Implementação Lógica do Jogo '*Quem Quer Ser Milionário*'

Pedro Reis
Escola de Engenharia
Lógica e Inteligência Artificial
Universidade do Minho
Braga, Portugal
PG59908@alunos.uminho.pt

Guilherme Pinto
Escola de Engenharia
Lógica e Inteligência Artificial
Universidade do Minho
Braga, Portugal
PG60225@alunos.uminho.pt

João Azevedo
Escola de Engenharia
Lógica e Inteligência Artificial
Universidade do Minho
Braga, Portugal
PG61693@alunos.uminho.pt

Luís Silva
Escola de Engenharia
Lógica e Inteligência Artificial
Universidade do Minho
Braga, Portugal
PG60390@alunos.uminho.pt

Diogo Lopes Azevedo
Escola de Engenharia
Lógica e Inteligência Artificial
Universidade do Minho
Braga, Portugal
PG61217@alunos.uminho.pt

***Abstract*—O desenvolvimento de um sistema de inteligência artificial simbólica que recria a mecânica do concurso televisivo *"Quem Quer Ser Milionário"*, implementado integralmente na linguagem *Prolog*. O projeto explora as potencialidades do paradigma declarativo para a modelação de árvores de decisão, gestão de estado recursivo e validação de conhecimento através de regras de inferência formal, incluindo a aplicação explícita de *Modus Ponens*, *Modus Tollens* e a falácia *Modus Mistaken*. A arquitetura da solução distingue-se pela sua modularidade e pela implementação de um sistema híbrido que integra lógica simbólica com modelos de linguagem (LLMs) via *Python*, demonstrando interoperabilidade entre paradigmas. Adicionalmente, o projeto apresenta um motor de renderização multimédia para terminal, capaz de processar animações ANSI e áudio assíncrono, elevando a experiência de interação homem-máquina num ambiente de consola clássico.**

***Index Terms*—Prolog, IA Simbólica, Sistemas Híbridos, LLMs**

I. Introdução

Este relatório apresenta o desenvolvimento de um sistema de inteligência artificial simbólica, implementado na linguagem *Prolog*, que simula a lógica e mecânica do concurso "*Quem Quer Ser Milionário*". O projeto foca-se na construção de um motor de inferência capaz de gerir bases de conhecimento dinâmicas, validar respostas através de dedução lógica.

A solução desenvolvida explora a capacidade da programação em lógica para modelar sistemas baseados em regras, onde o fluxo de execução não é determinado por uma sequência imperativa de comandos, mas sim pela satisfação de objetivos lógicos e pela unificação de factos.

A. Contextualização

A escolha do paradigma declarativo para a implementação deste sistema prende-se com a natureza do problema: um jogo de conhecimento. Ao contrário dos paradigmas imperativos, que definem **como** atingir um estado, o *Prolog* permite definir **o que** é verdade no sistema (factos) e como essas verdades se relacionam (regras).

O sistema proposto modela o concurso como uma árvore de decisão lógica, onde:

- **O Conhecimento** é estruturado como uma base de dados de factos.
- **O Progresso** é determinado por regras de inferência que validam a veracidade das respostas.
- **As Ajudas** são algoritmos que manipulam listas de opções para reduzir a entropia da decisão.

B. Motivação

A realização deste projeto é impulsionada por vetores técnicos distintos, procurando alinhar a teoria da programação em lógica com a prática de desenvolvimento de software. A escolha de implementar o jogo "*Quem Quer Ser Milionário*" sobre um motor *Prolog* fundamenta-se em três pilares principais:

- **Modelação de Conhecimento:** O formato do jogo, baseado em factos estáticos e validações lógicas, é o candidato ideal para demonstrar a programação orientada ao padrão e a manipulação de estruturas compostas. Permite estruturar bases de conhecimento hierárquicas em detrimento de estruturas de dados convencionais.
- **Aplicação de Lógica Formal:** O projeto serve como prova de conceito para a utilização de regras de inferência clássicas como mecanismos de controlo de fluxo. O objetivo é substituir as estruturas condicionais imperativas pela aplicação explícita de *Modus*.
- **Determinismo vs. Probabilidade:** A integração das "ajudas" motiva a exploração de inferência lógica aplicada a mecanismos de incerteza, como a eliminação de opções ou sugestões baseadas em probabilidade.

C. Desafios de Implementação

A natureza declarativa do *Prolog*, embora poderosa para resolução simbólica, impõe obstáculos significativos no desenvolvimento de aplicações interativas de estado persistente:

- **Gestão de Estado sem Mutabilidade:** Ao contrário das linguagens imperativas, o *Prolog* não permite a alteração direta de variáveis globais. O maior desafio reside na manutenção do "estado do jogo" através de controlo recursivo do fluxo. Cada novo estado deve ser passado como argumento para o predicado seguinte.
- **Abstração das Regras de Inferência:** Um desafio central é implementar as regras lógicas não apenas como teoria, mas como componentes funcionais do código. É necessário garantir que a falácia *Modus Mistaken* seja justificada como um raciocínio intencionalmente falacioso no fluxo de vitória.
- **Integração Multimédia e Interface:** O *Prolog* é nativamente vocacionado para processamento simbólico, não para multimédia. O desafio técnico reside na incorporação de efeitos sonoros e grafismo ASCII num ciclo de execução recursivo, exigindo uma sincronização precisa para garantir uma experiência imersiva sem comprometer a estabilidade do raciocínio lógico.

D. Objetivos do Projeto

O objetivo central é a implementação de um agente lógico robusto que garanta a integridade das regras do jogo. Para tal, o desenvolvimento do sistema estruturou-se em cinco eixos fundamentais:

1) **Arquitetura da Base de Conhecimento:** Estruturação hierárquica de perguntas e níveis de dificuldade, permitindo acesso e validação eficientes.
2) **Motor de Inferência Recursivo:** Implementação do ciclo de jogo através de predicados recursivos que gerem a transição de estados baseada na validação lógica das entradas.
3) **Simulação de Incerteza (Ajudas):** Desenvolvimento de algoritmos para as ajudas (50/50, Público, Telefone) que utilizem manipulação avançada de listas e geração controlada de factos.
4) **Aplicação de Regras Formais:** Integração explícita de regras de inferência (*Modus Ponens* e *Modus Tollens*) como controladores do fluxo narrativo do jogo.
5) **Interface Homem-Máquina:** Criação de uma interface em linha de comandos otimizada, com tratamento de erros e elementos visuais (ANSI e ASCII) que melhorem a usabilidade.

E. Estrutura do Documento

A estrutura remanescente deste documento organiza-se de forma a espelhar o ciclo de desenvolvimento da solução. A Secção II detalha a arquitetura lógica do sistema, dissecando a construção da Base de Conhecimento e a implementação dos mecanismos de inferência e recursividade que sustentam o motor de jogo.

De seguida, a Secção III explora a vertente criativa e de interface, descrevendo a integração de elementos gráficos e sonoros, bem como a lógica algorítmica por detrás das ajudas ao jogador. A validação do sistema é apresentada na Secção IV, onde se documentam os testes realizados e se demonstram os comportamentos do agente perante diferentes cenários de execução.

Por fim, o documento encerra com a Secção V, que oferece uma análise crítica dos resultados obtidos, sintetizando as competências adquiridas e propondo vias de otimização para trabalhos futuros.

II. Estrutura Lógica

A implementação do sistema "*Quem Quer Ser Milionário*" obedece a uma arquitetura modular, desenhada para promover a separação de funcionalidades e facilitar a manutenção do código. Em vez de uma abordagem monolítica, a solução encontra-se segmentada em componentes especializados que comunicam entre si através de predicados bem definidos.

Neste capítulo, dissecamos a engenharia de software da aplicação, analisando como o paradigma declarativo foi aplicado em três vertentes essenciais: a representação estática do conhecimento, o motor de inferência lógica e o controlo de fluxo recursivo. Ao longo desta secção é detalhada a implementação técnica de cada módulo.

A. Base de Conhecimento (`perguntas.pl`)

O ficheiro `perguntas.pl` constitui a camada de persistência estática do sistema. Ao contrário de abordagens imperativas que recorreriam a bases de dados externas, a arquitetura deste projeto tira partido da natureza homoicónica do *Prolog*. O módulo é composto exclusivamente por **factos**, servindo como a verdade imutável que alimenta o motor de inferência.

a) *O Predicado `pergunta/6`:*

O núcleo desta base de conhecimento é o predicado `pergunta/6`. A escolha de uma aridade de 6 permite encapsular, num único termo composto atómico, todo o contexto necessário para a gestão de uma questão.

O predicado é definido da seguinte forma:

```
pergunta(ID,    Enunciado,    [Opcoes],
RespostaCerta, Nivel, Categoria).
```

Onde:

- **ID:** Identificador numérico único para rastreabilidade.
- **Enunciado:** Átomo ou String com o texto da questão.
- **Opcoes:** Lista de strings com as quatro alternativas.
- **RespostaCerta:** Átomo que indica o índice da resposta correta.
- **Nivel/Categoria:** Metadados para filtragem lógica.

b) *Estruturas de Dados: A Lista de Opções:*

A decisão arquitetural mais relevante neste módulo foi o armazenamento das respostas numa **lista** (`['OpA', 'OpB', 'OpC', 'OpD']`) em vez de quatro argumentos individuais. Esta estrutura de dados oferece vantagens algorítmicas cruciais para o motor de jogo:

1) **Manipulação Dinâmica:** Permite que predicados externos apliquem `random_permutation/2` para selecionar aleatoriamente as respostas em tempo de

execução, garantindo que a posição visual da resposta certa varia sem alterar a integridade do facto original.

2) **Filtragem para Ajudas:** Facilita a implementação da ajuda "50/50". O sistema pode gerar, através de recursividade, uma nova sub-lista contendo apenas a resposta certa e uma errada, mantendo a compatibilidade com o predicado de visualização.

c) *Indexação Lógica:*

Os argumentos `Nivel` (1 a 5) e `Categoria` atuam como índices lógicos. Embora o ficheiro `perguntas.pl` não contenha regras de controlo de fluxo, a sua estrutura permite que o motor de inferência utilize *backtracking* para instanciar apenas as perguntas que satisfazem o estado atual do jogador (ex: `pergunta(_, _, _, _, 3, _)` unificará apenas com questões de nível 3).

B. Definição de Normas de Jogo (`regras.pl`)

O ficheiro `regras.pl` desempenha o papel de repositório normativo do sistema. Embora a **execução** destas regras ocorra no motor de jogo, é neste módulo que se encontra a **codificação textual dos parâmetros lógicos** que regem a partida. A sua função estrutural é garantir que o agente humano recebe a definição clara das restrições e objetivos antes do início do ciclo de inferência.

a) *Codificação dos Modos de Jogo:*

O módulo estrutura a apresentação de dois estados de lógica distintos, definindo as condições iniciais para cada um:

1) **Modo Normal:** Definido como a experiência padrão, onde o sistema concede tolerância ao erro (3 Vidas) e suporte à decisão (3 Ajudas).
2) **Modo Hardcore:** Uma variante lógica de "morte súbita", onde as variáveis de auxílio são inicializadas a zero (0 Ajudas) e a tolerância ao erro é nula (1 Vida), terminando a árvore de decisão na primeira falha.

b) *Estrutura de Progressão e Checkpoints:*

O código reflete a estrutura da árvore de decisão do jogo, informando o utilizador sobre:

- **Profundidade:** Uma sequência de 15 níveis de dificuldade incremental.
- **Salvaguarda de Estado:** A existência de *Checkpoints* nos níveis 5 e 10. Estes nós representam pontos de não-retorno lógicos, onde o valor acumulado se torna imutável, independentemente de falhas futuras.

c) *Encapsulamento da Informação:*

Do ponto de vista da engenharia de software, este módulo isola a "lógica de instrução" da "lógica de execução". Ao utilizar predicados de formatação posicional (como `write_at/3`), o sistema garante que estas regras são apresentadas de forma estruturada e imutável.

C. Módulo de Gestão de Visualização (`estudio.pl`)

O módulo `estudio.pl` implementa a camada de visualização do sistema para as perguntas. Do ponto de vista da estrutura lógica, a sua responsabilidade é a conversão de termos abstratos (provenientes da base de conhecimento) em representações concretas no terminal, isolando a complexidade das sequências de controlo ANSI e da manipulação de *streams*.

a) *Orquestração Sequencial de Apresentação:*

O predicado principal exportado, `mostrar_pergunta/5`, não desenha diretamente no ecrã. Em vez disso, atua como um **orquestrador lógico**, invocando uma composição sequencial de predicados atómicos:

```
mostrar_pergunta(...):-cls,
mostrar_background,
desenhar_texto(...), flush_output.
```

Esta estrutura garante que a renderização ocorre por camadas (Limpeza → Fundo → Texto), e a utilização final de `flush_output/0` assegura a sincronização imediata do *buffer* de saída com o terminal do utilizador.

b) *Tratamento de Exceções e Streams:*

A leitura do ficheiro de cenário (`.ans`) demonstra robustez lógica através do mecanismo de tratamento de exceções. O predicado `mostrar_background/0` envolve a abertura do *stream* num bloco `catch/3`:

```
catch((open(...),      ...),      Error,
(print_message(error, Error), ...))
```

Esta abordagem impede que a falha na leitura de um recurso externo termine abruptamente o ciclo de execução do jogo, mantendo a estabilidade do sistema lógico.

c) *Lógica de Posicionamento Relativo:*

Para garantir a simetria visual sem depender de valores absolutos rígidos, o sistema implementa uma lógica aritmética de centralização. O predicado `write_centered/2` calcula dinamicamente a coluna de inserção através da fórmula:

```
Offset is (ImageWidth - TextLen) // 2
```

Esta instrução, implementada com o operador `is/2`, permite que o sistema adapte a renderização a qualquer comprimento de texto (enunciados curtos ou longos).

D. Módulo de Ranking (`ranking.pl`)

O módulo `ranking.pl` implementa a memória de longo prazo do sistema. A sua responsabilidade arquitetural é dupla: assegurar a persistência dos resultados entre sessões de jogo e processar esses dados para apresentar uma tabela de classificação ordenada (*"Hall of Fame"*).

a) *Serialização de Dados em Prolog:*

Ao contrário de sistemas que requerem *parsers* complexos para formatos como JSON ou CSV, este projeto tira partido da homoiconicidade do *Prolog* como referido anteriormente. Os dados são persistidos no ficheiro `ranking.txt` diretamente como termos lógicos válidos.

O predicado `gravar_ranking/5` constrói e anexa um facto ao ficheiro utilizando o modo `append`:

```
format(Stream, "ranking('~w', ~d, ~d,
'~w', '~w', '~d/~d/~d').~n", ...)
```

Esta abordagem permite que a leitura subsequente seja realizada nativamente pelo predicado `read/2`, que interpreta o texto do ficheiro diretamente como código *Prolog*, eliminando a necessidade de análise sintática manual.

b) *Gestão Segura de Recursos (I/O):*

A leitura do histórico demonstra práticas robustas de engenharia de software através do predicado `ler_todos_rankings/1`. O sistema utiliza `setup_call_cleanup/3` para garantir a integridade dos recursos do sistema operativo:

```
setup_call_cleanup(
    open(File, read, Stream, ...),
    ler_termos(Stream, ListaBruta),
    close(Stream)
)
```

Este padrão assegura que o fluxo de leitura é obrigatoriamente encerrado, mesmo que ocorra uma falha ou exceção durante o processamento dos termos, prevenindo fugas de memória ou bloqueios de ficheiros.

c) *Lógica de Ordenação e Processamento:*

Após a leitura, o sistema realiza uma transformação de dados. A lista bruta de termos é mapeada para uma estrutura interna intermédia que facilita a ordenação. A lógica de classificação utiliza o predicado nativo otimizado `sort/4`:

```
sort(3,@>=,ListaProcessada,ListaOrdenada)
```

O argumento `3` instrui o motor a ordenar com base no terceiro elemento da estrutura (o valor do prémio), e o operador `@>=` define uma ordem decrescente, garantindo que as pontuações mais altas ocupam o topo da lista.

d) *Formatação Tabular Dinâmica:*

A camada de visualização deste módulo implementa algoritmos de formatação de texto para manter a integridade da tabela ASCII. O predicado `escrever_formatado/2` verifica o comprimento de cada átomo (nome do jogador, prémio) e aplica dinamicamente truncagem ou preenchimento com espaços, assegurando que as colunas da tabela permanecem perfeitamente alinhadas independentemente do tamanho dos dados inseridos.

E. Módulo de Navegação e Interação (`menu.pl`)

O módulo `menu.pl` constitui a interface de entrada principal do sistema. Do ponto de vista da arquitetura lógica, este componente atua como um controlador de estados de interação, responsável por capturar as preferências do utilizador e encaminhar o fluxo de execução para o motor de jogo.

a) *Máquina de Estados de Leitura:*

A inovação técnica mais relevante neste módulo é a implementação de um mecanismo de **input** natural robusto, capaz de lidar com a tecla de retrocesso (*Backspace*) em tempo real. O predicado `ler_loop_natural/2` implementa uma máquina de estados finita recursiva:

1) **Estado de Acumulação:** Quando um código de caractere gráfico é recebido, ele é empilhado numa lista acumuladora (`[Code|Acc]`) e o caractere é ecoado no terminal.
2) **Estado de Correção:** Se o código corresponde a *Backspace*, o sistema verifica se a lista não está vazia, remove o topo da pilha (`[_|Resto]`) e envia uma sequência de escape visual para apagar o caractere no ecrã.
3) **Estado de Terminação:** A receção do código *Carriage Return* termina a recursão e unifica o resultado com a inversão da lista acumulada.

b) *Abstração de Menus:*

O sistema adota uma abordagem declarativa para a construção de menus. Cada ecrã é encapsulado num predicado independente (ex: `menu_obter_modo/1`). Estes predicados seguem um padrão de projeto consistente:

1) Limpeza do contexto visual (`cls`).
2) Renderização da moldura base (`desenhar_caixa_base/1`).
3) Apresentação das opções específicas.
4) Bloqueio da execução à espera de uma tecla válida (`ler_tecla_simples/2`).

Esta estrutura modular permite a adição de novos menus ou a alteração da ordem de navegação sem impacto na lógica central do jogo.

F. Módulo de Integração Multimédia (`audio.pl`)

O módulo `audio.pl` representa a componente de integração de sistemas do projeto. Dada a ausência de capacidades nativas de processamento de sinal em *Prolog*, este módulo implementa uma ponte lógica para o sistema operativo, permitindo a reprodução de áudio assíncrono através de chamadas de sistema controladas.

a) *Arquitetura de Processos Concorrentes:*

A estrutura lógica deste módulo baseia-se no paralelismo. Se o áudio fosse executado no fluxo principal, o motor de inferência bloquearia à espera do fim de cada som. Para resolver isto, o sistema utiliza a biblioteca `thread`:

```
thread_create( ..., _, [detached(true)])
```

Cada efeito sonoro é lançado numa *thread* independente e "destacada", permitindo que o predicado retorne imediatamente (`true`) e o jogo prossiga enquanto o áudio é reproduzido em segundo plano pelo sistema operativo.

b) *Interoperabilidade com PowerShell:*

A reprodução efetiva é delegada no motor .NET do *Windows* através do *PowerShell*. O *Prolog* constrói dinamicamente um comando de *scripting* que instancia o objeto `Media.SoundPlayer`:

```
process_create(path(powershell), ['-c',
'... .PlaySync()'], ...)
```

Esta abordagem híbrida demonstra a extensibilidade da lógica *Prolog*, capaz de orquestrar recursos externos complexos sem perder o controlo do fluxo da aplicação.

c) *Gestão Dinâmica de Estado:*

Para permitir a interrupção de músicas de fundo, o módulo mantém um registo dinâmico dos identificadores de processo ativos:

1) Ao iniciar um loop, o PID (Identificador de Processo) do processo PowerShell é guardado na base de factos dinâmica (`assertz(pid_musica(Pid))`).
2) Quando é necessário mudar de faixa ou silenciar, o predicado `parar_sons/0` consulta este facto, ter-

mina o processo correspondente (`process_kill/1`) e limpa a memória (`retractall/1`).

G. Módulo de Renderização Sequencial (`animacao.pl`)

O módulo `animacao.pl` implementa um motor de reprodução de vídeo baseado em texto. Devido às limitações de velocidade de I/O num terminal *standard*, a estrutura lógica deste módulo foca-se na otimização de fluxo de dados e na sincronização precisa entre eventos visuais e auditivos.

a) *Otimização de I/O e Renderização:*

A reprodução fluida de animações exige uma abordagem eficiente à escrita no terminal. O predicado recursivo `play_loop/4` implementa duas técnicas de otimização críticas:

1) **Refresh sem Limpeza:** Em vez de limpar o ecrã a cada frame, o cursor é reposicionado na origem e a nova imagem sobrepõe a anterior.
2) **Transferência Binária:** Utiliza-se o predicado `copy_stream_data/2` para transferir o conteúdo do ficheiro diretamente para o *buffer* de saída (`current_output`), evitando o *overhead* de processamento de strings linha a linha.

```
copy_stream_data(Stream,
current_output), flush_output
```

b) *Controlo Temporal Baseado em Dados:*

A lógica de velocidade da animação não é rígida, mas sim definida declarativamente numa tabela de factos `config_anim/3`. Esta estrutura permite ajustar o comportamento do motor para diferentes tipos de eventos:

```
config_anim('intro', 2, 0.036). % Passo
2 (salta frames), Delay 36ms
```

O motor lê estes factos em tempo de execução. O parâmetro "Passo" permite que o sistema salte frames logicamente (`Next is N + Passo`) para acelerar animações pesadas sem perder a sincronização temporal, agindo como uma técnica de *frame skipping*.

c) *Orquestração Multimédia:*

O predicado `apresentar/2` atua como a interface de alto nível, encapsulando a complexidade da sincronização. Este predicado garante que o disparo da *thread* de áudio (via módulo `audio.pl`) ocorre no milissegundo anterior ao início do ciclo de renderização visual, assegurando uma experiência coesa onde o som e a imagem parecem operar como um único fluxo de dados.

H. Módulo de Consultoria Externa (`telefonar.pl`)

O módulo `telefonar.pl` expande a base de conhecimento fechada do sistema, permitindo o acesso a fontes de informação externas. Arquiteturalmente, este componente implementa uma **Foreign Function Interface** (FFI) ad-hoc, criando uma ponte lógica entre o ambiente declarativo do *Prolog* e o ecossistema imperativo do *Python*, necessário para aceder a Grandes Modelos de Linguagem (LLMs).

a) *Ponte entre Paradigmas (Hibridismo):*

O sistema demonstra uma arquitetura de IA Híbrida:

1) **Lógica Simbólica:** O *Prolog* gere o estado do jogo e as regras determinísticas.
2) **Lógica Probabilística:** O *Python* (via *script* `ask_ollama.py`) fornece heurísticas baseadas em redes neuronais para responder a perguntas não codificadas nos factos estáticos.

b) *Comunicação entre Processos (IPC):*

A estrutura lógica baseia-se na criação de processos filhos e na manipulação de canais de comunicação padrão. O predicado `consultar_ia/6` utiliza a biblioteca `process` para invocar o interpretador *Python*:

```
process_create(PathPython,[Script,
Pergunta,Opcoes...],
[stdout(pipe(Out)), stderr(pipe(Err))])
```

Diferente do módulo de áudio, aqui a execução é síncrona: o motor lógico do *Prolog* bloqueia e aguarda que o processo externo retorne uma inferência, capturando o resultado através de *pipes* de sistema.

c) *Serialização e Tratamento de Streams:*

Para garantir a integridade dos dados na transição entre linguagens, o módulo implementa uma gestão rigorosa de *streams*:

- **Encoding:** Força a codificação UTF-8 (`set_stream(..., encoding(utf8))`) para evitar corrupção de caracteres acentuados típicos da língua portuguesa.
- **Gestão de Erros:** O sistema lê separadamente o canal de erro (`stderr`). Se o script *Python* falhar, o *Prolog* deteta o conteúdo no *pipe* de erro e apresenta uma mensagem de falha controlada, impedindo o colapso do jogo principal.

d) *Interface de Espera Ativa:*

Dado que a inferência em LLMs locais pode ter latência variável, o módulo gere o estado de "Espera Ativa", apresentando *feedback* visual ao utilizador ("A contactar o Sabio…") enquanto o processo de fundo realiza a computação pesada, mantendo a responsividade percetível da interface.

I. Módulo de Lógica Probabilística (`ask_ollama.py`)

O *script* `ask_ollama.py` atua como a unidade de processamento neuronal do sistema. Enquanto os restantes módulos operam numa lógica booleana e determinística, este componente é responsável por interagir com o modelo de linguagem *Gemma* (via Ollama), introduzindo uma camada de inteligência probabilística no jogo.

a) *Engenharia de Prompt para Saída Estruturada:*

O maior desafio na integração de LLMs com sistemas lógicos rígidos como o *Prolog* é a imprevisibilidade da saída. Para mitigar isto, o *script* implementa uma técnica de **Prompt Engineering** com restrições fortes:

```
TAREFA: Qual é a opção correta?
REGRA: Responde APENAS com uma única
letra (A, B, C ou D).
```

Esta instrução explícita visa colapsar a "alucinação" ou a verbosidade típica do modelo numa resposta atómica única, compatível com os átomos do *Prolog*.

b) *Extração Determinística de Padrões:*

Mesmo com um *prompt* rigoroso, os modelos de linguagem tendem a gerar ruído. Para garantir a estabilidade do sistema, o *script* aplica uma camada de sanitização baseada em Expressões Regulares (Regex):

```
match = re.search(r'\b([A-D])\b', content)
```

Esta lógica filtra o texto gerado, extraindo apenas a primeira ocorrência isolada de uma letra válida de opção. Se o padrão não for encontrado, o sistema retorna um sinal de erro ("X"), permitindo que o *Prolog* trate a falha graciosamente em vez de tentar processar texto inválido.

c) *Gestão de Recursos de Hardware:*

Sendo um jogo de consola, a eficiência é prioritária. O *script* configura a chamada à API com o parâmetro `keep_alive=0`:

```
response = chat(..., keep_alive=0)
```

Esta instrução força o descarregamento imediato do modelo da VRAM ou RAM após a inferência. Esta decisão arquitetural é crucial para garantir que o jogo principal e as animações mantêm a fluidez, libertando recursos do sistema assim que a "ajuda telefónica" é concluída.

d) *Protocolo de Interface:*

A comunicação com o módulo pai (`telefonar.pl`) é normalizada através de fluxos padrão:

1) **Entrada:** Os argumentos (pergunta e opções) são recebidos via `sys.argv`, atuando como uma ponte de dados simples.
2) **Saída:** A resposta é escrita no `stdout` com codificação forçada em `UTF-8`, prevenindo erros de codificação no *pipe* de leitura do *Windows* que poderiam quebrar a execução do *Prolog*.

J. Motor de Inferência e Ciclo de Jogo (`milionario.pl`)

O módulo `milionario.pl` atua como o núcleo central do sistema, orquestrando a interação entre a base de conhecimento estática e a dinâmica do jogo. A sua arquitetura baseia-se num ciclo de inferência recursivo que mantém a persistência do estado do jogo através da passagem de argumentos.

a) *Gestão de Estado Recursivo:*

Ao contrário das linguagens imperativas que recorrem a variáveis globais mutáveis, o estado do jogo é gerido declarativamente. O predicado principal `ciclo_jogo/10` transporta consigo todo o contexto necessário para a próxima iteração:

```
ciclo_jogo(Nome, [Id|Resto], Nivel, Va, Up, Ajudas, ...)
```

A cada passo de sucesso (resposta correta), o sistema invoca-se a si mesmo (`Next is Nivel + 1`), atualizando os acumuladores (Nível, Prémio) e consumindo a cabeça da lista de perguntas, garantindo assim a progressão linear da partida.

b) *Implementação de Raciocínio Formal:*

O sistema não se limita a comparar *strings*; ele implementa explicitamente regras de lógica formal para validar as respostas, satisfazendo os requisitos teóricos do projeto:

1) ***Modus Ponens*:** Se a entrada do jogador unifica com a resposta certa, deduz-se o progresso.

```
raciocinio_logico(ponens, Input, RespC) :- Input == RespC, implica(..., ...).
```

2) ***Modus Tollens*:** Se a entrada não unifica com a resposta certa, deduz-se a falha.

```
raciocinio_logico(tollens, Input, RespC) :- Input \== RespC, ...
```

3) **Falácia *Modus Mistaken*:** No modo "*Hardcore*", o sistema podia ser introduzido uma verificação probabilística (`random_between`) que simula a "sorte". Mesmo que o raciocínio seja válido, o sistema alerta para a falácia de assumir conhecimento apenas pela observação do resultado positivo.

c) *Algoritmos de Redução de Entropia (Ajudas):*

As ajudas não são meros efeitos visuais, mas sim transformações lógicas sobre o espaço de procura:

- **50/50 (Lógica de Exclusão):** Utiliza o predicado `select/3` e `random_member/2` para construir dinamicamente uma nova lista de opções onde duas respostas incorretas são substituídas por máscaras (`---`), reduzindo a incerteza do agente em 50%.
- **Público (Distribuição):** Utiliza random_between/3 para garantir que a resposta correta (RespC) receba uma percentagem maioritária (entre 45% e 85%), simulando o conhecimento geral do público. O valor restante (100--C) é fragmentado aleatoriamente entre os distratores (R1, R2, R3), e o predicado auxiliar distribuir_votos/9 assegura que a maior percentagem seja mapeada corretamente para a variável correspondente à resposta certa (Pa, Pb, Pc ou Pd).
- **Chamada Telefónica (Integração de LLM):** Através do predicado `consultar_ia/6` (importado de `chamada/telefonar.pl`), o sistema expande a sua base de conhecimento ligando-se a um modelo de linguagem externo (Llama3). Este processo injeta "conhecimento do mundo real" no motor lógico, permitindo sugestões baseadas em inferência semântica e não apenas estatística.

III. Criatividade, Testes e Demonstração

Esta secção apresenta a simulação de um jogo, com ênfase na visualização gráfica do comportamento do sistema. Deste modo, serão demonstradas as principais funcionalidades que atribuem ao projeto a sua vertente criativa, assim como o interface gráfico ou mecânicas extra desenvolvidas.

A. Introdução e Ambiente Gráfico

A execução do predicado de entrada `jogar/0` despoleta, primeiramente, uma sequência de animação introdutória. Esta sequência utiliza o modulo de `animacao.pl` para criar imersão, limpando a consola e apresentando o logótipo animado antes de carregar o menu lógico.

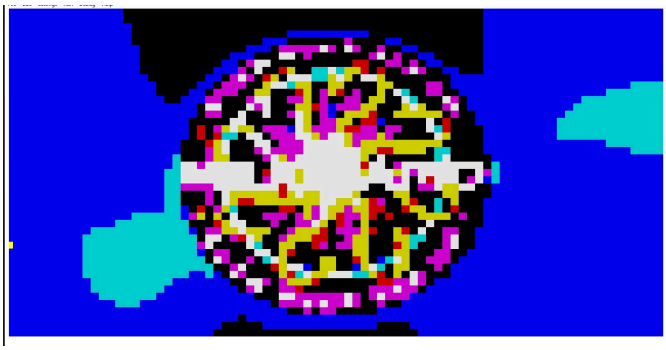

Fig. 1. Animação em *frames* ANSI executada no arranque.

B. Menu Inicial

A navegação inicial do sistema é gerida pelo predicado de menu, que apresenta as opções principais e encaminha o utilizador para o fluxo desejado.

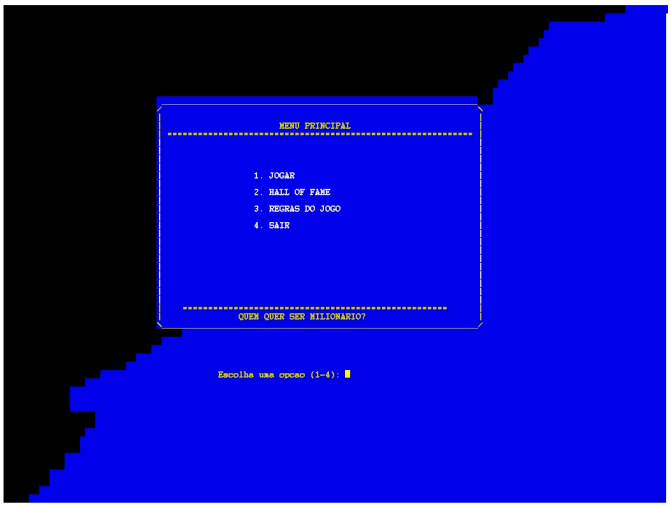

Fig. 2. Menu Principal: O ponto de entrada da aplicação(design em moldura azul e opções de navegação.)

C. Visualização das Regras

Antes de iniciar o jogo, o utilizador pode consultar as regras. A Fig. 3 demonstra a apresentação das instruções, onde são detalhados os modos de jogo (Normal e *Hardcore*), assim como os comandos disponíveis.

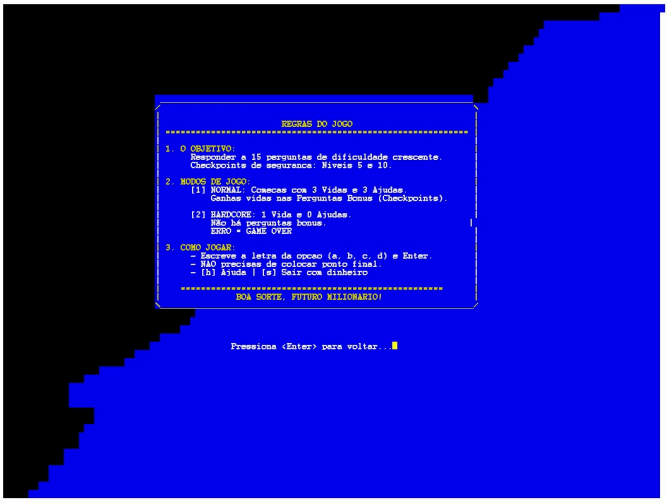

Fig. 3. Ecrã de Regras: Explicação detalhada do funcionamento do jogo.

D. Configuração de Jogo

O fluxo de início de jogo requer uma sequência de configurações que inicializam os factos dinâmicos do sistema.

Em primeiro lugar, é solicitada, ao jogador, a sua identificação para efeitos de registo no Ranking.

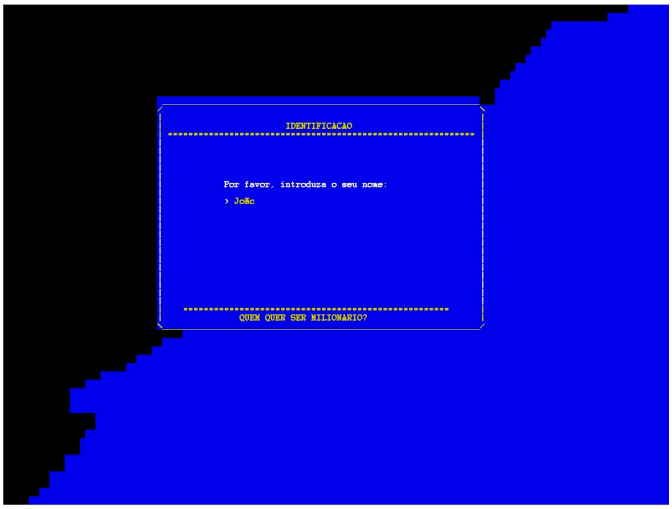

Fig. 4. Identificação do Jogador: Captura do nome para persistência de dados.

Posteriormente, o sistema pede ao utilizador para escolher o modo de jogo, que alterará parâmetros como a dificuldade e a gestão de vidas:

Fig. 5. Seleção de Modo: Escolha entre o modo Normal e *Hardcore*.

Finalmente, a seleção de temas demonstra a capacidade do sistema de filtrar a base de conhecimento de perguntas (`pergunta/6`) de acordo com a categoria escolhida pelo jogador:

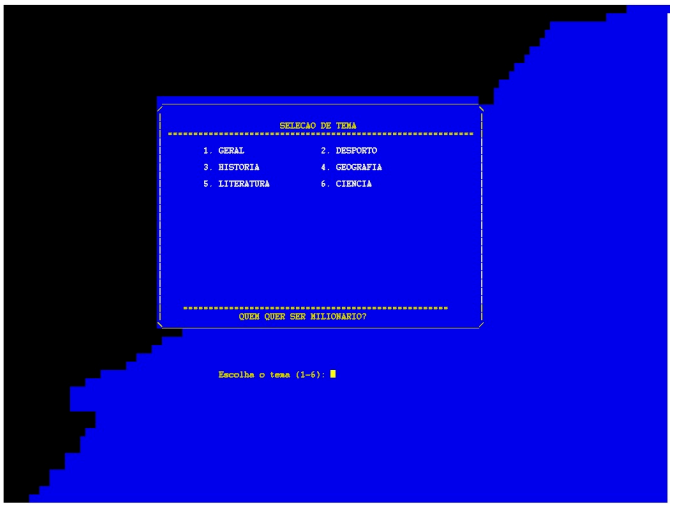

Fig. 6. Seleção de Tema: Menu para filtrar perguntas por categoria.

E. Mecânica de Jogo

Durante a partida, a interface apresenta o HUD (***Heads-Up Display***) com o estado atual do jogo.

a) *Apresentação de Perguntas e Estética Visual:*

As perguntas são carregadas aleatoriamente e apresentadas num **layout** que separa o enunciado das opções de resposta. Para mitigar a aridez típica de uma interface de linha de comandos, foi integrado um avatar gráfico renderizado em caracteres ANSI.

A estética deste avatar foi intencionalmente inspirada nos **sprites** dos treinadores da franquia **Pokémon** (Geração I/III). Como demonstrado na Fig. 7, o motor de renderização desenvolvido em Prolog converteu a **pixel art** original numa matriz de blocos coloridos, preservando a identidade visual da referência nostálgica enquanto se adapta às restrições do terminal.

As perguntas são carregadas aleatoriamente. A figura com perguntas mostra a disposição padrão, onde o sistema aguarda um input válido (A, B, C, D) ou um comando de gestão de jogo (H: Ajuda, S: Sair). Também é exibido um avatar que pretende transmitir algum humanismo ao jogo.

Fig. 7. Adaptação Estética: Interface do jogo em Prolog (topo) e o sprite original de inspiração Pokémon (fundo).

b) *Sistema de Ajudas:*

O sistema de ajudas está desenvolvido de modo a incentivar o espírito estratégico do utilizador, já que são limitadas.

Ajuda 50/50: Ao solicitar a ajuda 50/50, o sistema remove duas opções erradas da lista. A Fig. 8 demonstra o resultado visual, onde as opções "C" e "D" foram ocultadas, facilitando a decisão do jogador.

Fig. 8. Ajuda 50/50: Redução das opções de resposta visíveis.

Ajuda do Público: A ajuda do público gera percentagens simuladas baseadas na dificuldade da pergunta. O teste na Fig. 9 confirma que a soma das percentagens é 100% e mostra a apresentação gráfica integrada no topo do ecrã.

Fig. 9. Ajuda do Público: Percentagens de votação simulada por opção.

Ajuda Telefónica: Aquando da seleção da ajuda telefónica, o sistema conecta-se ao LLM do *Ollama* "O Sábio", onde o utilizador obtém uma resposta, dada pelo modelo, à pergunta. O teste na Fig. 10, mostra o logo do *Ollama* e a conexão ao modelo.

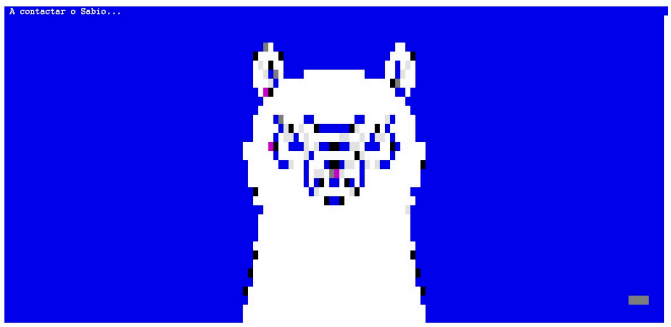

Fig. 10. Ajuda Telefónica: Conexão ao *Ollama*.

c) *Gestão Dinâmica de Vidas e Resiliência:*

Uma das inovações criativas deste sistema face à lógica padrão é a introdução de mecânicas visuais e lógicas para a gestão da "saúde" do jogador no Modo Normal.

Perda de Vida: Quando o motor de inferência deteta uma resposta errada via *Modus Tollens*, mas verifica que `Vidas > 1`, o jogo não termina. Em vez disso, é acionada a animação `perdeu_coracao` e o contador de vidas no HUD é decrementado.

Recuperação de Vida: Caso o jogador acerte na pergunta bónus, o sistema recompensa-o visualmente e logicamente. É instanciada a animação `ganhou_coracao` e o contador de vidas é incrementado na próxima iteração do ciclo recursivo.

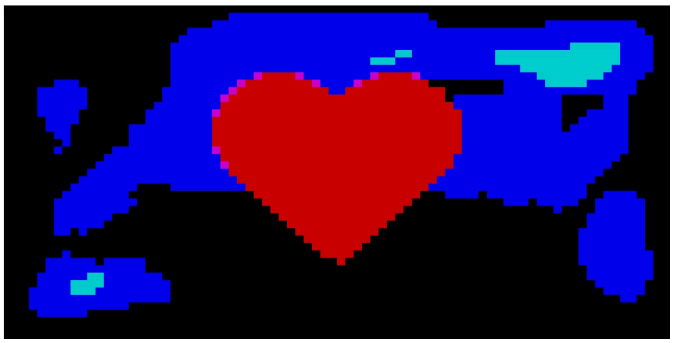

Fig. 11. ANSI do coração.

Mecânica de Redenção (Pergunta Bónus): O sistema implementa uma lógica de "Segunda Oportunidade". Ao ultrapassar um patamar de segurança (Checkpoint), o predicado `pergunta_extra/5` avalia se o jogador possui menos de 3 vidas. Se a condição for verdadeira, o fluxo normal é interrompido para apresentar uma "Pergunta de Bónus".

Esta mecânica demonstra a capacidade do Prolog de alterar dinamicamente a árvore de decisão do jogo, inserindo nós extra de avaliação sem quebrar a recursividade principal.

Fig. 12. Evento Especial: Interface da Pergunta Bónus para recuperação de vida.

d) *Evolução do HUD e Checkpoints:*

Para além das perguntas, o sistema de demonstração permite validar a atualização correta dos valores monetários seguros.

Esta validação experimental confirmou que a lógica de `Seg == 1` no predicado `patamar/4` está a propagar corretamente o valor seguro (`Up`) para as iterações futuras do `ciclo_jogo`.

e) *Estados Finais:*

O jogo termina em dois estados distintos testados:

1) ***Game Over*:** Ocorre quando acabam as vidas ou o jogador erra numa fase crítica (Fig. 13).

Fig. 13. Ecrã de *Game Over*.

2) **Vitória Total:** Ocorre ao responder corretamente à última pergunta, exibindo o prémio máximo (Fig. 14).

Fig. 14. Vitória: 1 Milhão de Euros.

F. Persistência de dados (Ranking)

Finalmente, verifica-se que todos os resultados, sejam desistências, derrotas ou vitórias, são corretamente gravados no ficheiro de texto e apresentados no *Hall of Fame*.

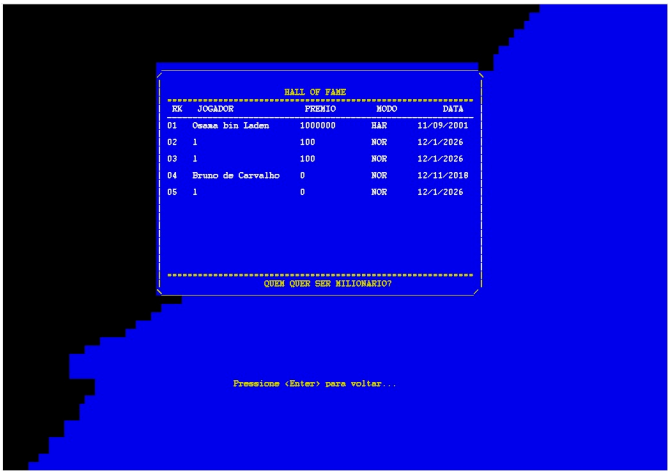

Fig. 15. Ranking Final: Tabela ordenada de pontuações.

IV. Análise Crítica

O desenvolvimento do projeto "Quem Quer Ser Milionário" permitiu consolidar os conceitos fundamentais da programação em lógica, oferecendo um contraste claro com os paradigmas imperativos e orientados a objetos habitualmente utilizados e ainda nos deixou abordar um paradigma híbrido, utilizando uma mistura de LLMs e lógica clássica.

Conclui-se que os objetivos propostos foram plenamente atingidos. O sistema implementado é robusto, não apresenta erros de execução (como *loops* infinitos ou falhas de *backtracking*) e cumpre os requisitos funcionais com um grau elevado de polimento visual e sonoro. Poderia no entanto ter controlo de entradas, ou seja, não permitir qualquer tipo de *input* que não uma permitida para o momento do jogo.

V. Conclusão

Este trabalho demonstrou que o *Prolog*, embora seja uma linguagem específica de domínio e com muitos anos, é extremamente capaz na modelação de sistemas baseados em regras e gestão de conhecimento, provando ser a ferramenta ideal para a lógica deste tipo de jogo, assim como a representação visual através da programação por módulos e construção de uma espécie de *engine*.