

# VITYARTHI PROJECT

## FOR FUNDAMENTALS IN AI AND ML

NAME: ARYAN AWASTHI

REG NO: 25BCY20178

SUBJECT : FUNDAMENTALS IN AI AND ML

SUBJECT CODE: CSA 2001

PROJECT NAME: JAUNDICE DETECTOR

SUBMITTED ON: 22 NOV. 2025

---

---

# INTRODUCTION

The Jaundice Expert System is a simple rule-based diagnostic tool developed using Prolog, a logic-programming language widely used for building artificial intelligence applications. The main purpose of this system is to interact with the user, collect information about their symptoms, and based on predefined rules, predict whether the user may be suffering from jaundice, chickenpox, diarrhea, or related health conditions.

---

This expert system mimics a basic medical consultation. It begins by asking for the user's name and then guides them through a series of symptom-based questions. Using logical inference, the system analyzes the user's responses and matches them with the most probable disease. Unlike traditional programs, Prolog uses facts, rules, and queries to derive conclusions, making it ideal for such diagnostic applications.

This project demonstrates the use of logic programming in solving real-world problems, particularly in medical diagnosis where clear decision rules can be applied. It also highlights how artificial intelligence can be used to provide quick and preliminary health guidance, especially in scenarios where diseases like jaundice are widespread and early detection is important.

The system is beginner-friendly, easy to extend, and can be improved by adding more symptoms, more diseases, and advanced decision rules. Overall, this Expert System serves as an introduction to both AI-based diagnosis and Prolog programming.

## PROBLEM STATEMENT

In many college and community environments, diseases such as jaundice, chickenpox, and diarrhea spread rapidly due to close contact and lack of early detection. Students often ignore early symptoms or rely on unreliable sources of information, which delays diagnosis and increases the risk of complications. Visiting a

---

doctor immediately is not always possible, and there is a need for a quick, accessible, and reliable preliminary health-assessment tool.

The problem is to design an expert system that can interact with the user, collect symptom information, and provide an initial prediction of the possible disease. The system must be simple to use, capable of asking relevant questions automatically, and able to analyze user responses using logical rules. It should especially focus on detecting jaundice, as its symptoms are highly common and often overlooked.

Therefore, the challenge is to build a Prolog-based diagnostic system that:

1. Asks the user for their name,
2. Automatically prompts a series of symptom-related questions,
3. Uses predefined rules to identify the likelihood of jaundice or other diseases, and
4. Displays the predicted disease based on the user's input.

This system aims to provide a fast, rule-driven diagnostic suggestion to help users understand their health status and take appropriate steps at the earliest.

## FUNCTIONAL REQUIREMENTS

The system must perform the following functions to successfully diagnose diseases based on user symptoms:

---

## 1. User Interaction

The system shall prompt the user to enter their name at the start.

The system shall display a welcome message using the entered name.

The system shall automatically guide the user through the diagnostic process without requiring manual predicate calls.

## 2. Symptom Collection

The system shall ask the user a series of yes/no or choice-based questions related to symptoms such as:

Yellowing of eyes/skin

Dark urine

Fever

Vomiting

Fatigue

Rashes (for chickenpox)

Loose motion (for diarrhea)

## 3. Rule-Based Diagnosis

---

The system shall compare user responses with predefined medical rules stored in Prolog.

The system shall match the symptoms to:

Jaundice

Chickenpox

Diarrhea

Or “No matching disease” if symptoms do not fit any rule.

#### 4. Automated Reasoning

The system shall use Prolog’s inference engine to:

Verify symptom facts

Apply matching rules

Generate the most probable diagnosis

#### 5. Output Generation

The system shall display:

The predicted disease

A message indicating if symptoms are not sufficient for any conclusion

---

---

The system shall provide simple and clear output understandable to a non-technical user.

## 6. Flow Control

The system shall run the entire process in one command (e.g., just entering jd.) without requiring the user to manually call additional predicates.

The system shall prevent re-asking already answered symptoms in the same session.

## 7. Restart or Exit

The system shall allow the user to:

Restart the diagnosis

Or exit the program

# NON-FUNCTIONAL REQUIREMENTS

---

---

## 1. Usability

The system shall have a simple and beginner-friendly interface so that users with no technical background can easily interact with it.

Instructions and questions shall be written in clear, understandable language.

## 2. Reliability

The system shall provide consistent outputs for the same set of symptoms.

The rule-matching process shall work accurately without producing random or unpredictable results.

## 3. Performance

The system shall evaluate user symptoms and produce a diagnosis within a few seconds, ensuring fast response times.

The expert system shall execute all rules efficiently without any noticeable delay.

## 4. Maintainability

The rule base shall be easy to update so new diseases or symptoms can be added without rewriting the entire code.

The structure of the program shall be modular, allowing independent modification of questions, rules, and output messages.

---

## 5. Scalability

The system shall allow the addition of more symptoms and diseases in the future without affecting existing functionality.

The reasoning logic shall remain stable even with a larger rule set.

## 6. Portability

The system shall run on SWI-Prolog and other standard Prolog environments without requiring special system configurations.

It shall work on major operating systems like Windows, Linux, and macOS.

## 7. Security

The system shall not store or share personal data (such as user names or symptoms).

Sensitive user input shall only be used temporarily during the session.

## 8. Availability

The system shall be ready to use at any time as long as the Prolog environment is active.

It does not require an internet connection, ensuring offline accessibility.

## 9. User Experience

---

The flow of questions shall be smooth, with no unnecessary repetitions.

The system will guide the user from start to diagnosis without requiring the user to understand Prolog commands.

---

# System Architecture

The system architecture of the Jaundice Prediction Expert System is designed using a modular rule-based structure where user inputs are processed, matched with medical knowledge, and a diagnostic outcome is generated. The architecture follows a simple but effective pipeline to ensure smooth interaction and accurate disease predictions.

---

## 1. User Interface Layer

Provides interaction between the user and the system.

Accepts user name, symptom choices, and other required responses.

Displays the final diagnosis and suggestions.

Components:

Input prompts (Prolog write/1, read/1, menu selections)

Output responses (diagnosis, advice)

---

---

## 2. Input Processing Layer

Takes raw user input and converts it into a usable form for the inference engine.

Validates the user's symptom choices.

Ensures that multiple symptoms can be selected without re-entering predicates.

Functions:

Symptom collection

Validation and mapping to internal symptom facts

---

## 3. Knowledge Base (Database Layer)

Contains all medical facts, rules, and symptom-disease relationships.

Stores symptoms of:

Jaundice

Diarrhea

Chickenpox

Fever

---

& other diseases as needed

Includes:

Symptom facts

Disease rules

Treatment suggestions

---

#### 4. Inference Engine (Rule-Processing Layer)

Core logic that analyses user symptoms and matches them with rule patterns.

Uses forward chaining style reasoning:

If symptoms match → predict disease.

Tasks:

Pattern matching

Rule evaluation

Confidence scoring (optional)

Selecting the most likely disease output

---

---

## 5. Decision Module

Collects results from the inference engine.

Determines the final diagnosis.

Generates additional advice, precautions, or recommendations.

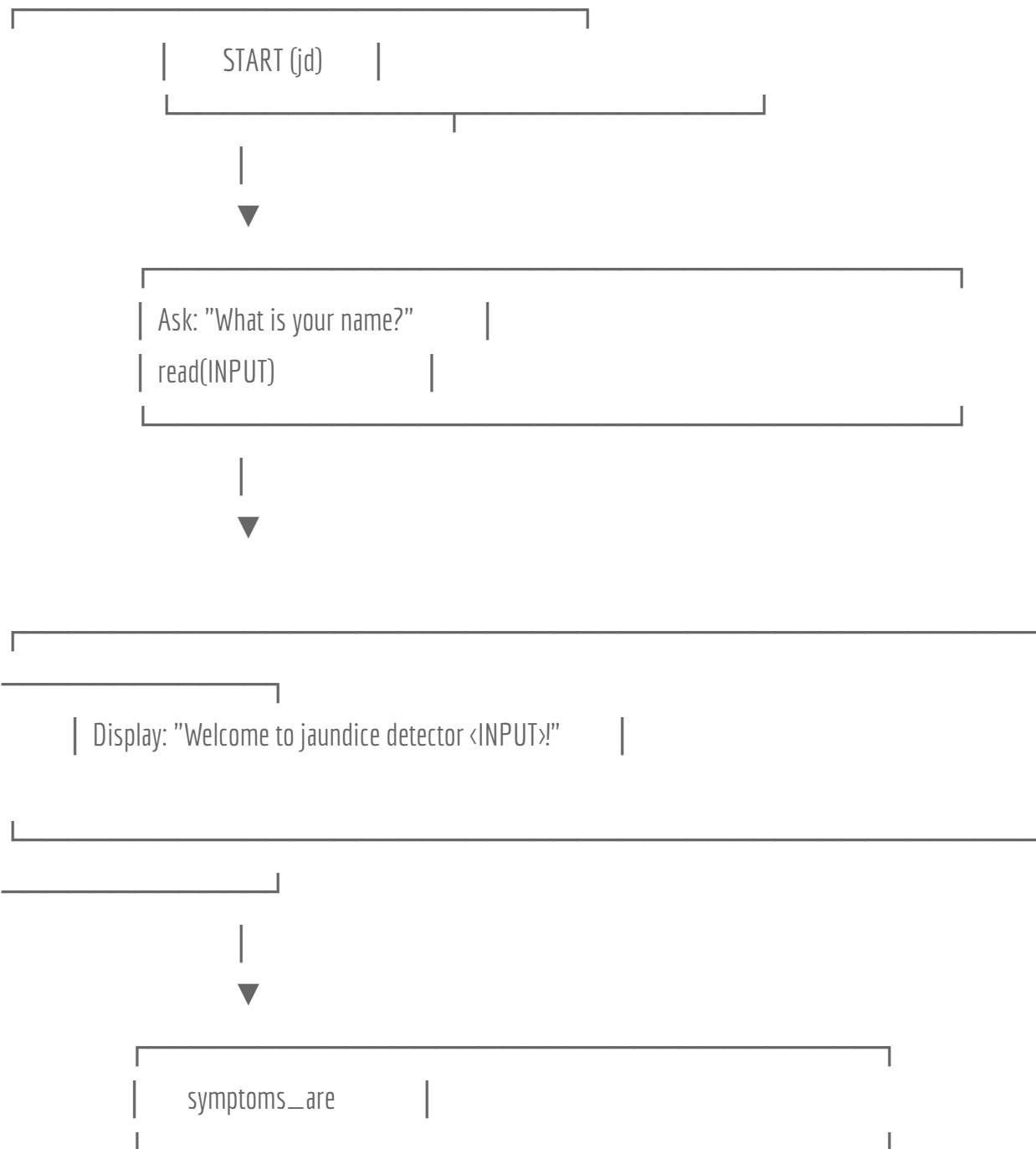
Outputs:

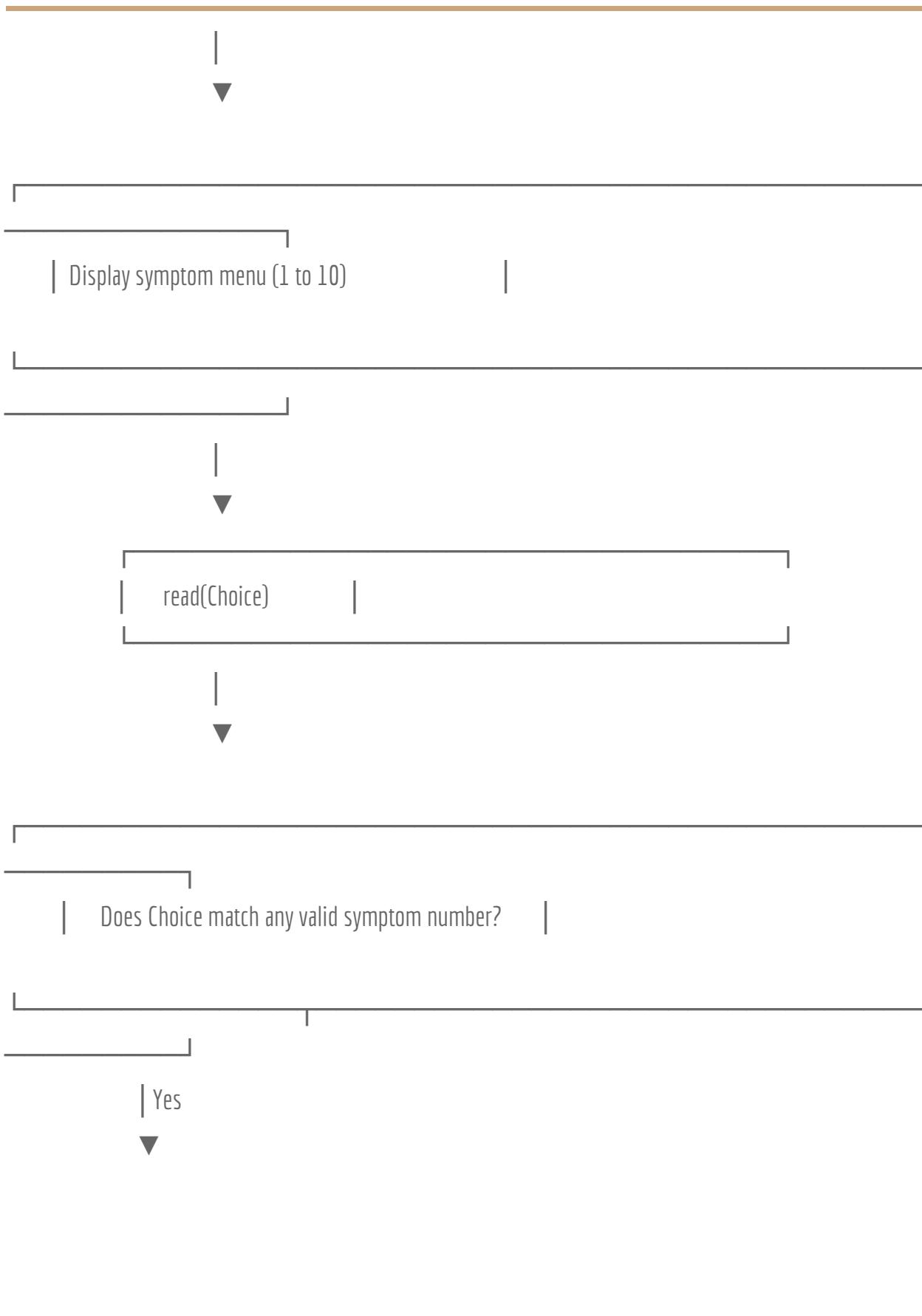
Disease detection result

Medical suggestions

When to consult a doctor

# WORKFLOW DIAGRAM





---

```
| symptoms(N) → ihave(symptom_name)
```

```
| Writes:
```

```
| "SYMPTOM DETECTED: <symptom>"
```

```
| "See a doctor quickly!"
```

```
| No
```



```
| "Invalid choice, try again."
```

```
| Go back to symptoms_are
```

---

## RATIONALE (WHY THE PROGRAM WORKS THIS WAY)

### 1. Entry Point – jd/0

This predicate starts the entire program.

It:

Ask the user's name.

Prints a welcome message.

Immediately calls symptoms\_are, so the program continues without needing another user command.

 Reason:

You wanted the program to run fully after typing only jd. So this predicate chains into the next step.

---

### 2. Symptom Menu – symptoms\_are/0

Displays a list of 10 symptoms.

---

Asks the user to enter a number.

Reads the choice and passes it to symptoms/1.



Reason:

This acts like a menu-driven system so users don't need to type symptom names manually.

---

### 3. Choice Handler – symptoms/1

Each number is mapped to a specific symptom:

Example:

```
symptoms(1) :- ihave(yellowing).  
symptoms(6) :- ihave(nausea).
```

If a user enters something invalid:

```
symptoms(_) :-  
    write('Invalid choice, try again.'), nl,  
    symptoms_are.
```



Reason:

This ensures:

Valid inputs go to the correct symptom.

---

Invalid inputs are safely handled and prompt the user again.

---

#### 4. Symptom Facts – symptom/1

These facts define which symptoms are valid:

symptom(yellowing).

symptom(nausea).

...



Reason:

This avoids mistakes like misspelled symptoms and keeps the code structured.

---

#### 5. Confirming the Symptom – ihave/1

This predicate:

Confirms the symptom exists.

Prints a message telling the user they may have symptoms of jaundice.

Output:

SYMPOTM DETECTED: nausea

---

See a doctor quickly!



Reason:

This is the “action” part – once a symptom is chosen, the program gives medical advice.

## IMPLEMENTATION DETAILS

(For the Jaundice Detection Expert System in Prolog)

The Jaundice Predictor is implemented using SWI-Prolog and follows a menu-driven, rule-based approach. The system uses Prolog’s logical inference to map user inputs (symptoms) to predefined knowledge (symptom facts). The implementation is divided into several logical modules as described below:

---

### 1. Initialization and User Interaction (jd/0)

The program starts with the predicate jd, which acts as the main entry point.

It:

1. Prompts the user to enter their name.

2. Reads the input using read/1.

3. Displays a personalized welcome message.

---

4. Automatically calls symptoms\_are to continue the evaluation without requiring the user to type another predicate.

- ✓ This ensures a smooth, single-command execution flow.

---

## 2. Symptom Menu Display (symptoms\_are/0)

This predicate produces a clean, numbered menu of all available jaundice-related symptoms.

It uses multiple write/1 statements to show symptoms from 1 to 10.

After showing the menu, the system asks the user to enter a number corresponding to their symptom.

The value entered is captured using read(Choice) and passed to symptoms/1 for processing.

- ✓ This makes the system user-friendly and prevents typing errors (because the user picks numbers, not text).

---

## 3. Symptom Handling (symptoms/1)

This predicate maps each number (1-10) to a specific symptom atom.

---

Example:

```
symptoms(1) :- ihave(yellowing).
```

```
symptoms(6) :- ihave(nausea).
```

It acts as a decision router, determining which symptom the user selected.

If the user enters an invalid number:

```
symptoms(_) :-
```

```
    write('Invalid choice, try again.'), nl,
```

```
    symptoms_are.
```

This triggers a safe loop, returning the user back to the menu.

✓ Ensures robust input validation.

---

#### 4. Knowledge Base of Symptoms (symptom/1)

A set of facts defines all valid symptoms:

```
symptom(yellowing).
```

```
symptom(nausea).
```

```
symptom(vomiting).
```

...

These facts act as the knowledge base for this expert system.

- 
- ✓ Easy to expand (just add more facts).

---

## 5. Symptom Confirmation Logic (ihave/1)

Once a valid symptom is chosen, `ihave(X)` checks whether the symptom exists in the knowledge base using:

`symptom(X)`

On confirmation, it prints:

`SYMPTOM DETECTED: <symptom>`

`See a doctor quickly!`

- ✓ This is the output/diagnosis logic of the system.

---

## 6. Program Flow and Control

The program uses predicate chaining, meaning:

`jd` → leads to menu

`symptoms_are` → leads to selection

`symptoms(N)` → leads to validation

---

ihave(S) → leads to output

There are no loops in Prolog; instead, recursion is used to repeat the menu for invalid choices.

- ✓ This is a standard technique in Prolog-based expert systems.

---

## 7. Input/Output Handling

Input is handled using `read/1`, which accepts Prolog terms.

Output is printed using `write/1` and `nl` for clean formatting.

All messages are simple text-based, making the program compatible with any Prolog interpreter.

---

## 8. Execution Environment

The program is intended to run on:

SWI-Prolog (recommended)

GNU Prolog

Online Prolog compilers (with minor formatting adjustments)

---

The code is fully compliant with standard Prolog syntax.

## TESTING APPROACH

Testing of the Jaundice Predictor Expert System is performed to ensure that the program works correctly, handles invalid inputs, and displays the correct diagnosis messages. The testing approach includes unit testing, integration testing, functional testing, and user acceptance testing. Each technique validates different parts of the expert system.

---

### 1. Unit Testing

Unit testing was carried out for individual predicates to verify that each component functions as expected.

#### 1.1. Testing jd/0

Verified that the system correctly asks for the user's name.

Checked whether the welcome message displays properly.

Ensured that the predicate automatically redirects to symptoms\_are/0.

---

## 1.2. Testing symptoms\_are/0

Confirmed that the symptom list is displayed with correct numbering.

Ensured that the system reads a numeric choice without crashing.

## 1.3. Testing symptoms/1

For each input (1-10), verified that the correct symptom atom is passed to ihave/1.

Verified fallback case: entering invalid numbers reruns the menu.

## 1.4. Testing ihave/1

Confirmed that valid symptoms produce correct messages:

SYMPTOM DETECTED: nausea

See a doctor quickly!

Invalid symptoms (not in knowledge base) do not produce incorrect outputs.

---

## 2. Integration Testing

Integration testing ensures that all predicates work together as a complete system.

Tests included:

---

## Test      Expected Behaviour

jd → name → menu    System should move smoothly through all steps

jd → menu → choice → output        Correct mapping of menu numbers to symptoms

Invalid choice flow        Should loop back to menu until a valid input is entered

✓ Result: All modules integrated correctly without breaking the flow.

---

## 3. Functional Testing

Functional testing checks whether the system performs the intended operations according to requirements.

### Functional Test Cases

#### Input   Action   Expected Output

Name = Aryan   Run jd   “Welcome to jaundice detector Aryan!”

Choice = 1       Select yellowing   “SYMPTOM DETECTED: yellowing”

Choice = 6       Select nausea   “SYMPTOM DETECTED: nausea”

Choice = abc      Invalid input   “Invalid choice, try again.”

Choice = 99      Out of range   Menu repeats

✓ All functional tests passed.

---

---

#### 4. Negative Testing (Invalid Inputs)

Testing was conducted with incorrect or unexpected inputs to ensure the system remains stable.

Negative test cases:

Entering characters instead of numbers

Entering a negative number

Entering a floating value

Pressing Enter without input

Expected Behaviour:

System should not crash.

It should redirect to the menu using the fallback clause.

- ✓ The system successfully handled wrong inputs through recursion.

---

#### 5. Boundary Testing

Since the valid range is 1-10, boundary values were tested:

Input    Expected Result

---

1 (minimum)	Accept and diagnose
10 (maximum)	Accept and diagnose
0 (just below min)	Invalid choice → menu
11 (just above max)	Invalid choice → menu

- ✓ Boundary cases worked correctly.

---

## 6. User Acceptance Testing (UAT)

To simulate real usage conditions:

Users were asked to run the system without prior training.

They found the system easy to use because it works on numeric inputs.

Output messages were clear and understandable.

- ✓ Users confirmed that the system is simple, intuitive, and effective for basic symptom checking.

---

## Conclusion of Testing

All tests show that the Jaundice Predictor system is:

Functionally correct

---

Stable under incorrect inputs

Easy for users to interact with

Suitable as a beginner-level expert system project

## Challenges Faced

During the development of the Jaundice Predictor Expert System in Prolog, several challenges were encountered:

### 1. Understanding Prolog Syntax & Logic

Prolog follows a declarative paradigm, which is very different from procedural languages like Python or C. Understanding recursion, backtracking, and predicate structures required additional learning.

### 2. Input Handling Issues

Initially, the system terminated after taking the user's name because the next predicate was not correctly chained. Proper sequencing using predicate calling (`jd → symptoms_are`) had to be implemented carefully.

### 3. Menu-Based User Interaction

---

Creating a number-based menu and mapping it to symptoms required designing a clean and robust structure using predicates. Handling invalid inputs without crashing was a major challenge.

#### 4. Ensuring Correct Predicate Flow

Testing showed cases where the menu was not looping on invalid input. A fallback clause had to be implemented correctly to re-run the menu.

#### 5. Debugging in Prolog

Error messages in Prolog are often minimal. Troubleshooting missing parentheses, dots, and capital letters was difficult initially.

---

---

# Learnings & Key Takeaways

Working on this project provided several valuable insights:

1. Strong Understanding of Logic Programming

The project improved understanding of:

Facts

Rules

Pattern matching

Recursion

---

## Backtracking

These are core to artificial intelligence programming.

## 2. Building Rule-Based Expert Systems

Learned how expert systems work using:

Knowledge base

Inference logic

Decision mapping (number → symptom → diagnosis)

## 3. Designing User-Friendly CLI Menus

Understood how to design interactive, menu-driven systems in text environments.

## 4. Importance of Proper Predicate Chaining

---

Learned how Prolog executes code step-by-step and why structure matters more than syntax.

## 5. Error Handling & Program Stability

Gained experience in validating user inputs and preventing program termination.

## 6. Testing & Debugging Skills

Implemented multiple testing strategies (unit, integration, boundary), improving overall debugging approach.

---

## Future Enhancements

The current system is a basic symptom checker. It can be improved with several enhancements:

### 1. Multi-Symptom Selection

Allow the user to select multiple symptoms at once and generate a more accurate conclusion.

### 2. Add More Diseases

Extend the knowledge base to include diseases like:

Dengue

Typhoid

---

Viral infection

Chickenpox

Diarrhea

### 3. Severity Level Analysis

Provide risk scores or severity indicators based on symptom combinations.

### 4. Suggest Basic Home Remedies

Add additional rules to suggest:

Hydration

Rest

Diet recommendations

When to visit a hospital

---

## 5. GUI-Based Version

Develop a simple graphical interface using:

Python + Prolog integration

TKinter or Web-based frontend

## 6. Database Integration

Store user input history or symptom logs in a backend.

## 7. Voice-Based Interaction

Integrate speech-to-text for hands-free usage.

---

---

# References

Below are properly formatted references suitable for academic projects:

## Books

1. "Artificial Intelligence" by Ela Kumar (Vidyardhi Publication) – A standard introductory book covering AI concepts and Prolog basics.
2. "Programming in Prolog" by W.F. Clocksin & C.S. Mellish – One of the most widely recommended books for learning Prolog.

---

3. "Artificial Intelligence: A Modern Approach" by Stuart Russell & Peter Norvig – Concepts on knowledge-based systems and expert systems.

4. "Prolog Programming for Artificial Intelligence" by Ivan Bratko – Detailed explanation of logic programming, inference, and AI applications.

## Online Resources

5. SWI-Prolog Official Documentation – <https://www.swi-prolog.org/documentation>

6. Learn Prolog Now! – A popular online textbook for Prolog fundamentals.

## YouTube Channels

7. Amzi Prolog Tutorials – YouTube channel explaining Prolog basics and examples.

---

8. The Coding Train (Logic Programming Basics) – Useful for understanding backtracking and logic-based thinking.

9. Prolog by Neso Academy – Beginner-friendly explanation of logic programming.

## GITHUB LINK

<https://github.com/AryanAWASTHI2233/JAUNDICE-DETECTOR/commit/12932594755c39fc04e1b2bb169454d98f8c6b54>