

Inferere

motor de inferência + tradução clausal
(c) Artur Marques, 2000

Índice

0 – Introdução

→ página [3]

1 - Escolha de um modelo para representação de fórmulas bem formadas [fbfs], na Lógica de Predicados.

→ páginas [4 – 6]

2 - escolha de um modelo para representação de cláusulas, na forma normal.

→ páginas [7 – 8]

3. implementação dos algoritmos de verificação de conformidade das fórmulas e das cláusulas, relativamente à representação escolhida.

→ página [9]

4. implementação dos algoritmos de tradução das fórmulas, para a forma clausal.

→ página [10]

5. escolha de um modelo de representação para a base de conhecimentos [*Knowledge Base / KB*], sobre a qual trabalhará o motor de inferência.

→ página [11]

6. implementação de um algoritmo de resolução para a frente, sobre as representações feitas.

→ página [12-17]

Guia para Utilização Imediata

→ página [18]

O - Introdução

A implementação de um motor de inferência constitui um problema muito interessante. Apesar de existirem algoritmos de alto nível, que descrevem com todo o detalhe, diversas formas de inferir novo conhecimento, a verdade é que – como em tudo – só passando pelos desafios e recompensas da sua implementação prática, em alguma ferramenta, é que se percebe o que está subjacente a muitos raciocínios...

O projecto «*Infer*» pode considerar-se o produto final das seguintes etapas incrementais:

- escolha de um modelo para representação de fórmulas bem formadas [fbfs], na Lógica de Predicados.
- escolha de um modelo para representação de cláusulas, na forma normal.
- implementação dos algoritmos de verificação de conformidade das fórmulas e das cláusulas, relativamente à representação escolhida.
- implementação dos algoritmos de tradução das fórmulas, para a forma clausal.
- escolha de um modelo de representação para a base de conhecimentos [*Knowledge Base / KB*], sobre a qual trabalhará o motor de inferência.
- implementação de um algoritmo de resolução para a frente, sobre as representações feitas.
- implementação de uma interface com o utilizador, suficientemente expressiva para que seja possível perceber todo o processamento que aconteceu, durante o funcionamento do algoritmo de inferência.
- comentar efusivamente todo o código fonte do programa, de forma a que constitua uma documentação interna que facilite a manutenção e eventual continuação do projecto.

*Se é um utilizador desejoso de experimentar o motor de inferência, dispensando a documentação que se segue, salte para a última página, onde pode encontrar um **guia para utilização imediata**.*

1 - Escolha de um modelo para representação de fórmulas bem formadas [fbfs], na Lógica de Predicados.

Uma vez que o projecto «*Inferre*» foi implementado em LISP, fazia todo o sentido optar por uma representação das fbfs, em lista, mercê das muitas funções disponíveis, para manipulação simbólica de listas.

A representação que se escolheu considera que:

- são binários os operadores lógicos de *implicação*, *e*, e *ou*, e os quantificadores existencial e universal.
- é unário o operador lógico de negação.
- um operador deve ser expresso antes dos seus operandos.
- todas as expressões deverão ser escritas na forma:

(operador operando[s])

Por exemplo:

(and a b) representa (a and b)
(and (and a b) c) representa ((a and b) and c)

- os quantificadores não são excepção a esta regra, se os entendermos como operadores e se aceitarmos que a variável que quantificam é como se fosse um argumento seu. Por exemplo:

(all x (and (p x) (q x)))

representa, numa notação talvez mais familiar

all x ((p(x) and q(x))

De início, o projecto «*Inferre*» investiu muito tempo numa tentativa de permitir que o utilizador final expressasse as fórmulas da maneira a que, eventualmente, estará habituado a fazê-lo, quando tem necessidade de trabalhar com lógica de predicados noutras situações. Desenvolveu-se mesmo uma modesta rede semântica, que pretendia representar toda a informação expressa, e mais tarde perceber a partir dela, o que eram predicados, funções, variáveis, etc... naquilo que o utilizador escrevia. A implementação dessas facilidades cedo se provou insensata, face ao Tempo disponível... mas a rede semântica é um anexo curioso deste projecto.

A melhor maneira de compreender a notação que se escolheu para as fbfs, é observando alguns exemplos. De notar que o utilizador deverá escrever as suas fbfs, utilizando esta notação, confortável... para o programa...

$\forall d(\text{facil}(d) \Rightarrow (\exists e(\text{inscrito}(e,d) \wedge (\text{contente}(e))))$

...deverá ser expressa, como se segue:

(all d (implies (facil d) (exists e (and (inscrito e d) (contente e))))))

$\forall d \forall e (\text{exame}(d) \wedge \text{inscrito}(e, d) \Rightarrow \neg(\text{contente}(e)))$

...deverá ser expressa, como se segue:

(all d (all e (implies (and (exame d) (inscrito e d)) (not (contente e))))))

$\neg(\forall d \text{exame}(d) \Rightarrow \neg \text{facil}(d))$

...deverá ser expressa, como se segue:

(not (all d (implies (exame d) (not (facil d))))))

Os ficheiros `predicados.lsp` e `termos.lsp`, contêm todo o código relevante para a verificação sintáctica de fbfs. Quando um utilizador tentar escrever algo que não é uma fbfs, a sua expressão será ignorada.

O que se entende por termos e fbfs, é algo que está para lá do propósito deste documento. Recomendam-se os acetatos de «Fundamentos de Inteligência Artificial», por Luís Custódio, que apresentam de forma sucinta, mas completa, todas as regras e algoritmos necessários para a verificação da expressão.

Na prática, a função mais importante que resulta dos dois ficheiros `.LSP` referidos é:

(fbf_p expressao)

que retorna NIL se expressão não for uma fbfs...
... ou T, caso contrário.

O projecto foi desenvolvido de uma forma incremental, de forma a que construções mais complexas pudessem *sempre* ser construídas, re-utilizando construções prévias, menos complexas. Assim, apesar de se fazer menção à função **(fbf_p expressao)**, ela não deverá ser utilizada pelo utilizador final do programa, para o qual será suficiente – conforme se detalha na secção devida – saber utilizar a função **(infere)**, para conseguir utilizar o software!

Repete-se, a documentação técnica do projecto, são os próprios [abundantes] comentários ao código fonte.

O utilizador do programa, deverá estar consciente de que a verificação da correcção das fórmulas é muito rigorosa, o que significa que deverão estar devidamente identificadas as entidades que se consideram constantes, variáveis, funções e predicados.

Qualquer nome que o utilizador quiser, pode ser utilizado para exactamente um propósito. Por exemplo *pluto*, poderá ser ou uma constante, ou uma função, mas nunca as duas coisas em simultâneo.

O ficheiro **user.lsp**, declara as listas onde o utilizador DEVE declarar que entidades são o quê, para o seu problema concreto. O utilizador deve ainda ter o cuidado de não utilizar nomes que possam ser gerados automaticamente pelo programa, durante o processo de tradução para a forma clausal.

Por exemplo, por defeito, o utilizador não deverá usar o nome VAR0, porque é um nome potencialmente gerável de forma automática, durante o funcionamento do software.

Os «nomes automáticos» e os próprios símbolos para *todos* os operadores, estão à escolha do utilizador... O ficheiro **globais.lsp**, declara algumas entidades de âmbito «global», por conveniência. Eis um aspecto *possível* para o ficheiro **globais.lsp**:

```
(setq *simbolo_exists* 'exists)
```

```
(setq *simbolo_all* 'all)
```

```
(setq *simbolo_implies* 'implies)
```

```
(setq *simbolo_or* 'or)
```

```
(setq *simbolo_and* 'and)
```

```
(setq *simbolo_not* 'not)
```

```
(setq *default_name_for_new_consts* 'new_cte)
```

```
(setq *default_name_for_new_funcs* 'new_func)
```

```
(setq *default_name_for_new_vars* 'var)
```

```
(setq *kb* nil) ; a KB
```

```
(setq *fkb* nil) ; as fórmulas, tal qual escritas no ficheiro original, sem tradução
```

Admite-se que o utilizador tem os conhecimentos LISP suficientes para ajustar o programa aos seus gostos, no que toca à liberdade que estas variáveis permitem.

Algumas notas: quando o programa precisa de uma nova constante, gera-a usando o nome base por defeito – NEW_CTE, neste exemplo – sufixando-o com um número, de forma a que a entidade resultante seja original; o nome gerado será automaticamente acrescentado à lista de constantes. O mesmo se passa para as variáveis e funções, mas o ficheiro **globais.lsp** nunca é modificado – as modificações só acontecem em *run-time*, na memória volátil.

2 - escolha de um modelo para representação de cláusulas, na forma normal.

Outra grande fatia do tempo investido no projecto, coube ao módulo de tratamento e tradução de fbfs, para cláusulas. Este módulo está implementado no ficheiro **clausulas.lsp**.

Seguiu-se exactamente a mesma estratégia que para a representação de fbfs: uma cláusula deverá ser uma lista!

Uma vez que se pretende que o programa trabalhe sobre cláusulas na «forma normal», em que basicamente não são explícitos quaisquer operadores, excepto a negação, e está implícita a conjunção de disjunções, o desafio que se colocou, foi como é que utilizando apenas listas, se conseguiria fazer isso.

A representação pela qual se optou fundamenta-se no seguinte:

- as cláusulas que correspondam à tradução de uma fbf, deverão ficar contidas numa só lista...
- ...essa lista deverá ter tantas sub-listas, quantas as conjunções (ANDs) na fbf original...
- ... e cada uma dessas sub-listas, terá tantas sub-listas, quantos os ORs que expressem.
- no momento da introdução das cláusulas na base de conhecimento, uma expressão que, por tradução, tenha gerado n cláusulas, produz n entradas.

Mais uma vez, acreditando na adequação dos exemplos, eis a representação que se faz, para as fbfs mencionadas em 1.:

FBF #1:

(ALL D (IMPLIES (FACIL D) (EXISTS E (AND (INSCRITO E D) (CONTENTE E))))))

FBF #2:

(ALL D (ALL E (IMPLIES (AND (EXAME D) (INSCRITO E D)) (NOT (CONTENTE E))))))

FBF #3:

(NOT (ALL D (IMPLIES (EXAME D) (NOT (FACIL D))))))

CLÁUSULA #1: (obtida por tradução de FBF #1)

((NOT (FACIL VAR5)) (INSCRITO (NEW_FUNC1 VAR5) VAR5))

CLÁUSULA #2: (obtida por tradução de FBF #1)

((NOT (FACIL VAR6)) (CONTENTE (NEW_FUNC1 VAR6)))

CLÁUSULA #3: (obtida por tradução de FBF #2)

((NOT (EXAME VAR7)) (NOT (INSCRITO VAR8 VAR7)) (NOT (CONTENTE VAR8)))

CLÁUSULA #4: (obtida por tradução de FBF #3)

((EXAME NEW_CTE0))

CLÁUSULA #5: (obtida por tradução de FBF #3)

((FACIL NEW_CTEO))

Os números que sufixam os nomes automáticos podem variar, de execução para execução, consoante o que se tenha feito com o programa.

3. implementação dos algoritmos de verificação de conformidade das fórmulas e das cláusulas, relativamente à representação escolhida.

Conforme documentado em comentário no código fonte, a verificação da conformidade do que o utilizador escreve, com as representações que o programa aceita, acontece recursiva e incrementalmente.

Por exemplo, eis a função que verifica se uma expressão AND é uma FBF, de acordo com `fbf_p`:

```
(defun and_fbf_p (e)
  (if
    ; todas as fbf têm de estar em notação de lista...
    (listp e)
    ; e uma and_fbf tem de começar pela notação para o and...
    ; aplicada a duas fbfs
    (and
      (= (length e) 3)
      (simbolo_and_p (car e))
      (fbf_p (cadr e))
      (fbf_p (caddr e))
    ); and
    ; caso contrário...
    nil
  ); if
); and_fbf_p
```

Os detalhes de implementação de todos os algoritmos são apenas mencionados no próprio código fonte.

4. implementação dos algoritmos de tradução das fórmulas, para a forma clausal.

Eis a função principal de **clausulas.lsp**, que traduz uma fbf para a forma clausal:

```
(defun traducaao_clausal (fbf)
  (cond
    (
      (fbf_p fbf)

      (setq fbf (distribuir_implies fbf))
      (setq fbf (distribuir_negacao fbf))
      (setq fbf (eliminar_exists_nao_univs fbf))
      (setq fbf (eliminar_exists_univs fbf))
      (setq fbf (eliminar_univs fbf))
      (setq fbf (distribuir_or fbf))
      (setq fbf (rep_conjs_and fbf))
      (setq fbf (corrige_todos_and_nested fbf))
      (setq fbf (rep_conjs_or fbf))
      (setq fbf (novas_vars_em_todas_as_clausulas fbf))
      fbf
    )
    (
      ; não recebi uma fbf : não fazer nada
      t
      nil
    )
  )
); cond
); traducaao_clausal
```

Em boa verdade, a função é um pouco maior, uma vez que, na sua versão final, pode receber um argumento opcional que, caso activo, faz com que cada um dos passos de tradução, dê *feedback* da representação intermédia a que chega.

Exceptuando o que se escreve / não se escreve, o único detalhe menos óbvio da função é uma chamada a **(corrige_todos_and_nested fbf)**, motivada pelo tratamento binário que se dá ao operador de conjunção lógica – AND. A representação que se faz das fbfs, pode obrigar na tradução para a forma clausal a um tratamento mais cuidado das sub-listas, daí a chamada a esta função.

5. escolha de um modelo de representação para a base de conhecimentos [Knowledge Base / KB], sobre a qual trabalhará o motor de inferência.

A KB é representada... como uma lista de listas, que representam cláusulas. Nada mais. Simples & efectivo.

As funções estritamente relacionadas com a KB, estão no ficheiro **kb.lsp**.

A função **(ler_kb)** representa na KB, todas as fbfs que estejam escritas num ficheiro que é ***default_file_name_for_kb_file***, por defeito.

Se o utilizador quiser aceder directamente a esta função para formar uma KB, poderá – opcionalmente – fornecer o nome do ficheiro que contém as suas expressões. Por exemplo, se o ficheiro **C:\KB.TXT** contiver uma KB sobre a qual se queira fazer inferência, bastará invocar:

```
(ler_kb "C:\\kb.txt")
```

...é preciso ter em atenção a forma de indicar *paths*, no sistema operativo que se utilize. Este é, aliás, o único detalhe de portabilidade que merece referir. Todo o código é altamente independente do dialecto LISP que se utilize, tendo sido testado com sucesso com os interpretadores *XLISP*, versões 2 [com 10 anos!] e 3, e *Allegro LISP*, versão 5.

Insiste-se que um utilizador final não precisa de aceder a esta função. A função **(infere)** e saber utilizar **user.lsp**, são elementos suficientes para a esmagadora maioria das necessidades que podem surgir.

6. implementação de um algoritmo de resolução para a frente, sobre as representações feitas.

O algoritmo de resolução utilizado, prova a consistência das informações em KB, ou a sua inconsistência, chamando a atenção para a falsidade de uma fórmula. Este comportamento pode ser utilizado para fazer demonstrações, por redução ao absurdo.

Se o utilizador começa por ter uma KB consistente, com informação que se tem como verdadeira, o programa deverá informar sobre isso mesmo. Mas se se introduzir em KB algo de inconsistente, o motor de inferência gerará a certa altura uma cláusula vazia que representa uma contradição a que se chega, e que é suficiente para concluir da verdade da negação da cláusula em causa, relativamente à KB em uso.

O processo obedece ao seguinte algoritmo:

- traduzir todas as fbfs para a forma clausal
- produzir uma lista de todos os símbolos de proposição, em todas as fórmulas que são potencialmente unificáveis com outros símbolos de proposição, noutras fórmulas.
- gerar as novas cláusulas que resultam da resolução dos elementos «efectivamente» unificáveis. Estas novas cláusulas são introduzidas na KB, se forem originais, isto é, se já lá não estiverem.
- investigar se foi produzida a cláusula vazia OU se a KB não se alterou. Se apareceu a cláusula vazia, isso pode significar que se demonstrou uma fbf, pela introdução da sua negação em KB; se a KB não se alterou, isso significa que, a partir das cláusulas presentes, não se conseguem gerar novidades, pelo que a KB é consistente.
- para o caso de não se verificar nenhum dos acontecimentos anteriores, repetir o terceiro passo...

A inferência faz-se pela função **(infere)**, de muito alto nível, com a seguinte implementação:

```
(defun infere (&optional (nome_ficheiro *default_file_name_for_kb_file*))
```

```
  (ler_kb nome_ficheiro)           ; ler a KB de um ficheiro

  (prog
    (
      (condicao1 nil)                ; KB com cláusula nil
      (condicao2 nil)                ; geração que não produz novas cláusulas
    )

    (do
      (
        (novas nil)
        (estado_ant nil)
        (iteracao 1)
      ); inicializações

      (
```

```

                (or condicao1 condicao2)
); condição de paragem

(escreve_kb)
(gera_novas_clausulas_from *KB*) ; acção

; cláusula vazia => contradição
(setq condicao1 (empty_clause_in_kb *KB*))

; não houve geração de novas cláusulas
(setq condicao2 (= (length *KB*) (length estado_ant)))

; guardar o estado anterior, para ver se há alterações
(setq estado_ant *KB*)
); do

(if
    condicao1
    (princ "KB inconsistente!")
    (princ "KB consistente!"))
); if
(return t)
); prog
); infere

```

A função **(gera_novas_clausulas_from *KB*)** é o coração do algoritmo de resolução. Esta função faz inferência a partir do ponto de partida que receber como argumento, e vai acrescentando as novas cláusulas a que chegar, à KB.

A função **(gera_novas_clausulas_from *KB*)** pode ainda receber um último argumento – que é T, por defeito – e que controla se há-de, ou não, de funcionar em modo informativo.

O funcionamento do algoritmo de resolução em modo informativo, consiste na indicação de que cláusulas foram usadas para inferir novas cláusulas, através de que substituições, e por que ordem. A observação do feedback de **(gera_novas_clausulas_from *KB*)** é suficiente para se ter acesso a toda a informação que a função está a utilizar internamente [na forma de listas...], mas de uma forma muito mais compreensível.

A cada chamada, **(gera_novas_clausulas_from *KB*)** retorna a lista das novas cláusulas geradas, com informação extra sobre que unificações aconteceram, para cada cláusula em particular. O ficheiro **resol.lsp** implementa o conjunto das funções mais directamente relacionadas com a resolução.

Para se perceber porque é que – efectivamente! – basta usar (infere) para experimentar o motor de inferência desenvolvido, eis alguns exemplos, em modo informativo, que ilustram, com todo o detalhe, aquilo que se passa:

Exemplo #1 – KB lida de KB2.TXT, que tem o seguinte aspecto:

```

KB2.TXT
(all x1 (implies (cao x1) (saliva x1)))
(cao pluto)
(not (saliva pluto))

```

Este exemplo pretende mostrar que é inconsistente dizer que:

- todos os cães salivam
- pluto é cão
- pluto não saliva

> (infere "Kb2.txt")

```
FORMULA #1= (ALL X1 (IMPLIES (CAO X1) (SALIVA X1)))  
FORMULA #2= (CAO PLUTO)  
FORMULA #3= (NOT (SALIVA PLUTO))  
INÍCIO DA INFERÊNCIA...
```

ITERACAO #1

```
CLÁUSULA #1: ((NOT (CAO VAR9)) (SALIVA VAR9))  
CLÁUSULA #2: ((CAO PLUTO))  
CLÁUSULA #3: ((NOT (SALIVA PLUTO)))
```

NOVA CLÁUSULA!! => ORIGEM (1 , 3)
CLÁUSULA= ((NOT (CAO PLUTO)))
Substituições: VAR9 / PLUTO ;

NOVA CLÁUSULA!! => ORIGEM (2 , 1)
CLÁUSULA= ((SALIVA PLUTO))
Substituições: VAR9 / PLUTO ;

ITERACAO #2

```
CLÁUSULA #1: ((NOT (CAO VAR9)) (SALIVA VAR9))  
CLÁUSULA #2: ((CAO PLUTO))  
CLÁUSULA #3: ((NOT (SALIVA PLUTO)))  
CLÁUSULA #4: ((NOT (CAO PLUTO)))  
CLÁUSULA #5: ((SALIVA PLUTO))
```

NOVA CLÁUSULA!! => ORIGEM (5 , 3)
CLÁUSULA= NIL
Substituições:

KB inconsistente!! => a cláusula representa uma FALSIDADE.
T

Conforme se lê, cada geração de cláusulas consiste numa iteração do algoritmo que pode resultar em nova informação para a KB; a partir dessa informação, pode eventualmente gerar-se novo conhecimento, e assim sucessivamente, até se estagnar, ou até se encontrar uma inconsistência.

Eis um exemplo que prova a consistência da KB anterior, desde que não se diga que pluto não saliva ☺:

Exemplo #2 – KB lida de KB3.TXT, que tem o seguinte aspecto:

KB3.TXT

```
(all x1 (implies (cao x1) (saliva x1)))  
(cao pluto)
```

> (infere "kb3.txt")

```
FORMULA #1= (ALL X1 (IMPLIES (CAO X1) (SALIVA X1)))  
FORMULA #2= (CAO PLUTO)  
INÍCIO DA INFERÊNCIA...
```

ITERACAO #1

```
CLÁUSULA #1: ((NOT (CAO VAR8)) (SALIVA VAR8))  
CLÁUSULA #2: ((CAO PLUTO))
```

NOVA CLÁUSULA!! => ORIGEM (2 , 1)
CLÁUSULA= ((SALIVA PLUTO))

Substituições: VAR8 / PLUTO ;

ITERACAO #2

CLÁUSULA #1: ((NOT (CAO VAR8)) (SALIVA VAR8))

CLÁUSULA #2: ((CAO PLUTO))

CLÁUSULA #3: ((SALIVA PLUTO))

KB consistente!! => as cláusulas presentes não se contradizem

T

Por fim, eis um exemplo mais complexo, que prova que se uma disciplina tem exame final, então não é fácil, isto para a respectiva KB:

Exemplo #3 – KB lida de KB4.TXT, que tem o seguinte aspecto:

KB4.TXT

(all d (implies (facil d) (exists e (and (inscrito e d) (contente e))))))

(all d (all e (implies (and (exame d) (inscrito e d)) (not (contente e))))))

(not (all d (implies (exame d) (not (facil d))))))

> (infere "kb4.txt")

FORMULA #1= (ALL D (IMPLIES (FACIL D) (EXISTS E (AND (INSCRITO E D) (CONTENTE E))))))

FORMULA #2= (ALL D (ALL E (IMPLIES (AND (EXAME D) (INSCRITO E D)) (NOT (CONTENTE E))))))

FORMULA #3= (NOT (ALL D (IMPLIES (EXAME D) (NOT (FACIL D))))))

INÍCIO DA INFERÊNCIA...

ITERACAO #1

CLÁUSULA #1: ((NOT (FACIL VAR4)) (INSCRITO (NEW_FUNC1 VAR4) VAR4))

CLÁUSULA #2: ((NOT (FACIL VAR5)) (CONTENTE (NEW_FUNC1 VAR5)))

CLÁUSULA #3: ((NOT (EXAME VAR6)) (NOT (INSCRITO VAR7 VAR6)) (NOT (CONTENTE VAR7)))

CLÁUSULA #4: ((EXAME NEW_CTE1))

CLÁUSULA #5: ((FACIL NEW_CTE1))

NOVA CLÁUSULA!! => ORIGEM (2 , 3)

CLÁUSULA= ((NOT (FACIL VAR5)) (NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)))

Substituições: VAR7 / (NEW_FUNC1 VAR5) ;

NOVA CLÁUSULA!! => ORIGEM (4 , 3)

CLÁUSULA= ((NOT (INSCRITO VAR7 NEW_CTE1)) (NOT (CONTENTE VAR7)))

Substituições: VAR6 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (5 , 1)

CLÁUSULA= ((INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1))

Substituições: VAR4 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (2 , 5)

CLÁUSULA= ((CONTENTE (NEW_FUNC1 NEW_CTE1)))

Substituições: VAR5 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (3 , 2)

CLÁUSULA= ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)) (NOT (FACIL VAR5)))

Substituições: VAR7 / (NEW_FUNC1 VAR5) ;

ITERACAO #2

CLÁUSULA #1: ((NOT (FACIL VAR4)) (INSCRITO (NEW_FUNC1 VAR4) VAR4))

CLÁUSULA #2: ((NOT (FACIL VAR5)) (CONTENTE (NEW_FUNC1 VAR5)))

CLÁUSULA #3: ((NOT (EXAME VAR6)) (NOT (INSCRITO VAR7 VAR6)) (NOT (CONTENTE VAR7)))

CLÁUSULA #4: ((EXAME NEW_CTE1))

CLÁUSULA #5: ((FACIL NEW_CTE1))

CLÁUSULA #6: ((NOT (FACIL VAR5)) (NOT (EXAME VAR6)))

(NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)))
 CLÁUSULA #7: ((NOT (INSCRITO VAR7 NEW_CTE1)) (NOT (CONTENTE VAR7)))
 CLÁUSULA #8: ((INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1))
 CLÁUSULA #9: ((CONTENTE (NEW_FUNC1 NEW_CTE1)))
 CLÁUSULA #10: ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6))
 (NOT (FACIL VAR5)))

NOVA CLÁUSULA!! => ORIGEM (8 , 3)
 CLÁUSULA= ((NOT (EXAME NEW_CTE1)) (NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))
 Substituições: VAR6 / NEW_CTE1 ; VAR7 / (NEW_FUNC1 NEW_CTE1) ;

NOVA CLÁUSULA!! => ORIGEM (9 , 3)
 CLÁUSULA= ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 NEW_CTE1) VAR6)))
 Substituições: VAR7 / (NEW_FUNC1 NEW_CTE1) ;

NOVA CLÁUSULA!! => ORIGEM (6 , 4)
 CLÁUSULA= ((NOT (FACIL VAR5)) (NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)))
 Substituições: VAR6 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (6 , 8)
 CLÁUSULA= ((NOT (FACIL NEW_CTE1)) (NOT (EXAME NEW_CTE1)))
 Substituições: VAR6 / NEW_CTE1 ; VAR5 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (7 , 8)
 CLÁUSULA= ((NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))
 Substituições: VAR7 / (NEW_FUNC1 NEW_CTE1) ;

NOVA CLÁUSULA!! => ORIGEM (7 , 2)
 CLÁUSULA= ((NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)) (NOT (FACIL VAR5)))
 Substituições: VAR7 / (NEW_FUNC1 VAR5) ;

NOVA CLÁUSULA!! => ORIGEM (10 , 8)
 CLÁUSULA= ((NOT (EXAME NEW_CTE1)) (NOT (FACIL NEW_CTE1)))
 Substituições: VAR6 / NEW_CTE1 ; VAR5 / NEW_CTE1 ;

ITERAÇÃO #3
 CLÁUSULA #1: ((NOT (FACIL VAR4)) (INSCRITO (NEW_FUNC1 VAR4) VAR4))
 CLÁUSULA #2: ((NOT (FACIL VAR5)) (CONTENTE (NEW_FUNC1 VAR5)))
 CLÁUSULA #3: ((NOT (EXAME VAR6)) (NOT (INSCRITO VAR7 VAR6)) (NOT (CONTENTE
 VAR7)))
 CLÁUSULA #4: ((EXAME NEW_CTE1))
 CLÁUSULA #5: ((FACIL NEW_CTE1))
 CLÁUSULA #6: ((NOT (FACIL VAR5)) (NOT (EXAME VAR6))
 (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)))
 CLÁUSULA #7: ((NOT (INSCRITO VAR7 NEW_CTE1)) (NOT (CONTENTE VAR7)))
 CLÁUSULA #8: ((INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1))
 CLÁUSULA #9: ((CONTENTE (NEW_FUNC1 NEW_CTE1)))
 CLÁUSULA #10: ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6))
 (NOT (FACIL VAR5)))
 CLÁUSULA #11: ((NOT (EXAME NEW_CTE1)) (NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))
 CLÁUSULA #12: ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 NEW_CTE1) VAR6)))
 CLÁUSULA #13: ((NOT (FACIL VAR5)) (NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)))
 CLÁUSULA #14: ((NOT (FACIL NEW_CTE1)) (NOT (EXAME NEW_CTE1)))
 CLÁUSULA #15: ((NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))
 CLÁUSULA #16: ((NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)) (NOT (FACIL VAR5)))
 CLÁUSULA #17: ((NOT (EXAME NEW_CTE1)) (NOT (FACIL NEW_CTE1)))

NOVA CLÁUSULA!! => ORIGEM (12 , 4)
 CLÁUSULA= ((NOT (INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1)))
 Substituições: VAR6 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (12 , 8)
 CLÁUSULA= ((NOT (EXAME NEW_CTE1)))
 Substituições: VAR6 / NEW_CTE1 ;

NOVA CLÁUSULA!! => ORIGEM (13 , 8)
 CLÁUSULA= ((NOT (FACIL NEW_CTE1)))
 Substituições: VAR5 / NEW_CTE1 ;

ITERACAO #4

CLÁUSULA #1: ((NOT (FACIL VAR4)) (INSCRITO (NEW_FUNC1 VAR4) VAR4))

CLÁUSULA #2: ((NOT (FACIL VAR5)) (CONTENTE (NEW_FUNC1 VAR5)))

CLÁUSULA #3: ((NOT (EXAME VAR6)) (NOT (INSCRITO VAR7 VAR6)) (NOT (CONTENTE VAR7)))

CLÁUSULA #4: ((EXAME NEW_CTE1))

CLÁUSULA #5: ((FACIL NEW_CTE1))

CLÁUSULA #6: ((NOT (FACIL VAR5)) (NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)))

CLÁUSULA #7: ((NOT (INSCRITO VAR7 NEW_CTE1)) (NOT (CONTENTE VAR7)))

CLÁUSULA #8: ((INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1))

CLÁUSULA #9: ((CONTENTE (NEW_FUNC1 NEW_CTE1)))

CLÁUSULA #10: ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 VAR5) VAR6)) (NOT (FACIL VAR5)))

CLÁUSULA #11: ((NOT (EXAME NEW_CTE1)) (NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))

CLÁUSULA #12: ((NOT (EXAME VAR6)) (NOT (INSCRITO (NEW_FUNC1 NEW_CTE1) VAR6)))

CLÁUSULA #13: ((NOT (FACIL VAR5)) (NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)))

CLÁUSULA #14: ((NOT (FACIL NEW_CTE1)) (NOT (EXAME NEW_CTE1)))

CLÁUSULA #15: ((NOT (CONTENTE (NEW_FUNC1 NEW_CTE1))))

CLÁUSULA #16: ((NOT (INSCRITO (NEW_FUNC1 VAR5) NEW_CTE1)) (NOT (FACIL VAR5)))

CLÁUSULA #17: ((NOT (EXAME NEW_CTE1)) (NOT (FACIL NEW_CTE1)))

CLÁUSULA #18: ((NOT (INSCRITO (NEW_FUNC1 NEW_CTE1) NEW_CTE1)))

CLÁUSULA #19: ((NOT (EXAME NEW_CTE1)))

CLÁUSULA #20: ((NOT (FACIL NEW_CTE1)))

NOVA CLÁUSULA!! => ORIGEM (18 , 8)

CLÁUSULA= NIL

Substituições:

KB inconsistente!! => a cláusula representa uma FALSIDADE.

T

Para terminar, leia o **Guia para Utilização Imediata**, que explica ainda a eventual importância do ficheiro **USER.LSP**. Obrigado pelo seu interesse. Espero que aprecie.

Guia para Utilização Imediata

Para experimentar o motor de inferência, precisa apenas de:

→ #1) Carregar os ficheiros .LSP necessários

- copie os ficheiros para a directoria do seu interesse, por exemplo **C:\INFERE**.
- certifique-se que a directoria default do seu interpretador LISP é agora a directoria **C:\INFERE**:

```
(setq *default-pathname-defaults* "C:\\INFERE\\")
```

- Carregue o ficheiro "LOADER.LSP" e tudo o resto acontecerá automaticamente:

```
(load "loader.lsp")
```

→ #2) Já pode experimentar fazer inferência sobre uma das três KBs [Knowledge Base] fornecidas como exemplo! Por defeito, fará inferências sobre a KB, cujas fórmulas estão escritas no ficheiro KB4.TXT. Escreva:

```
(infere)
```

E o programa acontece!

→ NOTAS:

Quando quiser escrever fórmulas da sua autoria, leia o restante deste documento e edite o ficheiro **USER.LSP**, de forma a que indique correctamente quais as suas constantes, variáveis, funções e predicados iniciais.

Segue-se o ficheiro **USER.LSP**, por defeito. Note-se que pode modificar a KB sobre a qual (infere) trabalha por defeito, modificando a variável ***default_file_name_for_kb_file***!

USER.LSP

; definições suficientes para testar com sucesso os exemplos KB.TXT , KB2.TXT , KB3.TXT e KB4.TXT

```
(setq *default_file_name_for_kb_file* "kb4.txt")
(setq *variaveis* '(v1 v2 v3 x1 x2 x3 x d e))
(setq *constantes* '(pluto))
(setq *funcoes* '(f1 f2 f3))
(setq *predicados* '(f c bc cea g cao saliva facil contente inscrito exame))
```

→ NÃO ESQUECER:

Se modificar aspectos que influenciam o comportamento do programa, como o ficheiro **USER.LSP**, precisa de fazer reload do projecto. A forma mais simples de fazê-lo, é voltar a escrever:

```
(load "loader.lsp")
```