

## Nota:

Nas páginas que se seguem, as palavras não estão acentuadas. Isto é propositado: a rede semântica em causa, foi inicialmente desenvolvida em plataforma VAX/VMS, onde o interpretador de LISP disponível não tolerava acentuação...

Com uma pequena alteração, foi possível fazer «correr» a rede semântica em qualquer computador pessoal! Disponibilizam-se os «ficheiros de arranque» quer para VAX/VMS quer para PC/DOS.

A rede semântica também corre em janela/écran de DOS, em OS/2 ou Windows. Todavia o desempenho é superior em OS/2, uma vez que se disponibiliza maior quantidade de memória RAM e um núcleo multi-tarefa mais «performante», vantajoso em operações de procura sobre redes de grande dimensão.

As páginas (não acentuadas) que se seguem, são o código fonte do programa, em linguagem LISP.

O LISP é geralmente interpretado, sendo bastante utilizado em aplicações de inteligência artificial. Para obviar as desvantagens da interpretação em relação à compilação, alguns fabricantes disponibilizam máquinas-LISP, cujos processadores estão absolutamente orientados à linguagem, com suporte directo para as instruções! É claro que o programa também corre nessas máquinas «dedicadas».

O poder da rede semântica desenvolvida é muito considerável, principalmente do ponto de vista de velocidade de desempenho, em relação a soluções alternativas! As potencialidades das redes semânticas, e a utilização desta rede em particular, são assunto para a documentação anexa.

O código fonte aqui listado, está segmentado por ficheiros, efusivamente documentados! Por outras palavras, toda a documentação interna consta nas páginas que se seguem.



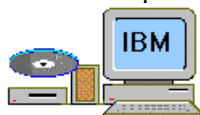
## Ficheiro: XRUN.LSP

Descricao:

Activa a rede semantica, em ambiente PC, utilizando a linguagem XLISP.  
Os ponto-e-virgula (;) marcam comentarios. Para carregar automaticamente a rede de exemplo (ficheiro "REDE.DAT"), remover o ponto-e-virgula da ultima linha deste ficheiro.

```
(expand 3)  
(load 'varios)  
(load 'descreve)  
(load 'grava)  
(load 'carrega)  
(load 'nos)  
(load 'inic)  
(load 'var)  
(load 'procura)  
(load 'prtostr.pc)  
(load 'merge.pc)  
;(carrega "rede.dat")
```

Artur Marques



2

## Ficheiro: RUN.LSP

Descricao:

Activa a rede semantica, em ambiente VAX/VMS, utilizando a linguagem LISP.  
Os ponto-e-virgula (;) marcam comentarios. Para carregar automaticamente a rede de exemplo (ficheiro "REDE.DAT"), remover o ponto-e-virgula da ultima linha deste ficheiro.

```
(load 'varios)
(load 'descreve)
(load 'grava)
(load 'carrega)
(load 'nos)
(load 'inic)
(load 'var)
(load 'procura)
(load 'merge)
;(carrega "rede.dat")
```



## Ficheiro: CARREGA.LSP

; Carrega em memoria uma rede arquivada pelo processo da funcao grava.  
; Caso nao se entre com a string correspondente ao nome do ficheiro em foi  
; gravada a rede, assume-se que a mesma foi gravada no ficheiro rede.dat  
; Como seria de esperar, o carregamento e feito em funcao do formato que  
; grava utilizou.  
; Esta funcao e destructiva, quer dizer, elimina a rede em que se estava a  
; trabalhar antes de carregar os dados da rede em ficheiro.  
; Provoca o desaparecimento de qualquer variavel existente.

```
(defun carrega (&optional (fich "rede.dat"))
  (setf *variaveis* nil)
  (let ((in (open fich :direction :input)))
    (setf *rede* nil)
    (setf *num-nos* (read in))
    (setf *relacoes* (read in))
    (carrega-nos (read in))
    (carrega-relacoes (read in))
    (close in)
    (terpri)
    t
  )
)
```

-----  
; Esta funcao processa a linha da forma ((no tipo-do-no) ...) do ficheiro  
; que contem a rede. Na pratica cria os nos com o respectivo identificador  
; e tipo e com as suas listas de relacoes ascendentes e descendentes  
; inicializadas a nil.

```
(defun carrega-nos (lista-nos)
  (dolist (lst lista-nos t)
    (let ((id (car lst)) (tipo (cadr lst)) (no nil))
      (setf no (existe-no id tipo))
      (cond ((not no) (setf *rede* (junta-fim *rede* (cria-no id tipo nil nil))))
    )
  )
)
```

-----  
; Esta funcao trata a segunda linha do ficheiro que se gravou.  
; Essa linha e uma lista de listas. Cada uma das listas em questao corresponde  
; a um certo no. Na realidade cada uma dessas listas e da forma  
; ((no1 rel? nodest?) ... (no1 rel? nodest?))  
; Dentro do ciclo que analisa cada lista ha um outro ciclo que trata cada um  
; dos ternos (no1 rel? nodest?). Esse tratamento consiste em - caso ja existam  
; ambos os nos origem e destino - ligar os mesmos pela relacao.  
; Esta ultima tarefa fica a cargo de liga-nos.

Artur Marques

4



```

(defun carrega-relacoes (lista-relacoes)
  (dolist (relacoes-no lista-relacoes t)
    (dolist (rel relacoes-no t)
      (let ((no1 nil) (no2 nil) (nome-no1 (car rel)) (nome-rel (nth 1 rel)) (nome-no2 (nth 2
rel))))
        (setf no1 (existe-no nome-no1))
        (setf no2 (existe-no nome-no2))
        (cond ((and no1 no2) (liga-nos no1 no2 nome-rel)))
      )
    )
  )
)

```

-----  
; Encarrega-se de actualizar sucessivamente as relacoes ascendentes e  
; descendentes dos nos. Invoca forma-inv para formar o nome das relacoes  
; inversas. Os nos que manipula foram previamente criados.

```

(defun liga-nos (no1 no2 nome-rel)
  (let ((rel-desc nil) (rel-asc nil))
    (setf rel-desc (cria-rel nome-rel no2))
    (setf rel-asc (cria-rel (forma-inv nome-rel) no1))
    (setf (no-relacoes-desc no1) (junta-fim (no-relacoes-desc no1) rel-desc))
    (setf (no-relacoes-asc no2) (junta-fim (no-relacoes-asc no2) rel-asc))
  )
)

```



## Ficheiro: DESCRVE.LSP

```
; Descr limita-se a chamar descreve-no. A necessidade de criacao desta funcao  
; deve-se a macro dump que desta forma conduz a descricao do no, apos  
; verificacao da sua existencia. A verificacao da existencia do no, nao e  
; feita por descreve-no, dai este mecanismo.
```

```
(defun descr (no-id &aux no)  
  (setf no (existe-no no-id))  
  (cond (no (descreve-no no))  
        (t (princ "O no "  
                 (princ no-id)  
                 (princ " nao existe !")  
                 (terpri)  
                 t  
              )  
        )  
  )  
); fim do descr - a ser chamado pela macro dump.
```

```
-----  
; Descreve - de forma textual - o no recebido como argumento. Reparar que  
; precisa mesmo dum no e nao do seu identificador !  
; Invoca a funcao escreve-rel para escrever as suas relacoes ascendentes e  
; descendentes.
```

```
(defun descreve-no (no)  
  (princ "-----")  
  (terpri)  
  (princ "Identificador: ")  
  (princ (no-nome-no no))  
  (terpri)  
  (princ "Tipo: ")  
  (princ (no-tipo-no no))  
  (terpri)  
  (princ "Relacoes Ascendentes / No destino: ")  
  (terpri)  
  (escreve-rel 'asc no)  
  (terpri)  
  (princ "Relacoes Descendentes / No destino: ")  
  (terpri)  
  (escreve-rel 'desc no)  
  (princ "-----")  
  (terpri)  
  t  
); fim de descreve-no
```

```
-----  
; Recebe no parametro qual 'asc ou 'desc conforme se deseje que esta funcao  
; escreva as relacoes ascendentes ou descendentes do no que se recebe como
```

Artur Marques

6





; A necessidade de \*ja-visitados\* deve-se a natureza recursiva da funcao info.

```
(defun chama-info (no-id)
  (setf *ja-visitados* nil)
  (info no-id)
  ) ; fim de chama-info
```

```
;-----
; Recursivamente descreve um no e os seus descendentes, tendo cuidado de nao
; repetir a escrita de nos que ja tenham sido descritos.
; Esta opcao so se justifica pela forma completa como cada no individual e
; descrito.
```

```
(defun info (no-id &aux no descendentes)
  (terpri)
  (setf no (existe-no no-id))
  (cond (no
        (descreve-no no)
        (setf *ja-visitados* (junta-fim *ja-visitados* (no-nome-no no)))
        (setf descendentes (devolve-lista-nos-apontados no))
        (do ( (lst descendentes (cdr lst))
              (no-actual nil)
            )
          ((null lst) t)
          (setf no-actual (no-nome-no (car lst)))
          (cond ((not (member no-actual *ja-visitados*))
                 (info no-actual)
                )
              )
          )
        ) ; do
      )
    (t (princ "O no ")
       (princ no-id)
       (princ " nao existe !")
       (terpri)
       t
      )
  ) ; cond
) ; fim info
```



## Ficheiro: INIC.LSP

; Esta funcao faz a apresentacao do programa e inicia um ciclo de leitura de comandos.  
; Durante o ciclo de leitura, alguns comandos serao aceites, outros nao. Para saber quais os comandos validos, devera consultar a documentacao do utilizador ou escrever "(ajuda)" - ao fazer isso ser-lhe-ao listados os comandos possiveis, embora os mesmos nao sejam descritos.  
; Para abandonar este interpretador de comandos devera escrever "(exit)" - o que causara a saida - inclusive -  
; do Lisp.

```
(defun rs ()  
  (apresentacao)  
  (loop  
    (terpri)  
    (princ "RS > ")  
    (let ((in (read)))  
      (cond ((equal in 'exit) (return nil))  
            (t (cond ((member (car in) *comandos*)  
                      (princ (eval in))  
                      (terpri)  
                      )  
                  (t (princ "Comando invalido.")  
                     (ajuda)  
                     )  
                  )  
            )  
      )  
    )  
  )  
)
```

-----  
; Informacao sobre o autor.

```
(defun apresentacao ()  
  (terpri)  
  (terpri)  
  (princ "          -- Programa em LISP para implementacao duma rede  
semantica. --")  
  (terpri)  
  (princ "          -- Por: Artur Manuel Sancho Marques")  
  (terpri)  
  (terpri)  
)
```

-----  
; Invocada por RS ou pelo utilizador e que lista os comandos possiveis.

Artur Marques

9



```

(defun ajuda ()
  (terpri)
  (terpri)
  (princ "Comandos validos :")
  (terpri)
  (princ "(assercao 'rel-1 'conjnos-1 .. 'rel-n 'conjnos-n)")
  (terpri)
  (princ "(build 'rel-1 'conjnos-1 .. 'rel-n 'conjnos-n)")
  (terpri)
  (princ "(def-rel 'rel-1 .. 'rel-n)")
  (terpri)
  (princ "(retira 'no-id)")
  (terpri)
  (princ "(descreve 'no-id)")
  (terpri)
  (princ "(dump 'no-id)")
  (terpri)
  (princ "(lista 'nos)")
  (terpri)
  (princ "(lista 'relacoes)")
  (terpri)
  (princ "(lista 'variaveis)")
  (terpri)
  (princ "(quantos-nos?)")
  (terpri)
  (princ "(pertence-rede? 'no-id)")
  (terpri)
  (princ "(carrega 'ficheiro-rede)")
  (terpri)
  (princ "(grava 'ficheiro-rede)")
  (terpri)
  (princ "(valor? 'var-id)")
  (terpri)
  (princ "(existe-var 'var-id)")
  (terpri)
  (princ "(tira-var 'var-id)")
  (terpri)
  (princ "(procura 'rel-1 'conjnos-1 ... 'rel-n 'conjnos-n)")
  (terpri)
  (princ "(procura-faz 'rel-1 'conjnos-1 ... 'rel-n 'conjnos-n)")
  (terpri)
  (princ "--> Em procura e em procura-faz, os conjnos podem ser variaveis: por ex '?var1.")
  (terpri)
  (princ "(merge-rede 'nome-do-ficheiro)")
  (terpri)
  (princ "(ajuda) - Este ecran.")
  (terpri)
  (princ "(exit)")
  (terpri)

```



```
t
)
```

-----

; Pode receber 2 argumentos : nos ou relacoes. Se receber nos lista os identificadores  
; dos nos existentes, se receber relacoes lista as relacoes ja definidas, tal como  
; estao na lista \*relacoes\*.  
; Recebendo outra coisa para la de nos ou relacoes, informa que recebeu um argumento  
invalido.

```
(defun lista (&optional o-que)
  (cond ((equal o-que 'nos)
    (cond ((null *rede*)
      (princ "Nao existem nos na rede.")
      (terpri)
    )
    (t (princ "Nos :")
      (terpri)
      (dolist (cabeca *rede* t)
        (format t "( ~A )" (no-nome-no cabeca))
        (terpri)
      )
    )
  )
)
)
(t (cond ((equal o-que 'relacoes)
  (cond ((null *relacoes*)
    (princ "Nao existe relacoes.")
    (terpri)
  )
  (t (princ "Relacoes :")
    (terpri)
    (dolist (cabeca *relacoes* t)
      (format t "( ~A )" cabeca)
      (terpri)
    )
  )
)
)
)
(t
  (cond ((equal o-que 'variaveis)
    (cond ((null *variaveis*)
      (princ "Nao existem variaveis.")
      (terpri)
    )
    (t
      (princ "Variaveis:")
      (terpri)
      (do* ((lst *variaveis* (cdr lst))
        (cabeca (car lst) (car lst))

```





## Ficheiro: GRAVA.LSP

```
; As seguintes funcoes estao todas relacionadas com o processo de gravacao
; da rede num ficheiro com nome a escolha pelo utilizador.
; O processo de gravacao da rede em ficheiro obedece a seguinte filosofia:
; 1) grava-se a lista que corresponde as relacoes definidas
; 2) grava-se uma lista cujos elementos sao eles proprios listas em que
;     o primeiro elemento e o nome do no e o segundo elemento o tipo do no.
; 3) grava-se uma lista da forma devolvida por info-relacoes (ver comentario
;     respectivo) e que visa arquivar informacao sobre as relacoes de cada
;     um dos nos presentes na rede.
```

```
(defun grava (&optional (nome-ficheiro "rede.dat"))
  (let ((out (open nome-ficheiro :direction :output :if-exists :new-version)))
    (print *num-nos* out)
    (print *relacoes* out)
    (print (info-nos *rede*) out)
    (print (info-relacoes *rede*) out)
    (close out)
    (terpri)
    t
  )
); fim grava
```

```
-----
; Limita-se a devolver uma lista de elementos que sao listas de 2 elementos.
; A lista devolvida e da forma ((nome-do-no1 tipo-do-no1) ...)
```

```
(defun info-nos (no-lista)
  (let ((info nil))
    (dolist (no no-lista info)
      (setf info (cons (list (no-nome-no no) (no-tipo-no no)) info))
    )
  )
); fim info-nos
```

```
-----
; Recebe uma lista de nos e que, inicialmente, devera ser a propria rede.
; Para cada um dos elementos dessa lista, forma uma lista em que os elementos
; sao listas da forma (no-origem relacao no-destino), a formacao dessa lista
; fica a cargo da funcao info-desc.
; Devolvera entao uma lista com um aspecto do genero
; ((n1 rel? ndest?) (n1 rel? ndest?) ...)
; ((n2 rel? ndest?) (n2 rel? ndest?) ...)
; ...
; )
```

```
(defun info-relacoes (no-lista)
  (let ((info nil))
```

Artur Marques

13



```

(dolist (no no-lista info)
  (let ((rel-desc (no-relacoes-desc no)))
    (cond (rel-desc (setf info (cons (info-desc no rel-desc) info))))
  )
)
); fim info-relacoes

```

-----  
; Esta funcao, recebe um no e a lista das suas relacoes descendentes.  
; Vai devolver uma lista - progressivamente construida - em que os  
; elementos sao eles proprios listas da forma (no-origem relacao no-destino)  
; conforme devolvidas pela funcao info-relacao.

```

(defun info-desc (no rel-desc)
  (let ((info nil))
    (dolist (rel rel-desc info)
      (setf info (cons (info-relacao no rel) info))
    )
  )
); fim info-desc

```

-----  
; Devolve uma lista em que os elementos sao ternos  
; (no-origem relacao no-destino). Esta lista e utilizada por info-desc.  
; Os parametros recebidos sao o no origem e uma estrutura relacao com o nome  
; rel. Isso significa que dessa estrutura podemos obter o seu nome e o no  
; por ela apontado.

```

(defun info-relacao (no rel)
  (list (no-nome-no no) (relacao-nome-rel rel) (no-nome-no (relacao-aponta-no rel)))
); fim info-relacao

```



## Ficheiro: MERGE.LSP

```
; Funcao que recebe o nome dum ficheiro correspondente a uma rede.  
; Coloca a rede presente nesse ficheiro em *buffer*.  
; E exactamente igual a funcao carrega, excepto em que faz o append para  
; os nomes das relacoes e, em vez de trabalhar para a variavel *rede*,  
; trabalha para a variavel *buffer*.
```

```
(defun carrega-buffer (fich)  
  (let ((in (open fich :direction :input)))  
    (read in) ; corresponde a leitura do numero de nos.  
    (setf *relacoes* (remove-duplicates (append *relacoes* (read in)))) ; corresponde a  
leitura das relacoes  
    (setf *buffer* nil) ; agora os nos sao lidos para um buffer.  
    (carrega-nos-buffer (read in))  
    (carrega-relacoes-buffer (read in))  
    (close in)  
    (terpri)  
    t  
  )  
)
```

```
-----  
; Esta funcao processa a linha da forma ((no tipo-do-no ...) do ficheiro  
; que contem a rede. Na pratica cria os nos com o respectivo identificador  
; e tipo e com as suas listas de relacoes ascendentes e descendentes  
; inicializadas a nil.
```

```
(defun carrega-nos-buffer (lista-nos)  
  (dolist (lst lista-nos t)  
    (let ((id (car lst)) (tipo (cadr lst)) (no nil))  
      (setf no (existe-no-buffer id tipo))  
      (cond ((not no)  
            (setf *buffer* (junta-cabeca *buffer* (cria-no id tipo nil nil)))  
            )  
      )  
    )  
  )  
)
```

```
-----  
; Recebe uma lista que corresponde as relacoes descendentes de certo no.  
; Devolve a lista que corresponde ao formato que deveria ser utilizado  
; para uma chamada build ou assercao para provocar a criacao do no com  
; aquelas relacoes descendentes.
```

```
(defun obtem-relacoes (lista)  
  (do* ((lst lista (cdr lst))  
        (cabeca (car lst) (car lst))
```

Artur Marques

15



```

    (rel nil)
    (no nil)
    (devolucao nil)
  )
  ((null lst) devolucao)
  (setf rel (relacao-nome-rel cabeca))
  (setf no (no-nome-no (relacao-aponta-no cabeca)))
  (setf devolucao (append devolucao (list rel no)))
)
)

```

```

;-----
; Eis a funcao de merge.
; Recebe o nome do ficheiro a fazer merge. De seguida tem um comportamento
; exactamente igual ao que teria a funcao carrega, mas, em vez de ser para
; a variavel *rede*, e para a variavel *buffer*.
; Tendo a rede que se quer fazer merge em *buffer*, basta percorre-la e
; ver quais as chamadas que deveriam ser feitas para criar cada um dos nos
; presentes nessa rede (*buffer*). Fazem-se entao os build ou assercao
; que conduziriam ao no existente, o build e o assercao encarregam-se de
; verificar tudo o resto, isto e, a existencia dum conceito igual em *rede*, etc.

```

```

(defun merge-rede (nome)
  (carrega-buffer nome)
  (do* ((lst *buffer* (cdr lst))
        (cabeca (car lst) (car lst))
        (tipo nil)
        (relacoes-desc nil)
        (devolve nil)
        )
    ((null lst) t)
    (setf tipo (no-tipo-no cabeca))
    (setf relacoes-desc (obtem-relacoes (no-relacoes-desc cabeca)))
    (cond ((equal tipo 'HIP)
           ; devolve e o suposto nome do no ja existente na rede.
           ; se vier uma lista e porque o conceito correspondente
           ; ja existia, logo por o nome no buffer igual ao nome na rede.
           (setf devolve (def-no 'HIP relacoes-desc))
           (cond ((listp devolve)
                  (setf devolve (cadr devolve))
                  (setf (no-nome-no cabeca) devolve)
                )
           )
         )
    ((equal tipo 'TRUE)
     (setf devolve (def-no 'TRUE relacoes-desc))
     (cond ((listp devolve)
            (setf devolve (cadr devolve))
            (setf (no-nome-no cabeca) devolve)
          )
    )
  )
)

```



```
)
)
)
)
)
```

```
-----
; Esta funcao trata a segunda linha do ficheiro que se gravou.
; Essa linha e uma lista de listas. Cada uma das listas em questao corresponde
; a um certo no. Na realidade cada uma dessas listas e da forma
; ((no1 rel? nodest?) ... (no1 rel? nodest?))
; Dentro do ciclo que analisa cada lista ha um outro ciclo que trata cada um
; dos ternos (no1 rel? nodest?). Esse tratamento consiste em - caso ja existam
; ambos os nos origem e destino - ligar os mesmos pela relacao.
; Esta ultima tarefa fica a cargo de liga-nos.
```

```
(defun carrega-relacoes-buffer (lista-relacoes)
  (dolist (relacoes-no lista-relacoes t)
    (dolist (rel relacoes-no t)
      (let ((no1 nil) (no2 nil) (nome-no1 (car rel)) (nome-rel (nth 1 rel)) (nome-no2 (nth 2
rel)))
        (setf no1 (existe-no-buffer nome-no1))
        (setf no2 (existe-no-buffer nome-no2))
        (cond ((and no1 no2) (liga-nos no1 no2 nome-rel)))
      )
    )
  )
)
```

```
-----
; Recebe um identificador dum no.
; Devolve nil se o no nao existir no buffer.
; Devolve o no se ele existir.
; Opcionalmente pode receber o tipo do no, no caso de existirem no buffer
; nos com o mesmo identificador, mas de tipos diferentes.
```

```
(defun existe-no-buffer (no-id &optional tipo-opcional)
  (cond ((null *buffer*) nil)
        (t (do (
                (lista *buffer* (cdr lista))
                (cabeca nil)
                (id nil)
                (tipo nil)
                (ja-encontrou nil)
                (devolve nil)
              )
          ((or ja-encontrou (null lista)) devolve)
          (setf cabeca (car lista))
          (setf id (no-nome-no cabeca))
          (setf tipo (no-tipo-no cabeca))
        )
  )
)
```

Artur Marques

17



```
(cond ((null tipo-opcional)
      (cond ((equal id no-id)
            (setf ja-encontrou t)
            (setf devolve cabeca)
            )
          )
      )
      )
      (t
        (cond ((and (equal id no-id) (equal tipo tipo-opcional))
              (setf ja-encontrou t)
              (setf devolve cabeca)
              )
          )
        )
      ); t
    ); cond primario
  )
)
```



## Ficheiro: NOS.LSP

```
; Eis a estrutura que representa um no.  
; Um no tem 4 campos: O seu nome e o seu tipo (atoms)  
; E duas listas cujos elementos serao eles proprios estruturas. Essas listas  
; sao as de relacoes ascendentes - relacoes-asc - e a de relacoes  
; descendentes - relacoes-desc.
```

```
(defstruct no  
  nome-no ; atom  
  tipo-no ; atom  
  relacoes-asc ; lista de estruturas  
  relacoes-desc ; lista de estruturas  
)
```

```
-----  
; Esta estrutura representa uma relacao. O primeiro campo e o nome da  
; relacao e o segundo campo um ponteiro para o no para o qual a relacao  
; aponta.
```

```
(defstruct relacao  
  nome-rel ; atomo  
  aponta-no ; ponteiro para um no  
)
```

```
-----  
(setf *rede* nil) ; lista de estruturas que sao os nos  
(setf *buffer* nil) ; lista de estruturas utilizada pela funcao merge  
(setf *relacoes* nil) ; lista de atomos que representam o nome das relacoes  
; lista dos comandos possiveis  
(setf *comandos* '(assercao build def-rel retira dump descreve lista carrega grava quantos-nos?  
pertence-rede? valor? existe-var tira-var procura procura-faz merge-rede ajuda exit))  
(setf *num-nos* 0) ; numero de nos criados ate ao momento na rede
```

```
-----  
; Para criar um no. Esta funcao e no entanto de baixo nivel e nao deve  
; ser utilizada pelo user da rede semantica ! Destina-se a ser invocada  
; exclusivamente pelas funcoes devidas.
```

```
(defun cria-no (nome tipo asc desc)  
  (make-no :nome-no nome :tipo-no tipo :relacoes-asc asc :relacoes-desc desc)  
)
```

```
-----  
; Para criar uma estrutura relacao. Mais uma vez esta funcao nao se destina  
; ao utilizador normal da rede, mas sim a um conjunto de funcoes de nivel  
; superior.
```



```
(defun cria-rel (nome no)
  (make-relacao :nome-rel nome :aponta-no no)
)
```

-----

; Devolve o numero de nos existentes na rede.

```
(defun quantos-nos? ()
  (length *rede*)
)
```

-----

; Esta funcao recebe o nome dum no e a respectiva lista de relacoes  
 ; descendentes. Essa lista vai ser analisada e todos os nos que sejam  
 ; apontados pelas relacoes descendentes do no argumento, deixam de ter as  
 ; suas relacoes ascendentes a apontar para ele. Isto e necessario para  
 ; garantir o principio da unicidade na rede aquando duma chamada a build  
 ; ou assercao para criar um no que refere um conceito, afinal, ja existente.  
 ; De facto o funcionamento de build e assercao, provoca sempre a criacao  
 ; dum no temporario que acabara por nao ser incluido na rede caso um conceito  
 ; identico ja exista. No entanto sera necessario apagar tambem as relacoes  
 ; ascendentes temporarias entretanto criadas.  
 ; Devolve t.

```
(defun elimina-relacoes-asc-temporarias (nome-no-origem lst-rel-desc)
  (do* ((lst lst-rel-desc (cdr lst))
        (cabeca (car lst) (car lst))
        (no-apontado nil)
        (nome-rel nil)
        )
        ((null lst) t)
        (setf no-apontado (relacao-aponta-no cabeca))
        (setf nome-rel (relacao-nome-rel cabeca))
        (apaga-rel-asc no-apontado nome-no-origem nome-rel)
  )
)
```

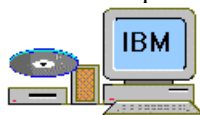
-----

; Esta funcao e chamada por elimina-relacoes-asc-temporarias e ve quais sao  
 ; as relacoes ascendentes do no que recebe como 1-o argumento que apontam  
 ; com um arco chamado nome-rel (3-o argumento) para o no cujo nome e  
 ; nome-no-origem (2-o argumento). So as relacoes ascendentes com esse nome  
 ; podem ser eliminadas, as restantes nao.  
 ; Devolve t.

```
(defun apaga-rel-asc (no nome-no-origem nome-rel)
  (do* ((lst (no-relacoes-asc no) (cdr lst))
        (cabeca (car lst) (car lst))
        (rel-actual nil)
        )
  )
)
```

Artur Marques

20





```

)
(t (princ "A relacao ")
  (princ elemento)
  (princ " ainda nao foi definida.")
  (terpri)
  (setf foi-criado-no nil)
  (setf lista nil)
)
)
)
)
); fim cond para elementos de indice par da lista de argumentos
(t
  (cond ((listp elemento) ; temos uma lista de nos em elemento
        (setf lista-relacoes (acrescenta lista-relacoes rel-tmp elemento))
        )
        (t (setf lista-relacoes (junta-fim lista-relacoes (list rel-tmp elemento))))))
)
(setf foi-criado-no t)
)
); cond
); do
); t
); cond
(cond (foi-criado-no
      (setf novos (gera-no lista-relacoes tipo)) ; novos e uma lista de novos nos
      (cond (novos
            (let* ((no (car novos)) (igual? (ja-existe-conceito-igual? no)))
              (cond (igual?
                    (elimina-relacoes-asc-temporarias (no-nome-no no) (no-relacoes-desc no))
                    ( princ (list 'ja-existe-conceito-igual-em igual?))
                    )
              (t
                (setf *rede* (append *rede* novos))
                (setf *num-nos* (1+ *num-nos*))
                (no-nome-no (car novos))
                )
              )
            )
      )
); cond
); let
)
)
)
); fim def-no

```

-----  
; Acrescenta os nomes de relacoes recebidas como argumentos a lista de  
; relacoes conhecidas pela rede semantica e que e mantida na variavel global  
; \*relacoes\*. Os nomes das relacoes precisam de ser entrados quoted e so  
; serao considerados se ainda nao existirem em \*relacoes\*

Artur Marques

22



; Devolve a lista com o nome das relacoes criadas.

```
(defun def-rel (&rest argumentos &aux tam el tmp-rel)
  (setf tmp-rel nil)
  (cond ((null argumentos)
    (terpri)
    (princ "Aviso: Esperava-se o nome duma relacao ou uma sequencia de relacoes.")
    (terpri)
    )
    (t
    (setf tam (length argumentos))
    (dolist (el argumentos t)
      (cond ( (not (atom el))
        (terpri)
        (princ "Erro: Argumentos nao correspondem a sintaxe de entrada.")
        (terpri)
        (princ "Exemplo para def-rel: (def-rel 'relacao1 ... 'relacao-n)")
        (terpri)
        )
        (t
        (cond ( (not (member el *relacoes*))
          (setf *relacoes* (junta-fim *relacoes* el))
          (setf tmp-rel (junta-fim tmp-rel el))
          )
          (t (terpri)
            (princ "Relacao ")
            (princ el)
            (princ " ja foi definida, por isso foi agora ignorada.")
            (terpri)
            )
          )
        )
        )
      )
    )
    )
  )
  tmp-rel
)
```

-----  
; Gera um no assercao ou hipotese logo de inicio e de acordo com o valor  
; do argumento tipo-recebido. Se tipo-recebido for 'hip faz um no hipotese  
; se for 'true faz um no assercao.  
; O outro argumento recebido e uma lista de relacoes (descendentes) a partir  
; do no que acabou-se de criar. Essa lista e tal que os seus elementos sao  
; listas da forma (nome-da-relacao no-destino).  
; Segue-se uma analise de cada uma dessas listas para ver se o no destino  
; ja existe ou precisa de ser criado.  
; As relacoes ascendentes e descendentes devidas sao tambem actualizadas.  
; Devolve uma lista cujos elementos sao TODOS os nos criados (ainda nao

Artur Marques

23

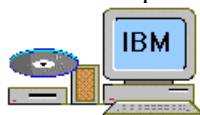


```

; existiam) durante o seu processamento. Essa lista de nos e recebida por
; def-no que os devera juntar a variavel *rede*. (ver def-no)

(defun gera-no (lista-relacoes tipo-recebido &aux id novo-no nova-rel-asc nova-rel-desc)
  (setf id (gera-nome "node" (1+ *num-nos*)))
  (setf novo-no (cria-no id tipo-recebido nil nil))
  ; formacao das relacoes ascendentes dos novos (ou ja existentes) nos para o novo no de
  identificador id.
  (do* (
    (lst lista-relacoes (cdr lst))
    (rel (caar lst) (caar lst))
    (no-id (cadar lst) (cadar lst))
    (lst-devolvida nil)
  )
    ; devolucao da lista de nos criada com o novo-no a cabeca
    ((null lst) (cons novo-no lst-devolvida))
    (setf nova-rel-asc (cria-rel (forma-inv rel) novo-no))
    (setf devolucao (existe-no no-id))
    (cond ((not devolucao)
      ; O teste seguinte e para verificar se o utilizador nao cria
      ; nos constantes com nome reservado para os nos assercao e
      ; hipotese - isto e, nome comecado por NODE.
      ; Caso isso acontecea NODE fica transformado em NOD e o resto
      ; do nome mantem-se.
      ; Tambem e verificado se nao se utilizam nomes comecados por ?
      ; isso porque, ao serem aceites esses nomes, aquando da
      ; utilizacao das funcoes de procura, os resultados obtidos
      ; nao seriam correctos caso se pretende-se criar uma variavel
      ; com nome identico ao dum no ja existente. Isto acontece
      ; porque as funcoes de procura assim que apanham um identificador
      ; comecado por ? consideram-o logo uma variavel e nao testam
      ; se ha a possibilidade de aquele nome estar ali para se referir
      ; a um no.
      ; Assim, caso o utilizador utilize um identificador comecado
      ; por ?, o ? e removido desse identificador. Se o identificador
      ; fosse so o proprio ?, entao e transformado em -?.
      (cond ((var? no-id)
        (setf no-id (tira-primeiro no-id))
      )
    )
  )
    ; Reparar que o teste tem de ser feito desta forma para o caso
    ; do utilizador entrar nomes duplamente perigosos, como e o
    ; caso de ?node33, por exemplo. Neste exemplo, o nome deve ser
    ; alterado para nod33.
    (setf no-id (comeca-por-node? no-id))
    (setf devolucao (cria-no no-id (tipo-para-novo no-id) (list nova-rel-asc) nil))
    (setf lst-devolvida (junta-fim lst-devolvida devolucao))
  )
  (t (setf (no-relacoes-asc devolucao) (junta-fim (no-relacoes-asc devolucao) nova-rel-
asc)))

```



```

)
(setf nova-rel-desc (cria-rel rel devolucao))
(setf (no-relacoes-desc novo-no) (junta-fim (no-relacoes-desc novo-no) nova-rel-
desc))
); do*
); fim gera-no

```

```

;-----
; Verifica se nao existe um conceito identico na rede.
; Retorna o nome do no onde existe um conceito igual e nil caso contrario.
; Percorre toda a rede para fazer esta averiguacao, parando a procura logo que
; encontre um conceito identico.
; Invoca a funcao descendentes-iguais? para chegar a conclusoes sobre a
; equivalencia dos conceitos.
; Recebe como argumento um no.

```

```

(defun ja-existe-conceito-igual? (no)
  (do ((lst *rede* (cdr lst))
      (no-actual nil)
      (devolve nil)
      (ja-encontrou nil))
    )
    ((or ja-encontrou (null lst)) devolve)
    (setf no-actual (car lst))
    (cond ((descendentes-iguais? (no-relacoes-desc no) (no-relacoes-desc no-actual))
          (cond ((equal (no-tipo-no no) (no-tipo-no no-actual))
                (setf ja-encontrou t)
                ; Os nos tem as mesmas relacoes descendentes, por isso e de suspeitar que se
                ; referem ao mesmo conceito. Depois se coincidirem no tipo entao acabou
                ; de ser encontrado um no igualzinho, deve-se cancelar a procura e devolver t.
                (setf devolve (no-nome-no no-actual))
                )
              )
          (t ; os nos so sao diferente no tipo, sao conceitos iguais, deve-se proceder
            ; a modificacao do tipo ja presente na rede, de hipotese para assercao.
            ; Substitui-se sempre o tipo do no presente na rede, de hipotese para o
            ; tipo do no recebido.
            (cond ((and (tipo-hip no-actual) (tipo-assert no))
                  (setf (no-tipo-no no-actual) (no-tipo-no no))
                  (setf devolve (no-nome-no no-actual))
                  )
              )
            (t (setf devolve (no-nome-no no-actual))))
            ) ; cond
          ) ; t
        ) ; cond
    ) ; do
  ) ; fim ja-existe-conceito-igual?

```

```

;-----

```

Artur Marques

25



; Recebe duas listas de relacoes.  
 ; Compara essas listas e devolve t se tiverem os mesmos elementos (mesmo que por ordem  
 ; diferente), devolvendo nil se houver um elemento (ou mais) que nao coincida.  
 ; E utilizada por ja-existe-conceito-igual? e utiliza listas-iguais?

```
(defun descendentes-iguais? (lista-d1 lista-d2 &aux lista1 lista2)
  (dolist (d1 lista-d1 t)
    (setf lista1 (cons (list (relacao-nome-rel d1) (no-nome-no (relacao-aponta-no d1)))
      lista1))
    )
  (dolist (d2 lista-d2 t)
    (setf lista2 (cons (list (relacao-nome-rel d2) (no-nome-no (relacao-aponta-no d2)))
      lista2))
    )
  (listas-iguais? lista1 lista2)
  )
)
```

-----  
 ; Recebe duas listas para comparacao, devolvendo t se as listas coincidirem quantos aos  
 ; elementos - mesmo que esses existam em ordem distinta. Devolve nil caso contrario.  
 ; E utilizada por descendentes-iguais?

```
(defun listas-iguais? (l1 l2 &aux tam1 tam2 tam-uniao)
  (let ((tam1 (length l1)) (tam2 (length l2)))
    (cond ((not (equal tam1 tam2)) nil)
          (t (let ((devolve t))
              (dolist (cabeca l1 devolve)
                (cond ((not (member cabeca l2 :test 'equal)) (setf devolve nil)))
              )
            )
          )
    )
  )
)
```

-----  
 ; Recebe o identificador do no a eliminar e retira-o da rede se ele existir e  
 ; nao tiver relacoes ascendentes para nenhum no.  
 ; Os descendentes do no sao tambem sujeitos a accao da funcao de acordo com  
 ; a seguinte filosofia:  
 ; - ao retirar um no, retiram-se tambem todos os seus descendentes que nao  
 ; sejam assercoes e que so tenham relacoes ascendentes para o no que foi  
 ; inicialmente eliminado.  
 ; Esta funcao invoca outras funcoes cujos nomes sao bastante sugestivos de  
 ; forma a reflectirem claramente que se seguiu a filosofia atras descrita.  
 ; A dada altura forma-se a lista dos descendentes directos deste no, e para  
 ; cada um dos elementos dessa lista invoca-se a funcao expulsa.  
 ; Devolve t se tiver sucesso na eliminacao do no. Da uma mensagem de erro  
 ; adequada se a eliminacao nao for possivel.



```

(defun retira-no (no-id &aux no)
  (setf no (existe-no no-id))
  (cond ((not no)
    (princ "Nao existe nenhum no com a identificacao ")
    (princ no-id)
    (princ " na rede.")
    (terpri)
  )
  (t (cond ((no-relacoes-asc no) ; existem relacoes ascendentes para o no em causa.
    (princ "Nao posso retirar o no com identificacao ")
    (princ no-id)
    (princ " porque esse tem relacoes ascendentes.")
    (terpri)
  )
  )
  (t ; o no nao tem relacoes ascendentes e por isso, pode ser retirado.
    ; O segundo parametro indica o tipo do no, importante para saber em sequencias
    ; de apagamento de nos quais os que se podem - ou nao - apagar.

    (setf descendentes-em-risco (devolve-lista-nos-apontados no))
    (setf *rede* (delete no *rede*))
    (dolist (cabeca descendentes-em-risco t)
      (expulsa cabeca (no-nome-no no))
    )
  )
  ) ; t
  ) ; cond
  ) ; t
  ) ; cond
  t ; indicar que correu conforme previsto
  ) ; fim retira-no

```

-----

; Chamada por retira-no (nao deve ser chamada pelo utilizador da rede)  
; e verifica se pode de facto eliminar o no recebido como argumento.  
; Recebe ainda o identificador do no para o qual o no argumento aponta.  
; Esse no apontado ja nao existe no entanto, porque foi submetido a accao  
; da funcao retira-no. So ha interesse em receber o seu identificador, por  
; uma questao de actualizacao das relacoes que se referiam a esse no.  
; Esta funcao tem em conta a filosofia de eliminacao de nos descrita nos  
; comentarios da funcao retira-no.

```

(defun expulsa (no id-apontado &aux descendentes-em-risco)
  (cond ( (not (tipo-assert no))
    (cond ((ascendentes-so-para-no-ja-apagado no id-apontado)
      (setf descendentes-em-risco (devolve-lista-nos-apontados no))
      (setf *rede* (delete no *rede*))
      (dolist (cabeca descendentes-em-risco t)
        (expulsa cabeca (no-nome-no no))
      )
    )
  )
  )
  (t ; no que nao e assert e tem ascendentes para outros nos, para la do apagado

```

Artur Marques

27



```

        (apaga-relacoes 'asc no id-apontado)
      )
    )
    (t ; no e do tipo assert
      (apaga-relacoes 'asc no id-apontado)
    )
  ); cond
); fim do expulsa

;-----
; Apaga relacoes ascendentes ou descendentes - conforme especificado por 'asc ou 'desc em
tipo-lista - relacoes essas
; que apontam para o no cujo identificador vem em id-apontado. Essas relacoes sao do no
recebido como argumento.
; O argumento tipo-lista vem a 'asc ou 'desc de forma a saber-se se sao para
; apagar - respectivamente - relacoes ascendentes ou descendentes.

(defun apaga-relacoes (tipo-lista no id-apontado &aux lista-rel insucesso nome-devolvido)
  (setf insucesso nil)
  (cond ((equal tipo-lista 'asc) (setf lista-rel (no-relacoes-asc no)))
        ((equal tipo-lista 'desc) (setf lista-rel (no-relacoes-desc no)))
        (t
         (princ "apaga-relacoes recebeu tipo-de-lista desconhecido: ")
         (princ tipo-lista)
         (terpri)
         (setf insucesso t)
        )
  )
  (cond ((not insucesso) ; recepcao de parametros validos
        (dolist (cabeca lista-rel t)
          (setf nome-devolvido (no-nome-no (relacao-aponta-no cabeca)))
          (cond ((equal nome-devolvido id-apontado)
                (setf lista-rel (delete cabeca lista-rel))
                )
          )
        )
        ; lista-rel fica a ser a lista de relacoes correcta para o no
        (cond ((equal tipo-lista 'asc) (setf (no-relacoes-asc no) lista-rel))
              (t (setf (no-relacoes-desc no) lista-rel))
        )
  )
  )
); fim apaga-relacoes

;-----
; Chamada por retira-no e expulsa de forma a saber se o no esta em condicoes
; de ser apagado. Esta funcao informa se o no so tinha relacoes ascendentes
; para um no entretanto ja eliminado.
; Devolve t se o no argumento so tem relacoes ascendentes para o no que ja foi apagado

```



; Devolve nil se afinal o no tinha relacoes ascendentes para outros nos

```
(defun ascendentes-so-para-no-ja-apagado (no id-apontado &aux lista-rel nome-apontado)
  (setf lista-rel (no-relacoes-asc no))
  (do ((lista lista-rel (cdr lista))
      (devolve t)
      )
      ((null lista) devolve)
      (setf nome-apontado (no-nome-no (relacao-aponta-no (car lista))))
      (cond ((not (equal nome-apontado id-apontado))
              (setf devolve nil) ; dizer que afinal o no tem relacoes ascendentes para outro no, para
              la do apagado
              (setf lista nil) ; forcar a saida, uma vez que ja se sabe o resultado
              )
            )
      )
  )
)
```

-----  
; Recebe um no e devolve a lista cujos elementos sao os seus nos descendentes  
; em primeiro grau (descendentes directos).

```
(defun devolve-lista-nos-apontados (no &aux no-actual lista-nos-apontados)
  (cond ((no-relacoes-desc no)
        (do* ((lista-relacao-actual (no-relacoes-desc no) (cdr lista-relacao-actual)))
            ((null lista-relacao-actual) lista-nos-apontados)
            (setf no-actual (relacao-aponta-no (car lista-relacao-actual)))
            (setf lista-nos-apontados (junta-fim lista-nos-apontados no-actual))
            )
        )
  )
) ; fim devolve-lista-nos-apontados
```

-----  
; Recebe um identificador dum no.  
; Devolve nil se o no nao existir na rede.  
; Devolve o no se ele existir.  
; Opcionalmente pode receber o tipo do no, no caso de existirem na rede  
; nos com o mesmo identificador, mas de tipos diferentes.

```
(defun existe-no (no-id &optional tipo-opcional)
  (cond ((null *rede*) nil)
        (t (do (
                (lista *rede* (cdr lista))
                (cabeca nil)
                (id nil)
                (tipo nil)
                (ja-encontrou nil)
                (devolve nil)
                )
          )
  )
)
```

Artur Marques

29





```

)
(t
(cond ((and (equal (string (char no-id 0)) "N")
                (equal (string (char no-id 1)) "O")
                (equal (string (char no-id 2)) "D")
                (equal (string (char no-id 3)) "E"))
)
; formar o resto da string
(cond ((= tam 4) 'NOD)
      (t
       (do ((contador 4 (1+ contador))
            (resto ""
              (concatenate 'string resto (string (char no-id contador))))
           )
         )
       ((= contador tam)
        (intern (string-upcase (concatenate 'string "NOD" resto)))
       )
      )
)
) ; t
)
); par cond / accao
(t (intern (string-upcase no-id)))
); cond
); t
); cond
)

```



## Ficheiro: PROCURA.LSP

```
-----  
; Esta funcao recebe como argumento uma lista cujos elementos sao nos  
; tal qual existem internamente na rede, isto e, sao estruturas.  
; A sua accao consiste em analisar os identificadores dos nos e devolver  
; uma lista cujos elementos sao esses mesmos identificadores.  
  
(defun forma-lista-nomes (lista-nos)  
  (cond ((null lista-nos)  
        nil  
        )  
        (t (do* ((lst lista-nos (cdr lst))  
                (cabeca (car lst) (car lst))  
                (devolucao nil)  
                )  
              ((null lst) devolucao)  
              (setf devolucao (junta-fim devolucao (no-nome-no cabeca))))  
          )  
        )  
  )  
)  
)
```

```
-----  
; Eis a funcao de procura.  
; A sua sintaxe de chamada e:  
; (procura 'rel1 'conjnos1 ... 'reln 'conjnosn)  
; Ter em conta que nos conjnos permitidos, pode vir o nome duma variavel.  
; Para assinalar o nome duma variavel, dever-se-a utilizar um identificador  
; que comece por "?".  
; Por exemplo para guardar na variavel QUAL o valor da classe a que  
; pertence a orca, poder-se-ia fazer:  
; (procura 'membro 'orca 'classe '?qual).  
; Depois para inspeccionar o valor presente em qual, dever-se-ia fazer:  
; (valor? 'qual).  
; Para saber como invocar esta funcao e sobre as suas potencialidades, devera  
; consultar o manual do utilizador da segunda fase.  
; Forma de actuacao:  
; 1) Transforma os seus argumentos num formato mais pratico, de acordo  
; com a actuacao da funcao trata-argtos.  
; 2) Recorre a funcao procura2 para obter os nos que verificam as relacoes  
; indicadas nos argumentos.  
; 3) A partir da lista de argumentos e dos nos que, desde logo sabe que  
; satisfazem as condicoes expressas aquando da chamada, passa ao  
; tratamento dos valores a associar as variaveis requisitadas.  
; 4) Forma-se assim uma lista chamada vars-a-criar, que e da forma  
; ((rel id-var1) ... (rel id-varn)) e que vai ser depois utilizada.  
; 5) Para cada um dos elementos da lista referida em 4), atribui-se-lhe os  
; valores plausiveis de acordo com a funcao forma-var.
```

Artur Marques



; Esta funcao devolve a lista dos identificadores dos nos que satisfazem  
; os argumentos.

```
(defun procura (&rest argumentos)  
  (proc argumentos)  
  )
```

; A funcao de procura acaba por ser a funcao proc.  
; Optou-se por esta divisao entre as funcoes proc e procura para facilitar  
; a implementacao da funcao procura-faz, que deste modo pode ser fortemente  
; baseada na funcao proc.

```
(defun proc (argumentos &aux devolucao nos-dest-excluidos vars-a-criar)  
  (setf argumentos (trata-argtos argumentos))  
  (cond (argumentos  
    (setf devolucao (procura2 *rede* argumentos))  
    (setf nos-dest-excluidos (car (forma-excluidos-e-vars argumentos)))  
    (setf vars-a-criar (cadr (forma-excluidos-e-vars argumentos)))  
    (do* ((lst vars-a-criar (cdr lst))  
          (cabeca (car lst) (car lst))  
          (rel (car cabeca) (car cabeca))  
          (var-id (cadr cabeca) (cadr cabeca)))  
      )  
      ((null lst) t)  
      (cond ((not (devolve-var var-id))  
            (cria-var var-id nil)  
            )  
      )  
      (setf (var-valor (devolve-var var-id))  
            (forma-var rel devolucao nos-dest-excluidos))  
      )  
    )  
  (forma-lista-nomes devolucao)  
  ) ; fecha a condicao numa lista de argumentos nao nula  
  ) ; fecha o cond que encaminhou para o teste numa lista de argumentos nao nula  
  ) ; fim procura
```

-----  
; Recebe a lista de argumentos que o utilizador utiliza para invocar  
; a funcao procura. Encarrega-se de devolver essa lista com um formato  
; mais pratico.  
; Assim se por exemplo receber (membro (a b c) classe d) , vai devolver  
; ((membro a) (membro b) (membro c) (classe d)) . Ou seja, explicita os  
; pares (relacao no-destino) o que facilita o trabalho na funcao procura.  
; Esta funcao analisa ainda a correccao da chamada feita a procura.

```
(defun trata-argtos (lista-arg &aux rel-tmp)  
  (do* ((lst lista-arg (cdr lst))  
        (cabeca (car lst) (car lst))  
        (contador 0 (1+ contador)))
```

Artur Marques

33



```

      (devolucao nil)
    )
  ((null lst) devolucao)
  (cond ((and (evenp contador) (relacao? cabeca))
        (cond ((null (cdr lst))
                (princ "ERRO: A seguir a relacao ")
                (princ cabeca)
                (princ " esperava-se um no ou conjunto de nos.")
                (terpri)
                (setf lst nil)
                (setf devolucao nil) ; garantir que devolve nil em caso
                                   ; de erro na chamada.
              )
          (t (setf rel-tmp cabeca))
        )
    )
  )
  ((not (evenp contador))
   (setf devolucao (junta-par devolucao rel-tmp cabeca))
  )
  (t (princ "ERRO: A relacao ")
     (princ cabeca)
     (princ " ainda nao foi definida !")
     (terpri)
     (setf lst nil)
     (setf devolucao nil) ; garantir que devolve nil em caso
                          ; dum chamada incorrecta.
  )
)
)
) ; fim trata-argtos

```

-----  
; Recebe o identificador dum relacao e devolve nil se nao existir relacao  
; cujo nome seja aquele identificador.

```

(defun relacao? (rel-id)
  (member rel-id *relacoes*)
) ; fim relacao?

```

-----  
; Funcao utilizada pela funcao trata-argtos e que serve para acrescentar  
; a lista recebida como argumento, novos elementos, eles proprios da forma  
; (relacao no-id). Assim se por exemplo receber rel=membro e nos=(a b), vai  
; devolver a lista argumento com os elementos (membro a) e (membro b)  
; acrescentados na sua cauda.

```

(defun junta-par (lista rel nos)
  (cond ((listp nos)
        (do* ((lst nos (cdr lst))
              (cabeca (car lst) (car lst))

```

Artur Marques

34



```

)
((null lst) lista)
(setf lista (junta-fim lista (list rel cabeca)))
)
)
(t (setf lista (junta-fim lista (list rel nos))))
)
); fim junta-par

```

-----

; Utilizada pela funcao procura e que serve para formar a devolucao da referida. De facto - nao fosse a necessidade de manipular as variaveis eventualmente envolvidas - no final desta funcao, estaria praticamente findo o trabalho da funcao procura.

; Esta funcao devolve uma lista cujos elementos sao os nos da rede que satisfazem as condicoes expressas aquando da chamada a procura.

; Recorre para determinar se os nos satisfazem - ou nao - as condicoes requisitadas - a funcao selecciona.

```

(defun procura2 (onde argumentos)
  (do* ((lst argumentos (cdr lst))
        (cabeca (car lst) (car lst))
        (devolucao onde)
        )
    ((null lst) devolucao)
    (setf devolucao (selecciona devolucao cabeca))
  )
)
)

```

-----

; Utilizada directamente por procura2.

; Recebe como argumentos:

; onde - lista cujos elementos sao nos, eventualmente ja seleccionados dum conjunto inicial de nos. Esta lista vai-se tornando mais pequena por eliminacao de nos que a certa altura estavam presentes mas deixaram de respeitar certa condicao exigida.

; par - um par (rel no-id)

; O par atras referido e utilizado para determinar os nos que satisfazem as condicoes mencionadas aquando da chamada a procura. Assim um no satisfaz a condicao representada pelo par, se tiver uma relacao com aquele nome e para o no destino correspondente. Se o no destino for uma variavel, considera-se que o no satisfaz a condicao a partir do momento que tenha a relacao em causa, independentemente dos nos apontados por essa relacao (descendente).

; E assim feita uma seleccao dos nos recebidos pelo argumento onde, atraves da condicao que o par representa.

; Devolve a lista de nos que satisfazem a condicao.

```

(defun selecciona (onde par &aux rel no)
  (setf rel (car par))

```

Artur Marques







; A segunda lista tem elementos da forma (rel var-id) e indica as variaveis  
; cuja criacao ou actualizacao, o utilizador requisitou.

```
(defun forma-excluidos-e-vars (arg)
  (do* ((lst arg (cdr lst))
        (cabeca (car lst) (car lst))
        (rel (car cabeca) (car cabeca))
        (no-id (cadr cabeca) (cadr cabeca))
        (excluidos nil)
        (vars nil)
        )
    ((null lst) (list excluidos vars))
    (cond ((not (var? no-id))
           (setf excluidos (junta-fim excluidos (list rel no-id)))
           )
          (t
           (setf vars (junta-fim vars (list rel (tira-primeiro no-id))))
           )
          )
    )
  )
); fim forma-excluidos-e-vars
```

-----  
; Recebe a lista cujos elementos sao nos. Essa lista refere a lista de  
; relacoes descendentes para certo no.  
; Analisa o seu argumento e devolve uma lista da forma:  
; ((rel no-id) ... (rel no-id)) em que rel e o identificador da relacao  
; e no-id e o identificador do no a que essa relacao descendente conduz.

```
(defun arranja-nomes (lista-rel-desc)
  (do* ((lst lista-rel-desc (cdr lst))
        (cabeca (car lst) (car lst))
        (no-id nil)
        (rel nil)
        (devolucao nil)
        )
    ((null lst) devolucao)
    (setf rel (relacao-nome-rel cabeca))
    (setf no-id (no-nome-no (relacao-aponta-no cabeca)))
    (setf devolucao (junta-fim devolucao (list rel no-id)))
    )
  )
); fim arranja-nomes
```

-----  
; Devera devolver a lista cujos elementos sao identificadores de nos.  
; Esses identificadores correspondem as possiveis instanciacoas para  
; as variaveis utilizadas numa chamada a procura.  
; Neste caso os argumentos sao:  
; rel- a relacao para a qual esta-se agora a estudar instanciacoas possiveis  
; para a variavel.

Artur Marques

38



```

; nos-ok - os nos que a procura devolveu, ou seja os que satisfizeram as
;          condicoes assinaladas na chamada a procura.
; nos-dest-exc - lista cujos elementos sao pares (rel conjnos) e que indica
;               quais os nos que nao tem hipotese de serem instanciados
;               com a variavel, pelo facto de serem conhecidos pelo
;               utilizador aquando duma chamada a procura.

```

```

(defun forma-var (rel nos-ok nos-dest-exc)
  (do* ((lst nos-ok (cdr lst))
        (cabeca (car lst) (car lst))
        (devolucao nil)
        )
    ((null lst) devolucao)
    (setf lst-desc (no-relacoes-desc cabeca))
    (setf nomes-desc (arranja-nomes lst-desc))
    (do* ((lstpares nomes-desc (cdr lstpares))
          (parcabeca (car lstpares) (car lstpares))
          )
      ((null lstpares) t)
      (cond ((and
              (not (member parcabeca nos-dest-exc :test 'equal))
              (equal rel (car parcabeca))
              )
             (cond ((not (member (cadr parcabeca) devolucao :test 'equal))
                    (setf devolucao (junta-fim devolucao (cadr
parcabeca))))
              )
            )
      )
    )
  )
) ; fim forma-var

```

```

;-----
; Esta funcao faz exactamente o mesmo que a funcao procura, excepto se nao
; existirem nos que verifiquem as condicoes requisitadas.
; Caso o no procurado nao exista, esta funcao procede a sua criacao.
; E so nesse aspecto que esta funcao difere da funcao procura, pois de resto,
; quando o no existe, devolve-o e cria eventuais variaveis, tal qual a funcao
; procura faria.
; Ter em conta que a criacao do no pode nao ser possivel caso existam
; variaveis envolvidas na chamada.
; Assim esta funcao devolve uma lista cujos elementos sao os identificadores
; dos nos que verificam as condicoes indicadas ou o identificador do no
; criado. Eventualmente procede a criacao de variaveis ou alteracao de
; variaveis ja existentes, conforme requisitado pelo utilizador.

```

```

(defun procura-faz (&rest argumentos &aux tratados resultado)
  (setf tratados (trata-argtos argumentos))

```

Artur Marques

39



```

(cond (tratados
      (setf resultado (proc argumentos))
      (cond ((sem-vars tratados) ; Nao ha variaveis, se nao existir, o no pode ser criado.
            (cond ((not resultado) ; Ainda nao existe aquele no
                  (def-no 'hip argumentos) ; cria-se o no.
                  )
              (t ; o no existe, deve-se devolver o resultado da procura.
                (proc argumentos)
                )
            )
      ) ; cond secundario
    )
    (t
      (cond (resultado resultado) ; ha variaveis e o(s) no(s) existe(m).
            (t ; ha variaveis e o no nao existe e nao pode ser criado.
              (princ "ERRO: Nao e possivel a criacao do no, pois estao
requisitadas variaveis !")
              (terpri)
            )
          )
    )
  ) ; cond fechado
) ; fecha a condicao numa lista de argumentos nao nula.
) ; fecha o cond que encaminhou para o teste numa lista de argumentos nao nula.
) ; procura-faz

```

```

;-----
; Funcao que recebe uma lista da forma ((rel no-id) ... (rel no-id)).
; Devolve nil se existir na lista recebida algum identificador de variavel.
; Devolve t se na lista recebida nao houver nenhum identificador de variavel.

```

```

(defun sem-vars (arg)
  (do* ((lst arg (cdr lst))
        (cabeca (car lst) (car lst))
        (no-id (cadr cabeca) (cadr cabeca))
        (devolucao t)
        )
    ((null lst) devolucao)
    (cond ((var? no-id)
          (setf devolucao nil)
          (setf lst nil) ; forcar a saida
          )
    )
  )
)

```



## Ficheiro: VAR.LSP

(setf \*variaveis\* nil) ; lista com as variaveis cuja criacao o utilizador  
; requisitou com chamadas a procura.

-----  
; Estrutura que representa uma variavel.  
; Considera-se assim que uma variavel consiste no respectivo identificador  
; e no valor associado. O identificador da variavel devera ser um atomo.

```
(defstruct var  
  id ; identificador da variavel  
  valor ; valor da variavel  
)
```

-----  
; Funcao bastante simples que se limita a determinar se o identificador  
; no-id, recebido como argumento, e - ou nao - um identificador reservado  
; para nomes de variaveis. Um identificador que se quer associar a uma  
; variavel devera ser comecar por "?" , embora o identificador da variavel  
; acabe depois por ignorar o ? como primeiro caracter.

```
(defun var? (no-id)  
  (setf no-id (string no-id))  
  (cond ((equal "?" (string (char no-id 0))) t)  
        (t nil)  
  )  
) ; var?
```

-----  
; A rede semantica mantem agora - na fase 2 do seu desenvolvimento - uma  
; nova variavel global - \*variaveis\* .  
; Esta variavel vai conter todas as variaveis cuja criacao tenha sido  
; motivada pela interacao do utilizador com a rede.  
; De forma a criar variaveis existe a funcao cria-var, que nao se destina  
; ao utilizador da rede, mas e antes uma funcao de nivel inferior a ser  
; utilizada por outras de nivel superior.  
; Esta funcao recebe 2 argumentos: o primeiro e o identificador que se  
; quer associado a variavel. O segundo e o valor inicial a associar a  
; variavel.

```
(defun cria-var (id valor)  
  (setf *variaveis* (junta-fim *variaveis* (make-var :id id :valor valor)))  
)
```

-----  
; Esta funcao recebe como argumento o nome duma suposta variavel.  
; Caso exista de facto uma variavel cujo identificador seja aquele nome,  
; entao essa variavel e devolvida pela funcao. Se nao existir devolve nil.

Artur Marques



; Pelo comportamento atras descrito, conclui-se que a funcao pode ser  
; utilizada para determinar a existencia duma variavel e para obtencao da  
; mesma.

```
(defun devolve-var (nome)
  (do* ((lst *variaveis* (cdr lst))
        (cabeca (car lst) (car lst))
        (devolucao nil)
        (ja-achou nil)
        )
    ((or ja-achou (null lst)) devolucao)
    (cond ((equal (var-id cabeca) nome)
            (setf ja-achou t)
            (setf devolucao cabeca)
          )
          )
  )
)
```

-----  
; Funcao que recebe o identificador duma variavel e que vai devolver - caso  
; essa variavel exista - o valor que lhe esta associado.  
; Esta funcao esta completamente destinada ao utilizador da rede, na medida  
; em que e um meio eficaz que permite consultar o valor associado as  
; variaveis criadas durante o processo de interacao com a rede semantica.

```
(defun valor? (var-id)
  (setf var-id (devolve-var var-id))
  (cond ((null var-id)
        (princ "Ainda nao foi criada nenhuma variavel com esse identificador.")
        (terpri)
        )
        (t (var-valor var-id))
  )
)
```

-----  
; Funcao destinada ao utilizador da rede atraves da shell.  
; Devolve t se o identificador recebido corresponder ao nome duma variavel.  
; Devolve nil se o identificador nao corresponder ao nome de nenhuma variavel.

```
(defun existe-var (var-id)
  (cond ((devolve-var var-id) t)
        (t nil)
  )
)
```

-----  
; Funcao definida a pensar no utilizador da shell.  
; Recebe como argumento o identificador duma suposta variavel.

Artur Marques



; Caso a variavel exista elimina-a da lista de variaveis.  
; Se nao existir informa o utilizador do erro.

```
(defun tira-var (var-id)
  (cond ((devolve-var var-id)
        (setf *variaveis* (remove (devolve-var var-id) *variaveis*))
        t
        )
        (t (princ "ERRO: Nao existe nenhuma variavel com o identificador ")
           (princ var-id)
           (princ " .")
           (terpri)
          )
        )
  )
)
```



## Ficheiro: VARIOS.LSP

```
-----  
; A funcao junta-fim, acrescenta ao final da lista especificada o elemento  
; referido. Desta forma as listas em questao, tem sempre os elementos mais  
; recentes em ultimo lugar. Ter em conta que, por si so, esta funcao nao e  
; capaz de produzir alteracoes na sua lista argumento. Ela limita-se a  
; devolver uma lista que corresponde a alteracao desejada. Para mudar a  
; lista argumento e preciso uma instrucao como (setf l (junta-fim l el)).  
  
(defun junta-cabeca (lista el)  
  (cons el lista)  
)  
  
(defun junta-fim (lista el)  
  (append lista (list el))  
)  
  
; Recebe um identificador - que e um atomo - e devolve um atomo igualzinho  
; ao recebido, excepto que tem o sinal - no fim. E utilizada para formacao  
; dos nomes das relacoes inversas.  
  
(defun forma-inv (id)  
  (intern (string-upcase (concatenate 'string (string id) (string '-))))  
)  
  
-----  
; Recebe uma string e um numero. Gera uma string que e a concatenacao da  
; string argumento com a string correspondente ao num.  
; Devolve o atomo correspondente a string gerada.  
  
(defun gera-nome (nome num)  
  (cond ((not (stringp nome))  
        (princ "Erro: Esperava-se uma string como primeiro parametro !")  
        nil  
        )  
        ((not (numberp num))  
        (princ "Erro: Esperava-se um numero inteiro como segundo parametro !")  
        nil  
        )  
        (t (setf num (princ-to-string num))  
          (setf nome (concatenate 'string nome num))  
          (intern (string-upcase nome))  
          )  
        )  
)  
)  
  
-----  
; Devolve o nome do no recebido como argumento
```

Artur Marques

44



```
(defmacro nome-no (no)
  `(no-nome-no ,no)
)
```

```
;-----
; Devolve o tipo do no recebido como argumento
```

```
(defmacro tipo-no (no)
  `(no-tipo-no ,no)
)
```

```
;-----
; Devolve a lista de relacoes ascendentes do no recebido como argumento
```

```
(defmacro relacoes-asc (no)
  `(no-relacoes-asc ,no)
)
```

```
;-----
; Devolve a lista de relacoes descendentes do no recebido como argumento
```

```
(defmacro relacoes-desc (no)
  `(no-relacoes-desc ,no)
)
```

```
;-----
; Cria um no em que o sistema acredita. O tipo do no e assercao, sendo esse
; facto assinalado no seu campo TIPO como 'true.
```

```
(defun assercao (&rest argumentos)
  (def-no 'true argumentos)
)
```

```
;-----
; Cria um no do tipo 'hip , quer dizer, o campo TIPO na representacao interna
; deste no fica com o valor hip. E desta forma que se criam os nos hipotese.
```

```
(defun build (&rest argumentos)
  (def-no 'hip argumentos)
)
```

```
;-----
; Descreve o no e todos os nos que directa ou indirectamente sao apontados
; por relacoes descendentes pelo no argumento. Opta - tratando-se duma
; descricao textual - por nao repetir a escrita dos nos que ja tenham sido
; descritos, mesmo que sejam atingidos por caminhos diferentes.
```

```
(defmacro descreve (no-id)
  `(chama-info ,no-id)
```





```
(defun acrescenta (result relacao lista)
  (dolist (el lista result)
    (setf result (junta-fim result (list relacao el)))
  )
)
```

