ΠΑΡΑΡΤΗΜΑ 2

Το Σύστημα Κανόνων CLIPS

Το CLIPS (C L*anguage* I*ntegrated* P*roduction* S*ystem*) είναι ένα περιβάλλον που προσφέρει δυνατότητες για προγραμματισμό με κανόνες, αντικείμενα και συναρτήσεις. Αναπτύχθηκε από τη NASA με σκοπό να αποτελέσει μια χαμηλού κόστους πλατφόρμα ανάπτυξης έμπειρων συστημάτων, αντικαθιστώντας τα ήδη υπάρχοντα συστήματα τα οποία βασίζονταν στη γλώσσα LISP.

Η ανάπτυξη του CLIPS άρχισε το 1984 και η πρώτη έκδοση ήταν έτοιμη την άνοιξη του 1985. Για την ανάπτυξή του χρησιμοποιήθηκε η γλώσσα C, έτσι ώστε το τελικό προϊόν να είναι μεταφέρσιμο σε όλα τα γνωστά λειτουργικά συστήματα (DOS, Windows, UNIX, VMS) και να είναι εύκολα επεκτάσιμο. Η πρώτη πλήρης έκδοση του CLIPS ήταν η 3.0, η οποία ήταν έτοιμη το καλοκαίρι του 1986. Την έκδοση αυτή ακολούθησαν οι εκδόσεις 4.0, 4.1 και 4.2, οι οποίες περιείχαν σημαντικές βελτιώσεις στον αρχικό κώδικα, χωρίς να προσθέτουν όμως σημαντικά στοιχεία. Η έκδοση 5.0, που παρουσιάστηκε το 1991, επέκτεινε το αρχικό σύστημα σημαντικά, υποστηρίζοντας εκτός από προγραμματισμό με κανόνες, διαδικαστικό (Procedural Programming) και αντικειμενοστραφή προγραμματισμό (Object Oriented Programming). Η αντικειμενοστραφής γλώσσα προγραμματισμού που παρέχεται με το CLIPS ονομάζεται COOL (CLIPS Object-Oriented Language). Η έκδοση 5.1 υποστήριζε καινούργια διεπαφή χρήστη (user interface) για τα νέα λειτουργικά συστήματα και εμφανίστηκε το 1991. Η τελευταία έκδοση είναι η 6.31 η οποία παρουσιάστηκε το καλοκαίρι του 2019. Η αρχική σχεδίαση του συστήματος έγινε έτσι ώστε να είναι συμβατό με το ART, ένα εμπορικό εργαλείο ανάπτυξης έμπειρων συστημάτων.

Το παράρτημα αυτό αποτελεί μια σύντομη παρουσίαση του προγραμματισμού με κανόνες και συναρτήσεις του CLIPS. Το παρόν κείμενο δε φιλοδοξεί να γίνει πλήρης οδηγός εκμάθησης, αλλά να χρησιμεύσει ως μια μικρή εισαγωγή στο περιβάλλον και τη γλώσσα προγραμματισμού. Πλήρης περιγραφή των δυνατοτήτων του συστήματος περιέχεται στα εγχειρίδια χρήσης. Τα παραδείγματα συστημάτων γνώσης που υπάρχουν στο τέλος του Παραρτήματος βρίσκονται στην ιστοσελίδα του βιβλίου https://aibook.gr.

Π1.1 Δομή του CLIPS

Το CLIPS είναι ένα *διερμηνευόμενο τυπικό σύστημα παραγωγής* (*interpreted production system*), το οποίο υποστηρίζει την *ορθή ακολουθία εκτέλεσης* (*forward chaining*). Τα κύρια μέρη του συστήματος είναι:

- Η *λίστα γεγονότων* (*facts list*), η οποία αντιστοιχεί στη *μνήμη εργασίας* (*working memory*) των συστημάτων παραγωγής. Όπως δηλώνει και το όνομά της, είναι ο χώρος στον οποίο αποθηκεύονται τα *γεγονότα* (*facts*), τόσο εκείνα που ορίζονται κατά την εκκίνηση του συστήματος, όσο και εκείνα που δημιουργούνται κατά την εκτέλεσή του.
- Η *βάση κανόνων* (*rule base / knowledge base*), όπου περιέχονται οι κανόνες. Αν και οι κανόνες μπορούν να ορισθούν μέσα από το περιβάλλον του συστήματος, συνήθως είναι αποθηκευμένοι σε κάποιο αρχείο απλού κειμένου (text document), το οποίο φορτώνεται στο σύστημα.
- Ο *μηχανισμός εξαγωγής συμπερασμάτων* (*inference engine*), ο οποίος ελέγχει τη λειτουργία ολόκληρου του συστήματος. Ο μηχανισμός αυτός προσφέρει ένα πλήθος από *στρατηγικές επίλυσης συγκρούσεων* (*conflict resolution strategies*) για την επιλογή του κανόνα που θα πυροδοτηθεί. Το σύνολο των υποψήφιων κανόνων για πυροδότηση αποτελεί το σύνολο σύγκρουσης ή την *ατζέντα* (*conflict set* ή *agenda*) του συστήματος.

Ένα πρόγραμμα στο CLIPS είναι ένα σύνολο από κανόνες και γεγονότα και η εκτέλεση του συνίσταται σε μια ακολουθία από πυροδοτήσεις κανόνων, των οποίων οι συνθήκες ικανοποιούνται. Η ικανοποίηση των συνθηκών γίνεται μέσω ταυτοποίησής τους με τα γεγονότα που υπάρχουν στη λίστα γεγονότων. Η εκτέλεση τερματίζεται όταν δεν υπάρχουν άλλοι κανόνες προς πυροδότηση ή όταν κληθεί συγκεκριμένη εντολή τερματισμού. Ο κύκλος λειτουργίας του συστήματος είναι ο τυπικός κύκλος λειτουργίας ενός συστήματος παραγωγής:

```
1. Εύρεση όλων των κανόνων των οποίων οι συνθήκες ικανοποιούνται
   και προσθήκη τους στην ατζέντα (agenda - conflict set).
2. Αν η ατζέντα είναι κενή τότε η εκτέλεση τερματίζεται.
3. Επιλογή ενός κανόνα με βάση τη στρατηγική επίλυσης ανταγωνισμού
   (conflict resolution) και εκτέλεσή του.
4. Επιστροφή στο βήμα 1, εκτός αν υπάρχει εντολή τερματισμού (halt).
```

Π1.2 Σύνταξη του CLIPS

Η σύνταξη της γλώσσας που προσφέρει το σύστημα CLIPS είναι απλή και θυμίζει εκείνη της γλώσσας προγραμματισμού LISP. Στη συνέχεια, παρουσιάζεται η σύνταξη αυτή ξεκινώντας από τα βασικά δομικά στοιχεία της και συνεχίζοντας με την παρουσίαση των μεταβλητών, των γεγονότων και των κανόνων. Θα πρέπει να σημειωθεί ότι στο CLIPS υπάρχει διαχωρισμός κεφαλαίων και πεζών χαρακτήρων (case-sensitive).

Π1.2.1 Βασικά Δομικά Στοιχεία

Τα βασικά δομικά στοιχεία της γλώσσας του CLIPS είναι τα ακόλουθα:

Σύμβολα (*symbols*): Σύμβολο μπορεί να είναι οποιαδήποτε ακολουθία χαρακτήρων η οποία ξεκινά με οποιονδήποτε χαρακτήρα εκτός από τους ακόλουθους:

```
< | \& ( ) $ ? + -
```

και δεν περιέχει τους χαρακτήρες:

```
< | \& ( ) ;
```

Για παράδειγμα: **cat-and-dog**, **airway_32**, **memory_relocation_fail**.

Αλφαριθμητικά (*strings*): Ένα αλφαριθμητικό ξεκινά και τελειώνει με διπλά εισαγωγικά, όπως για παράδειγμα: **"This is a nice program"**, **"123 Lotus"**, κτλ.

Αριθμοί (*numbers*): Το CLIPS υποστηρίζει τις ακόλουθες αναπαραστάσεις αριθμών (η σημειολογία θεωρείται γνωστή, καθώς είναι κοινή σε πολλές γλώσσες προγραμματισμού):

```
23, 43, 90, -23, +45, 23.34, 45.78, 4e10, 4E10
```

Σχόλια: Το CLIPS θεωρεί σχόλιο ότι ακολουθεί το χαρακτήρα **";"** μέχρι το τέλος μιας γραμμής. Επίσης, ορισμένες εντολές περιέχουν στη δομή της σύνταξής τους ειδικό όρισμα για εισαγωγή σχολίων, τα οποία συνήθως περικλείονται σε διπλά εισαγωγικά.

Π1.2.2 Μεταβλητές

Οι μεταβλητές στο CLIPS είναι σύμβολα, τα οποία ξεκινούν *απαραίτητα* με τους χαρακτήρες **?** ή **$?** και με τον περιορισμό ο πρώτος χαρακτήρας που ακολουθεί να μην είναι αριθμός. Υπάρχουν δύο είδη μεταβλητών, οι *μονότιμες* και οι *πολλαπλών τιμών*.

- Οι *μονότιμες* (*singlevalue*) μεταβλητές μπορεί να πάρουν σαν τιμή μόνο ένα σύμβολο, αριθμό ή αλφαριθμητικό, όπως άλλωστε δηλώνει και το όνομα τους. Αυτές ξεκινούν με το χαρακτήρα **?**. Για παράδειγμα, το σύμβολο **?var1** είναι μια μονότιμη μεταβλητή με παραδείγματα επιτρεπτών τιμών τα **symbol1**, **flight326**, **Moday_meeting**, **32**, **456**, κτλ.
- Οι μεταβλητές *πολλαπλών τιμών* (*multivalue*) παίρνουν σαν τιμές ένα ή περισσότερα σύμβολα. Οι μεταβλητές αυτές ξεκινούν με τους χαρακτήρες **$?.** Παράδειγμα τέτοιας μεταβλητής είναι η **$?days_of_week** που μπορεί να πάρει σαν τιμή **(mon tue wed thu fri sat sun)** ή **(name Alekos Alexandrou)**, κτλ.

Οι μεταβλητές εμφανίζονται τόσο στις συνθήκες όσο και στις ενέργειες ενός κανόνα, αλλά παίρνουν τιμές κυρίως στις *συνθήκες* των κανόνων μέσω της διαδικασίας ταυτοποίησης. Ανάθεση τιμής σε μεταβλητή στις ενέργειες ενός κανόνα είναι βέβαια δυνατή με τη χρήση κατάλληλης συνάρτησης, αλλά καλό είναι να αποφεύγεται. Τέλος, θα πρέπει να τονισθεί ότι η εμβέλεια των μεταβλητών περιορίζεται στον κανόνα που αυτές εμφανίζονται.

Π1.2.3 Γεγονότα

Τα γεγονότα αποτελούν την πληροφορία την οποία το σύστημα "γνωρίζει" και στην οποία βασίζεται ώστε να εξάγει συμπεράσματα. Τα γεγονότα είναι λίστες από σύμβολα τα οποία περικλείονται σε παρενθέσεις, όπως για παράδειγμα:

```
(person John Smith)
(day monday)
(flight_time_arrival 18:45)
(list of days (mon tue wen thu fri sat) list of months (Jan Feb Mar ))
```

Όπως προαναφέρθηκε τα γεγονότα αποθηκεύονται στη λίστα γεγονότων και είναι δυνατό να εμφανιστούν στο χρήστη με δύο τρόπους:

- Με την εντολή **(facts)** από τη γραμμή εντολών του CLIPS.
- Μέσω του αντίστοιχου παραθύρου (facts window) (το περιβάλλον του CLIPS περιγράφεται στην αντίστοιχη ενότητα στο τέλος του Παραρτήματος).

Τα γεγονότα που υπάρχουν στη μνήμη εργασίας δεν είναι στατικά και μπορούν να εισαχθούν ή να διαγραφούν από αυτή κατά την εκτέλεση του προγράμματος. Τέλος, κάθε γεγονός το οποίο είναι αποθηκευμένο στη λίστα γεγονότων έχει ένα χαρακτηριστικό αριθμό (*fact index*), ο οποίος ορίζεται αυτόματα από το σύστημα κατά την καταχώρηση του γεγονότος στη λίστα. Ο αριθμός αυτός χαρακτηρίζει μοναδικά το γεγονός και χρησιμεύει τόσο στη διαγραφή των γεγονότων όσο και στο να γνωρίζει το σύστημα και ο χρήστης ποια γεγονότα ενεργοποίησαν έναν κανόνα.

Εισαγωγή και διαγραφή γεγονότων

Δύο είναι οι βασικοί τρόποι εισαγωγής γεγονότων στη λίστα γεγονότων: με τη εντολή **assert** και την εντολή **deffacts**. Η διαγραφή ενός γεγονότος από τη λίστα γίνεται με την εντολή **retract**. Παρακάτω δίνεται η σύνταξη των εντολών αυτών.

assert

Σύνταξη: **(assert <fact>)**

Η εντολή **assert** εισάγει ένα γεγονός στη λίστα γεγονότων. Χρησιμοποιείται συνήθως στο δεξιό μέρος των κανόνων.

Παράδειγμα:

Η επόμενη εντολή εισάγει το γεγονός **(water tank empty)** στη λίστα γεγονότων:

```
(assert (water tank empty))
```

deffacts

Σύνταξη:

```
(deffacts <fact-set-name>    ;Όνομα του συνόλου των γεγονότων
   "comment"                 ;Σχόλιο που αφορά το σύνολο των γεγονότων
   (fact 1)                  ;Τα γεγονότα τα οποία
   (fact 2)                  ;θα εισαχθούν στη λίστα γεγονότων.
   ...
   (fact n)
)
```

Η εντολή χρησιμοποιείται για τη μαζική εισαγωγή γεγονότων κατά την εκκίνηση ενός προγράμματος, για παράδειγμα ενός συστήματος γνώσης, στο CLIPS. Για να εισαχθούν τα γεγονότα που ορίζονται με την παραπάνω εντολή στη λίστα γεγονότων θα πρέπει όχι μόνο να φορτωθεί το αντίστοιχο αρχείο αλλά και να εκτελεστεί η εντολή **(reset)** που παρουσιάζεται αργότερα.

Παράδειγμα:

```
(deffacts type-A-extinguisher
   (exctinguisher dry_chemicals type A)
   (exctinguisher water type A)
   (exctinguisher water-based type A)
)
```

retract

Σύνταξη: **(retract <fact-index>)**

Η διαγραφή ενός γεγονότος από τη λίστα γίνεται με την εντολή **retract**. Το όρισμα **fact-index** είναι ο μοναδικός αριθμός που αντιστοιχεί το CLIPS στο συγκεκριμένο γεγονός. Η ίδια εντολή μπορεί να χρησιμοποιηθεί τόσο από τη γραμμή εντολών του CLIPS όσο και στις ενέργειες των κανόνων. Όταν όμως χρησιμοποιείται στις ενέργειες των κανόνων πρέπει να χρησιμοποιηθεί ο ειδικός τελεστής "**<-**" για να ανατεθεί το **fact-index** του συγκεκριμένου γεγονότος σαν τιμή σε μια μεταβλητή.

Παραδείγματα:

1) Αν η λίστα γεγονότων περιέχει τα ακόλουθα:

```
f-1 (day mon)
f-2 (month Jan)
```

όπου **f-1**, **f-2** είναι οι μοναδικοί αριθμοί τους οποίους δίνει το σύστημα αυτόματα στα γεγονότα (fact index), τότε αν δοθεί στη γραμμή εντολών η εντολή:

```
(retract 2)
```

θα αφαιρεθεί το γεγονός **(month Jan)** από τη λίστα.

2) Παρακάτω δίνεται ένα παράδειγμα χρήσης της εντολής μέσα σε έναν κανόνα. Η σύνταξη των κανόνων δίνεται στα επόμενα.

```
(defrule example
   "This is an example rule"
   (day monday)
   (time noon)
   (had lunch)
   ?x <- (should go to coffee shop)   ;Χρήση του ειδικού τελεστή
=>
   (assert (went to coffee shop))
   (retract ?x)                    ;Διαγραφή του γεγονότος από τη λίστα
)
```

Ο ειδικός τελεστής "**<-**" αναθέτει σαν τιμή στη μεταβλητή **?x** το **fact-index** του γεγονότος "**(should go to the coffee shop)**". Η εκτέλεση του παραπάνω κανόνα θα έχει ως αποτέλεσμα τη διαγραφή του συγκεκριμένου γεγονότος.

Π1.2.4 Κανόνες

Οι κανόνες στο CLIPS, όπως και σε όλα τα συστήματα παραγωγής, αποτελούνται από δύο μέρη, τις *συνθήκες* και τις *ενέργειες*, είναι δηλαδή της μορφής:

if (*συνθήκες*) then (*ενέργειες*)

Οι *συνθήκες* είναι τις περισσότερες φορές ένα σύνολο από γεγονότα τα οποία θα πρέπει να υπάρχουν στη λίστα γεγονότων για να ικανοποιείται ο κανόνας. Οι συνθήκες ενός κανόνα μπορεί να περιέχουν μεταβλητές και έτσι να ταυτοποιούνται με περισσότερα του ενός γεγονότα της λίστας γεγονότων. Αυτό επιτρέπει να ικανοποιείται ο κανόνας με περισσότερους από έναν τρόπους, έχοντας κάθε φορά διαφορετικές αναθέσεις τιμών στις μεταβλητές του. Στην περίπτωση αυτή στην ατζέντα εισάγεται ο κανόνας τόσες φορές, όσοι είναι οι διαφορετικοί τρόποι με τους οποίους ικανοποιείται και φυσικά με διαφορετικές αναθέσεις στις αντίστοιχες μεταβλητές.

Στις *ενέργειες* του κανόνα περιγράφεται το τι θα λάβει χώρα κατά την πυροδότησή του (firing). Οι ενέργειες μπορεί να περιλαμβάνουν οποιαδήποτε συνάρτηση του CLIPS, όπως για παράδειγμα την εισαγωγή ή διαγραφή ενός γεγονότος από τη λίστα γεγονότων, την εκτύπωση αποτελεσμάτων, υπολογισμό τιμών, κτλ. Οι κανόνες ορίζονται μέσω της συνάρτησης *defrule* του CLIPS.

defrule

Σύνταξη:

```
(defrule < rule-name >     ;Όνομα Κανόνα (μοναδικό)
   "< comments >"          ;Επεξηγηματικά σχόλια για τον κανόνα
   (condition 1)           ;Συνθήκη 1
   (condition 2)           ;Συνθήκη 2
   ...
   (condition n)           ;Συνθήκη n
=>
   (command 1)          ;Εντολή 1
   (command 2)             ;Εντολή 2
   ...
   (command n)             ;Εντολή n
)
```

Το σύμβολο "**=>**" διαχωρίζει τις συνθήκες από τις ενέργειες του κανόνα. Το όνομα του κανόνα πρέπει να είναι μοναδικό. Αν και είναι δυνατό εντολές της παραπάνω μορφής να εισαχθούν από τη γραμμή εντολών του συστήματος, συνήθως αποθηκεύονται σε κάποιο αρχείο κειμένου (text document) και φορτώνονται στο CLIPS.

Παράδειγμα:

```
(defrule fire-type-rule
   "finding the type of fire"
   (burning ?material)
   (material ?material is of type ?type)
=>
   (assert (fire-type ?type))
)
```

Ο κανόνας του παραδείγματος συμπεραίνει τον τύπο της φωτιάς βάσει του υλικού που καίγεται (**burning ?material**) και του γεγονότος που χαρακτηρίζει το είδος του υλικού (**material ?material is of type ?type**).

Π1.2.5 Ταυτοποίηση

Στα προηγούμενα, αναφέρθηκε η ταυτοποίηση των συνθηκών των κανόνων με τα γεγονότα που βρίσκονται στη λίστα γεγονότων. Η διαδικασία ταυτοποίησης έχει σα στόχο να κάνει "όμοια" τη συγκεκριμένη συνθήκη με το αντίστοιχο γεγονός, μέσω κατάλληλων αναθέσεων τιμών στις μεταβλητές. Η διαφορά της από την ενοποίηση που υπάρχει στο λογικό προγραμματισμό είναι ότι στην ταυτοποίηση η ανάθεση τιμών γίνεται μόνο στο ένα μέρος (συνθήκες κανόνα) ενώ στην ενοποίηση η ανάθεση τιμών γίνεται και στις δύο εκφράσεις. Παρακάτω δίνονται κάποια παραδείγματα (Πίνακας Π2.1 και Πίνακας Π2.2) που αποσαφηνίζουν τη διαδικασία ταυτοποίησης του CLIPS.

Πίνακας Π2.1: Παραδείγματα ταυτοποίησης γεγονότων.

Συνθήκη	Γεγονός με το οποίο ταυτοποιείται	Αναθέσεις τιμών στις μεταβλητές
`(day ?d ?t)`	`(day fri 12)`	`?d=fri ?t=12`
`(list $?lst)`	`(list a b c d e f)`	`$?lst=(a b c d e f)`

`(car ?c model $?m license ?l)`	`(car 1 model BMW FIAT license wqw45)`	`?c=1`
		`$?m=(BMW FIAT)`
		`?l=wqw45`

Πίνακας Π2.2: Παραδείγματα μη ταυτοποίησης γεγονότων.

Συνθήκη	Γεγονός	Γιατί δεν ταυτοποιείται
`(day ?d ?t)`	`(days fri 12)`	Το πρώτο σύμβολο των δύο εκφράσεων είναι διαφορετικό
`(list a b $?lst)`	`(list 1 2 c d e f)`	Το 2ο και 3ο σύμβολο είναι διαφορετικά.
`(car ?c type $?m license ?l)`	`(car 1 2 type BMW FIAT license wqw45)`	Η μονότιμη μεταβλητή `?c` δε μπορεί να πάρει σαν τιμές τα σύμβολα `1` και `2`.

Ιδιαίτερη προσοχή πρέπει να δίνεται όταν στις συνθήκες ενός κανόνα εμφανίζονται μεταβλητές πολλαπλών τιμών. Για παράδειγμα, η συνθήκη `(list $?a ?b $?c)` μπορεί να ταυτοποιηθεί με το γεγονός `(list a b c d)` με τέσσερις διαφορετικούς τρόπους, όπως δείχνει ο Πίνακας Π2.3. Πρέπει να τονισθεί ιδιαίτερα ότι ο κανόνας στον οποίο εμφανίζεται η παραπάνω συνθήκη μπαίνει στην agenda τέσσερις φορές, με διαφορετικές κάθε φορά τιμές στις αντίστοιχες μεταβλητές.

Πίνακας Π2.3: Παράδειγμα ταυτοποίησης μεταβλητών πολλαπλών τιμών.

$?a	?b	$?c
()	a	(b c d)
(a)	b	(c d)
(a b)	c	(d)
(a b c)	d	()

Π1.3 Βασικές Εντολές Περιβάλλοντος

Στα επόμενα δίνονται μερικές βασικές εντολές του περιβάλλοντος του CLIPS, οι οποίες είναι απαραίτητες για την εκτέλεση ενός προγράμματος.

load

Σύνταξη: `(load "<File_Name>")`
Η εντολή αυτή φορτώνει στο σύστημα ένα αρχείο με ορισμούς κανόνων, γεγονότων και συναρτήσεων. Στο παραθυρικό περιβάλλον η ίδια λειτουργία γίνεται μέσω του μενού *File→LoadConstructs*.
Παράδειγμα:

```
(load "rules_example.clp")
```

Η παραπάνω εντολή φορτώνει τις εντολές που είναι αποθηκευμένες στο αρχείο **`rules_example.clp`**. Συνήθως οι εντολές που περιέχονται σε αρχεία είναι εντολές ορισμού κανόνων, εισαγωγής γεγονότων και ορισμού συναρτήσεων.

reset

Σύνταξη: `(reset)`
Η εντολή αυτή διαγράφει από τη λίστα γεγονότων τα γεγονότα που έχουν πιθανόν δημιουργηθεί κατά την εκτέλεση του προγράμματος και επαναφέρει σε αυτή τα αρχικά γεγονότα που ορίζονται στο αρχείο που έχουμε φορτώσει στο σύστημα. Επίσης εισάγει στη λίστα γεγονότων και το γεγονός `(initial-fact)`. Θα πρέπει να σημειωθεί ότι ακόμη και αν φορτωθούν γεγονότα από αρχείο, αυτά δεν εισάγονται αυτόματα στη λίστα γεγονότων εάν δεν προηγηθεί η εντολή `(reset)`.

run

Σύνταξη: `(run)`
Εκκινεί την εκτέλεση των κανόνων που έχουν φορτωθεί στη μνήμη. Η εντολή μπορεί να συνταχθεί και σαν `(run N)`, όπου N είναι ο αριθμός των κύκλων λειτουργίας που θα εκτελεστούν. Μετά την εκτέλεση N κύκλων λειτουργίας το σύστημα θα σταματήσει. Η τελευταία αυτή δυνατότητα χρησιμοποιείται κυρίως κατά τη διαδικασία αποσφαλμάτωσης. Η εκτέλεση μπορεί να συνεχιστεί από το σημείο όπου σταμάτησε με μια νέα εντολή `(run)` ή `(run N)`.

clear

Σύνταξη: `(clear)`
Η εντολή αυτή "καθαρίζει" τελείως το περιβάλλον της γλώσσας, διαγράφοντας όλα τα γεγονότα από τη λίστα γεγονότων και όλους τους κανόνες που τυχόν έχουν φορτωθεί στο σύστημα.

Π1.4 Συναρτήσεις

Οι συναρτήσεις στο CLIPS έχουν παρόμοια μορφή με εκείνη των συναρτήσεων στην LISP. Η συνάρτηση είναι συντακτικά μια δομή που περικλείεται μέσα σε παρενθέσεις, όπου το πρώτο στοιχείο της παρένθεσης είναι το όνομα της συνάρτησης και τα επόμενα στοιχεία τα ορίσματα αυτής. Το διαχωριστικό σύμβολο μεταξύ των ορισμάτων είναι το κενό. Η κλήση μιας συνάρτησης γίνεται με τη δήλωση:

(<όνομα συνάρτησης> *όρισμα1 όρισμα2 ... όρισμαN*)

Για παράδειγμα, με την έκφραση `(+ 2 3)` θα κληθεί η συνάρτηση `+` με ορίσματα `2` και `3`. Το αποτέλεσμα της συνάρτησης θα επιστραφεί στο σημείο που αυτή εμφανίζεται. Για παράδειγμα, αν μέσα σε ένα πρόγραμμα CLIPS υπάρχει η εντολή `(assert (The number is (+ 2 3)))` τότε το γεγονός το οποίο θα αποθηκευτεί στη μνήμη θα είναι το `(The number is 5)`. Στα επόμενα δίνονται βασικές συναρτήσεις της γλώσσας, όπως οι αριθμητικές συναρτήσεις, οι συναρτήσεις ελέγχου και οι συναρτήσεις για τη σύγκριση αριθμών.

Π1.4.1 Βασικές Αριθμητικές Συναρτήσεις

Το CLIPS υποστηρίζει όλες τις βασικές αριθμητικές συναρτήσεις (Πίνακας Π2.4). Θα πρέπει να σημειωθεί ότι οι συναρτήσεις δέχονται περισσότερα από δύο ορίσματα.

Πίνακας Π2.4: Βασικές αριθμητικές συναρτήσεις.

Συνάρτηση	Σύνταξη
Πρόσθεση αριθμών	`(+ <ορίσματα>)`
Αφαίρεση αριθμών	`(- <ορίσματα>)`
Πολλαπλασιασμός αριθμών	`(* <ορίσματα>)`
Διαίρεση αριθμών	`(/ <ορίσματα>)`

Παραδείγματα:

```
(+ 2 3 10)          (- 5 2)          (* 2 3 10)
15                  3                60
```

Π1.4.2 Σύγκριση Αριθμών

Οι συναρτήσεις σύγκρισης αριθμών στο CLIPS ακολουθούν και αυτές τη σύνταξη των αντίστοιχων εντολών στη LISP. Και στην περίπτωση αυτή οι συναρτήσεις δέχονται περισσότερα από δύο ορίσματα. Ο Πίνακας Π2.5 παραθέτει τις διαθέσιμες συναρτήσεις της κατηγορίας.

Πίνακας Π2.5: Συναρτήσεις σύγκρισης αριθμών.

Συνάρτηση	Σύνταξη	Επιστρέφει TRUE...
Αριθμητική ισότητα	`(= <αριθμητικά ορίσματα>)`	...αν όλα τα ορίσματα είναι ίσα.
Μεγαλύτερο	`(> <αριθμητικά ορίσματα>)`	...αν τα ορίσματα είναι κατά φθίνουσα σειρά.
Μικρότερο	`(< <αριθμητικά ορίσματα>)`	...αν τα ορίσματα είναι κατά αύξουσα σειρά.

Μεγαλύτερο ή ίσο	(>= <αριθμητικά ορίσματα>)	...αν τα ορίσματα είναι κατά φθίνουσα σειρά (όχι απόλυτη).
Μικρότερο ή ίσο	(<= <αριθμητικά ορίσματα>)	...αν τα ορίσματα είναι κατά αύξουσα σειρά (όχι απόλυτη).
Διάφορο	(<> <αριθμητικά ορίσματα>)	...αν τα ορίσματα δεν είναι ίσα.

Παράδειγμα:
Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE

```
(>= 5 5 4 2 2 1)    (<=  2 3 3 4 6)     (<> 3 5)
```

Π1.4.3 Λογικές Συναρτήσεις

Το CLIPS δίνει τη δυνατότητα να εκφραστούν πολύπλοκες συνθήκες κανόνων με τη χρήση των κλασικών λογικών συναρτήσεων (Πίνακας Π2.6). Οι λογικές συναρτήσεις επιτρέπουν το συνδυασμό γεγονότων και κατά συνέπεια την πιο συμπαγή διατύπωση κανόνων.

Πίνακας Π2.6: Λογικές συναρτήσεις.

Συνάρτηση	Σύνταξη	Επιστρέφει TRUE...
Λογικό ΚΑΙ	(and <ορίσματα>)	...αν όλα τα ορίσματα της είναι TRUE.
Λογικό Ή	(or <ορίσματα>)	...αν ένα τουλάχιστον όρισμα είναι TRUE.
Λογική άρνηση	(not <όρισμα>)	...αν το όρισμα είναι FALSE.
Ισότητα	(eq <ορίσματα>)	...αν τα ορίσματα είναι ίσα κατά τύπο και τιμή. Τα ορίσματα είναι οποιεσδήποτε συμβολοσειρές.
Ανισότητα	(neq <ορίσματα>)	...αν τα ορίσματα δεν είναι ίσα κατά τύπο και τιμή. Τα ορίσματα είναι οποιεσδήποτε συμβολοσειρές.

Παραδείγματα:
Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE:

```
(and (>= 5 5) (> 3 2 1) (= 10 10))
(and (not = 10 7) (= 9 9))
```

Στο παράδειγμα κανόνα που ακολουθεί, φαίνεται η χρήση της λογικής συνάρτησης **or** στις συνθήκες. Χωρίς τη χρήση της συνάρτησης θα έπρεπε να γραφούν δύο κανόνες, ένας για κάθε περίπτωση.

```
(defrule days
   (month Jan mon)
   (or (hour 12) (hour 13))
=>
   (assert (day is Monday noon time))
)
```

Ένα περισσότερο πολύπλοκο παράδειγμα φαίνεται παρακάτω:

```
(defrule days
   (month Jan mon)
   (or (and (hour 12) (lunch break)) (and (hour 17) (departure time)) )
=>
   (assert (office closed at this hour))
)
```

Η σύνθετη συνθήκη διαβάζεται: "(*η ώρα είναι 12 ΚΑΙ υπάρχει διάλειμμα για γεύμα*) Ή (*η ώρα είναι 17 ΚΑΙ είναι ώρα αναχώρησης*)".

Π1.4.4 Έλεγχος Τύπου

Σε πολλές περιπτώσεις είναι δυνατό να πρέπει να εξακριβωθεί ο τύπος της τιμής μιας μεταβλητής, ώστε να γίνει η κατάλληλη επεξεργασία. Ο Πίνακας Π2.7 παρουσιάζει τις διαθέσιμες συναρτήσεις για έλεγχο τύπων.

Πίνακας Π2.7: Συναρτήσεις ελέγχου τύπων

Συναρτήσεις ελέγχου	Σύνταξη	Επιστρέφει TRUE...
Αριθμών	(**numberp** <όρισμα>)	...αν το όρισμα είναι αριθμός.
Κινητής Υποδιαστολής	(**floatp** <όρισμα>)	...αν το όρισμα είναι αριθμός κινητής υποδιαστολής.
Ακεραίου	(**integerp** <όρισμα>)	...αν το όρισμα είναι ακέραιος αριθμός.
Αλφαριθμητικό	(**stringp** <όρισμα>)	...αν το όρισμα είναι αλφαριθμητικό.
Σύμβολο	(**symbolp** <όρισμα>)	...αν το όρισμα είναι σύμβολο.

Παράδειγμα:
Οι ακόλουθες συναρτήσεις επιστρέφουν TRUE:

```
(numberp 4)   (symbolp day)
```

Π1.4.5 Χειρισμός Πολλαπλών Τιμών

create$

Σύνταξη: (**create$** <ορίσματα>)
Επιστρέφει μια τιμή που μπορεί να ανατεθεί σε μεταβλητή πολλαπλών τιμών (πολλαπλή τιμή), την οποία δημιουργεί από τα ορίσματα.
Παράδειγμα:
Η εντολή

```
(create$ the day is (grey as it was))
```

επιστρέφει:

```
(the day is grey as it was)
```

explode$

Σύνταξη: (**explode$** <string>)
Επιστρέφει μια πολλαπλή τιμή την οποία δημιουργεί από το αλφαριθμητικό. Η εντολή είναι όμοια με την προηγούμενη, διαφέροντας μόνο στο είδος των ορισμάτων που δέχεται.
Παράδειγμα:
Η εντολή

```
(explode$ "the night was blue")
```

θα επιστρέψει:

```
(the night was blue)
```

implode$

Σύνταξη: `(implode$ <multivalue>)`
Επιστρέφει το αντίστοιχο αλφαριθμητικό από μια πολλαπλή τιμή.
Παράδειγμα:
Η εντολή

```
(implode$ (explode$ "the night was blue"))
```

θα επιστρέψει `"the night was blue"`.

nth$

Σύνταξη: `(nth$ N <multivalue>)`
Επιστρέφει το N-οστό πεδίο μιας πολλαπλής τιμής.
Παράδειγμα:
Η εντολή

```
(nth$ 2 (create$ 3 4 5))
```

θα επιστρέψει την τιμή `4`.

member$

Σύνταξη: `(member$ <symbol> <multivalue>)`
Επιστρέφει τη θέση του πεδίου `<symbol>` μέσα στην τιμή `<multivalue>` εφόσον αυτό υπάρχει. Εάν δεν υπάρχει επιστρέφει FALSE.
Παράδειγμα:
Η εντολή

```
(member$ c (create$ a b c))
```

θα επιστρέψει την τιμή `3`.

first$

Σύνταξη: `(first$ <multivalue>)`
Επιστρέφει το πρώτο στοιχείο μιας πολλαπλής τιμής, αλλά σε μορφή λίστας.
Παράδειγμα:
Η εντολή

```
(first$ (create$ 3 4 5))
```

θα επιστρέψει `(3)`.

rest$

Σύνταξη: `(rest$ <multivalue>)`
Επιστρέφει τα υπόλοιπα στοιχεία εκτός από το πρώτο στοιχείο μιας πολλαπλής τιμής.
Παράδειγμα:
Η εντολή

```
(rest$ (create$ 3 4 5))
```

θα επιστρέψει `(4 5)`.

Π1.4.6 Είσοδος - Έξοδος

printout

Σύνταξη: (**printout <device> <expression>**)
Αποστέλλει την έκφραση **<expression>** στη συγκεκριμένη συσκευή **<device>**. Η συσκευή μπορεί να είναι οποιαδήποτε συσκευή I/O, για παράδειγμα ένα αρχείο ή η οθόνη. Στη περίπτωση που η συσκευή είναι η οθόνη, η εντολή συντάσσεται με τιμή **t** (terminal) στην παράμετρο **<device>**.
Παράδειγμα:

```
(printout t "The day was " ?type crlf)
```

Όταν εκτελεστεί η παραπάνω εντολή και η τιμή της μεταβλητής **?type** είναι για παράδειγμα **sunny**, τότε θα τυπωθεί στην οθόνη το μήνυμα: **The day was sunny**. Το σύμβολο **crlf** δηλώνει ότι μετά την εκτύπωση του μηνύματος στην οθόνη ο δρομέας θα αλλάξει γραμμή.

read

Σύνταξη: (**read**)
Η εντολή εισάγει από την προκαθορισμένη συσκευή εισόδου (standard input) το επόμενο σύμβολο. Συνήθως χρησιμοποιείται σε συνδυασμό με την εντολή **bind** η οποία, όπως παρουσιάζεται στα επόμενα, αναθέτει τιμή σε μια μεταβλητή στις ενέργειες ενός κανόνα.
Παράδειγμα:

```
(defrule get-user-answer
   "get the user's answer"
=>
   (printout t "What is your first name: ")
   (bind ?name (read))
   (assert (user-name ?name))
)
```

Όπως φαίνεται στο παραπάνω παράδειγμα η εντολή **(read)** θα "διαβάσει" από το πληκτρολόγιο ένα αλφαριθμητικό το οποίο θα αναθέσει σαν τιμή στη μεταβλητή **?name**.

Π1.4.7 Ανάθεση Τιμής σε Μεταβλητή

bind

Σύνταξη: (**bind <variable> <value>**)
Η εντολή χρησιμοποιείται για να ανατεθεί η τιμή **<value>** σε μια μεταβλητή **<variable>** στις ενέργειες των κανόνων.
Παράδειγμα:

```
(defrule rule1
   "example rule"
   (oldcost ?cost)
   (newcost ?newcost)
=>
   (bind ?total_cost (+ ?newcost ?oldcost))
```

```
    (assert (cost ?total_cost))
    (printout t "The total cost is " ?total_cost crlf)
)
```

Με τη χρήση της εντολής **bind** στο παραπάνω παράδειγμα, υπολογίζεται το συνολικό κόστος (άθροισμα των μεταβλητών **?oldcost ?newcost**) μόνο μια φορά, το αποτέλεσμα αποθηκεύεται στη μεταβλητή **?total_cost** και χρησιμοποιείται στη συνέχεια σαν όρισμα σε δύο διαφορετικές συναρτήσεις.

Π1.4.8 Έλεγχος Ροής Προγράμματος

while

Σύνταξη:

```
(while (condition) do     ;συνθήκη
   (command 1)          ;εντολή 1
   (command 2)          ;εντολή 2
   ...
   (command n)          ;εντολή n
)
```

Η συνάρτηση έχει την ίδια ακριβώς λειτουργία με τις αντίστοιχες συναρτήσεις των άλλων γλωσσών προγραμματισμού, δηλαδή όσο ικανοποιείται μία συνθήκη, εκτελείται ένα σύνολο συναρτήσεων. Πολλές φορές είναι απαραίτητη η χρήση της εντολής **bind** για να αλλάξει τις τιμές μεταβλητών που συμμετέχουν στη συνθήκη.

Παράδειγμα:

Έστω ότι υπάρχει ένα γεγονός στη λίστα γεγονότων της μορφής **(num 1)**. Ο επόμενος κανόνας θα τυπώσει διαδοχικά όλους τους αριθμούς από το 1 μέχρι το 9, με τη φράση **"the num is:"** σαν πρόθεμα.

```
(defrule test
   (num ?n)
=>
   (while (< ?n 10)
      do
         (printout t "the num is: " ?n crlf)
         (bind ?n (+ 1 ?n))
   )
)
```

if...then...else

Σύνταξη:

```
(  if (condition) ;συνθήκη
   then
      (command 1) ;εντολή 1
      (command 2) ;εντολή 2
      ...
      (command M) ;εντολή M
   else
      (command A) ;εντολή A
```

```
      ...
      (command N) ;εντολή N
)
```

Η κλασική συνάρτηση ελέγχου η οποία είναι κοινή σε όλες σχεδόν τις γλώσσες προγραμματισμού. Εκτελεί υπό συνθήκη κάποια σύνολα εντολών.
Παράδειγμα:
Ο επόμενος κανόνας θα τυπώσει **positive**, **negative** ή **zero**, αν ο αριθμός **?n** είναι αντίστοιχα μεγαλύτερος, μικρότερος ή ίσος με μηδέν.

```
(defrule sign
   (num ?n)
=>
   (if   (> ?n 0)
      then (printout t "positive" crlf))
      else
         (if (< ?n 0)
            then (printout t "negative" crlf)
            else (printout t "zero" crlf)
         )
   )
)
```

Π1.4.9 Ορισμός Συναρτήσεων Χρήστη

Εκτός από τις προκαθορισμένες συναρτήσεις, η γλώσσα δίνει τη δυνατότητα να ορίσει ο χρήστης τις δικές του συναρτήσεις. Ο ορισμός αυτών των συναρτήσεων γίνεται μέσω μιας ειδικής συνάρτησης, της **deffunction**.

deffunction

Σύνταξη:

```
(deffunction <function name>    ;το όνομα της νέας συνάρτησης
      (<variables>)          ;μια λίστα με μεταβλητές που είναι
                           ;ορίσματα της συνάρτησης
   (command 1)               ;η πρώτη εντολή
   (command 2)
   ...
   (command n)             ;η τελευταία εντολή
)
```

Θα πρέπει εδώ να σημειωθεί ότι η συνάρτηση επιστρέφει την τιμή της τελευταίας εντολής που εκτελείται. Η εντολή αυτή μπορεί να είναι μια συνάρτηση, ένα σύμβολο, ή μια μεταβλητή της οποίας η τιμή επιστρέφεται. Οι ορισμοί των συναρτήσεων, όπως και οι ορισμοί γεγονότων και κανόνων, μπορεί να περιέχονται σε ένα αρχείο κειμένου και να φορτώνονται μαζικά στο σύστημα.
Παράδειγμα:
Η ακόλουθη συνάρτηση υπολογίζει τη μέση τιμή τεσσάρων αριθμών:

```
(deffunction mean-value
   (?v1 ?v2 ?v3 ?v4)
   ( / (+ ?v1 ?v2 ?v3 ?v4) 4)
```

```
)
```

και η εκτέλεση της θα δώσει:

```
(mean-value 4 5 6 7)
5.5
```

Π1.5 Περιορισμοί στις Συνθήκες των Κανόνων

Εκτός από απλή ταυτοποίηση γεγονότων οι συνθήκες των κανόνων μπορεί να περιέχουν και περιορισμούς, καταλήγοντας έτσι σε πιο σύνθετες και εκφραστικές μορφές. Οι περιορισμοί εκφράζονται είτε με τη χρήση συγκεκριμένων συνδετικών ή με την εισαγωγή συνθηκών υπό μορφή συναρτήσεων (test conditions) και αποτελούν ένα ακόμη επίπεδο ελέγχου ικανοποίησης των κανόνων. Ο Πίνακας Π2.8 περιέχει τα διαθέσιμα συνδετικά της πρώτης κατηγορίας.

Για παράδειγμα, η συνθήκη **(month Jan day ~mon)** ικανοποιείται εάν υπάρχει ένα ή περισσότερα γεγονότα στη λίστα γεγονότων με τη μορφή **(month Jan day <symbol>)** όπου το τελευταίο σύμβολο της λίστας *δεν* είναι το **mon**. Με παρόμοιο τρόπο χρησιμοποιείται το συνδετικό "|" (or): η συνθήκη **(hour 12 | 13)**, μπορεί να ταυτοποιηθεί είτε με το γεγονός **(hour 12)** ή με το **(hour 13)**. Για παράδειγμα, ο κανόνας:

```
(defrule days
   (month Jan day ~mon)
   (hour 12|13)
      =>
   (printout t " Day is not Monday, but it is noon time!!!" crlf)
)
```

θα ενεργοποιηθεί εάν υπάρχει ένα γεγονός στη λίστα όπου το τελευταίο του πεδίο να μην είναι **mon** και ένα γεγονός **(hour 12)** ή **(hour 13)**.

Πίνακας Π2.8: Συνδετικά συνθηκών κανόνων.

Λογική Πράξη	Συνδετικό
λογική άρνηση	~
λογική διάζευξη (ή)	|
λογική σύζευξη (και)	&

Το συνδετικό "**&**" (and) χρησιμοποιείται συνήθως σε συνδυασμό με άλλους περιορισμούς. Σύμφωνα με τα παραπάνω, ο κανόνας του προηγούμενου παραδείγματος μπορεί να γραφεί με την ακόλουθη μορφή:

```
(defrule days
   (month Jan day ?day&~mon)
   (hour 12|13)
      =>
   (printout t " Day is " ?day ", but it is noon time!!!"   crlf)
)
```

Η συνθήκη **?day&~mon** δηλώνει ουσιαστικά "στη θέση αυτή πρέπει να υπάρχει κάποιο σύμβολο *και* το σύμβολο αυτό να *μην είναι* το *mon*". Η ουσιαστική διαφορά μεταξύ των συνθηκών των παραπάνω κανόνων είναι ότι στη δεύτερη περίπτωση υπάρχει και η συγκεκριμένη τιμή του πεδίου η οποία ενεργοποίησε τον κανόνα σαν τιμή στη μεταβλητή **?day**, όπως φαίνεται στο παράδειγμα.

Έστω ότι υπάρχουν τα παρακάτω γεγονότα στη λίστα γεγονότων του CLIPS:

```
(deffacts elements
   (element a)
   (element b)
   (element c)
)
```

και απαιτείται να τυπωθούν στην οθόνη οι διαφορετικοί συνδυασμοί τους ανά δύο. Ο κανόνας:

```
(defrule cartesian
   (element ?a)
   (element ?b)
      =>
   (printout t "Elements: " ?a " " ?b crlf)
)
```

θα τυπώσει και τους συνδυασμούς **(Elements: a a) (Elements: b b) (Elements: c c)** οι οποίοι δεν είναι επιθυμητοί. Για να αποφευχθεί η παραπάνω συμπεριφορά εισάγονται περιορισμοί στη συνθήκη του κανόνα:

```
(defrule cartesian
   (element ?a)
   (element ?b&~?a)  ;η μεταβλητή ?b παίρνει τιμές ≠ από την ?a
      =>
   (printout t "Elements: " ?a " " ?b crlf)
)
```

Ο συνδυασμός των συνδετικών επιβάλλει στην τιμή της μεταβλητής **?b** να είναι διαφορετική από εκείνη της **?a**.

Εκτός από τη χρήση συνδετικών επιτρέπεται και η χρήση λογικών συναρτήσεων στις συνθήκες των κανόνων. Οι λογικές συναρτήσεις παρουσιάστηκαν στα προηγούμενα και είναι οι **(and ...)**, **(or ....)** και **(not....)**. Προφανώς όλοι οι συνδυασμοί τους είναι επιτρεπτοί. Για παράδειγμα, οι δύο επόμενοι κανόνες:

```
(defrule emerg1
   (emerg type fire)
      =>
   (assert (evacuate building))
)

(defrule emerg2
   (fire drill)
      =>
   (assert (evacuate building))
)
```

μπορεί να γραφούν σαν ένας:

```
(defrule emerg1
   (or (emerg type fire) (fire drill))
      =>
   (assert (evacuate building))
)
```

Θα πρέπει να σημειωθεί ότι, παρόμοια με τη γλώσσα PROLOG, αν εμφανίζονται μεταβλητές μέσα σε κάποιο **(not .....)** τότε δε γίνεται ανάθεση τιμών σε αυτές. Για παράδειγμα, ο επόμενος κανόνας προκαλεί λάθος εκτέλεσης εφόσον η μεταβλητή **?b** δε θα πάρει τιμή στις συνθήκες του κανόνα και κατά συνέπεια η εντολή εκτύπωσης δε θα λειτουργήσει σωστά:

```
(defrule wrong-rule
   (not (element ?b))
      =>
   (printout t " not element" ?b crlf)
)
```

Τέλος, θα πρέπει να αναφερθεί ότι είναι δυνατό να γίνουν και συγκρίσεις στις συνθήκες ενός κανόνα, χρησιμοποιώντας τη συνάρτηση **(test ....)**. Για παράδειγμα, ο κανόνας **cartesian** μπορεί να γραφεί:

```
(defrule cartesian
   (element ?a)
   (element ?b)
   (test (neq ?a ?b));η μεταβλητή ?b παίρνει τιμές ≠ από την ?a
      =>
   (printout t "Elements: " ?a " " ?b crlf)
)
```

όπου ο έλεγχος για τις διαφορετικές τιμές των μεταβλητών γίνεται στην τελευταία γραμμή των συνθηκών με τη χρήση της σχετικής εντολής.

Παράδειγμα

Ένα ενδιαφέρον παράδειγμα που συνδυάζει όλα τα παραπάνω είναι το ακόλουθο. Έστω ότι από μια λίστα βιβλίων που παριστάνεται με γεγονότα της μορφής:

```
(deffacts prices
   (book A price 34)
   (book B price 20)
   (book C price 68)
)
```

απαιτείται να επιλεγεί το πιο ακριβό. Ο κανόνας που θα επιστρέψει το ακριβότερο βιβλίο είναι ο ακόλουθος:

```
(defrule select-exp2
   (book ?Book price ?price)
   (not  (and  (book ?Book2&~?Book price ?price2)
               (test (> ?price2 ?price)) ) )
      =>
   (printout t "The Book with the highest price is:" ?Book crlf)
```

```
)
```

Ο παραπάνω κανόνας μπορεί να ερμηνευτεί ως "*Εάν υπάρχει βιβλίο* **?Book** *με τιμή* **?price** *και δεν υπάρχει άλλο βιβλίο* **?Book2** *διαφορετικό από το* **?Book** *το οποίο να έχει μεγαλύτερη τιμή από την* **?price** *τότε τύπωσε το βιβλίο* **?Book".**

Π1.6 Πρότυπα Γεγονότων

Μέχρι τώρα στο σύνολο των παραδειγμάτων που αναφέρθηκαν δεν υπήρχε η ανάγκη για ορισμό μεγάλων σε μέγεθος γεγονότων. Ωστόσο σε μεγαλύτερα προγράμματα εμφανίζεται σχεδόν πάντα η ανάγκη αναπαράστασης της διαθέσιμης πληροφορίας με τέτοια γεγονότα. Για παράδειγμα, έστω ότι υπάρχει μια βάση δεδομένων μαθητών όπου καταχωρούνται διάφορα στοιχεία τους, κάθε εγγραφή της οποίας αναπαριστάται με ένα γεγονός της μορφής:

```
(student name <name> surname <surname> sex <sex> age <age> classes <classes>)
```

όπως για παράδειγμα:

```
(student name john surname ref sex male age 28 classes math physics chem)
```

Η χρήση του παραπάνω γεγονότος ως συνθήκη σε κάποιο κανόνα, ή ακόμη και οι πιθανές αλλαγές που θα έπρεπε να γίνουν στα στοιχεία της βάσης, απαιτούν να γραφούν κανόνες με αρκετά μεγάλες συνθήκες, καθώς το γεγονός θα έπρεπε να αναφέρεται κάθε φορά με όλες τις παραμέτρους του. Για παράδειγμα, ένας κανόνας που απλώς τυπώνει τα ονόματα των μαθητών σε ένα τέτοιο πρόγραμμα γράφεται:

```
(defrule print-students
   (student name ?n sex ?s age ?a classes $?cl)
      =>
   (printout t "Student: " ?n crlf)
)
```

Το CLIPS προσφέρει έναν εναλλακτικό τρόπο δημιουργίας και διαχείρισης τέτοιων γεγονότων μέσω ορισμού *προτύπων γεγονότων*. Τα πρότυπα γεγονότων είναι μια δομή με την οποία μπορεί να οριστεί η μορφή που θα έχουν τα γεγονότα σε ένα πρόγραμμα. Κάθε πρότυπο έχει ένα σύνολο από *ιδιότητες* (*slots*) στις οποίες μπορεί να ανατεθούν τιμές αυτόνομα, ενώ επιπλέον μπορεί να ορισθούν και οι τύποι των τιμών αυτών ώστε να γίνονται οι απαραίτητοι έλεγχοι. Ο ορισμός πρότυπων γεγονότων γίνεται με τη συνάρτηση **deftemplate**.

deftemplate

Σύνταξη:

```
(deftemplate <template name>
   (slot <slotname1>    (type <type1>))        ;ιδιότητα 1 τύπου1
   (multislot <slotname2> (type <type2>))  ;ιδιότητα 2 τύπου2
   ...
   (slot <slotnameN>    (type <typeN>))        ;ιδιότητα N τύπουN
)
```

όπου **slotnameN** είναι το όνομα της ιδιότητας και το όρισμα **(type <typeN>)** καθορίζει τον τύπο της τιμής της συγκεκριμένης ιδιότητας. Ο καθορισμός τύπου είναι προαιρετικός και αν δεν ορισθεί για κάποια ιδιότητα αυτή μπορεί να δεχθεί τιμή οποιουδήποτε τύπου. Υπάρχουν δύο είδη ιδιοτήτων: οι ιδιότητες **slot** που μπορεί να πάρουν σαν τιμή μόνο ένα σύμβολο ή αριθμό και οι ιδιότητες **multislot** οι οποίες δέχονται πολλαπλές τιμές. Μπορεί να θεωρηθεί ότι οι πρώτες αντιστοιχούν στις μονότιμες μεταβλητές, ενώ οι δεύτερες στις μεταβλητές πολλαπλών τιμών.
Παράδειγμα:
Χρησιμοποιώντας τα παραπάνω, ο ορισμός του προτύπου για τη βάση δεδομένων των μαθητών θα είναι:

```
(deftemplate student
   (slot name)
   (slot surname)
   (slot sex)
   (slot age)
   (multislot classes)
)
```

και ο κανόνας ο οποίος τυπώνει τα ονόματα όλων των αρρένων μαθητών:

```
(defrule print-students
   (student (name ?name) (sex male))
      =>
   (printout t "Student: " ?name crlf)
)
```

Όπως παρατηρείται στον παραπάνω κανόνα, δε χρειάζεται να γραφεί στις συνθήκες ολόκληρο το γεγονός, αλλά μόνο το όνομα του προτύπου και τα ονόματα των ιδιοτήτων που ενδιαφέρουν. Ο Πίνακας Π2.9 περιέχει τους διαθέσιμους τύπους τιμών για τις ιδιότητες των προτύπων.

Πίνακας Π2.9: Τύποι τιμών για τις ιδιότητες των προτύπων.

Τύπος	Η ιδιότητα μπορεί να περιέχει
SYMBOL	σύμβολα
STRING	αλφαριθμητικά
LEXEME	σύμβολα ή αλφαριθμητικά
INTEGER	ακέραιες τιμές
FLOAT	πραγματικές τιμές
NUMBER	ακέραιες ή πραγματικές τιμές
FACT-ADDRESS	διεύθυνση γεγονότος
INSTANCE-ADDRESS	διεύθυνση αντικειμένου της COOL
INSTANCE-NAME	όνομα αντικειμένου της COOL
INSTANCE	όνομα ή διεύθυνση αντικειμένου της COOL
?VARIABLE	τιμές οποιουδήποτε τύπου

Σύμφωνα με τα παραπάνω, ένα ολοκληρωμένο πρότυπο που αφορά τους μαθητές μπορεί να γραφεί:

```
(deftemplate student
```

```
(slot name (type SYMBOL))
(slot surname (type SYMBOL))
(slot sex (type SYMBOL))
(slot age (type INTEGER)
(multislot classes (type SYMBOL))  )
```

Στην περίπτωση που επιχειρηθεί να ανατεθεί μη επιτρεπτή τιμή σε κάποια ιδιότητα, το σύστημα θα επιστρέψει μήνυμα λάθους.

Εκτός από τον επιτρεπτό τύπο της τιμής της ιδιότητας, το CLIPS δίνει τη δυνατότητα να απαριθμηθούν οι επιτρεπτές τιμές που μπορεί να πάρει αυτή. Η απαρίθμηση γίνεται χρησιμοποιώντας τη δήλωση **(allowed-prefix <values>)**, όπου το **prefix** μπορεί να πάρει μία από τις τιμές **symbols**, **strings**, **lexemes**, **integers**, **floats**, **numbers**, **values**, όπως φαίνεται και παρακάτω:

```
(allowed-symbols <symbols>)
(allowed-integers <integers>)
(allowed-numbers <numbers or floats>)
(allowed-values <values>)
...
```

Για παράδειγμα, στο πρότυπο **student** η παρακάτω δήλωση περιορίζει τις τιμές της ιδιότητας **sex** σε **male** ή **female** :

```
(deftemplate student
   ...
   (slot sex (type SYMBOL) (allowed-symbols male female)) ;περιορισμός
   ...
)
```

Ειδικά για τις αριθμητικές τιμές είναι δυνατό να ορισθεί το επιτρεπτό εύρος τιμών που επιδέχεται η ιδιότητα, με μια δήλωση του τύπου **(range <min value> <max value>)**. Για παράδειγμα, μπορεί να οριστεί ότι οι επιτρεπτές ηλικίες (ιδιότητα **age**) των μαθητών της βάσης είναι μεταξύ 18 και 60, με μια δήλωση της μορφής:

```
(deftemplate student
   ....
   (slot age (type INTEGER) (range 18 60))
   ...
)
```

Εάν η ιδιότητα δεν πρέπει να έχει άνω ή κάτω όριο τότε εισάγεται το σύμβολο **?VARIABLE** στην αντίστοιχη θέση. Η παρακάτω δήλωση θέτει μόνο κάτω όριο στο εύρος τιμών της ιδιότητας:

```
(deftemplate student
   ....
   (slot age (type INTEGER) (range 18 ?VARIABLE))
   ...
)
```

Στις ιδιότητες πολλαπλών τιμών (*multislots*) μπορεί επίσης να περιοριστεί το πλήθος των συμβόλων που μπορεί να δοθούν σαν τιμή, με μια δήλωση της μορφής

(cardinality <min> <max>). Για παράδειγμα, αν πρέπει να δηλωθεί ότι τα επιτρεπτά μαθήματα ανά μαθητή είναι το πολύ τέσσερα, ο ορισμός του προτύπου **student** γίνεται:

```
(deftemplate student
   ...
   (multislot classes (type SYMBOL) (cardinality 1 4))
   ...
)
```

Τέλος, είναι δυνατή η δήλωση προκαθορισμένων τιμών (default) σε όλες τις ιδιότητες. Οι τιμές αυτές εισάγονται με τη δήλωση **(default <value>)**. Για παράδειγμα:

```
(deftemplate student
   ...
   (slot age (type INTEGER) (default 18))
   ...
)
```

Μέσα σε μια δήλωση **default** μπορεί να δοθούν δύο σύμβολα με ειδική σημασία:

- **(default ?DERIVE):** Στην περίπτωση αυτή δίνεται σαν τιμή στην ιδιότητα μια από τις επιτρεπόμενες τιμές, όπως αυτές προκύπτουν από τους περιορισμούς που υπάρχουν για την ιδιότητα. Αυτή είναι και η "εξ' ορισμού" δήλωση προκαθορισμένης τιμής.
- **(default ?NONE):** Δηλώνει ότι δεν υπάρχει προκαθορισμένη τιμή για την ιδιότητα, που σημαίνει ότι πρέπει οπωσδήποτε να δοθεί τιμή στη συγκεκριμένη ιδιότητα κατά τη δημιουργία του αντίστοιχου γεγονότος. Σε αντίθετη περίπτωση το γεγονός δε θα εισαχθεί στη λίστα γεγονότων.

Παράδειγμα:

```
(deftemplate student
   ...
   (slot age (type INTEGER) (range 18 ?VARIABLE) (default ?DERIVE))
   ...
)
```

Στο παραπάνω παράδειγμα, αν εισαχθεί γεγονός που να βασίζεται στο πρότυπο **student** χωρίς τιμή για την ιδιότητα **age**, τότε αυτή θα πάρει την τιμή **18**.

```
(deftemplate student
   ...
   (slot name (type SYMBOL) (default ?NONE))
   ...
)
```

Στο παραπάνω παράδειγμα δε θα εισαχθεί γεγονός που να βασίζεται στο πρότυπο **student**, αν δεν υπάρχει τιμή για την ιδιότητα **name**.

Π1.6.1 Έλεγχος Τιμών

Το CLIPS διαθέτει δύο επίπεδα ελέγχου τιμών που εισάγονται στα πρότυπα γεγονότων, το στατικό και το δυναμικό. Ο στατικός έλεγχος τιμών αφορά τα γεγονότα τα οποία εισάγονται στο σύστημα από αρχείο, μέσα από τις δηλώσεις **deffacts**, **defrule**, κτλ. Ενεργοποιείται από την εντολή:

```
(set-static-constraint-checking TRUE|FALSE )
```

Το FALSE απενεργοποιεί το στατικό έλεγχο των τιμών, ενώ το TRUE έχει το αντίθετο αποτέλεσμα. Η τρέχουσα τιμή επιστρέφεται με την εντολή:

```
(get-static-constraint-checking)
```

Στο δυναμικό έλεγχο κάθε γεγονός που εισάγεται στη λίστα γεγονότων ελέγχεται για την εγκυρότητα των τιμών του, ακόμη και αν η εισαγωγή αυτή γίνεται κατά την εκτέλεση του προγράμματος. Η ενεργοποίηση/απενεργοποίηση αυτού του επιπέδου ελέγχου γίνεται με εντολές παρόμοιες με τις προηγούμενες, δηλαδή:

```
(set-dynamic-constraint-checking TRUE|FALSE )
```

ενώ η τρέχουσα τιμή δίνεται με την εντολή:

```
(get-dynamic-constraint-checking)
```

Σε περίπτωση ύπαρξης σφάλματος στις τιμές που δίνονται σε κάποια ιδιότητα το CLIPS επιστρέφει μήνυμα λάθους και σταματά την εκτέλεση των κανόνων.

Π1.6.2 Αλλαγή Τιμής Ιδιότητας

Η αλλαγή της τιμής μιας ιδιότητας προτύπου γίνεται με την εντολή **modify**.

modify

Σύνταξη: (**modify <fact-index> (<slot> <νέα τιμή slot>)**)
Η εντολή μεταβάλλει την τιμή της ιδιότητας **<slot>** σε **<νέα τιμή slot>** στο γεγονός με αριθμό **fact-index**. Θα πρέπει να σημειωθεί ότι η εντολή δίνει τη δυνατότητα αλλαγής της τιμής κάθε ιδιότητας ενός γεγονότος που ακολουθεί κάποιο πρότυπο, ανεξάρτητα από τις άλλες ιδιότητες. Η χρήση του χαρακτηριστικού αριθμού για την εφαρμογή της εντολής επιβάλλει τη χρήση του τελεστή **<-** στις συνθήκες του κανόνα.
Παράδειγμα:
Ο ακόλουθος κανόνας αλλάζει ταυτόχρονα τις τιμές των ιδιοτήτων **name** και **classes** στα γεγονότα **student** που βρίσκονται στη μνήμη εργασίας.

```
(defrule change-information
   ?x <- (student (name ?name))
      =>
   (modify ?x (name noname) (classes (create$ math physics chem)))
)
```

Πρέπει να τονισθεί ότι η εντολή **modify** αφαιρεί το παλαιό γεγονός από τη λίστα και προσθέτει σε αυτή ένα καινούργιο με τις απαραίτητες αλλαγές. Αυτό σημαίνει ότι ο παραπάνω κανόνας θα εκτελείται επ' άπειρον, καθώς θα υπάρχει πάντα σε κάθε κύκλο ένα "νέο" γεγονός (με νέο **fact index**) το οποίο θα ικανοποιεί τις συνθήκες του. Μια ορθότερη εκδοχή του κανόνα θα ήταν:

```
(defrule change-information
   ?x <- (student (name ?name&~noname))
      =>
   (modify ?x (name noname) (classes (create$ math physics chem)))
)
```

Π1.7 Επίλυση Συγκρούσεων Κανόνων

Όπως έχει αναφερθεί στην εισαγωγή, όλοι οι κανόνες των οποίων οι συνθήκες ικανοποιούνται εισάγονται στην *ατζέντα* (*agenda*), η οποία αντιστοιχεί στο *σύνολο συγκρούσεων* (*conflict set*) των κλασικών συστημάτων παραγωγής. Από το σύνολο των κανόνων αυτών επιλέγεται κάθε φορά ένας κανόνας, ο οποίος και πυροδοτείται με βάση δύο κριτήρια: την προτεραιότητα των κανόνων και τη στρατηγική επίλυσης συγκρούσεων. Στην πράξη, η ατζέντα συμπεριφέρεται σα μια στοίβα (stack) όπου όσο μεγαλύτερη προτεραιότητα έχει ένας κανόνας τόσο πιο ψηλά βρίσκεται σε αυτή. Ο κανόνας ο οποίος εκτελείται είναι εκείνος ο οποίος βρίσκεται στην κορυφή της στοίβας. Ένας νέος κανόνας τοποθετείται στην ατζέντα σύμφωνα με τα ακόλουθα κριτήρια:

1. Οι νέοι κανόνες μπαίνουν "πάνω" από όλους τους κανόνες με μικρότερη ή ίση προτεραιότητα (salience) και "κάτω" από όλους τους κανόνες με μεγαλύτερη προτεραιότητα.
2. Στους κανόνες με ίδια προτεραιότητα χρησιμοποιείται η τρέχουσα στρατηγική επίλυσης συγκρούσεων για να καθοριστεί η σειρά τους.
3. Εάν κάποιοι κανόνες ενεργοποιήθηκαν από το ίδιο σύνολο γεγονότων και τα προηγούμενα βήματα δεν μπόρεσαν να ορίσουν μία σειρά, τότε δίνεται σε αυτούς μια αυθαίρετη σειρά (όχι τυχαία), η οποία εξαρτάται από την υλοποίηση του συστήματος.

Στη συνέχεια παρουσιάζεται σύντομα η προτεραιότητα των κανόνων και των διαθέσιμων στρατηγικών επίλυσης συγκρούσεων.

Π1.7.1 Προτεραιότητα Κανόνων

Η προτεραιότητα ενός κανόνα ορίζεται μέσω της δήλωσης

```
(declare (salience <number>))
```

στη συνάρτηση ορισμού του κανόνα. Για παράδειγμα:

```
(defrule cartesian
   (declare (salience 30)) ;ορισμός προτεραιότητας κανόνα
   (element ?a)
   (element ?b)
=>
   (printout t "Elements: " ?a " " ?b crlf)
)
```

Η αριθμητική τιμή καθορίζει την προτεραιότητα του κανόνα. Όσο μεγαλύτερη είναι η τιμή αυτή τόσο μεγαλύτερη είναι και η προτεραιότητα του συγκεκριμένου κανόνα. Οι επιτρεπτές τιμές της προτεραιότητας είναι από -10000 έως 10000. Εάν δεν υπάρχει δήλωση, ο κανόνας θεωρείται ότι έχει την προκαθορισμένη τιμή μηδέν.

Π1.7.2 Στρατηγικές Επίλυσης Συγκρούσεων

Το CLIPS διαθέτει επτά διαφορετικές στρατηγικές επίλυσης συγκρούσεων. Σε κάθε χρονική στιγμή μόνο μια από αυτές είναι ενεργή και χρησιμοποιείται για την επιλογή του κανόνα από την ατζέντα. Η τρέχουσα στρατηγική δίνεται από τη συνάρτηση:

```
(get-strategy)
```

η οποία επιστρέφει το όνομα της στρατηγικής.

Ο ορισμός της επιθυμητής στρατηγικής επίλυσης συγκρούσεων γίνεται με τη χρήση της συνάρτησης:

```
(set-strategy <strategy>)
```

όπου <**strategy**> είναι μια από τις τιμές: **depth**, **breadth**, **simplicity**, **complexity**, **lex**, **mea**, **random**. Η σημασία των τιμών αυτών είναι η εξής:

- **depth**: Σύμφωνα με αυτήν τη στρατηγική, οι "νέοι" κανόνες μπαίνουν "πάνω" από τους "παλαιούς". Η σύγκριση γίνεται με βάση το πότε εισήχθηκαν τα γεγονότα που ικανοποιούν τον κανόνα στη λίστα γεγονότων. Το όνομα της στρατηγικής έχει να κάνει με το γεγονός ότι η εκτέλεση ενός συνόλου κανόνων με αυτήν τη στρατηγική συμπεριφέρεται σαν τον αλγόριθμο *αναζήτησης πρώτα σε βάθος* (*depth-first search*).
- **breadth**: Αντίθετα με την προηγούμενη στρατηγική, οι "νέοι" κανόνες μπαίνουν "κάτω" από τους "παλαιούς". Το όνομα της στρατηγικής έχει να κάνει με το γεγονός ότι η εκτέλεση ενός συνόλου κανόνων με αυτήν τη στρατηγική συμπεριφέρεται σαν τον αλγόριθμο *αναζήτησης πρώτα σε πλάτος* (*breadth-first search*).
- **simplicity**: Οι κανόνες με απλούστερες συνθήκες κατατάσσονται "πάνω" από τους κανόνες με τις περισσότερο πολύπλοκες. Η πολυπλοκότητα ενός κανόνα εξαρτάται από τον αριθμό των συνθηκών και από τις συγκρίσεις (περιορισμούς) που λαμβάνουν χώρα στις συνθήκες του κανόνα.
- **complexity**: Αντίθετα με την προηγούμενη στρατηγική, οι πιο πολύπλοκοι κανόνες μπαίνουν "πάνω" από τους "απλούστερους".
- **lex**: Η στρατηγική αυτή πρωτοεμφανίστηκε στο σύστημα OPS5. Οι κανόνες οι οποίοι ενεργοποιούνται από "νεότερα" γεγονότα κατατάσσονται υψηλότερα στην ατζέντα. Για τους κανόνες οι οποίοι κατατάσσονται στην "ίδια ομάδα" με βάση το προηγούμενο κριτήριο, υψηλότερα κατατάσσονται οι κανόνες οι οποίοι έχουν περισσότερες συνθήκες. Ουσιαστικά αποτελεί συνδυασμό των στρατηγικών **depth** και **complexity**.
- **mea**: Και αυτή η στρατηγική εμφανίστηκε στο σύστημα OPS5. Εξετάζεται το γεγονός το οποίο αντιστοιχεί στην *πρώτη συνθήκη* και οι κανόνες διατάσσονται με βάση το πότε εισήχθηκε αυτό στη λίστα. Όσο νεότερο είναι το γεγονός τόσο "ψηλότερα" μπαίνει ο κανόνας. Για κανόνες οι οποίοι έχουν την ίδια σειρά με βάση αυτό το κριτήριο χρησιμοποιείται η στρατηγική **lex**.
- **random**: Οι κανόνες μπαίνουν στην ατζέντα με τυχαίο τρόπο.

Υπενθυμίζεται ότι η στρατηγική επίλυσης συγκρούσεων εφαρμόζεται σε κανόνες ίδιας προτεραιότητας.

Π1.8 Η Γλώσσα COOL

Το CLIPS υποστηρίζει τη δυνατότητα αντικειμενοστραφούς προγραμματισμού μέσω της γλώσσας COOL (CLIPS Object-Oriented Language). Η COOL έχει πέντε βασικά χαρακτηριστικά των αντικειμενοστραφών γλωσσών προγραμματισμού: *αφαίρεση*

(*abstraction*), *εγκλεισμό* (*encapsulation*), *κληρονομικότητα* (*inheritance*), *πολυμορφισμό* (*polymorphism*) και *δυναμική δέσμευση* (*dynamic binding*).

Η *αφαίρεση* είναι μία περισσότερο διαισθητική, υψηλού επιπέδου αναπαράσταση μίας σύνθετης έννοιας. Στη γλώσσα COOL, ο ορισμός νέων κλάσεων επιτρέπει την αφαίρεση νέων τύπων δεδομένων. Οι ιδιότητες (slots) και οι μέθοδοι (message-handlers) αυτών των κλάσεων περιγράφουν τις ιδιότητες (properties) και τη συμπεριφορά (behavior) μίας καινούριας ομάδας αντικειμένων.

Τα χαρακτηριστικά ενός αντικειμένου δεν είναι απευθείας προσβάσιμα στον υπόλοιπο κόσμο, δηλαδή στο υπόλοιπο πρόγραμμα. Συνήθως η εσωτερική κατάσταση του αντικειμένου αποκρύπτεται. Αυτή η ιδιότητα ονομάζεται *εγκλεισμός* των ιδιοτήτων του αντικειμένου. Η COOL υποστηρίζει την ιδιότητα του εγκλεισμού, απαιτώντας την αποστολή μηνυμάτων (messages) για το χειρισμό στιγμιοτύπων (instances) των κλάσεων, ορισμένων από το χρήστη. Ένα στιγμιότυπο δε μπορεί να αποκριθεί σε ένα μήνυμα για το οποίο δεν έχει καθορισμένη μέθοδο.

Οι κλάσεις είναι συνήθως οργανωμένες σε ιεραρχίες. Οι πιο γενικές κλάσεις είναι τοποθετημένες ψηλά στην ιεραρχία, ενώ οι πιο συγκεκριμένες χαμηλότερα. Οι κλάσεις που βρίσκονται ψηλά στην ιεραρχία έχουν κάποια γενικά χαρακτηριστικά και μεθόδους τα οποία είναι κοινά για όλες τις κλάσεις που βρίσκονται χαμηλότερα στην ιεραρχία. Για την αποφυγή της επανάληψης ορισμού κοινών χαρακτηριστικών και μεθόδων υπάρχει η *κληρονομικότητα*, σύμφωνα με την οποία η δομή και η συμπεριφορά μιας γενικότερης κλάσης κληρονομείται στις περισσότερο συγκεκριμένες.

Ακόμη, η COOL υποστηρίζει την *πολλαπλή κληρονομικότητα* (*multiple inheritance*), όπου μία κλάση δε συνδέεται ιεραρχικά μόνο με μία γενικότερη κλάση αλλά με περισσότερες. Η κλάση που συνδέεται με πολλαπλές γενικότερες κλάσεις κληρονομεί χαρακτηριστικά και μεθόδους από όλες. Η COOL, χρησιμοποιώντας την υπάρχουσα ιεραρχία των κλάσεων, ορίζει μία λίστα, τη *λίστα προτεραιότητας κλάσεων* (*class precedence list*), για μία νέα κλάση. Αντικείμενα, τα οποία είναι στιγμιότυπα της νέας κλάσης, μπορούν να κληρονομήσουν ιδιότητες και συμπεριφορά από κάθε μία από τις κλάσεις της λίστας αυτής. Η λέξη *προτεραιότητα* υποδηλώνει ότι μία μέθοδος μίας κλάσης, που είναι πρώτη στη λίστα, υπερισχύει της ίδιας μεθόδου άλλης κλάσης που βρίσκεται πιο μετά στη λίστα.

Δύο αντικείμενα διαφορετικής κλάσης μπορεί να απαντήσουν στο ίδιο μήνυμα με ένα εντελώς διαφορετικό τρόπο. Αυτή η ιδιότητα ονομάζεται *πολυμορφισμός*. Ο πολυμορφισμός επιτυγχάνεται επισυνάπτοντας μεθόδους που έχουν το ίδιο όνομα αλλά εκτελούν διαφορετικές ενέργειες, στις κλάσεις των δύο αντικειμένων αντίστοιχα.

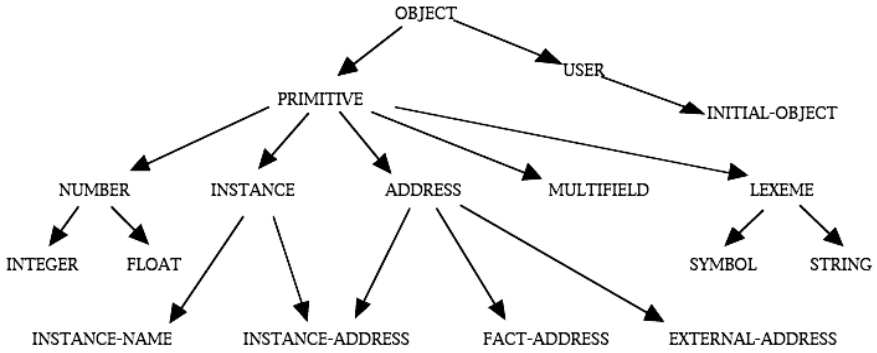
Η ιδιότητα της *δυναμικής δέσμευσης* υποστηρίζεται στην COOL με την έννοια ότι μία *αναφορά αντικειμένου* (*object reference*) κατά την κλήση μίας συνάρτησης δεν έχει τιμή μέχρι τη στιγμή της εκτέλεσης. Για παράδειγμα, ένα όνομα στιγμιότυπου (instance-name) ή μία μεταβλητή μπορεί να αναφέρεται σε ένα αντικείμενο όταν στέλνεται ένα μήνυμα και να αναφέρεται σε ένα άλλο αντικείμενο κάποια άλλη στιγμή αργότερα.

Π1.8.1 Προκαθορισμένες Κλάσεις Συστήματος

Η COOL ορίζει δεκαεπτά κλάσεις συστήματος: **OBJECT**, **USER**, **INITIAL-OBJECT**, **PRIMITIVE**, **NUMBER**, **INTEGER**, **FLOAT**, **INSTANCE**, **INSTANCE-NAME**, **INSTANCE-ADDRESS**, **ADDRESS**,

FACT-ADDRESS, **EXTERNAL-ADDRESS**, **MULTIFIELD**, **LEXEME**, **SYMBOL** και **STRING**. Αυτές οι κλάσεις δεν πρέπει να διαγραφούν ή να τροποποιηθούν. Στο Σχήμα Π2.1 απεικονίζονται οι σχέσεις κληρονομικότητας μεταξύ αυτών των κλάσεων.

Όλες οι κλάσεις συστήματος, εκτός της **INITIAL-OBJECT**, είναι *αφηρημένες* (*abstract*) κλάσεις, δηλαδή χρησιμοποιούνται μόνο για λόγους κληρονομικότητας. Άμεσα αντικείμενα αυτών των κλάσεων δεν επιτρέπονται. Καμία από αυτές τις κλάσεις δεν έχει ιδιότητες και, εκτός από την **USER**, καμία δεν έχει μεθόδους. Ωστόσο, ο χρήστης μπορεί να επισυνάψει μεθόδους σε όλες τις κλάσεις συστήματος εκτός από τις κλάσεις **INSTANCE**, **INSTANCE-ADDRESS** και **INSTANCE-NAME**. Η κλάση **OBJECT** είναι η υπερκλάση όλων των άλλων κλάσεων, συμπεριλαμβανομένων και των ορισμένων από το χρήστη κλάσεων.



Σχήμα Π2.1: Ιεραρχία των προκαθορισμένων κλάσεων συστήματος.

Όλες οι κλάσεις, που ορίζονται από το χρήστη, θα έπρεπε, αλλά δεν απαιτείται, να κληρονομούν άμεσα ή έμμεσα από την κλάση **USER**, γιατί αυτή η κλάση έχει όλες τις βασικές μεθόδους του συστήματος, όπως *αρχικοποίηση* (*initialization*) και *διαγραφή* (*deletion*).

Π1.8.2 Ορισμός Κλάσεων Χρήστη

Στην COOL οι κλάσεις ορίζονται μέσω της ειδικής συνάρτησης **defclass**. Με τη **defclass** καθορίζονται οι ιδιότητες και η συμπεριφορά μίας κλάσης αντικειμένων.

defclass

Σύνταξη:

```
(defclass <name> [<comment>]
      (is-a <superclass-name>+)
      [<role>]
      [<pattern-match-role>]
      <slot>*
)
```

όπου **<name>** το όνομα της κλάσης, **<superclass-name>+** μία ή περισσότερες υπερκλάσεις από τις οποίες η νέα κλάση κληρονομεί ιδιότητες και μεθόδους, **<role>** ένας προσ-

διοριστής, ο οποίος καθορίζει εάν θα δημιουργούνται άμεσα αντικείμενα της νέας κλάσης, **<pattern-match-role>** ένας προσδιοριστής, ο οποίος καθορίζει εάν τα αντικείμενα αυτής της κλάσης μπορούν να ταιριάζουν με πρότυπα αντικειμένων στο αριστερό μέρος των κανόνων και **<slot>*** μία λίστα ιδιοτήτων που ορίζονται στη νέα κλάση.
Παράδειγμα:
Παρακάτω φαίνεται ο ορισμός δύο κλάσεων **vehicle** (όχημα) και **car** (αυτοκίνητο). Η πρώτη περιγράφει γενικά ένα όχημα και πρόκειται για βασική κλάση (υποκλάση της κλάσης συστήματος **USER**) και έχει χαρακτηριστικά, όπως το είδος καυσίμου, τη χωρητικότητα της δεξαμενής καυσίμων του και την ποσότητα καυσίμου με την οποία ανεφοδιάστηκε κατά τον πιο πρόσφατο ανεφοδιασμό. Η δεύτερη περιγράφει ένα αυτοκίνητο ως εξειδίκευση του οχήματος, επεκτείνοντας τα παραπάνω γενικά χαρακτηριστικά με πιο συγκεκριμένα, όπως ρυθμός κατανάλωσης καυσίμου και ένδειξη προσωρινού χιλιομετρητή.

```
(defclass vehicle
   (is-a USER)
   (slot fuel-type (type SYMBOL))
   (slot tank-capacity (type INTEGER))
   (slot fuel-loaded (type INTEGER))
)

(defclass car
   (is-a vehicle)
   (slot consumption-rate (type FLOAT))
   (slot reset-able-counter (type INTEGER))
)
```

Λίστα υπερκλάσεων

Όλες οι ορισμένες από το χρήστη κλάσεις πρέπει να κληρονομούν τουλάχιστον από μία κλάση, και για αυτό το σκοπό η COOL παρέχει προκαθορισμένες κλάσεις συστήματος για να χρησιμοποιηθούν ως βάση για την παραγωγή νέων κλάσεων, όπως είναι η **USER** και η **INITIAL-OBJECT**. Η λίστα υπερκλάσεων υποδηλώνει προτεραιότητα στην κληρονόμηση ιδιοτήτων και μεθόδων. Επιπρόσθετα, οι ιδιότητες που ορίζονται στη συνάρτηση **defclass** υπερισχύουν αυτών που κληρονομούνται από άλλες κλάσεις. Για παράδειγμα, στην ακόλουθη δήλωση **defclass** προτεραιότητα στον ορισμό ιδιοτήτων και μεθόδων έχει η κλάση **A** και στη συνέχεια η προτεραιότητα στην κληρονόμηση ορίζεται με τη σειρά **(B C D)**.

```
(defclass A
   (is-a B C D)
   ...
)
```

Αφηρημένες κλάσεις

Με τον προσδιοριστή **role** μπορεί μία κλάση να οριστεί ως *αφηρημένη* (*abstract*) ή *συγκεκριμένη* (*concrete*).

```
(defclass A
   ...
```

```
    (role abstract/concrete)
    ...
)
```

Μία αφηρημένη κλάση χρησιμοποιείται μόνο για λόγους κληρονόμησης ιδιοτήτων και μεθόδων σε υποκλάσεις της, και δεν έχει άμεσα στιγμιότυπα. Αν χρησιμοποιηθεί ποτέ η συνάρτηση **make-instance** (βλ. παρακάτω) για μία τέτοια κλάση θα εμφανιστεί μήνυμα λάθους

Μία συγκεκριμένη κλάση μπορεί να έχει άμεσα στιγμιότυπα. Αν δεν υπάρχει ο συγκεκριμένος προσδιοριστής στον ορισμό μιας κλάσης, καθορίζεται μέσω της κληρονομικότητας. Μία υποκλάση της κλάσης συστήματος **USER** θεωρείται εξ' ορισμού **concrete** κλάση, αν δεν προσδιορίζεται ο ρόλος της.

Ταυτοποιήσιμες κλάσεις

Με τον προσδιοριστή **pattern-match** μπορεί μία κλάση να οριστεί ως *ταυτοποιήσιμη/αντιδραστική* (*reactive*) ή *μη ταυτοποιήσιμη* (*non-reactive*).

```
(defclass A
    ...
    (pattern-match reactive/non-reactive)
    ...
)
```

Όταν μία κλάση χαρακτηρίζεται ως **reactive** τότε τα αντικείμενά της μπορούν να ταυτοποιηθούν στη συνθήκη ενός κανόνα, δηλαδή η δημιουργία ενός νέου αντικειμένου θα προκαλέσει την ενεργοποίηση του μηχανισμού ταυτοποίησης σε όλους τους κανόνες οι οποίοι αναφέρονται σε αντικείμενα αυτής της κλάσης. Αντίθετα, όταν μία κλάση χαρακτηρίζεται ως **non-reactive** τότε τα αντικείμενά της δεν μπορούν να ταυτοποιηθούν στη συνθήκη ενός κανόνα, ακόμα και αν θεωρητικά "ταιριάζουν". Οι αφηρημένες κλάσεις δεν μπορεί να είναι **reactive**, επειδή δεν έχουν αντικείμενα.

Αν δεν υπάρχει ο συγκεκριμένος προσδιοριστής στην **defclass**, τότε η τιμή του καθορίζεται μέσω της κληρονομικότητας. Μία υποκλάση της κλάσης συστήματος **USER** θεωρείται **reactive** κλάση, αν δεν προσδιορίζεται ο ρόλος της, εκτός και αν είναι **abstract**.

Ιδιότητες

Οι ιδιότητες (slots) αποθηκεύουν τιμές που σχετίζονται με τα στιγμιότυπα-αντικείμενα κάθε κλάσης. Το όνομα της ιδιότητας μπορεί να είναι οποιοδήποτε σύμβολο, εκτός από τις λέξεις-κλειδιά **is-a** και **name** τα οποία χρησιμοποιούνται στις συνθήκες των κανόνων για να προσδιορίσουν την κλάση και το όνομα του αντικειμένου, αντίστοιχα. Υπάρχουν δύο είδη ιδιοτήτων, όπως και στα πρότυπα γεγονότων: οι *μονότιμες ιδιότητες* (**slot**) και οι *ιδιότητες πολλαπλών τιμών* (**multislot**).

Κάθε αντικείμενο έχει ένα αντίγραφο από τις ιδιότητες της άμεσης κλάσης του, καθώς και των ιδιοτήτων που κληρονομεί η κλάση. Οι ιδιότητες κληρονομούνται από τις κλάσεις με τη σειρά που ορίζεται από τη λίστα προτεραιότητας κλάσεων. Αν μία ιδιότητα κληρονομείται από δύο διαφορετικές κλάσεις, τότε χρησιμοποιείται ο ορισμός από την πιο συγκεκριμένη κλάση, δηλαδή αυτή που βρίσκεται πιο αριστερά στη λίστα

προτεραιότητας. Επίσης, είναι δυνατό να υπάρξουν και ιδιότητες που δεν κληρονομούνται λόγω της όψης (`propagation no-inherit`), εκτός από τις ιδιότητες που έχουν την όψη (`source composite`) (βλ. παρακάτω).

Οι *όψεις* (*facets*) είναι προσδιοριστές που υπάρχουν μέσα στην περιγραφή μιας ιδιότητας και τη χαρακτηρίζουν. Υπάρχουν οι εξής όψεις:

- *Προκαθορισμένη τιμή* (`default`), με την οποία μπορεί να προκαθοριστεί κάποια τιμή για μια ιδιότητα, όταν δε δίνεται τιμή στην ιδιότητα αυτή κατά τη δημιουργία αντικειμένου. Για την όψη αυτή ισχύουν όσα έχουν παρουσιαστεί στα πρότυπα γεγονότων.
- *Περιορισμοί τιμών* (*constraints*), με τους οποίους καθορίζονται ο τύπος των δεδομένων (`type`), οι επιτρεπόμενες τιμές (`allowed-values`), το εύρος αριθμητικών τιμών (`range`) και ο επιτρεπόμενος αριθμός τιμών σε ιδιότητες πολλαπλών τιμών (`cardinality`), όπως ακριβώς παρουσιάστηκε στα πρότυπα γεγονότων.
- *Αποθήκευση τιμής* (`storage`), η οποία καθορίζει εάν η τιμή μιας ιδιότητας αποθηκεύεται σε κάθε αντικείμενο ξεχωριστά (`local`), οπότε κάθε αντικείμενο μπορεί να έχει διαφορετική τιμή. Εάν η τιμή αυτής της όψης είναι `shared`, τότε η τιμή της ιδιότητας για κάθε αντικείμενο αποθηκεύεται στην κλάση, συνεπώς όλα τα αντικείμενα της κλάσης έχουν την ίδια τιμή για τη συγκεκριμένη ιδιότητα. Επίσης, όταν αλλάζει η τιμή της ιδιότητας για ένα αντικείμενο, αλλάζει για όλα. Με τη χρήση `shared` ιδιοτήτων μπορούν ουσιαστικά να αποθηκευθούν ιδιότητες που αφορούν την κλάση και όχι τα στιγμιότυπά της (*class variable*). Η εξ' ορισμού τιμή για την όψη αυτή είναι η `local`.
- *Πρόσβαση τιμής* (`access`), η οποία καθορίζει εάν η τιμή της ιδιότητας μπορεί να εγγραφεί και να διαβαστεί (`read-write`). Η τιμή `read-only` καθορίζει ότι η τιμή της ιδιότητας μπορεί μόνο να διαβαστεί. Συνεπώς ο μόνος τρόπος για να τεθεί κάποια τιμή για ένα αντικείμενο είναι κατά τη δημιουργία του αντικειμένου με τη χρήση της όψης `default`. Όμως μία `read-only` ιδιότητα με `default` τιμή ουσιαστικά είναι `storage shared`, καθώς όλα τα αντικείμενα θα έχουν την `default` τιμή για την ιδιότητα αυτή. Τέλος, η τιμή `initialize-only` υποδηλώνει ότι η τιμή της ιδιότητας καθορίζεται μόνο μία φορά τη στιγμή που δημιουργείται το αντικείμενο με τη συνάρτηση `make-instance` και στη συνέχεια η ιδιότητα μπορεί μόνο να διαβαστεί. Η εξ' ορισμού τιμή για την όψη αυτή είναι η `read-write`.
- *Προώθηση κληρονόμησης τιμής* (`propagation`), η οποία όταν έχει την τιμή `inherit` καθορίζει ότι η ιδιότητα κληρονομείται κανονικά από τις υποκλάσεις της κλάσης στην οποία υπάρχει ο ορισμός της ιδιότητας. Όταν η όψη έχει την τιμή `no-inherit` η ιδιότητα δεν κληρονομείται από τις υποκλάσεις και την έχουν μόνο τα στιγμιότυπα της συγκεκριμένης κλάσης. Η εξ' ορισμού τιμή για την όψη αυτή είναι η `inherit`.
- *Κληρονόμηση όψεων* (`source`), η οποία καθορίζει τον τρόπο με τον οποίο κληρονομούνται οι όψεις μιας ιδιότητας από υπερκλάση σε υποκλάση ή στην περίπτωση της πολλαπλής κληρονομικότητας, όταν φυσικά η ίδια ιδιότητα ορίζεται σε περισσότερες από μία κλάσεις της λίστας υπερκλάσεων. Η τιμή `exclusive`

καθορίζει πώς μόνο οι όψεις της πιο συγκεκριμένης κλάσης λαμβάνονται υπόψη, ενώ με την τιμή **composite** κληρονομούνται οι όψεις από όλες τις κλάσεις στην ιεραρχία και όχι μόνο από την πιο συγκεκριμένη. Πρακτικά, με τη χρήση της δυνατότητας **composite** μπορεί να επανακαθοριστεί κάποια όψη στην πιο συγκεκριμένη κλάση χωρίς να χρειαστεί να οριστεί ξανά ολόκληρη η ιδιότητα. Για παράδειγμα, όταν πρέπει να αλλάξει η **default** τιμή κάποιας ιδιότητας και όχι να κληρονομηθεί, τότε στην υποκλάση ορίζεται μερικώς η ιδιότητα και δηλώνεται ως **source composite** μαζί με τη νέα τιμή για την όψη **default**. Η εξ' ορισμού τιμή για την όψη αυτή είναι η **exclusive**.

- *Δυνατότητα ταυτοποίησης* (**pattern-match**), η οποία καθορίζει αν η αλλαγή τιμής της ιδιότητας προκαλεί ή όχι την ενεργοποίηση κάποιου κανόνα του οποίου η συνθήκη αναφέρεται σε αντικείμενα με τη συγκεκριμένη ιδιότητα. Οι εναλλακτικές τιμές για την όψη αυτή είναι οι **reactive** και **non-reactive**, με την προφανή σημασία όπως στον αντίστοιχο προσδιοριστή της κλάσης. Η εξ' ορισμού τιμή για την όψη αυτή είναι η **reactive**.
- *Ορατότητα τιμής* (**visibility**), η οποία καθορίζει αν επιτρέπεται η άμεση πρόσβαση στην τιμή της ιδιότητας, παρακάμπτοντας την αποστολή μηνυμάτων, σε μεθόδους των υποκλάσεων της συγκεκριμένης κλάσης στην οποία ορίζεται η ιδιότητα. Η τιμή **private** περιορίζει την άμεση πρόσβαση μόνο σε μεθόδους (message-handlers) της συγκεκριμένης κλάσης, ενώ τη τιμή **public** επιτρέπει την άμεση πρόσβαση και σε υποκλάσεις της. Η εξ' ορισμού τιμή για την όψη αυτή είναι η **private**.
- *Δημιουργία μεθόδων πρόσβασης* (**create-accessor**), η οποία καθορίζει αν κατά τη δημιουργία της κλάσης δημιουργούνται αυτόματα οι απαραίτητες μέθοδοι για την πρόσβαση (ανάγνωση, εγγραφή) στην τιμή της ιδιότητας. Η τιμή **read** καθορίζει ότι θα δημιουργηθεί μόνο η μέθοδος **get-<*slotname*>** για ανάγνωση της ιδιότητας, που είναι η εξ' ορισμού συμπεριφορά όταν η όψη **access** έχει την τιμή **read-only**. Η τιμή **write** καθορίζει ότι θα δημιουργηθεί μόνο η μέθοδος **put-<*slotname*>** για εγγραφή τιμής στην ιδιότητα. Η τιμή **read-write** καθορίζει ότι θα δημιουργηθούν και οι δύο παραπάνω μέθοδοι και αποτελεί την εξ' ορισμού συμπεριφορά όταν η όψη **access** έχει την τιμή **read-write**, επίσης. Τέλος, η τιμή **?NONE** υπαγορεύει πως καμία μέθοδος πρόσβασης δε δημιουργείται και αποτελεί την εξ' ορισμού συμπεριφορά όταν η όψη **access** έχει την τιμή **initialize-only**.
- *Υπερκάλυψη μηνύματος* (**override-message**), η οποία επιτρέπει στον προγραμματιστή να αλλάξει το όνομα της μεθόδου που χρησιμοποιείται για εγγραφή τιμής στην ιδιότητα από **put-<*slotname*>** σε οποιοδήποτε άλλο. Αυτό δε σημαίνει πως η μέθοδος **put-<*slotname*>** δε δημιουργείται. Απλά σημαίνει ότι οι συναρτήσεις **make-instance**, **initialize-instance**, **message-modify-instance**, **message-duplicate-instance**, οι οποίες χρησιμοποιούνται στη διαχείριση ενός αντικειμένου, δεν καλούν πλέον τη μέθοδο αυτή, αλλά τη μέθοδο που ορίζεται με την όψη **override-message**.

Π1.8.3 Ορισμός Στιγμιότυπων

Η δημιουργία και η αρχικοποίηση ενός στιγμιότυπου μιας κλάσης ορισμένης από το χρήστη επιτυγχάνονται μέσω της συνάρτησης **make-instance**.

make-instance

Σύνταξη:

```
(make-instance
      [<instance-name-expression>] of <class-name-expression>
      (<slot-name-expression> <expression>*)*
)
```

όπου **<instance-name-expression>** το όνομα της νέου αντικειμένου (τύπου **SYMBOL** ή **INSTANCE-NAME**), το οποίο είναι προαιρετικό, **<class-name-expression>** το όνομα της κλάσης του αντικειμένου και **(<slot-name-expression> <expression>*)*** μία λίστα ζευγών ιδιότητα-τιμή η οποία αρχικοποιεί τις αντίστοιχες ιδιότητες του αντικειμένου με τιμές.

Παράδειγμα:

Το παρακάτω παράδειγμα ορίζει ένα στιγμιότυπο με όνομα **[truck1]** της κλάσης **vehicle** που ορίστηκε προηγουμένως.

```
(make-instance [truck1] of vehicle
   (fuel-type diesel)
   (tank-capacity 100)
   (fuel-loaded 50)
)
```

Η συνάρτηση **make-instance** στέλνει αρχικά ένα μήνυμα δημιουργίας και έπειτα ένα μήνυμα αρχικοποίησης στο νέο αντικείμενο. Επίσης, επιτρέπει την αλλαγή οποιωνδήποτε προκαθορισμένων τιμών των ιδιοτήτων του συγκεκριμένου στιγμιότυπου. Η τιμή που επιστρέφει η συνάρτηση **make-instance** είναι το όνομα του νέου στιγμιότυπου, ενώ επιστρέφει το σύμβολο **FALSE** στην περίπτωση που η συνάρτηση αποτύχει. Εάν δεν καθοριστεί το όνομα του στιγμιότυπου, τότε καλείται η συνάρτηση **gensym*** για να δημιουργήσει ένα όνομα της μορφής **[genXXX]**, όπου **XXX** ένας ακέραιος αριθμός. Αν υπάρχει ήδη κάποιο αντικείμενο με το ίδιο όνομα, τότε η **make-instance** το διαγράφει πρώτα και το ξαναδημιουργεί με τις νέες τιμές.

Εδώ θα πρέπει διευκρινιστεί ότι υπάρχουν δύο τρόποι με τους οποίους καθορίζεται η ταυτότητα ενός αντικειμένου. Ο πρώτος είναι το *όνομα αντικειμένου* (**INSTANCE-NAME**) που έχει τη μορφή συμβόλου που περικλείεται μέσα σε αγκύλες, όπως για παράδειγμα: **[pump-1]**, **[foo]**, **[+++]**, **[123-890]**. Οι αγκύλες δεν αποτελούν μέρος του ονόματος αλλά ένδειξη πως πρόκειται για όνομα αντικειμένου.

Ο δεύτερος τρόπος καθορισμού της ταυτότητας ενός αντικειμένου είναι η *διεύθυνση αντικειμένου* (**INSTANCE-ADDRESS**), η οποία πρόκειται για εσωτερική αναπαράσταση της διεύθυνσης του αντικειμένου στη μνήμη. Όταν η διεύθυνση αντικειμένου τυπώνεται στην οθόνη έχει τη μορφή **<Instance-XXX>** όπου **XXX** είναι το όνομα του αντικειμένου, χωρίς αυτό να σημαίνει ότι έτσι ακριβώς είναι η μορφή με την οποία αναπαρίσταται

στη μνήμη. Η πρόσβαση στην τιμή της διεύθυνσης γίνεται μέσα από τις συνθήκες των κανόνων με τη βοήθεια της έκφρασης (pattern):

```
?var <- (object … )
```

όπως γίνεται και με τη διεύθυνση των γεγονότων. Εναλλακτικά το όνομα ενός αντικειμένου μπορεί να μετατραπεί σε διεύθυνση με τη χρήση της συνάρτησης **instance-address**.

Ένας άλλος τρόπος ορισμού μιας ομάδας στιγμιότυπων, κυρίως μέσα από αρχεία, είναι με τη χρήση της συνάρτησης **definstances**.

definstances

Σύνταξη:

```
(definstances <definstances-name> [<comment>]
      ([<instance-name-expression>] of <class-name-expression>
         (<slot-name-expression> <expression>*)*
      )*
)
```

όπου **<definstances-name>** είναι το όνομα της ομάδας των νέων αντικειμένων, το οποίο έχει μόνο διαχειριστική σημασία, **<comment>** είναι ένα προαιρετικό σχόλιο, ενώ οι υπόλοιπες παράμετροι είναι ίδιες όπως στη συνάρτηση **make-instance**.

Παράδειγμα:

Το παρακάτω είναι ένα παράδειγμα ορισμού μιας ομάδας αυτοκινήτων **family-cars** η οποία περιλαμβάνει δύο αυτοκίνητα, τα **fiat_brava** και **hyundai_lantra**.

```
(definstances family-cars
   (fiat_brava of car
      (fuel-type benzine)
      (tank-capacity 45)
      (fuel-loaded 30)
      (consumption-rate 10.5)
      (reset-able-counter 250)
   )
   (hyundai_lantra of car
      (fuel-type benzine)
      (tank-capacity 58)
      (fuel-loaded 47)
      (consumption-rate 12.3)
      (reset-able-counter 110)
   )
)
```

Τα αντικείμενα που έχουν οριστεί με την **definstances** δημιουργούνται στη μνήμη κάθε φορά που εκτελείται η εντολή **reset**, η οποία διαγράφει όλα τα τρέχοντα στιγμιότυπα και καλεί (εσωτερικά) τη συνάρτηση **make-instance** για κάθε στιγμιότυπο που καθορίζεται σε μια συνάρτηση **definstances**.

Η συνάρτηση **definstances** δεν μπορεί να χρησιμοποιήσει κλάσεις, οι οποίες δεν έχουν προηγουμένως οριστεί, μέσα στο ίδιο ή σε προηγούμενα αρχεία. Τα στιγμιότυπα

αυτής της συνάρτησης δημιουργούνται με τη σειρά, και αν η δημιουργία κάποιου στιγμιότυπου αποτύχει, τα υπόλοιπα στιγμιότυπα της **definstances** δε θα δημιουργηθούν.

Π1.8.4 Χειρισμός Στιγμιότυπων

Ο χειρισμός των αντικειμένων, δηλαδή η δημιουργία και διαγραφή αντικειμένων, καθώς και η ανάκληση και τροποποίηση των τιμών των ιδιοτήτων τους, πραγματοποιείται μέσω της αποστολής μηνυμάτων σε αυτά με τη χρήση της συνάρτησης **send**.

send

Σύνταξη:

```
(send <object-expression> <message-name-expression> <expression>*)
```

όπου **<object-expression>** είναι το όνομα ή η διεύθυνση του αντικειμένου στο οποίο αποστέλλεται το μήνυμα, **<message-name-expression>** είναι το όνομα του μηνύματος και **<expression>** είναι μία προαιρετική λίστα παραμέτρων εισόδου για το μήνυμα.

Παράδειγμα:

Στα παρακάτω παραδείγματα φαίνονται ένα μήνυμα τύπου **get-XXX** για ανάκληση τιμής ιδιότητας από αντικείμενο, το οποίο δεν έχει παραμέτρους εισόδου, ένα μήνυμα τύπου **put-XXX** για αποθήκευση τιμής ιδιότητας, όπου η παράμετρος είναι η νέα τιμή της ιδιότητας και ένα μήνυμα **print** για εκτύπωση (όχι επιστροφή) όλων των ιδιοτήτων ενός αντικειμένου. Οι συναρτήσεις **get-XXX** και **put-XXX** δημιουργούνται αυτόματα για κάθε ιδιότητα ενός αντικειμένου κατά τον ορισμό του.

```
CLIPS> (send [fiat_brava] get-fuel-type)
benzine
CLIPS> (send [fiat_brava] put-fuel-type petroleum)
petroleum
CLIPS> (send [fiat_brava] print)
[fiat_brava] of car
(fuel-type petroleum)
(tank-capacity 45)
(fuel-loaded 30)
(consumption-rate 10)
(reset-able-counter 250)
```

Η κλήση της συνάρτησης **send** ουσιαστικά προκαλεί την εκτέλεση της αντίστοιχης μεθόδου (message-handler) στο πλαίσιο του αντικειμένου-παραλήπτη του μηνύματος (βλ. παρακάτω). Το αποτέλεσμα που επιστρέφει η συνάρτηση **send** συμπίπτει με το αποτέλεσμα της αντίστοιχης μεθόδου.

Η προσπέλαση των τιμών των ιδιοτήτων ενός στιγμιότυπου επιτυγχάνεται είτε με την αποστολή μηνυμάτων στο στιγμιότυπο, όπως στα παραπάνω παραδείγματα, είτε με άμεση προσπέλαση, η οποία επιτρέπεται μόνο για τις μεθόδους που ορίζονται στην κλάση στην οποία ανήκει το αντικείμενο. Πηγές εξωτερικές προς το αντικείμενο, όπως κανόνες ή συναρτήσεις, μπορούν να προσπελάσουν τιμές από ιδιότητες μόνο αποστέλλοντας μηνύματα στο αντικείμενο.

Για να αλλάξουν οι τιμές πολλών ιδιοτήτων ταυτόχρονα σε ένα αντικείμενο χωρίς να χρειαστεί να γραφούν πολλές εντολές **send**, χρησιμοποιείται η συνάρτηση **modify-instance**.

modify-instance

Σύνταξη:

```
(modify-instance <instance>
      (<slot-name-expression> <expression>*)*
)
```

όπου **<instance>** είναι το όνομα ή η διεύθυνση του αντικειμένου του οποίου οι ιδιότητες θα αλλάξουν και **(<slot-name-expression> <expression>*)*** είναι μία λίστα ζευγών ιδιότητα-τιμή η οποία αποθηκεύει στις συγκεκριμένες ιδιότητες του αντικειμένου τις αντίστοιχες τιμές.

Παράδειγμα:

Το παρακάτω παράδειγμα μεταβάλει τις τιμές των ιδιοτήτων **reset-able-counter** και **fuel-loaded** στο αντικείμενο **[fiat_brava]**.

```
(modify-instance [fiat_brava]
   (reset-able-counter 0)
   (fuel-loaded 45)
)
```

Η συνάρτηση **modify-instance** εσωτερικά εκτελεί πολλές συναρτήσεις **send** και επιστρέφει **TRUE** εάν όλες οι αλλαγές τιμών έγιναν με επιτυχία, αλλιώς επιστρέφει **FALSE**.

Για τη διαγραφή ενός αντικειμένου πρέπει να αποσταλεί το μήνυμα **delete** στο αντικείμενο αυτό. Για παράδειγμα, εάν είναι επιθυμητό να διαγραφεί το αντικείμενο **[truck1]**, που έχει οριστεί προηγουμένως, τότε θα σταλεί το παρακάτω μήνυμα:

```
(send [truck1] delete)
```

Π1.8.5 Ορισμός Μεθόδων

Η συμπεριφορά μιας κλάσης αντικειμένων στη λήψη ενός μηνύματος καθορίζεται από την αντίστοιχη *μέθοδο* ή *χειριστή-μηνύματος* (*message handler*), όπως αποκαλείται στην ορολογία του CLIPS, που δεν είναι τίποτε άλλο παρά ένα κομμάτι *διαδικαστικού κώδικα* (*procedural code*). Το αποτέλεσμα της εκτέλεσης μιας μεθόδου και συνεπώς του αντίστοιχου μηνύματος είναι μία επιστρεφόμενη τιμή ή/και μία ή περισσότερες *παρενέργειες* (*side-effects*), δηλαδή ενέργειες οι οποίες δεν είναι "ορατές" στον αποστολέα του μηνύματος.

Για ένα συγκεκριμένο μήνυμα, μπορεί κάθε υπερκλάση, από την οποία κληρονομεί η συγκεκριμένη κλάση, να έχει ορισμένες μεθόδους. Με αυτόν τον τρόπο, η κλάση και όλες οι υπερκλάσεις της μοιράζονται την εργασία του χειρισμού ενός μηνύματος. Οι μέθοδοι κάθε κλάσης χειρίζονται εκείνο το τμήμα του μηνύματος που προορίζεται για τον εαυτό τους.

Επιπλέον, στο εσωτερικό μιας κλάσης, για κάθε μήνυμα μπορεί να υπάρχουν τέσσερις διαφορετικές μέθοδοι οι οποίες ανήκουν στις κατηγορίες: **primary**, **before**, **after** και **around**. Οι μέθοδοι **before** και **after** είναι βοηθητικές και χρησιμοποιούνται μόνο

για παρενέργειες. Οι τιμές που επιστρέφουν αγνοούνται. Οι **before** μέθοδοι εκτελούνται πριν τις **primary** μεθόδους, ενώ οι **after** μέθοδοι εκτελούνται μετά. Γενικά, η επιστρεφόμενη τιμή ενός μηνύματος αποδίδεται από τις **primary** μεθόδους. Ωστόσο και οι **around** μέθοδοι μπορούν να επιστρέψουν μία τιμή. Οι **around** μέθοδοι δημιουργούν ένα περιβάλλον για την εκτέλεση των υπόλοιπων μεθόδων. Ξεκινούν την εκτέλεση πριν από τις άλλες μεθόδους και συνεχίζουν αφού τελειώσουν όλες οι υπόλοιπες μέθοδοι, έως ότου εκτελεστούν όλες οι εντολές τους.

Οι μέθοδοι ορίζονται με τη βοήθεια της ειδικής συνάρτησης **defmessage-handler**.

defmessage-handler

Σύνταξη:

```
(defmessage-handler <class-name> <message-name>
                    [<handler-type>] (<parameter>*)
                    <action>*
)
```

όπου **<class-name>** είναι το όνομα της κλάσης στην οποία ανήκει η μέθοδος, **<message-name>** το όνομα της μεθόδου, **<handler-type>** η κατηγορία της μεθόδου, **<parameter>*** η λίστα των τυπικών παραμέτρων (μεταβλητών) της μεθόδου και **<action>*** οι εντολές ή εκφράσεις που υλοποιούν τη μέθοδο. Η επιστρεφόμενη τιμή της μεθόδου είναι η τιμή της τελευταίας έκφρασης.

Παράδειγμα:

Στο παρακάτω παράδειγμα ορίζεται η μέθοδος **remaining-distance** της κλάσης **car** που ορίστηκε προηγουμένως. Η μέθοδος αυτή υπολογίζει την εναπομείνασα απόσταση που μπορεί να διανύσει το αυτοκίνητο χωρίς ανεφοδιασμό. Η έκφραση **?self:fuel-loaded** επιστρέφει την τιμή του χαρακτηριστικού **fuel-loaded** του αντικειμένου το οποίο έλαβε το μήνυμα **remaining-distance**. Γενικά, η μεταβλητή **?self** αντιπροσωπεύει το *ενεργό αντικείμενο* (*active instance*), δηλαδή το αντικείμενο που έλαβε το μήνυμα και στο πλαίσιο (*context*) του οποίου εκτελείται η μέθοδος.

```
(defmessage-handler car remaining-distance primary ()
   (bind ?fuel-consumed
      (* (/ ?self:reset-able-counter 100) ?self:consumption-rate))
   (bind ?remaining-fuel
      (- ?self:fuel-loaded ?fuel-consumed))
   (* 100 (/ ?remaining-fuel ?self:consumption-rate))
)
```

Για να χρησιμοποιηθεί η έκφραση **?self:fuel-loaded** στη μέθοδο **remaining-distance** πρέπει να οριστεί η ιδιότητα **fuel-loaded** της κλάσης **vehicle** ως **public**, ειδάλλως δεν μπορεί να γίνει απευθείας πρόσβαση στην ιδιότητα αυτή από μέθοδο υποκλάσης της κλάσης **vehicle**.

```
(defclass vehicle
   (is-a USER)
   (slot fuel-type (type SYMBOL))
   (slot tank-capacity (type INTEGER))
   (slot fuel-loaded (type INTEGER) (visibility public))
```

```
)
```

Τέλος, η έκφραση **?self:<slot-name>** μπορεί να χρησιμοποιηθεί σε συνδυασμό με την εντολή **bind** και για την αποθήκευση τιμής σε μία ιδιότητα, μόνο μέσα από μεθόδους της (των) κλάσης (κλάσεων) που έχει (έχουν) δικαίωμα απευθείας πρόσβασης στην ιδιότητα αυτή. Για παράδειγμα, η παρακάτω μέθοδος ανεφοδιάζει ένα αυτοκίνητο με συγκεκριμένη ποσότητα καυσίμου, μηδενίζοντας επίσης τον προσωρινό χιλιομετρητή του. Η μέθοδος επίσης ελέγχει αν η ποσότητα καυσίμου ξεπερνάει τη χωρητικότητα του ρεζερβουάρ. Στην περίπτωση αυτή η μέθοδος επιστρέφει την πραγματική ποσότητα καυσίμου που προστέθηκε, ενώ αλλιώς επιστρέφει την αρχική ποσότητα καυσίμου.

```
(defmessage-handler car full-fuel-tank primary (?l)
   (bind ?target-fuel (+ ?self:fuel-loaded ?l))
   (if (<= ?target-fuel ?self:tank-capacity)
      then
         (bind ?actual-fuel ?l)
         (bind ?self:fuel-loaded ?target-fuel)
      else
         (bind ?actual-fuel (- ?l (- ?target-fuel ?self:tank-capacity)))
         (bind ?self:fuel-loaded ?self:tank-capacity)
   )
   (bind ?self:reset-able-counter 0)
   ?actual-fuel
)
```

Π1.8.6 Χρήση Αντικειμένων σε Κανόνες

Τα αντικείμενα των κλάσεων που ορίζει ο χρήστης μπορούν να ταυτοποιηθούν στη συνθήκη ενός κανόνα και μεταβληθούν από τις ενέργειες του κανόνα. Η σύνταξη των συνθηκών που αναφέρονται σε αντικείμενα είναι η ακόλουθη:

```
(object
   (is-a <constraint>) |
   (name <constraint>) |
   (<slot-name> <constraint>*)*
)
```

Ο περιορισμός **is-a** χρησιμεύει στον περιορισμό των αντικειμένων που ταιριάζουν στη συνθήκη σε αυτά που ανήκουν σε μία συγκεκριμένη κλάση ή υποκλάσεις της. Ο περιορισμός **name** χρησιμεύει στον περιορισμό της ταυτοποίησης μόνο σε συγκεκριμένα αντικείμενα. Αν στη θέση του περιορισμού υπάρχει μεταβλητή, τότε απλά επιστρέφεται το όνομα του αντικειμένου που ταίριαξε. Για τους υπόλοιπους περιορισμούς, ισχύουν τα ίδια με τα πρότυπα γεγονότων.

Παράδειγμα:

Ο παρακάτω κανόνας "αναζητά" αντικείμενα της κλάσης **car** (ή πιθανών υποκλάσεών της) για τα οποία η τιμή της ιδιότητας fuel-type είναι ίση με benzine. Επίσης, ανακαλείται το όνομα του αντικειμένου στη μεταβλητή ?x η οποία χρησιμοποιείται στην ενέργεια του κανόνα για εκτύπωση.

```
(defrule ask-data
```

```
   (object (is-a car) (name ?x) (fuel-type benzine))
      =>
   (printout t "Car: " ?x crlf)
)
```

Εάν στην οθόνη πρέπει να εκτυπωθεί όχι το όνομα αλλά η διεύθυνση του αντικειμένου, τότε ο παραπάνω κανόνας πρέπει να διατυπωθεί ως εξής:

```
(defrule ask-data
   ?y <- (object (is-a car) (fuel-type benzine))
      =>
   (printout t "Car: " ?y crlf)
)
```

Εάν πρέπει να τυπωθούν όχι μόνο τα αυτοκίνητα αλλά όλα τα βενζινοκίνητα οχήματα τότε πρέπει να αλλάξει ο περιορισμός **is-a**:

```
(defrule ask-data
   (object (is-a vehicle) (name ?x) (fuel-type benzine))
      =>
   (printout t "Vehicle: " ?x crlf)
)
```

Η παράλειψη του περιορισμού **is-a** έχει ως αποτέλεσμα την επιλογή όλων των αντικειμένων που έχουν την ιδιότητα **fuel-type** με τη συγκεκριμένη τιμή, ανεξαρτήτως κλάσης:

```
(defrule ask-data
   (object (name ?x) (fuel-type benzine))
      =>
   (printout t "Object: " ?x crlf)
)
```

Όταν ένα αντικείμενο δημιουργείται ή διαγράφεται επηρεάζονται όλοι οι κανόνες των οποίων η συνθήκη αναφέρεται στο αντικείμενο αυτό. Όταν αλλάζει μία ιδιότητα επηρεάζονται μόνο οι κανόνες που αναφέρονται στη συγκεκριμένη ιδιότητα και όχι κανόνες που αναφέρονται στο ίδιο αντικείμενο αλλά όχι στη συγκεκριμένη ιδιότητα.

Η παραπάνω συμπεριφορά αποτελεί μια σημαντική διαφορά των αντικειμένων με τα πρότυπα γεγονότων, πέρα βέβαια από τις υπόλοιπες διαφορές, όπως η ύπαρξη ιεραρχίας και κληρονομικότητας, ο εγκλεισμός, κτλ.

Παράδειγμα:

Το παρακάτω παράδειγμα επιδεικνύει την παραπάνω διαφορά στη συμπεριφορά των αντικειμένων και των προτύπων γεγονότων. Συγκεκριμένα, και οι δύο κανόνες αλλάζουν τις τιμές των ιδιοτήτων **reset-counter** και **fuel-loaded**, ο πρώτος χρησιμοποιώντας αντικείμενα ενώ ο δεύτερος χρησιμοποιώντας πρότυπα γεγονότων. Οι συνθήκες και των δύο κανόνων δεν αναφέρονται στις τιμές των δύο παραπάνω ιδιοτήτων. Παρόλα αυτά ο κανόνας **stopped-at-gas-station2** θα εκτελούνταν επ' άπειρο για το ίδιο αυτοκίνητο, γιατί η αλλαγή κάποιας ιδιότητας στην ενέργεια του κανόνα προκαλεί τη διαγραφή του γεγονότος και την εισαγωγή νέου, με αποτέλεσμα το σύστημα CLIPS να "νομίζει" ότι πρόκειται για ενεργοποίηση του κανόνα με νέα δεδομένα. Αντίθετα, η

αλλαγή ιδιοτήτων ενός αντικειμένου δεν αλλάζει την ταυτότητά του, συνεπώς ο κανόνας **stopped-at-gas-station1** εκτελείται μόνο μία φορά για κάθε διαφορετικό αντικείμενο που ικανοποιεί τους περιορισμούς στη συνθήκη του κανόνα.

```
(defrule stopped-at-gas-station1
   (object (is-a car) (name ?x) (tank-capacity ?c))
   =>
   (modify-instance ?x (reset-counter 0) (fuel-loaded ?c))
)

(defrule stopped-at-gas-station2
   ?y <- (car (name ?x) (tank-capacity ?c))
   =>
   (modify ?y (reset-counter 0) (fuel-loaded ?c))
)
```

Π1.8.7 Ερωτήσεις και Ενέργειες σε Ομάδες Αντικειμένων

Η COOL δίνει τη δυνατότητα εκτέλεσης ερωτήσεων και ενεργειών σε πολλά αντικείμενα μαζί βάσει συγκεκριμένων κριτηρίων αναζήτησης που θέτει ο χρήστης, με τη χρήση συναρτήσεων, χωρίς να χρειάζεται δηλαδή η συγγραφή κάποιων κανόνων. Η δυνατότητα αυτή "θυμίζει" *γλώσσα ερωταπαντήσεων* (*query language*) σε σύστημα διαχείρισης βάσεων δεδομένων, όπως για παράδειγμα η SQL.

Πίνακας Π2.10: Συναρτήσεις ομάδων αντικειμένων.

Συνάρτηση	Σκοπός
any-instancep	Ελέγχει αν υπάρχει έστω και μία ομάδα αντικειμένων που ικανοποιεί μια συνθήκη.
find-instance	Επιστρέφει την πρώτη ομάδα αντικειμένων που ικανοποιεί μια συνθήκη.
find-all-instances	Επιστρέφει όλες τις ομάδες αντικειμένων που ικανοποιούν μια συνθήκη.
do-for-instance	Εκτελεί ένα σύνολο ενεργειών πάνω στην πρώτη ομάδα αντικειμένων που ικανοποιεί μια συνθήκη.
do-for-all-instances	Εκτελεί ένα σύνολο ενεργειών πάνω σε κάθε ομάδα αντικειμένων που ικανοποιεί μια συνθήκη.
delayed-do-for-all-instances	Πρώτα βρίσκει όλες τις ομάδες αντικειμένων που ικανοποιούν μια συνθήκη και στη συνέχεια εκτελεί ένα σύνολο ενεργειών σε αυτές.

Συνολικά υπάρχουν έξι τέτοιες συναρτήσεις (Πίνακας Π2.10). Σε όλες, ο τρόπος με τον οποίο προσδιορίζονται τα αντικείμενα στα οποία θα ενεργήσει η εντολή είναι κοινός. Συγκεκριμένα, σε κάθε εντολή ορίζεται ένα *πρότυπο ομάδας αντικειμένων* (*instance set template*) με την ακόλουθη μορφή:

$$((?var_1\ class_1)\ (?var_2\ class_2)\ \ldots\ (?var_N\ class_N))$$

καθορίζοντας ότι η μεταβλητή $?var_i$ παίρνει ως τιμές τα ονόματα όλων αντικειμένων-στιγμιότυπων της κλάσης $class_i$ και ότι εντολή θα επενεργήσει στο καρτεσιανό γινόμενο των αντικειμένων των κλάσεων $class_1 \times class_2 \times \ldots \times class_N$. Κάθε μέλος αυτού του

καρτεσιανού γινομένου ονομάζεται *ομάδα αντικειμένων* (*instance set*). Η λέξη "ομάδα" χρησιμοποιείται αντί της λέξης "σύνολο" για να γίνει ξεκάθαρο ότι ο αριθμός των αντικειμένων είναι σταθερός και καθορισμένος (για κάθε ερώτηση βέβαια) και όχι απροσδιόριστος, όπως συνήθως συμβαίνει με τα σύνολα.

any-instancep

Εφαρμόζει μια ερώτηση-συνθήκη σε κάθε ομάδα αντικειμένων που ταιριάζει με το πρότυπο που ορίζει ο χρήστης. Εάν υπάρχει τουλάχιστον μία ομάδα αντικειμένων που ικανοποιεί τη συνθήκη, τότε τερματίζει επιστρέφοντας **TRUE**, αλλιώς επιστρέφει **FALSE**.
Σύνταξη:

```
(any-instancep
      <instance-set-template>
      <query>
)
```

όπου **<instance-set-template>** είναι το πρότυπο της ομάδας αντικειμένων που θα "ερωτηθούν" και **<query>** είναι μια απλή ή σύνθετη λογική συνάρτηση η οποία παίζει το ρόλο της ερώτησης.
Παράδειγμα:
Η παρακάτω ερώτηση ελέγχει αν υπάρχουν άνδρες ηλικίας άνω των 30.

```
(any-instancep
   ((?man MAN))
   (> ?man:age 30)
)
```

Η έκφραση **((?man MAN))** αποτελεί ένα πρότυπο ομάδας αντικειμένων η οποία έχει μόνο ένα αντικείμενο της κλάσης **MAN**. Η μεταβλητή **?man** αντιπροσωπεύει τα ονόματα των αντικειμένων της κλάσης **MAN**. Η έκφραση **?man:age** στη λογική έκφραση (ερώτηση) **(> ?man:age 30)** ανακαλεί την τιμή της ιδιότητας **age** από το κάθε αντικείμενο της κλάσης **MAN**.

find-instance

Εφαρμόζει μια ερώτηση-συνθήκη σε κάθε ομάδα αντικειμένων που ταιριάζει με το πρότυπο που ορίζει ο χρήστης. Εάν υπάρχει κάποια ομάδα αντικειμένων που ικανοποιεί τη συνθήκη, τότε τερματίζει επιστρέφοντας μία λίστα με αυτήν την ομάδα των αντικειμένων, αλλιώς επιστρέφει μία κενή λίστα.
Σύνταξη:

```
(find-instance
      <instance-set-template>
      <query>
)
```

Παράδειγμα:
Η παρακάτω ερώτηση αναζητάει ένα ζευγάρι ενός άνδρα και μιας γυναίκας με την ίδια ηλικία και επιστρέφει το πρώτο που βρίσκει ή τίποτα αν δε βρει κανένα.

```
(find-instance
```

```
    ((?m MAN) (?w WOMAN))
    (= ?m:age ?w:age)
)
```

Στο παράδειγμα αυτό, η ομάδα αντικειμένων αποτελείται από ζευγάρια αντικειμένων, το πρώτο από την κλάση **MAN** και το δεύτερο από την **WOMAN**. Ουσιαστικά το συγκεκριμένο πρότυπο **((?m MAN) (?w WOMAN))** δημιουργεί το καρτεσιανό γινόμενο όλων των αντικειμένων της κλάσης **MAN** με όλα τα αντικείμενα της κλάσης **WOMAN**. Οι συνδυασμοί γίνονται θέτοντας μία συγκεκριμένη τιμή για την πρώτη μεταβλητή **?m** και στη συνέχεια συνδυάζοντας όλες τις δυνατές τιμές της δεύτερης μεταβλητής **?w**. Η διαδικασία επαναλαμβάνεται για τη δεύτερη, τρίτη, κ.ο.κ., τιμή της μεταβλητής **?m**. Το αποτέλεσμα στην παραπάνω ερώτηση θα μπορούσε να είναι μια λίστα της μορφής **([Man-1] [Woman-1])**, η οποία περιέχει τις πρώτες τιμές των δύο μεταβλητών που επαληθεύουν τη συνθήκη της ερώτησης, ή μία κενή λίστα **()**.

find-all-instances

Εφαρμόζει μια ερώτηση-συνθήκη σε κάθε ομάδα αντικειμένων που ταιριάζει με το πρότυπο που ορίζει ο χρήστης. Οι ομάδες αντικειμένων που ικανοποιούν τη συνθήκη αποθηκεύονται σε μία λίστα και όταν εξαντληθούν όλες οι δυνατές ομάδες αντικειμένων, αυτή η λίστα επιστρέφεται ως αποτέλεσμα. Φυσικά, αν καμία ομάδα αντικειμένων δεν ικανοποιεί τη συνθήκη, τότε επιστρέφεται η κενή λίστα.
Σύνταξη:

```
(find-all-instances
      <instance-set-template>
      <query>
)
```

Παράδειγμα:
Η παρακάτω ερώτηση επιστρέφει όλα τα ζευγάρια ενός άνδρα και μιας γυναίκας με την ίδια ηλικία.

```
(find-all-instances
   ((?m MAN) (?w WOMAN))
   (= ?m:age ?w:age)
)
```

Μία πιθανή απάντηση στην παραπάνω ερώτηση θα μπορούσε να είναι η λίστα **([Man-1] [Woman-1] [Man-2] [Woman-2])** όπου στα στοιχεία της λίστας εναλλάσσονται οι διαφορετικές τιμές των δύο μεταβλητών. Αν υπάρχουν n στιγμιότυπα σε κάθε ομάδα αντικειμένων και m ομάδες αντικειμένων που ικανοποιούν τη συνθήκη, τότε το μήκος της επιστρεφόμενης λίστας θα είναι $n \times m$. Τα πρώτα n στοιχεία ανήκουν στην πρώτη ομάδα αντικειμένων, τα δεύτερα n στοιχεία στη δεύτερη ομάδα αντικειμένων κ.ο.κ. Συνεπώς, η συγκεκριμένη συνάρτηση θα πρέπει να χρησιμοποιείται με προσοχή γιατί λόγω συνδυαστικής έκρηξης μπορεί να καταναλώσει τεράστια ποσά μνήμης.

do-for-instance

Εφαρμόζει μια ερώτηση-συνθήκη σε κάθε ομάδα αντικειμένων που ταιριάζει με το πρότυπο που ορίζει ο χρήστης. Αν μία ομάδα αντικειμένων ικανοποιεί τη συνθήκη,

τότε εκτελείται μία ακολουθία ενεργειών **<action>*** για αυτήν την ομάδα αντικειμένων και η συνάρτηση τερματίζει. Το αποτέλεσμα της συνάρτησης είναι η τιμή της τελευταίας ενέργειας που εκτελείται για την εν λόγω ομάδα αντικειμένων που ικανοποιεί τη συνθήκη. Αν καμία ομάδα αντικειμένων δεν ικανοποιεί τη συνθήκη, τότε η συνάρτηση επιστρέφει **FALSE**.

Σύνταξη:

```
(do-for-instance
      <instance-set-template>
      <query>
      <action>*
)
```

Παράδειγμα:

Η παρακάτω ερώτηση τυπώνει την πρώτη τριάδα διαφορετικών μεταξύ τους ανθρώπων που βρίσκει οι οποίοι έχουν την ίδια ηλικία. Στη λογική συνθήκη της ερώτησης υπάρχουν οι συναρτήσεις **neq** ώστε οι τριάδες όντως να περιέχουν διαφορετικούς ανθρώπους.

```
(do-for-instance
   ((?p1 PERSON) (?p2 PERSON) (?p3 PERSON))
   (and (= ?p1:age ?p2:age ?p3:age)
      (neq ?p1 ?p2) (neq ?p1 ?p3) (neq ?p2 ?p3))
   (printout t ?p1 " " ?p2 " " ?p3 crlf)
)
```

Αν υποτεθεί ότι υπάρχει μια τριάδα ανθρώπων που ικανοποιούν την παραπάνω ερώτηση μια πιθανή εκτύπωση θα μπορούσε να είναι η εξής:

```
[Girl-2] [Boy-2] [Boy-3]
```

do-for-all-instances

Εφαρμόζει μια ερώτηση-συνθήκη σε κάθε ομάδα αντικειμένων που ταιριάζει με το πρότυπο που ορίζει ο χρήστης. Αν μία ομάδα αντικειμένων ικανοποιεί τη συνθήκη, τότε εκτελείται μία ακολουθία ενεργειών **<action>*** για αυτήν την ομάδα αντικειμένων. Το αποτέλεσμα της συνάρτησης είναι η τιμή της τελευταίας ενέργειας που εκτελείται για την τελευταία ομάδα αντικειμένων που ικανοποιεί τη συνθήκη. Αν καμία ομάδα αντικειμένων δεν ικανοποιεί τη συνθήκη, τότε η συνάρτηση επιστρέφει **FALSE**.

Σύνταξη:

```
(do-for-all-instances
      <instance-set-template>
      <query>
      <action>*
)
```

Παράδειγμα:

Η παρακάτω ερώτηση τυπώνει όλες τις τριάδες διαφορετικών μεταξύ τους ανθρώπων οι οποίοι έχουν την ίδια ηλικία.

```
(do-for-all-instances
   ((?p1 PERSON) (?p2 PERSON) (?p3 PERSON))
```

```
   (and (= ?p1:age ?p2:age ?p3:age)
        (neq ?p1 ?p2) (neq ?p1 ?p3) (neq ?p2 ?p3))
   (printout t ?p1 " " ?p2 " " ?p3 crlf)
)
```

Αν υποτεθεί ότι υπάρχουν αρκετές τριάδες ανθρώπων που ικανοποιούν την παραπάνω ερώτηση μια πιθανή εκτύπωση θα μπορούσε να είναι η εξής:

```
[Girl-2] [Boy-3] [Boy-2]
[Girl-2] [Boy-4] [Boy-2]
[Girl-2] [Boy-4] [Boy-3]
[Boy-4] [Boy-3] [Boy-2]
...
[Boy-3] [Girl-2] [Boy-2]
...
```

Η ίδια τριάδα ανθρώπων θα τυπωθεί με όλους τους δυνατούς τρόπους, καθώς η λογική συνθήκη δεν μπορεί να το αποκλείσει. Μία πιο "εξεζητημένη" λογική συνθήκη η οποία δεν αφήνει να τυπωθούν όμοιες τριάδες δύο φορές είναι η ακόλουθη:

```
(do-for-all-instances
   ((?p1 PERSON) (?p2 PERSON) (?p3 PERSON))
   (and (= ?p1:age ?p2:age ?p3:age)
        (> (str-compare ?p1 ?p2) 0)
        (> (str-compare ?p2 ?p3) 0))
   (printout t ?p1 " " ?p2 " " ?p3 crlf)
)
```

Η παραπάνω συνθήκη στηρίζεται στο γεγονός πως τα ονόματα των αντικειμένων είναι ουσιαστικά σύμβολα τα οποία μπορούν να συγκριθούν και να διαταχθούν με τη βοήθεια της συνάρτησης **str-compare**.

delayed-do-for-all-instances

Η συνάρτηση αυτή είναι παρόμοια με την **do-for-all-instances** εκτός του ότι συγκεντρώνει σε μία προσωρινή λίστα όλες τις ομάδες αντικειμένων που ικανοποιούν τη λογική συνθήκη και μετά εκτελεί την ακολουθία ενεργειών πάνω σε κάθε ομάδα αντικειμένων που βρίσκεται σε αυτή τη λίστα. Το αποτέλεσμα της συνάρτησης είναι η τιμή της τελευταίας ενέργειας που εκτελείται για την τελευταία ομάδα αντικειμένων που ικανοποιεί τη συνθήκη. Αν καμία ομάδα αντικειμένων δεν ικανοποιεί τη συνθήκη, τότε η συνάρτηση επιστρέφει **FALSE**. Μετά το πέρας της εκτέλεσης η προσωρινή λίστα διαγράφεται από τη μνήμη. Η συνάρτηση αυτή μπορεί να καταναλώσει τεράστια ποσά μνήμης, όπως και η **find-all-instances**. Ο μοναδικός λόγος που μπορεί να χρησιμοποιηθεί αυτή η συνάρτηση αντί της **do-for-all-instances** είναι όταν η εκτέλεση των ενεργειών στο πλαίσιο μιας ομάδας αντικειμένων μπορεί να επηρεάσει το αποτέλεσμα της ερώτησης για κάποιο άλλη ομάδα αντικειμένων.
Σύνταξη:

```
(delayed-do-for-all-instances
      <instance-set-template>
      <query>
      <action>*
```

```
)
```

Παράδειγμα:
Η παρακάτω ερώτηση διαγράφει όλα τα αγόρια με τη μεγαλύτερη ηλικία.

```
(delayed-do-for-all-instances
   ((?b1 BOY))
   (not (any-instancep ((?b2 BOY)) (> ?b2:age ?b1:age)))
   (send ?b1 delete)
)
```

Η λογική έκφραση αποτελείται από μία άλλη ερώτηση ομάδας αντικειμένων (**any-instancep**) η οποία ελέγχει αν για το τρέχον αντικείμενο **?b1** της εξωτερικής ερώτησης υπάρχει άλλο αντικείμενο **?b2** της εσωτερικής ερώτησης με ηλικία μεγαλύτερη από αυτή του **?b1**. Αν ισχύει αυτό, το **?b1** δεν είναι επιλέξιμο γιατί απλά δεν έχει τη μέγιστη ηλικία. Αν χρησιμοποιηθεί η **do-for-all-instances** στο παράδειγμα αυτό, τότε κάθε φορά που βρίσκεται κάποιο αγόρι με τη μέγιστη ηλικία αυτό θα διαγράφεται με αποτέλεσμα η ερώτηση να συνεχίζεται για τα υπόλοιπα αγόρια όπου η μέγιστη ηλικία θα αντιστοιχεί πλέον σε άλλη χαμηλότερη τιμή! Άρα τελικά θα διαγραφούν πολύ περισσότερα αντικείμενα από όσα θα έπρεπε.

Π1.9 Παραδείγματα

Στη συνέχεια παρουσιάζονται τρία παραδείγματα πλήρων συστημάτων γνώσης σε CLIPS. Το πρώτο χρησιμοποιεί αδόμητα γεγονότα, το δεύτερο χρησιμοποιεί πρότυπα γεγονότων, ενώ το τρίτο χρησιμοποιεί αντικείμενα της COOL.

Π1.9.1 Αντιμετώπιση Άμεσου Κινδύνου

Το παρακάτω παράδειγμα είναι η υλοποίηση ενός απλού προβλήματος: ο χρήστης ερωτάται για τον τύπο του προβλήματος το οποίο εμφανίστηκε σε κάποιο σημείο ενός κτιριακού συγκροτήματος και μετά για τη συγκεκριμένη περιοχή που εμφανίστηκε το πρόβλημα. Έπειτα το πρόγραμμα εμφανίζει στην οθόνη τις ενέργειες που πρέπει να γίνουν στο συγκεκριμένο κτίριο αλλά και στα διπλανά σε αυτό κτίρια.

```
;;; Ο κανόνας ζητά από το χρήστη να του δώσει τον τύπο της κατάστασης
;;; ανάγκης καθώς και το κτίριο στο οποίο εκδηλώθηκε.
;;; Κατόπιν εισάγει το κατάλληλο γεγονός στη λίστα.
(defrule ask-for-emergency
   "emergency-type"
=>
   (printout t "Please enter type of emergency: ")
   (bind ?type (read))
   (printout t crlf "Please enter Building: ")
   (bind ?build (read))
   (assert (emergency ?type building ?build))
)

;;; Εισάγει ένα γεγονός "εκκένωσης κτιρίου" και παράλληλα
;;; τυπώνει στην οθόνη οδηγίες για την περίπτωση φωτιάς.
```

```
(defrule handle-fire
   "rule that handles the fire "
   (emergency fire building ?which)
=>
   (assert (evacuate ?which))
   (printout t "Open fire extinguishing system in " ?which crlf)
   (printout t "Call fire brigade" crlf)
)

;;; Εισάγει ένα γεγονός "εκκένωσης κτιρίου" και παράλληλα
;;; τυπώνει στην οθόνη οδηγίες για την περίπτωση πλημμύρας.
(defrule hanlde-flood
   "rule that handles floods"
   (emergency flood building ?which)
=>
   (assert (evacuate ?which))
   (printout t "Call fire brigade" crlf)
)

;;; Εισάγει ένα γεγονός "εκκένωσης κτιρίου" και παράλληλα
;;; τυπώνει στην οθόνη οδηγίες για την περίπτωση βόμβας.
(defrule handle-bomb
   "rule that handles a bomb warning"
   (emergency bomb building ?which)
=>
   (assert (evacuate ?which))
   (printout t "Call the bomb squad" crlf)
)

;;; Τυπώνει εντολές εκκένωσης κτιρίων.
(defrule evacuation
   "evacuation"
   (evacuate ?which)
=>
   (printout t "Evacuate  building " ?which crlf)
)

;;; Εντοπίζει κτιριακά συγκροτήματα και εισάγει
;;; και για τα διπλανά κτίρια εντολές εκκένωσης.
(defrule find-buidlings-nearby
   "the rule that evacuates the nearby buildings"
   (evacuate ?which)
   (near ?which ?other)
=>
   (assert (evacuate ?other))
)

;;; Το σύνολο των γεγονότων τα οποία δηλώνουν ποια κτίρια είναι γειτονικά.
(deffacts building-topology
   "which building is near to which"
   (near A B)
   (near B A)
```

```
    (near C D)
    (near D C)
)
```

Π1.9.2 Κίνηση Ρομπότ

Το παράδειγμα αυτό αποτελεί υλοποίηση σε CLIPS του παραδείγματος κίνησης ρομπότ που παρουσιάστηκε στο κεφάλαιο των *Συστημάτων Κανόνων*. Η υλοποίηση διαφέρει σε μερικά σημεία από τη θεωρητική παρουσίαση του παραδείγματος εφόσον οι στρατηγικές επίλυσης συγκρούσεων του συστήματος CLIPS δε συμπίπτουν με αυτές του θεωρητικού παραδείγματος. Συγκεκριμένα, στο CLIPS υπάρχουν επτά προκαθορισμένες στρατηγικές οι οποίες δεν μπορούν να συνυπάρχουν την ίδια χρονική στιγμή, με αποτέλεσμα να μην μπορεί να υπάρξει ο συνδυασμός των στρατηγικών που απαιτεί το θεωρητικό παράδειγμα, δηλαδή *αποφυγή επανάληψης*, *επιλογή του πιο ειδικού* και *τυχαία επιλογή*, με αυτήν τη σειρά.

Το πρόβλημα έγκειται κυρίως στο ότι δεν μπορούν να συνυπάρξουν οι στρατηγικές της επιλογής του πιο ειδικού με την τυχαία επιλογή. Στο συγκεκριμένο παράδειγμα, η τυχαία επιλογή είναι απαραίτητη για την επιλογή τυχαίας κατεύθυνσης όταν το ρομπότ πέφτει πάνω σε εμπόδια, ενώ η επιλογή του πιο ειδικού χρειάζεται για να δώσει προτεραιότητα στους κανόνες αποφυγής εμποδίων έναντι αυτών της κίνησης. Για το δεύτερο υπάρχει εναλλακτική λύση μέσω της χρήσης των προτεραιοτήτων κανόνων του CLIPS (salience), έτσι η στρατηγική που χρησιμοποιείται τελικά είναι η **random**.

Άλλες διαφοροποιήσεις σε σχέση με το θεωρητικό παράδειγμα είναι ο κανόνας αρχικοποίησης ο οποίος εισάγει τα δυναμικά γεγονότα, αν και η χρήση του δεν είναι απαραίτητη, η αποφυγή της εξόδου του ρομπότ από το πλέγμα, ανάγοντάς το σε αποφυγή εμποδίων, και ο τερματισμός της εκτέλεσης όταν βρεθεί κάποιο αντικείμενο.

Τέλος, το παράδειγμα χρησιμοποιεί πρότυπα γεγονότων για την αναπαράσταση των εμποδίων, των αντικειμένων και του ρομπότ. Η δομή και των τριών προτύπων είναι όμοια καθώς περιλαμβάνει ως ιδιότητες τις συντεταγμένες στις δύο διαστάσεις και περιορισμούς που αφορούν τον τύπο τιμών (ακέραιες) και το εύρος τιμών (από 1 έως 10). Η χρήση προτύπων αντί απλών γεγονότων καθιστά απλούστερη την αναπαράσταση των κανόνων κίνησης, καθώς γίνεται αναφορά μόνο στην ιδιότητα (συντεταγμένη) που επηρεάζεται από την κίνηση στον οριζόντιο ή στον κατακόρυφο άξονα.

```
;;; Πρότυπο obstacle_at που αναπαριστά τα εμπόδια στο πλέγμα.
;;; Υπάρχουν δύο ιδιότητες Xpos, Ypos για τις δύο συντεταγμένες
;;; στο διδιάστατο πλέγμα. Ο τύπος των ιδιοτήτων είναι ακέραιος
;;; με εύρος τιμών από 1 έως 10.
(deftemplate obstacle_at
   (slot Xpos (type INTEGER) (range 1 10))
   (slot Ypos (type INTEGER) (range 1 10))
)

;;; Πρότυπο object_at που αναπαριστά τα αντικείμενα στο πλέγμα.
;;; Η δομή είναι όμοια με το πρότυπο obstacle_at.
(deftemplate object_at
```

```
   (slot Xpos (type INTEGER) (range 1 10))
   (slot Ypos (type INTEGER) (range 1 10))
)

;;; Πρότυπο robot_at που αναπαριστά το ρομπότ στο πλέγμα.
;;; Η δομή είναι όμοια με το πρότυπο obstacle_at.
(deftemplate robot_at
   (slot Xpos (type INTEGER) (range 1 10))
   (slot Ypos (type INTEGER) (range 1 10))
)

(deffacts static-facts
   ;;; Εμπόδια
   (obstacle_at (Xpos 4) (Ypos 2))  (obstacle_at (Xpos 5) (Ypos 2))
   (obstacle_at (Xpos 9) (Ypos 2))  (obstacle_at (Xpos 2) (Ypos 3))
   (obstacle_at (Xpos 7) (Ypos 4))  (obstacle_at (Xpos 5) (Ypos 6))
   (obstacle_at (Xpos 7) (Ypos 7))  (obstacle_at (Xpos 3) (Ypos 8))
   (obstacle_at (Xpos 6) (Ypos 8))
   ;;; Δυνατές κατευθύνσεις κίνησης
   (choice w)  (choice e)  (choice n)  (choice s)
   ;;; Αντικείμενα
   (object_at (Xpos 7) (Ypos 2))  (object_at (Xpos 4) (Ypos 4))
   (object_at (Xpos 8) (Ypos 5))  (object_at (Xpos 10) (Ypos 6))
   (object_at (Xpos 4) (Ypos 7))
)

;;; Κανόνας αρχικοποίησης. Αρχικά δυναμικά γεγονότα. Στρατηγική random.
(defrule begin
   (initial-fact)
=>
   (set-strategy random)
   (assert (robot_at (Xpos 6) (Ypos 4)))
   (assert (direction e))
)

;;; Κανόνας εντοπισμού αντικειμένου και τερματισμού της εκτέλεσης
(defrule detect_object
   (robot_at (Xpos ?x) (Ypos ?y))
   (object_at (Xpos ?x) (Ypos ?y))
=>
   (printout t "object is found at position " ?x "-" ?y " !" crlf)
   (halt)
)

;;; Κανόνας μετακίνησης ρομπότ προς τα αριστερά
(defrule move_west
   ?f <- (robot_at (Xpos ?x))
   (direction w)
=>
   (modify ?f (Xpos (- ?x 1)))
```

```
)

;;; Κανόνας κίνησης ρομπότ προς τα δεξιά
(defrule move_east
   ?f <- (robot_at (Xpos ?x))
   (direction e)
=>
   (modify ?f (Xpos (+ ?x 1)))
)

;;; Κανόνας κίνησης ρομπότ προς τα πάνω
(defrule move_north
   ?f <- (robot_at (Ypos ?y))
   (direction n)
=>
   (modify ?f (Ypos (+ ?y 1)))
)

;;; Κανόνας κίνησης ρομπότ προς τα κάτω
(defrule move_south
   ?f <- (robot_at (Ypos ?y))
   (direction s)
=>
   (modify ?f (Ypos (- ?y 1)))
)

;;; Κανόνας αποφυγής εμποδίου κατά την κίνηση προς τα κάτω
;;; Έχει μεγαλύτερη προτεραιότητα από τον αντίστοιχο κανόνα κίνησης
;;; Εμπόδιο θεωρείται και η έξοδος από το πλέγμα
(defrule avoid_obstacle_south
   (declare (salience 1))
   (robot_at (Xpos ?x) (Ypos ?y))
   ?f <- (direction s)
   (or
      (obstacle_at (Xpos ?x) (Ypos =(- ?y 1)))    ;;; Εμπόδιο προς τα κάτω
      (test (= ?y 1))           ;;; Έξοδος από το πλέγμα (προς τα κάτω)
   )
   (choice ?nd)
=>
   (retract ?f)
   (assert (direction ?nd))
)

;;; Κανόνας αποφυγής εμποδίου κατά την κίνηση προς τα πάνω
(defrule avoid_obstacle_north
   (declare (salience 1))
   (robot_at (Xpos ?x) (Ypos ?y))
   ?f <- (direction n)
   (or
      (obstacle_at (Xpos ?x) (Ypos =(+ ?y 1)))    ;;; Εμπόδιο προς τα πάνω
```

```
      (test (= ?y 10))          ;;; Έξοδος από το πλέγμα (προς τα πάνω)
   )
   (choice ?nd)
=>
   (retract ?f)
   (assert (direction ?nd))
)

;;; Κανόνας αποφυγής εμποδίου κατά την κίνηση προς τα δεξιά
(defrule avoid_obstacle_east
   (declare (salience 1))
   (robot_at (Xpos ?x) (Ypos ?y))
   ?f <- (direction e)
   (or
      (obstacle_at (Xpos =(+ ?x 1)) (Ypos ?y))    ;;; Εμπόδιο προς τα δεξιά
      (test (= ?x 10))          ;;; Έξοδος από το πλέγμα (προς τα δεξιά)
   )
   (choice ?nd)
=>
   (retract ?f)
   (assert (direction ?nd))
)

;;; Κανόνας αποφυγής εμποδίου κατά την κίνηση προς τα αριστερά
(defrule avoid_obstacle_west
   (declare (salience 1))
   (robot_at (Xpos ?x) (Ypos ?y))
   ?f <- (direction w)
   (or
      (obstacle_at (Xpos =(- ?x 1)) (Ypos ?y))    ;;; Εμπόδιο προς τα αριστερά
      (test (= ?x 1))           ;;; Έξοδος από το πλέγμα (προς τα αριστερά)
   )
   (choice ?nd)
=>
   (retract ?f)
   (assert (direction ?nd))
)
```

Π1.9.3 Διάγνωση Βλάβης Βασισμένης σε Μοντέλο

Το παρακάτω παράδειγμα είναι η υλοποίηση του παραδείγματος της διάγνωσης βλαβών σε ένα μοντέλο ενός μηχανικού συστήματος που αποτελείται από αλληλοσυνδεδεμένα εξαρτήματα και αισθητήρες, το οποίο παρουσιάστηκε στο κεφάλαιο των *Εξελιγμένων Συλλογιστικών* των συστημάτων γνώσης. Αρχικά ο χρήστης ερωτάται για το ποιοι αισθητήρες παρουσιάζουν ασυμφωνία με τις θεωρητικές τιμές που προβλέπονται από το μοντέλο. Στο συγκεκριμένο πρόγραμμα, δεν ενδιαφέρει πώς υπολογίζονται οι θεωρητικές τιμές ή πώς διαπιστώνεται η ασυμφωνία, αλλά σκοπός είναι να εντοπιστεί το (αν είναι δυνατόν, ένα και μοναδικό) εξάρτημα που δυσλειτουργεί. Όμως, έτσι όπως

είναι διαμορφωμένο το σύστημα, υπάρχει περίπτωση να δυσλειτουργεί και ένας ή περισσότεροι αισθητήρες.

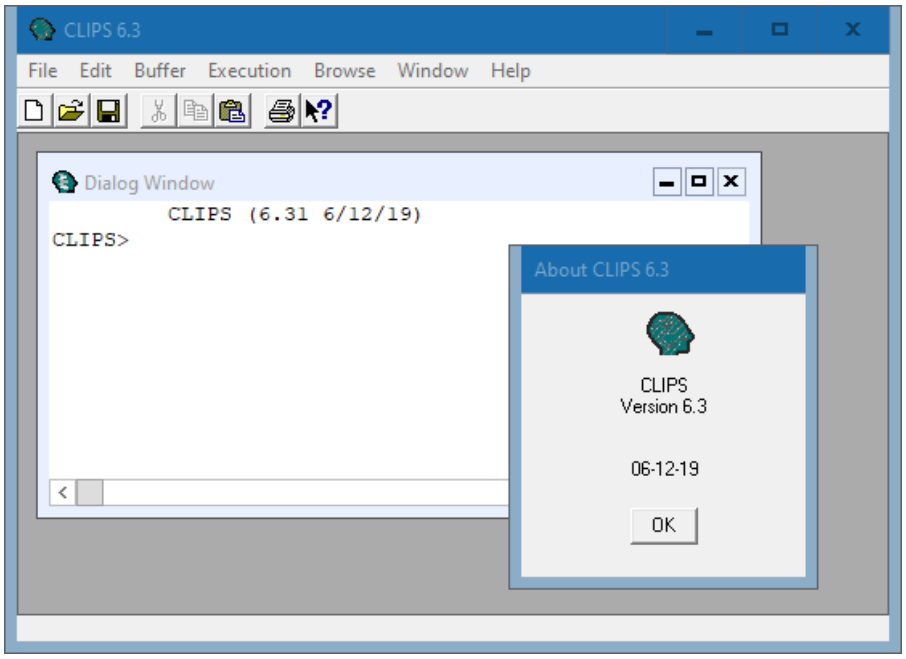
```
;;; Κλάση component. Είναι abstract γιατί περιγράφει από κοινού
;;; τους αισθητήρες και τα εσωτερικά εξαρτήματα. Έχει μία ιδιότητα suspect
;;; που δηλώνει αν το εξάρτημα είναι ύποπτο για τη βλάβη ή όχι.
(defclass component
   (is-a USER)
   (role abstract)
   (slot suspect (type SYMBOL) (allowed-symbols yes no) (default no))
)

;;; Κλάση sensor. Περιγράφει τους αισθητήρες. Κληρονομεί από την κλάση
;;; component χωρίς να προσθέτει άλλες ιδιότητες.
(defclass sensor
   (is-a component)
   (role concrete)
   (pattern-match reactive)
)

;;; Κλάση internal-component. Περιγράφει τα εσωτερικά εξαρτήματα του
;;; συστήματος. Κληρονομεί από την κλάση component, προσθέτοντας την
;;; ιδιότητα connects_with η οποία συνδέει ένα εξάρτημα με άλλα εξαρτήματα
;;; ή αισθητήρες.
(defclass internal-component
   (is-a component)
   (role concrete)
   (pattern-match reactive)
   (multislot connects_with (type INSTANCE-NAME))
)

;;; Ορισμός αντικειμένων. Αναπαριστούν το παράδειγμα μηχανικού συστήματος
;;; που υπάρχει στο κεφάλαιο της συλλογιστικής βασισμένης σε μοντέλα
(definstances sensors-and-internal-components
   (m1 of sensor)
   (m2 of sensor)
   (m3 of sensor)
   (m4 of sensor)
   (m5 of sensor)
   (s14 of internal-component (connects_with [m5]))
   (s21 of internal-component (connects_with [m1] [m2]))
   (s22 of internal-component (connects_with [m2] [m3]))
   (s23 of internal-component (connects_with [m3] [m4]))
   (s11 of internal-component (connects_with [s21] [s22]))
   (s12 of internal-component (connects_with [s22]))
   (s13 of internal-component (connects_with [s23] [m5]))
)

;;; Αρχικός Κανόνας Εισαγωγής Δεδομένων
(defrule initial-question
   ?x <- (initial-fact)
```

```
=>
   (retract ?x)
   (bind $?sensors (find-all-instances ((?s sensor)) TRUE))
   (set-strategy mea)
   (printout t "Which sensors show discrepancy? " $?sensors " ")
   (bind $?answer (explode$ (readline)))
   (assert (discrepancy $?answer))
   (assert (goal make-suspects))
)

;;; Κανόνας Αρχικής Ενοχοποίησης Αισθητήρων
(defrule init-suspects
   (goal make-suspects)
   (discrepancy $? ?s $?)
   (object (is-a sensor)
      (name =(symbol-to-instance-name ?s))
      (suspect no))
=>
   (modify-instance (symbol-to-instance-name ?s) (suspect yes))
)

;;; Κανόνας Ενοχοποίησης Εσωτερικών Εξαρτημάτων
(defrule propagate-suspect
   (goal make-suspects)
   (object (is-a component)
         (name ?c)
         (suspect yes))
   (object (is-a internal-component)
         (name ?c1)
         (connects_with $? ?c $?)
         (suspect no))
=>
   (modify-instance ?c1 (suspect yes))
)

;;; Κανόνας Αλλαγής Στόχου από Ενοχοποίηση σε Αθώωση εξαρτημάτων
(defrule make-suspects-continue
   ?x <- (goal make-suspects)
=>
   (retract ?x)
   (assert (goal exonerate-components))
)

;;; Κανόνας Αθωοποίησης Εξαρτημάτων Λόγω Σύνδεσης με Αθώα Εξαρτήματα
(defrule exonerate-components
   (goal exonerate-components)
   (object (is-a internal-component)
         (name ?c1)
         (suspect yes)
         (connects_with $? ?c $?))
   (object (is-a component)
```

```
         (name ?c)
         (suspect no))
=>
   (modify-instance ?c1 (suspect no))
)

;;; Κανόνας Αθωοποίησης Εξαρτημάτων Λόγω μη-Εξήγησης Ασυμφωνίας
(defrule exonerate-components-special
   (goal exonerate-components)
   (object (is-a internal-component)
         (name ?c)
         (suspect yes)
         (connects_with $?sensors))
   (not (object (is-a sensor)
               (name ?s&:(member$ ?s $?sensors))
               (suspect no)))
   (object (is-a sensor)
         (name ?s1&:(not (member$ ?s1 $?sensors)))
         (suspect yes))
=>
   (modify-instance ?c (suspect no))
)

;;; Κανόνας Αλλαγής Στόχου από Αθωοποίηση Εξαρτημάτων
;;; σε Ανακοίνωση Αποτελεσμάτων
(defrule exonerate-components-cont
   ?x <- (goal exonerate-components)
=>
   (retract ?x)
   (assert (goal announce-result))
)

;;; Κανόνας Ανακοίνωσης Χαλασμένου Εξαρτήματος
(defrule announce-result
   (goal announce-result)
   (object (is-a component) (name ?c)
         (suspect yes))
   (not (object (is-a internal-component)
               (name ~?c)
               (suspect yes)
               (connects_with $? ?c $?)))
=>
   (printout t (class ?c) " "
         (instance-name-to-symbol ?c)
         " malfunctions!" crlf)
)
```

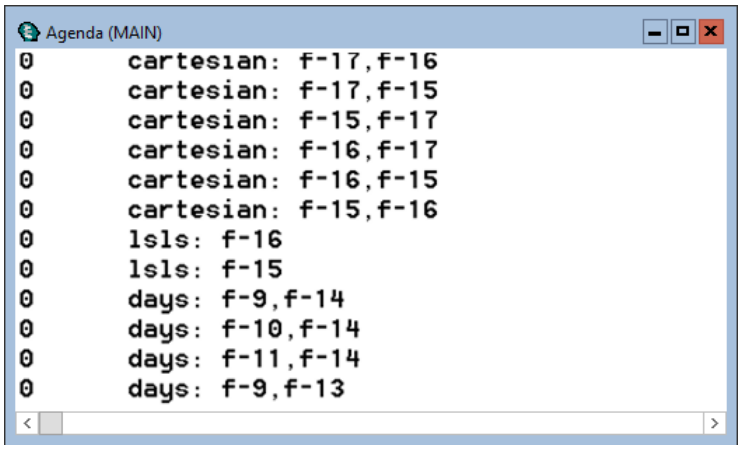
Π1.10 Το Περιβάλλον του CLIPS

Σε λειτουργικό σύστημα MS-Windows, το περιβάλλον του CLIPS εκκινεί με την εκτέλεση του αρχείου **CLIPSWIN.EXE**. Αμέσως μετά εμφανίζεται στην οθόνη το "παραθυρικό" περιβάλλον της γλώσσας, όπως φαίνεται στο Σχήμα Π2.2. Για να φορτωθεί ένα πρόγραμμα CLIPS επιλέγεται από το μενού *File* η επιλογή *Load Constructs* και μέσα από το σχετικό παράθυρο επιλέγεται το επιθυμητό αρχείο.



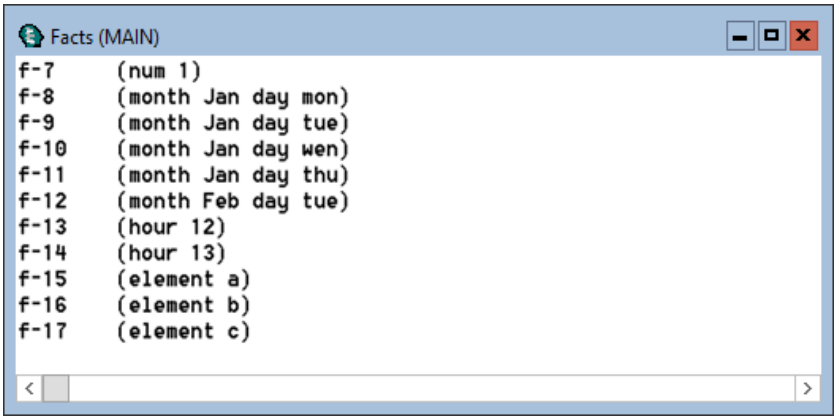
Σχήμα Π2.2: Το περιβάλλον CLIPS.

Από τα διάφορα παράθυρα που εμφανίζονται μέσω των μενού του CLIPS δύο είναι σημαντικά: το ένα εμφανίζει κάθε στιγμή το περιεχόμενο της λίστας γεγονότων και το άλλο το περιεχόμενο της agenda. Τα παράθυρα αυτά εμφανίζονται από τις επιλογές *Window→Facts Window* και *Window→Agenda Window* αντίστοιχα.



Σχήμα Π2.3: Το παράθυρο της ατζέντας.

Όπως φαίνεται στο Σχήμα Π2.3, στο παράθυρο της ατζέντας εμφανίζεται σα λίστα το σύνολο των κανόνων που ενεργοποιούνται. Μαζί με το όνομα του κανόνα εμφανίζονται και τα **fact-index** των γεγονότων τα οποία τον ικανοποιούν. Αντίστοιχα στο παράθυρο των γεγονότων (Σχήμα Π2.4) εμφανίζονται τα γεγονότα μαζί με το αντίστοιχο **fact index** τους.



Σχήμα Π2.4: Το παράθυρο της λίστας γεγονότων.

Για να εκτελεστεί κάποιο πρόγραμμα γραμμένο σε CLIPS πρέπει να πληκτρολογηθεί σε πρώτη φάση η εντολή **(reset)**. Η εντολή αυτή καταχωρεί στη μνήμη όλα τα γεγονότα τα οποία περιγράφονται στο αρχείο που φορτώθηκε (βλ. εντολή **deffacts**, **defrule**, κτλ). Για να ξεκινήσει η εκτέλεση του κύκλου λειτουργίας πρέπει να πληκτρολογηθεί η εντολή **(run)**.

Βιβλιογραφία

Ένα αρκετά καλό βιβλίο που αφορά τόσο τη θεωρία των συστημάτων γνώστης όσο και το σύστημα παραγωγής CLIPS είναι το [Giarratano & Riley, 2004]. Θα πρέπει επίσης να σημειωθεί ότι το εγχειρίδιο χρήσης που συνοδεύει το πρόγραμμα είναι αρκετά κατατοπιστικό [Riley, 2019]. Περισσότερες πληροφορίες, ιστορικά στοιχεία, παραδείγματα γραμμένα σε CLIPS, καθώς και το ίδιο το σύστημα CLIPS είναι διαθέσιμα στη διεύθυνση http://clipsrules.net/.

Αναφορές

[Giarratano & Riley, 2004] J. Giarratano and G. Riley, *Expert Systems*: *Principles and Programming*, 4th edition, Course Technology, 2004.

[Riley, 2019] G. Riley, *Clips Basic Programming Guide*, Version 6.31, 2019.