UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Guia de programació en CLIPS

Extractes de la guia de programació bàsica per a SIN

Alfons Juan

DSIC

Departament de Sistemes Informàtics i Computació

Índex

Pròleg: història i documentació de CLIPS

- 1984: el grup d'IA del *NASA's Johnson Space Center* decideix desenvolupar una eina C de construcció de sistemes experts
- 1985: es desenvolupa la versió prototip de ***C Language Integrated Production System (CLIPS)***, idònia per a formació
- 1986: CLIPS es comparteix amb grups externs
- 1987–2002: millores de rendiment i noves funcionalitats; per exemple, programació procedural, OO i interfícies gràfiques
- 2008–2020: Gary Riley manté CLIPS fora de la NASA [1, 2]
- Documentació:
  - ***Manual de referència I: Guia de programació bàsica*** [3]
  - Manual de referència II: Guia de programació avançada [4]
  - Manual de referència III: Guia d'interfícies [5]
  - Guia de l'usuari [6]

1 Introducció

- Presentació basada en la ***Guia de programació bàsica*** [3]:
  - ***Mateixa numeració de seccions que la guia!***
  - Exclou seccions innecessàries en SIN
  - Per a Linux amb execució batch (no interactiva)
  - Secció 2: resum de CLIPS i terminologia bàsica
    - ↦ La secció 2.1 inclou guia ràpida extra amb “hola.clp”
  - Secs. 4-7: constructors deffacts, defrule, defglobal i deffunction
  - Secció 12: accions i funcions CLIPS
  - Secció 13: ordres típicament interactives

2 Resum de CLIPS

2.1 Interacció amb CLIPS i guia ràpida

- *Ordres des de la línia d'ordres:*

```
1 $ clips
2          CLIPS (6.30 3/17/15)
3 CLIPS> (+ 3 4)                          ; cridada a funció
4 7
5 CLIPS> (exit)                           ; o Ctrl-C
```

- ***Entrada i càrrega d'ordres automàtica:***

  `clips [-f <f.clp>|-f2 <f.clp>|-l <f.clp>]`

  - `-f <f.clp>`: CLIPS executa les ordres del fitxer `<f.clp>`
  - `-f2 <f.clp>`: Com `-f`, peró les ordres no es mostren
  - `-l <f.clp>`: CLIPS fa `(load <f.clp>)` inicialment
  - ***Gastarem -f2 !***

- ***CLIPS*** és una eina per a construir SBRs amb tres components:

1. ***Base de fets (BF):***
   - Cada estat del problema sol representar-se amb un únic fet d'acord amb un cert patró de ***fet-estat*** que caldrà definir
   - A cada pas d'execució, els fets-estat de la BF representen estats del problema ja explorats o pendents d'exploració
   - La resta de fets conté informació ***estàtica*** del problema
2. ***Base de regles (BR):***
   - Cada possible acció aplicable a un o més estats del problema sol representar-se amb una única regla `esquerra=>dreta`
   - La part esquerra tria el conjunt d'estats al qual és aplicable
   - La part dreta sol resultar en nous fets-estat afegits a la BF
3. ***Motor d'inferència:*** instanciació, selecció i execució de regles

- ***Motor d'inferència:***
  - ***Entrada:*** base de fets i base de regles inicials, BF i BR
  - ***Eixida:*** base de fets final, BF
  - ***Mètode:***

    $CC = \emptyset$ // conjunt conflicte d'instàncies de regles

    repetir

      // afegim noves instàncies al CC a partir de nous fets:
      $CC =$ Instancia(BF, BR, CC)

      si $CC = \emptyset$: **eixir** // objectiu no aconseguit

      // seleccionem una instància amb algun criteri:
      $InstRule =$ Selecciona(CC)

      // executem $InstRule$ i actualitzem BF i CC:
      $(BF, CC) =$ Executa$(BF, CC, InstRule)$

    fins_a objectiu aconseguit

- ***Tres passos bàsics en inferència:***

1. ***Instancia:*** afegeix noves instàncies al CC a partir de nous fets, sense repetir instàncies afegides anteriorment ***(refracció)***

2. ***Selecciona:*** aplica un criteri de selecció com ara:
   - ***Profunditat:*** primer la instància més recent
   - ***Amplària:*** primer la instància més antiga
   - ***Prioritat:*** primer la instància de la regla més prioritària

3. ***Executa:*** aplica les ordres de la instància seleccionada:
   - ***Eliminació de fets*** en la BF
   - ***Eliminació d'instàncies*** en el CC amb fets eliminats
   - ***Inserció de fets*** nous en la BF ***sense repeticions***

▸ *Un SBR senzill: hola.clp*

2.1.hola.clp

```
1  (deffacts bf (pendent Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendent ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendent $?y)))
8  (defrule acaba (pendent) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

clips -f2 2.1.hola.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (pendent Manel Nora Laia)
3  ==> Activation 0      saluda: f-1
4  Hola Manel
5  <== f-1     (pendent Manel Nora Laia)
6  ==> f-2     (pendent Nora Laia)
7  ==> Activation 0      saluda: f-2
8  Hola Nora
9  <== f-2     (pendent Nora Laia)
10 ==> f-3     (pendent Laia)
11 ==> Activation 0      saluda: f-3
12 Hola Laia
13 <== f-3     (pendent Laia)
14 ==> f-4     (pendent)
15 ==> Activation 0      acaba: f-4
```

▸ *hola.clp sense eliminació de fets: hola2.clp*

2.1.hola2.clp

```
1  (deffacts bf (pendent Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendent ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6  ;;  (retract ?f)
7    (assert (pendent $?y)))
8  (defrule acaba (pendent) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

clips -f2 2.1.hola2.clp

```
1  <== f-0     (initial-fact)
2  ==> f-0     (initial-fact)
3  ==> f-1     (pendent Manel Nora Laia)
4  ==> Activation 0      saluda: f-1
5  Hola Manel
6  ==> f-2     (pendent Nora Laia)
7  ==> Activation 0      saluda: f-2
8  Hola Nora
9  ==> f-3     (pendent Laia)
10 ==> Activation 0      saluda: f-3
11 Hola Laia
12 ==> f-4     (pendent)
13 ==> Activation 0      acaba: f-4
```

2.3 Elements de programació bàsics

2.3.1 Tipus de dades

- ***nombre:*** ***sencer (integer)*** i ***real (float)***
  `237 +12 15.09 -32.3e-7`

- ***símbol:*** seqüència de caràcters imprimibles fins a ***delimitador***
  `foo bad_value 127A 456-93-039 @+=-\%`

- ***cadena:*** `"foo" "a and b" "1 number"`

- ***fet:*** llista de valors atòmics referenciats per posició o nom;
  adreça de fet: `<Fact-XXX>` on `XXX` és l'índex del fet

- ***valor:*** ***únic*** (camp) o ***multicamp***
  `(a) (1 bar foo) () (x 3.0 "red" 567)`

2.3.2 Funcions

- Codi amb nom que retorna un valor (***funció***) o no (***ordre***):
  - ***Definides per l'usuari en CLIPS:*** `deffunction`
  - ***Predefinides [3, ap. H]:***
    `!= * ** + - / < <= <> = >= >= abs acos ...`
  - ***Cridades:*** en notació prefix, `(+ 3 4 5)`

2.3.3 Constructors

- `defglobal:` definició de variables globals
- `deffacts:` fets automàticament inserits amb `reset`
- `deffunction:` funcions definides per l'usuari
- `defrule:` definició de regles

2.4 Abstracció de dades

2.4.1 Fets

- ***Ordenats:*** llista de símbols entre parèntesis on el primer (distint de `test`, `and`, etc.) indica la "relació" `(compra all oli)`
- ***Ordres:*** `assert retract`
- La inserció d'un fet repetit no té efecte (s'ignora)
- L'índex o adreça d'un fet pot obtenir-se en la part esquerra d'una regla o com a valor retornat d'`assert`
- ***Fets inicials:*** amb `deffacts`

2.4.3 Variables globals

Es defineixen amb `defglobal`

2.5 Representació del coneixement

2.5.1 Coneixement heurístic: regles

- ***Regles:*** consten de dues parts, la LHS i la RHS
  - ***Antecedent o part esquerra (LHS):***
    - ↦ Condicions a complir perquè s'execute la RHS
    - ↦ ***Patró:*** tipus de condició molt important
  - ***Conseqüent o part dreta (RHS):***
    - ↦ Accions a executar si es compleix la LHS
- ***Motor de inferència:*** fa el ***pattern matching*** (encaix de patrons)
- ***Estratègia de resolució de conflictes:*** decideix quina regla executa si hi ha més d'una aplicable

2.5.2 Coneixement procedimental

- *Codi com el dels llenguatges convencionals:*
  - *Funcions definides per l'usuari en CLIPS:* `deffunction`

4 Constructor deffacts

- ***deffacts*** insereix (o reconstrueix) la llista de fets amb **`reset`**

```
1 (deffacts <deffacts-name> [<comment>] <RHS-pattern>*)
```

  - *Exemple:*

4.deffacts.clp

```
1 (deffacts bf (pendent Manel Nora Laia))
2 (watch facts)
3 (reset)
4 (exit)
```

clips -f2 4.deffacts.clp

```
1 <== f-0     (initial-fact)
2 ==> f-0     (initial-fact)
3 ==> f-1     (pendent Manel Nora Laia)
```

5 Constructor defrule

- ***Regla:***
  - Condicions i accions a executar si les condicions es compleixen
  - S'executa o dispara (*fire*) en funció de l'existència o no de fets amb els quals es complisquen les condicions
  - El motor d'inferència és l'encarregat d'encaixar (fer *matching* de) fets amb regles
    - ↦ Anomenem ***instàncies*** d'una regla als diferents matchings de fets amb la regla que es puguen fer (zero, un o més)
    - ↦ ***Agenda o conjunt conflicte:*** conjunt de instàncies de totes les regles pendents d'execució

5.1 Definint regles

- ***defrule*** defineix una regla amb:

```
1 (defrule <rule-name> [<comment>]
2   [<declaration>]                ; Rule Properties
3   <conditional-element>*         ; Left-Hand Side (LHS)
4   =>
5   <action>*)                     ; Right-Hand Side (RHS)
```

- *Exemple:*

5.1.defrule.clp

```
1 (defrule esquerra
2   (robot ?x ?y)
3   =>
4   (assert (robot (- ?x 1) ?y)))
```

5.2 Cicle bàsic d'execució de regles

- ***Motor d'inferència:*** bucle amb els següents passos bàsics

a) ***Selecció d'una instància (de regla) de l'agenda***

- Si no hi ha cap instància en l'agenda, s'acaba

b) ***Execució de la part dreta de la regla seleccionada***

c) ***Activació i desactivació d'instàncies de regles*** com a conseqüència de l'execució de la regla seleccionada

- Les activades s'afegeixen a l'agenda
- Les desactivades s'eliminen de l'agenda

d) ***Re-avaluació de prioritats dinàmiques d'instàncies en l'agenda:*** si utilitzem prioritats dinàmiques amb `salience`

5.3 Estratègies de resolució de conflictes

- ***Ordenació d'instàncies de regles en l'agenda:***
  - ***Per prioritat:*** les noves instàncies se situen damunt (davant) de les de menor prioritat i baix (darrere) de les de major prioritat
  - ***En cas d'empat a prioritat:*** apliquem l'***estratègia de resolució de conflictes*** actual, com ara "les més noves davant"
    - ↦ En cas d'empat a prioritat i amb dos o més instàncies activades al mateix temps (per una mateixa inserció o esborrat d'un fet), potser no podem ordenar-les amb la dita estratègia
      - Aleshores, s'ordenen arbitràriament (*no aleatòriament*), en general d'acord amb l'ordre de definició de les regles, com ara "les definides més noves (últimes definides) davant"

5.3.1 Profunditat (*depth*)

- Les noves instàncies se situen damunt de totes les d'igual prioritat; és l'estratègia per omissió

5.3.2 Amplària (*breadth*)

- Les noves regles se situen baix de totes les de igual prioritat

5.4 Sintaxi de la LHS

- ***Elements condicionals (CEs):*** sèrie de zero, un o més elements de que consta la LHS d'una regla i que s'han de satisfer perquè s'afegisca una instància de la regla a l'agenda

- Hi ha huit tipus de CEs, però només fem ús de cinc:
  - ***CEs patró:*** restriccions sobre els fets que el satisfan
  - ***CEs test:*** avaluen expressions durant l'encaix de patrons
  - ***CEs or:*** donat un grup de CEs, almenys un s'ha de satisfer
  - ***CEs and:*** donat un grup de CEs, tots s'han de satisfer
  - ***CEs not:*** donat un CE, *no* s'ha de satisfer

```
1  <conditional-element> ::=
2    <pattern-CE> | <assigned-pattern-CE> |
3    <test-CE> | <or-CE> | <and-CE> | <not-CE>
```

5.4.1 CE patró

- ***CE patró:*** llista ordenada amb un símbol inicial, seguit de constants, comodins i variables, potser precedida d'una adreça de patró
  - ***Restriccions literals:*** constants
  - ***Comodins:***
    - ***Mono-avaluats:*** `?`
    - ***Multi-avaluats:*** `$?`
  - ***Variables:***
    - ***Mono-avaluades:*** `?<var>`
    - ***Multi-avaluades:*** `$?<var>`
  - ***Restriccions de valor de retorn:*** `=<func>`
  - ***Adreces de patró:*** `?<var> <- <CE patró>`

▸ *Restriccions literals:* constants

5.4.1.literals.clp

```
1  (deffacts data-facts
2  (data 1.0 blue "red")
3  (data 1 blue)
4  (data 1 blue red)
5  (data 1 blue RED)
6  (data 1 blue red 6.9))
7  (defrule find-data (data 1 blue red) =>)
8  (watch facts)
9  (watch activations)
10 (reset)
11 (exit)
```

clips -f2 5.4.1.literals.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (data 1.0 blue "red")
3  ==> f-2     (data 1 blue)
4  ==> f-3     (data 1 blue red)
5  ==> Activation 0      find-data: f-3
6  ==> f-4     (data 1 blue RED)
7  ==> f-5     (data 1 blue red 6.9)
```

- ***Comodins mono-avaluats i multi-avaluats:*** `?` encaixa amb un camp exactament i `$?` amb zero o més

5.4.1.comodins.clp

```
1  (deffacts data-facts
2  (data 1.0 blue "red")
3  (data 1 blue)
4  (data 1 blue red)
5  (data 1 blue RED)
6  (data 1 blue red 6.9))
7  (defrule find-data (data ? blue red $?) =>)
8  (watch facts)
9  (watch activations)
10 (reset)
11 (exit)
```

clips -f2 5.4.1.comodins.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (data 1.0 blue "red")
3  ==> f-2     (data 1 blue)
4  ==> f-3     (data 1 blue red)
5  ==> Activation 0      find-data: f-3
6  ==> f-4     (data 1 blue RED)
7  ==> f-5     (data 1 blue red 6.9)
8  ==> Activation 0      find-data: f-5
```

- ***Variables mono-avaluades i multi-avaluades:*** `?<var>` encaixa amb un camp exactament i `$?<var>` amb zero o més

5.4.1.variables.clp

```
1  (deffacts data-facts (data 1 blue) (data 1 blue red)
2    (data 1 blue red 6.9))
3  (defrule find-data-1
4    (data ?x $?y ?z)
5    => (printout t "?x=" ?x " $?y=" $?y " ?z=" ?z crlf ))
6  (watch facts)
7  (watch activations)
8  (set-strategy breadth) ; per omissió és depth
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.1.variables.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (data 1 blue)
3  ==> Activation 0      find-data-1: f-1
4  ==> f-2     (data 1 blue red)
5  ==> Activation 0      find-data-1: f-2
6  ==> f-3     (data 1 blue red 6.9)
7  ==> Activation 0      find-data-1: f-3
8  ?x=1 $?y=() ?z=blue
9  ?x=1 $?y=(blue) ?z=red
10 ?x=1 $?y=(blue red) ?z=6.9
```

▸ Restriccions de valor de retorn: =<func>

5.4.1.retorn.clp

```
1 (deffacts bf (xy 1 2) (xy 3 3) (xy 4 8) (xy 5 8))
2 (defrule troba-doble (xy ?x =(* 2 ?x)) =>
3   (printout t "?x=" ?x crlf))
4 (watch facts)
5 (watch activations)
6 (reset)
7 (run)
8 (exit)
```

clips -f2 5.4.1.retorn.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (xy 1 2)
3 ==> Activation 0      troba-doble: f-1
4 ==> f-2     (xy 3 3)
5 ==> f-3     (xy 4 8)
6 ==> Activation 0      troba-doble: f-3
7 ==> f-4     (xy 5 8)
8 ?x=4
9 ?x=1
```

▸ *Adreces de patró:* `?<var> <- <pattern-CE>`

5.4.1.adreces.clp

```
1 (deffacts bf (color roig) (color verd))
2 (defrule troba-color
3   ?f <- (color ?c)
4   => (printout t ?c " trobat en el fet " ?f crlf))
5 (watch facts)
6 (watch activations)
7 (reset)
8 (run)
9 (exit)
```

clips -f2 5.4.1.adreces.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (color roig)
3 ==> Activation 0      troba-color: f-1
4 ==> f-2     (color verd)
5 ==> Activation 0      troba-color: f-2
6 verd trobat en el fet <Fact-2>
7 roig trobat en el fet <Fact-1>
```

5.4.2 CE test

- `(test <func>)` se satisfà si `<func>` no retorna fals

5.4.2.test.clp

```
1  (deffacts bf (tenim 6 plats) (tenim 5 gots))
2  (defrule tenim-mes
3    (tenim ?n ?x)
4    (tenim ?m ?y)
5    (test (> ?n ?m))
6    =>
7    (printout t "Tenim més " ?x " que " ?y crlf))
8  (watch facts)
9  (watch activations)
10 (reset)
11 (run)
12 (exit)
```

clips -f2 5.4.2.test.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (tenim 6 plats)
3 ==> f-2     (tenim 5 gots)
4 ==> Activation 0      tenim-mes: f-1,f-2
5 Tenim més plats que gots
```

5.4.3 CE or

- `(or <CE>+)` se satisfà si qualsevol dels `<CE>+` ho fa

5.4.3.or.clp

```
1  (deffacts bf (obstacle 5 3) (robot 4 3))
2  (defrule obstacle-al-costat
3    (robot ?x ?y)
4    (or (obstacle =(- ?x 1) ?y) (obstacle =(+ ?x 1) ?y))
5    =>
6    (printout t "Tenim obstacle al costat" crlf))
7  (watch facts)
8  (watch activations)
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.3.or.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (obstacle 5 3)
3  ==> f-2     (robot 4 3)
4  ==> Activation 0      obstacle-al-costat: f-2,f-1
5  Tenim obstacle al costat
```

5.4.4 CE and

- `(and <CE>+)` se satisfà si tots els `<CE>+` ho fan

5.4.4.and.clp

```
1  (deffacts bf (obstacle 3 3) (obstacle 5 3) (robot 4 3))
2  (defrule blocat-pels-costats
3    (robot ?x ?y)
4    (and (obstacle =(- ?x 1) ?y) (obstacle =(+ ?x 1) ?y))
5    =>
6    (printout t "Robot blocat pels costats" crlf))
7  (watch facts)
8  (watch activations)
9  (reset)
10 (run)
11 (exit)
```

clips -f2 5.4.4.and.clp

```
1  ==> f-0     (initial-fact)
2  ==> f-1     (obstacle 3 3)
3  ==> f-2     (obstacle 5 3)
4  ==> f-3     (robot 4 3)
5  ==> Activation 0      blocat-pels-costats: f-3,f-1,f-2
6  Robot blocat pels costats
```

5.4.5 CE not

- (not <CE>+) se satisfà si <CE> no ho fa

5.4.5.not.clp

```
1 (deffacts bf (obstacle 1 3) (robot 4 3))
2 (defrule esquerra
3   (robot ?x ?y) (not (obstacle =(- ?x 1) ?y))
4   => (assert (robot (- ?x 1) ?y)))
5 (watch facts)
6 (watch activations)
7 (reset)
8 (run)
9 (exit)
```

clips -f2 5.4.5.not.clp

```
1 ==> f-0     (initial-fact)
2 ==> f-1     (obstacle 1 3)
3 ==> f-2     (robot 4 3)
4 ==> Activation 0      esquerra: f-2,
5 ==> f-3     (robot 3 3)
6 ==> Activation 0      esquerra: f-3,
7 ==> f-4     (robot 2 3)
```

5.4.10 Declaració de propietats de regla

- Amb `(salience <sencer>)` definim la prioritat de la regla
  - ***Prioritat mínima:*** -10 000
  - ***Prioritat per omissió:*** 0
  - ***Prioritat màxima:*** 10 000

5.4.10.salience.clp

```
1 (deffacts bf (fet-estat a b))
2 (defrule obj
3   (declare (salience 1)) ; entre -10000 i +10000; 0 per omissió
4   (fet-estat a b)
5   =>
6   (printout t "Solucio trobada!" crlf))
7 (reset)
8 (run)
9 (exit)
```

clips -f2 5.4.10.salience.clp

```
1 Solucio trobada!
```

6 Constructor defglobal

- `defglobal` defineix una ***variable global*** i li dona valor:

```
1 (defglobal [<defmodule-name>] <global-assignment>*)
2 <global-assignment> ::= <global-variable> = <expression>
3 <global-variable>   ::= ?*<symbol>*
```

- ***Perill d'ús inapropiat pels programadors principiants:***
  - A diferència de la inserció o esborrat de fets, la modificació de variables globals no provoca l'activació o desactivació d'instàncies de regles; són memòria al marge de la cerca en arbre
  - Les utilitzem per a comptar nodes (fets de l'arbre de cerca) inserits o limitar la profunditat de l'arbre de cerca; poc més

6.defglobal.clp

```
1  (defglobal ?*N* = 0)
2  (deffacts bf (L a b a b a))
3  (defrule R
4    ?f <-(L ?x $?y ?x $?z)
5    =>
6    (printout t ?x" "?y" "?z
     ↪  crlf)
7    (retract ?f)
8    (assert (L $?y ?x $?z))
9    (bind ?*N* (+ ?*N* 1)))
10 (watch facts)
11 (watch activations)
12 (watch globals)
13 (set-strategy breadth)
14 (reset)
15 (run)
16 (printout t "N=" ?*N* crlf)
17 (exit)
```

clips -f2 6.defglobal.clp

```
1  :== ?*N* ==> 0 <== 0
2  ==> f-0     (initial-fact)
3  ==> f-1     (L a b a b a)
4  ==> Activation 0      R: f-1
5  ==> Activation 0      R: f-1
6  a (b a b) ()
7  <== f-1     (L a b a b a)
8  <== Activation 0      R: f-1
9  ==> f-2     (L b a b a)
10 ==> Activation 0      R: f-2
11 :== ?*N* ==> 1 <== 0
12 b (a) (a)
13 <== f-2     (L b a b a)
14 ==> f-3     (L a b a)
15 ==> Activation 0      R: f-3
16 :== ?*N* ==> 2 <== 1
17 a (b) ()
18 <== f-3     (L a b a)
19 ==> f-4     (L b a)
20 :== ?*N* ==> 3 <== 2
21 N=3
```

7 Constructor deffunction

- `deffunction` defineix funcions d'usuari

```
1 (deffunction <name> [<comment>]
2   (<regular-parameter>* [<wildcard-parameter>])
3   <action>*)
4 <regular-parameter>  ::= <single-field-variable>
5 <wildcard-parameter> ::= <multifield-variable>
```

  - ***Zero o més variables mono-avaluades:*** en primer lloc té zero o més paràmetres convencionals, tots ells variables mono-avaluades, per la qual cosa hem de passar-li tants arguments com variables mono-avaluades tinga

  - ***Seguides d'una variable multi-avaluada opcional:*** en últim lloc té una variable multi-avaluada opcional; si la té, podem passar-li tants arguments addicionals com vullgam

7.deffunction.clp

```
1 (deffunction print-args (?a ?b $?c)
2   (printout t ?a " " ?b " and " (length ?c) " extras: " ?c
    ↪  crlf))
3 (print-args 1 2)
4 (print-args a b c d)
5 (exit)
```

clips -f2 7.deffunction.clp

```
1 1 2 and 0 extras: ()
2 a b and 2 extras: (c d)
```

12 Accions i funcions

- ***Accions, funcions i ordres:*** en realitat són tot funcions que es poden utilitzar en regles, funcions definides per l'usuari o en la línia d'ordres

- Triem un o altre nom només per matisos:

  - ***Funció:*** sol referir-se a una funció que torna un valor

  - ***Acció:*** funció que no torna cap valor però que realitza alguna operació bàsica com a efecte secundari (`printout`)

  - ***Ordre:*** funció que sol utilitzar-se en la línia d'ordres i no torna cap valor (`reset`) o potser sí (`set-strategy`)

12.1 Funcions predicat

12.1.proves.clp

```
1  (numberp 23)                    ; prova de nombre sencer/real
2  (floatp 3.0)                    ; prova de real
3  (integerp 3)                    ; prova de sencer
4  (lexemep SIN)                   ; prova de cadena o símbol
5  (stringp "SIN")                 ; prova de cadena
6  (symbolp SIN)                   ; prova de símbol
7  (evenp 2)                       ; prova de nombre parell
8  (oddp 3)                        ; prova de nombre imparell
9  (multifieldp (create$ a b))     ; prova de multicamp
10 (exit)
```

clips -f 12.1.proves.clp

```
1  CLIPS> (numberp 23)                    ; prova de nombre sencer/real
2  TRUE
3  CLIPS> (floatp 3.0)                    ; prova de real
4  TRUE
5  CLIPS> (integerp 3)                    ; prova de sencer
6  TRUE
7  CLIPS> (lexemep SIN)                   ; prova de cadena o símbol
8  TRUE
9  CLIPS> (stringp "SIN")                 ; prova de cadena
10 TRUE
11 CLIPS> (symbolp SIN)                   ; prova de símbol
12 TRUE
13 CLIPS> (evenp 2)                       ; prova de nombre parell
14 TRUE
15 CLIPS> (oddp 3)                        ; prova de nombre imparell
16 TRUE
17 CLIPS> (multifieldp (create$ a b))     ; prova de multicamp
18 TRUE
19 CLIPS> (exit)
```

12.1.comparacions.clp

```
1 (eq foo foo foo foo)   ; TRUE si 1r arg igual a resta
2 (neq foo bar yak bar)  ; TRUE si 1r arg distint a resta
3 (= 3 3.0)              ; TRUE si 1r nombre igual a resta
4 (<> 4 4.1)             ; TRUE si 1r nombre distint a resta
5 (> 5 4 3)              ; TRUE si args en ordre decreixent
6 (>= 5 5 3)             ; TRUE si args en ordre no creixent
7 (< 3 4 5)              ; TRUE si args en ordre creixent
8 (<= 3 5 5)             ; TRUE si args en ordre no decreix.
9 (exit)
```

clips -f 12.1.comparacions.clp

```
1  CLIPS> (eq foo foo foo foo)   ; TRUE si 1r arg igual a resta
2  TRUE
3  CLIPS> (neq foo bar yak bar)  ; TRUE si 1r arg distint a resta
4  TRUE
5  CLIPS> (= 3 3.0)              ; TRUE si 1r nombre igual a resta
6  TRUE
7  CLIPS> (<> 4 4.1)             ; TRUE si 1r nombre distint a resta
8  TRUE
9  CLIPS> (> 5 4 3)              ; TRUE si args en ordre decreixent
10 TRUE
11 CLIPS> (>= 5 5 3)             ; TRUE si args en ordre no creixent
12 TRUE
13 CLIPS> (< 3 4 5)              ; TRUE si args en ordre creixent
14 TRUE
15 CLIPS> (<= 3 5 5)             ; TRUE si args en ordre no decreix.
16 TRUE
17 CLIPS> (exit)
```

12.1.logiques.clp

```
1 (and TRUE (> 2 1))      ; TRUE si tots els args son TRUE
2 (or FALSE (> 2 1))      ; TRUE si qualsevol arg es TRUE
3 (not (evenp 3))         ; TRUE si arg fals
4 (exit)
```

clips -f 12.1.logiques.clp

```
1 CLIPS> (and TRUE (> 2 1))      ; TRUE si tots els args son TRUE
2 TRUE
3 CLIPS> (or FALSE (> 2 1))      ; TRUE si qualsevol arg es TRUE
4 TRUE
5 CLIPS> (not (evenp 3))         ; TRUE si arg fals
6 TRUE
```

12.2 Funcions multicamp

12.2.clp

```
(create$ a b)                      ; crea valor multicamp
(nth$ 2 (create$ a b))             ; n-esim camp del multicamp
(member$ b (create$ a b b))        ; posicio(ns) de valor en mcamp
(member$ (create$ b b) (create$ a b b))
(member$ c (create$ a b b))
(subsetp (create$ b a) (create$ a b b)) ; mcamp1 en mcamp2?
(subsetp (create$ A) (create$ a b b))
(delete$ (create$ a b b) 2 3) ; esborra mcamp de pos1 a pos2
(explode$ "a b")                   ; explota cadena a mcamp
(implode$ (create$ a b))           ; implota mcamp a cadena
(subseq$ (create$ a b b) 2 3) ; extreu mcamp de p1 a p2
(replace$ (create$ a b b) 2 3 B) ; subs mcamp-p1-p2 per valor
(insert$ (create$ a b b) 2 B)     ; ins en mcap-pos valor
(first$ (create$ a b c))           ; 1r camp de mcamp
(rest$ (create$ a b c))            ; resta de mcamp (= esborra 1r)
(length$ (create$ a b c))          ; llargaria de mcamp
(delete-member$ (create$ a b a c) b a) ; esborra vals d mcamp
(delete-member$ (create$ a b a c b a) (create$ b a))
(replace-member$ (create$ a x a y) z x y) ; subs v2- x v1
(exit)
```

```
CLIPS> (create$ a b)                    ; crea valor multicamp
(a b)
CLIPS> (nth$ 2 (create$ a b))           ; n-esim camp del multicamp
b
CLIPS> (member$ b (create$ a b b))      ; posicio(ns) de valor en mcamp
2
CLIPS> (member$ (create$ b b) (create$ a b b))
(2 3)
CLIPS> (member$ c (create$ a b b))
FALSE
CLIPS> (subsetp (create$ b a) (create$ a b b)) ; mcamp1 en mcamp2?
TRUE
CLIPS> (subsetp (create$ A) (create$ a b b))
FALSE
CLIPS> (delete$ (create$ a b b) 2 3) ; esborra mcamp de pos1 a pos2
(a)
CLIPS> (explode$ "a b")                 ; explota cadena a mcamp
(a b)
CLIPS> (implode$ (create$ a b))         ; implota mcamp a cadena
"a b"
CLIPS> (subseq$ (create$ a b b) 2 3) ; extreu mcamp de p1 a p2
(b b)
CLIPS> (replace$ (create$ a b b) 2 3 B) ; subs mcamp-p1-p2 per valor
(a B)
CLIPS> (insert$ (create$ a b b) 2 B)    ; ins en mcap-pos valor
(a B b b)
CLIPS> (first$ (create$ a b c))         ; 1r camp de mcamp
(a)
CLIPS> (rest$ (create$ a b c))          ; resta de mcamp (= esborra 1r)
(b c)
CLIPS> (length$ (create$ a b c))        ; llargaria de mcamp
3
CLIPS> (delete-member$ (create$ a b a c) b a) ; esborra vals d mcamp
(c)
CLIPS> (delete-member$ (create$ a b a c b a) (create$ b a))
(a c)
CLIPS> (replace-member$ (create$ a x a y) z x y) ; subs v2- x v1
(a z a z)
```

12.3 Funcions per a cadenes

12.3.clp

```
1  (str-cat "cad" 1 sim 3.1)   ; crea cadena per concatenacio
2  (sym-cat "cad" 1 sim 3.1)   ; crea simbol per concatenacio
3  (sub-string 2 3 "abc")      ; extreu subcadena entre posicions
4  (str-index "bc" "abcbc")    ; index de cad1 en cad2 (1a ocur.)
5  (eval "(+ 3 4)")            ; avalua cad com una funcio
6  (build "(defrule R (a)=>(assert(b)))") ; avalua constructor
7  (rules)
8  (lowcase "HolA")
9  (str-compare "cad" "cad")    ; compara cads i sims (0 si =)
10 (str-compare "cada" "cadb") ; -1 si la 1a es menor
11 (str-compare "cadb" "cada") ;  1 si la 1a es major
12 (str-length "abcd")         ; llargaria de cadena o simbol
13 (check-syntax "(defrule R =>)") ; comprova sintaxi; FALSE=ok
14 (string-to-field "3.4") ; conversio de cad/sim a tipus basic
15 (exit)
```

clips -f 12.3.clp

```
CLIPS> (str-cat "cad" 1 sim 3.1)  ; crea cadena per concatenacio
"cad1sim3.1"
CLIPS> (sym-cat "cad" 1 sim 3.1)  ; crea simbol per concatenacio
cad1sim3.1
CLIPS> (sub-string 2 3 "abc")     ; extreu subcadena entre posicions
"bc"
CLIPS> (str-index "bc" "abcbc")   ; index de cad1 en cad2 (1a ocur.)
2
CLIPS> (eval "(+ 3 4)")           ; avalua cad com una funcio
7
CLIPS> (build "(defrule R (a)=>(assert(b)))") ; avalua constructor
TRUE
CLIPS> (rules)
R
For a total of 1 defrule.
CLIPS> (lowcase "HolA")
"hola"
CLIPS> (str-compare "cad" "cad")   ; compara cads i sims (0 si =)
0
CLIPS> (str-compare "cada" "cadb") ; -1 si la 1a es menor
-1
CLIPS> (str-compare "cadb" "cada") ;  1 si la 1a es major
1
CLIPS> (str-length "abcd")         ; llargaria de cadena o simbol
4
CLIPS> (check-syntax "(defrule R =>)") ; comprova sintaxi; FALSE=ok
FALSE
CLIPS> (string-to-field "3.4") ; conversio de cad/sim a tipus basic
3.4
```

12.4 Sistema d'entrada/eixida

- *Noms lògics:* `stdin stdout ...`

12.4.clp

```
1  (printout t ":)" crlf)   ; (printout <nomlogic> <expressio>*)
2  (open "xy" f "w")        ; (open <nomf> <nomlogic> [<mode>])
3  (printout f "x y" crlf)
4  (close f)                ; (close [<nomlogic>])
5  (system "cat xy")
6  (open "xy" f)            ; mode per omissio: "r"
7  (read f)
8  (read f)
9  (read f)
10 (close f)
11 (open "xy" f)
12 (readline f)
13 (close)
14 ; ... format rename remove get-char read-number set-locale
15 (exit)
```

clips -f 12.4.clp

```
1  CLIPS> (printout t ":)" crlf)  ; (printout <nomlogic> <expressio>*)
2  :)
3  CLIPS> (open "xy" f "w")       ; (open <nomf> <nomlogic> [<mode>])
4  TRUE
5  CLIPS> (printout f "x y" crlf)
6  CLIPS> (close f)               ; (close [<nomlogic>])
7  TRUE
8  CLIPS> (system "cat xy")
9  x y
10 CLIPS> (open "xy" f)           ; mode per omissio: "r"
11 TRUE
12 CLIPS> (read f)
13 x
14 CLIPS> (read f)
15 y
16 CLIPS> (read f)
17 EOF
18 CLIPS> (close f)
19 TRUE
20 CLIPS> (open "xy" f)
21 TRUE
22 CLIPS> (readline f)
23 "x y"
24 CLIPS> (close)
25 TRUE
26 CLIPS> ; ... format rename remove get-char read-number set-locale
```

12.5 Funcions matemàtiques

12.5.clp

```
(+ 2 3 4)                    ; suma
(- 12 3 4)                   ; resta
(* 2 3 4)                    ; multiplicacio
(/ 24 3 4)                   ; divisio
(div 5 2)                    ; divisio sencera
(max 3.0 4 2.0)              ; maxim numeric
(min 4 0.1 -2.3)             ; minim numeric
(abs -2)                     ; valor absolut
(float -2)                   ; conversio a real
(integer 4.0)                ; conversio a sencer
(cos 0)                      ; cosinus (cosh sin sinh tan  ...)
(acos 1.0)                   ; arccosinus (acosh asin asinh ...)
(deg-grad 90)                ; graus: deg-rad grad-deg rad-deg pi
(sqrt 9)                     ; arrel quadrada
(** 3 2)                     ; exponenciacio
(exp 1)                      ; exponenciacio natural
(log 2.71828182845905)       ; logarisme natural, log10 decimal
(round 3.6)                  ; arrodoniment al sencer mes proxim
(mod 5 2)                    ; residu
(exit)
```

```
CLIPS> (+ 2 3 4)                ; suma
9
CLIPS> (- 12 3 4)               ; resta
5
CLIPS> (* 2 3 4)                ; multiplicacio
24
CLIPS> (/ 24 3 4)               ; divisio
2.0
CLIPS> (div 5 2)                ; divisio sencera
2
CLIPS> (max 3.0 4 2.0)          ; maxim numeric
4
CLIPS> (min 4 0.1 -2.3)         ; minim numeric
-2.3
CLIPS> (abs -2)                 ; valor absolut
2
CLIPS> (float -2)               ; conversio a real
-2.0
CLIPS> (integer 4.0)            ; conversio a sencer
4
CLIPS> (cos 0)                  ; cosinus (cosh sin sinh tan  ...)
1.0
CLIPS> (acos 1.0)               ; arccosinus (acosh asin asinh ...)
0.0
CLIPS> (deg-grad 90)            ; graus: deg-rad grad-deg rad-deg pi
100.0
CLIPS> (sqrt 9)                 ; arrel quadrada
3.0
CLIPS> (** 3 2)                 ; exponenciacio
9.0
CLIPS> (exp 1)                  ; exponenciacio natural
2.71828182845905
CLIPS> (log 2.71828182845905)   ; logarisme natural, log10 decimal
1.0
CLIPS> (round 3.6)              ; arrodoniment al sencer mes proxim
4
CLIPS> (mod 5 2)                ; residu
1
```

12.6 Funcions procedimentals

- `bind` assigna valors a variables:

12.6.bind.clp

```
1  (defglobal ?*x* = 3.4)   ; def vble global i li dona valor
2  ?*x*
3  (bind ?*x* (+ 8 9))      ; modif valor vble global
4  ?*x*
5  (bind ?a 3)              ; crea vble local i li dona valor
6  ?a                       ; cal CLIPS v6.30+
7  (deffunction f() (bind ?a 3) (bind ?a (- ?a 1)) ?a)
8  (f)
9  (exit)
```

clips -f 12.6.bind.clp

```
1  CLIPS> (defglobal ?*x* = 3.4)   ; def vble global i li dona valor
2  CLIPS> ?*x*
3  3.4
4  CLIPS> (bind ?*x* (+ 8 9))      ; modif valor vble global
5  17
6  CLIPS> ?*x*
7  17
8  CLIPS> (bind ?a 3)              ; crea vble local i li dona valor
9  3
10 CLIPS> ?a                       ; cal CLIPS v6.30+
11 3
12 CLIPS> (deffunction f() (bind ?a 3) (bind ?a (- ?a 1)) ?a)
13 CLIPS> (f)
14 2
15 CLIPS> (exit)
```

- (if <exp> then <action>* [else <action>*]):

12.6.ifthenelse.clp

```
1  (defglobal ?*prof* = 60)
2  (deffunction inici () ; xa sistemes CLIPS interactius
3    (reset)
4    (printout t "Profunditat maxima: ")
5    (bind ?*prof* (read))
6    (printout t "Amplaria (1) o profunditat (2): ")
7    (bind ?a (read))
8    (if (= ?a 1)
9      then (set-strategy breadth)
10     else (set-strategy depth)))
```

clips interactiu

```
1  CLIPS> (load "12.6.ifthenelse.clp")
2  Defining defglobal: prof
3  Defining deffunction: inici
4  TRUE
5  CLIPS> (inici)
6  Profunditat maxima: 50
7  Amplaria (1) o profunditat (2): 1
8  depth
9  CLIPS> (exit)
```

- (while <expression> [do] <action>*):

12.6.while.clp

```
1 (deffunction bucle (?n) ; FALSE si no acaba amb return
2   (bind ?i 0)
3   (while (< ?i ?n) (printout t ?i crlf) (bind ?i (+ ?i 1))))
4 (bucle 4)
5 (exit)
```

clips -f 12.6.while.clp

```
1 CLIPS> (deffunction bucle (?n) ; FALSE si no acaba amb return
2   (bind ?i 0)
3   (while (< ?i ?n) (printout t ?i crlf) (bind ?i (+ ?i 1))))
4 CLIPS> (bucle 4)
5 0
6 1
7 2
8 3
9 FALSE
```

- `(loop-for-count <range-spec> [do] <action>*)`

  `<range-spec> ::= (<loop-var> <start> <end>)`:

12.6.loop-for-count.clp

```
1 (loop-for-count (?i 0 3) (printout t ?i crlf))
2 (exit)
```

clips -f 12.6.loop-for-count.clp

```
1 CLIPS> (loop-for-count (?i 0 3) (printout t ?i crlf))
2 0
3 1
4 2
5 3
6 FALSE
7 CLIPS> (exit)
```

- `(return [<expression>])`: fi d'execució d'una funció

12.6.return.clp

```
1 (deffunction signe (?n)
2   (if (> ?n 0)
3   then (return 1)
4   else (if (< ?n 0) then (return -1))))
5 (signe 2)
6 (signe -2)
7 (exit)
```

clips -f 12.6.return.clp

```
1 CLIPS> (deffunction signe (?n)
2   (if (> ?n 0)
3   then (return 1)
4   else (if (< ?n 0) then (return -1))))
5 CLIPS> (signe 2)
6 1
7 CLIPS> (signe -2)
8 -1
```

▸ *Altres:*

- ▷ `(progn <exp>*)` avalua els args i torna el valor de l'últim
- ▷ `(progn$ <mcamp-esp> <exp>*)` aplica accions a cada camp
- ▷ `(break)` trenca l'execució d'un bucle while, loop...
- ▷ `(switch...)` execució per casos segons valor d'exp
- ▷ `(foreach...)` execució d'accions per cada camp d'un mcamp

12.7 Funcions vàries

- `(random [<startint> <endint>])` i `(seed <int>)`:

12.7.random.clp

```
1 (seed 23)
2 (random 1 6) ; tira dau
3 (random 1 6)
4 (exit)
```

clips -f 12.7.random.clp

```
1 CLIPS> (seed 23)
2 CLIPS> (random 1 6) ; tira dau
3 3
4 CLIPS> (random 1 6)
5 1
```

- (length <cadena-o-mcamp>):
  - length$ fa el mateix

12.7.length.clp

```
1 (length (create$ a b c d e))
2 (length "gat")
3 (exit)
```

clips -f 12.7.length.clp

```
1 CLIPS> (length (create$ a b c d e))
2 5
3 CLIPS> (length "gat")
4 3
```

- `(sort <fcomp> <exp>*)`
  - `fcomp(?x ?y)` `TRUE` si ordenats

12.7.sort.clp

```
1 (sort > 4 3 5 7 2 7)
2 (deffunction strcmp (?a ?b) (> (str-compare ?a ?b) 0))
3 (sort strcmp Laia Pere Manel Pau)
4 (exit)
```

clips -f 12.7.sort.clp

```
1 CLIPS> (sort > 4 3 5 7 2 7)
2 (2 3 4 5 7 7)
3 CLIPS> (deffunction strcmp (?a ?b) (>= (str-compare ?a ?b) 0))
4 CLIPS> (sort strcmp Laia Pere Manel Pau)
5 (Laia Manel Pau Pere)
```

12.9 Funcions per a fets

- `(assert <RHS>+)`: insereix fet(s); torna adreça (FALSE si està)

12.9.assert.clp

```
1 (assert (color roig))
2 (assert (color verd) (valor (+ 3 4)))
3 (assert (color roig))
4 (exit)
```

clips -f 12.9.assert.clp

```
1 CLIPS> (assert (color roig))
2 <Fact-0>
3 CLIPS> (assert (color verd) (valor (+ 3 4)))
4 <Fact-2>
5 CLIPS> (assert (color roig))
6 FALSE
```

- `(retract <fet>|<int>|*)`: esborra fet(s)

2.1.hola.clp

```
1  (deffacts bf (pendent Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendent ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendent $?y)))
8  (defrule acaba (pendent) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

13 Ordres

13.1 Ordres d'entorn

- `(load[*] <fitxer>)`: carrega constructors (* silenciosa)
- `(save <fitxer>)`: grava constructors (deffacts i defrules)
- `(clear)`: elimina constructors i dades associades (agenda)
- `(exit <int>)`
- `(reset)`: reinicia CLIPS (amb deffacts i defrules)
- `(batch[*] <fitxer>)`: carrega constructors (* silenciosa)
- Altres: `bload bsave options system apropos...`

13.2 Ordres de depuració

- `([un]watch all|globals|rules|activations|facts)`

2.1.hola.clp

```
1  (deffacts bf (pendent Manel Nora Laia))
2  (defrule saluda
3    ?f <- (pendent ?x $?y)
4    =>
5    (printout t "Hola " ?x crlf)
6    (retract ?f)
7    (assert (pendent $?y)))
8  (defrule acaba (pendent) => (halt))
9  (watch facts)
10 (watch activations)
11 (reset)
12 (run)
13 (exit)
```

13.4 Ordres per a fets

- `(facts)`: mostra la base de fets (BF)
- `(load-facts <fitxer>)`: insereix els fets del fitxer en la BF
- `(save-facts <fitxer>)`: grava els fets de la BF en fitxer

13.5 Ordres deffacts

- `(ppdeffacts <nom-deffacts>)`: mostra els fets indicats
- `(list-deffacts)`: mostra els noms de tots els `deffacts`
- `(undeffacts <nom-deffacts>)`: esborra els fets indicats

13.6 Ordres defrule

- `(ppdefrule <nom-regla>)`: mostra una regla
- `(list-defrules)`: mostra els noms de totes les regles
- `(undefrule <nom-regla)`: esborra la regla indicada
- `(matches <nom-regla> [verbose|succint|terse])`
- Altres: `set-break remove-break show-breaks...`

13.6.matches.clp

```
1  (deffacts bf (f a b c))
2  (defrule R (f $?x ?y $?z)
3    => (printout t "x=" ?x " y=" ?y " z="?z crlf))
4  (reset)
5  (matches R)
6  (reset)
7  (run)
8  (exit)
```

clips -f 13.6.matches.clp

```
1  CLIPS> (deffacts bf (f a b c))
2  CLIPS> (defrule R (f $?x ?y $?z)
3    => (printout t "x=" ?x " y=" ?y " z="?z crlf))
4  CLIPS> (reset)
5  CLIPS> (matches R)
6  Matches for Pattern 1
7  f-1
8  f-1
9  f-1
10 Activations
11 f-1
12 f-1
13 f-1
14 CLIPS> (reset)
15 CLIPS> (run)
16 x=() y=a z=(b c)
17 x=(a) y=b z=(c)
18 x=(a b) y=c z=()
```

13.7 Ordres d'agenda

- `(agenda)`: mostra totes instàncies de l'agenda
- `(run [<int>])`: executa el nombe de passos indicat
- `(halt)`: acaba l'execució (en la RHS de la regla objectiu)
- `(set-strategy depth|breadth|...)`: resolució conflictes
- `(get-strategy)`: estratègia de resolució de conflictes actual
- `(set-salience-evaluation <val>)`: avaluació de prioritats
  - `when-defined`: quan es defineixen
  - `when-activated`: quan s'activen
  - `every-cycle`: cada cicle
- `(get-salience-evaluation)`: criteri d'avaluació actual
- `(refresh-agenda)`: re-avalua prioritats en l'agenda

13.8 Ordres defglobal

- `(undefglobal nom-defglobal)`: esborra les vbles indicades
- `(show-defglobals)`: mostra els noms de tots els `defglobals`
- `(set-reset-globals <bool>)`: `TRUE` per omissió
- `(get-reset-globals)`: `reset` reinicia globals?

13.9 Ordres deffunction

- `(ppdeffunction <nom-deffunction)`: mostra la funció
- `(list-deffunctions)`: mostra els noms de totes les funcions
- `(undeffunction <nom-deffunction>)`: esborra funció

13.15 Ordres d'anàlisi computacional

- `(set-profile-percent-threshold [0,100])`: 0 d'inici
- `(get-profile-percent-threshold)`
- `(profile-reset)`: reinicia l'anàlisi computacional
- `(profile-info)`: mostra l'anàlisi
- `(profile constructs | user-functions | off)`

13.15.clp

```
1 (progn (profile user-functions) (run) (profile off)
↪   (profile-info) (exit))
```

clips -f 13.15.clp

```
1 CLIPS> (progn (profile user-functions) (run) (profile off)
  ↪   (profile-info) (exit))
2 Profile elapsed time = 3e-06 seconds
3 Function Name  Entries       Time     %  Time+Kids %+Kids
4 -------------  -------  --------- ------ --------- ------
5 run                  1  0.000001 33.33%  0.000001 33.33%
6 profile              1  0.000001 33.33%  0.000001 33.33%
```

Referències

[1] G. Riley. CLIPS: A Tool for Building Expert Systems. URL.

[2] G. Riley. CLIPS: SourceForge Project Page. URL.

[3] C. Culbert et al. CLIPS Reference Manual I: Basic Programming Guide (v6.31). URL.

[4] C. Culbert et al. CLIPS Reference Manual II: Advanced Programming Guide (v6.31). URL.

[5] C. Culbert et al. CLIPS Reference Manual III: Interfaces Guide (v6.31). URL.

[6] J. Giarratano. CLIPS User's Guide (v6.30). URL.