

**LAUREA MAGISTRALE IN INFORMATICA**

**PROJECT FOR ARTIFICIAL INTELLIGENCE USING PROLOG CALLED AS**

**THE HEALTH BUDDY**

***Helping people to find a doctor for their disease based on their symptoms.***

*BASKARAN SHANMUGAPRIYA*

[*s.basakran@studenti.uniba.it*](mailto:s.basakran@studenti.uniba.it)

*725189*

*DASS ALEX PRAVEEN*

[*a.dass@studenti.uniba.it*](mailto:a.dass@studenti.uniba.it)

*725190*

# TABLE OF CONTENTS

[1. Introduction 3](#_Toc136887509)

[1.1 Project Summary 4](#_Toc136887510)

[2. Requirements 4](#_Toc136887511)

[2.1 Goals 5](#_Toc136887512)

[3. Conceptualization 6](#_Toc136887513)

[3.1 Entities and Relationships 6](#_Toc136887514)

[3.1.1 Entities: 6](#_Toc136887515)

[3.1.2 Relationships: 7](#_Toc136887516)

[3.2 Functions 7](#_Toc136887517)

[4. Inference 9](#_Toc136887518)

[4.1 Rules and its Structure 9](#_Toc136887519)

[4.2 Backward 9](#_Toc136887520)

[4.3 Forward 10](#_Toc136887521)

[5. Knowledge Base 11](#_Toc136887522)

[6. Interface Implementation 14](#_Toc136887523)

[6.1 XPCE Interface : 16](#_Toc136887524)

[6.2 Terminal Interface : 18](#_Toc136887525)

[7. Conclusions and Future Developments 18](#_Toc136887526)

# Introduction

This project has been named **Health Buddy**, an advanced medical decision support system! It is based on a **knowledge-based decision support system** in Prolog. It aims to provide an intelligent and efficient solution for

* Diagnosing Diseases based on the input of various symptoms.
* Suggesting Specialized Doctors
* Recommending appropriate Treatments and Medications.

that can also be used for the therapies at home. By leveraging the power of Prolog, this system has been developed with a robust knowledge base that encompasses a wide range of medical conditions, symptoms, doctors' expertise, and treatment options.

This decision support system operates on a simple principle: input the symptoms, and it will analyze the information using its vast repository of medical knowledge, by carefully matching our symptoms with known disease patterns.

The idea of creating this project is to provide every individual with the right set of healthcare information even in the remote places where the health care isn't accessible.

**Example consider this scenario**: A small clinic in the mountain region of a developing country. Where there is only one doctor and a few nurses, and the doctor is not available. By consulting with the software, the Health Buddy software, a nurse can suggest a medication for a non-life threating disease for a patient or consult with the relevant volunteer doctor by calling them over the phone as health buddy not only a doctor's name but also his phone number if available and his clinical address.

## Project Summary

We have decoupled the core logic from both the data and the interface. In this project by organizing the project into different modules and defining clear interfaces between them, the project achieves a level of decoupling.

This modular structure allows for easier maintenance, extensibility, and testability of the system. Changes made to one module are less likely to affect other modules, promoting a more flexible and robust codebase.

In cases where the interface needs to interact with the core logic, we have decided to use hooks, to keep it as decoupled as possible.

To achieve that, it’s easier to create the logic first and evaluate it using the SWI-Prolog development environment, and only then create an interface that demonstrate that logic.

We have designed the program in a way that it can be extended, with new hooks, or a more complex logic, without having to majorly redesign the other parts.

Implementing recursion in the forward and backward inference would not require reworking the interface. Implementing new interfaces can be done easily using hooks.

Here we have the advantage of Prolog’s pattern matching clauses for clearer syntax, in this project we have used it to simplify writing main.pl and the hook modules.

# Requirements

Here are the requirements for Health Buddy project:

* **Symptoms and Disease Datasets:** We have acquired comprehensive and up-to-date Datasets of symptoms associated with various diseases, ensuring accuracy and relevance.
* **Doctor Datasets:** We have acquired Datasets of qualified doctors, including their specialties, address, and contact information.
* **Medication Datasets:** We created a Datasets of medications, including dosage information, side effects, and potential drug interactions for the list of diseases that are available in our system.
* **Treatment Datasets:** The software not only provides the medication but also explains the probable treatments that can be taken to resolve the issue.
* **User Interface**: Develop an intuitive and user-friendly interface that allows users to input their symptoms and interact with the system effortlessly.
* **Error Handling and Exception Reporting**: Implement error handling mechanisms to manage unexpected scenarios and provide meaningful error messages to users.

All the data sets have been extracted as raw data from the Kaggle Website. <https://www.kaggle.com/datasets> .

## 2.1 Goals

The goal of the given code is to define a set of rules that associate various diseases with their corresponding categories and symptoms.

By querying the knowledge base with specific symptoms, we can determine,

* **Introduction:** First out of these 150+ diseases that are available we need to arrange the predicates in such a way that the user doesn’t need to check/answer for each disease that is available for every disease. So, we must categorize the available disease as follows,
* **Categories of Diseases:** The respective categories are of,
  + - Infectious
    - STIs
    - Mental health disorders
    - Blood disorders
    - Tumors
    - Endocrine disorders
    - Digestive system disorders
    - Cardiovascular diseases
    - Neurological disorders
    - Respiratory disorders
    - Renal disorders
    - Metabolic disorders.
* **Symptom matching**: Once we have determined the category then the system should start matching the symptoms with the diseases to find out the possible disease using the system knowledge base inference.
* **Doctor:** The next goal after finding the disease is to find the correct doctor that matches the specialization of the disease’s description.
* **Address and Contact Details:** As we are using a dataset externally that is in a .csv format once the disease is determined the system has to match the disease from the .csv file to bring us the appropriate doctor, specialization and the treatments and medicine.

# Conceptualization

In this section we formalize the knowledge about degree courses, identifying the entities

of interest in making the system work properly and satisfying the requirements.

## 3.1 Entities and Relationships

In our project, there are many entities and relationships that can be identified. Here's an overview of the entities and their relationships:

### 3.1.1 Entities:

1. **Symptoms**: Represents the various symptoms associated with diseases. They are extracted from the knowledge base.

2. **Diseases**: Represents different diseases. They are identified based on the symptoms and their associations in the knowledge base.

3. **Treatments**: Represents the treatments available for different diseases. Treatments are associated with specific diseases and include information such as descriptions, medications, doctors, addresses, and specializations.

4. **Categories**: Represents the categories or types of diseases. Diseases are classified into distinct categories.

5. **Diagnosis**: Represents the input diagnosis, which includes the symptoms reported by a patient.

### 3.1.2 Relationships:

1. **Symptoms -> Diseases**: Symptoms are associated with diseases in the knowledge base. The presence or absence of specific symptoms helps identify potential diseases.

2. **Diseases -> Treatments**: Diseases have associated treatments. Each disease is linked to one or more treatment options.

3. **Categories -> Diseases:** Diseases are classified into various categories. Each disease is assigned to a specific category.

4**. Diagnosis -> Symptoms**: The input diagnosis consists of reported symptoms. The symptoms reported by the patient are used to determine potential diseases.

5**. Categories -> Diagnosis**: The chosen category is associated with the input diagnosis. The category helps narrow down the potential diseases based on the reported symptoms.

6**. Diagnosis -> Hypotheses**: The input diagnosis and knowledge base are used to generate hypotheses about potential diseases.

7. **Hypotheses -> Treatments**: The generated hypotheses are matched with the available treatments. The treatments corresponding to the identified diseases are presented as potential treatment options.

These entities and relationships form the basis of the diagnostic system, allowing the program to process the input diagnosis, generate hypotheses, match them with treatments, and present the results to the user.

## 3.2 Functions

* + **`read\_treatments/2`: `read\_treatments(TreatmentsFile, Treatments)`**
* `**TreatmentsFile**`: The file name or path from which to read the treatments data.
* `**Treatments**`: The variable that will be unified with the list of treatments read from the file.
  + **`read\_diagnosis/2`: `read\_diagnosis(DiagnosisFile, Diagnosis)`**
* `**DiagnosisFile**`: The file name or path from which to read the diagnosis data.
* `**Diagnosis**`: The variable that will be unified with the list of diagnosis data read from the file.
  + **`get\_knowledge/3`: `get\_knowledge(Diagnosis, Condition, Knowledge)`**
* `**Diagnosis**`: The list of diagnosis data.
* `**Condition**`: The condition or criteria to match for retrieving knowledge.
* `**Knowledge**`: The variable that will be unified with the knowledge data matching the condition.
  + **`generate\_knowledge/3`: `generate\_knowledge(HookModule, RequiredKnowledge, Knowledge)`**
* `**HookModule**`: The module containing the hook predicate for generating knowledge.
* `**RequiredKnowledge**`: The required knowledge data.
* `**Knowledge**`: The variable that will be unified with the generated knowledge based on the hook module.
  + **`forward/4`: `forward(Knowledge, Diagnosis, Condition, Hypotheses)`**
* `**Knowledge**`: The knowledge data used for forward inference.
* `**Diagnosis**`: The list of diagnosis data.
* `**Condition**`: The condition or criteria for the forward inference.
* `**Hypotheses**`: The variable that will be unified with the hypotheses generated by the forward inference.
  + **`backward/3`: `backward(HookModule, Diagnosis, Hypothesis)`**
* `**HookModule**`: The module containing the hook predicate for backward chaining.
* `**Diagnosis**`: The list of diagnosis data.
* `**Hypothesis**`: The variable that will be unified with the generated hypothesis by backward chaining.
  + **`write\_treatment/2`: `write\_treatment(Hypothesis, Treatments)`**
* `**Hypothesis**`: The specific hypothesis or disease for which the treatment information is required.
* `**Treatments**`: The list of treatment data.
  + **`choose\_category/2`: `choose\_category(Diagnosis, Category)`**
* `**Diagnosis**`: The list of diagnosis data.
* `**Category**`: The variable that will be unified with the chosen disease category.
  + **`change\_category/2`: `change\_category(Window, Category)`**
* `**Window**`: The graphical window object.
* `**Category**`: The selected disease category.
  + **`update\_diseases/3`: `update\_diseases(Window, Category, SymptomsChain)`**
* `**Window**`: The graphical window object.
* `**Category**`: The selected disease category.
* `**SymptomsChain**`: The selected symptoms for the disease category.
  + **`show\_treatment/2`: `show\_treatment(Window, Disease)`**
* `**Window**`: The graphical window object.
* `**Disease**`: The selected disease for which the treatment information is to be displayed.
  + **`clear\_treatment/1`: `clear\_treatment(Window)`**
* `**Window**`: The graphical window object.

# Inference

To incorporate backward and forward chaining, we opted to implement the domain knowledge using Prolog’s inference. This allowed us to construct an inferential engine that leverages Prolog's capabilities.

## 4.1 Rules and its Structure

The new rule structure has been defined, together with new operators to make the code more readable.

:- module( definitions, [

    op(1200, xfx, if),

    op(1000, xfy, and)

] ).

* **:- module(definitions, [ ... ]).** is a directive that defines a module named **definitions**. The module encapsulates a set of related predicates and provides a namespace for those predicates.
* **op(1200, xfx, if)** is a predicate that defines an operator. The **op/3** predicate is used to specify the precedence, associativity, and name of an operator.
* **op(1200, xfx, if)** declares the operator **if** with a precedence of 1200 and the **xfx** associativity. This means that the **if** operator has higher precedence than other operators and it is used for non-associative rule conditions.
* **op(1000, xfy, and)** declares the operator **and** with a precedence of 1000 and the **xfy** associativity. This means that the **and** operator has lower precedence than the **if** operator and it is used for right-associative conjunction of conditions within a rule.

## 4.2 Backward

The structure of backward chaining in our project can be represented as follows:

```prolog

backward(Query, KnowledgeBase, Result).

```

- `**Query**` represents the goal or the statement we want to prove or satisfy.

- `**KnowledgeBase**` represents the available knowledge or the set of rules and facts that we have.

- `**Result**` represents the result of the backward chaining process, which will be unified with a valid solution or a proof.

The backward chaining algorithm starts with the given `Query` and searches for a set of facts and rules in the `KnowledgeBase` that can be used to prove the `Query`. It applies backward inference, starting from the goal and working backward through the rules and facts, until it finds a solution or exhausts all possible paths.

The algorithm checks if the `Query` matches any facts in the `KnowledgeBase`. If there is a match, the goal is satisfied. If not, it looks for rules in the `KnowledgeBase` whose consequence matches the `Query`. It then recursively applies backward chaining to prove the conditions of those rules.

The process continues until a solution is found or all possible paths have been explored. If a solution is found, the `Result` will be unified with the valid solution or proof that satisfies the `Query`.

Here's an example of the syntax of backward chaining in our project:

```prolog

backward(has\_disease(patient, flu), KnowledgeBase, Result).

```

In this example, we are using backward chaining to determine if the patient has the disease "flu" based on the available `KnowledgeBase`. The `Result` will be unified with a valid solution or proof if the patient has flu.

## 4.3 Forward

The structure of forward chaining in our project can be represented as follows:

```prolog

forward(Facts, KnowledgeBase, Rule, Result).

```

In this structure:

- `**Facts**` represents the initial set of known facts or information.

- `**KnowledgeBase**` represents the collection of rules and facts that we have.

- `**Rule**` represents a specific rule that we want to apply in the forward chaining process.

- `**Result**` represents the result of the forward chaining process, which will be unified with the inferred facts or conclusions.

The forward chaining algorithm starts with the given set of `Facts` and searches for rules in the `KnowledgeBase` whose conditions or premises can be satisfied by the available facts. It applies forward inference, using those rules to derive new facts or conclusions.

The algorithm iteratively checks each rule in the `KnowledgeBase` and evaluates if its conditions are satisfied based on the available facts. If a rule's conditions are met, its consequence or conclusion is added to the set of facts. This process continues until no more new facts can be inferred.

Here's an example of the syntax of forward chaining in our project:

```prolog

forward([has\_symptom(patient, cough)], KnowledgeBase, has\_disease(patient, flu), Result).

```

In this example, we are using forward chaining to infer if the patient has the disease "flu" based on the initial fact that the patient has the symptom "cough". The `Result` will be unified with the inferred fact `has\_disease(patient, flu)` if the condition is satisfied and the inference is successful.

In our project, the knowledge base serves as a repository of information about medical conditions, symptoms, diseases, and treatments. It contains the facts and rules that the system uses for reasoning and providing accurate medical diagnosis.

# Knowledge Base

A knowledge base is a repository of organized information or data that represents knowledge about our domain. It contains facts, rules, and relationships that can be used for reasoning and decision-making. The knowledge base serves as a foundation for an intelligent system or application to draw conclusions, make inferences, and provide answers or solutions based on the available knowledge.

```prolog

:- module(get\_knowledge, [

    get\_knowledge/3,

    generate\_knowledge/3

]).

```

This line declares the module `get\_knowledge` and specifies the predicates `get\_knowledge/3` and `generate\_knowledge/3` that are exported by this module.

```prolog

:- use\_module('../definitions.pl'). % Import operator definitions

```

This line imports the operator definitions from the `definitions.pl` module, which is located in the parent directory.

```prolog

get\_knowledge(Rules, Template, Knowledge) :- get\_knowledge(Rules, Template, [], RKnowledge), reverse(RKnowledge, Knowledge).

```

This is a wrapper predicate that takes `Rules`, `Template`, and `Knowledge` as arguments. It calls the predicate `get\_knowledge/4` with an empty list `[]` as the accumulator and then reverses the accumulator to obtain the `Knowledge` in the correct order.

```prolog

get\_knowledge([], \_, Knowledge, Knowledge) :- !.

```

This is the base case of the `get\_knowledge/4` predicate. When the list of `Rules` becomes empty, it unifies `Knowledge` with the accumulated knowledge in the accumulator and stops.

```prolog

get\_knowledge([(Fact if Condition)|Rules], Template, ExistingKnowledge, NewKnowledge) :-

    (

        \+ \+ Fact = Template

    ->

        get\_knowledge\_(Condition, ExistingKnowledge, TempKnowledge)

    ;

        TempKnowledge = ExistingKnowledge

    ),

    get\_knowledge(Rules, Template, TempKnowledge, NewKnowledge).

```

This rule matches a rule of the form `(Fact if Condition)` in the `Rules` list. It checks if the `Fact` matches the `Template` without committing the unification.

If the match succeeds, it calls the predicate `get\_knowledge\_/3` to gather knowledge for that fact and updates the accumulator with `TempKnowledge`.

If the match fails, the accumulated knowledge remains unchanged. Then, it recursively calls `get\_knowledge/4` to process the remaining rules.

```prolog

get\_knowledge\_(Condition1 and Condition2, ExistingKnowledge, KnowledgePost) :- !,

    get\_knowledge\_(Condition1, ExistingKnowledge, TempKnowledge),

    get\_knowledge\_(Condition2, TempKnowledge, KnowledgePost).

```

This rule manages the conjunction of two conditions (`Condition1 and Condition2`). It recursively calls `get\_knowledge\_/3` for both conditions, passing the updated accumulator to the next condition.

```prolog

get\_knowledge\_(know(Knowledge), ExistingKnowledge, ExistingKnowledge) :-

    member(Knowledge, ExistingKnowledge), !.

get\_knowledge\_(know(Knowledge), ExistingKnowledge, [Knowledge|ExistingKnowledge]) :- !.

```

These two rules manage the `know(Knowledge)` condition. The first rule checks if the `Knowledge` already exists in the `ExistingKnowledge` list, ensuring that it is not queried twice. The second rule adds the `Knowledge` to the `ExistingKnowledge` list if it doesn't already exist.

```prolog

get\_knowledge\_(\_, Knowledge, Knowledge).

```

This rule serves as a catch-all clause that returns the `Knowledge` as is, allowing any other condition that doesn't match the previous rules.

```prolog

generate\_knowledge(\_, [], []).

```

This is the base case of the `generate\_knowledge/3` predicate. When the list of `RequiredKnowledge` becomes empty, it unifies the generated knowledge list as an empty list `[]` and stops.

```prolog

generate\_knowledge(KnowledgeHook, [RK1|RKs

], [RK1-TruthValue|Ks]) :-

    (

        call(KnowledgeHook, knowledge(RK1)) -> TruthValue = true

    ;

        TruthValue = false

    ),

    generate\_knowledge(KnowledgeHook, RKs, Ks).

```

This rule generates the knowledge for each element (`RK1`) in the `RequiredKnowledge` list.

It is called the `KnowledgeHook` predicate with `knowledge(RK1)` as an argument.

If the call succeeds, it assigns `TruthValue` as `true`; otherwise, it assigns `TruthValue` as `false`.

It then recursively calls `generate\_knowledge/3` to process the remaining elements in `RequiredKnowledge`.

Overall, the knowledge base code provides the necessary predicates and rules to gather and generate the knowledge required for decision-making and inference.

It manages conditions, facts, and conjunctions to collect relevant information from the existing knowledge base and generate new knowledge based on certain criteria.

# Interface Implementation

Our Intentions were to provide an interactive interface where the user can input symptoms, the program can infer potential diseases using forward and backward chaining, and treatment recommendations can be displayed based on the diagnosed disease.

But we thought providing just a simple terminal interface was not going to give any favorable user experience. Hence, we have Implemented the XPCE.

1. The `**terminal**` module (`**terminal.pl**`) and `**xpce**` module (`**xpce.pl**`) are the entry points for running the program in terminal mode or graphical mode, respectively.

2. The `terminal` module uses the `read\_data` module (`**read\_data.pl**`) to read treatment and diagnosis data from files. It also uses the `hook\_ask` module (`**hook\_ask.pl**`) and `hook\_ask\_remember` module (`**hook\_ask\_remember.pl**`) for knowledge handling and querying the user.

3. The `xpce` module uses the `read\_data` module to read treatment and diagnosis data from files. It also uses the `hook\_preset` module **(`hook\_preset.pl**`) for preset knowledge handling.

4. The `read\_data` module provides predicates for reading treatment and diagnosis data from CSV and Prolog files.

5. The `hook\_ask` module defines a knowledge hook that asks the user for input whenever knowledge is queried. It uses the `library(readutil)` to read user input from the terminal and provides a `question/1` predicate to ask questions.

6. The `hook\_ask\_remember` module defines a knowledge hook that remembers answers to questions asked by the `hook\_ask` module. It uses Prolog's dynamic predicates (`dynamic/1`) to store the known knowledge and provides a `reset/0` predicate to clear the stored knowledge.

7. The `hook\_preset` module defines a knowledge hook that takes a list of preset knowledge and returns the appropriate knowledge when queried. It is used in the XPCE interface mode.

8. The `get\_knowledge` module (`get\_knowledge.pl`) is a core module that generates the necessary knowledge based on a diagnosis and a goal. It uses the `hook\_ask\_remember` hook to generate knowledge that is later used for forward and backward chaining.

9. The `backward` module (`backward.pl`) implements the backward chaining algorithm, which is used to infer diagnoses based on symptoms and knowledge. It considers the knowledge generated by the hooks and checks for consistency.

10. The `forward` module (`forward.pl`) implements the forward chaining algorithm, which is used to infer diagnoses based on symptoms and knowledge. It also considers the knowledge generated by the hooks and checks for consistency.

11. The `treatments` module (`treatments.pl`) provides a predicate for writing treatment information based on a diagnosis and a list of treatments. It is used to display treatment information in both the terminal and graphical interfaces.

12. The `diagnosis\_window` class in the `xpce` module is an XPCE class that represents the graphical interface window.

It consists of menus for selecting categories, symptoms, and diseases, as well as text boxes for displaying treatment information.

It interacts with the other modules to update the displayed information based on user selections.

In summary, the interface in this project is managed through a combination of terminal-based input/output, knowledge hooks for user interaction, and a graphical interface implemented using XPCE.

The different modules manage data reading, knowledge handling, inference algorithms, and interface display and interaction.

Here is the screenshot of how the User Interface looks like in both XPCE and Terminal.

## 6.1 XPCE Interface :

As we can see in the below example only one category at a time can be selected to filter out the diseases.

i.e., In the figure 1.1 below we can see that the Cardiovascular category has been selected for the choice. (Highlighted in Red for the Illustration purposes only).



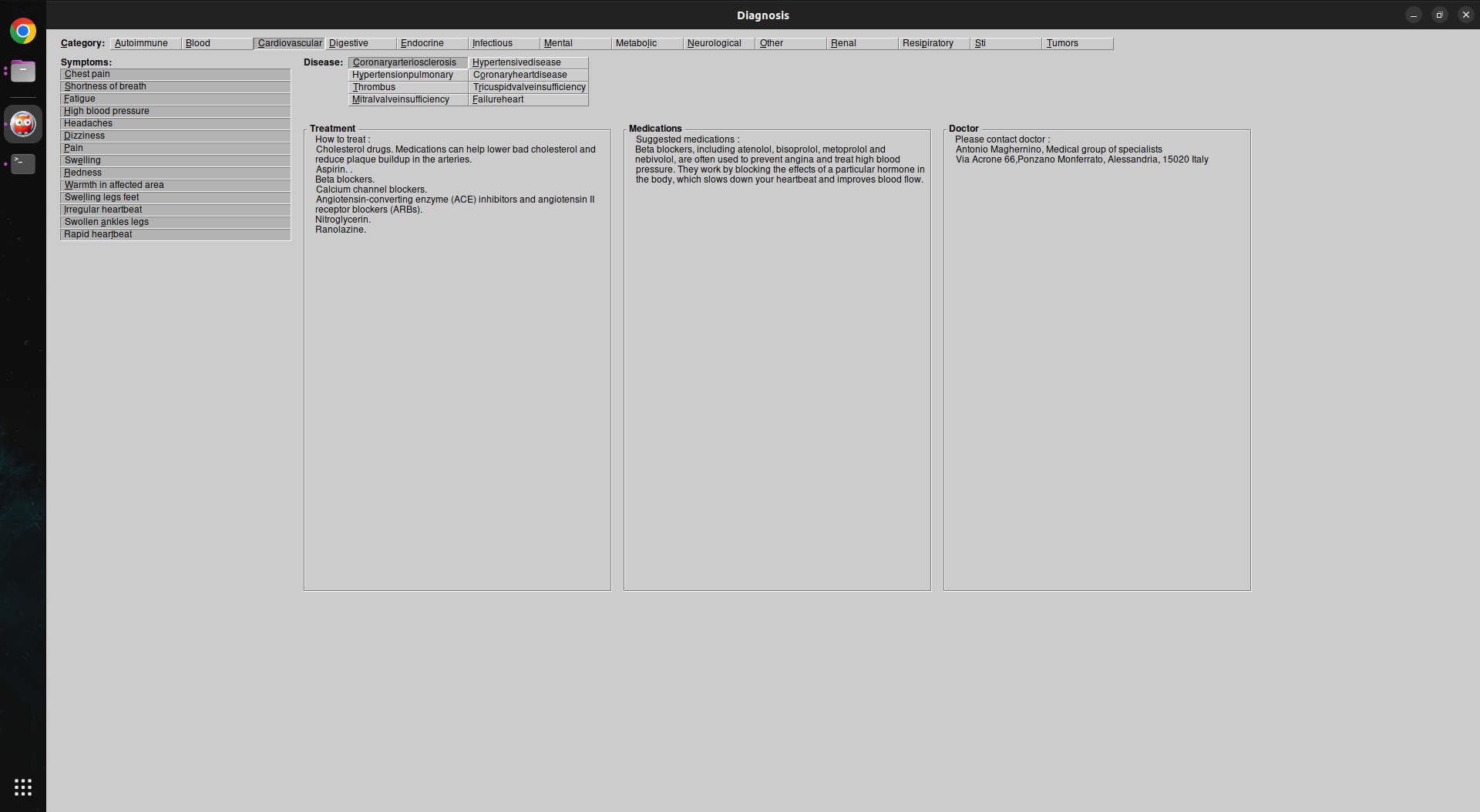
**1.1**

i.e., In the figure 1.1 below we can see that the Infectious category has been selected for the choice.



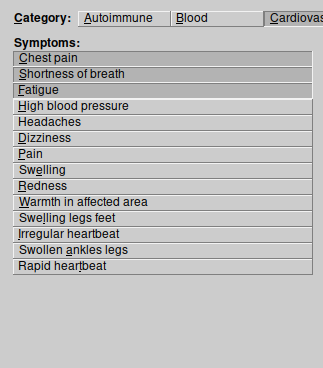
**1.2**

In the figure 1.3 we can see all the Symptoms are in the selected state by default (Selected ones appear Darker) after we have selected the category, This is because the user has to deselect the symptoms which he feels like he doesn’t have. So that we could be able to narrow down the given list of diseases.



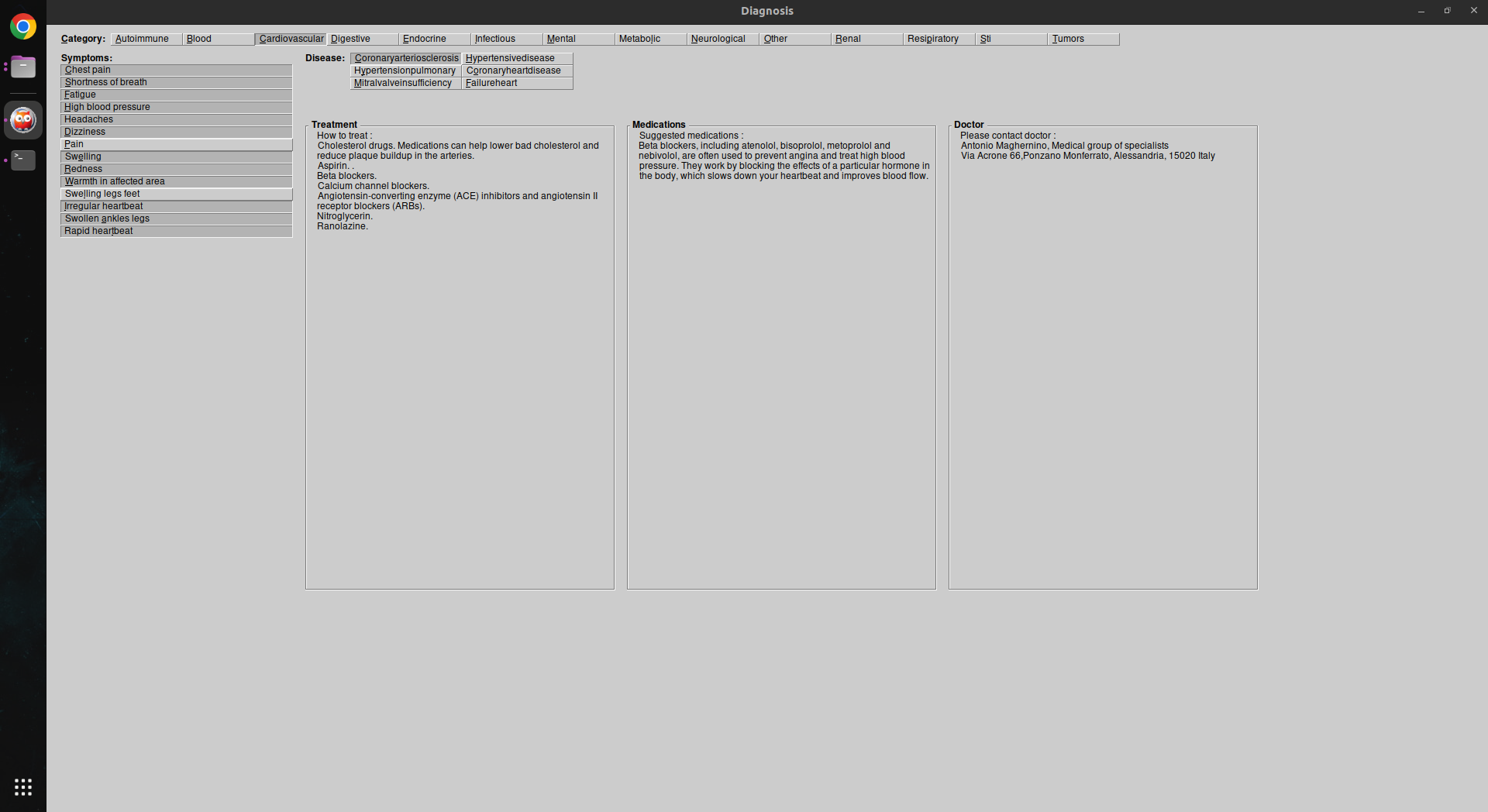
**1.3**

The figure 1.4 represents the symptoms where some of the symptoms are being deselected by the user to narrow down the disease for right diagnosis.



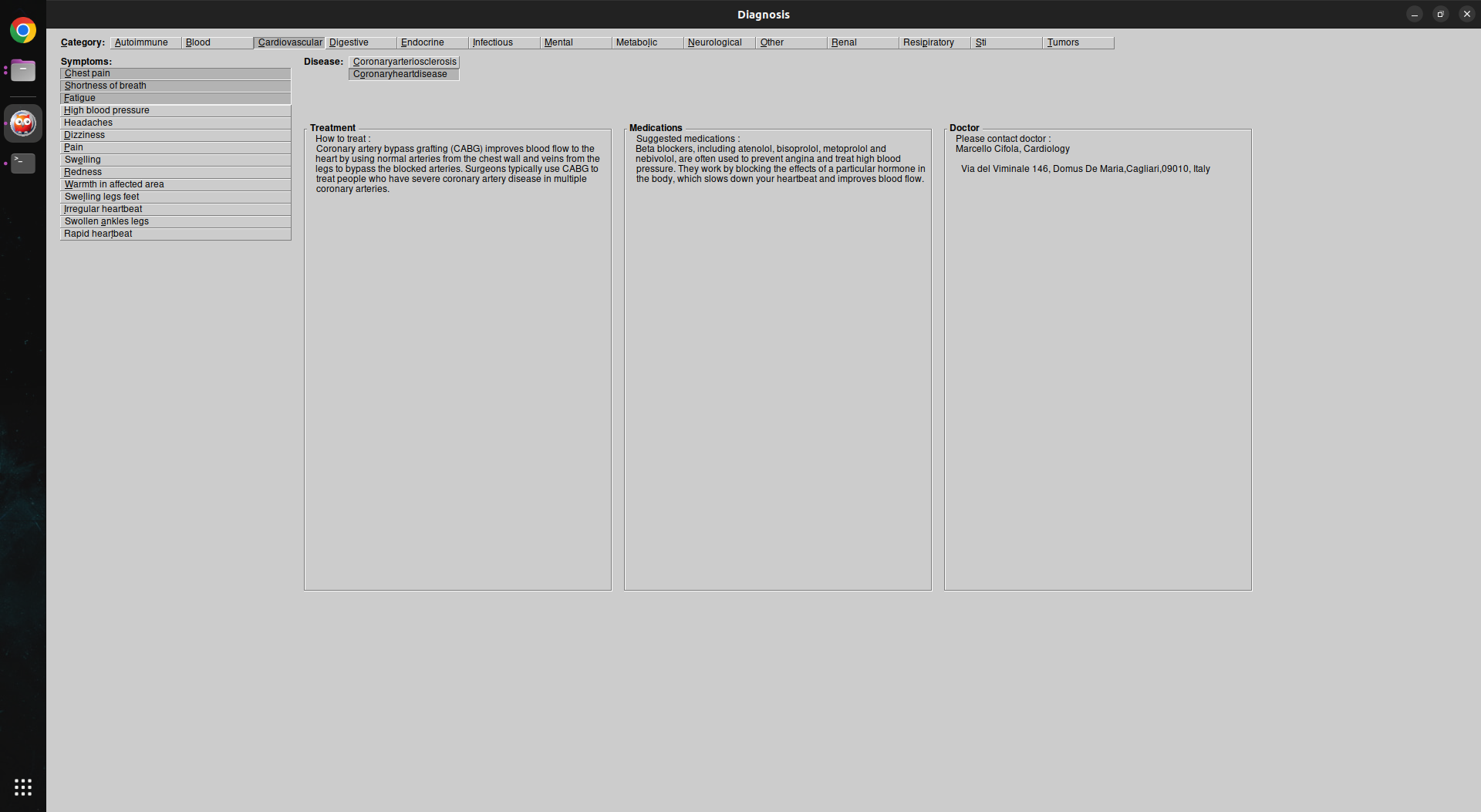
**1.4**

In the figure 1.5 we can see some of the diseases appear to be less in number now compared to the figure 1.3.



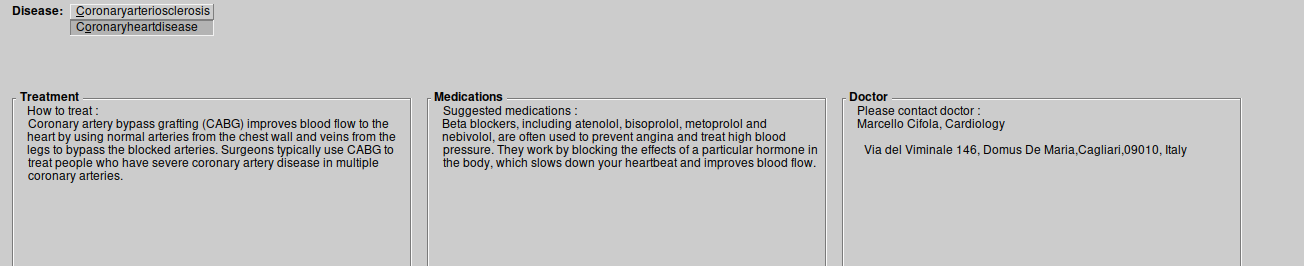
**1.5**

Finally in the figure 1.6 we have the completely narrowed down single/same disease under different name is being displayed once all the symptoms are compiled.



**1.6**

This figure 1.7 illustrates us the final suggestion for the treatment, Medications and Doctor for the diagnosed disease.

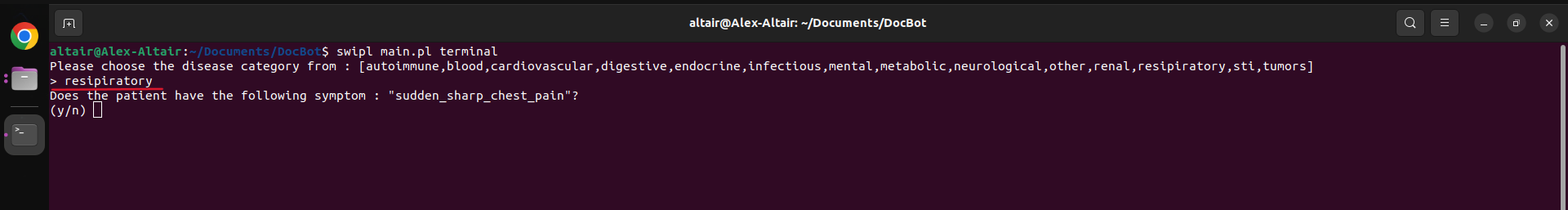


**1.7**

## Terminal Interface :

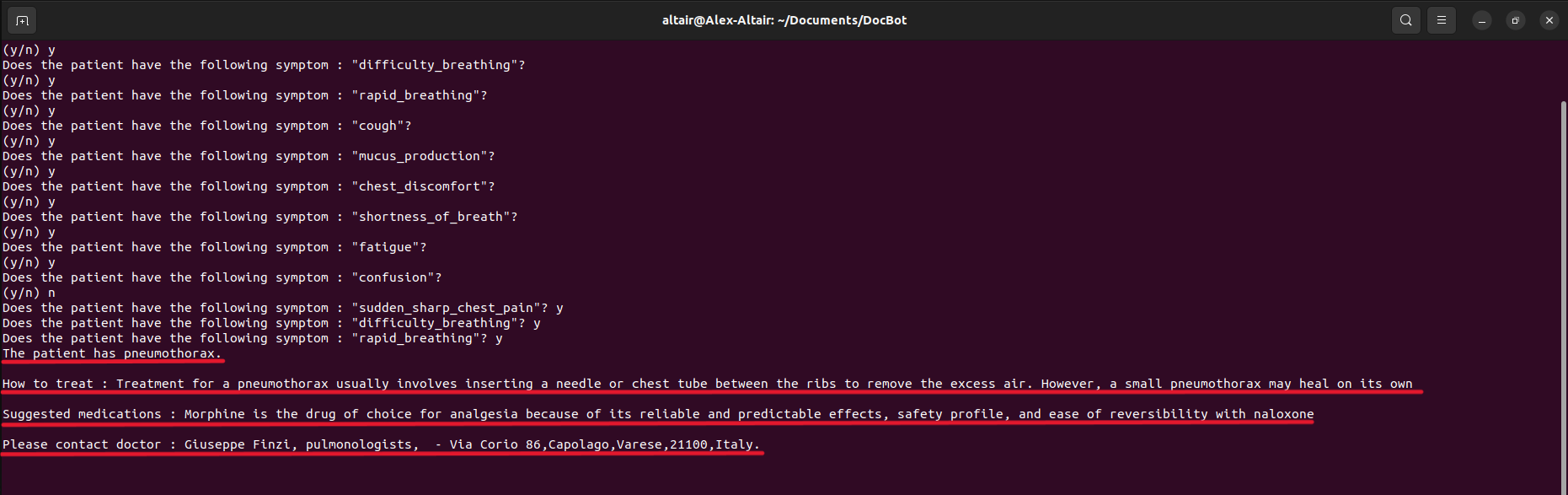
The terminal interface in this project provides a command-line interface for interacting with the symptom checker program.

The figure 1.8 shows the terminal asking to choose the category of the disease.



**1.8**

The figure 1.9 shows the terminal asking to answer Yes or No to the symptoms one after the other. Once it has all the matching symptoms with the disease that falls under the selected category of the Disease then it suggests the Diagnosis, Treatment Suggestions, Doctor on that specific specialization of disease and the medications.



**1.9**

# Conclusions and Future Developments

Certainly! Here's the revised version of the suggestions, written as if we were explaining them to someone:

1. We believe one area that could be improved in this project is the code organization and modularity. Currently, all the code is in a single file, which can make it difficult to navigate and maintain. We suggest separating the code into logical modules or classes, each responsible for a specific functionality. This will make the codebase more manageable and easier to maintain in the long run.

2. Another improvement could be in error handling and input validation. Right now, there is limited error checking and handling of invalid user input. It would be beneficial to add appropriate checks to ensure that the user enters valid input and provide meaningful error messages or prompts when necessary.

3. The graphical interface implemented using XPCE could also be enhanced to provide a more intuitive and visually appealing user experience. We could improve the current layout, design, and usability to make it more user-friendly and aesthetically pleasing.

4. In terms of the knowledge base, we noticed it's a bit limited in the `knowledge.pl` file. To make the diagnosis more comprehensive and accurate, we suggest expanding and refining the knowledge base with additional diseases, symptoms, and treatment options.

5. We think it would be interesting to explore more advanced diagnostic algorithms for improved accuracy. The current diagnosis process follows a simple rule-based approach, but incorporating more advanced techniques like probabilistic reasoning, machine learning, or expert systems could be beneficial. This could involve using statistical data, machine learning models, or more sophisticated inference mechanisms.

6. Lastly, we recommend implementing unit tests for critical functionalities. This will help ensure the correctness and robustness of the project. It would also be good to enhance the error handling mechanisms to gracefully manage unexpected situations, making the program more dependable overall.

These are some suggestions we have for making the project better in terms of functionality, usability, and maintainability. They might require some additional development work, but they can contribute to making the project more effective and user-friendly.