ΑΣΚΗΣΗ 6

Πρωταρχική μέθοδος αναπαράστασης γνώσης στο CLIPS είναι οι κανόνες. Στις δύο προηγούμενες εργαστηριακές ασκήσεις είδαμε κανόνες με τρεις τύπους συνθηκών: Συνθήκες Προτύπου, Συνθήκες Διεύθυνσης Προτύπου και Εκτιμήσιμες Συνθήκες. Σε αυτήν την εργαστηριακή άσκηση εξακολουθούμε να ασχολούμαστε με τους κανόνες γενικά, ενώ παρουσιάζονται επιπλέον και **κανόνες με Λογικούς Συνδέσμους** στις συνθήκες τους.

Κανόνες στο CLIPS

- Δημιουργήστε ένα αρχείο .clp και μέσα σε αυτό ορίστε τους παρακάτω κανόνες:

```
(defrule r0
   ?s<-(start)
   =>
   (assert (step1))
   (retract ?s)
   (printout t "Shmera "))

(defrule r1
   ?s<-(step1)
   =>
   (assert (step2))
   (retract ?s)
   (printout t "einai "))

(defrule r2
   ?s<-(step2)
   =>
   (assert (step3))
   (retract ?s)
   (printout t "mia "))

(defrule r3
   ?s<-(step3)
   =>
   (assert (step4))
   (retract ?s)
   (printout t "poly "))

(defrule r4
   ?s<-(step4)
   =>
   (assert (step5))
   (retract ?s)
   (printout t "kalh mera. "))
```

```
(defrule r5
   ?s<-(step5)
   =>
   (assert (step6))
   (retract ?s)
   (printout t "Xtes "))

(defrule r6
   ?s<-(step6)
   =>
   (assert (step7))
   (retract ?s)
   (printout t "vevaia "))

(defrule r7
   ?s<-(step7)
   =>
   (assert (step8))
   (retract ?s)
   (printout t "h mera "))

(defrule r8
   ?s<-(step8)
   =>
   (assert (step9))
   (retract ?s)
   (printout t "htan "))

(defrule r9
   ?s<-(step9)
   =>
   (retract ?s)
   (printout t "kalyterh." crlf))
```

Παρατηρήστε τους παραπάνω κανόνες για να καταλάβετε τι κάνουν. Φορτώστε το αρχείο σας στο CLIPS, ανοίξτε τα παράθυρα Facts και Agenda, εισάγετε το γεγονός (start) και δώστε την εντολή (run). Ποιο είναι το αποτέλεσμα της εκτέλεσης των κανόνων; Παρατηρήστε τα γεγονότα (start), (step 1),…,(step9). Τα γεγονότα αυτά αποτελούν γεγονότα ελέγχου (**control facts**) γιατί χρησιμοποιούνται σε έναν κανόνα, με μοναδικό σκοπό την ενεργοποίηση και εκτέλεση ενός άλλου κανόνα.

- Εκτελέστε τις παρακάτω εντολές και παρατηρήστε τι συμβαίνει:

    - (set-break r3)
    - (assert (start))
    - (run)

  Την εντολή **set-break** την είδαμε και στην προηγούμενη εργαστηριακή άσκηση. Θυμηθείτε τι κάνει. Ποιο το περιεχόμενο του παραθύρου Agenda μετά την εκτέλεση; Αν δώσουμε ξανά την εντολή (run) τι θα συμβεί; Επιβεβαιώστε.

- Εκτελέστε τώρα τις παρακάτω εντολές και παρατηρήστε τι συμβαίνει στα παράθυρα Facts και Agenda:

    - (remove-break r3)
    - (show-breaks)
    - (set-break r2)
    - (set-break r7)
    - (set-break r9)
    - (rules)
    - (show-breaks)
    - (assert (start))

  Τι κάνει η εντολή **remove-break** και τι η εντολή **show-breaks**; Δώστε διαδοχικά όσες φορές χρειαστεί την εντολή (run) μέχρι την στιγμή που δεν υπάρχουν πλέον ενεργοποιημένοι κανόνες για να εκτελεστούν. Παρατηρήστε το αποτέλεσμα των διακοπών που έχουμε εισάγει πριν από τους συγκεκριμένους κανόνες. Οι διακοπές είναι πολύ χρήσιμες όταν θέλουμε να κάνουμε debugging.

- Αφαιρέστε τις διακοπές που τέθηκαν πιο πάνω και προσθέστε μια διακοπή στον κατάλληλο κανόνα, ώστε με την πρώτη εντολή (run) να εκτυπώνεται η πρώτη από τις δύο προτάσεις μέχρι την τελεία και να χρειάζεται να δοθεί δεύτερη εντολή (run) προκειμένου να εκτυπωθούν οι λέξεις της δεύτερης πρότασης. Τρέξτε ξανά τους κανόνες και επιβεβαιώστε ότι συμβαίνει αυτό που ζητείται.

- Όπως έχουμε πει σε προηγούμενη εργαστηριακή άσκηση, στο CLIPS μπορούν να ορίζονται διαφορετικά modules που λειτουργούν ανεξάρτητα μεταξύ τους. Για να δείτε πως λειτουργεί αυτό με τους κανόνες αλλάξτε το αρχείο σας όπως φαίνεται πιο κάτω:

```
(defrule r0
   ?s<-(start)
   =>
   (assert (step1))
   (retract ?s)
   (printout t "Shmera "))

(defrule r1
   ?s<-(step1)
   =>
   (assert (step2))
   (retract ?s)
   (printout t "einai "))

(defrule r2
   ?s<-(step2)
   =>
   (assert (step3))
   (retract ?s)
   (printout t "mia "))

(defrule r3
   ?s<-(step3)
   =>
   (assert (step4))
   (retract ?s)
   (printout t "poly "))

(defrule r4
   ?s<-(step4)
   =>
   (retract ?s)
   (printout t "kalh mera. "))
```

```
(defmodule OTHER)

(defrule r0
   ?s<-(start)
   =>
   (assert (step1))
   (retract ?s)
   (printout t "Xtes "))

(defrule r1
   ?s<-(step1)
   =>
   (assert (step2))
   (retract ?s)
   (printout t "vevaia "))

(defrule r2
   ?s<-(step2)
   =>
   (assert (step3))
   (retract ?s)
   (printout t "h mera "))

(defrule r3
   ?s<-(step3)
   =>
   (assert (step4))
   (retract ?s)
   (printout t "htan "))

(defrule r4
   ?s<-(step4)
   =>
   (retract ?s)
   (printout t "kalyterh. " crlf))
```

Τι κάνει η εντολή (defmodule OTHER); Θυμηθείτε πως όταν ορίζουμε έναν κανόνα που έχει ίδιο όνομα με έναν ήδη υπάρχοντα κανόνα προκαλούμε την αντικατάσταση του παλιού κανόνα από τον καινούργιο. Βάσει αυτών, τι περιμένετε να συμβεί με τον ορισμό της δεύτερης ομάδας κανόνων που έχουν ίδια ονόματα με αυτούς της πρώτης ομάδας αν όλοι οι κανόνες οριστούν στο ίδιο module; Σε ποιο module ανήκουν οι κανόνες της πρώτης ομάδας και σε ποιο εκείνοι της δεύτερης ομάδας; Από που το καταλαβαίνουμε αυτό αφού δεν δηλώνεται ρητά; Κάνετε τα παρακάτω αφού καθαρίσετε το περιβάλλον του CLIPS και φορτώσετε το νέο αρχείο:

- (get-current-module)
- (rules)

- (ppdefrule r0)

- (set-current-module MAIN)
- (rules)
- (ppdefrule r0)

Ο κανόνας r0 που εκτυπώνεται την πρώτη φορά είναι ο ίδιος με τον κανόνα r0 που εκτυπώνεται την δεύτερη φορά; Σε ποιο σημείο διαφέρουν οι κανόνες αυτοί και τι άλλο έχουν κοινό εκτός από το όνομα. Αν εισάγουμε το γεγονός (start) που ικανοποιεί και τους δύο κανόνες θα ενεργοποιηθούν και οι δύο; Ελέγξτε το. Δώστε την εντολή (run). Ποιος από τους δύο κανόνες r0 εκτελέστηκε και γιατί;

- Δώστε την εντολή (retract *). Μεταφερθείτε με την κατάλληλη εντολή στο module OTHER. Εισάγετε το γεγονός (start) και δείτε αν ενεργοποιούνται και οι δύο κανόνες r0 αυτή τη φορά. Στη συνέχεια δώστε ξανά την εντολή (run). Ποιος από τους δύο κανόνες r0 θα έπρεπε να εκτελεστεί και γιατί; Ποιος εκτελέστηκε τελικά; Όπως παρατηρείτε, ενώ αρχικά είμαστε το module OTHER και ενώ έχουμε εισάγει το γεγονός που ικανοποιεί την μοναδική συνθήκη του κανόνα r0 του συγκεκριμένου module, ο κανόνας αυτός ενεργοποιείται αλλά δεν εκτελείται όταν δώσουμε την εντολή (run). Αυτό ισχύει γιατί **εξ' ορισμού, το module που έχει την «άδεια» να εκτελεί κανόνες** (στο CLIPS αυτό εκφράζεται ως το focus module) **είναι το module MAIN**. Όταν η «άδεια» εκτέλεσης κανόνων (**focus**) δίνεται σε ένα module μπλοκάρεται αυτόματα η εκτέλεση κανόνων σε όλα τα υπόλοιπα modules ακόμα κι αν υπάρχουν κανόνες τους που ικανοποιούνται. Δώστε την εντολή (get-current-module) και δείτε σε ποιο module είμαστε μετά την εντολή (run).

- **Για να μπορέσουν να εκτελεστούν οι κανόνες ενός module πρέπει το module αυτό να πάρει την ιδιότητα focus με την εντολή: (focus <module-name>).** Κάνετε τα παρακάτω και παρατηρήστε τι συμβαίνει:

  - (focus OTHER)
  - (get-focus-stack)
  - (facts) *;Το γεγονός start το έχουμε ήδη προσθέσει εδώ*
  - (run)
  - (get-focus-stack)
  - (get-current-module)

Η εντολή **(get-focus-stack)** επιστρέφει την στοίβα η οποία περιέχει σε σειρά τα modules στα οποία έχει δοθεί ως τώρα η ιδιότητα focus. Όταν η εντολή focus καθιστά ένα module τρέχων focus module, τότε το νέο focus module γίνεται pushed στην στοίβα αυτή.

Αφού εκτελεστούν οι κανόνες του τρέχοντος focus module το module γίνεται popped από το focus stack και τρέχων focus module γίνεται το αμέσως επόμενο module της στοίβας,

Η παραπάνω διαδικασία επαναλαμβάνεται έως την στιγμή που δεν υπάρχουν πλέον άλλα modules στην στοίβα, οπότε focus module είναι το module MAIN.

- Συνεχίστε με τις επόμενες ενέργειες:

    - (set-current-module MAIN)
    - (assert (start))
    - (run)

  Παρατηρήστε πως όταν εισάγουμε το γεγονός (start) και δίνουμε την εντολή (run) η ιδιότητα focus είναι αυτή που καθορίζει αν θα εκτελεστεί ο ενεργοποιημένος κανόνας r0 του ενός ή του άλλου module. Συνεχίστε ως εξής:

    - (focus OTHER)
    - (assert (start))
    - (set-current-module MAIN)
    - (assert (start))
    - (run)
    - (run)

  Εξηγείστε τι συμβαίνει στις δύο τελευταίες εντολές. Προσθέστε μια διακοπή στον κανόνα r2 του module MAIN και μια διακοπή στον κανόνα r3 του module OTHER. Στην συνέχεια κάνετε τα εξής και θυμηθείτε πώς κάποιες εντολές μπορούν να χρησιμοποιηθούν και εξειδικευμένα για κάποιο module. Όταν δεν καθορίζεται κάποιο module σαν όρισμα αυτών των εντολών, εννοείται το module MAIN.

    - (set-current-module MAIN)
    - (ppdefrule r0)
    - (ppdefrule OTHER::r0)
    - (show-breaks)
    - (show-breaks OTHER)
    - (assert (start))

    - (set-current-module OTHER)
    - (ppdefrule r0)
    - (ppdefrule MAIN::r0)
    - (show-breaks)
    - (show-breaks MAIN)
    - (assert (start))
    - (focus OTHER)
    - (run)
    - (run)
    - (run)
    - (run)

  Παρατηρήστε τον τρόπο με τον οποίο αναφερόμαστε μέσα σε κάποιο module σε έναν κανόνα ενός άλλου module. Με βάση αυτό αλλάξτε κατάλληλα τους ορισμούς των κανόνων του αρχείου σας αν θεωρήσουμε πως η εντολή (defmodule OTHER) δεν βρίσκεται εκεί που την βάλαμε αρχικά αλλά στην αρχή σαν πρώτη εντολή του αρχείου. Αφού φορτώσετε τις αλλαγές ελέγξτε αν όλα είναι όπως πρέπει, εκτυπώνοντας τους κανόνες κάθε module για να επιβεβαιώσετε πως βρίσκονται στη σωστή θέση.

- Οι κανόνες στο CLIPS μπορούν, εκτός από συνθήκες προτύπου, διεύθυνσης προτύπου και εκτιμήσιμες συνθήκες, να έχουν και λογικά συνδεόμενες συνθήκες, δηλαδή συνθήκες που συνδέονται μεταξύ τους με ένα από τα τρία λογικά συνδετικά **and**, **or** και **not**. Αρχικά θα δούμε το στοιχείο or που επιτρέπει σε περισσότερα από ένα διαφορετικά γεγονότα να ενεργοποιήσουν έναν κανόνα. Έστω πως έχουμε τους επόμενους τρεις κανόνες:

```
(defrule system-fault1
  ?x<-(error-status unknown)
  (pump off)
   =>
  (printout t "The system has a fault." crlf)
  (assert (error-status confirmed))
  (retract ?x))

(defrule system-fault2
  ?x<-(error-status unknown)
  (valve broken)
  =>
  (printout t "The system has a fault." crlf)
  (assert (error-status confirmed))
  (retract ?x))

 (defrule system-fault3
   ?x<-(error-status unknown)
   (temp high)
   =>
   (printout t "The system has a fault." crlf)
   (assert (error-status confirmed))
   (retract ?x))
```

Ποια είναι τα μόνα σημεία στα οποία διαφέρουν οι κανόνες αυτοί; Χρησιμοποιήστε το στοιχείο or προκειμένου να αντικαταστήσετε τους τρεις αυτούς κανόνες με έναν μόνο κανόνα system-fault που θα κάνει το ίδιο πράγμα με αυτούς. Θυμηθείτε πως μια συνθήκη or έχει την εξής μορφή: **(or <fact-pattern1> <fact-pattern2> … <fact-patternN>)**. Ορίστε τον κανόνα σε ένα αρχείο .clp, φορτώστε το στο CLIPS , εισάγετε το (error-status unknown) και ένα από τα γεγονότα: (temp high), (valve broken) ή (pump off) και εκτελέστε τον κανόνα.

- Ορίστε και τον επόμενο κανόνα στο αρχείο σας, δώστε την εντολή (clear), και φορτώστε ξανά το αρχείο στο CLIPS:

```
(defrule system-flow
  (error-status confirmed)
  (or (and (temp high)
           (valve closed))
      (and (temp low)
           (valve open)))
   =>
  (printout t "The system is having a flow problem." crlf))
```

Στην συνέχεια κάνετε τα παρακάτω και δείτε πως λειτουργεί το στοιχείο and σε συνδυασμό με το στοιχείο or:

- (retract *))
- (assert (error-status confirmed))

- (assert (temp high))
- (assert (valve open))
- (agenda)
- (retract *)
- (assert (error-status confirmed))

- (assert (temp low))
- (assert (valve closed))
- (agenda)
- (retract *)
- (assert (error-status confirmed))

- (assert (temp high))
- (assert (valve closed))
- (agenda)
- (run)
- (retract *)
- (assert (error-status confirmed))

- (assert (temp low))
- (assert (valve open))
- (agenda)
- (run)
- (retract *)

➢ Ορίστε μέσα σε ένα αρχείο CLIPS τους επόμενους κανόνες για να δείτε πως λειτουργεί το στοιχείο not:

```
(defrule high-flow-rate
  (temp high)
  (valve open)
  (not (error-status confirmed))
   =>
  (printout t "Recommend closing of valve due to high temp"  crlf))

(defrule check-valve
  (check-status ?valve)
  (not (valve-broken ?valve))
  =>
  (printout t "Device " ?valve " is OK" crlf))

(defrule double-pattern
  (data red)
  (not (data red ?x ?x))
  =>
  (printout t "No patterns with red " ?x ?x "!" crlf ))
```

Φορτώστε το αρχείο στο CLIPS, αφού πρώτα δώσετε την εντολή (clear). Σας βγάζει κάποιο μήνυμα το CLIPS; Μπορείτε να εξηγήσετε γιατί συμβαίνει αυτό; Αλλάξτε την εντολή εκτύπωσης του κανόνα double-pattern ώστε να εκτυπώνει τα εξής: (printout t "No patterns with red and 2 same colours! " crlf), φορτώστε ξανά το αρχείο και κάνετε τα παρακάτω:

- (assert (error-status confirmed))
- (assert (temp high))
- (assert (valve open))
- (agenda)
- (run)

Ποιος κανόνας ενεργοποιείται; Γιατί δεν ενεργοποιείται ο high-flow-rate; Κάνετε αυτό που χρειάζεται προκειμένου να ενεργοποιηθεί και εκτελέστε τον. Συνεχίστε με τις επόμενες ενέργειες:

- (retract *)
- (assert (valve-broken valve1))
- (assert (check-status valve1))
- (agenda)
- (assert (check-status valve2))
- (agenda)
- (run)

- (retract *)
- (assert (data red))
- (assert (data red green green))
- (agenda)

Δώστε την εντολή (retract *) και έπειτα εισάγετε ξανά το γεγονός (data red) και εισάγετε το γεγονός (data red green blue). Στην συνέχεια δώστε την εντολή (run).

> Το στοιχείο not δηλώνει ποιο γεγονός δεν πρέπει να υπάρχει ώστε να ικανοποιείται μια συνθήκη σε κάποιον κανόνα. Το not μπορεί να χρησιμοποιηθεί για ένα μόνο γεγονός κάθε φορά. Πρέπει να χρησιμοποιείται με προσοχή όταν συνδυάζεται με τα στοιχεία and και or αλλά και όταν χρησιμοποιούνται δεσμεύσεις μεταβλητών μέσα σε μια συνθήκη not.
>
> Μεταβλητές που έχουν δεσμευτεί σε προηγούμενες συνθήκες χρησιμοποιούνται χωρίς προβλήματα μέσα σε μια συνθήκη not.
>
> Αντίθετα, μεταβλητές που δεσμεύονται πρώτη φορά σε μια συνθήκη not μπορούν να ξαναχρησιμοποιηθούν μόνο μέσα στην ίδια συνθήκη και όχι εκτός αυτής.

- Στην προηγούμενη εργαστηριακή άσκηση είδαμε το στοιχείο συνθήκης (**Conditional Element - CE**) **test**. Αυτό το στοιχείο επιτρέπει να κάνουμε συγκρίσεις μέσα στις συνθήκες ενός κανόνα και δημιουργεί εκτιμήσιμες

συνθήκες. Εκτός από αυτό υπάρχουν δύο ακόμα αντίστοιχα στοιχεία: το **exists** και το **forall**. Το πρώτο ελέγχει την ύπαρξη ενός τουλάχιστον συνδυασμού γεγονότων που ικανοποιεί ένα σύνολο συνθηκών κάποιου κανόνα. Το δεύτερο ελέγχει αν ένα σύνολο συνθηκών κάποιου κανόνα ικανοποιείται από όλους τους αντίστοιχους συνδυασμούς γεγονότων της λίστας γεγονότων. Παρατηρήστε πως λειτουργούν αυτά τα δύο στοιχεία κάνοντας τα εξής: Ορίστε τους επόμενους κανόνες και δομές deffacts σε ένα αρχείο .clp:

```
(defrule rule
   (simple rule)
   (woman ?x)
    =>
   (printout t "There is a woman!" crlf))

(defrule exist
   (existrule)
   (exists (woman ?x))
    =>
   (printout t "There is a woman!" crlf))

(defrule forall
   (forallrule)
   (forall (student ?name)
   (reading ?name)
   (writing ?name)
   (arithmetic ?name))
   =>
   (printout t "All students passed." crlf))

(deffacts women
   (woman Eleni)
   (woman Maria)
   (woman Mirsini)
   (woman Antonia))

(deffacts students
   (student Michael)
   (reading Michael)
   (writing Michael)
   (arithmetic Michael)
   (student Bill)
   (reading Bill)
   (writing Bill)
   (arithmetic Bill)
   (student Helen)
   (reading Helen)
   (writing Helen)
   (arithmetic Helen))
```

Φορτώστε το αρχείο σας, αφού καθαρίσετε το CLIPS. Στην συνέχεια κάνετε τις παρακάτω ενέργειες:

- (reset)
- (assert (simple rule))
- (agenda)
- (run)

- (reset)
- (assert (existrule))
- (agenda)
- (run)

- (reset)
- (assert (forallrule))
- (agenda)
- (run)
- (assert (student Bob))
- (agenda)
- (assert (reading Bob) (writing Bob))
- (agenda)
- (assert (arithmetic Bob))
- (agenda)
- (run)

Εξηγείστε τα αποτελέσματα των παραπάνω ενεργειών. Παρατηρήστε την διαφορά που υπάρχει στην εκτέλεση όταν χρησιμοποιούμε το στοιχείο exists και όταν δεν το χρησιμοποιούμε. Παρατηρήστε επίσης την λειτουργία του στοιχείου forall.

- Κλείστε το CLIPS είτε με την εντολή (exit) στο Dialog Window είτε με File->Exit.