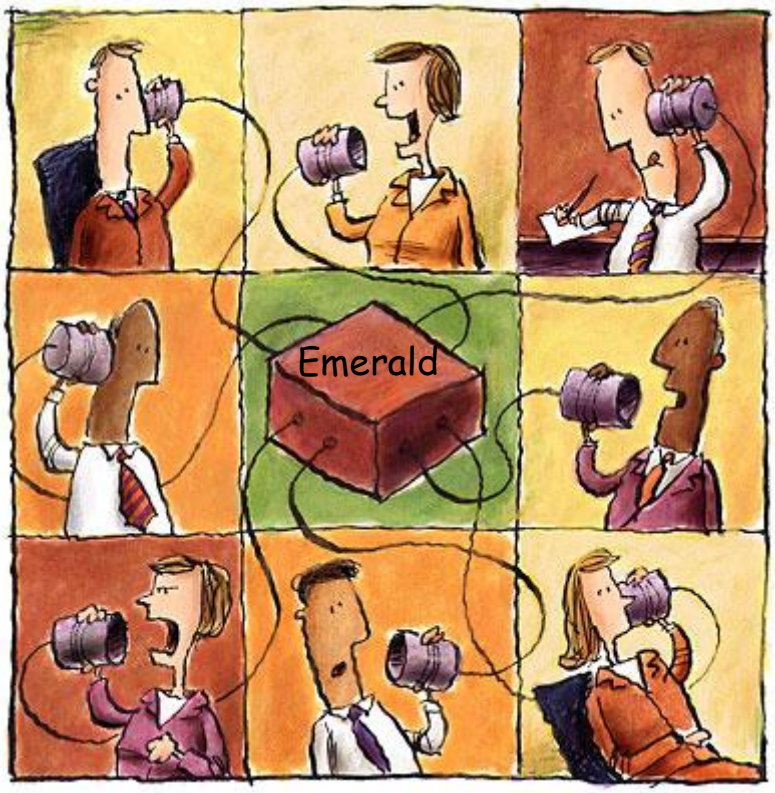
Extended Emerald

A Knowledge-based Framework for Semantic Web Agents



ΟΜΑΔΑ 1

Ζαμπετάκης Σταμάτης
zabetak@csd.uoc.gr

Ζωγραφιστού Δήμητρα
dzograf@csd.uoc.gr

Κλεισαρχάκη Σοφία
kleisar@csd.uoc.gr

Κονσολάκη Κωνσταντίνα
konsolak@csd.uoc.gr

Κουτράκη Μαρία
kutraki@csd.uoc.gr

Μαθιουδάκης Γιώργος
gmathiou@csd.uoc.gr

Μαυρίδου Ευανθία
mavridou@csd.uoc.gr

Μπουλουκάκης Γιώργος
boulouk@csd.uoc.gr

Παπατριανταφύλλου Άγγελος
angelpap@csd.uoc.gr

Συμεονίδου Δανάη
simeon@csd.uoc.gr

Τρουλλινού Γεωργία
troulin@csd.uoc.gr

Φαφαλιός Παύλος
fafalios@csd.uoc.gr

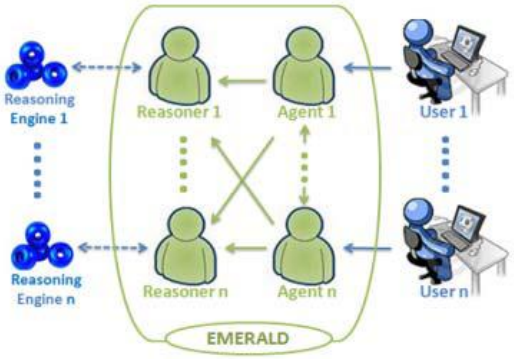
Περιεχόμενα

1. Εισαγωγή

Η ανάπτυξη του Παγκόσμιου Ιστού (World Wide Web) κατέστησε το διαδίκτυο προσβάσιμο σε εκατομμύρια χρήστες, επιτρέποντας την απρόσκοπτη δημοσιοποίηση και πρόσβαση σε έγγραφα στο διαδίκτυο. Η εκρηκτική ανάπτυξη του Παγκόσμιου Ιστού δημιούργησε προβλήματα «πληροφοριακής υπερφόρτισης». Η παγκόσμια ερευνητική κοινότητα έχει στραφεί εδώ και λίγα χρόνια σε μία νέα κατεύθυνση εξέλιξης του ιστού, η οποία ονομάζεται «Σημασιολογικός Ιστός» (Semantic Web) και περιλαμβάνει τη σαφή αναπαράσταση του νοήματος των πληροφοριών και των εγγράφων, επιτρέποντας την αυτόματη επεξεργασία και ενοποίηση διαδικτυακών πόρων από «έξυπνα» προγράμματα-πράκτορες. Ο Σημασιολογικός Ιστός θα επιτρέψει τον γρήγορο και ακριβή εντοπισμό πληροφοριών στον παγκόσμιο ιστό καθώς και την ανάπτυξη ευφυών διαδικτυακών πρακτόρων οι οποίοι θα διευκολύνουν την επικοινωνία μεταξύ πληθώρας ετερογενών ηλεκτρονικών συσκευών με πρόσβαση στο διαδίκτυο.

1.1 Εισαγωγή στο Emerald

Στην περιοχή του semantic web και πιο συγκεκριμένα στα πλαίσια της επικοινωνίας ετερογενών πρακτόρων, οι οποίοι εφαρμόζουν διαφορετικά πρότυπα και λογική, δίνεται η δυνατότητα στους πράκτορες να ανταλλάσουν πληροφορίες ώστε να περιγράψουν και να αιτιολογήσουν την θέση τους σε συγκεκριμένα ερωτήματα. Μια πλατφόρμα που υλοποιεί αυτή την ιδέα είναι το *emerald*.



Πιο συγκεκριμένα, τα δομικά συστατικά αυτής της εφαρμογής είναι ο πράκτορας (agent) και ο *DR-Reasoner*.

Κάθε πράκτορας διαθέτει ένα πλήθος χαρακτηριστικών. Αρχικά, εμπεριέχει μια βάση γνώσης η οποία περιγράφει όλη τη διαθέσιμη γνώση. Επίσης, η περιγραφή της γνώσης για το περιβάλλον, το πρότυπο συμπεριφοράς όπως επίσης και την στρατηγική/πολιτική επιτυγχάνεται μέσω της βάσης κανόνων (rule bases).

Το δεύτερο δομικό συστατικό, ο *reasoner*, είναι μια ανεξάρτητη τρίτη υπηρεσία που έχει την ιδιότητα, λαμβάνοντας τη βάση γνώσης και τη defeasible logic βάση κανόνων από τον πράκτορα, να την επεξεργάζεται και να εξάγει αποτελέσματα στη μορφή RDF αρχείων.

Πιο αναλυτικά, ρόλος του *reasoner* είναι να "ακούει" διαρκώς για καινούρια μηνύματα (ACL μηνύματα) και να φροντίζει ώστε να επικοινωνήσει με την κατάλληλη μηχανή και να ενημερώσει για τα τελικά αποτελέσματα. Ουσιαστικά, αποτελεί το προφίλ της reasoning μηχανής στην οποία θα εκτελεστεί η επερώτηση του χρήστη. Συνεπώς, για την επικοινωνία του πράκτορα με τον *reasoner* ανταλλάσσονται δυο κατηγορίες ACL μηνυμάτων, τα REQUEST και INFORM αντίστοιχα.

Μέχρι τώρα το *Emerald* υποστηρίζει τέσσερα reasoning engines, τα οποία χρησιμοποιούν ένα πλήθος από λογικές. Συγκεκριμένα:

- **DR-DEVICE**: (defeasible reasoning)
- **R-DEVICE**: (deductive, Datalog-like rules)
- **SPINdle**:(defeasible logic engine)
- **Prova**: (prolog-like rule engine)

2. Επεκτάσεις & Υλοποίηση

Σκοπός της παρούσας δουλειάς είναι να επεκτείνει το device reasoning, με κατάλληλες τροποποιήσεις, ώστε πλέον να υποστηρίζει *DR-Prolog* επεκτείνοντας ταυτόχρονα τη λογική και τις δυνατότητες των πρακτόρων. Οι αρμοδιότητες της δικής μας ομάδας είναι οι εξής:

- Κατασκευή προτύπου για τα *ruleML* αρχεία.
- Δημιουργία βάση του παραπάνω προτύπου των αρχείων όπου εκφράζονται οι προτιμήσεις και οι απαιτήσεις του «πελάτη», στο παράδειγμα broker – customer.
- Κατασκευή αναλυτή (parser) με σκοπό την μετατροπή των *ruleML* αρχείων σε *DR-Prolog* σύνταξη.
- Κατασκευή αναλυτή (parser) με σκοπό τη μετατροπή της βάσης γνώσης από *RDF* σε Prolog facts.
- Κατασκευή αναλυτή (parser) με σκοπό τη μετατροπή της λίστας των τελικών αποτελεσμάτων που προκύπτουν από την επερώτηση σε RDF μορφή.
- Επέκταση του metaprogram ώστε να υποστηρίζει αριθμητικές παραστάσεις, συγκρίσεις και τον κανόνα *naf*.
- Δημιουργία μηχανής για την εκτέλεση ερωτημάτων και την επιστροφή αποτελεσμάτων με βάση τις προτιμήσεις του «πελάτη».

Παρακάτω, περιγράφονται λεπτομέρειες τόσο της λειτουργικότητας όσο και της υλοποίησης αυτών των θεμάτων.

2.1 Πρότυπο RuleML

Οι κανόνες που εκφράζουν τις απαιτήσεις (*carlo_1.ruleml*) και προτιμήσεις (*carlo_2.ruleml*) του πελάτη εκφράστηκαν με ένα νέο πρότυπο που δημιουργήσαμε το οποίο είναι επέκταση της RuleML σύνταξης. Για τη δημιουργία του προτύπου βασιστήκαμε στο υπάρχον πρότυπο RuleML που χρησιμοποιεί η DR-Prolog και εν μέρη στα RuleML αρχεία από το Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης. Η σύνταξη που επιλέξαμε για το πρότυπο είναι τέτοια ώστε να είναι εφαρμόσιμη στην DR-Prolog, εύκολα αναγνώσιμη και επεκτάσιμη. Το σχήμα του προτύπου (DTD) επισυνάπτεται στο φάκελο της εργασίας (*dr-prolog.dtd*).

2.2 Μετατροπή ruleML σε DR-Prolog: RuleMLParser

Για την εξαγωγή των κανόνων DR-Prolog, δημιουργήθηκε ο μετατροπέας RuleMLparser. Σκοπός του είναι από τα RuleML αρχεία να εξάγει τους DR-Prolog κανόνες. Ο μετατροπέας έχει γραφτεί εξ' ολοκλήρου στη γλώσσα Java και αποτελείται από δύο κλάσεις, την κεντρική κλάση RuleMLparser.java και την βοηθητική κλάση Tagger.java. Δέχεται σαν είσοδο ένα αρχείο RuleML και δημιουργεί ένα νέο αρχείο .P με τους κανόνες εκφρασμένους σε DR-Prolog σύνταξη.

Η κλάση RuleMLparser.java περιέχει τέσσερις στατικές μεθόδους. Η πρώτη μέθοδος είναι η "void createRules(File ruleml)". Αυτή η μέθοδος διαβάσει το RuleML αρχείο και δημιουργεί τους κανόνες. Πιο αναλυτικά, χωρίζει το αρχείο σε διακριτά μέρη έτσι ώστε να ξεχωρίσει τους κανόνες. Στη συνέχεια, χωρίζει κάθε κανόνα σε επιμέρους τμήματα και καλεί τις κατάλληλες συναρτήσεις για να τα επεξεργαστούν. Κάθε επιμέρους συνάρτηση δέχεται ένα από τα παραπάνω τμήματα και παράγει το αντίστοιχο μέρος του κάθε κανόνα. Στο τέλος, όλα τα τμήματα ενώνονται,

σχηματίζοντας έτσι τους τελικούς κανόνες. Για παράδειγμα, από το παρακάτω κομμάτι RuleML σύνταξης (εικόνα 1), παράγεται ο κανόνας της εικόνας 2.

```
<Implies ruletype="defeasiblerule">
   <oid>r1</oid>
   <head>
      <Atom neg="no">
         <Rel>acceptable</Rel>
         <Slot type="var">X</Slot>
         <Slot type="var">Y</Slot>
         <Slot type="var">Z</Slot>
         <Slot type="var">W</Slot>
      </Atom>
</head>
   <body>
      <part type="atom">
         <Rel>name</Rel>
         <Slot type="var">X</Slot>
         <Slot type="var">X</Slot>
      </part>
         <part type="atom">
         <Rel>price</Rel>
         <Slot type="var">X</Slot>
         <Slot type="var">Y</Slot>
      </part>
         <part type="atom">
         <Rel>size</Rel>
         <Slot type="var">X</Slot>
         <Slot type="var">Z</Slot>
      </part>
         <part type="atom">
          <Rel>gardenSize</Rel>
          <Slot type="var">X</Slot>
          <Slot type="var">W</Slot>
      </part>
       </body>
</Implies>
```

Εικόνα 1. Τμήμα ruleML αρχείου

```
defeasible(r1,acceptable(X,Y,Z,W),[name(X,X),price(X,Y),size(X,Z),gardenSize(X,W)]).
```

Εικόνα 2. Κανόνας σε DR-Prolog σύνταξη

Η δεύτερη μέθοδος είναι η "String getQuery(File ruleml)". Αυτή η μέθοδος βρίσκει το ερώτημα (Query) στο RuleML, το μετατρέπει σε DR-Prolog σύνταξη και επιστρέφει ένα αλφαριθμητικό που το αναπαριστά (εικόνα 3).

```
defeasibly(acceptable(X,Y,Z,W))
%(apartment, price, size, gardenSize)
```

Εικόνα 3. Ερώτημα σε DR-Prolog σύνταξη

Η τρίτη μέθοδος είναι η "*String getQueryMode(File ruleml)*". Αυτή η μέθοδος επιστρέφει τον τύπο του ερωτήματος. Ο τύπος μπορεί να είναι είτε answer είτε proof.

Η τελευταία μέθοδος είναι η "*String getQueryVars(File ruleml)*". Η μέθοδος αυτή επιστρέφει τις μεταβλητές του ερωτήματος. Σκοπός της είναι η εύρεση των μεταβλητών του ερωτήματος για την κατασκευή της δομής της επιστρεφόμενης λίστας.

Η κλάση "Tagger.java" είναι μία βοηθητική κλάση για την ανάγνωση XML κώδικα. Περιέχει ένα σύνολο από συναρτήσεις που βρίσκουν ετικέτες και εξάγουν το περιεχόμενο τους.

Για παράδειγμα, η συνάρτηση "*String getFirstTagData(String theTag)*", δέχεται ένα όνομα ετικέτας, πχ "Head" και επιστρέφει ένα αλφαριθμητικό με το περιεχόμενο αυτής της ετικέτας.

2.3 Μετατροπή RDF σε DR-Prolop: RdfParser

Στα πλαίσια της ενσωμάτωσης της DR-Prolog στη μηχανή του Emerald, κρίθηκε απαραίτητη η μετατροπή της βάσης γνώσης που χρησιμοποιεί, από τη μορφή RDF στην οποία βρίσκεται, σε Prolog facts. Για το σκοπό αυτό υλοποιήθηκε η κλάση RdfParser. Η κλάση αυτή είναι γραμμένη εξ' ολοκλήρου σε java και πραγματοποιεί αυτή τη μετατροπή. Συγκεκριμένα, η μέθοδος *toProLog* δέχεται ως είσοδο το rdf αρχείο της βάσης γνώσης και το όνομα του παραγόμενου prolog αρχείου. Κάνοντας χρήση του μοντέλου **S**emantic **W**eb **K**nowledge **M**iddleware ανακτάται η πληροφορία υπό τη μορφή τριπλετών και στη συνέχεια, μέσω της μεθόδου toString γράφεται σε Prolog facts. Ένα παράδειγμα που περιγράφει την μετατροπή αυτή φαίνεται παρακάτω. Στην πρώτη εικόνα παρουσιάζεται η αρχική μορφή της βάσης γνώσης ενώ στη δεύτερη η παραγόμενη Prolog μορφή:

```
<tour:Hotel rdf:ID="CretaMareRoyal">
   <tour:resortID>1</tour:resortID>
   <tour:hotelName>Creta Mare
Royal</tour:hotelName>
   <tour:hotelStars>6</tour:hotelStars>

<tour:hotelCategory>Business</tour:hotelCa
tegory>
   <tour:parking>true</tour:parking>
```

Εικόνα 4. RDF μορφή

```
fact(type(CretaMareRoyal,Hotel)).
fact(resortID(CretaMareRoyal,1)).
fact(hotelName(CretaMareRoyal,Creta Mare
Royal)).
fact(hotelStars(CretaMareRoyal,6)).
fact(hotelCategory(CretaMareRoyal,Business
)).
fact(parking(CretaMareRoyal,true)).
fact(swimmingPool(CretaMareRoyal,true)).
fact(breakfast(CretaMareRoyal,true)).
```

Εικόνα 5. DR-Prolog facts

```
public static void toProlog (String fromFile, String toFile)
```

Εικόνα 6. Υπογραφή Μεθόδου

2.4 Μετατροπή αποτελεσμάτων σε RDF: ResultParser

Ένας από τους στόχους της εργασίας ήταν, παρά την ενσωμάτωση της DR-Prolog στο Emerald, να μην επηρεαστεί η αλληλεπίδραση με το χρήστη όσον αφορά τη μορφή των μηνυμάτων που ανταλλάσσονται μεταξύ τους. Για το σκοπό αυτό κρίθηκε αναγκαία η δημιουργία ενός μηχανισμού ο οποίος θα μετατρέπει το αποτέλεσμα του query, το οποίο επιστρέφεται υπό μορφή λίστας Prolog, σε μορφή *rdf*. Δημιουργήσαμε έτσι την κλάση *ResultParser*, η οποία είναι υπεύθυνη γι αυτή τη μετατροπή. Συγκεκριμένα, η μέθοδος *toRDF* δέχεται ως ορίσματα το query, το όνομα του και τη λίστα των αποτελεσμάτων και επιστρέφει ένα αρχείο *rdf* με τα αποτελέσματα του query. Το τελικό rdf δεν περιέχει πλέον όλη την πληροφορία που διατηρείται στη βάση για τα επιστρεφόμενα αποτελέσματα, παρά μόνο αυτήν που ζητείται μέσω του query. Για το λόγο αυτό χρειάζονται οι μεταβλητές του query, ούτως ώστε να μπορεί να γίνει η αντιστοίχηση των τιμών των αποτελεσμάτων με αυτές. Στο παράδειγμα που ακολουθεί περιγράφεται η παραπάνω λειτουργικότητα:

```
nonDeterministicGoal(X, defeasibly(acceptable(X,Y,Z,W)), ListModel)
```

Εικόνα 7. Query

```
Results = [a3,350,65,0,a5,350,55,15]
```

Εικόνα 8. Λίστα αποτελεσμάτων σε Prolog

```
public void toRDF(String query, String queryName, String result, String fileName)
```

Εικόνα 9 Υπογραφή Μεθόδου

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [<!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-
ns#'>
<!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
<!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>
<!ENTITY dr-device 'file:/c:/jade/EMERALD/conclusions/projectvo58.rdf#'>
]>
<rdf:RDF
      xmlns:dr-
device="file:/c:/jade/EMERALD/conclusions/projectvo58.rdf#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
      <rdfs:Class rdf:about="acceptable">
            <rdfs:label>acceptable</rdfs:label>
      </rdfs:Class>
      <rdf:Property rdf:about="#X">
            <rdfs:domain rdf:resource="#acceptable"/>
      </rdf:Property>
      <rdf:Property rdf:about="#Y">
            <rdfs:domain rdf:resource="#acceptable"/>
      </rdf:Property>
      <rdf:Property rdf:about="#Z">
            <rdfs:domain rdf:resource="#acceptable"/>
      </rdf:Property>
      <rdf:Property rdf:about="#W">
            <rdfs:domain rdf:resource="#acceptable"/>
      </rdf:Property>
      <dr-device:acceptable rdf:about="#acceptable0">
            <dr-device:X>a3</dr-device:X>
            <dr-device:Y>350</dr-device:Y>
            <dr-device:Z>65</dr-device:Z>
            <dr-device:W>0</dr-device:W>
      </dr-device:acceptable>
      <dr-device:acceptable rdf:about="#acceptable1">
            <dr-device:X>a5</dr-device:X>
            <dr-device:Y>350</dr-device:Y>
            <dr-device:Z>55</dr-device:Z>
            <dr-device:W>15</dr-device:W>
      </dr-device:acceptable>
</rdf:RDF>
```

Εικόνα 10. Τελικό αποτέλεσμα σε RDF

2.5 Metaprogram

Η υπάρχουσα έκδοση του metaprogram δεν υποστήριζε κάποιες λειτουργίες που ήταν απαραίτητες για την εκτέλεση των ερωτημάτων. Γι' αυτό το λόγο χρειάστηκε να αναβαθμιστεί ώστε να υποστηρίζει τις παρακάτω λειτουργίες:

- Σύγκριση αριθμών και υπολογισμός αριθμητικών παραστάσεων
- Υποστήριξη του κανόνα naf

2.5.1 Σύγκριση αριθμών και υπολογισμός αριθμητικών παραστάσεων

Για την σύγκριση αριθμών εισάγαμε τους παρακάτω 5 κανόνες:

```
definitely(X<Y):- compute([X,+,0],K), compute([Y,+,0],M), K @< M.
definitely(X>Y):- compute([X,+,0],K), compute([Y,+,0],M), K @> M.
definitely(X>=Y):- compute([X,+,0],K), compute([Y,+,0],M), K @>= M.
definitely(X=<Y):- compute([X,+,0],K), compute([Y,+,0],M), K @=< M.
definitely(X\==Y):- X \== Y.
```

Ο πρώτος κανόνας ελέγχει αν το X είναι μικρότερο του Y. Ο δεύτερος αν ο X είναι μεγαλύτερος του Y. Ο τρίτος αν ο X είναι μεγαλύτερος ή ίσος του Y. Ο τέταρτος αν ο X είναι μικρότερος ή ίσος του Y και ο τελευταίος αν ο X είναι διαφορετικός από το Y. Καθώς μπαίνουμε στο κάθε κανόνα καλείται ο κανόνας compute. Η παρουσία του κανόνα compute είναι απαραίτητη και ο λόγος είναι ότι θέλουμε ταυτόχρονα να συγκρίνουμε και αριθμούς αλλά και αριθμητικές παραστάσεις. Ο λόγος για τον οποίο καλούμε τη compute με λίστα είναι ότι οι αριθμητικές παραστάσεις πρέπει να έχουν συγκεκριμένη μορφή. Για παράδειγμα μια απλή αριθμητική παράσταση είναι η [100,+,200] η οποία εκφράζει την πρόσθεση δυο αριθμών, του 100 με το 200.

Γενικώς, οι αριθμητικές παραστάσεις θα πρέπει να εκφράζονται σε μορφή λίστας και η κάθε λίστα εμφωλευμένη ή μη θα πρέπει να έχει τρία μέλη: αριστερό μέλος(αριθμός ή λίστα), δεξί μέλος(αριθμός ή λίστα) και μεσαίο μέλος(τελεστής πράξης). Για παράδειγμα αν θέλαμε να κάνουμε τη πράξη του πολλαπλασιασμού δυο αριθμών όπου ο δεύτερος αριθμός προκύπτει από τη πρόσθεση άλλων δυο (θέλουμε να κρατήσουμε τη προτεραιότητα των πράξεων), τότε προκύπτει η εξής έκφραση: [[10,+,20], *, 5]. Στη περίπτωση που έχουμε μόνο αριθμούς να συγκρίνουμε τότε καλείται πάλι η compute. Επειδή όμως θα αντιμετωπίζαμε πρόβλημα από τη στιγμή που η compute δέχεται μόνο λίστες, έτσι αναγκαστήκαμε να μετατρέψουμε τον αριθμό σε λίστα με έναν πολύ απλό τρόπο: [X,+,0], το οποίο θα επιστρέφει πάντα X. Με αυτό το τρόπο παρακάμπτουμε την compute.

Η υλοποίηση της compute είναι απλή και βασίζεται στη λογική της επίσκεψης κάθε κόμβου ενός δέντρου. Από τη στιγμή που μια αριθμητική παράσταση μπορεί να αποτελείται από πολλά επίπεδα εμφωλευμένων πράξεων – λιστών είναι σαν να δουλεύουμε πάνω σε κάποιο δέντρο. Ο υπολογισμός των πράξεων ξεκινάει από χαμηλά, από τα φύλλα του δέντρου και καταλήγει στη ρίζα όπου και επιστρέφεται το τελικό αποτέλεσμα. Παρακάτω παρατίθεται η υλοποίηση της:

```
compute([H1, H2, H3|_], K2) :- compute(H1, K), compute([K, H2, H3], K2).
compute([X, Y, Z|_], K) :- number(X), number(Z), Y == *, K is X * Z.
compute([X, Y, Z|_], K) :- number(X), number(Z), Y == /, K is X / Z.
compute([X, Y, Z|_], K) :- number(X), number(Z), Y == +, K is X + Z.
compute([X, Y, Z|_], K) :- number(X), number(Z), Y == -, K is X - Z.
compute([H1, H2, H3|_], K2) :- compute(H3, K), compute([H1, H2, K], K2).
```

2.5.2 Κανόνας naf

Η τελευταία αναβάθμιση που έγινε ήταν η εισαγωγή του κανόνα naf:

```
definitely(naf(X,Z,M)):- not(naf(X,Z,M)).
```

Ο παραπάνω κανόνας χρειάστηκε στο παράδειγμά μας για τις προτιμήσεις του πελάτη.

2.6 Μηχανή

Για τον έλεγχο των λειτουργιών που υλοποιήσαμε κατασκευάσαμε μία μηχανή η οποία αρχικά φορτώνει τις απαιτήσεις και προτιμήσεις του πελάτη (ruleML αρχεία), τη βάση γνώσης (facts.p) και τέλος το metaprogram.

Στη συνέχεια καλείται ο αναλυτής ο οποίος παράγει τους κανόνες, τους αποθηκεύει σε ξεχωριστό αρχείο και ταυτόχρονα τους φορτώνει στη μηχανή. Με συναρτήσεις του αναλυτή, εξάγουμε το ερώτημα, τον τύπο και τις μεταβλητές του καθώς και το τι αντιπροσωπεύει η κάθε μεταβλητή. Για παράδειγμα, η μεταβλητή X του σχήματος 3 αντιπροσωπεύει το όνομα του διαμερίσματος.

Τέλος, σαν αποτέλεσμα επιστρέφεται μία λίστα η οποία έχει μορφή ανάλογη των μεταβλητών του ερωτήματος. Για παράδειγμα, για το ερώτημα του σχήματος 3, η απάντηση είναι:

```
result=[a3,350,65,0,a5,350,55,15,a7,375,65,12]
```

Το παραπάνω αποτέλεσμα αποθηκεύεται σε ένα αρχείο με όνομα result.txt.

3. Παραδοτέα

1. carlo_1.ruleml
2. carlo_2.ruleml
3. dr-prolog.dtd
4. amb_metaprogram_add.P
5. RdfParser.java
6. ResultArray.java
7. ResultParser.java
8. ResultType.java
9. Engine.java
10. RuleMLparser.java
11. Tagger.java