Relatório do Projeto prolog GOT
Lógica para Programação

Angel Willyan Rodrigues de Lima[1], Diego Rafael Gomes de França[2], João Lucas Vieira dos Santos[3], Renan Martiniano Santana de Lima[4]

Centro de Informática – Universidade Federal de Pernambuco (UFPE)
[1]awrl@cin.ufpe.br,[2]drgf@cin.ufpe.br,[3]jlvs@cin.ufpe.br,[4]rmsl3@cin.ufpe.br

Resumo.*O sistema geral do funcionamento da lógica como disciplina é compreendida nas questões mais amplas da lógica tradicional utilizada no dia a dia. Tal qual, por exemplo, podemos chegar em deduções, isto é, conclusões por meio de premissas estabelecidas previamentes com o auxílio de regras, como, 'para todo', 'existe', 'ou', 'e', 'se-então', etc. podemos chegar em resultados satisfatórios da lógica que condizem com situações reais do dia a dia. Para entendermos melhor essa questão geral, fizemos um trabalho utilizando a linguagem de programação prolog e a série aclamada de tv 'Game of Thrones'. Nosso objetivo era criar novas regras de acordo com o âmbito da série para concluirmos nossas considerações.*

Palavras-Chave: Programação, Prolog.

1. Introdução

Esse relatório está relacionado ao projeto da cadeira de Lógica para Programação, e foi desenvolvido por quatro alunos do curso de Sistema de Informação da Universidade Federal de Pernambuco(UFPE). Tem como objetivo responder a todas as tarefas que foram propostas nas instruções do projeto. Ao todo são 6 tarefas que serão respondidas imediatamente abaixo. O projeto teve como base a linguagem de programação prolog e a série de tv "Game Of Thrones", Por meio de acontecimentos da série, tais quais, mortes de personagens, relação entre os personagens, famílias, casas, etc. vamos estabelecer regras específicas para retornarem nossas necessidades, por exemplo, se um personagem está morto, se um personagem pertence a uma casa X ou Y. Caso haja uma situação, precisamos verificar se está tudo ocorrendo certo, se o código está retornando um 'false' indevido, etc. tudo isso vamos nos propor a fazer para resolvermos as tarefas propostas no programa.

2. Detalhes da implementação

Tarefa 1: Nossa primeira tarefa foi explicar o uso do precidato "tell_me_about". O predicado tell_me_about(x) retorna todos os status e relacionamentos de um personagem, para isso ele utiliza de outros predicados. Como, o predicado "alive_

or_dead" que recebe uma incógnita, isto é, o nome de um personagem X. e, baseando-se no predicado "status", verifica se um personagem está vivo ou morto com base na lista pré estabelecida. Outro predicado que ele utiliza é o predicado "parents". O predicado parents é curioso, ele utiliza 2 situações na mesma função. A primeira é caso o X escolhido (personagem) tenha pais conhecidos e a segunda é caso ele não tenha. Para isso, ele utiliza-se como base a pré-definição "parent" da lista. Assim, ele verifica todos os pais "disponíveis" para o X. Continuando, o predicado "tell_me_about" também utiliza-se da função "children" para retornar os filhos do personagem (X). O predicado "children" tem a mesma peculiaridade da função "parents", ela tem 2 versões, uma para caso o X tenha filhos e outra para caso ele não tenha retornando none. Para verificar esse status, ele observa se o personagem é pai de alguém, caso ele seja, retorna o "alguém" que ele é pai. Caso não seja, é retornado o none, indicando que o filho não existe. Por último, o predicado "tell_me_about" utiliza o predicado "list_siblings" para retornar a lista dos irmãos de X. O predicado "list_siblings" utiliza a função "siblings" para gerar a lista. Esse predicado "siblings" funciona com base no predicado "sibling". O predicado sibling verifica se tem um Y que tenha os mesmos pais que X através da função "parent". Assim, ele retorna cada irmão/irmã, vai armazenando em uma lista no predicado de "siblings", e essa lista é retornada no predicado "tell_me_about" através do predicado "list_siblings". Quando se é possível retornar uma resposta, mesmo que nula em alguns parâmetros, ele mostra True no final da resposta. No final do código da função é utilizado um “cut” (!), que evita um possível backtracking inútil da função. Já o predicado descendant retorna todos os descendentes do personagem.

Tarefa 2:

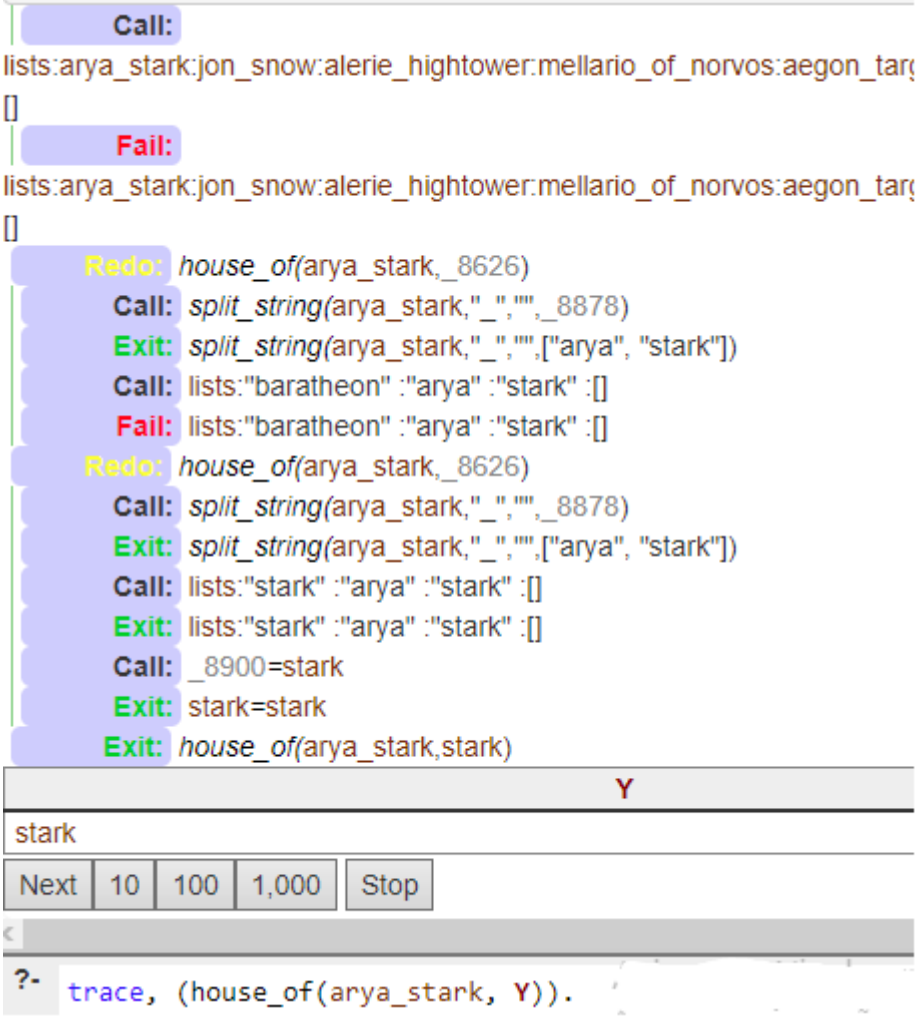
```
clan(jon_snow, stark).
clan(jon_snow, targaryen).
clan(alerie_hightower, tyrell).
clan(mellario_of_norvos, martell).
clan(aegon_targaryen, targaryen).
clan(aegon_targaryen, martell).
clan(rhaenys_targaryen, targaryen).
clan(rhaenys_targaryen, martell).
clan(alannys_harlaw, greyjoy).
clan(gendry, baratheon).

house_of(X, Y) :-
    member(X, [jon_snow, alerie_hightower, mellario_of_norvos, aegon_targary
    ,clan(X, Y)
    ;split_string(X, "_", "", L), member("baratheon", L), Y = baratheon
    ;split_string(X, "_", "", L),member("stark", L), Y = stark
    ;split_string(X, "_", "", L),member("lannister", L), Y =  lannister
    ;split_string(X, "_", "", L),member("targaryen", L),Y = targaryen
    ;split_string(X, "_", "", L),member("martell", L), Y = martell
    ;split_string(X, "_", "", L),member("sand", L), Y = martell
    ;split_string(X, "_", "", L),member("greyjoy", L), Y = greyjoy
    ;split_string(X, "_", "", L),member("tyrell", L), Y = tyrell.
```

A exigência da tarefa 2 era a seguinte: precisávamos fazer um predicado chamado ‘house_of’, onde, ele recebia uma pessoa pré estabelecida, por exemplo, o ‘jon_snow’ e ele recebia um argumento não especificado, por exemplo, um X ou Y. Essa função, então, tem o papel de retornar o clã (ou clãs, em alguns casos).

Para encontrarmos esses casos específicos, utilizamos uma lista pré-estabelecida que contém os seguintes membros [jon_snow, alerie_hightower, mellario_of_norvos, aegon_targaryen, rhaenys_targaryen, alannys_harlaw, gendry]. A escolha desses membros é feita com base na nossa função que pega o nome do clã. No caso, para esse ocorrido utilizamos o “split_string”, ele vai separar a string em uma lista de 2 elementos com base no nome da pessoa. Com isso, ‘arya_stark’ vai se tornar ‘[“arya”, “stark”]. Após isso, usamos a função member para saber se ela está em determinado clã e alocamos o Y como aquele clã.

A lista com os membros específicos serve porque o código normal não se aplica a eles, por exemplo, “jon_snow” não está no clã snow, ele está na casa Stark e na casa Targaryen. Além disso, existem pessoas que estão em 2 casas, como é o caso de “aegon_targaryen” que está na casa Targaryen (como diz o nome dele) e na casa Martell. Nessas situações, criamos um predicado auxiliar anterior chamado “clan” que aponta qual o clã dos personagens específicos da lista.

O funcionamento do código pode ser visto no trace abaixo:



Como podemos ver com o trace do código com a personagem “arya_star”, ele começa verificando se ela está dentro da lista de exceções. Logo após, retorna falso e vai verificar se os nomes dos outros clãs estão na lista dela ["arya", “stark”]). Retornando Fail para o clã Baratheon. Então, o código verifica com a casa stark e “stark” está na lista do nome de Arya. Então, ele continua o código, atribui a Y o valor “stark” e retorna.

Agora vamos ver como o programa age com “jon_snow”:

O programa é bem mais simples porque "jon_snow" está na lista inicial. Então, o código segue outro rumo e vai para o predicado "clan" que retorna jon_snow como Stark e Targaryen.

Tarefa 3:

```
%TAREFA 3
power_of(X, Y):-
        setof(Z,((male(Z);female(Z)), (house_of(Z, Clan),(X = Clan))), Lista),
        length(Lista, Y).
```

A implementação da tarefa 3 é baseada primeiramente na criação de uma lista com os membros do clan que foi pedido em X, para isso usamos o SETOF que funciona na maneira (variável, objetivo, lista) onde se o o objetivo for cumprido, guarda a variável na lista, passando todas as pessoas com a função Z, primeiro checamos se é homem ou mulher(assim obtendo todas pessoas de gênero conhecido), depois, para cada pessoa ele vai pegar o clan dela usando o predicado house_of, que vai retornar em Clan a casa de cada pessoa, e por fim, ele checa se a casa da pessoa checada é a casa que estamos buscando, se for verdade, ele adiciona pessoa na lista, e por fim, ele anexa em Y o tamanho da lista usando a função length

```
   Call: (16) hightower==targaryen ? creep
   Fail: (16) hightower==targaryen ? creep
   Redo: (15) house_of(alerie_hightower, _G3036) ? creep
   Call: (16) hightower==tyrell ? creep
   Fail: (16) hightower==tyrell ? creep
   Redo: (15) house_of(alerie_hightower, _G3036) ? creep
   Call: (16) alerie_hightower==aegon_targaryen ? creep
   Fail: (16) alerie_hightower==aegon_targaryen ? creep
   Redo: (15) house_of(alerie_hightower, _G3036) ? creep
   Call: (16) hightower==snow ? creep
   Fail: (16) hightower==snow ? creep
   Redo: (15) house_of(alerie_hightower, _G3036) ? creep
   Call: (16) alerie_hightower==gendry ? creep
   Fail: (16) alerie_hightower==gendry ? creep
   Redo: (15) house_of(alerie_hightower, _G3036) ? creep
   Call: (16) alerie_hightower==rhaenys_targaryen ? creep
   Fail: (16) alerie_hightower==rhaenys_targaryen ? creep
   Fail: (15) house_of(alerie_hightower, _G3036) ? creep
^  Exit: (8) setof(_G3028, user: ((male(_G3028);female(_G3028)), ho
l, margaery_tyrell, olenna_tyrell]) ? creep
   Call: (8) length([alerie_hightower, loras_tyrell, luthor_tyrell,
   Exit: (8) length([alerie_hightower, loras_tyrell, luthor_tyrell,
 T Exit: (7) power_of(tyrell, 6)
   Exit: (7) power_of(tyrell, 6) ? creep
Y = 6.
```

Assim mostram os resultados com o trace ligado, assim como a resposta final do "power_of(tyrell, Y)" usado de exemplo, o trace nos mostra o passo a passo a checagem com todos personagens, até retornar o resultado final Y.

Tarefa 4:

```
is_single(X) :-
        not(parent(X, Y)),
        Y = none.
```

A implementação da tarefa 4 é bem simples, a função PARENT retorna false quando X não tem filhos, então basta usar um NOT antes da chamada de função para inverter os resultados, retornando true quando X não tem filhos.

```
[trace] 29 ?- is_single(jon_snow).
   Call: (7) is_single(jon_snow) ? creep
^  Call: (8) not(parent(jon_snow, _G8351)) ? creep
   Call: (9) parent(jon_snow, _G8351) ? creep
   Fail: (9) parent(jon_snow, _G8351) ? creep
^  Exit: (8) not(user:parent(jon_snow, _G8351)) ? creep
   Call: (8) _G8351=none ? creep
   Exit: (8) none=none ? creep
   Exit: (7) is_single(jon_snow) ? creep
true.
```

Esse é o resultado com o Trace ligado, retornando true quando checamos se o jon snow não tem filhos.

Tarefa 5:

```
list_tios(Uncles, Y) :-
    setof(Z, uncle(Z,Y), Uncles);
    Uncles = [].

list_tias(Aunts, Y) :-
    setof(Z, aunt(Z,Y), Aunts);
    Aunts = [].

list_sobrinhos(Nephews, Y) :-
    setof(Z, nephew(Z,Y), Nephews);
    Nephews = [].

list_sobrinhas(Neices, Y) :-
    setof(Z, neice(Z,Y), Neices);
    Neices = [].

verifica_clan(Clan, Pessoa ,List) :-
    cac = [],
    house_of(Pessoa, X), house_of(Clan, Y), dif(X, Y), append(clan, cac, list).

marriage_power(X, Y, Z):-
    is_single(X), is_single(Y), male(X), female(Y)
    ancestors(Y, A), verifica_clan(A, X, L), write(L),
    list_siblings(Y, B),
    list_tias(C, Y),
    list_tios(D, Y),
    list_sobrinhos(E, Y),
    list_sobrinhas(F, Y);
    is_single(X), is_single(Y), female(X), male(Y)
    ancestors(Y, A),
    list_siblings(B, Y),
    list_tias(C, Y),
    list_tios(D, Y),
    list_sobrinhos(E, Y),
    list_sobrinhas(F, Y).
```

Apesar de não conseguirmos completar a tarefa 5, a ideia primária era estabelecer 2 condições. A primeira é se ambos são solteiros com X masculino e Y feminino. E a segunda era ambos serem solteiros com Y masculino e X feminino. A ideia era ambos utilizarem as mesmas funções. No caso, as funções de retornar listas.

As funções de retorno de listas estão corretas, onde, a “list_tios” retorna uma lista com todos os tios de Y, a “list_tias” retorna uma lista com todas as tias e assim sucessivamente. A ideia era verificar se cada lista tinha membros que pertenciam ao mesmo clã de X, pois, caso pertencessem, eles não seriam adicionados no final. Devido ao tempo e complicações durante a elaboração do projeto, com a sintaxe da linguagem, não foi possível finalizar essa ideia.

Tarefa 6:

A tarefa 6, infelizmente, também foi uma pendência para todos do grupo. Não conseguimos estabelecer uma lógica para fazer as devidas aplicações. Isso se deve ao

fato de todos os membros estarem trabalhando em outro projeto mais criterioso, sobrando pouco tempo para o projeto prolog. Diferentemente da tarefa 5 a gente

3. Conclusão

A partir desse projeto, foi-se possível ter conhecimentos básicos sobre a linguagem de programação prolog. Apesar das dificuldades estabelecidas, tais quais, como fazer uma espécie de “if else” ou a análise de listas, conseguimos estabelecer uma excelente abordagem ao lidar com as tarefas apontadas. Sem sombra de dúvidas, a maior dificuldade foi a sintaxe da linguagem. Apesar das pendências das tarefas 5 e 6, a ideia por trás do sistema geral de lógica estabelecido nas aulas, pôde ser visto durante a abordagem do código. Esse conhecimento com certeza será melhor aproveitado durante os próximos semestres e ficará durante um bom tempo na mente de todos os envolvidos na elaboração do projeto.

4. Bibliografia

https://www.freecodecamp.org/news/how-to-learn-prolog-by-watching-game-of-thrones-4852ea960017

https://www.swi-prolog.org/pldoc/doc_for?object=maplist/2

https://stackoverflow.com/questions/14743005/how-to-check-if-each-element-in-prolog-list-is-greater-than-0

https://www.ic.unicamp.br/~meidanis/courses/mc336/problemas-prolog/

https://www.educba.com/prolog-not-equal/

https://www.tutorialspoint.com/prolog/prolog_lists.htm

https://pt.wikibooks.org/wiki/Prolog/Listas