: Exercise1.5.rkt

2.2.1 Computational complexity

In computer science is always important to be aware of the computational cost of the programmes designed. Although it is true that efficiency has not been present in all the proposed solutions, it was an important factor in some of them, searching for the most efficient way taking into account two different measurements of complexity. For both of them it is critical to be able to find the parameter (most solved problems could be defined by only one) that determines the size of the problem. A function $S(n)$ has an order of growth $\Theta(f(n))$ if there are two positives constants $k_1, k_2 > 0$ independent of n such that it is fulfilled the relation from Equation 7.

$$k_1 f(n) \leq S(n) \leq k_2 f(n) \tag{7}$$

The first measure contemplated refers to temporal complexity, i.e., the number of steps or operations needed to be done. While the second one refers to the spatial complexity, i.e., the memory necessary to evaluate a process. One of the most useful aspects of knowing these complexities is that it allows the user to predict how the process is going to behave when solving a much bigger problem with the same code.

As every beginner coder has experienced, the first stages of learning a new language involve evaluating line by line a given code to understand how it works and the internal steps. Often, this is accompanied by simple tries with a very small size. Sometimes this lead to a wrong conclusion and the code does not provide successful results when subjected to a normal size problem, it has a very high computational cost. Exercises such as *Exercise1.22.rkt*, *Exercise1.23.rkt* or *Exercise1.24.rkt* explore this topic computing the running time when testing for primality in increasing orders of magnitude for

different variations of programmes. Besides, *Exercise1.14.txt* offers a great example of how, computing temporal and spatial complexity demands different efforts.

2.2.2 Recursion vs Iteration

When thinking about recursion, one the simplest mathematical examples that comes to mind is the computation of the factorial of a number. Through it, it will be illustrated the difference between linear recursion and iteration. As well, the same function will be used to explain the differences between the substitution model and the evaluation model, both serve to visualise the internal order followed by the procedure, although the first one becomes useless once more complex programming is developed.

Next, it can be found the code responsible for each version of the factorial procedure. Usually, the recursive procedure, Code 2.4, is much easier to programmed and to understand, as it express the function with the most elementary ideas. Although it can not be said that the iterative procedure, Code 2.5, is difficult, it requires a more sophisticated idea, especially when stipulating the new arguments for the call to the iterative function. However, *Exercise1.37.rkt* proved to be the opposite. From a human angle, it is easier to think recursively, so it comes more naturally to write a recursive code. In contrast, from a computational viewpoint, it is better to use a iterative structure, especially considering efficiency and optimisation. While recursion offers a conceptual simplicity for the user, pragmatism leads user to find iterative solutions.

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Code 2.4: Recursive definition of `factorial`.

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter
          product
          counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Code 2.5: Iterative definition of `factorial`.

A recursive process can be interpreted as an expansive one, where the operations are delayed until only primitive procedures are left, resulting in a contraction. This expansion takes a toll, as the pending operations need to be tracked. Both the number of operations (temporal complexity) and the number of steps (spatial complexity) grow linearly with n, being n the number the factorial is wanted to be calculated.

On the other hand, in an iterative process there are no delayed operations. Hence, the spatial complexity does not grow with n. The values that determine the final result are kept in each step. These are called *state variables*. In this particular case, `product, counter and max-count` are those variables. Regarding the temporal complexity, it is the same one as in the recursive one, $\Theta(n)$.

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Code 2.6: Recursive substitution of `factorial`.

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Code 2.7: Iterative substitution of `factorial`.

It has been displayed the substitution model for the two analysed options for computing a factorial: in Code 2.6, the recursive definition and in in Code 2.7, the recursive definition. Unfortunately, this model does not represent faithfully the process that is made by the computer. It is very useful as a didactic tool and to help the user to gain knowledge about the computation details. As a matter of fact, a lot of commented ideas have been purposely left in some exercises to show the reader how to approach some of the most complex problems and the substitution model was used plenty of times [1].

In chapter 3 of *SICP*, as a consequence of introducing assignment as another tool to incorporate to the computer science knowledge, the environmental model was presented. It replaces the substitution model because it is more specific, expressing in a more realistic way how procedures are applied to its arguments. In Figure 5 and Figure 6 environment diagrams for the recursive and iterative procedures are represented. It is critical to remark that both figures encapsulate the environment after the evaluation of `(factorial 6)`, as it could have been also represented the environment after evaluating the definition of them. An example of such of that diagrams can be found in *Exercise3.10.pdf* or *Exercise3.11.pdf*.

In both diagrams, when the evaluation is made, a new environment *E1* is created with the parameter *n* linked to the argument *6*. In Figure 5, this triggers another call to the same procedure, with the new argument being *5* instead of *6* and so on. In Figure 6, there is a clear difference after creating *E1*, as the body of the `factorial` procedure is not the same. The values of the *state variables* are updated during the iterative process.

[1] The 8-queen problem *(Exercise2.42.rkt)* is the best example. In chapter 3, it will be explained, but the code shown is a reduced final version of the solution.

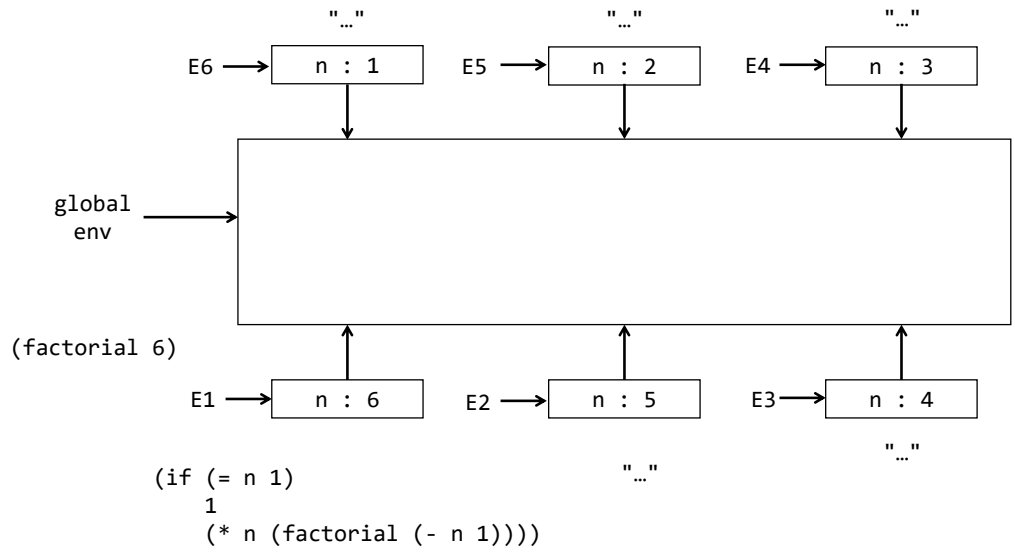


Figure 5: Environment structure created by evaluating `(factorial 6)` for the recursive procedure.

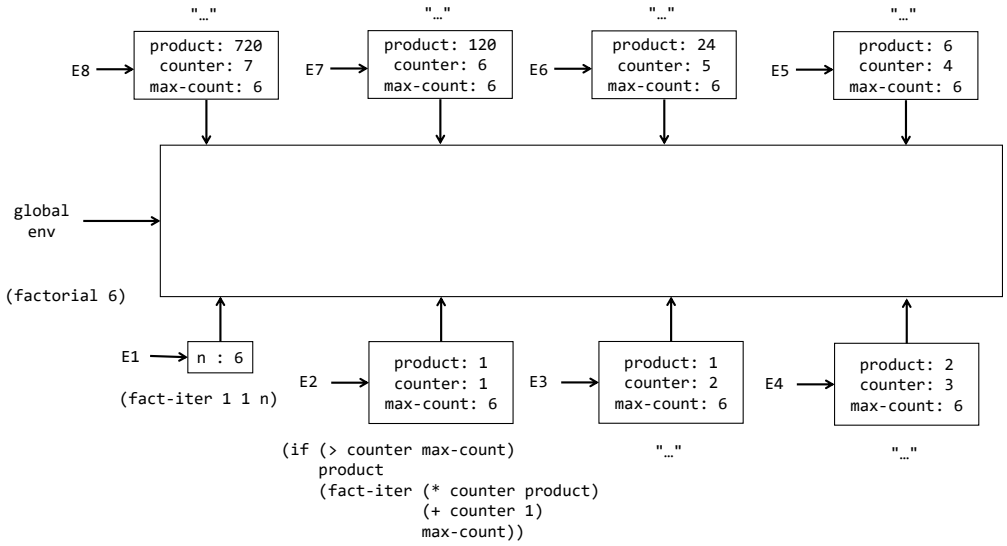


Figure 6: Environment structure created by evaluating `(factorial 6)` for the iterative procedure.

3 Techniques on Scheme: Approaches to Solving Problems

3.1 List structures

Scheme offers several ways to build lists and sequence structures for both numbers and arbitrary symbols. For a reader of Abelson and Sussman [1996], the first introduced primitive procedure is called `cons`, which serves for building pairs given two arguments. As the data inside the pair should be able to be extracted and manipulated, two already mentioned primitive procedures appear naturally, `car` for obtaining the first element and `cdr` for the second. Then, `(car (cons 1 2))` returns `1` and `(cdr (cons 1 2))` gives `2`.

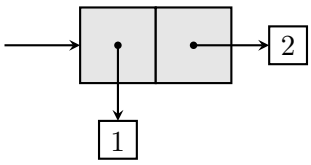


Figure 7: Box-and-pointer representation of `(cons 1 2)`.

In the Figure 7 there is a representation of the data object created by the procedure `cons`, this kind of diagrams are very useful for visualising more complex structures, as the arguments for `cons` can be other pairs, enabling the capacity to create hierarchical structures.

If it is wanted to work with longer sequences than pairs, as `cons` accepts as an argument a pair, one way to build them would be by concatenating evaluations of the `cons` procedure such as `(cons 1 (cons 2 3))`. However, it could also be done `(cons (cons 1 2) 3)`. When evaluating both expressions, they do not produce the same data object. This is one of the reasons for the use of other primitive procedure, `list`, which would be the equivalent to nest calls to the `cons` procedure in the way showed in Expression 8.

$$\begin{array}{c} \texttt{(list } a_1 \ a_2 \ \ldots \ a_n\texttt{)} \\ \Vert\Vert \\ \texttt{(cons } a_1 \ \texttt{(cons } a_2 \ \texttt{(cons } \ldots \ \texttt{(cons } a_n \ \texttt{nil}\text{[1]}\texttt{) } \ldots\texttt{)))} \end{array} \tag{8}$$

This equivalence between `list` and `cons` allows the use of `car` and `cdr` with `list` too, making them very powerful tools. Indeed, they can be combined to form procedures that are equivalent to apply them sequentially. For example, the procedure `caar` corresponds to applying `car` twice in a row. Using the same kind of representation as in Figure 7, it will be illustrated in Figure 8 the structure of the list `(list 1 2 3 4)`.

[1] In Abelson and Sussman [1996] it is used `nil` for naming an empty element, if using the names provide by DrRacket (the used software), it has to be used the word `empty`. The language used by each script present in the repository of GitHub is determined by the first line, being `#lang sicp` when using the same notation as Abelson and Sussman [1996] and `#lang racket` when using the one by default in *DrRacket*.

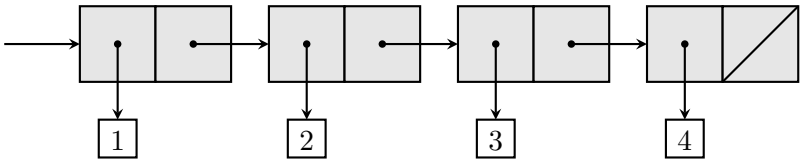


Figure 8: Box-and-pointer representation of `(list 1 2 3 4)`.

The diagram from Figure 8 permits to visualise better the outputs when using `car`, `cdr` and their combinations. Evaluating `(car (list 1 2 3 4))` gives `1`, which is the expected product. In the case of `(cdr (list 1 2 3 4))`, it is returned the data object `(2 3 4)`, i.e., a list formed by three elements. For extracting the element `2` it is necessary to apply `car` to the list generated by `cdr` in the original one. Then, `(cadr (list 1 2 3 4))` satisfies the question.

3.1.1 Quote operator

Up to this point, only numbers have been used to form sequences. If it had been used an arbitrary symbol, it would have lead to an error, since it would have been interpreted as an undefined variable. Nevertheless, it can be possible to build sequences of arbitrary symbols utilising the quote operator `'`. A relevant observation is that, by doing this, it is sacrificed the syntax followed until now with the use of parentheses to delimit compound expressions. When evaluating the primitive procedure `list` in `(list 'a 'b)`, the quote operator is a single-character abbreviation for the procedure `quote`. It is analogous to build the defined list using `(list (quote a) (quote b))`.

Another useful application of `'` is the ability of writing lists. When using the syntax from the language present by default in the software *DrRacket* (#lang racket), and a list is the object returned from an evaluation, that list is preceded by the quote operator[2]. Thus, by writing `'(1 2 3)`, this will be interpreted as `(list 1 2 3)`.

The power of the quote operator is explored along [Abelson and Sussman, 1996, section 2.3]. For example, in the trio of problems *Exercise2.56.rkt*, *Exercise2.57.rkt* and *Exercise2.58.rkt* it is examined how to represent algebraic expressions and implement symbolic differentiation, culminating in being able to evaluate expressions such as `(deriv '(* x y (+ x 3)) 'x)`.

3.1.2 Structures present in imperative languages

The introduction of sequences gives rise to multiple procedures that operate with them. Learning how to manipulate them allows the user to gain much flexibility and creativity when facing the resolution of some problems as the 8-queens puzzle presented in subsection 3.2.2. It can be started from very simple tasks such as defining a procedure for returning the last element of a list or reversing the order of them.

Just using the few tools explained together with the special form `let`, which allows the user to bind variables locally, often easing the readability of the code, and using linear recursion, the procedures of Code 3.1 and Code 3.2 are built.

It is worth mentioning that in Code 3.2 it is used the primitive procedure `append`, but the one defined from an imperative perspective. So, the *rules* from Code 1.2 are not used. Instead of it, the Code 3.3 is the definition employed here.

[2]When using #lang sicp, `'` does not appear before the parenthesis, but the quote operator can still be used to do the same construction as the one explained.

```
(define (last-element l)
  (let ((f (cdr l)))
  (if (null? f)
      l
      (last-element f))))
```

Code 3.1: Procedure for obtaining the last element of a list.

```
(define (reverse l)
   (if (null? l)
       nil
       (append (reverse (cdr l))
               (list (car l)))))
```

Code 3.2: Procedure for reversing the order of a list.

```
(define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
              (append (cdr list1) list2))))
```

Code 3.3: Definition of *append* procedure

Multiple higher-order procedures can be defined to abstract common operations that are usually done with sequences. The most basic one is `map`, whose arguments are a function and a list and it applies the function to all the elements in the list. Then, when using `map`, it is important to use a procedure that needs as many arguments as the variables passed to it.

Nested loops in Scheme

There are some frequent structures present in imperative languages that have not explicitly appeared yet, as the loops. Now it will be shortly expounded how this idea can be implemented operating with sequences. Using the `map` function it can be done a simple loop over a list. In order to get closer to a for loop, one option is to use a function to generate a list that contains all the integer numbers in a given interval. This is achieved with the procedure `enumerate-interval` from Code 3.4. Once done it, it can be used `map` with the wanted procedure to be applied over each number. A simple example is displayed in Code 3.5, where using a `lambda` function, it is cubed every number from 1 to 20.

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low
            (enumerate-interval
             (+ low 1)
             high))))
```

Code 3.4: Procedure for obtaining a sequence of integers in a given range.

```
(map (lambda (i) (* i i i))
     (enumerate-interval 1 20))
```

Code 3.5: Example of a simple for loop structure in Scheme.

It will be used another new higher-order procedure named `accumulate` that, given a list, an operation and an initial value (depending on the operation), it will apply the same operation over the elements of the list, accumulating the result in one variable. One instance of its use is `(accumulate cons nil (list 1 2 3 4 5))`, resulting in the sequence `(1 2 3 4 5)`. Some of this higher-order procedures can be redefined using other of them in the definition, some examples can be found at *Exercise2.33.rkt*, where `map` is defined using `accumulate`. To see other applications of `accumulate` *Exercise2.34.rkt* or *Exercise2.37.rkt* can be examined.

A very popular construction is the combination of the procedures `accumulate` and `map`. This is very useful when generating sequences that order pairs of numbers. From *Exercise2.41.rkt* it is taken the procedure `unique-triples` to show, in Code 3.6, a more complicated nested structure. The aim

of this procedure is to create a sequence of triples (i, j, k) (it is a list of lists) that fulfil being positive, distinct and being less or equal than n. Besides, the pairs are ordered, so $i < j < k$. The trivial case would be evaluating `(unique-triples 3)`, that would return `((1 2 3))`. It is pertinent noting here that, as the procedure is built to construct a sequence of triples, if only one triple can be generated, as it is in the trivial case, it is still preserved the structure of list of lists.

```
(define (unique-triples n)
  (accumulate append
              nil
              (map
               (lambda (i)
                 (accumulate append
                             nil
                             (map
                              (lambda (j)
                                (map (lambda (k)
                                       (list k j i))
                                     (enumerate-interval 1 (- j 1))))
                              (enumerate-interval 1 (- i 1)))))
               (enumerate-interval 1 n))))
```

Code 3.6: Example of a combination of mapping and accumulating.

The frequency of the combination of mapping and accumulating results in the definition of another higher-procedure, `flatmap`, displayed in Code 3.7, very useful when wanting to condense the code and avoid multiple nesting, as it will be the proven in Code 3.13.

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

Code 3.7: Definition of `flatmap`.

3.1.3 Mutable Lists

In the previous subsection it has been explained how to construct list structures and several higher-order procedures have been analysed to understand operations that can be employed when working with sequences. The next step would be how to modify those structures. Two primitive procedures, `set-car!` and `set-cdr!` are used for this purpose. Both of them take two arguments and the first one has to be a pair (or a list, from the equivalence 8). Using Figure 7 and Figure 8 it can be visualise better the modification done. It is implemented as a replacement of the pointer of the `car` or `cdr`, setting it to the second argument proportioned to the function.

The ability to mutate lists allows the user to work with the concepts of *queue* and *table*. The latter one is one of the basic concepts behind the implementation of the logical programming interpreter. In a two dimensional table, the used one, information is saved inside it making use of two keys. For this, it is utilised the procedure `put`, whose general form is `(put key1 key2 item)`. For extracting it, the procedure `get`, `(get key1 key2)`, is the one employed.

[Abelson and Sussman, 1996, section 3.3.3] contains a comprehensive presentation and explication of this concept and all its possible features, culminated with the exercises *Exercise3.24.rkt*, *Exercise3.25.rkt*, *Exercise3.26.rkt* and *Exercise3.27.rkt* (accompanied by its environment diagram *Exercise3.27_Diagram.pdf*).

3.1.4 Streams

Although being similar to a list, the data structure called *stream* gives solution to some problems when working with the data objects presented until this point. This new object grants the representation of very large or even infinite sequences without incurring in memory problems. Then, from an efficiency point of view, streams are a much more powerful tool.

When using the higher-order procedures seen along this section, the programs are obliged to fully build and manipulate the data structures at every step of the followed process. For a large size, this supposes a very high cost in time and space. An example of this situation is when trying to find a number that fulfils certain property in an interval. In *Exercise1.27.rkt* it was defined a predicate procedure to check if a given number is a Carmichael number[3] or not. If it is tried to evaluate the expression from Code 3.8, it is needed to construct a list of almost one million elements. Depending on the memory capacity, this instruction can be computed. The reason is that it needs to store the whole interval in memory.

```
(caddr (filter charmicael?
               (enumerate-interval 10000 1000000)))
```

Code 3.8: Expression for finding the third Carmichael number in the interval from 10,000 to 1,000,000.

It would be much adequate to be able to treat each number one by one and stop computing after the desired number is founded, which is 29,341. This is one of the motivations of defining streams. The idea is that it is a sequence partially constructed and it is only extended if needed. Streams have their own building procedures, although the idea with the respect to the ones explained remains. The constructor is called `cons-stream` and the two selectors, `stream-car` and `stream-cdr`. From here, extensions of other higher-order procedures can be implemented too such as `stream-map`.

The key of the stream definition is the use of the special form `delay`, which is accompanied by another special form named `force`. If the expression `(delay` ⟨`subexp`⟩`)` is evaluated, the ⟨`subexp`⟩ is not evaluated. Instead, it is created a *delayed object* that will be evaluated in an undefined moment of the future, because it might not even be necessary to evaluate it. Then, if `force` is used with the *delayed object*, the evaluation is performed at that moment.

The constructor `cons-stream` is not really a procedure, but a special form, as it uses `delay` to build a stream. Actually, the expression `(cons-stream 1 2)` is equivalent to `(cons 1 (delay 2))`. The selector `stream-car` has the same definition as in the lists context, however, when using `stream-cdr`, `force` appears, evaluating the *delayed object* that was pending in the construction of the stream. In Code 3.9 it is displayed how to solve the same problem it was attempted in Code 3.8 using streams instead of lists.

```
(stream-car
 (stream-cdr
  (stream-cdr
   (stream-filter charmicael?
                  (stream-enumerate-interval 10000 1000000)))
```

Code 3.9: Expression for finding the third Carmichael number in the interval from 10,000 to 1,000,000 using streams.

[3] A Carmichael number is a number that fools the Fermat test for primality.

Taking a closer look to what happens in Code 3.9, the procedure `stream-enumerate-interval` returns a stream whose `car` is 10,000 and the `cdr` is a delayed object that is not evaluated, i.e., it is not created a sequence of almost a million elements as it was done earlier. Then, when filtering[4] the stream and finding the Charmicael numbers, firstly it is only tested the `stream-car`. Since 10,000 is not a Charmicael number, `stream-filter` tries to analyse then the numbers contained in the `stream-cdr` of the input stream. This results in generating the next number of the interval, 10,001 and the same process will be repeated. 10,585 is the first Charmicael number greater than 10,000. When 10,585 is reached, as it fulfils the set condition, an stream will be built, being 10,585 its `stream-car` and its `stream-cdr` will be a *delayed object* where the filtering of the original interval will continue following the same process until the integer 29,341 is reached. The structure of the final stream returned is shown in Code 3.10. For a more exhaustive explanation of the implementation of streams and the details of all the definitions of the mentioned and used procedures, the reader is encouraged to go through [Abelson and Sussman, 1996, section 3.5] as well as all the set of exercises from *Exercise3.50.rkt* to *Exercise3.76.rkt*, where more advanced topic on streams are covered.

```
(cons 10585
      (cons 15841
            (cons 29341
                  (delay
                    (stream-filter
                     charmicael?
                     (cons 29342
                           (delay
                             (stream-enumerate-interval 29343
                                                        1000000)))))))))
```

Code 3.10: Structure of the stream generated by Code 3.9.

3.2 Selection of problems

3.2.1 Adjacency relation within a graph

Here it is presented a graph, the one in Figure 9, with 9 nodes and a set of edges between some of them. Code 3.11 tries to implement a database with a fact for each edge present in the graph and a rule to know if two nodes are adjacent or not.

The concept explored with this problem is how to deal with symmetric relations. If the graph is not an oriented one, the adjacency relation between two nodes is symmetric. That is what the rule defined in Code 3.11 is trying to define, in a very straightforward way.

However, when passing some very simple queries, several conflicts arise. As the adjacency relation should be symmetric, one expects that, if the query `(adjacent 7 5)` is passed to the interpreter, it returns that is true. For the logical programming Scheme interpreter, this would be expressed by returning `(adjacent 7 5)` too. Although it recognises the query and its truthiness, it enters in an infinite loop showing `(adjacent 7 5)` without stopping.

The conflict resides in that, when querying `(adjacent 7 5)`, first it is checked if there is any fact that matches that. As it is not any, it applies the rule. When doing this, within the rule, it is tried to satisfy `(adjacent 5 7)`. As it is a fact in the database, the successful return is given, but the rule is applied again, entering in the recursive loop. It would happen the same loop if the first query was a

[4] An implementation of the procedure `stream-filter` can be found in [Abelson and Sussman, 1996, section 3.5.1].

simple checking of one of the defined edges such as `(adjacent 4 5)`.

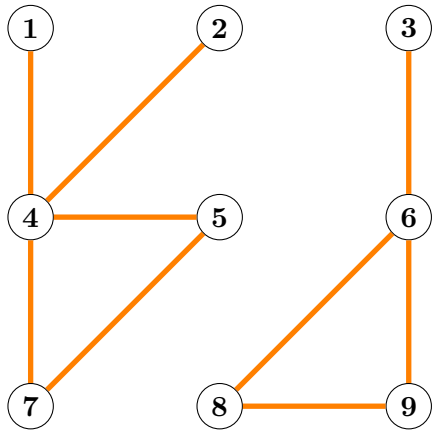


Figure 9: Graph representing the database from Code 3.11.

```
(adjacent 1 4)
(adjacent 2 4)
(adjacent 3 6)
(adjacent 4 5)
(adjacent 4 7)
(adjacent 5 7)
(adjacent 6 8)
(adjacent 6 9)
(adjacent 8 9)

(rule (adjacent ?x ?y)
      (adjacent ?y ?x))
```

Code 3.11: First version of the *adjacent* database

Trying a different form of a query with `(adjacent 4 ?y)` will be generated an infinite loop, going through the outputs `(adjacent 4 7)`, `(adjacent 4 5)`, `(adjacent 4 2)` and `(adjacent 4 1)`. Another flaw is that is unable of reaching a false answer when querying about two nodes that are not adjacent, as `(adjacent 2 5)`. In a Prolog interpreter, it would result a indefinitely running by recursively searching for `(adjacent 5 2)`. In the logical programming interpreter implemented, this will be noted differently, as the interpreter would request to the user itself to give the results, but no response is given after that.

All in all, a redefinition is necessary. There can be more than one option to solve that, the one chosen has been to include a restriction in the definition of the rule to stop the recursion. Maintaining the same facts, the only change in the database Code 3.11 would be the one in the Code 3.12. In a later chapter it will be explained the role of `lisp-value` with more details, but it allows the application of a predicate to some arguments. In this case, it forces `?y` to be smaller than `?x`, introducing the order of the natural numbers in the adjacent relation.

```
(rule (adjacent ?x ?y)
      (and (lisp-value < ?y ?x)
           (adjacent ?y ?x)))
```

Code 3.12: Alternative definition for *adjacent* rule.

Another option would have been to use a different definition for the facts and rules, like inserting the concept of orientation to the graph. This would need to build the facts being aware of the orientation of the edge, being able to define the adjacent relation as two nodes that have two opposite oriented edges between them.

3.2.2 8-queens problem

The 8-queens puzzle is a classical mathematical and computer science's problem. The objective is to place eight queens on a chessboard (8×8 squares) in a way that no queen is in check from any other. It implies that no two queens are in the same row, column or diagonal. One possible solution is in the Figure 10. This problem was quickly extended to an arbitrary dimension n and with in the codes presented in these section have as an argument the dimension of the chessboard (same as the number of queens to be placed), having the capacity to generate solutions to smaller and larger chessboards

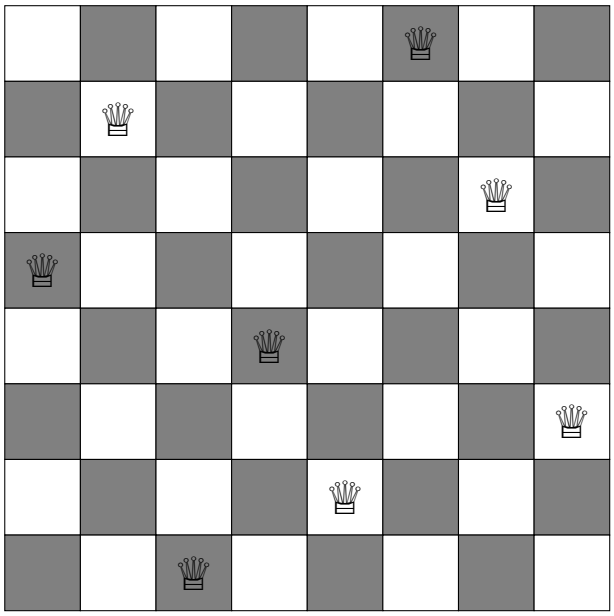


Figure 10: A solution to the eight-queens problem.

than the traditional one.

For the conventional puzzle, there is a total of 4,426,165,368 possible arrangements, as it is the result of the possible combinations of choosing 8 squares out of 64, $C(64,8) = \frac{64!}{8!56!}$. Taking into account all possible symmetries, there are 92 solutions, although only 12 of them are considered fundamental, i.e., solutions that are different to the rest even if operating with rotations and reflections of the board. Just as a curiosity, the solution in Figure 10 is the only fundamental one that has no three queens in a straight line.

This puzzle is going to be used to exemplified the differences between imperative and logical programming paradigms. Firstly, it will be explained the core of the code that was designed in Scheme to solve this problem. This served as a very good exercise to get used to manipulating numerical data from an abstract perspective and setting more basis to gaining a deeper understanding of how to handle sequences in this language. At the same time, it is very useful to compare it to the logical programming solving method, as it can be seen the differences between how the instructions have to be very precise in Scheme, having high-order procedures and a more complex code, while in Prolog (using the CLPFD library), a much easier commands are needed.

Scheme implementation

The Code 3.13 contains a reduced version from the code of the *Exercise2.42.rkt*, maintaining only the procedures that were written for solving the problem. It also uses the procedures *accumulate, enumerate-interval, filter and flatmap* given by Abelson and Sussman [1996] that appear in the script of the indicated exercise.

The solution given by the main procedure `queens` is a list of all the different solutions, for $n = 8$, it returns the 92 different combinations, being each solution a list of the positions of the 8 queens, indicating pairs with the row first and then the column. The solution displayed in Figure 10 is given by the list `((3 8) (6 7) (8 6) (2 5) (4 4) (1 3) (7 2) (5 1))`.

There is a very detailed description of all the internal processes that are done inside each of the three procedures in order to reach the desired solution that serves as a guide of the followed steps to

first understand the idea proposed in Abelson and Sussman [1996] to solve it and then to implemented. The aim here is to express all that mechanism in a more concise way so that the reader is able to comprehend the ideas behind Code 3.13.

The general strategy consists in solving the problem by placing one queen in each column of the board and checking for every new queen that it does not check any of the queens already on the board. This can be achieved in a recursive way: using the assumption that it has been already generated all the possible successful sequences for the earlier columns.

```
; Adjoins a new row-column position to a set of positions
(define (adjoin-position new-row new-col rest-positions)
  (cons (list new-row new-col) rest-positions))
; Determines for a set of positions, whether the queen in the
; kth column is safe with respect to the others
(define (safe? newcol positions)
  (define (length items)
    (if (null? items)
        0
        (+ 1 (length (cdr items)))))
  (let ((newrow (caar positions)))
    (let ((listrows (filter
                     (lambda (checked-queens)
                       (not (= newrow (car checked-queens))))
                     (cdr positions)))
          (listdiag (filter
                     (lambda (checked-queens)
                       (not (= (abs
                                (- newrow (car checked-queens)))
                               (abs
                                (- newcol (cadr checked-queens))))))
                     (cdr positions))))
      (and (= (length (cdr positions)) (length listrows))
           (= (length (cdr positions)) (length listdiag))))))
; Returns a sequence of all solutions to the problem of placing
; n queens on a n×n chessboard (n is board-size)
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map
             (lambda (new-row) (adjoin-position new-row k rest-of-queens))
             (enumerate-interval 1 board-size)
             ))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

Code 3.13: Reduce version of the 8-queen problem in Scheme.

Therefore, it will be produce an extension of each preexisting sequence by adding a potential po-

sition for the new queen. This is where the procedure `adjoin-position` is used. In the $n = 8$ case, every time this is done, the number of existing sequences will be multiplied by eight, as there are eight different positions in each column, one for each row. Looking into the internal procedure `queens-cols`, this described process is computed in the most nested part of the code, with the second `lambda` function, that has as an input `(enumerate-interval 1 board-size)`, containing the different values of the rows.

Due to how the evaluation is performed in Scheme, the instruction `(queen-cols (- k 1))` will be needed to be evaluated before applying the different functions that use it. In particular, the procedure `flatmap` has two arguments, a procedure and a list, being the second in this case the list is the one generated in the call to `(queen-cols (- k 1))`. Here it resides the recursive behaviour and it is the reason behind in the procedure `safe?` it is only needed to check the coordinates of the last added queen with respect to the rest. If the `safe?` procedure is well implemented, `(queen-cols (- k 1))` only returns list of queens' positions that are valid.

So far, it has been seen how, out of the three procedures of Code 3.13, the first one, `adjoin-position` is almost trivial and the main one, `queens`, requires a deep understanding of the developed procedures in Abelson and Sussman [1996] to manipulate structures and how the relations between are implemented in this case in the nested lines. If a reader of the book has been doing, as it has been the case, all the exercises until this point, it has already gotten familiar with the functioning of this procedures. Hence, the most challenging one to implement is `safe`, specially in an efficient way, i.e., without doing more calculations than the necessary ones, as the required tests only involve the new queen.

By construction, when it is evaluated the predicate `safe?` within `queens`, it is easy to verify that, in the sequence parameter, the position of the new queen is the first one on the sequence. Besides, as it is the main strategy, it is already known that the new queen is placed in a new column, so that is not necessary to check, only the rows and diagonals. Inside `safe?` it is defined a simple procedure to calculate the length of a list. This is done because the idea that it's going to be employed is to filter the list of the positions of the already placed queens. If any of them is checked by the queen in the new position, it will be deleted from the list. This is done for both the rows and diagonals, represented in the temporary variables `listrows` and `listdiag`. Finally, to inspect if all the already placed queens survive, the length of the lists before and after the filter are compared, meaning that, if they are equal, the configuration is a valid one!

As indicated, a much more exhaustive and accurate description of all the details involve in this strategy and implementation can be found at *Exercise2.42.rkt*, where it was started to understand the task with the $n = 4$ case and then extended the idea for an arbitrary dimension. In the last 92 lines of the file, all the solutions for the $n = 8$ are shown, which it helps to gain awareness of how the different possibilities are built.

Prolog implementation

Code 3.14 encases the solution for the N-queens problem implemented in Prolog. In the first line it is imported a library `CLPFD`, from which the predicates `between`, `all_distinct` and `length` are used. In contrast to the Scheme's version, all the necessary procedures are defined and contained within the displayed lines. As this problem has been widely studied, there are also many solutions. In the section 4.5 of Bratko [2001] there can be found other three different alternatives to the proposed one, exploring diverse ways to display the positions of the final solutions.

When reading the Code 3.14, even a non Prolog user would be able to understand the process followed to find the solutions, as the predicates are simpler and the names involved are much more intuitive. In comparison to all the intricate instructions of the Scheme version, it can be highlighted

how uncomplicated the rules established are.

The strategy followed by this implementation to solve the puzzle is to generate all the possible list of positions which are valid, forcing that all rows, columns and diagonals in both direction are different.

```
:- use_module(library(clpfd)).
; Checks if a queen's position is valid on a n×n chessboard
valid_queen((Row, Col), N) :-
 between(1, N, Col),
 between(1, N, Row).
; Ensures that all queens in the list have valid positions
valid_board([], _).
valid_board([Head|Tail], N) :-
 valid_queen(Head, N),
 valid_board(Tail, N).
; 4 predicates that extract information from a list of queen positions
; Row numbers
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
 rows(QueensTail, RowsTail).
; Column numbers
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
 cols(QueensTail, ColsTail).
; One diagonal
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
 Diagonal is Col - Row,
 diags1(QueensTail, DiagonalsTail).
; Other diagonal
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
 Diagonal is Col + Row,
 diags2(QueensTail, DiagonalsTail).
; Main predicate
queens(Board, N) :-
 length(Board, N),
 valid_board(Board, N),

 rows(Board, Rows),
 cols(Board, Cols),
 diags1(Board, Diags1),
 diags2(Board, Diags2),

 all_distinct(Rows),
 all_distinct(Cols),
 all_distinct(Diags1),
 all_distinct(Diags2).
```

Code 3.14: Reduce version of the 8-queen problem in Prolog.

In order to do it, first it ensures, using the predicates, `valid_queen` and `valid_board`, that the

positions of the queens are valid with respect to the chessboard's dimension. In the second one it is used recursion, as it checks one by one the validity of each queen. The next four predicates (`rows`, `cols`, `diags1` and `diags2`) have the aim of obtaining the information of the positions of a list of queens, considering four different coordinates: rows, columns, diagonals in one direction and diagonals in the other direction. This is done with two rules, in a similar way to the *append* predicate from 1.1. One for the empty case and a recursive one for a non empty list.

Finally, in the main predicate is `queens`, the solutions are generated and stored in the `Board` variable. In the last four lines, it can be seen how the strategy is assembled. The query for getting the solutions for the $n = 8$ case would be `?- queens(Board,8).`, receiving as an output the 92 possibilities too, with the form `Board = [(1, 5), (2, 7), (3, 1), (4, 4), (5, 2), (6, 8), (7, 6), (8, 3)]`, which express the solution displayed in Figure 10.

Implementation attempt in Scheme's logical programming interpreter

The limitations of the interpreter and the potential will be covered in more detail in Chapter 5, although it is relevant to comment here some of them as it was done an effort to adapt the Code 3.14 to a Scheme syntax as done in previous example to obtain valid solutions. This topic was first addressed in 1.1.6 when talking about the *gcd* predicate and it will be fully explained when analysing the interpreter and the constructed database in Chapter 4.

In Code 3.15 it is displayed how it was tried to write the two first predicates from Code 3.14. In `valid-queen`, as the library `CLPFD` is from Prolog, instead of using `between`, it was tried to generate the same conditions by means of the `lisp-value` form. This was not the problem, but the use of primitive procedures *car* and *cdr*. When implementing the logical programming interpreter it was lost the ability of dealing with primitive procedures and list structures. However, it was still useful as an exercise the attempt to implement it, having in total three different perspectives of the same puzzle: from a functional paradigm in Scheme, from a logical paradigm in Prolog and from a logical paradigm in Scheme.

```
(rule (valid-queen ?queen ?n)
      (and (and (lisp-value <= 1 (cadr ?queen))
                (lisp-value >= ?n (cadr ?queen)))
           (and (lisp-value <= 1 (car ?queen))
                (lisp-value >= ?n (car ?queen)))))

(rule (valid-board ?board ?n)
      (or (null? ?board)
          (and (valid-queen (car ?board) n)
               (valid-board (cdr ?board) n))))
```

Code 3.15: Attempt to implement the first two predicates from Code 3.14 in a Scheme syntax.

3.3 Message passing and Data-Directed: The data as a program and the use of quote.

Many of the examples showed in this bachelor's thesis and many of the exercises that can be found in the GitHub repository present just a few procedures. However, if added all the related procedures of some of them, it can be constructed a system. For a correct understanding and use of a programming system, it is needed to develop some tools in order to navigate through it and manage its different component easily. Two different techniques for organising and managing the code can be found in

Scheme: data-directed programming and message passing.

Data-directed programming is the one that it has been used to implement the logical programming interpreter and it needs the construction of a two dimensional table. This technique relies on modularity, as the idea is to use a pertinent procedure depending on the type of data. When a procedure has to be applied, the idea is to look for the appropriate one depending on the type of the data object that will be the argument, as there will be more than one procedure for doing the same operation. This is very useful to handle data structures that can have more than one representation. Depending on the operation, it can be easier to use one representation or another.

For instance, when manipulating complex numbers, rectangular form is more suited to add or subtracting them, while polar form is more adequate to multiply or dividing, although all four operations can be implemented in both forms. To identify the procedures with the type they are able to handle, they are *tagged*, using the quote operator `'`. Using that, whole packages can be defined for each representation. The packages usually consist in a set of definitions of procedures together with inserting the procedures to the table with the procedure `put` and the correct keys.

Message-passing is a way to manage the code that can remind to the philosophy in object-oriented programming. Here, the focus is on individual objects or identities, where the procedures are defined within them, recalling the way methods are defined in object-oriented programming. For achieving this, it is usually defined a procedure named `dispatch`, which accepts one argument, containing a tag that triggers one of the procedures defined within the object. To see more clearly how this works, it is going to be used already explained procedures, the constructor `cons` and the selectors `car` and `cdr`. Code 3.16 reflects a slight variation of the way this primitive procedures are defined in Scheme is by means of message-passing. In reality, instead of using the predicates `(eq? m 'car)` and `(eq? m 'cdr)`, these are replaced by `(= m 0)` and `(= m 1)`. However, for showing a more general yet basic use of message-passing, it has been adapted the `cons` definition.

```
; Constructor cons is defined
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Argument not 0 or 1 -- CONS" m))))
  dispatch)
; If it is constructed an object z with cons
(define z (cons 1 2))
; Now, selectors are defined
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(car z) ; Returns 1
(cdr z) ; Returns 2
```

Code 3.16: Definition of `cons` procedure using message-passing.

Detailing the implementation of Code 3.16, when defining the object `z` as `(cons 1 2)`, `z` is now a procedure, because in the definition of `(cons x y)`, the output is the function `dispatch`. Then, when specifying how `car` and `cdr` work, this is done by passing to `z` the corresponding argument for each selector.

4 Interpretation of Languages in Functional Programming

If a programmer starts to dissect the programming language that he/she is using, it can end up leading to question how the language itself processes and evaluates the expressions that are being created. Thus, the programmer would finally check that almost every programming language has an *evaluator* or *interpreter* written in its own language that has the capacity of performing all the required actions when evaluating an expression of the same language.

The reason behind this strategy is to achieve, among others, two mean objectives:

1. Developing the programming language. This is accomplished by extending the functionalities present in the interpreter. So, if the computer programmer wants the language he/she is designing to incorporate a wider range of capabilities, the interpreter is first modified, allowing new expressions to be evaluated and expanding the language.

2. Proving the usefulness of the programming language. Usually, when presenting a new programming language to the computer science community, it should be done by exhibiting a big project. An interpreter is an example of such a project.

In the following section, it will first be seen some details of how it is built the interpreter for imperative programming in Scheme, focusing on how different expressions are handled and the core of the evaluator. Finally, it will be explained the elements of the logical programming interpreter and a description of the database implemented, as well as a guide of how to use it.

4.1 The core of the evaluator

The interpreter built in [Abelson and Sussman, 1996, section 4.1] receives the name of *Metacircular evaluator*, as it is written in the same language that it evaluates. To understand the method for constructing this interpreter is very useful to gain a deep knowledge about how the environment diagrams, such as the ones from Figure 5 and Figure 6, are designed. In the environment model, the evaluation of a combination is done following the same rules as the ones explained in Section 2.2. Now, there are another two rules, that refer to the application of procedures:

1. A procedure is created by evaluating a `lambda` expression relative to a given environment. Then, it is generated an object related to the procedure that is a pair of two elements: the body of the `lambda` function and a pointer directed to the environment in where the procedure was defined.

2. When applying the procedure to a set of arguments, it is constructed a frame, creating a binding between the parameters and the arguments. Then, the body is evaluated in the context of the new environment.

In conclusion, the evaluation process of an expression is a cycle where the input expression is firstly reduced to a procedure with a given set of arguments, which are also reduced to other expressions in different environments until it is reached a level of reduction where everything left are symbols or

primitive procedures. In *Exercise3.10.pdf* there is a good example of a more detailed process involving different environments that evolve with different calls.

The mentioned cycle is described in [Abelson and Sussman, 1996, section 4.1.1] with the procedures `eval` and `apply`. Focusing on the first one, when receiving an expression and an environment, the `eval` procedure first classifies it depending on the type. For doing this, there are several predicates such as `if?` or `lambda?`. The idea behind this implementation is similar to the one explained when talking about data-directed style in the sense that each function has a label or tag identifying it. Indeed, this tag is the name of the procedure. The key function for developing this method is `tagged-list?`, presented in Code 4.1. Its purpose is to read the label of each procedure.

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

Code 4.1: Definition of the procedure `tagged-list?`.

Then, the predicates such as `if?` or `lambda?` are defined simply by using `tagged-list?`. Apart from this, it is necessary to process each part of the expression given to the evaluator. Later, for each component of a special form or procedure represented in the evaluator, it is constructed a selector. The idea is that every expression can be handled as a list. For example, if it is evaluated the following expression `(lambda (x) (x+1))`, if thought as a list, its `car` is `'lambda`. Developing this idea, every procedure can be divided in several parts. For both predicates `if?` and `lambda?`, this implementation is captured in Code 4.2.

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
```

Code 4.2: Syntax specification for procedures `lambda?` and `if?`.

One interesting aspect here is that there are less contemplated options than an user could expect, since some special forms that have been used until now such as `cond` or `let` do not appear explicitly. These are called *derived expressions* because they can be defined in terms of other special forms. In the case of `cond`, in [Abelson and Sussman, 1996, section 4.1.2] it is defined an auxiliary procedure to convert it into an `if` expression.

It will be seen now with more detail how the evaluation of a `let` expression is transformed in a way that it can be interpreted as a `lambda` function by the implemented metacircular evaluator. The general form of a `let` expression is represented in 9, but the idea is to generate the same bindings by means of a `lambda` expression, as reflected in 10. Then, if having to evaluate an expression with the

structure represented in 9, the first step is to identify the components of it and determine which list selector is needed for each case. The idea is to generate an analogous decomposition as in Code 4.2 and end it with a procedure that uses those parts and assembles them in a different way to build a `lambda` function with the structure displayed in 10, this procedure is called `let->combination`. The necessary code for extracting the pieces of information needed to process a special form `let` is presented in Code 4.3, taken from *Exercise4.6.rkt*.

```
(let ((⟨var1⟩ ⟨exp1⟩)
      (⟨var2⟩ ⟨exp2⟩)
      .
      .                    (9)
      .
      (⟨varn⟩ ⟨expn⟩))
    ⟨body⟩)
```

```
((lambda (⟨var1⟩ ... ⟨varn⟩)
    ⟨body⟩)
 ⟨exp1⟩
 .                         (10)
 .
 .
⟨expn⟩)
```

```
; First procedure to identify the special form let
(define (let? exp) (tagged-list? exp 'let))
; Selectors
(define (let-body exp) (cddr exp))
(define (let-varexps exp) (cadr exp))
(define (let-variables exp) (map car (let-varexps exp)))
(define (let-expressions exp) (map cadr (let-varexps exp)))
; Final procedure to construct a lambda function
(define (let->combination exp)
  (cons (make-lambda (let-variables exp)
                     (let-body exp))
        (let-expressions exp)))
; Auxiliar procedure
(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

Code 4.3: Syntax specification for procedure `let`.

After this complex classification inside `eval`, depending on the type, it is done the evaluation and, when needed, it is done a call to the `apply` function, that has as arguments a procedure and a set of arguments. Along the section 4.1 of Abelson and Sussman [1996] there is a profound explanation of how all the evaluator is built, and the complemented exercises to fully understand it are located in the folder *SICP. Chapter 4* of the GitHub repository.

Lastly, when building an evaluator, even though it is not necessary to address the halting problem, being able to understand it allows the user to learn more about the theoretical limitations of any computing system. Hence, it will next be briefly covered.

4.1.1 The Halting Problem

The first formulation and proof of the halting problem is attributed to Alan Turing, based on a paper that was published in 1936, although its name appears to come from a 1958 book written by Martin David [Lucas, 2021]. The halting problem is a decision problem in computer science and it is based on whether an algorithm can determine if, given an arbitrary program and an input, the program will terminate (or *halt*) or if it will keep running forever. The halting problem has been proven to be

undecidable, this means that it doesn't exists such algorithm.

To show that the halting problem is not solvable, it is constructed a proof by contradiction, as it is motivated in *Exercise4.15.rkt*. The code used in Scheme to construct this proof is displayed in Code 4.4. There, it is defined first a procedure under the assumption that there is already a predicate `halts?` that correctly determines if a one-argument procedure terminates when given an input. In the comments of Code 4.4 it is explained how the contradiction is reached for any possible outcome, proving how the assumed predicate `halts?` cannot exist.

```
; Definition of procedure named try to test the predicate halts?
(define (try p)
  (if (halts? p p)
      (run-forever)
      'halted))

; It is evaluated the following expression
(try try)

; Using the substitution model it is analysed every possible outcome
(if (halts? try try)
    ; if (halts? try try) is true,
    ; -> by definition of halts?, (try try) halts, i.e., terminates
    (run-forever)
    ; (run-forever) is triggered by definition of try -> contradiction

    ; if (halts? try try) is false,
    ; -> by definition of halts?, (try try) runs forever
    'halted)
    ; 'halted is triggered by definition of try -> contradiction
```

Code 4.4: Proof by contradiction of the undecidability of the halting problem.

4.2 Logical programming interpreter

The only script that accompanies this document contains the implementation of the logical programming interpreter, together with a complete database in where it can be found the examples showed along the chapters and a set of exercises from [Abelson and Sussman, 1996, section 4.4]. The same script can also be found in the GitHub repository, at *LogicProgramming_Interpreter.rkt*.

4.2.1 Details of the implementation

Three different parts can be distinguished within the presented script. Firstly, it was needed to use old procedures that had been previously studied. As explained in different sections of the document, several of the analysed procedures and data objects are used in the implementation of the interpreter. The most relevant ones are the procedures to make tables and then a set of shorter procedures to incorporate all the tools related to streams.

Secondly, starting with the `query-driver-loop`, it begins the proper implementation of the logical programming interpreter. For that, it was followed [Abelson and Sussman, 1996, section 4.4.4]. This first procedure is the one in charge of reading the input queries. It also allows the user to add new rules and assertions to the previously set database for quick examples and exercises.

The evaluator in this interpreter is formed by the `qeval` procedure, where it is used data-directed dispatch with the procedures `get` and `put` and it differentiates between special form queries, the ones that use the logical operators AND, OR and NOT and `lisp-value`, and simple queries, having all their related procedures defined afterwards.

Pattern matching and *unification*

The following functions of the interpreter implement all the necessary procedures to put into action the operations of *pattern matching* and *unification*. The former one is the mechanism that it is needed for simple and compound queries. When a simple query is given as input, in order to process it and determine if it is true or not, all database facts (or assertions) present are scanned to select the ones that match the input. As this is very costly, it is usually performed an indexing of the database, which is also implemented in the interpreter. If the match is produced, it is created a binding between the input and the information in the database, if needed to match again, for example in a compound query that uses the logical operator AND, those bindings can not be broken, if so, the match requirements are not fulfilled. All the process of creating frames and bindings is organised by the use of streams.

In order to handle the rules, it is needed to generalise the operations of *pattern matching* and use the concept of *unification*. This is the most technical part of the implementation and it can be similar to solve a system of equations, as deduction is required. An example will be shown step-by-step to complete the explanation. It is attempted to unify `(?x ?x)` and `((a ?y c) (a b ?z))`. The *unification* process should be able to make the following bindings: `?x` with `(a b c)`, `?y` with `b` and `?z` with `c`. This could also serve as an example of *pattern matching*, as all the variables become bound. However, the way *unification* generalises *pattern matching* is that it allows undetermined variable values. If it is tried to unify `(?x a)` and `((b ?y) ?z)`, the furthest bindings that can be made are `?x` with `(b ?y)` and `?z` with `a`. Then, the value of `?y` is not restricted, but the value of `?x` is, when `?y` is given a value, then the value of `?x` will be known.

Finally, it will be analysed how these two operations are used with some examples of the constructed database. In the first part of the database, there is general information about the workers of a company, including address, position, salary or supervisors. For each worker, a set of facts are used to describe this information, for example, for Louis Reasoner, his personal information is displayed in Code 4.5. If the input was a simple query such as `(job (Reasoner Louis) ?pos)`, *pattern matching* would be applied, and it would be tried to match the query with all the facts that have the same pattern. In the implemented database, as said, there are several people and, for each of them, it is defined its position, by an assertion with the pattern `(job (lastname name) (position))`, where `(position)` can be formed by one or more words. Seeing the given simple query, is the database is well implemented and assuming that Louis Reasoner occupies only one position in the company, a match is found, and there would be an output query giving the information of the position of Louis Reasoner.

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))
```

Code 4.5: Example of a set of facts of the used database.

In order to determine if it is worth sharing a car to go to the office together, it can be implemented a rule to know if two people live near enough, stating that they do if they share the same town. The rule `lives-near?` can be found at Code 4.6, where the procedure `(same ?x ?x)` is the same one from Example 1.1.3. So, in this query, the employee Louis Reasoner is interested in knowing who lives near him and gives the following query to the interpreter `(lives-near ?person (Reasoner Louis))`.

```
(rule (lives-near ?person-1 ?person-2)
          (and (address ?person-1 (?town . ?rest-1))
               (address ?person-2 (?town . ?rest-2))
               (not (same ?person-1 ?person-2))))
```

Code 4.6: Example of a rule of the used database.

As there are not any assertions for this relation, the *pattern matching* operation will fail and *unification* is necessary, as it will be worked with a rule. The conclusion of the rule from Code 4.6 has a match with the given query, binding `?person` with `?person-1` (still undetermined) and `Reasoner Louis` with `?person-2`. Then, it is passed to the conditions required to achieve the conclusion of the rule. If a value is found for `?person-1`, then the variable `?person` is also determined.

Indexing the database

After implementing all the procedures for the described operations of *pattern matching* and *unification*, there is a set of functions with the aim of providing a more efficient way of searching matches. If it is constructed a database with information as the one displayed in Code 4.5 for a big company, there will be so many employees, resulting in an immense amount of facts. If a person is interested in finding out who is the supervisor of an employee, it is convenient to avoid *pattern matching* in all the pieces of information that are irrelevant in this case.

The idea is to separate the assertions that give the same type of information in different streams. Each stream would be characterised by the first word of each fact, and then, when doing *pattern matching*, it will be first checked the key that characterises the input query with the one of each stream. For rules it is done a similar indexation, but the key is given by the name of the conclusion. In the case of the rule from Code 4.6, it would be `lives-near`.

Final procedures

After implementing the indexing in both facts and rules, there are still some procedures left, a few of them are for adding more streams operations, others are related to the syntax of the queries, involving predicate procedures similar to the ones explained in the Section 4.1. Then, some simple procedures are defined for working with bindings and frames, which is a list of bindings.

Initialising the database

Once all the procedures to implement the interpreter are constructed, in order to be able to work with a given database containing the facts and rules to be applied, it is needed a procedure that will be used together with `query-driver-loop` to start the interpreter. This procedure has been named `init-data-base` and it has one argument that is the name of the database to be initialised. Its code it is represented in Code 4.7, containing many commented lines in order to understand better the internal processes of the procedure.

This procedure helps to capture the idea of how all the facts (assertions) and rules from a created database are incorporated into all the procedures that compose the interpreter. It goes entry by entry of the database and first it uses the procedure `query-syntax-process` for passing the query from a more familiar syntax for the user to an easier syntax for the evaluator. Then, it is performed a classification between rules and assertions and stored the index of each of them for the efficient *pattern matching*. Finally, using the procedures `put` and `get`, the table is constructed to use the data-directed dispatch.

```
(define (init-data-base rules-and-assertions)
  (define (deal-out r-and-a rules assertions)
    (cond ((null? r-and-a) ; If the data-base has no rules or assertions
           (set! THE-ASSERTIONS assertions)
           (set! THE-RULES rules)
           'done)
          (else ; This goes one by one
           (let ((s (query-syntax-process (car r-and-a))))
             ; query-syntax-process transform pattern variables in the
             ; expression to the internal format
             ; For example, a query such as (job ?x ?y) is actually
             ; represented internally by the system as (job (? x) (? y))
             (cond ((rule? s)
                    ; Once is done it, it breaks the process in two
                    ; types: rules and assertions
                    (store-rule-in-index s)
                    ; It stores the rule
                    (deal-out (cdr r-and-a) ; It continues with the rest
                              (cons s rules) ; It updates the rules
                              assertions)) ; It doesn't update assertions
                   (else ; If it is not a rule, it is an assertion
                    (store-assertion-in-index s)
                    ; It stores the assertion
                    (deal-out (cdr r-and-a) ; It continues with the rest
                              rules ; It does not update rules
                              ; It updates assertions
                              (cons s assertions)))))))))
  ; The different options for the qeval procedure are added here
  ; qeval procedure identifies special forms by a data-directed dispatch
  ; using get and put
  ; Any query that is not identified as a special form is assumed to be a
  ; simple query and it is processed by simple-query
  (let ((operation-table (make-table)))
    (set! get (operation-table 'lookup-proc))
    (set! put (operation-table 'insert-proc!)))
  (put 'and 'qeval conjoin)
  (put 'or 'qeval disjoin)
  (put 'not 'qeval negate)
  (put 'lisp-value 'qeval lisp-value)
  (put 'always-true 'qeval always-true)
  ; Final call to finish the initialization
  (deal-out rules-and-assertions '() '()))
```

Code 4.7: Initialisation of the database.

4.2.2 User guide

This final section is intended to be a short guide for the given logical programming interpreter. The recommended software for running the script *LogicProgramming_Interpreter.rkt* is *DrRacket*, very suitable for Lisp and Scheme languages and it has been the one used during all the bachelor's thesis, all the files written in Scheme have the extension *.rkt*.

When run, it will appear in the console the message ;;; Query input: followed by a textbox where it has to be written the wanted query. For this, there are several options:

1. Simple query. It can be understood as a simple question it is asked in order to check if it is true or not. It can also be viewed as a way to test which objects of the database fulfil a condition or to gain more information that can be deducted from all the facts and rules present in the database. An instance can be the one explained in the example 1.1.3, `(mortal Socrates)`.

2. Compound query. Using the logical operators AND, OR and NOT and the special form `lisp-value`, more complex questions can be formulated. An example, related to the constructed database, can be the one from Code 4.8, where it is wanted to find all people whose salary is less than Ben Bitididdle's, together with their salary and Ben Bitididdle's salary.

```
(and (salary (Bitdiddle Ben) ?number)
             (salary ?person ?amount)
             (lisp-value < ?amount ?number))
```

Code 4.8: Example of a compound query.

3. Add a fact to the database. Using the procedure `assert!` it can be added a new fact to the database. This fact is a temporary one, as when the script is run again, the fact is no longer present. This can be useful to have a more interactive experience with the interpreter and add forgotten data, although altering the database can be achieved easily. It is important to point that if it is tried to change an already present fact, it will not work. Looking at the Code 4.5, if it is wanted to change the salary of Louis Reasoner from $30,000 to $40,000, it would not be useful to write `(assert! (salary (Reasoner Louis) 40000))`. If done so, there would be two different assertions in the database and it would be considered that Louis Reasoner has two different salaries, the original fact would not be altered. After using `assert!`, a message will be displayed informing that the fact has been successfully added to the database and it will allow the user to introduce an input again.

4. Add a rule to the database. The procedure `assert!` can also be applied to introduce whole new rules to the interpreter.

The name of the built database is `logic-programming-data-base` and inside it it can be found multiple of the facts and rules that appear in [Abelson and Sussman, 1996, section 4.4] together with other facts or rules that were added as a consequence of solving some exercises from that same section. Multiple comments accompany the exercises in order to give more precise instructions or details of how those rules work. Besides, the example from Chapter 1 are included. Any reader is encouraged to modify the database and extend it to make it richer, the syntax for adding more facts or rules, directly in the console or in the database, is simple.

After introducing a query in the interpreter, if no error message is triggered, it will appear the message ;;; Query results: with the fact or set of facts deducted from the inquiring input. If there is more than one answer, there is no need to introduce any command, all of them will be displayed in the console.

5 Conclusions and Future Work

In this bachelor's thesis, a logic programming interpreter has been successfully implemented in the Scheme programming language. In order to achieve this objective, it has been deeply studied the basis of logical programming, including its first language, PROLOG, which has served as a constant complement to better understand how to approach problem resolution from a logical perspective, specially when analysing the SLD resolution for Horn clauses, the main feature of the implemented interpreter.

With the help of many examples and constant references to the solved exercises from the book Abelson and Sussman [1996], it has been showed all the fundamental concepts needed to gain a strong understanding of the basis of logical programming, revised a theoretical computation model as λ-calculus, which has a great impact on the development of Lisp (therefore Scheme) language and all the imperative and functional programming tools that are necessary for the proposed interpreted.

However, at the same time, the limitations of the interpreted are acknowledged, as it could be verified in some examples. The given implementation is based on the one explained in [Abelson and Sussman, 1996, section 4.4] and it is a very adequate benchmark. In other words, all the presented exercises, examples and scripts can serve as a starting point for future work and more profound analyses of the logical programming paradigm.

There are some proposed directions in where a future student or user can progress in order to generate a more complete interpreter.

- Incorporating libraries. As it could be seen when presenting the 8-queens problem, it was not possible to solve the puzzle using the designed interpreter. Examining PROLOG's version, if it could be found a way of developing some libraries for the interpreter that could be imported in order to be able to use such procedures within the interpreter implementation, it could be a huge step. In [Abelson and Sussman, 1996, section 2.4.3] it is described a way to construct packages for the different forms of expressing complex numbers that could be worth to replicate in some way.
- Primitive procedures in the interpreter. There is a considerable set of primitive procedures in Scheme, such as `remainder`. However, as it was explained in Section 1.1.6, the interpreter is not able to recognise such primitive procedures. In fact, the message error that appears when trying to add the rules from Code 1.11 is that the interpreter is unable to process the subexpression `(remainder ?a ?b)` because it is expecting a value, not a subexpression, even when `?a` and `?b` are substituted by numbers. Revising the rules for evaluation, one expected the evaluator to behave in a way that the subexpression is evaluated beforehand, but apparently this is not the case. Therefore, it is recommended a thoroughly examination of the order of evaluation in the interpreter in order to understand if the problem is located there or if it arises from a different part.

Anyway, although it is conceded that the presented interpreted has limitations, it can also be affirmed that it is good enough to explore the equivalences between programming paradigms and comprehend the ideas behind the thesis of Church-Turing.

Another interesting perspective that this bachelor's thesis provides for a mathematics or physics student is the experience of working with logical programming. Usually, the languages that have been introduced to such student during both of these bachelors are placed within imperative or object-oriented programming. This is very useful, as the student is able to learn the tools that most often will use in a future job or research. However, most of them share a common perspective and offer the user a same approach to problem solving.

On the other hand, logical programming allows the student to develop a completely different method and way of thinking when trying to find a solution for a given problem. Going beyond, in the same sense that the Sapir-Whorf hypothesis proposes that the structure of a (linguistic) language affects the speakers' cognition and perception of reality, it could be defended the same argument when talking about programming languages, since the same problem requires absolutely different strategies when using different programming paradigms.

Bibliography

Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. 1996.

Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. Fifty years of prolog and beyond, 2022.

Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16 (1):14–21, 1951. doi: 10.2307/2268661.

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified horn clauses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 869–882, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Robert Kowalski. Predicate logic as programming language. *IFIP Congr.*, 74:569–574, 01 1974.

Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, jul 1982. ISSN 0004-5411. doi: 10.1145/322326.322339.

Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Amsterdam, 2004. ISBN 978-1-55860-932-7.

Ivan Bratko. *PROLOG. Programming for Artificial Intelligence.* Pearson Education, 3rd editon edition, 2001.

Harold Abelson and Gerald Jay with Julie Sussman Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996.

T.X.N. *CLIPS Reference Manual, Version 6.0*. NASA-Lyndon B. Johnson Space Center, 1993.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, apr 1960. ISSN 0001-0782.

Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941.

Salvador Lucas. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming*, 121:100687, 2021. ISSN 2352-2208.