ΑΝΩΤΑΤΟ
ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ
ΚΡΗΤΗΣ

Σχολή Τεχνολογικών Εφαρμογών

Τμήμα Ηλεκτρολογίας

Μετασχηματισμός Λογικών Προγραμμάτων

Σαντιπαντάκης Γιώργος

Πτυχιακή Εργασία

Ηράκλειο, Οκτώβριος 2002

Τεχνολογικό Εκπαιδευτικό Ίδρυμα
Ηρακλείου
Σχολή Τεχνολογικών Εφαρμογών
Τμήμα Ηλεκτρολογίας

Μετασχηματισμός Λογικών Προγραμμάτων

Εργασία που υποβλήθηκε από τον
Σαντιπαντάκη Γεώργιο
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
πτυχίου στο τμήμα Ηλεκτρολογίας

Συγγραφέας:

____
Σαντιπαντάκης Γεώργιος
Τμήμα Ηλεκτρολογίας
ΤΕΙ Κρήτης

Εισηγητής:

____
Δρ. Μαρακάκης Εμμανουήλ
Επικ. Καθηγητής
ΤΕΙ Κρήτης

Ηράκλειο, Οκτώβριος 2002

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή 4
2. Θεωρητικό υπόβαθρο 5
  2.1. Λογική 5
  2.2. Λογική «πρώτης τάξης» (First order theory) 7
  2.3. Λογικός Προγραμματισμός και Prolog 8
    2.3.1. Έρευνα 11
    2.3.2. Αναδρομές 12
    2.3.3. Λίστες 15
  2.5. Μετά-προγραμματισμός 17
3. Μετασχηματισμός προγραμμάτων 19
4. Πρόγραμμα αντικείμενο σε Βασικούς Όρους και αλγόριθμοι χειρισμού του 22
    4.1. Αναπαράσταση 22
    4.2. Αντικατάσταση (substitution) 24
    4.3. Αλγόριθμος εφαρμογής αντικατάστασης (apply substitution) 25
    4.4. Σύνθεση αντικαταστάσεων (composition of substitutions) 27
    4.5. Αλγόριθμος ταυτοποίησης (Unify) 28
    4.6. Αλγόριθμος μετονομασίας μεταβλητών (rename) 33
5. Μετασχηματισμοί 35
    5.1. Νέος Ορισμός (New Definition) 35
    5.2. Κανόνας μετασχηματισμού ανάπτυξης (Unfold) 37
    5.3. Κανόνας μετασχηματισμού πτύξης (Folding) 40
6. Υλοποίηση συστήματος 42
7. Εφαρμογή του συστήματος – Ένα παράδειγμα 47
8. Συμπεράσματα – Μελλοντική επέκταση του συστήματος 62
9. Βιβλιογραφία 63
Παράρτημα 64
  Α. Πηγαίος Κώδικας του συστήματος σε Sicstus Prolog 64
- Kernel.pl 64
- Rename.pl 88
- Unify.pl 96
- Apply.pl 100
- Unfold.pl 101
- Fold.pl 105

1. Εισαγωγή

Οι μετασχηματισμοί λογικών προγραμμάτων είναι μια πολύ σημαντική μεθοδολογία για την ανάπτυξη λογισμικού.

Η βασική ιδέα έχει ως εξής:

Αρχικά, υλοποιούμε ένα πρόγραμμα το οποίο μπορεί να κατασκευαστεί και να συντηρηθεί εύκολα, όμως δεν είναι αποτελεσματικό. Δηλαδή, κατασκευάζουμε το πρόγραμμα χωρίς να λαμβάνουμε υπόψην την αποτελεσματικότητα του. Ένα τέτοιο πρόγραμμα, θα έχει δίχως άλλο ευδιάκριτη δομή, δεν θα έχει όμως την μέγιστη δυνατή αποτελεσματικότητα. Στην συνέχεια, εφαρμόζουμε μετασχηματισμούς στο παραπάνω πρόγραμμα, προκειμένου να γίνει αποτελεσματικότερο. Συνήθως, ένα αποτελεσματικό πρόγραμμα έχει δυσανάγνωστη δομή και κατά συνέπεια είναι δύσκολες οι όποιες αλλαγές.

Το παρόν σύγγραμμα, αποσκοπεί στην μελέτη και ανάπτυξη ορισμένων τεχνικών μετασχηματισμού, οι οποίες εφόσον εφαρμοστούν σε ένα πρόγραμμα, θα το μετατρέψουν σε περισσότερο αποτελεσματικό. Οι τεχνικές μετασχηματισμού λογικών προγραμμάτων που θα μελετηθούν και θα αναπτυχθούν παρακάτω, είναι οι εξής:

Folding - Πτύξη
Unfolding - Ανάπτυξη
New definition – Νέος ορισμός

Για τις παραπάνω τεχνικές, θα μελετηθεί ο τρόπος αναπαράστασης του προγράμματος, σε βασική αναπαράσταση (Ground representation).

Επιπλέον, θα αναπτυχθούν αλγόριθμοι όπως :

Renaming – Μετονομασία μεταβλητών
Substitution - Αντικατάσταση
Unify - Ταυτοποίηση
Apply - Εφαρμογή αντικαταστάσεων
Composition of substitutions - Σύνθεση αντικαταστάσεων

2. Θεωρητικό υπόβαθρο

Όταν ο Robinson παρουσίασε τον κανόνα επίλυσης (resolution rule) το 1965, η αρχή για τον λογικό προγραμματισμό είχε γίνει. Το 1972, ο Kowalski και ο Colmerauer έδειξαν ότι η λογική μπορεί να χρησιμοποιηθεί ως γλώσσα προγραμματισμού. Η ιδέα αυτή ήταν επαναστατική επειδή μέχρι το 1972, η λογική είχε χρησιμοποιηθεί μόνο ως επεξηγηματική γλώσσα στην επιστήμη των υπολογιστών. Η εκτέλεση ενός λογικού προγράμματος, είναι η προσπάθεια απόδειξης του στόχου με δεδομένα τα αξιώματα στο λογικό πρόγραμμα. Δηλαδή, η ιδέα αυτή μπορεί να τυποποιηθεί ως:

Ένα πρόγραμμα είναι ένα σύνολο αξιωμάτων (first order theory)

και

Ο υπολογισμός είναι μια (κατασκευασμένη) απόδειξη του στόχου από το πρόγραμμα (άρα και συμπέρασμα από την θεωρία).

Ο Roussel υλοποίησε τον πρώτο interpreter για την Prolog, στη γλώσσα Algol-W, στη Marseille το 1972. Η λέξη Prolog, είναι αρκτικόλεξο και σημαίνει PROgrammation en LOGique, το οποίο ενσωματώνει την διαδικαστική ερμηνεία λογικής του Kowalski, και αντανακλά τον τύπο «αλγόριθμος = λογική + έλεγχος».

Σε μια ιδανική γλώσσα λογικού προγραμματισμού, ο προγραμματιστής θα έπρεπε να ανησυχεί μόνο για την δήλωση των συστατικών του αλγορίθμου, και να αφήσει να ασκηθεί ο έλεγχος αποκλειστικά και μόνο από το ίδιο το λογικό σύστημα προγραμματισμού. Με άλλα λόγια, ένα ιδανικό σύστημα λογικού προγραμματισμού, ορίζεται από απόλυτα δηλωτικό προγραμματισμό.

2.1. Λογική

Η λογική παρέχει έναν τρόπο για την αποσαφήνιση και την τυποποίηση της διαδικασίας της ανθρώπινης σκέψης. Μας επιτρέπει να συλλογιζόμαστε για την ορθότητα συμπερασμάτων, να αναπαριστούμε προβλήματα αλλά και να τα επιλύουμε. Η ανάγκη για μια τέτοια φορμαλιστική αναπαράσταση της ανθρώπινης σκέψης προήλθε από το γεγονός ότι η φυσική γλώσσα, είναι πολυσήμαντη, ασαφής, περιέχει συμφραζόμενα, κ.α.

Η Μαθηματική Λογική έχει τις απαρχές της στην αρχαιότητα, στη θεωρία του *Συλλογισμού* του Αριστοτέλη. Η μοντέρνα μαθηματική λογική ξεκίνησε με τις εργασίες των Descartes (1596-1650) και Leibniz (1646-1716) και συνεχίστηκε από την εργασία του Boole επάνω στη *Μαθηματική Ανάλυση της Λογικής*. Η Μαθηματική Λογική είναι η συστηματική μελέτη των έγκυρων ισχυρισμών με χρήση εννοιών από τα μαθηματικά. Ένας ισχυρισμός

αποτελείται από συγκεκριμένες δηλώσεις (ή προτάσεις), τις υποθέσεις, από τις οποίες παράγεται μια άλλη δήλωση που ονομάζεται συμπέρασμα. Για παράδειγμα, ο επόμενος ισχυρισμός μας λέει ότι

Όλοι οι άνθρωποι είναι θνητοί,
Ο Σωκράτης είναι άνθρωπος,

επομένως, ο Σωκράτης είναι θνητός

Η Συμβολική Λογική είναι μια στενογραφία της κλασσικής λογικής. Οι ισχυρισμοί μελετώνται ανεξάρτητα από την περιοχή στην οποία ανήκουν. Αυτό επιτυγχάνεται εκφράζοντάς τους στη λογική σε συμβολική μορφή.

Για παράδειγμα

$$υπόθεση_1 : \forall \mathrm{X}, άνθρωπος(\mathrm{X}) \rightarrow θνητός(\mathrm{X})$$
$$υπόθεση_2 : άνθρωπος(Σωκράτης),$$

συμπέρασμα : θνητός(Σωκράτης)

2.2. Λογική «πρώτης τάξης» (First order theory)

Το σύνολο των αξιωμάτων που ανήκουν στα πλαίσια της λογικής πρώτης τάξης (first order theory), περιγράφεται από τις προτάσεις.

Μία πρόταση μπορεί να έχει δύο πιθανές τιμές είτε αληθής ή ψευδής. Για παράδειγμα, η ακόλουθη πρόταση μπορεί να είναι είτε αληθής ή ψευδής.

Ο Γιάννης είναι τριτοετής σπουδαστής στο Τμήμα Πληροφορικής.

Η εκφραστική δύναμη συστημάτων η αναπαράσταση γνώσεων των οποίων στηρίζεται στην λογική έγκειται στον τρόπο με τον οποίο κτίζεται η γνώση. Αρχικά, έννοιες πάνω στις οποίες απλές ιδέες μπορούν να εκφραστούν είναι η έννοια της αλήθειας και της μη αλήθειας με επιπλέον έννοιες και σύμβολα όπως οι λογικοί σύνδεσμοι, τα κατηγορήματα, κτλ. περισσότερο εκφραστικές λογικές μπορούν να δημιουργηθούν, κατά συνέπεια πιο πολύπλοκες και πιο λεπτές ιδέες μπορούν να αναπαρασταθούν.

Η μελέτη της λογικής σαν μέθοδος αναπαράστασης γνώσεων και εξαγωγής συμπερασμάτων περιλαμβάνει τα εξής:

1. Την *συντακτική μορφή* των προτάσεων. Δηλαδή ποια μορφή θα έχουν οι απλές προτάσεις και πως θα δημιουργούνται οι σύνθετες προτάσεις.
2. Την *ερμηνεία* και *αλήθεια* των προτάσεων. Μια πρόταση ανάλογα με το πεδίο του προβλήματος μπορεί να έχει διαφορετικές ερμηνείες. Για παράδειγμα, η σύνθετος πρόταση "p *or* q" μπορεί να έχει την εξής ερμηνεία στο χώρο των ακέραιων αριθμών «Ο Ν είναι άρτιος ακέραιος *ή* ο Ν είναι περιττός ακέραιος». Στο χώρο της μετεωρολογίας μπορεί να έχει την εξής ερμηνεία «Ο καιρός είναι βροχερός ή ο καιρός είναι συννεφιασμένος». Η αλήθεια της σύνθετης πρότασης εξαρτάται από την αλήθεια των απλών προτάσεων p και q.
3. Την *απόδειξη* ή *εξαγωγή* νέων προτάσεων από τις υπάρχουσες προτάσεις

Η αναπαράσταση γνώσεων σε λογική είναι δηλωτική. Τα πλεονεκτήματα της είναι τα εξής:

α. Η γνώση μπορεί εύκολα ν‘ αλλάζει
β. Επέκταση της γνώσης γίνεται με συμπερασματικούς κανόνες οι οποίοι εξάγουν επιπλέον γνώση πέρα απ‘ αυτή που σαφώς έχει οριστεί.
γ. Μπορεί να γίνει επεξεργασία της γνώσης από αναδρομικά προγράμματα. Συνεπώς, ένα σύστημα βασισμένο σε λογική μπορεί να απαντήσει σε ερωτήσεις για το τι γνωρίζει.

Ο προτασιακός λογισμός είναι η πιο απλή μορφή λογικής, ασχολείται με την αναπαράσταση πληροφοριών σαν προτάσεις καθώς και με την εξαγωγή συμπερασμάτων από προτάσεις. Ο προτασιακός λογισμός είναι μια συμβολική λογική η οποία ασχολείται με τις λογικές ιδιότητες συνθέτων προτάσεων. Μία πρόταση μπορεί να έχει μια τιμή από τις τιμές αληθείας, *αληθής* και *ψευδής*. Για παράδειγμα η πρόταση «*η Κρήτη είναι νησί*» έχει τιμή

αληθείας «*αληθή*», ενώ η πρόταση η «*η Πελοπόνησσος είναι νησί*» έχει την τιμή «*ψευδής*»

Αντίθετα, οι εκφράσεις, α) δυο συν τρία και β) ο πατέρας του Κώστα,

δεν είναι προτάσεις κατά συνέπεια δεν μπορούμε να τους δώσουμε κάποια τιμή αληθείας.

Ο ισχυρισμός, *η Κρήτη είναι νησί και βρίσκεται νοτίως της Θήρας*, είναι μια σύνθετη πρόταση η οποία αποτελείται από τις εξής δύο προτάσεις:

α) Η Κρήτη είναι νησί και β) η Κρήτη βρίσκεται νοτίως της Θήρας

Αυτές οι δυο προτάσεις συνδέθηκαν με το **και** για να σχηματίσουν την παραπάνω σύνθετη πρόταση. Απλές προτάσεις μπορούν να συνδέονται με λογικούς συνδέσμους για σχηματισμό πιο σύνθετων προτάσεων. Η χρήση λογικών συνδέσμων σε προτάσεις δημιουργεί την πιο απλή μορφή λογικής, τον προτασιακό λογισμό. Με τον προτασιακό λογισμό μπορούμε να εκφράσουμε σύνθετες προτάσεις.

Παραδείγματα
- Ο Γιάννης σπουδάζει Πληροφορική και βρίσκεται στο τρίτο έτος.
- Εάν ο Γιάννης είναι τριτοετής σπουδαστής Πληροφορικής τότε έχει εγγραφεί στο μάθημα Τεχνητή Νοημοσύνη.

Ένα σύνολο προτάσεων της παραπάνω μορφής ορίζουν ένα λογικό πρόγραμμα πρώτης τάξης.

Η εξέλιξη στον χώρο του λογικού προγραμματισμού, οδήγησε στην εισαγωγή νέων εννοιών όπως ο «μεταπρογραμματισμός» και οι «λογικοί μετασχηματισμοί».

2.3. Λογικός Προγραμματισμός και Prolog

Η ανάγκη φορμαλισμού της λογικής, οδήγησε στον λογικό προγραμματισμό. Είναι απαραίτητο επομένως να ορίσουμε τυπικά κάποιες έννοιες τις οποίες θα χρησιμοποιήσουμε στην συνέχεια.

Θεωρούμε ως σύμβολα **σταθερές** ονόματα που έχουν το πρώτο γράμμα τους πεζό και ακολουθεί είτε ψηφίο ή υπογράμμιση ή πεζό ή κεφαλαίο γράμμα, π.χ. a,s1,const,…

Σύμβολα **συναρτήσεων** είναι ονόματα που έχουν το πρώτο γράμμα τους πεζό και ακολουθεί είτε ψηφίο ή υπογράμμιση ή πεζό ή κεφαλαίο γράμμα, π.χ. a,s1,const,… Επιπλέον ο συμβολισμός f/ν χρησιμοποιείται για τις συναρτήσεις, όπου f είναι το όνομα της συνάρτησης, και ν το πλήθος των ορισμάτων της.

Σύμβολα **μεταβλητών** είναι ονόματα που έχουν το πρώτο γράμμα τους κεφαλαίο και ακολουθεί είτε ψηφίο ή υπογράμμιση ή πεζό ή κεφαλαίο γράμμα,: X,Y,Z,X1,…

Σύμβολα **κατηγορημάτων** είναι ονόματα με το πρώτο γράμμα πεζό και ακολουθεί είτε ψηφίο ή υπογράμμιση ή πεζό ή κεφαλαίο γράμμα, όπως: αρέσει, είναι, έχει, γονιός, πατέρας, κτλ… Ο συμβολισμός P/ν χρησιμοποιείται και για τα κατηγορήματα όπως και για τις συναρτήσεις.Δηλαδή, όπου P είναι το όνομα του κατηγορήματος, και ν το πλήθος των ορισμάτων του.

Ορισμός 2.1: **Πλειάδα** (tuple ή ν-tuple όπου ν>0 και ν πεπερασμένος αριθμός), είναι μια σειρά από ν όρους. Τα στοιχεία b, Y και Z, μπορούν να ταξινομηθούν στην πλειάδα <b, Y, Z>. Άλλη μια δυνατή ταξινόμηση είναι η πλειάδα <Y, b, Z>, η οποία έχει διαφορετική σημασία από την προηγούμενη.

Οι λογικοί τελεστές and, or και not αντικαθιστούνται από τα (,), (;) και (\+) αντίστοιχα (όπως στην Prolog) για ομοιομορφία στον συμβολισμό. Αυτό αποσκοπεί στην απλοποίηση της πτυχιακής εργασίας.

Ορισμός 2.2: *Ένας* **όρος** (term), είναι είτε μια σταθερά, ή μια μεταβλητή, ή μια συνάρτηση εφαρμοζόμενη σε μια πλειάδα (tuple) όρων όπως στα παρακάτω παραδείγματα:

a, X, f(b,f(b, X, Y), Y)

Ορισμός 2.3: Ένας **ατομικός τύπος** (atomic formula), είναι ένα κατηγόρημα εφαρμοζόμενο σε μια πλειάδα όρων όπως στα παρακάτω παραδείγματα:

έχει(γιάννης, βιβλίο) είναι(X,μητέρα(μαρία))

Ορισμός 2.4: Ένας **στοιχειώδης τύπος**, είναι είτε ένας ατομικός τύπος ή ένας αρνητικός ατομικός τύπος. Για παράδειγμα τα p(X,Y), ¬p(X,Y), q(a,Z), ¬q(a,Z) είναι στοιχειώδεις τύποι.

Ορισμός 2.5: *Μία* **πρόταση** (clause) έχει την μορφή:

$$h:-l_1, l_2, \ldots, l_n$$

όπου το h είναι ένας ατομικός τύπος και τα l_1, l_2, …, l_n είναι στοιχειώδεις τύποι. Επίσης, το h ονομάζεται *κεφαλή* (*head*) της πρότασης και το τμήμα l_1, l_2, …, l_n *σώμα* (*body*).

Αν n≥0 η πρόταση ονομάζεται *κανόνας* (*rule*), ενώ αν n=0 ονομάζεται *γεγονός* (*fact*) (ή *μοναδιαία πρόταση -unit clause*) και γράφεται, απλά, h. 'Ενας στόχος (η ερώτημα - query) σ' ένα λογικό πρόγραμμα, συντάσσεται ως «?- l_1,…, l_i, …, l_n.» όπου l_i στοιχειώδης τύπος και i θετικός ακέραιος αριθμός. Κάθε πρόταση ή στόχος τερματίζεται από μια τελεία. Όπως φαίνεται στις παραπάνω προτάσεις, στα λογικά προγράμματα το σύμβολο της συνεπαγωγής "**←**" μιας πρότασης Horn γράφεται ":-".

Μια συλλογή προτάσεων που έχουν το ίδιο όνομα κατηγορήματος για κεφαλή ορίζει μια *σχέση* (relation) ή μια *διαδικασία* (procedure).

Για παράδειγμα, θεωρούμε το λογικό πρόγραμμα 2.1 που αναπαριστά οικογενειακές σχέσεις:

```
father(manos,giannhs).
father(manos,maria).
mother(maria,nikos).
mother(maria,anna).
parent(X,Y) :-father(X,Y)
parent(X,Y) :-mother(X,Y).
```

Πρόγραμμα 2.1 Σχέσεις οικογένειας

Στους παρακάτω στόχους, θα έχουμε τα εξής αποτελέσματα:

```
?-parent(Who,giannhs).
Who = manos ;
no

?-parent(Who,anna).
Who = maria ;
no

?-parent(maria,Who).
Who = nikos ;
Who = anna ;
no
```

Το κατηγόρημα father(X,Y) δηλώνει ότι ο X είναι πατέρας του Y. Παρόμοια σχέση ορίζεται από το κατηγόρημα mother(X,Y). Τα δύο αυτά κατηγορήματα ορίζονται από σύνολα γεγονότων. Επίσης, το κατηγόρημα parent/2 ορίζεται από δύο κανόνες που περιγράφονται από τα κατηγορήματα father/2 και mother/2. Δηλαδή, ο X είναι γονιός του Y, αν ο X είναι πατέρας του Y, ή αν η X είναι μητέρα του Y. Η απάντηση της ερώτησης (ή στόχου) «?-parent(maria, Who).» είναι το σύνολο των αντικαταστάσεων {Who/nikos, Who/anna}.

Η γλώσσα Prolog, αποτελεί σημαντικό προγραμματιστικό εργαλείο, ιδιαίτερα εφαρμοζόμενη σε προγράμματα που περιέχουν συμβολικές ή μη-αριθμητικές πράξεις. Για αυτό τον λόγο, χρησιμοποιείται για προγραμματισμό εφαρμογών Τεχνητής Νοημοσύνης, όπου ο χειρισμός συμβόλων και η εξαγωγή συμπερασμάτων από αυτά, είναι συχνές λειτουργίες. Η Prolog, όπως και κάθε άλλη γλώσσα λογικού προγραμματισμού, ακολουθεί τις εξής αρχές [9]:

1. Εκτελεί έρευνα σε ένα δέντρο που ονομάζεται δέντρο έρευνας για να βρει μια λύση.
2. Εκτελεί οπισθοδρόμιση είτε για να βρει εναλλακτικές λύσεις ή σε πρίπτωση αποτυχίας.
3. Χρησιμοποιεί σαν δομές δεδομένων τους όρους και ιδιαίτερα την λίστα η οποία σαν όρος παριστάνεται σαν ένα δυαδικό δέντρο.

2.3.1. Έρευνα

Ας υποθέσουμε ότι έχουμε το πρόγραμμα 2.2 (βάση γνώσεων):

```
eats(fred,pears).
eats(fred,t_bone_steak).
eats(fred,apples).
```

Πρόγραμμα 2.2 Βάση γνώσεων

Μέχρι τώρα μπορούσαμε μόνο να ρωτάμε αν ο fred τρώει συγκεκριμένα είδη. Υποθέτουμε τώρα ότι θέλουμε να απαντήσουμε στην ερώτηση «Ποια είναι όλα τα είδη που ο fred τρώει;». Το ερώτημα αυτό θα έχει ως εξής:

```
?- eats(fred,FoodItem).
```

Η Prolog στο παραπάνω ερώτημα θα απαντήσει

```
FoodItem = pears
```

Αυτό συμβαίνει επειδή βρήκε την πρώτη πρόταση στην βάση δεδομένων και η οποία ικανοποιούσε την ερώτηση. Σε αυτό το σημείο, η Prolog μας επιτρέπει να ρωτήσουμε αν υπάρχει άλλη δυνατή λύση. Αν λοιπόν ρωτήσουμε ξανά, παίρνουμε την απάντηση

```
FoodItem = t_bone_steak
```

Αν ρωτήσουμε ξανά για μια ακόμα λύση, η Prolog θα επιστρέψει:

```
FoodItem = apples
```

Αν ζητήσουμε περισσότερες λύσεις, η απάντηση θα είναι *no*, αφού υπάρχουν μόνο τρεις δυνατές απαντήσεις για το τι τρώει ο fred. Ο μηχανισμός εύρεσης πολλαπλών λύσεων ονομάζεται *backtracking*.

Μπορούμε επίσης να έχουμε backtracking σε κανόνες. Για παράδειγμα, θεωρούμε το πρόγραμμα 2.3.

```
hold_party(X):-
        birthday(X), happy(X).
birthday(tom).
birthday(fred).
birthday(helen).
happy(mary).
happy(jane).
happy(helen).
```

Πρόγραμμα 2.3 Παράδειγμα backtracking

Αν τώρα θέσουμε το ερώτημα:

```
?- hold_party(Who).
```

Προκειμένου να λυθεί το ερώτημα αυτό, η Prolog ισοδύναμα αντικαθιστά τον αρχικό στόχο με τον «?- birthday(Who), happy(Who).». Η Prolog πρώτα προσπαθεί να ικανοποιήσει τον πρώτο από αριστερά στόχο, δηλαδή *«?-birthday(Who).»*. Η έρευνα αυτή δεσμεύει την μεταβλητή Who με την τιμή tom.

Στη συνέχεια, ερευνάται ο στόχος «?-happy(tom)». Το «υπό-ερώτημα» αυτό αποτυγχάνει, καθώς δεν ταιριάζει με κάποιο από τα στοιχεία της βάσης γνώσεων. Αυτό έχει ως αποτέλεσμα, η Prolog να ξαναπροσπαθήσει (backtrack), επιστρέφοντας στον προηγούμενο στόχο και ψάχνοντας κάποιον άλλο κανόνα για ικανοποίηση του «?-birthday(Who)». Αυτή τη φορά θα χρησιμοποιηθεί το δεύτερο γεγονός, δεσμεύοντας το Who στον fred. Αυτό έχει ως συνέπεια, να αναζητηθεί λύση στον στόχο happy(fred). Ξανά αυτός ο στόχος θα αποτύχει αφού ικανοποιείται από την βάση γνώσεων και θα επιστρέψει στον αρχικό στόχο (backtrack). Αυτή τη φορά βρίσκει τον τρίτο κανόνα, δεσμεύοντας την μεταβλητή Who στην τιμή helen. Στη συνέχεια, όπως και στις προηγούμενες περιπτώσεις, δοκιμάζεται ο στόχος happy(helen), ο οποίος είναι αληθής, αφού ταιριάζει με την τρίτη πρόταση της βάσης γνώσεων των προτάσεων του happy/1. Συνεπώς, ο στόχος «?-hold_party(Who).» επιτυγχάνει με Who=helen.

2.3.2. Αναδρομές

Συχνά επιθυμούμε να υλοποιήσουμε κάποιες λειτουργίες επαναληπτικά, για παράδειγμα είτε σε όλα τα στοιχεία μιας λίστας, ή μέχρι να ευρεθεί ένα συγκεκριμένο στοιχείο της. Ο τρόπος με τον οποίο τις περισσότερες φορές επιτυγχάνεται αυτό, είναι η αναδρομή (recursion). Αυτό

σημαίνει, ότι κάποιο πρόγραμμα, καλεί τον εαυτό του, έως ότου καλέσει την μη αναδρομική πρόταση. Σε κάποιο πρόγραμμα Prolog η πρώτη πρόταση συνήθως είναι η μη αναδρομική πρόταση (συνθήκη τερματισμού της αναδρομής). Αυτή η πρόταση ακολουθείται από τις αναδρομικές προτάσεις. Αυτή συμβαίνει επειδή η Prolog εξετάζει τις προτάσεις με την σειρά που έχουν καταχωρηθεί στο πρόγραμμα.

Οι αναδρομικοί κανόνες αυξάνουν σημαντικά την εκφραστική δύναμη του λογικού προγραμματισμού. Για παράδειγμα θεωρούμε το πρόγραμμα 2.1, το οποίο θα επεκτείνουμε με αναδρομικές προτάσεις. Για παράδειγμα, αν θέλουμε να ορίσουμε τη σχέση ancestor/2 (πρόγονος) με χρήση της σχέσης parent/2, το πιθανότερο είναι να χρειαστεί να γράψουμε ένα άπειρο σύνολο κανόνων της μορφής

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Z), parent(Z,W), parent(W,Y).
…
```

Αξιοποιώντας όμως την αναδρομή, αρκεί να προστεθεί στο παραπάνω πρόγραμμα η εξής περιγραφή:

1) πρόγονος κάποιου είναι ο γονιός του,
2) πρόγονος κάποιου Y είναι ο X, εάν ο X είναι πατέρας κάποιου Z, ο οποίος είναι πρόγονος του Y. Δηλαδή,

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Στην ερώτηση

```
?- ancestor(manos,Who).
```

Η έρευνα επιστρέφει τα αποτελέσματα:

```
Who = giannhs ;
Who = maria ;
Who = nikos ;
Who = anna ;
no
```

Παρατίθεται επίσης το πρόγραμμα 2.4 σαν ένα επιπλέον παράδειγμα αναδρομής:

```
on_route(rome).
on_route(Place):-
        move(Place,Method,NewPlace),
        on_route(NewPlace).
```

```
move(home,taxi,halifax).
move(halifax,train,gatwick).
move(gatwick,plane,rome).
```

Πρόγραμμα 2.4 Μετακινήσεις

Παρατηρούμε ότι το on_route είναι ένα αναδρομικό κατηγόρημα. Το παραπάνω πρόγραμμα ελέγχει εαν είναι δυνατό να ταξιδέψει κάποιος στην Ρώμη, από ένα συγκεκριμένο σημείο. Η πρώτη πρόταση του κατηγορήματος on_route/1, ελέγχει εαν έχουμε ήδη φτάσει στην Ρώμη και σε αυτή την περίπτωση, το πρόγραμμα τερματίζει. Η δεύτερη πρόταση, ελέγχει εαν υπάρχει κάποια επιτρεπτή κίνηση από το τρέχον σημείο σε κάποιο καινούργιο, και τότε αναδρομικά εξετάζεται εαν το καινούριο σημείο (NewPlace) βρίσκεται στην διαδρομή (on_route) για την Ρώμη. Στην συνέχεια, ακολουθεί η βάση δεδομένων για τις επιτρεπτές κινήσεις με το κατηγόρημα move.

Ας μελετήσουμε τι συμβαίνει θέτοντας την ερώτηση

```
?- on_route(home).
```

Το ερώτημα αυτό ταιριάζει με την δεύτερη πρόταση του on_route (δεν μπορεί να ταιριάξει με την πρώτη πρόταση, επειδή οι τοποθεσίες home και rome, δεν ταυτίζονται). Η πρόταση αυτή, αποτελείται από δύο υποστόχους. Ο πρώτος ερευνά αν είναι δυνατό να κινηθεί κάποιος από το home σε κάποια νέα τοποθεσία δηλαδή, move(home,Method,NewPlace). Όπως φαίνεται αυτό επιτυγχάνει για Method = taxi, NewPlace = halifax. Η απάντηση αυτή, σημαίνει ότι μπορούμε να μετακινηθούμε από την τοποθεσία home στο halifax, χρησιμοποιώντας taxi. Στην συνέχεια, αναδρομικά ερευνούμε αν μπορούμε να βρούμε μια διαδρομή από το halifax στη rome. Δηλαδή, εκτελείται ο νέος υποστόχος

```
?-on_route(halifax).
```

Ο υποστόχος ?-move(halifax,Method,NewPlace) επιτυγχάνει, επειδή μπορούν να ταυτοποιηθούν τα Method=train και NewPlace=gatwick. Γνωρίζοντας τον νέο υποστόχο on_route(gatwick) και ο οποίος καλείται αναδρομικά, ερευνούμε αν υπάρχει τρόπος μετάβασης από το gatwick στη rome.

Κατά την κλήση αυτή, ερευνάται πρώτα το κατηγόρημα move, όπου αυτή την φορά το Place έχει δεσμευτεί στο gatwick. Αυτό το ερώτημα ταιριάζει με την τρίτη πρόταση της βάσης δεδομένων των move. Το αποτέλεσμα είναι Method=plane, NewPlace=rome και εκτελείται ξανά

αναδρομικά ο νέος υποστόχος ?-on_route(rome). Η νέα αυτή κλήση ταιριάζει με την πρώτη από τις προτάσεις on_route και επομένως η διαδικασία επιτυγχάνει, αφού η πρόταση αυτή είναι ένα γεγονός. Ως αποτέλεσμα αυτού είναι η επιτυχία όλων των προηγούμενων ερωτημάτων, συνέπως και του αρχικού ερωτήματος
«?- on_route(home).», οπότε η Prolog επιστρέφει yes.

2.3.3. Λίστες

Μέχρι αυτό το σημείο, θεωρήσαμε απλά ορίσματα στα προγράμματά μας. Ωστόσο, στην Prolog μια πολύ συνηθισμένη δομή δεδομένων είναι οι λίστες. Οι λίστες έχουν την ακόλουθη σύνταξη. Πάντοτε ξεκινούν και τελειώνουν με αγκύλες, ενώ τα περιεχόμενα της διαχωρίζονται με κόμμα. Για παράδειγμα, μια λίστα σύμφωνα με τα προηγούμενα έχει την εξής μορφή [a,freddie,A_Variable,apple].

Η Prolog επίσης διαθέτει μια ιδιαίτερη λειτουργία διαχωρισμού του πρώτου στοιχείου μια λίστας (ονομαζόμενο ως «κεφάλι») από τα υπόλοιπα (ονομαζόμενο ως «ουρά»). Τοποθετούμε στην περίπτωση αυτή ένα ειδικό σύμβολο | (κάθετη) στην λίστα, για να διαχωρίσουμε το πρώτο στοιχείο από την υπόλοιπη λίστα. Για παράδειγμα, ας θεωρήσουμε τα ακόλουθα, όπου ο τελεστής = εκτελεί την πράξη της ταυτοποίησης:

[first,second,third] = [A|B]

όπου A = first και B=[second,third]

Η ταυτοποίηση εδώ επιτυγχάνει. Το A δεσμεύεται στο πρώτο στοιχείο της λίστας και το B στο υπόλοιπο μέρος της.

Μελετώντας ορισμένες συγκρίσεις μεταξύ λιστών, παρατηρούμε τα εξής:

?- [a,b,c] = [Head|Tail] επιστρέφει τα Head=a και Tail=[b,c]

?- [a] = [H|T] επιστρέφει τα H=a και T=[]

?- [a,b,c] = [a|T] επιστρέφει το T=[b,c]

?- [a,b,c] = [b|T] επιστρέφει η Prolog no.

?- []= [H|T] επιστρέφει η Prolog no.

[] = []. Δύο άδειες λίστες πάντοτε ταυτοποιούνται.

Επιπλέον, θεωρούμε το παρακάτω γεγονός σαν μέρος του προγράμματος μας:

p([H|T], H, T).

Οι απαντήσεις της Prolog σε τρεις διαφορετικούς στόχους, έχουν ως εξής:

1. ?- p([a,b,c], X, Y).

X=a
Y=[b,c]
yes

2. ?- p([a], X, Y).
X=a
Y=[]
yes

3. ?- p([], X, Y).
no.

2.5. Μετά-προγραμματισμός

Μετα-γλώσσα είναι μια γλώσσα που περιγράφει κάποια άλλη γλώσσα. Η περιγραφόμενη γλώσσα ονομάζεται γλώσσα αντικείμενο. Για παράδειγμα «ο πληθυντικός των ουσιαστικών στην Αγγλική σχηματίζεται προσθέτοντας την κατάληξη –s πλην των ουσιαστικών που τελειώνουν σε –ch, -sh, -o, -x και –ss στα οποία προστίθεται η κατάληξη –es», για να σχηματισθεί αυτή η πρόταση, η Ελληνική γλώσσα χρησιμοποιήθηκε ως μεταγλώσσα για να περιγράψει την αγγλική γλώσσα (γλώσσα-αντικείμενο).

Ορισμός 2.5: **Μετα-πρόγραμμα** είναι ένα πρόγραμμα το οποίο δέχεται σαν δεδομένα ένα άλλο πρόγραμμα. Το πρόγραμμα που χρησιμοποιείται σαν δεδομένα ονομάζεται **πρόγραμμα αντικείμενο**. Μετα-προγράμματα αναλύουν, μετασχηματίζουν, κατασκευάζουν κτλ, προγράμματα-αντικείμενα.

Για παράδειγμα, μεταφραστές, μεταγλωττιστές, κτλ, είναι μεταπρογράμματα. Η γλώσσα στην οποία γράφεται ένας μεταγλωττιστής ή ένας μεταφραστής είναι η μετα-γλώσσα και η γλώσσα την οποία μεταγλωττίζει ή μεταφράζει, είναι η γλώσσα αντικείμενο.

Ο μετα-προγραμματισμός προϋποθέτει τον σωστό διαχωρισμό των μεταβλητών μεταξύ του προγράμματος αντικειμένου, και του μετα-προγράμματος. Για τον σκοπό αυτό εισάγουμε τις έννοιες *βασικός όρος*, *μη βασικός όρος*, *βασικός ατομικός τύπος* και *μη βασικός ατομικός*, με σκοπό να περιγράψουμε την «*βασική*» και «*μη βασική*» αναπαράσταση.

Ορισμός 2.6: **Βασικός όρος** είναι ένας όρος, ο οποίος δεν περιέχει μεταβλητές. Για παράδειγμα, έστω a, b σταθερές, X, Y μεταβλητές, f/1, g/2 συναρτήσεις και p/1, q/2 κατηγορήματα. Οι όροι a, b, f(a), f(b), g(a, b), g(a, a), g(b, f(a)), f(f(b)), κτλ. είναι βασικοί όροι. Αντίθετα, οι X, Y, f(X), f(Y), f(f(X)), g(Y, b), g(X, f(Y)), κτλ. δεν είναι βασικοί όροι.

Ορισμός 2.7: **Βασικός ατομικός τύπος**, είναι ένας ατομικός τύπος, όλα τα ορίσματα του οποίου είναι βασικοί όροι. Για παράδειγμα, έστω a, b σταθερές, X, Y μεταβλητές, f/1, g/2 συναρτήσεις και p/1, q/2 κατηγορήματα. Οι ατομικοί τύποι p(a), p(f(b)), p(f(f(a))), q(a,f(b)), q(f(a), g(a, b)) κτλ. είναι βασικοί ατομικοί τύποι ενώ οι p(X), p(f(X)), q(a, f(X)), q(X, g(a, X)) κτλ. δεν είναι βασικοί ατομικοί τύποι.

Η αναπαράσταση σε βασικούς όρους παρέχει την δυνατότητα ευκρίνειας στο μετα-πρόγραμμα, δηλαδή τα στοιχεία (μεταβλητές, σταθερές, κτλ) του προγράμματος αντικειμένου είναι διακριτά από τα αντίστοιχα του μετα-προγράμματος. Επιπλέον, μπορούν να γραφούν μετα-προγράμματα που ανήκουν στην λογική πρώτης τάξης δηλαδή, να δοθεί σε κάθε μεταπρόγραμμα απλή και ακριβής σημασιολογία στα πλαίσια της λογικής πρώτης τάξης. Τέλος, το πρόγραμμα-αντικείμενο μπορεί να τροποποιηθεί με λογικό τρόπο.

Το μειονέκτημα της αναπαράστασης σε βασικούς όρους είναι ότι τα μετά-προγράμματα δεν είναι αποτελεσματικά. Αυτό συμβαίνει επειδή οι

μεταβλητές της γλώσσας-αντικείμενο πρέπει να παρασταθούν σαν σταθερές της μεταγλώσσας. Αυτό συνεπάγεται ότι πρέπει να γραφούν πολύπλοκα προγράμματα τα οποία θα εκτελούν την μετονομασία των μεταβλητών, την ταυτοποίηση των όρων και την εφαρμογή των αντικαταστάσεων στους όρους και στους τύπους.

Το πλεονέκτημα της αναπαράστασης σε μη βασικούς όρους είναι ότι η γλώσσα-αντικείμενο και η μετα-γλώσσα μπορούν να χρησιμοποιήσουν τα ίδια κατηγορήματα που είναι ενσωματωμένα στην γλώσσα του λογικού προγραμματισμού, για μετονομασία μεταβλητών για ταυτοποίηση όρων και για εφαρμογή των αντικαταστάσεων στους όρους και τους τύπους.

Στον αντίποδα, τα μειονεκτήματα είναι ότι η γλώσσα-αντικείμενο και η μετα-γλώσσα χρησιμοποιούν τις ίδιες μεταβλητές, με αποτέλεσμα τα μετα-προγράμματα να μην έχουν ακριβή σημασιολογία στην λογική πρώτης τάξης. Επίσης, το πρόγραμμα-αντικείμενο δεν μπορεί να τροποποιηθεί με λογικό τρόπο. Δηλαδή, να τροποποιηθεί από ένα μετα-πρόγραμμα το οποίο να έχει ακριβή σημασιολογία στην λογική πρώτης τάξης.

3. Μετασχηματισμός προγραμμάτων

Στις μέρες μας, το ενδιαφέρον στην προσέγγιση του προγραμματισμού μέσω μετασχηματιστικών τεχνικών έχει αυξηθεί, με σκοπό την εξέλιξη του τρόπου προγραμματισμού. Τα μετασχηματιστικά βοηθήματα ποικίλλουν, από απλούς κειμενογράφους έως και ισχυρά διαλογικά συστήματα μετασχηματισμού, ακόμα και αυτόματα εργαλεία σύνθεσης προγραμμάτων [12], [13].

Ο προγραμματισμός είναι πάντα μια δύσκολη διαδικασία, χαρακτηριζόμενη από το πρόβλημα κυριαρχίας επί της πολυπλοκότητας. Υπάρχουν πολλά βήματα μεταξύ της ανάλυσης ενός προβλήματος και της αποτελεσματικής επίλυσής του. Είναι κοινώς αποδεκτό το γεγονός ότι οι δυσκολίες που σχετίζονται με την κατασκευή σωστών προγραμμάτων, μπορούν να υπερνικηθούν εάν ολόκληρη η διαδικασία διαιρεθεί σε ικανοποιητικά μικρά και μεθοδικά βήματα. Για του λόγου το αληθές, ένα σύνθετο πρόβλημα, επιλύεται ευκολότερα εαν διασπαστεί σε μικρότερα προβλήματα και κάθε ένα απο αυτά αντιμετωπιστούν ξεχωριστά. Οι μεγάλοι οργανισμοί και επιχειρήσεις ανάπτυξης λογισμικού, έχουν ασπαστεί την ανάγκη για μεθοδική ανάπτυξη λογισμικού το οποίο έχει οριστεί τυπικά και για εργαλεία υποστήριξης της διαδικασίας ανάπτυξής του. Το γεγονός αυτό τεκμηριώνεται άλλωστε και από την προσπάθεια ανάπτυξης ενός προγραμματιστικού περιβάλλοντος εργασίας στην Ada για το Υπουργείο Εθνικής Άμυνας των Η.Π.Α.

Σε κάθε προγραμματιστική μεθοδολογία, η δημιουργικότητα συμβαδίζει με πολλές αυτοματοποιημένες διαδικασίες. Ουσιώδης υποστήριξη στην ανάπτυξη λογισμικού, είναι λογικό να αναμένεται εάν τα μέρη που μπορούν να αυτοματοποιηθούν εκτελούνται από την μηχανή και ο προγραμματιστής είναι ελεύθερος να συγκεντρωθεί στο δημιουργικό τμήμα. Τα συστήματα ανάπτυξης λογισμικού, σταδιακά μετατρέπονται σε απαραίτητα εργαλεία για τον προγραμματισμό.

Ορισμός 3.1: Ένα **πρόγραμμα** είναι η περιγραφή της μεθοδολογίας ενός υπολογισμού, εκφραζόμενη σε μια συμβατική γλώσσα.

Ορισμός 3.2: Οι **κανόνες μετασχηματισμού** είναι διαδικασίες, η εφαρμογή των οποίων σε κάποιο πρόγραμμα οδηγεί στον έγκυρο μετασχηματισμό του. Εφαρμογή ενός κανόνα μετασχηματισμού, απλά σημαίνει εφαρμογή της συγκεκριμένης διαδικασίας.

Ορισμός 3.3: **Μετασχηματιστικός προγραμματισμός**, είναι η μεθοδολογία της κατασκευής ενός προγράμματος από διαδοχικές εφαρμογές των κανόνων μετασχηματισμού. Συνήθως, αυτή η διαδικασία αρχίζει με μια (συμβατική) δήλωση, δηλαδή μια συμβατική έκθεση του προβλήματος ή της λύσης του και τελειώνει με ένα εκτελέσιμο πρόγραμμα. Οι ξεχωριστές αλλαγές ανάμεσα στις διάφορες εκδόσεις ενός προγράμματος, γίνονται με την εφαρμογή κανόνων μετασχηματισμού οι οποίοι φυσικά είναι έγκυροι. Είναι

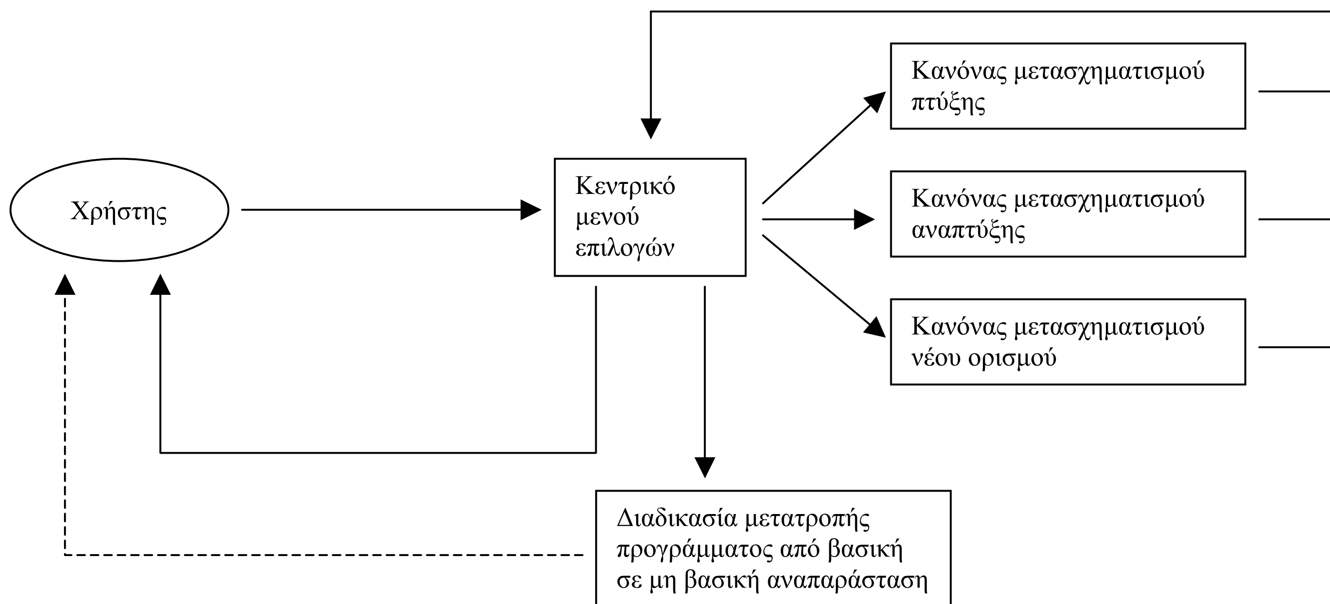
βέβαιο ότι η τελική έκδοση του προγράμματος θα ικανοποιεί την αρχική δήλωση.

Παρότι η ιδιαίτερη έμφαση του μετασχηματιστικού προγραμματισμού μπορεί να ποικίλει (κάποια συστήματα μπορεί να δίνουν περισσότερη βαρύτητα στο μέγεθος του προγράμματος και κάποια άλλα στην ταχύτητα), η κατασκευαστική προσέγγιση στην ανάπτυξη λογισμικού παραμένει σταθερή. Αυτό αντιτίθεται για παράδειγμα, στην καθαρά θεωρητική προσέγγιση, όπου το ζήτημα επαλήθευσης του νέου προγράμματος αγνοείται.

Τελικά, ένα σύστημα μετασχηματισμού, είναι ένα σύστημα το οποίο εκτελεί μετασχηματιστικό προγραμματισμό. Όμως, δεν αρκεί κάποιος να μάθει μόνο για τις τεχνικές λεπτομέρειες όπως για παράδειγμα την αναπαράσταση του προγράμματος εισόδου, την μέθοδο και αναπαράσταση της εξόδου, τον τρόπος χειρισμού ή τις εσωτερικές αναπαραστάσεις και τεχνικές υλοποίησης που έχουν χρησιμοποιηθεί [11].

Στο σύστημα μετασχηματισμού που υλοποιήσαμε, αξιοποιήσαμε τους έγκυρους κανόνες μετασχηματισμού fold/unfold όπως έχουν εφαρμοστεί από τους Tamaki και Sato [15]. Οι εφαρμογές αυτών των κανόνων μετασχηματισμού, γινονται όπως έχουμε αναφέρει στο πρόγραμμα αντικείμενο που εισάγουμε σε βασική αναπαράσταση. Βεβαίως, παρέχεται η δυνατότητα στον χρήστη να «μεταφέρει» το πρόγραμμα-αντικείμενο από βασική αναπαράσταση σε μη βασική, στο τέλος όμως των μετασχηματισμών. Η αναφορά αυτή γίνεται με σκοπό να διασφαλιστεί οτι η διαδικασία αυτή δεν συμπεριλαμβάνεται στους κανόνες μετασχηματισμού, καθώς δεν μετασχηματίζεται η δομή του προγράμματος, αλλά απλά αλλάζει ο τρόπος αναπαράστασής του.

Επιπλέον, το σύστημα που υλοποιήσαμε αξιοποιεί περισσότερους του ενός κανόνες μετασχηματισμού. Κρίνεται λοιπόν απαραίτητο να παρουσιαστεί ο τρόπος σύνδεσης των κανόνων αυτών μέσα στο σύστημα, ώστε να γίνει εύκολα κατανοητή η λειτουργία του. Μια γενική ιδέα του συστήματός μας, παρουσιάζεται στο παρακάτω σχήμα:



Υπόμνημα

——▶ Πρόγραμμα αντικείμενο σε βασική αναπαράσταση

- - -▶ Πρόγραμμα αντικείμενο σε μη βασική αναπαράσταση

Σχήμα 3.1 – Σχηματική αναπαράσταση του συστήματός μας.

Στο σχήμα 3.1, ο χρήστης εισάγει το πρόγραμμα αντικείμενο σε βασική αναπαράσταση. Στη συνέχεια αυτό οδηγείται στο κεντρικό μενού επιλογών, όπου θα επιλεγεί ο κανόνας μετασχηματισμού που θα εφαρμοστεί, ή η επιστροφή του προγράμματος αντικειμένου στον χρήστη σε βασική ή μη αναπαράσταση. Οι εφαρμογές των κανόνων μετασχηματισμού, μπορούν να επαναληφθούν πολλές φορές και για τον λόγο αυτό, μετά το πέρας τους επιστρέφουν το νέο πια πρόγραμμα αντικείμενο στο κεντρικό μενού επιλογών. Τελικά, όταν ο χρήστης το κρίνει απαραίτητο, το πρόγραμμα επιστρέφεται σε αυτόν στην αναπαράσταση που επιλέξει.

4. Πρόγραμμα αντικείμενο σε Βασικούς Όρους και αλγόριθμοι χειρισμού του

4.1. Αναπαράσταση

Κατά την αναπαράσταση του προγράμματος-αντικείμενο, σε βασικούς όρους, όλες οι σταθερές, μεταβλητές, σύνθετοι όροι και τύποι της γλώσσας-αντικείμενο, παριστάνονται μοναδικά σαν όροι της μεταγλώσσας. Ένας τρόπος παράστασης λογικών προγραμμάτων είναι ο εξής:

1. Κάθε σταθερά της γλώσσας-αντικείμενο παριστάνεται από μια μοναδική σταθερά της μετα-γλώσσας.
2. Κάθε μεταβλητή, της γλώσσας-αντικείμενο παριστάνεται από μια μοναδική σταθερά της μετα-γλώσσας.
3. Κάθε συνάρτηση / όρος f, με πλήθος ορισμάτων n>0 (f/n), της γλώσσας-αντικειμένου παριστάνεται από μια μοναδική συνάρτηση / όρο f, με πλήθος ορισμάτων n>0 (f/n) της μεταγλώσσας.
4. Κάθε κατηγόρημα p, με πλήθος ορισμάτων n>0 της γλώσσας αντικείμενο παριστάνεται από μια μοναδική συνάρτηση p με πλήθος ορισμάτων n>0 της μετα-γλώσσας.
5. Κάθε λογικός σύνδεσμος με πλήθος τελεστών n της γλώσσας-αντικείμενο παριστάνεται από μια μοναδική συνάρτηση της μετα-γλώσσας με πλήθος ορισμάτων n [7].

Η μετα-γλώσσα περιέχει φυσικά και άλλα σύμβολα εκτός από αυτά που χρησιμοποιεί για παράσταση της γλώσσας αντικείμενο. Για παράδειγμα ένας τρόπος παράστασης προγράμματος-αντικειμένου σε βασικούς όρους είναι ο εξής:

<u>Πρόγραμμα-αντικείμενο (μη βασική αναπαράσταση)</u>

```
p(X,Y):-q(X,Y).
p(X,Z) :-q(X,Y),p(Y,Z).
q(a,b).
q(b,c).
```

<u>Πρόγραμμα-αντικείμενο (βασική αναπαράσταση)</u>

```
clause(p(v(1),v(2)), [q(v(1),v(2))]).
clause(p(v(1),v(3)), [q(v(1),v(2)),p(v(2),v(3))]).
clause(q(a,b),[true]).
clause(q(b,c),[true]).
```

Παρατηρούμε ότι οι σταθερές a, b, c όπως στην μη βασική μορφή του προγράμματος, έτσι και εδώ αναπαρίστανται από σταθερές, όμως αυτή τη φορά είναι σταθερές της μετα-γλώσσας. Οι μεταβλητές X, Y, Z, παριστάνονται ως βασικοί όροι της μορφής v(N), όπου N θετικός ακέραιος αριθμός, δηλαδή v(1), v(2), v(3), αντίστοιχα.

Ένας άλλος τρόπος παράστασης του προγράμματος-αντικείμενο σε βασικούς όρους ο οποίος επιτρέπει δυναμικό χειρισμό του, είναι η παράστασή του σαν όρο του μετα-επιπέδου (μετα-προγράμματος).

Με αυτό τον τρόπο παράστασης τα ενσωματωμένα στην Prolog κατηγορήματα asserta/1, assertz/1 και retract/1, δεν χρειάζεται να χρησιμοποιηθούν για την αλλαγή του προγράμματος–αντικείμενο. Αυτός ο τρόπος παράστασης σε βασικούς όρους ονομάζεται **απεριόριστος**.

Ένας ακόμη τρόπος παράστασης του προγράμματος-αντικειμένου σαν όρος του μετα-προγράμματος (και τον οποίο τελικά επιλέξαμε) είναι ο ακόλουθος :

```
[
  [p(v(1),v(2)), q(v(1),v(2))],
  [p(v(3),v(5)), (v(3),v(4)), p(v(4),v(5))],
  [q(a,b)],
  [q(b,c)]
].
```

Δηλαδή, το πρόγραμμα αντικείμενο παριστάνεται σαν μια λίστα, κάθε στοιχείο της οποίας είναι επίσης μια λίστα, η οποία περιέχει μια πρόταση του προγράμματος-αντικειμένου. Σε αυτή την αναπαράσταση οι μεταβλητές του προγράμματος-αντικειμένου, παρίστανται ως v(1),v(2), κτλ. Όπως όμως έχει αναφερθεί, η εμβέλεια μιας μεταβλητής στην Polog, είναι ο κανόνας στον οποίο ανήκει. Μια μεταβλητή με την ίδια ονομασία σε διαφορετικούς κανόνες, δεν είναι η ίδια μεταβλητή. Για τον λόγο αυτό, κατά την κωδικοποίηση του προγράμματος-αντικειμένου σε βασικούς όρους με την παραπάνω μορφή, οι δείκτες των μεταβλητών σε κάθε λίστα-κανόνα διαφέρουν από τις μεταβλητές μιας άλλης λίστας-κανόνα, έστω κι αν οι μεταβλητές αυτές είναι ίδιες στην μη βασική μορφή.

Τέλος, το σύστημα που υλοποιήσαμε, παρέχει την δυνατότητα μετατροπής του μετασχηματισμένου προγράμματος-αντικείμενο από βασική σε μη βασική αναπαράσταση. Η δυνατότητα αυτή αποσκοπεί στην επεκτασιμότητα του συστήματος, με τρόπο τέτοιο ώστε τα προγράμματα που παράγονται να είναι άμεσα εκτελέσιμα από την ίδια την Prolog. Η αντίστροφη διαδικασία δεν παρέχεται και έτσι το πρόγραμμα αντικείμενο θα πρέπει να εισάγεται σε βασική αναπαράσταση με την μορφή που αναφέρεται παραπάνω.

4.2. Αντικατάσταση (substitution)

Η έννοια της αντικατάστασης, χρησιμοποιείται ευρέως στην διαδικασία μετασχηματισμού λογικών προγραμμάτων. Κάθε αντικατάσταση προκύπτει μέσα από την διαδικασία ταυτοποίησης και χρησιμοποιείται στους αλγόριθμους πτύξης (fold) και ανάπτυξης (unfold). Είναι φανερό λοιπόν ότι η αντικατάσταση είναι μια πολύ σημαντική έννοια στον μετασχηματισμό λογικών προγραμμάτων.

Ορισμός 4.2: Κάθε στοιχείο της μορφής X_i/T_i ονομάζεται **δέσμευση** του X_i.

Ορισμός 4.3: **Αντικατάσταση** (substitution) είναι ένα σύνολο δεσμεύσεων της μορφής $\{X_1/T_1,\ldots,X_k/T_k\}$ όπου κάθε X_i $(1 \le i \le k)$ είναι μια μεταβλητή διαφορετική από τις υπόλοιπες και κάθε T_i είναι ένας όρος διαφορετικός από την μεταβλητή X_i.

Ορισμός 4.4: **Βασική αντικατάσταση** (ground substitution) ονομάζεται η αντικατάσταση $\theta=\{X_1/T_1,\ldots,X_k/T_k\}$εφόσον όλα τα T_i $(1 \le i \le k)$ είναι βασικοί όροι.

Ορισμός 4.5: **Αντικατάσταση μετονομασίας** (renaming substitution) ονομάζεται η αντικατάσταση $\theta=\{X_1/T_1,\ldots,X_k/T_k\}$ της οποίας όλα τα T_i και X_i $(1 \le i \le k)$ είναι μεταβλητές. Τέλος, η αντικατάσταση θ={} ονομάζεται **κενή ή ταυτοτική αντικατάσταση**.

Για παράδειγμα, εάν a,b είναι σταθερές, X,Y,Z,W είναι μεταβλητές και f/1, g/2 είναι συναρτήσεις τότε η αντικατάσταση θ_1={X/f(a),Y/g(a,f(b))} είναι βασική αντικατάσταση ενώ η αντικατάσταση θ_2={X/Z,Y/W} είναι αντικατάσταση μετονομασίας.

Στο σύστημα που υλοποιήσαμε, η δέσμευση X/Y παριστάνεται ως subst(X,Y). Επιπλέον, η αντικατάσταση θ στο σύστημά μας παρουσιάζεται σαν μια λίστα δεσμεύσεων. Για παράδειγμα, η αντικατάσταση θ={X/f(a),Y/g(a,f(b))}, θα έχει την μορφή [subst(X,f(a)),subst(Y,g(a,f(b)))]. Για το ίδιο παράδειγμα, η βασική αντικατάσταση σύμφωνα με τα όσα έχουν αναφερθεί, θα είναι [subst(v(1),f(a)),subst(v(2),g(a,f(b)))].

4.3. Αλγόριθμος εφαρμογής αντικατάστασης (apply substitution)

Έστω Ε μια πρόταση και $\theta=\{x_1/t_1,\ldots,x_k/t_k\}$ μια αντικατάσταση. Ένα **στιγμιότυπο** (instance) $E\circ\theta$ του Ε λαμβάνεται εάν ταυτόχρονα αντικατασταθεί κάθε εμφάνιση του x_i στην Ε με t_i. Η διαδικασία αυτή ονομάζεται εφαρμογή αντικατάστασης.

Η πράξη αυτή, συμβολίζεται με $\circ$ και προκύπτει από την εφαρμογή των στοιχείων της αντικατάστασης στα δεξιά του συμβόλου, στον όρο που βρίσκεται αριστερά του.

Ορισμός 4.6: Πρέπει να αναφερθεί επίσης, ότι μια αντικατάσταση θ είναι **αυτοαπορροφητική** (indempotent) εάν κατά την εφαρμογή της θ στον «εαυτό» της, προκύψει η ίδια θ ($\theta=\theta\circ\theta$). Δηλαδή, για την αντικατάσταση $\theta=\{x_1/t_1,\ldots,x_k/t_k\}$ καμία μεταβλητή x_i $(1\leq i\leq k)$ δεν θα πρέπει να υπάρχει σε κάποιον από τους όρους $\{t_1,\ldots,t_k\}$.

Κάθε νόμιμη αντικατάσταση κατά τον αλγόριθμο της ταυτοποίησης, θα πρέπει να ικανοποιεί την ιδιότητα της αυτοαπορρόφησης. Για παράδειγμα, η αντικατάσταση $\theta=\{X/Y,Y/a\}$ δεν είναι αυτοαπορροφητική, καθώς η εφαρμογή της θ στην θ είναι $\{X/a,Y/a\}\neq\theta$.

Το κριτήριο της αυτοαπορρόφησης δεν είναι το μοναδικό χαρακτηριστικό των αντικαταστάσεων. Θεωρούμε τις αντικαταστάσεις $\theta_1,\theta_2,\theta_3$, την έκφραση Ε και τη κενή αντικατάσταση ϵ. Όλες οι αντικαταστάσεις πρέπει να ικανοποιούν τις ακόλουθες ιδιότητες:

1. $\theta_1\circ\epsilon = \epsilon\circ\theta_1=\theta_1$.
2. $(E\circ\theta_1)\circ\theta_2=E\circ(\theta_1\circ\theta_2)$.
3. $(\theta_1\circ\theta_2)\circ\theta_3=\theta_1\circ(\theta_2\circ\theta_3)$.
4. $E\circ\theta_1\circ\theta_2\neq E\circ\theta_2\circ\theta_1$.

Ο αλγόριθμος 4.1 είναι ο ψευδοκώδικας της διαδικασίας «εφαρμογή αντικατάστασης» που χρησιμοποιήθηκε για την υλοποίηση της αντίστοιχης διαδικασίας του συστήματος.

Procedure Εφαρμογή_αντικατάστασης
    Input: *Πρόγραμμα*, *Αντικατάσταση*
        Output: *Νέο_Πρόγραμμα*

Begin
  Νέο_Πρόγραμμα = **κενό**
  For each *Πρόταση* **in** *Πρόγραμμα*
    Begin
      Νέα_πρόταση = **κενή**
      For each *στοιχείο* **in** *Πρόταση*
        Begin

```
            Νέο_στοιχείο = κενό
            if στοιχείο είναι άτομο
                then Νέο_στοιχείο = στοιχείο
            if στοιχείο είναι μεταβλητή then
              begin
                for each δέσμευση in Αντικατάσταση
                  begin
                    if (πρώτο στοιχείο της δέσμευσης) = στοιχείο
        then Νέο_στοιχείο = αντικατάσταση του στοιχείου
                  end
              end
            if στοιχείο είναι όρος
             then κλήση procedure Εφαρμογή_αντικατάστασης
                                  (Input: στοιχείο,Αντικατάσταση),
                                          (Output: Νέο_στοιχείο)
            Νέα_πρόταση = {Νέα_πρόταση} ∪ {Νέο_στοιχείο}
        end
      Νέο_Πρόγραμμα ={Νέο_Πρόγραμμα} ∪ {Νέα_πρόταση}
    end
end
```

Αλγόριθμος 4.1 Εφαρμογή Αντικατάστασης

4.4. Σύνθεση αντικαταστάσεων (composition of substitutions)

Η σύνθεση αντικαταστάσεων είναι μια πολύ σημαντική λειτουργία κατά τον μετασχηματισμό λογικών προγραμμάτων, ιδιαίτερα κατά την πράξη της πτύξης. Το αποτέλεσμα αυτής της λειτουργίας είναι η εφαρμογή μιας αντικατάστασης σε κάποια άλλη, με σκοπό την σύνθεσή τους και τελικά την συγχώνευσή τους σε μια αντικατάσταση, παράγωγο των προηγούμενων δύο.

Σύμφωνα με όσα έχουν γραφεί προηγουμένως, θεωρούμε τις αντικαταστάσεις

$$s_1 = \{ X_1/U_1, \ldots, X_n/U_n\}$$

και

$$s_2 = \{Y_1/V_1, \ldots, Y_m/V_m\},$$

όπου U_i, V_j είναι όροι (terms) και X_i, Y_j μεταβλητές με i=1,…,n; j=1,…,m. Η σύνθεση του s_1 και του s_2 γράφεται $s_1 \circ s_2$.

Για την σύνθεση των παραπάνω αντικαταστάσεων s_1 και s_2, αρχικά εφαρμόζεται η αντικατάσταση s_2 στους όρους της s_1, τα αποτελέσματα της εφαρμογής ενώνονται με τα στοιχεία της s_2, οπότε προκύπτει:

$$\{X_1/U_1 \circ S_2, \ldots, X_n/U_n \circ S_2, Y_1/V_1, \ldots, Y_m, V_m\}$$

Στη συνέχεια, διαγράφονται όποια $X_k/U_k \circ S_2$ για τα οποία ισχύει: $X_k = U_k \circ S_2$, καθώς επίσης και όσα Y_i/V_i (i=1,…,n) για τα οποία είναι: $Y_i \in \{X_1, \ldots, X_n\}$.

Ως εφαρμογή των παραπάνω, παρουσιάζεται το παρακάτω παράδειγμα:

(a) Θεωρούμε τις αντικαταστάσεις

$$s_1 = \{Z/g(X, Y)\}$$

και

$$s_2 = \{X/a, Y/b, W/c, Z/d\},$$

τότε από την σύνθεσή τους προκύπτει:

$$s_1 \circ s_2 = \{Z/g(a, b), X/a, Y/b, W/c\}$$

(b) Ομοίως, οι αντικαταστάσεις

$$s_1 = \{X/f(Y), Y/Z\}$$

και

$$s_2 = \{X/john, Y/bill, Z/Y\}$$

οπότε η σύνθεσή τους έχει ως αποτέλεσμα:

$$s_1 \circ s_2 = \{X/f(bill), Y/bill, Z/Y\}$$

4.5. Αλγόριθμος ταυτοποίησης (Unify)

Ταυτοποίηση θα μπορούσε να χαρακτηριστεί η διαδικασία, κατά την οποία δημιουργείται μια αντικατάσταση θ, τέτοια ώστε για δύο απλές εκφράσεις E_1 και E_2 να ισχύει η σχέση $E_1\theta=E_2\theta$. Η αντικατάσταση αυτή, ονομάζεται εφαρμογή ταυτοποίησης.

Ορισμός 4.1: Μια αντικατάσταση θ_1 είναι **περισσότερο γενική** από μια αντικατάσταση θ_2 και συμβολίζεται $\theta_1 \geq \theta_2$ εάν υπάρχει αντικατάσταση θ_3 ώστε να ισχύει $\theta_2=\theta_1\circ\theta_3$. Δηλαδή, η αντικατάσταση $\theta_1=\{X/f(Z),Y/b\}$ είναι περισσότερο γενική από την αντικατάσταση $\theta_2=\{X/f(a),Y/b\}$ γιατί εάν $\theta_3=\{Z/a\}$ τότε $\theta_2=\theta_1\circ\theta_3$.

Για παράδειγμα εάν $E_1=p(X,Y)$ και $E_2=p(Z,a)$ τότε

- ένας ταυτοποιητής θ_1 των E_1 και E_2 είναι ο $\theta_1=\{X/b,Y/a,Z/b\}$ έτσι ώστε $E_1\theta_1=E_2\theta_1=p(b,a)$.

- Ένας άλλος ταυτοποιητής θα ήταν ο $\theta_2=\{X/a,Y/a,Z/a\}$ τέτοιος ώστε $E_1\theta_2=E_2\theta_2=p(a,a)$.

- Ομοίως, ταυτοποιητής είναι και ο $\theta_3=\{X/Z,Y/a\}$ για τον οποίο έχουμε $E_1\theta_3=E_2\theta_3=p(Z,a)$.

- Τέλος, ταυτοποιητής των E_1 και E_2 είναι και ο $\theta_4=\{X/f(W),Y/a,Z/f(W)\}$ για τον οποίο προκύπτει $E_1\theta_4=E_2\theta_4=p(f(W),a)$.

Ορισμός 4.2: Ένας ταυτοποιητής θ_1 των απλών εκφράσεων E_1 και E_2 ονομάζεται ο **πλέον γενικός ταυτοποιητής** (mgu) εάν για κάθε άλλο ταυτοποιητή θ_2 των E_1 και E_2 υπάρχει μια αντικατάσταση θ_3 τέτοια ώστε να ισχύει $\theta_2=\theta_1\theta_3$.

Για παράδειγμα, από τους παραπάνω ταυτοποιητές, θ_1, θ_2, θ_3, θ_4 ο ταυτοποιητής θ_3 είναι ο πλέον γενικός ταυτοποιητής των εκφράσεων E_1 και E_2 καθώς ισχύει:

α) $\theta_3\{Z/b\}=\{X/b,Y/a,Z/b\}=\theta_1$.
β) $\theta_3\{Z/a\}=\{X/a,Y/a,Z/a\}=\theta_2$.
γ) $\theta_3\{Z/f(W)\}=\{X/f(W),Y/a,Z/f(W)\}=\theta_4$.

Υπάρχουν πολλά διαφορετικά είδη αλγορίθμων ταυτοποίησης διαθέσιμα για τον υπολογισμό του πλέον γενικού ταυτοποιητή. Ο περισσότερο διαδεδομένος, είναι ο αλγόριθμος ταυτοποίησης Robinson κατά τον οποίο εάν δοθούν σαν είσοδος δύο ατομικοί τύποι $A_1 = P(t_1,\ldots,t_v)$ και $A_2 = P(r_1,\ldots,r_v)$ ο αλγόριθμος βρίσκει εάν είναι τυποποιήσιμοι ή όχι. Εάν ναι επιστρέφει τον πγτ, διαφορετικά επιστρέφει αποτυχία [3], [9]. Ο αλγόριθμος

χρησιμοποιεί μια στοιβάδα S στην οποία καταχωρούνται τα ζεύγη των αντίστοιχων όρων των A_1 και A_2, δηλαδή
S: = [$(t_1, r_1),\ldots,(t_v, r_v)$].

Όταν σε ένα ζεύγος όρων (t, r) είτε το t είναι μεταβλητή και το r σύνθετος όρος ή αντίστροφα τότε ο αλγόριθμος κάνει τον *έλεγχο–ύπαρξης* γνωστό ως *occur–check*. Αυτός ο έλεγχος σκοπό έχει να μην επιτρέψει αυτοαναφερόμενες δεσμεύσεις όπως για παράδειγμα X/f(X). Μία τέτοια δέσμευση καταχωρεί στην μεταβλητή X ένα μη-πεπερασμένο όρο f(f(f(…))) ενώ όλες οι εκφράσεις πρέπει να είναι πεπερασμένες. Επειδή ο *έλεγχος–ύπαρξης* έχει υψηλό υπολογιστικό κόστος, για αυτό πολλές γλώσσες λογικού προγραμματισμού παραλείπουν τον *έλεγχο–ύπαρξης* από τον αλγόριθμο ταυτοποίησης που χρησιμοποιούν. Αυτή η παράλειψη έχει σαν συνέπεια ο αλγόριθμος ταυτοποίησης να χάσει την ορθότητά του.

Είσοδος
- Οι δύο ατομικοί τύποι $A_1 = P(t_1,\ldots,t_v)$ και $A_2 = P(r_1,\ldots,r_v)$ οι οποίοι θα ταυτοποιηθούν.

Έξοδος
- Η αντικατάσταση θ, ο πλέον γενικός ταυτοποιητής των E_1 και E_2, ή αποτυχία.

Αλγόριθμος

- Αρχική τιμή στην αντικατάσταση θ το κενό σύνολο, θ := { }.
- Αρχική τιμή στη στοιβάδα S τα ζεύγη των όρων $(t_1, r_1),\ldots,(t_v,r_v)$,

S := [$(t_1, r_1),\ldots,(t_v, r_v)$].
- Αρχική τιμή ψευδές στην μεταβλητή Αποτυχία, Αποτυχία := ψευδής.

Repeat loop

Πάρε την κορυφή (t, r) της στοιβάδας S;
if t και r είναι διαφορετικές μεταβλητές **then**
θ′ = θ ∘ {t/r};
αντικατέστησε στην στοιβάδα S την μεταβλητή t με τον όρο r;
else if
t μεταβλητή *και* r όρος (σύνθετος ή μη) στον οποίο δεν υπάρχει η μεταβλητή t
then
θ′ = θ ∘ {t/r};
αντικατέστησε στην στοιβάδα S την μεταβλητή t με τον όρο r;
else if
r μεταβλητή *και* t όρος (σύνθετος ή μη) στον οποίο δεν υπάρχει η μεταβλητή r.
then
θ′ = θ ∘ {r/t};
αντικατέστησε στην στοιβάδα S την μεταβλητή r με τον όρο t;
else if

```
                t και r είναι ίδιες σταθερές ή ίδιες μεταβλητές
        then
                συνέχισε;
        else if
                t και r είναι σύνθετοι όροι με ίδιο όνομα συνάρτησης
                και ίδια πληθυκότητα κ, t = F(s1,…,sK) και
                r = F(u1,…,uK)
        then
                καταχώρησε στην στοιβάδα S τα ζεύγη
                των όρων (s1,u1),…,(sK, uK)
        else
                Αποτυχία : = αληθής
until ( S = { } ) or Αποτυχία;

if Αποτυχία then
        έξοδος αποτυχία
else
        έξοδος θ;
```

Αλγόριθμος 4.2 - Ταυτοποίηση

Ο αλγόριθμος 4.1, περιέχει στον κύκλο σύγκρισης, τον έλεγχο:

else if
r μεταβλητή *και* t όρος (σύνθετος ή μη) στον
οποίο δεν υπάρχει η μεταβλητή r.

Αυτός ο έλεγχος ονομάζεται «επαναληπτικός έλεγχος» (occur check) και στόχος του είναι να εμποδίσει τις αυτό-αναφερόμενες δεσμεύσεις της μορφής X/f(X). Για παράδειγμα στην περίπτωση που ο επαναληπτικός έλεγχος παραλειπόταν από τον αλγόριθμο

ταυτοποίησε τα άτομα p(f(X),f(f(X))) *και* p(W,W)

θα προέκυπτε προφανώς η δέσμευση X/f(X) η οποία δημιουργεί το ατέρμονο κατηγόρημα f(f(f…))), ενώ όλες οι εκφράσεις στην γλώσσα μας είναι πεπερασμένες.

Ο πειρασμός της παράληψης του επαναληπτικού ελέγχου από τον αλγόριθμο της ταυτοποίησης είναι πολύ ισχυρός, γνωρίζοντας την μεγάλη επεξεργαστική ισχύ που δαπανάται εαν συμπεριληφθεί. Άλλωστε, είναι ο μόνος έλεγχος στον κύκλο σύγκρισης ο οποίος πρέπει να εξετάσει εξονυχιστικά το εσωτερικό ενός κατηγορήματος, ενώ όλοι οι υπόλοιποι έλεγχοι απλά εξετάζουν μόνο τα κύρια (εξώτατα) σύμβολα του κατηγορήματος. Συνεπώς μέχρι τώρα, οι περισσότερες λογικές γλώσσες προγραμματισμού, αμελούσαν τον επαναληπτικό έλεγχο από τους αλγόριθμους ταυτοποίησής τους, κάτι που έχει ήδη αρχίσει να αλλάζει με την υφιστάμενη εξέλιξη στην υπολογιστική ισχύ.

Το τίμημα της παράβλεψης του ελέγχου επανάληψης, είναι η ενδεχόμενη απώλεια ορθότητας του αποτελέσματος. Για παράδειγμα, στο πρόβλημα αυτό θεωρούμε την εκτέλεση του στόχου

?- p(f(X),f(f(X)))

σε πρόγραμμα το οποίο περιέχει την πρόταση p(W,W). Αγνοώντας τον επαναληπτικό έλεγχο επιστρέφεται μια φαινομενική λύση με την υπολογισμένη απάντηση

p(f(X),f(f(X)))θ όπου θ={X/f(X)}

Η λύση προφανώς δεν είναι ορθή, παρότι ο παραπάνω στόχος και η πρόταση είναι αποδεκτές. Ένα απλό μοντέλο τους είναι το ακόλουθο:

έστω το πεδίο ορισμού των φυσικών αριθμών **N** = {0,1,2,…}
συσχέτισε το ‘f’ με την συνάρτηση $\mathbf{N} \to \mathbf{N}$ δίνοντας στο $X \in \mathbf{N}$, X+1
συσχέτισε το ‘p’ με την σχέση ισότητας από το $\mathbf{N} \times \mathbf{N}$.

Το παράδειγμα αυτό σε κώδικα Prolog, θα έχει την μορφή

```
f(X):-
        X1 is X+1, f(X1).
p(X):-
        f(X).
```

Προφανώς στο πρόγραμμα αυτό δεν υπάρχει λύση στον στόχο ?- p(Y). Ομοίως, η υπολογιζόμενη απάντηση θ={X/f(X)} στο προηγούμενο πρόγραμμα, δεν μπορεί να είναι ένα λογικό συμπέρασμα. Ακόμα χειρότερα, σε μια προσπάθεια παρουσίασης του αποτελέσματος η γλώσσα προγραμματισμού οδηγείται σε μια ατέρμονη ανακύκλωση, καθώς δεν υπάρχει επανάληψη n, τέτοια ώστε $\theta^n = \theta^{n+1}$.

Στην περίπτωση της τυπικής Prolog, (η οποία παραβλέπει τον έλεγχο), αναμένεται από τον προγραμματιστή να αναλάβει την ευθύνη για την αποφυγή προβλημάτων όπως παραπάνω. Κάποιες εκδόσεις της Prolog βέβαια, περιλαμβάνουν τον επαναληπτικό έλεγχο ως επιλογή που μπορεί να ενεργοποιηθεί από τον προγραμματιστή, ενώ άλλες παρέχουν ενσωματωμένα κατηγορήματα που μπορούν να επεμβαίνουν στο πρόγραμμα με σκοπό να εκτελέσουν έγκυρες ταυτοποιήσεις. Εκτός από αυτές τις δυνατότητες όμως, υπάρχουν και άλλες γλώσσες λογικού προγραμματισμού – όπως η Prolog II του Alain Colmerauer – οι οποίες είναι ειδικά κατασκευασμένες για τον σωστό χειρισμό των αυτό-αναφερόμενων σχέσεων , με στόχο την επίλυση προβλημάτων σε πεδία εκφραζόμενα από αφηρημένους κανόνες.

Η «επιβράδυνση» του συστήματος εξαιτίας του ελέγχου αυτού δεν είναι αισθητή στα σημερινά υπολογιστικά συστήματα. Έτσι, το τίμημα της επιπλέον υπολογιστικής δαπάνης και χρόνου, δεν είναι πια ιδιαίτερα

σημαντικό. Αντίθετα πολύ σημαντικό κριτήριο ενός συστήματος, είναι η ορθότητά και η ευκολία χειρισμού του. **Στο σύστημα που κληθήκαμε να υλοποιήσουμε, επιλέξαμε να συμπεριλάβουμε τον έλεγχο ύπαρξης στον αλγόριθμο ταυτοποίησης. Στόχος μας φυσικά ήταν να κάνουμε το σύστημά μας ορθό και περισσότερο λειτουργικό, απαλλάσσοντας τον χρήστη από την ευθύνη ελέγχου του προγράμματος πριν το εισάγει στο σύστημα.**

4.6. Αλγόριθμος μετονομασίας μεταβλητών (rename)

Η μετονομασία μεταβλητών είναι μια διαδικασία απαραίτητη για τον έγκυρο μετασχηματισμό του προγράμματος αντικειμένου. Προηγείται της εφαρμογής κάθε κανόνα μετασχηματισμού και είναι καθοριστική για την ορθότητα του αποτελέσματος.

Όπως έχει αναφερθεί, το πρόγραμμα αντικείμενο έχει επιλεγεί να αναπαρίσταται ως λίστα, τα στοιχεία της οποίας είναι οι προτάσεις, που και αυτές με την σειρά τους αναπαρίστανται ως λίστες. Πριν από την εφαρμογή ενός κανόνα μετασχηματισμού επομένως, θα έπρεπε είτε ο χρήστης, ή το σύστημα να λάβει τέτοια μέτρα ώστε οι αντικαταστάσεις που θα προκύπτουν ανα πάσα στιγμή κατά την εφαρμογή του κανόνα, να είναι έγκυρες. Αυτό επιτυγχάνεται όταν κατά τον μετασχηματισμό οι μεταβλητές της πρότασης στην οποία εφαρμόζεται κάθε φορά, δεν «συγκρούονται» με τις μεταβλητές της πρότασης που χρησιμοποιείται για την εφαρμογή του.

Στο σύστημά μας επιλέξαμε με γνώμονα την λειτουργικότητα, η διαδικασία μετονομασίας να είναι αυτόματη. Έτσι, πριν την εφαρμογή του κανόνα μετασχηματισμού ανάπτυξης, μετονομάζονται οι μεταβλητές όλου του προγράμματος αντικειμένου. Αντίθετα, κατά την εφαρμογή της πτύξης, μετονομάζονται οι μεταβλητές μόνο της πρότασης στην οποία θα γίνει η εφαρμογή του κανόνα μετασχηματισμού. Και στις δύο όμως περιπτώσεις η διαδικασία που ακολουθείται είναι η ίδια.

Αρχικά, σαρώνονται όλες οι μεταβλητές του προγράμματος αντικειμένου και του νέου ορισμού. Όπως έχει αναφερθεί οι μεταβλητές παριστάνονται ως v(X), όπου X θετικός ακέραιος αριθμός και δείκτης της μεταβλητής. Με την σάρωση αυτή λοιπόν, εντοπίζεται ο μέγιστος δείκτης μεταβλητής που υπάρχει στο πρόγραμμα. Γνωρίζοντας τον δείκτη αυτό, μπορούμε να μεταφέρουμε τις μεταβλητές μιας πρότασης πάνω από αυτον, εξασφαλίζοντας έτσι οτι οι δείκτες των μεταβλητών όπως προκύπτουν από αυτή την διαδικασία, δεν επαναλαμβάνονται σε άλλη πρόταση. Αυτό συνεπάγεται οτι οι δεσμεύσεις των αντικαταστάσεων που θα δημιουργηθούν αργότερα κατα την εφαρμογή κάποιου από τους κανόνες μετασχηματισμού πτύξης / ανάπτυξης, θα είναι έγκυρες.

Ο αλγόριθμος της διαδικασίας μετονομασίας όπως υλοποιήθηκε στο σύστημά μας, παρουσιάζεται παρακάτω:

Procedure μετονομασία(**Input:**_Πρόγραμμα_, **Output:**_Πρόγραμμα_)
    Begin
        Αριθμός=0
        For each _δείκτη_μεταβλητής_ **in** _Πρόγραμμα_
            Begin
                if _δείκτη_μεταβλητής>Αριθμός_
                        then _Αριθμός=δείκτη_μεταβλητής_
            end
        For each _δείκτη_μεταβλητής_ **in** _Πρόγραμμα_
            Begin

```
            δείκτη_μεταβλητής=(μέγιστος_δείκτης*2)+δείκτη_μεταβλητής
        end
    return Πρόγραμμα
end
```

Αλγόριθμος 4.3 Μετονομασία Μεταβλητών

5. Μετασχηματισμοί

Όπως προκύπτει από όσα έχουν αναφερθεί παραπάνω, είναι δυνατόν σε ένα σύστημα λογικού μετασχηματισμού προγραμμάτων, να υπάρχουν πολλές επαναλήψεις εφαρμογών κανόνων μετασχηματισμού. Επειδή οι κανόνες μετασχηματισμού που εφαρμόζονται είναι περισσότεροι του ενός, μας ενδιαφέρει εκτός από το μετασχηματισμένο νέο πρόγραμμα αντικείμενο, να έχουμε την δυνατότητα να παρακολουθούμε και τις εκδόσεις του που προκύπτουν μετά από κάθε εφαρμογή κανόνα. Έτσι, εισάγουμε μια νέα έννοια, την **μετασχηματιστική ακολουθία**.

Ορισμός 5.1: Έστω P_0 το αρχικό πρόγραμμα και D_0 το σύνολο των νέων ορισμών, το οποίο αρχικά είναι $D_0=\emptyset$. Μια **μετασχηματιστική ακολουθία** είναι η ακολουθία των ζευγαριών

$$(P_0,D_0),\ldots,(P_i,D_i),\ldots,(P_n,D_n)$$

όπου κάθε ζεύγος (P_i,D_i) και (P_{i+1},D_{i+1}), $0 \le i < n$, συνδέεται με την εφαρμογή ενός εκ των κανόνων μετασχηματισμού πτύξης, ανάπτυξης ή την εισαγωγή νέου ορισμού [6].

5.1. Νέος Ορισμός (New Definition)

Θεωρούμε ένα πρόγραμμα P, το οποίο περιγράφει μια διαδικασία επίλυσης κάποιου προβλήματος. Για κάποιο λόγο, υποθέτουμε ότι στο αρχικό πρόγραμμα πρέπει να προστεθεί ένας επιπλέον ορισμός που συμμετέχει στην επίλυση του προβλήματος. Η ανάγκη προσθήκης νέου ορισμού, μας οδηγεί στην διαδικασία μετασχηματισμού του αρχικού προγράμματος P με την βοήθεια του νέου κανόνα και των κατηγορημάτων του.

Ορισμός 5.2: Έστω ένα πρόγραμμα P και ένα σύνολο προτάσεων D το οποίο ονομάζουμε σύνολο νέων ορισμών. Έστω $S=\{p(\bar{t}_1)\leftarrow Q_1,\ldots,p(\bar{t}_n)\leftarrow Q_n\}$ όπου n προτάσεις, $\bar{t}_1, \ldots, \bar{t}_n$ πλειάδες n όρων και p ένα κατηγόρημα που δεν υπάρχει σε ένα από τα P, D ή $Q_1,\ldots,Q_n$. Τέλος, έστω ότι $P'=P\cup S$ και $D'=D\cup S$. Τότε τα P',D' προκύπτουν από τα P, D, με το μετασχηματισμό του **νέου ορισμού**.

Για παράδειγμα, έστω το αρχικό πρόγραμμα $P_0=\{C1, C2, C3\}$ και $D_0=\emptyset$.

```
C1:   subseq([],XS).
C2:   subseq([X|XS],[X|YS]) ←subseq(XS,YS).
C3:   subseq([X|XS],[Y|YS]) ←subseq([X|XS],YS).
```

Πρόγραμμα 5.1 Παράδειγμα νέου ορισμού

το οποίο εξετάζει εαν από τις λίστες L1 και L2 του subseq(L1, L2), η L1 υπάρχει στην L2. Συγκεκριμένα, στους στόχους που ακολουθούν λαμβάνουμε τις εξής απαντήσεις:

1. ?- subseq([a,b,c],[d,a,b,c]). η Prolog επιστρέφει yes.
2. ?- subseq([d,e,f],[a,b,c]). η Prolog επιστρέφει no.
3. ?- subseq([d,a,b,c],[a,b,c]). η Prolog επιστρέφει no.

Η εφαρμογή του κανόνα μετασχηματισμού του νέου ορισμού πραγματοποιείται όταν στο πρόγραμμα 5.1 προστεθεί η πρόταση

C4: csub(XS,YS,ZS) ← subseq(XS,YS), subseq(XS,ZS).

Από τον μετασχηματισμό αυτό προκύπτει το πρόγραμμα P_1 και ο νέος ορισμός D_1:

P1 = {C1,C2,C3,C4}, D1={C4}

Η προσθήκη νέου ορισμού εμπλουτίζει την διαδικασία επίλυσης του προβλήματος για το οποίο έχει σχεδιαστεί το πρόγραμμα. Στο παραπάνω παράδειγμα, η πρόταση C4 δείχνει ότι το csub(xs,ys,zs) είναι αληθές όταν το xs είναι ακολουθία του ys και του zs. Αυτό έχει ως αποτέλεσμα λοιπόν, το πρόγραμμα να παρουσιάζει περισσότερες επιλογές κατά την επίλυση ενός στόχου. Παράλληλα όμως, η προσθήκη αυτή συμβάλλει στην αύξηση του μεγέθους του προγράμματος, με αποτέλεσμα αυτό να γίνεται πιο αργό. Η μεταβολή αυτή στην δομή του προγράμματος, δημιουργεί την ανάγκη μετασχηματισμού του με βάση τον νέο ορισμό και με σκοπό την συγχώνευση του στο αρχικό πρόγραμμα. Είναι προφανές ότι το κεφάλι του νέου ορισμού δεν πρέπει να υπάρχει ήδη στο αρχικό πρόγραμμα, καθώς ο νέος ορισμός θα πρέπει να περιγράφει μια πραγματικά νέα σχέση. Αντίθετα, το σώμα του πρέπει να αποτελείται από κατηγορήματα που έχουν ήδη οριστεί στο πρόγραμμα, ώστε να επιτυγχάνεται η λογική συνοχή του με το αρχικό πρόγραμμα.

Στην πραγματικότητα, κάθε μετασχηματισμός πτύξης ή ανάπτυξης, είναι το αποτέλεσμα αλληλεπίδρασης του νέου ορισμού με τις προτάσεις του προγράμματος που ήδη υπάρχουν σε αυτό. Οι διαδικασίες αυτές, μετασχηματίζουν το αρχικό πρόγραμμα σε μια νεότερη μορφή του και ουσιαστικά ο νέος ορισμός «απορροφάται» από το πρόγραμμα, δίνοντάς του την μετασχηματισμένη μορφή P_i. Η εισαγωγή της έννοιας του νέου ορισμού, έχει ως φυσικό επακόλουθο λοιπόν την ερμηνεία των μετασχηματισμών πτύξης και ανάπτυξης, στις οποίες και θα αναφερθούμε στις ενότητες που ακολουθούν.

5.2. Κανόνας μετασχηματισμού ανάπτυξης (Unfold)

Η εφαρμογή του κανόνα μετασχηματισμού ανάπτυξης αναπτύσσει τις προτάσεις ενός προγράμματος με την βοήθεια μιας συγκεκριμένης πρότασης. Η εφαρμογή του κανόνα μετασχηματισμού ανάπτυξης, είναι αποδεδειγμένα ένας έγκυρος μετασχηματισμός [3], [15].

Η περιγραφή του κανόνα μετασχηματισμού ανάπτυξης όπως τον υλοποιήσαμε στο σύστημά μας είναι η εξής:

Ορισμός 5.4: Θεωρούμε ένα πρόγραμμα P. Επίσης θεωρούμε μια πρόταση C του προγράμματος P της μορφής

$$C: \quad A \leftarrow Q_1, B, Q_2$$

όπου A και B είναι ατομικοί τύποι και Q_1, Q_2 σύζευξη στοιχειωδών τύπων. Επιπλέον, θεωρούμε τις προτάσεις

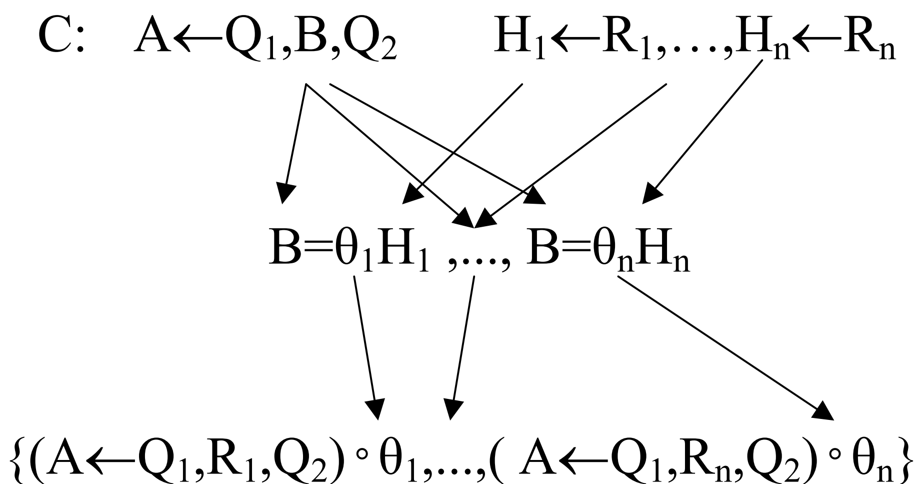
$$H_1 \leftarrow R_1, \ldots, H_n \leftarrow R_n$$

του προγράμματος P των οποίων οι κεφαλές μπορούν να ταυτοποιηθούν με το B. Έστω $\theta_1, \ldots, \theta_n$ οι αντίστοιχες αντικαταστάσεις μετά την ταυτοποίηση. Τότε, το αποτέλεσμα της ανάπτυξης του C στο B είναι το σύνολο των προτάσεων

$$\{(A \leftarrow Q_1, R_1, Q_2) \circ \theta_1, \ldots, (A \leftarrow Q_1, R_n, Q_2) \circ \theta_n\}$$

Τελικά το πρόγραμμα P_i μετασχηματίζεται σε P_{i+1} αντικαθιστώντας την πρόταση C από το παραπάνω σύνολο προτάσεων.

Σχηματικά, ο μετασχηματισμός ανάπτυξης μπορεί να παρασταθεί ως:



Σχήμα 5.1 Σχηματική παράσταση κανόνα μετασχηματισμού ανάπτυξης

Παράδειγμα: Σε συνέχεια του προγράμματος 5.1, εφαρμόζουμε τον κανόνα μετασχηματισμού ανάπτυξης, αναπτύσσοντας την πρόταση C4 στις προτάσεις του προγράμματος P1={C1, C2, C3, C4}, D1={C4}.

Η ταυτοποίηση του όρου subseq(XS,YS) της πρότασης C4 με κάθε ένα από τα κεφάλια των προτάσεων C1, C2, C3, επιστρέφει τα εξής:

θ1= unify(subseq(XS,YS),subseq([],XS1)={XS/[],YS/XS1}

θ2=unify(subseq(XS,YS),subseq([X2|XS2],[X2|YS2])=
{XS/[X2|XS2],YS/[X2|YS2]}

θ3= unify(subseq(XS,YS),subseq([X3|XS3],[Y3|YS3])=
{XS/[X3|XS3],YS/[Y3|YS3]}

Εφαρμόζοντας τις αντικαταστάσεις θ_1, θ_2 και θ_3 στις προτάσεις C1, C2 και C3, προκύπτουν αντίστοιχα

C5: (csub(XS,YS,ZS):-subseq(XS,ZS))θ1

με αποτέλεσμα

C5: csub([ ],YS,ZS)←subseq([ ],ZS).

C6: (csub(XS,YS,ZS):-subseq(XS2,YS2),subseq(XS,ZS))θ_2

με αποτέλεσμα

C6: csub([X|XS],[X|YS],ZS)←
subseq(XS,YS),subseq([X|XS],ZS).

C7: (csub(XS,YS,ZS):-subseq([X3|XS3],YS3),subseq(XS,ZS))θ_3

με αποτέλεσμα

C7: csub([X|XS],[Y|YS],ZS)←
subseq(([X|XS],YS),subseq([X|XS],ZS).

Το νέο πρόγραμμα τελικά είναι P2={C1, C2, C3, C5, C6, C7} και το σύνολο νέων ορισμών D2={C4}.

Συνοπτικά μπορούμε να αναφέρουμε ότι η ανάπτυξη ενός προγράμματος μειώνει τον χρόνο υπολογισμού. Ο λόγος είναι ότι αναπτύσσοντας ένα πρόγραμμα, αυξάνονται οι εμφανίσεις κάποιου κατηγορήματος στις προτάσεις από όπου θα μπορούσε να διερευνηθεί σε πιθανό ερώτημα και έτσι ο χρόνος αναζήτησης μειώνεται. Παράλληλα όμως, η ανάπτυξη του προγράμματος συνεπάγεται και αύξηση του όγκου του, με

αποτέλεσμα την απαίτηση μεγαλύτερης μνήμης για αποθήκευσή του από ότι θα απαιτούσε φυσιολογικά το αρχικό πρόγραμμα.

5.3. Κανόνας μετασχηματισμού πτύξης (Folding)

Πτύξη (folding) είναι μια τεχνική μετασχηματισμού προγραμμάτων η οποία είναι βασικά το αντίστροφο της ανάπτυξης (unfolding). Το αποτέλεσμα κάθε απλού βήματος πτύξης είναι η εξαγωγή ενός νέου κανόνα από το μέρος των υφιστάμενων κανόνων. Ο νέος αυτός κανόνας αντικαθιστά τον παλιότερο δημιουργόντας το καινούργιο πρόγραμμα.

Η θεωρητική προσέγγιση της διαδικασίας πτύξης, αντικατοπτρίζεται στον ορισμό από τους Tamaki – Sato [15]. Επιπλέον, έχει αποδειχτεί ότι η εφαρμογή του κανόνα μετασχηματισμού πτύξης, είναι ένας ορθός μετασχηματισμός [3].

Ο ορισμός της πτύξης περιγράφεται ως εξής:

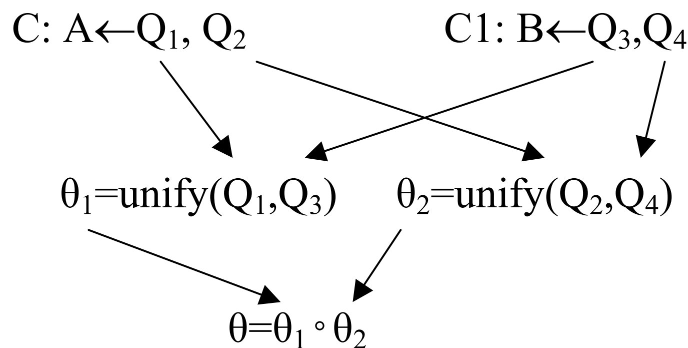
Ορισμός 5.3: Θεωρούμε P ένα πρόγραμμα και D το σύνολο των ορισμών που εισάγονται με το νέο ορισμό. Επιπλέον, θεωρούμε την πρόταση C του P της μορφής $A \leftarrow Q \circ \theta, Q_1$, όπου Q και Q_1 είναι σύζευξη στοιχειωδών τύπων. Ας θεωρήσουμε επίσης ότι C1 είναι μια πρόταση $B \leftarrow Q$ στο D, η οποία δεν είναι παραλλαγή του C. Τότε, το αποτέλεσμα της πτύξης του C χρησιμοποιόντας το C1 είναι η πρόταση C2 της μορφής:

C2: $A \leftarrow B \circ \theta, Q_1$

Επιπλέον αν η C2 αναπτυχθεί στο $B \circ \theta$ χρησιμοποιόντας τις προτάσεις του D, τότε το σύνολο {C} θα παραχθεί. Το πρόγραμμα P_i μετασχηματίζεται τελικά στο P_{i+1} μέσω πτύξης, αντικαθιστώντας την πρόταση C από την C2.

Σχηματικά ο κανόνας μετασχηματισμού πτύξης θα μπορούσε να παρασταθεί ως:

Έστω η πρόταση C του P της μορφής $A \leftarrow Q_1, Q_2$, και C1: $B \leftarrow Q_3, Q_4$ στο D.



Τελικά η νέα πρόταση είναι:

C2: $A \leftarrow B \circ \theta$

Σχήμα 5.2 Σχηματική παράσταση κανόνα μετασχηματισμού πτύξης

Στο παράδειγμα 5.1, το πρόγραμμα μας P2 μετά την εφαρμογή του κανόνα μετασχηματισμού ανάπτυξης είναι όπως φαίνεται στο πρόγραμμα 5.2

```
C1:    subseq([],XS).
C2:    subseq([X|XS],[Y|YS]) ←subseq(XS,YS).
C3:    subseq([X|XS],[Y|YS]) ←subseq([X|XS],YS).
C5:    csub([ ],YS,ZS)←subseq([ ],ZS).
C6:    csub([X|XS],[X|YS],ZS)←
                    subseq(XS,YS),subseq([X|XS],ZS).
C7:    csub([X|XS],[Y|YS],ZS)←
                    subseq(([X|XS],YS),subseq([X|XS],ZS).
```

Πρόγραμμα 5.2 Παράδειγμα πτύξης

Ένα παράδειγμα εφαρμογής του κανόνα μετασχηματισμού πτύξης στο πρόγραμμα 5.2 είναι η πτυξη της πρότασης C7 χρησιμοποιώντας την πρόταση C4 από το D2. Από τις ταυτοποιήσεις των δύο προτάσεων, προκύπτουν τα εξής:

θ1=unify(subseq([X|XS],YS),subseq(XS1,YS1))=
{XS1/[X|XS],YS/YS1}

και θ2=unify(subseq([X|XS],ZS),subseq(XS1,ZS1))={XS1/[X|XS],ZS/ZS1}

Η σύνθεσή των θ1 και θ2 μας δίνει:

θ1 ◦ θ2= θ= {XS1/[X|XS],Y/YS1,ZS/ZS1}

Η εφαρμογή της αντικατάστασης θ στο csub(XS1,YS1,ZS1) έχει ως αποτέλεσμα την καινούργια πρόταση C8:

C8: csub([X|XS],[Y|YS],ZS:-(csub(XS1,YS1,ZS1))θ

ή ισοδύναμα

C8: csub([X|XS],[Y|YS],ZS) ← csub([X|XS],YS,ZS).

Το νέο πρόγραμμα P3 είναι P3={C1, C2, C3, C5, C6, C8} και το σύνολο νέων ορισμών είναι D3={C4}.

6. Υλοποίηση συστήματος

Το σύστημα μετασχηματισμού που υλοποιήθηκε, έπρεπε οπωσδήποτε να είναι λειτουργικό, ευέλικτο και αποτελεσματικό. Ένα σύστημα θεωρείται λειτουργικό, όταν απλοποιεί τον τρόπο εργασίας του χρήστη, ευέλικτο όταν μπορεί να προσαρμόζεται σε κάθε πρόβλημα και αποτελεσματικό, όταν επιτυγχάνει τον θεμιτό στόχο με την μικρότερη δαπάνη χρόνου και υπολογιστικής ισχύος.

Κατά την υλοποίηση του συστήματός μας, κληθήκαμε να καλύψουμε εκτός από τα παραπάνω κριτήρια και τον τρόπο εισαγωγής και εξαγωγής των δεδομένων και των αποτελεσμάτων του συστήματος. Υλοποιήσαμε λοιπόν ένα στοιχειώδες γραφικό περιβάλλον, έτσι ώστε να είναι δυνατή η πλοήγηση του χρήστη ανάμεσα στις επιλογές και τις λειτουργίες του συστήματος με ταχύ και εύχρηστο τρόπο. Επιπλέον, για κάθε λειτουργία του συστήματος, επιλέξαμε να εμφανίζεται μια σύντομη περιγραφή για τον τρόπο με τον οποίο αυτή θα εκτελεστεί, τι θα θεωρηθεί δεδομένο, τι θα εξαχθεί από την διαδικασία αυτή καθώς και με ποιό ή ποιούς τρόπους μπορούμε να το χειριστούμε.

Για να επιτευχθούν τα παραπάνω, δημιουργήσαμε εκτός των κατηγορημάτων που αφορούν τις βασικές λειτουργίες του συστήματος μετασχηματισμού, όπως για παράδειγμα των διαδικασίων πτύξης και ανάπτυξης, κατηγορήματα δημιουργίας πλαισίων, μενού επιλογών και διάταξης των προτάσεων του προγράμματος σε μορφή αναγνώσιμη από τον χρήστη. Επίσης, η δυνατότητα εξαγωγής του μετασχηματισμένου πλέον προγράμματος αντικειμένου σε μη βασική μορφή, κάνει το σύστημα άκομα λειτουργικότερο, καθώς δίνεται η δυνατότητα εκτέλεσής του από την ίδια την Prolog.

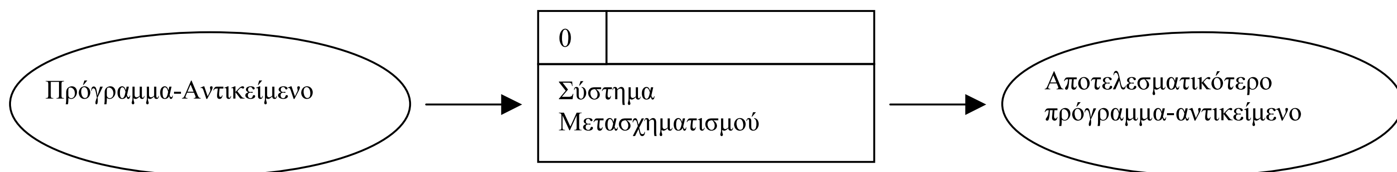
Για την απλοποίηση του χειρισμού του συστήματος, επιλέξαμε να παρέχεται η δυνατότητα στον χρήστη να εισάγει το πρόγραμμα αντικείμενο εκτός από την διαδικασία πληκτρολόγησης, και μέσω αποθηκευμένου αρχείου. Το αρχείο αυτό φυσικά θα πρέπει να περιέχει το πρόγραμμα αντικείμενο, δομημένο σε βασική αναπαράσταση. Η λειτουργία αυτή θεωρείται πολύτιμη στις περιπτώσεις όπου το πρόγραμμα αντικείμενο είναι είτε πολύπλοκο στην βασική του μορφή, είτε περιέχει πολυπληθείς μεταβλητές, γεγονός που το καθιστά δυσανάγνωστο. Επίσης, η λειτουργία αυτή, παρέχει την δυνατότητα αυτοματοποίησης ενός ευρύτερου συστήματος, προσθέτοντας μια διαδικασία μετατροπής ενός προγράμματος αντικειμένου από μη βασική σε βασική μορφή, και έτσι, να δοθεί η δυνατότητα μετασχηματισμού προγραμμάτων Prolog, χωρίς την ουσιαστική επέμβαση του χρήστη.

Η εκτέλεση των λειτουργιών, έχει διαλογικό χαρακτήρα, με συνεχή αλληλεπίδραση χρήστη και συστήματος. Μέσα από διαδοχικά μενού επιλογών, και με την δυνατότητα ανα πάσα στιγμή εμφάνισης βοήθειας για κάθε λειτουργία και επιλογή, ο χρήστης μπορεί να χειριστεί το σύστημα αυτό, με την λειτουργικότητα του απλού εργαλείου.

Μετά από κάθε επεξεργασία του προγράμματος αντικειμένου, το σύστημα ενημερώνει τον χρήστη για το αποτέλεσμα εμφανίζοντας παράλληλα σε διαδοχικά βήματα την διαδικασία μετατροπής του. Για παράδειγμα, σε επιλογή του χρήστη εφαρμογής στο πρόγραμμα αντικείμενο τον μετασχηματισμό ανάπτυξης, το πρόγραμμα αρχικά θα ζητήσει το πρόγραμμα αντικείμενο σε βασική μορφή από την χρήστη. Όπως αναφερθήκαμε προηγουμένως, αυτό μπορεί να είναι είτε αποθηκευμένο σε αρχείο, είτε καταχωρημένο από προηγούμενους μετασχηματισμούς, είτε ένα νέο το οποίο καταχωρείται σε αυτό το στάδιο από τον χρήστη. Στη συνέχεια το σύστημα θα ζητήσει τον νέο ορισμό με βάση τον οποίο θα εφαρμοστεί η διαδικασία ανάπτυξης καθώς και το κατηγόρημα του ορισμού αυτού, το οποίο θα χρησιμοποιηθεί στο τμήμα ταυτοποίησης μεταξύ του νέου ορισμού και των προτάσεων του προγράμματος αντικειμένου. Μετά από αυτό το σημείο, η διαδικασία μετασχηματισμού αναλαμβάνεται πλήρως από το σύστημα, το οποίο ακολουθεί την δεδομένη μεθοδολογία του μετασχηματισμού. Στο τμήμα αυτό, κάθε βήμα εκτέλεσης περιγράφεται από το σύστημα. Αναφέρεται ποια πρόταση χειρίζεται την εκάστοτε στιγμή, ποια κατηγορήματα ταυτοποιούνται και ποια αποτυγχάνουν, εμφανίζονται οι αντικαταστάσεις όπως αυτές προκύπτουν μετά την ταυτοποίηση, και τελικά το νέο πρόγραμμα. Στο σημείο αυτό, δίνεται η δυνατότητα καταχώρησης και κατα συνέπεια αντικατάστασης του αρχικού προγράμματος αντικειμένου από αυτό που μόλις έχει δημιουργηθεί από το σύστημα, γεγονός που δίνει την δυνατότητα εφαρμογής περισσότερων του ενός μετασχηματισμών στο πρόγραμμα αντικείμενο που εμείς αρχικά καταχωρήσαμε. Τέλος, η αναλυτική περιγραφή των διαδικασιών του κάθε μετασχηματισμού, κάθως επίσης και η εμφάνιση των αποτελεσμάτων στα διακριτά βήματα, δίνει την δυνατότητα ελέγχου και επαλήθευσης των αποτελεσμάτων του συστήματος ανά πάσα στιγμή, και έτσι αποφεύγονται εσφαλμένα αποτελέσματα.

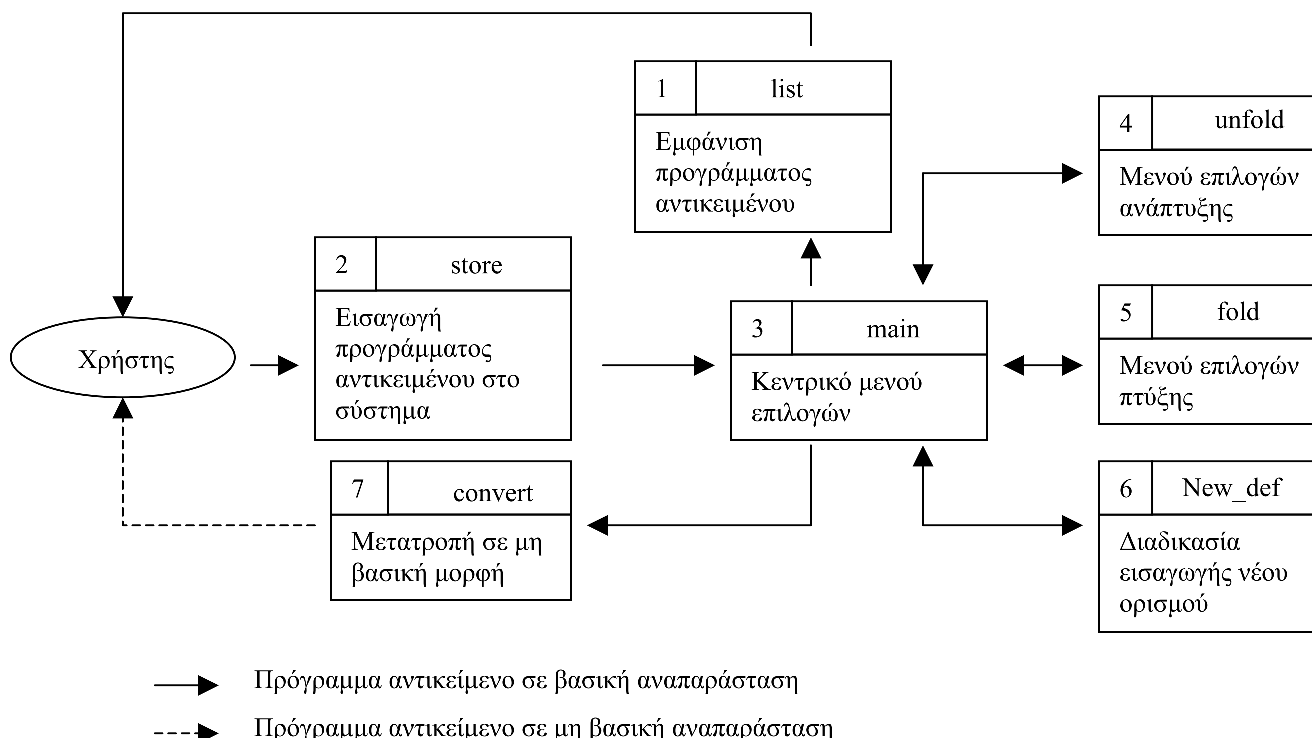
Η ευελιξία του συστήματος, οφείλεται στην ίδια την Prolog, η οποία παρέχει την δυνατότητα υλοποίησης τέτοιων συστημάτων, κυριώς λόγω του τρόπου λειτουργίας της. Είναι γεγονός, οτι η υλοποίηση ενός τέτοιου συστήματος σε οποιαδήποτε άλλη γλώσσα προγραμματισμού, θα ήταν τρομερά δύσκολη. Τέλος, η αποτελεσματικότητά του, ορίζεται από την αποτελεσματικότητα των μεθόδων μετασχηματισμού που επιλέξαμε να εφαρμόσουμε. Δηλαδή, από τον αλγόριθμο ταυτοποίησης, πτύξης, ανάπτυξης και αντικατάστασης. Τα κατηγορήματα που αφορούν την μορφοποίηση και παρουσίαση του αποτελέσματος στον χρήστη, δεν επηρρεάζουν την αποτελεσματικότητα του συστήματος, ούτε και επιβραδύνουν την εκτέλεσή του, γεγονός που τα καθιστά απαραίτητες προσθήκες.

Το σύστημα μετασχηματισμού στην πιο γενική του μορφή εκφράζεται από το διάγραμμα 6.1:



Διάγραμμα 6.1 Το σύστημα μετασχηματισμού

Πρέπει όμως να αναφερθούμε αναλυτικότερα στις διαδικασίες που συγκροτούν το σύστημα μετασχηματισμού, έτσι είναι απαραίτητο το διάγραμμα 6.2:



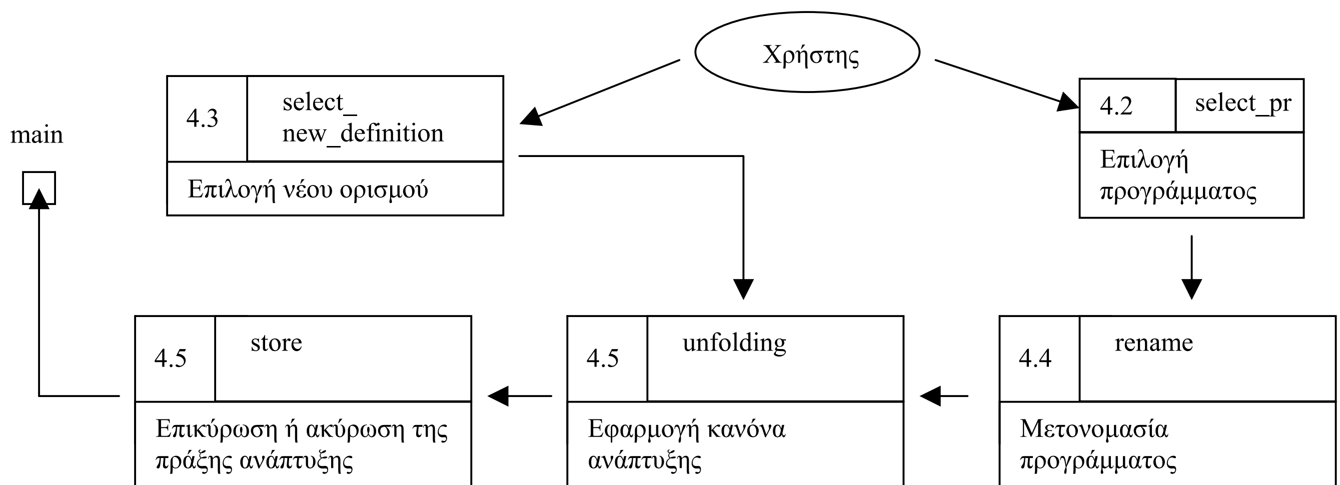
Διάγραμμα 6.2 Το σύστημα σε διαδικασίες

Τα πλαίσια 1, 2 και 7, αντιπροσωπεύουν διαδικασίες διαχείρισης του προγράμματος αντικειμένου. Συγκεκριμένα, η διαδικασία 1 επιστρέφει στην οθόνη του χρήστη το πρόγραμμα αντικείμενο σε βασική αναπαράσταση. Η διαδικασία 2, δίνει την δυνατότητα στον χρήστη να εισάγει στο σύστημα το πρόγραμμα αντικείμενο, ενώ τελικά η διαδικασία 7, αναλαμβάνει να μετατρέψει το πρόγραμμα αυτό από βασική σε μη βασική αναπαράσταση. Αντίθετα με τις διαδικασίες 1, 2 και 6, η διαδικασία στο πλαίσιο 3 είναι ο πυρήνας του συστήματος, καθώς μέσω αυτής ο χρήστης επιλέγει τον κανόνα μετασχηματισμού που εφαρμόζεται μέσα από τις διαδικασίες 4, 5 και 6.

Οι διαδικασίες 4, 5 αξιοποιούν τους αλγόριθμους που έχουν αναφερθεί προηγουμένως. Εκτελούν δηλαδή πράξεις ταυτοποίησης,

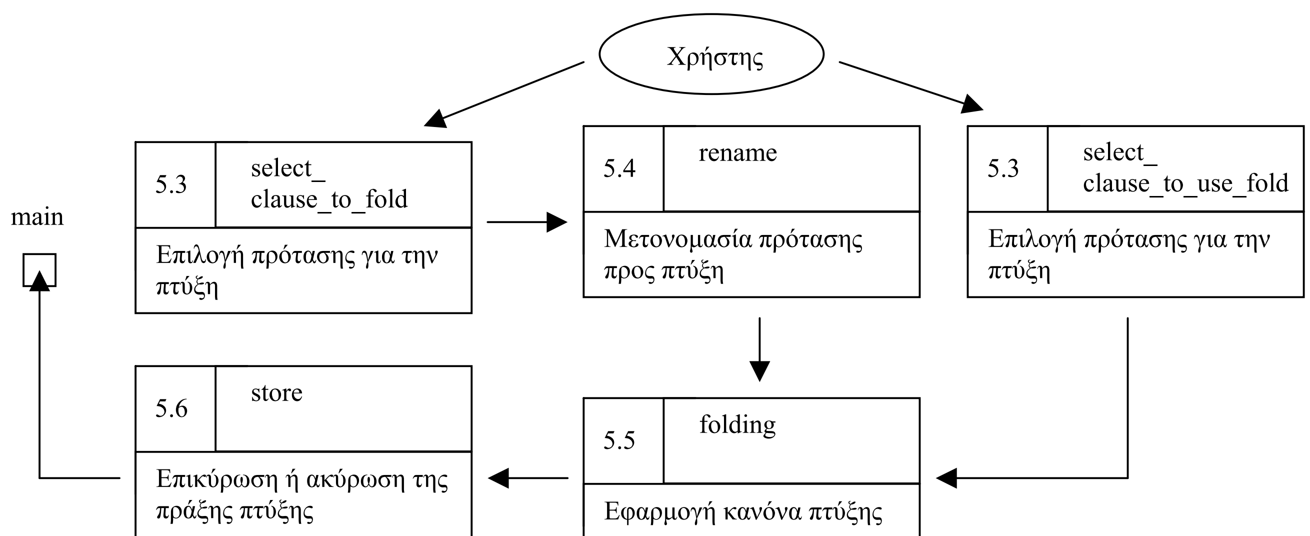
μετονομασίας, σύνθεσης και εφαρμογής αντικαταστάσεων, με σκοπό την τήρηση και εφαρμογή των ορισμών που έχουν δωθεί για τους αντίστοιχους κανόνες μετασχηματισμού. Με το τέλος των διαδικασιών 4 και 5, το πρόγραμμα έχει επιστραφεί στο μεντρικό μενού επιλογών (3) και το σύστημα περιμένει την επόμενη επιλογή του χρήστη.

Συγκεκριμένα, για την διαδικασία εφαρμογής του κανόνα μετασχηματισμού ανάπτυξης παρουσιάζεται το διάγραμμα 6.3



Διάγραμμα 6.3 Μενού Ανάπτυξης

Η διαδικασία εφαρμογής του κανόνα μετασχηματισμού πτύξης φαίνεται στο διάγραμμα 6.4

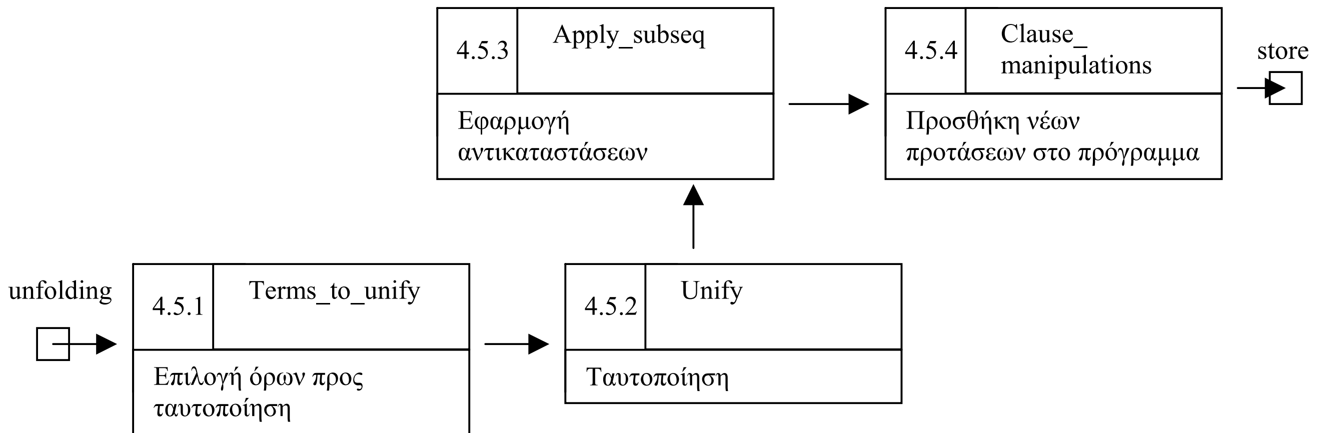


Διάγραμμα 6.4 Μενού Πτύξης

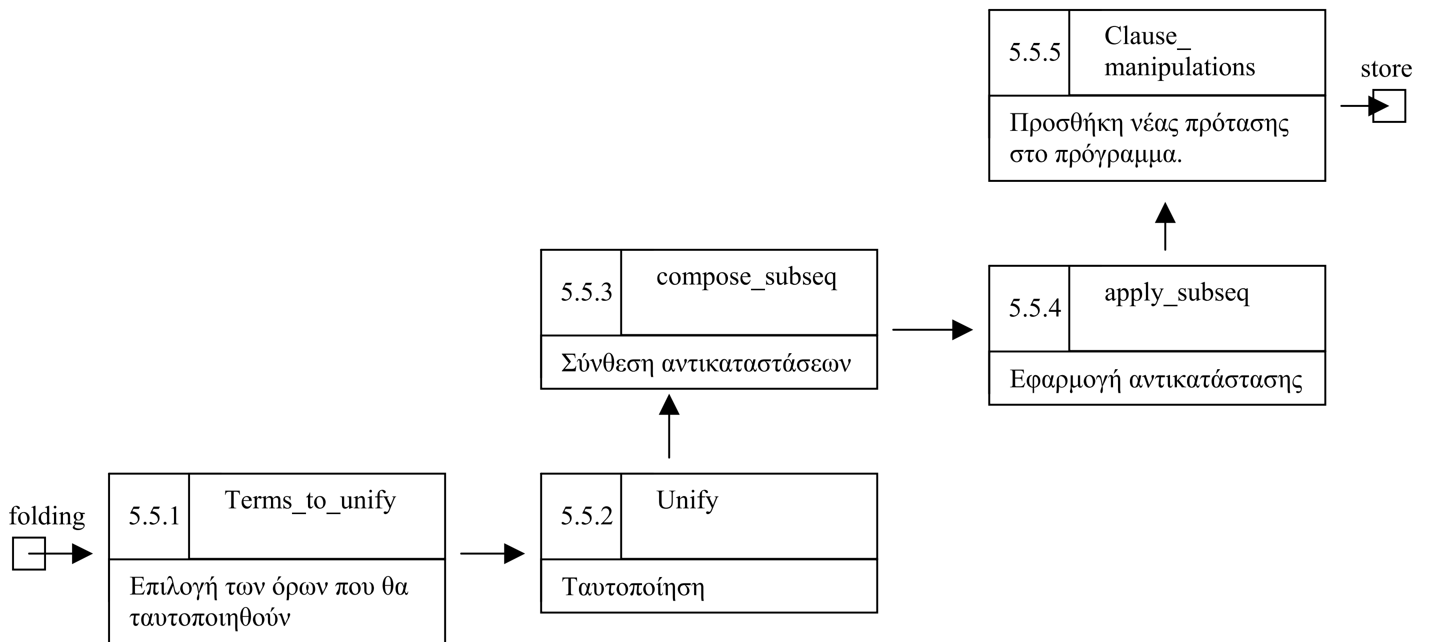
Αξίζει να σημειωθεί ότι η αρίθμηση των διαδικασιών αναφέρεται στο επίπεδο που βρίσκονται και στην ιεραρχία τους. Δηλαδή, η 4.1 (start) είναι μια από τις διαδικασίες που περιγράφουν την διαδικασία 4 (unfold) του διαγράμματος 6.2.

Προφανώς, η διαδικασία 5.1 (start) δεν έχει σχέση με την 4.1, καθώς αυτή αναφέρεται στην διαδικασία 5 (fold) του διαγράμματος 6.2.

Μπορούν να αναλυθούν περισσότερο οι διαδικασίες 4.5 και 5.5 (unfolding, folding) των διαγραμμάτων 6.3 και 6.4 αντίστοιχα. Η διαδικασία 4.5 αναλύεται στο διάγραμμα 6.5 και η διαδικασία 5.5 στο διάγραμμα 6.6.



Διάγραμμα 6.5 Διαδικασία ανάπτυξης



Διάγραμμα 6.6 Διαδικασία πτύξης

7. Εφαρμογή του συστήματος – Ένα παράδειγμα

Για την ανάγκη της ανάλυσης του συστήματος, θεωρούμε το αρχικό πρόγραμμα P0 = {C1, C2, C3} και το σύνολο νέων ορισμών D=∅.

C1: subseq([],XS).
C2: subseq([X|XS],[Y|YS]) ←subseq(XS,YS).
C3: subseq([X|XS],[Y|YS]) ←subseq([X|XS],YS).

Πρόγραμμα 7.1 Αρχικό πρόγραμμα αντικείμενο

Το πρόγραμμα αυτό εξετάζει αν μια λίστα είναι υποσύνολο μιας άλλης [6].

Βήμα 1^{ο}: Νέος ορισμός (New Definition)

Εισάγουμε τον νέο ορισμό C4, παρακινούμενοι από κάποια ανάγκη για μια γενική σχέση των subsequence

C4: csub(XS,YS,ZS) ← subseq(XS,YS), subseq(XS,ZS)

Το πρόγραμμα και ο νέος ορισμός τότε είναι:

P1 = {C1,C2,C3,C4}, D1={C4}

Βήμα 2^{ο}: Ανάπτυξη (Unfolding)

Αναπτύσσουμε τον κανόνα C4 στις προτάσεις του αρχικού προγράμματος. Ορίζουμε κατά την ταυτοποίηση με τις κεφαλές των κανόνων να χρησιμοποιείται το κατηγόρημα subseq(XS,YS) του νέου ορισμού C4. Από την ταυτοποίηση θα έχουμε για κάθε πρόταση του προγράμματος

θ1= unify(subseq(XS,YS),subseq([],XS1)= {XS/[],YS/XS1}

θ2=unify(subseq(XS,YS),subseq([X2|XS2],[X2|YS2])=
{XS/[X2|XS2],YS/[X2|YS2]}

θ3= unify(subseq(XS,YS),subseq([X3|XS3],[Y3|YS3])=
{XS/[X3|XS3],YS/[Y3|YS3]}

Οι προτάσεις C5, C6 και C7 όπως προκύπτουν από τις αντίστοιχες C1, C2, C3 παρουσιάζονται παρακάτω:

C5: (csub(XS,YS,ZS) ←subseq(XS,ZS))θ1 =>

csub([ ],YS,ZS)←subseq([ ],ZS).

C6: (csub(XS,YS,ZS) ←subseq(XS2,YS2),subseq(XS,ZS))θ_2 =>

csub([X|XS],[X|YS],ZS)←
subseq(XS,YS),subseq([X|XS],ZS).

C7: (csub(XS,YS,ZS) ←subseq([X3|XS3],YS3),subseq(XS,ZS))θ_3 =>

csub([X|XS],[Y|YS],ZS)←
subseq(([X|XS],YS),subseq([X|XS],ZS).

Το νέο πρόγραμμα τελικά είναι P2={C1, C2, C3, C5, C6, C7} και το σύνολο νέων ορισμών D2={C4}.

Βήμα 3ο: Ανάπτυξη (Unfolding)

Στη συνέχεια υποθέτουμε οτι θέλουμε να αναπτύξουμε την πρόταση C5 στο πρόγραμμα. Στην περίπτωση αυτή, η ταυτοποίηση επιτυγχάνει μόνο με το κεφάλι της πρότασης C1, οπότε θα έχουμε:

C8: csub([],YS,ZS).

Το νέο πρόγραμμα τελικά είναι P3={C1, C2, C3, C6, C7, C8} και το σύνολο νέων ορισμών D3={C4}.

Βήμα 4ο: Πτύξη (folding)

Συνεχίζοντας την διαδικασία μετασχηματισμού στο πρόγραμμα P2={C1, C2, C3, C6, C7, C8}, D2={C4}, υποθέτουμε ότι εκτελούμε την πτύξη του C7 χρησιμοποιώντας το C4.
Δηλαδή τον κανόνα

C7: csub([X|XS],[Y|YS],ZS)←
subseq(([X|XS],YS),subseq([X|XS],ZS).

με τον νέο ορισμό

C4: csub(XS,YS,ZS) ← subseq(XS,YS), subseq(XS,ZS).

Σύμφωνα με όσα έχουν αναφερθεί σε προηγούμενες ενότητες, επιλέγονται σε αυτό το στάδιο να ταυτοποιηθούν τα subseq(([X|XS],YS) (από τον C7) και subseq(XS1,YS1) (από τον C4) δίδοντας τα

θ1=unify(subseq([X|XS],YS),subseq(XS1,YS1))= {XS1/[X|XS],YS/YS1}

και

θ2=unify(subseq([X|XS],ZS),subseq(XS1,ZS1))= {XS1/[X|XS],ZS/ZS1}

Η σύνθεσή των θ1 και θ2 μας δίνει:

$\theta 1 \circ \theta 2 = \theta = \{XS1/[X|XS], Y/YS1, ZS/ZS1\}$

Με την εφαρμογή της αντικατάστασης θ στο csub(XS1,YS1,ZS1) προκύπτει η πρόταση C9:

C9: csub([X|XS],[Y|YS],ZS)← (csub(XS1,YS1,ZS1))θ =>

csub([X|XS],[Y|YS],ZS) ← csub([X|XS],YS,ZS).

Το νέο πρόγραμμα P4 είναι P4={C1, C2, C3, C6, C8, C9} και το σύνολο νέων ορισμών είναι D4={C4}.

Βήμα 5°: Νέος Ορισμός (New Definition)

Υποθέτουμε οτι θέλουμε να εισάγουμε μια νέα πρόταση στο πρόγραμμα, της οποίας το σώμα είναι το σώμα της πρότασης C6. Δηλαδή την πρόταση

C10: csub1(X,XS,YS,ZS) ←
subseq(XS,YS), subseq([X|XS],ZS).

Όπως προηγουμένως, η εφαρμογή του κανόνα μετασχηματισμού νέου ορισμού θα έχει ως αποτέλεσμα

P5={C1, C2, C3, C6, C8, C9, C10}, D5={C4, C10}.

Βήμα 6°: Πτύξη (Folding)

Στη συνέχεια αποφασίζουμε να εφαρμόσουμε τον κανόνα μετασχηματισμού πτύξης στην πρόταση C6 χρησιμοποιόντας την πρόταση C10. Η νέα πρόταση που προκύπτει τότε είναι

C11: csub([X|XS],[X|YS],ZS) ←
csub1(X,XS,YS,ZS).

Το νέο πρόγραμμα μετά από την εφαρμογή αυτού του κανόνα μετασχηματισμού είναι P6={C1, C2, C3, C8, C9, C10, C11}, D6={C4, C10}

Βήμα 7°: Ανάπτυξη (Unfolding)

Στο σημείο αυτό, αναπτύσσουμε την πρόταση C10 χρησιμοποιώντας τον δεύτερο στοιχειώδη τύπο του σώματός της, δηλαδή το subseq([X|XS],ZS). Από τον μετασχηματισμό αυτό, προκύπτουν οι προτάσεις:

```
C12:    csub1(X,XS,YS,[X|ZS]) ←
                subseq(XS,YS),subseq(XS,ZS).
C13:    csub1(X,XS,YS,[Z|ZS]) ←
                subseq(XS,YS),subseq([X|XS],ZS).
```

Το νέο πρόγραμμα και το σύνολο νέων ορισμών σε αυτό το στάδιο γίνεται P7={C1, C2, C3, C8, C9, C11, C12, C13}, D7={C4, C10}.

Βήμα 8ο: Πτύξη (Folding)

Τέλος, εφαρμόζουμε τον κανόνα μετασχηματισμού πτύξης στην πρόταση C13 χρησιμοποιώντας την C10. Προκύπτει τελικά η πρόταση:

```
C15:    csub1(X,XS,YS,[Z|ZS]) ←
                csub1(X,XS,YS,ZS).
```

Το νέο πρόγραμμα μετά τον μετασχηματισμό πτύξης, είναι P8={C1,C2,C3,C8,C9,C11,C14,C15} και το σύνολο νέων ορισμών είναι: D8={C4, C10}.

Το τελικό πρόγραμμα επομένως είναι:

```
C1:     subseq([],XS).
C2:     subseq([X|XS],[Y|YS]) ←
                        subseq(XS,YS).
C3:     subseq([X|XS],[Y|YS]) ←
                        subseq([X|XS],YS).
C8:     csub([],YS,ZS).
C9:     csub([X|XS],[Y|YS],ZS) ←
                        csub([X|XS],YS,ZS).
C11:    csub([X|XS],[X|YS],ZS) ←
                        csub1(X,XS,YS,ZS).
C14:    csub1(X,XS,YS,[X|ZS]) ←
                        csub(XS,YS,ZS).
C15:    csub1(X,XS,YS,[Z|ZS]) ←
                        csub1(X,XS,YS,ZS).
```

Πρόγραμμα 7.2 Τελικό πρόγραμμα αντικείμενο

Η εφαρμογή του παραπάνω παραδείγματος στο σύστημα μας, παρουσιάζεται από τα παρακάτω στιγμιότυπα:

1) Με την εκκίνηση του συστήματος, εμφανίζεται το κεντρικό μενού επιλογών

```
                    TEI of Crete
        Department of Electrical Engineering
         Diploma work on Meta-programming
             by George Santipantakis
                   07/11/2002
                                                  ver(1.09)

         Main Menu
 * (u) or unfold. ______to enter unfold menu
 * (f) or fold. _______to enter fold menu
 * n_definition. ________to import a new definition
 *       about. ________to read information about this project
 *       hlp. __________for help
 *       halt. _________to quit
Ready
|: _
```

2) Το κατηγόρημα *hlp*, εμφανίζει το μενού βοήθειας των εντολών του συστήματος.

```
Ready
|: hlp.
  Help: type hlp to get this message
  Existing Predicates:
        sort.
        unify(T1,T2,Subst).
        apply(Clauses,Theta,New_Clauses).
        unfold.
        fold.
        n_definition.
        main.
        store.
        list.
        ver.
        input_fl(X).
        store_fl(X).
        convert.
        help_on(Topic), where Topic can be the name of one of
        the above terms.
Ready
|:
```

3) Αναλυτικότερη βοήθεια για κάθε κατηγόρημα ξεχωριστά, δίνεται με το κατηγόρημα *help_on(Topic)*. Για παράδειγμα, γράφοντας *help_on(apply)* και *help_on(unfold)*, το σύστημα επιστρέφει

```
        help_on(Topic), where Topic can be the name of one of
        the above terms.

Ready
|: help_on(apply).
        apply(Clauses,Theta,New_Clauses).

applies list of substitutions Theta on list of clauses Clauses to
derive New_Clauses
e.g. apply([a(c,v(1))],[subst(v(1),b)],New_Clauses). will return
 New_Clauses=[a(c,b)]

Ready
|: help_on(unfold).
                    unfold (or u).

      Unfold is a program transformation technique,
 as applied by Tamaki - Sato. Transforms a logic program
 into a new one with more clauses.
 Type unfold (or u) to enter unfold menu

Ready
|: _
```

4) Πρέπει να αναφερθεί, οτι είναι δυνατή η καταχώρηση ενος προγράμματος-αντικειμένου σε βασική μορφή στο σύστημα με την βοήθεια του κατηγορήματος *store*, ώστε ο χρήστης να μην υποχρεώνεται να εισάγει το πρόγραμμα αυτό κάθε φορά πριν από ένα μετασχηματισμό.

```
        apply(Clauses,Theta,New_Clauses).

applies list of substitutions Theta on list of clauses Clauses to
derive New_Clauses
e.g. apply([a(c,v(1))],[subst(v(1),b)],New_Clauses). will return
 New_Clauses=[a(c,b)]

Ready
|: help_on(unfold).
                    unfold (or u).

      Unfold is a program transformation technique,
 as applied by Tamaki - Sato. Transforms a logic program
 into a new one with more clauses.
 Type unfold (or u) to enter unfold menu

Ready
|: store.
type program to be stored:
|: [[subseq(nil,v(1))],[subseq(.(v(2),v(3)),.(v(2),v(4))),subseq(v(3),v(4))],[su
bseq(.(v(5),v(6)),.(v(7),v(8))),subseq(.(v(5),v(6)),v(8))]].

Ready
|:
```

5) Στη συνέχεια εκτελούμε έναν μετασχηματισμό νέου ορισμού για την πρόταση C4 του παραδείγματός μας γράφοντας *n_definition* και στην συνέχεια την νέα πρόταση,

```
type program to be stored:
|: [[subseq(nil,v(1))],[subseq(.(v(2),v(3)),.(v(2),v(4))),subseq(v(3),v(4))],[su
bseq(.(v(5),v(6)),.(v(7),v(8))),subseq(.(v(5),v(6)),v(8))]].

Ready
|: n_definition.

New Definition Input

Type the new definition you wish to add, or type cancel to abort

|: [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))].
Process completed

          Main Menu

 * (u) or unfold. ______to enter unfold menu
 * (f) or fold. _______to enter fold menu
 * n_definition. _______to import a new definition
 *       about. ________to read information about this project
 *       hlp. _________for help
 *       halt. ________to quit

Ready
|:
```

6) Για να ακολουθήσει ο μετασχηματισμός ανάπτυξης γράφουμε *unfold* και για την εκκίνησή του, *start*.

```
 * (f) or fold. _______to enter fold menu
 * n_definition. _______to import a new definition
 *       about. ________to read information about this project
 *       hlp. _________for help
 *       halt. ________to quit

Ready
|: unfold.

          Unfold Menu

 *        start. _______to start unfold process
 *        store. _______to store a new program
 *        list. ________to display the program
 *        exit. ________to return to main menu

(Unfold)
Ready
|: start.

Starting unfold process...

Enter program (type "stored" to use stored program):

|:
```

7) Στο σημείο αυτό, το σύστημα ζητάει το πρόγραμμα που θα χρησιμοποιηθεί, είτε με την εισαγωγή ενός νέου ή δηλώνοντας *stored* για αυτό που ήδη υπάρχει στο σύστημα

```
 *        start. _______to start unfold process
 *        store. _______to store a new program
 *        list. ________to display the program
 *        exit. ________to return to main menu

(Unfold)
Ready
|: start.

Starting unfold process...

Enter program (type "stored" to use stored program):

|: stored.
      C1: [subseq(nil,v(1))]
      C2: [subseq([v(2)|v(3)],[v(2)|v(4)]),subseq(v(3),v(4))]
      C3: [subseq([v(5)|v(6)],[v(7)|v(8)]),subseq([v(5)|v(6)],v(8))]
      C4: [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]

Select clause to use for unfolding above program

Enter clauses number:

|:
```

7) Πρέπει πριν τον μετασχηματισμό, να δηλώσουμε την πρόταση που θα χρησιμοποιηθεί για την ανάπτυξη του προγράμματος, γράφοντας τον αντίστοιχο αριθμό (στο παράδειγμά μας και το παρακάτω στιγμιότυπο τον αριθμό 4)

```
|: start.
Starting unfold process...
Enter program (type "stored" to use stored program):
|: stored.
     C1: [subseq(nil,v(1))]
     C2: [subseq([v(2)|v(3)],[v(2)|v(4)]),subseq(v(3),v(4))]
     C3: [subseq([v(5)|v(6)],[v(7)|v(8)]),subseq([v(5)|v(6)],v(8))]
     C4: [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]
Select clause to use for unfolding above program
Enter clauses number:
|: 4.
Selected clause
[csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]
Unfold using term (of [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(1
1))])-No "[]":
|:
```

8) Τέλος, είναι απαραίτητο να δηλώσουμε ποιος όρος της πρότασης που επιλέξαμε θα ταυτοποιηθεί με τις κεφαλές των υπολοίπων προτάσεων, όπως απαιτεί ο μετασχηματισμός ανάπτυξης. Γράφουμε subseq(v(9),v(10)) και το σύστημα αναλαμβάνει τον μετασχηματισμό, δίδοντας τελικά το παρακάτω στιγμιότυπο.

```
          Old program
     C1: [subseq(nil,v(17))]
     C2: [subseq([v(18)|v(19)],[v(18)|v(20)]),subseq(v(19),v(20))]
     C3: [subseq([v(21)|v(22)],[v(23)|v(24)]),subseq([v(21)|v(22)],v(24))]
     Clause used for unfolding:
     [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]
     Term used for unification:subseq(v(9),v(10))
          New program
     C1: [subseq(nil,v(17))]
     C2: [csub(nil,v(10),v(11)),subseq(nil,v(11))]
     C3: [subseq([v(18)|v(19)],[v(18)|v(20)]),subseq(v(19),v(20))]
     C4: [csub([v(18)|v(19)],[v(18)|v(20)],v(11)),subseq(v(19),v(20)),subseq([
v(18)|v(19)],v(11))]
     C5: [subseq([v(21)|v(22)],[v(23)|v(24)]),subseq([v(21)|v(22)],v(24))]
     C6: [csub([v(21)|v(22)],[v(23)|v(24)],v(11)),subseq([v(21)|v(22)],v(24)),
subseq([v(21)|v(22)],v(11))]
Keep program?(y/n)_
```

9) Τυπώνουμε *y* στην ερώτηση αν θα κρατήσουμε το πρόγραμμα που δημιουργήθηκε (P_1) στην θέση του προηγούμενου (P_0). Στην συνέχεια, το σύστημα επιστρέφει στο κεντρικό μενού επιλογών.

```
program stored - It is suggested that you use sort command after unfolding.

        Main Menu

  * (u) or unfold. ______to enter unfold menu
  * (f) or fold. ________to enter fold menu
  *      sort. ________to sort variables of stored program
  *     about. ________to read information about this project
  *     hlp. __________for help
  *     halt. _________to quit
Ready
|: _
```

10) Ομοίως και για την ανάπτυξη της πρότασης C5 του τρίτου βήματος του παραδείγματός μας, οπότε το αποτέλεσμα θα είναι:

```
       C1: [subseq(nil,v(65))]
       C2: [subseq([v(66)|v(67)],[v(66)|v(68)]),subseq(v(67),v(68))]
       C3: [csub([v(66)|v(67)],[v(66)|v(68)],v(59)),subseq(v(67),v(68)),subseq([
v(66)|v(67)],v(59))]
       C4: [subseq([v(69)|v(70)],[v(71)|v(72)]),subseq([v(69)|v(70)],v(72))]
       C5: [csub([v(69)|v(70)],[v(71)|v(72)],v(59)),subseq([v(69)|v(70)],v(72)),
subseq([v(69)|v(70)],v(59))]

       Clause used for unfolding:
       [csub(nil,v(10),v(11)),subseq(nil,v(11))]

       Term used for unification:subseq(nil,v(11))

                New program

       C1: [subseq(nil,v(65))]
       C2: [csub(nil,v(10),v(11))]
       C3: [subseq([v(66)|v(67)],[v(66)|v(68)]),subseq(v(67),v(68))]
       C4: [csub([v(66)|v(67)],[v(66)|v(68)],v(59)),subseq(v(67),v(68)),subseq([
v(66)|v(67)],v(59))]
       C5: [subseq([v(69)|v(70)],[v(71)|v(72)]),subseq([v(69)|v(70)],v(72))]
       C6: [csub([v(69)|v(70)],[v(71)|v(72)],v(59)),subseq([v(69)|v(70)],v(72)),
subseq([v(69)|v(70)],v(59))]

Keep program?(y/n)y._
```

11) Για την εφαρμογή του κανόνα μετασχηματισμού πτύξης του τέταρτου βήματος, από το κεντρικό μενού επιλογών τυπώνουμε f ή fold, στην συνέχεια start και για να χρησιμοποιήσουμε το καταχωρημένο πρόγραμμα, stored.

```
         Fold Menu

 *       start. _______to start fold process
 *       list. ________to display the program
 *       exit. ________to return to main menu

(Fold)
Ready
|: start.
Starting folding process...

Enter program (type "stored." to use stored program):

|: stored.
      ||C1: [subseq(nil,v(65))]
      ||C2: [csub(nil,v(10),v(11))]
      ||C3: [subseq([v(66)|v(67)],[v(66)|v(68)]),subseq(v(67),v(68))]
      ||C4: [csub([v(66)|v(67)],[v(66)|v(68)],v(59)),subseq(v(67),v(68)),subseq([
v(66)|v(67)],v(59))]
      ||C5: [subseq([v(69)|v(70)],[v(71)|v(72)]),subseq([v(69)|v(70)],v(72))]
      ||C6: [csub([v(69)|v(70)],[v(71)|v(72)],v(59)),subseq([v(69)|v(70)],v(72)),
subseq([v(69)|v(70)],v(59))]
No. of clause to apply fold transformation(1-6)
|: _
```

12) Επιλέγουμε την πρόταση από το αποθηκευμένο πρόγραμμα στην οποία θα εφαρμοστεί ο κανόνας μετασχηματισμού πτύξης τυπώνοντας τον αντίστοιχο αριθμό και στην συνέχεια τον αριθμό της πρότασης από το σύνολο νέων ορισμών που θα χρησιμοποιηθεί για την εφαρμογή του κανόνα. Με βάση το παράδειγμά μας, οι αριθμοί αυτοί είναι το 6 και το 1 αντίστοιχα.

```
      ||C3: [subseq([v(66)|v(67)],[v(66)|v(68)]),subseq(v(67),v(68))]
      ||C4: [csub([v(66)|v(67)],[v(66)|v(68)],v(59)),subseq(v(67),v(68)),subseq([
v(66)|v(67)],v(59))]
      ||C5: [subseq([v(69)|v(70)],[v(71)|v(72)]),subseq([v(69)|v(70)],v(72))]
      ||C6: [csub([v(69)|v(70)],[v(71)|v(72)],v(59)),subseq([v(69)|v(70)],v(72)),
subseq([v(69)|v(70)],v(59))]
No. of clause to apply fold transformation(1-6)
|: 6.
*****************************************************
Fold transformation will be applied on C6
*****************************************************

     Clause after rename is:
      || C6: [csub([v(213)|v(214)],[v(215)|v(216)],v(203)),subseq([v(213)|v(214)]
,v(216)),subseq([v(213)|v(214)],v(203))]
*****************************************************

Folding uses clauses from New definition set to fold clauses of program.

Currently available are clauses:

      ||D1: [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]
fold using clause(No. 1-1:
|: 1.
```

13) Στην συνέχεια ορίζουμε τους στοιχειώδεις ατομικούς τύπους που θα χρησιμοποιηθούν για τον υπολογισμό των αντικαταστάσεων θ_1 και θ_2 με σκοπό την τελική σύνθεση του θ. Όπως στο αντίστοιχο βήμα του παραδείγματός μας, το αποτέλεσμα την εφαρμογής του κανόνα μετασχηματισμού πτύξης είναι

```
Theta1:[subst(v(10),v(216)),subst(v(9),[v(213)|v(214)])]
Theta2:[subst(v(11),v(203)),subst(v(9),[v(213)|v(214)])]
New_Theta:[subst(v(10),v(216)),subst(v(9),[v(213)|v(214)])]
-----

Thetas merged:
[subst(v(10),v(216)),subst(v(11),v(203)),subst(v(9),[v(213)|v(214)]),subst(v(9),
[v(213)|v(214)])]
-----

removed any subst(x,x)
[subst(v(10),v(216)),subst(v(11),v(203)),subst(v(9),[v(213)|v(214)]),subst(v(9),
[v(213)|v(214)])]
*
*****************************************************
Composed theta:
[subst(v(10),v(216)),subst(v(11),v(203)),subst(v(9),[v(213)|v(214)])]
B derived:csub(v(9),v(10),v(11))
B*theta:
[csub([v(213)|v(214)],v(216),v(203))]
*****************************************************
Folded clause:
[csub([v(213)|v(214)],[v(215)|v(216)],v(203)),csub([v(213)|v(214)],v(216),v(203)
)]
Keep folded clause?(y/n)y._
```

14) Στο επόμενο βήμα αποφασίζουμε να εφαρμόσουμε έναν ακόμα κανόνα μετασχηματισμού νέου ορισμού, οπότε όπως προηγουμένως, τυπώνουμε n_definition και στην συνέχεια την πρόταση που θέλουμε να εισάγουμε σε βασική αναπαράσταση

```
     *       about. ________to read information about this project
     *       hlp. __________for help
     *       halt. _________to quit

Ready
|: n_definition.

New Definition Input

Type the new definition you wish to add, or type cancel to abort

|: [csub1(v(1),v(2),v(3),v(4)),subseq(v(2),v(3)),subseq([v(1)|v(2)],v(4))].
Process completed

             Main Menu

     * (u) or unfold. ______to enter unfold menu
     * (f) or fold. ________to enter fold menu
     * n_definition. ________to import a new definition
     *       about. ________to read information about this project
     *       hlp. __________for help
     *       halt. _________to quit

Ready
|: _
```

15) Στο έκτο βήμα του παραδείγματός μας, εφαρμόζουμε τον κανόνα μετασχηματισμού πτύξης στην πρόταση C6 με την βοήθεια της πρότασης C10. Όπως προηγουμένως, τυπώνουμε τους αντίστοιχους αριθμούς των προτάσεων του συστήματός μας, δηλαδή τους αριθμούς 4 και 2 αντίστοιχα. Αφού τυπώσουμε και τους στοιχειώδεις ατομικούς τύπους που θα χρησιμοποιηθούν για τον υπολογισμό των αντικαταστάσεων θ_1 και θ_2, τελικά προκύπτει το παρακάτω στιγμιότυπο

```
Theta1:[subst(v(3),v(204)),subst(v(2),v(203))]
Theta2:[subst(v(4),v(195)),subst(v(2),v(203)),subst(v(1),v(202))]
New_Theta:[subst(v(3),v(204)),subst(v(2),v(203))]
-----

Thetas merged:
[subst(v(3),v(204)),subst(v(4),v(195)),subst(v(2),v(203)),subst(v(2),v(203)),sub
st(v(1),v(202))]
-----

removed any subst(x,x)
[subst(v(3),v(204)),subst(v(4),v(195)),subst(v(2),v(203)),subst(v(2),v(203)),sub
st(v(1),v(202))]
*
****************************************************
Composed theta:
[subst(v(3),v(204)),subst(v(4),v(195)),subst(v(2),v(203)),subst(v(1),v(202))]
B derived:csub1(v(1),v(2),v(3),v(4))
B*theta:
[csub1(v(202),v(203),v(204),v(195))]
****************************************************
Folded clause:
[csub([v(202)|v(203)],[v(202)|v(204)],v(195)),csub1(v(202),v(203),v(204),v(195))
]
Keep folded clause?(y/n)y._
```

16) Σύμφωνα με το παράδειγμά μας, στο σημείο αυτό επιλέγουμε να αναπτύξουμε την πρόταση C10 χρησιμοποιώντας τον δεύτερο στοιχειώδη ατομικό τύπο. Έτσι, τυπώνουμε τον αριθμό 6 ο οποίος αντιστοιχεί στην πρόταση C10 του παραδείγματος, και τον στοιχειώδη ατομικό τύπο subseq([v(1)|v(2)],v(4)). Τελικά προκύπτει

```
       ||C6: [csub([v(634)|v(635)],[v(634)|v(636)],v(627)),csub1(v(634),v(635),v(6
36),v(627))]
       ||
       ||Clause used for unfolding:
       ||[csub1(v(1),v(2),v(3),v(4)),subseq(v(2),v(3)),subseq([v(1)|v(2)],v(4))]
       ||
       ||Term used for unification:subseq([v(1)|v(2)],v(4))
       ||
       ||        New program
       ||
       ||C1: [subseq(nil,v(497))]
       ||C2: [csub(nil,v(442),v(443))]
       ||C3: [subseq([v(498)|v(499)],[v(498)|v(500)]),subseq(v(499),v(500))]
       ||C4: [csub1(v(1),v(2),v(3),[v(1)|v(500)]),subseq(v(2),v(3)),subseq(v(2),v(
500))]
       ||C5: [subseq([v(501)|v(502)],[v(503)|v(504)]),subseq([v(501)|v(502)],v(504
))]
       ||C6: [csub1(v(1),v(2),v(3),[v(503)|v(504)]),subseq(v(2),v(3)),subseq([v(1)
|v(2)],v(504))]
       ||C7: [csub([v(645)|v(646)],[v(647)|v(648)],v(635)),csub([v(645)|v(646)],v(
648),v(635))]
       ||C8: [csub([v(634)|v(635)],[v(634)|v(636)],v(627)),csub1(v(634),v(635),v(6
36),v(627))]
       ||
Keep program?(y/n)y.
```

17) Ακολουθεί η εφαρμογή του κανόνα μετασχηματισμού πτύξης της πρότασης C12 χρησιμοποιώντας την C4. Τυπώνουμε ξανά τα αντίστοιχα νούμερα των προτάσεων και τους στοιχειώδεις ατομικούς τύπους. Δηλαδή, δίνουμε 4 και 1 για τις αντίστοιχες προτάσεις και ακολουθούν διαδοχικά τα subseq(v(1002),v(1003)), subseq(v(9),v(10)), subseq(v(1002), v(1500)), subseq(v(9),v(11)). Η πρόταση που προκύπτει είναι

```
Theta1:[subst(v(10),v(1003)),subst(v(9),v(1002))]
Theta2:[subst(v(11),v(1500)),subst(v(9),v(1002))]
New_Theta:[subst(v(10),v(1003)),subst(v(9),v(1002))]
-----

Thetas merged:
[subst(v(10),v(1003)),subst(v(11),v(1500)),subst(v(9),v(1002)),subst(v(9),v(1002
))]
-----

removed any subst(x,x)
[subst(v(10),v(1003)),subst(v(11),v(1500)),subst(v(9),v(1002)),subst(v(9),v(1002
))]
*
****************************************************
Composed theta:
[subst(v(10),v(1003)),subst(v(11),v(1500)),subst(v(9),v(1002))]
B derived:csub(v(9),v(10),v(11))
B*theta:
[csub(v(1002),v(1003),v(1500))]
****************************************************
Folded clause:
[csub1(v(1001),v(1002),v(1003),[v(1001)|v(1500)]),csub(v(1002),v(1003),v(1500))]

Keep folded clause?(y/n)y.
```

18) Για την εφαρμογή του κανόνα μετασχηματισμού πτύξης στην πρόταση C13 χρησιμοποιώντας την πρόταση C10, τυπώνουμε τους αριθμούς 5 και 2 αντίστοιχα και ακολουθούν οι στοιχειώδεις ατομικοί τύποι. Τελικά προκύπτει η πρόταση

```
Theta2:[subst(v(4),v(1512)),subst(v(2),v(1010)),subst(v(1),v(1009))]
New_Theta:[subst(v(3),v(1011)),subst(v(2),v(1010))]
-----

Thetas merged:
[subst(v(3),v(1011)),subst(v(4),v(1512)),subst(v(2),v(1010)),subst(v(2),v(1010))
,subst(v(1),v(1009))]
-----

removed any subst(x,x)
[subst(v(3),v(1011)),subst(v(4),v(1512)),subst(v(2),v(1010)),subst(v(2),v(1010))
,subst(v(1),v(1009))]
*
****************************************************
Composed theta:
[subst(v(3),v(1011)),subst(v(4),v(1512)),subst(v(2),v(1010)),subst(v(1),v(1009))
]
B derived:csub1(v(1),v(2),v(3),v(4))
B*theta:
[csub1(v(1009),v(1010),v(1011),v(1512))]
****************************************************
Folded clause:
[csub1(v(1009),v(1010),v(1011),[v(1511)|v(1512)]),csub1(v(1009),v(1010),v(1011),
v(1512))]
Keep folded clause?(y/n)y.
```

19) Αφού επιλέξουμε να κρατήσουμε και αυτή την πρόταση, μπορούμε να εμφανίσουμε το πρόγραμμά μας στην μορφή που έχει πάρει μετά από τους παραπάνω μετασχηματισμούς σε βασική αναπαράσταση, γράφοντας list.

```
|   *        halt. _________to quit                                    |

Ready
|: list.
        ||C1: [subseq(nil,v(497))]
        ||C2: [csub(nil,v(442),v(443))]
        ||C3: [subseq([v(498)|v(499)],[v(498)|v(500)]),subseq(v(499),v(500))]
        ||C4: [subseq([v(501)|v(502)],[v(503)|v(504)]),subseq([v(501)|v(502)],v(504
))]
        ||C5: [csub([v(645)|v(646)],[v(647)|v(648)],v(635)),csub([v(645)|v(646)],v(
648),v(635))]
        ||C6: [csub([v(634)|v(635)],[v(634)|v(636)],v(627)),csub1(v(634),v(635),v(6
36),v(627))]
        ||C7: [csub1(v(1001),v(1002),v(1003),[v(1001)|v(1500)]),csub(v(1002),v(1003
),v(1500))]
        ||C8: [csub1(v(1009),v(1010),v(1011),[v(1511)|v(1512)]),csub1(v(1009),v(101
0),v(1011),v(1512))]

New Definition:
        ||D9: [csub(v(9),v(10),v(11)),subseq(v(9),v(10)),subseq(v(9),v(11))]
        ||D10: [csub1(v(1),v(2),v(3),v(4)),subseq(v(2),v(3)),subseq([v(1)|v(2)],v(4
))]

Ready
|:
```

20) Επιπλέον, δίνεται η δυνατότητα μετατροπής του προγράμματός μας σε μη βασική αναπαράσταση, γράφοντας convert.

```
        ||C4: [subseq([v(8)|v(9)],[v(7)|v(10)]),subseq([v(8)|v(9)],v(10))]
        ||C5: [csub([v(12)|v(13)],[v(11)|v(14)],v(15)),csub([v(12)|v(13)],v(14),v(1
5))]
        ||C6: [csub([v(16)|v(17)],[v(16)|v(18)],v(19)),csub1(v(16),v(17),v(18),v(19
))]
        ||C7: [csub1(v(20),v(21),v(22),[v(20)|v(23)]),csub(v(21),v(22),v(23))]
        ||C8: [csub1(v(25),v(26),v(27),[v(24)|v(28)]),csub1(v(25),v(26),v(27),v(28)
)]
Converting program to non-ground form:
subseq(nil,X1).
csub(nil,X2,X3).
subseq([X4|X5],[X4|X6]):-
        subseq(X5,X6).
subseq([X8|X9],[X7|X10]):-
        subseq([X8|X9],X10).
csub([X12|X13],[X11|X14],X15):-
        csub([X12|X13],X14,X15).
csub([X16|X17],[X16|X18],X19):-
        csub1(X16,X17,X18,X19).
csub1(X20,X21,X22,[X20|X23]):-
        csub(X21,X22,X23).
csub1(X25,X26,X27,[X24|X28]):-
        csub1(X25,X26,X27,X28).
Ready
|:
```

Το αρχικό πρόγραμμα αντικείμενο (πρόγραμμα 7.1) φαινομενικά διαφέρει από το πρόγραμμα που παρουσιάζεται τελικά μετά τους παραπάνω μετασχηματισμούς (πρόγραμμα 7.2).

Παρατηρούμε όμως οτι για τους στόχους

?- csub([a,b,c],[x,y,z,b,d,c,a],[x,z,y,a,b,c]).]). η Prolog επιστρέφει no

?- csub([a,b,c],[x,y,a,b,c,z],[x,a,b,c,y,z]).]). η Prolog επιστρέφει yes

τρέχοντάς τους διαδοχικά και στα δύο προγράμματα. Ωστόσο, η αποτελεσματικότητα των δύο προγραμμάτων δεν είναι η ίδια. Για παράδειγμα, για τον στόχο

?- csub([a,b,c],[x,y,a,b,c,z],[x,a,b,c,y,z]).

διαπιστώσαμε ότι η Prolog είναι 1,96 φορές ταχύτερη στην απάντησή της με το τελικό πρόγραμμα P_8 από ότι στο αρχικό P_0. Με αύξηση των εφαρμογών

των κανόνων μετασχηματισμού, αναμένεται αντίστοιχα περαιτέρω βελτίωση του προγράμματος αντικειμένου. Το παράδειγμα αυτό, επικυρώνει την σημασία των συστημάτων μετασχηματισμού λογικών προγραμμάτων, λαμβάνοντας υπόψη τον όγκο πληροφοριών που καλούνται τα περισσότερα υπολογιστικά συστήματα να καλύψουν στις μέρες μας.

8. Συμπεράσματα – Μελλοντική επέκταση του συστήματος

Το σύστημα που υλοποιήθηκε τελικά στα πλαίσια της πτυχιακής εργασίας, δεν περιορίζεται στον μετασχηματισμό των προγραμμάτων. Όπως κάθε αυτόνομη εφαρμογή οφείλει να παρέχει στον χρήστη ευελιξία με σκοπό την όσο το δυνατόν πιο σύντομη επίτευξη του στόχου.

Εκτός λοιπόν των αλγορίθμων fold, unfold, unify και apply, που αποτελούν τον πυρήνα των μετασχηματισμών, σχηματίστηκαν βοηθητικά κατηγορήματα που αποσκοπούσαν στον σχεδιασμό μενού επιλογών, βοήθειας του χρήστη και καλύτερης επικοινωνίας με το περιβάλλον της Prolog.

Φυσικά, υπάρχουν αρκετές λειτουργίες που θα μπορούσαν να προστεθούν με σκοπό την βελτίωση του συστήματος.

Για παράδειγμα, στην επιλογή εξαγωγής του τελικού προγράμματος σε μη βασική μορφή, θα μπορούσε να προστεθεί η επιλογή αποθήκευσης του αποτελέσματος σε αρχείο. Στην τρέχουσα μορφή του συστήματος, το αποτέλεσμα των μετασχηματισμών στο πρόγραμμα αντικείμενο δεν αποθηκεύεται σε κάποιο αρχείο, πέρα από την εμφάνιση του στην οθόνη της κονσόλας. Οπωσδήποτε, μια τέτοια προσθήκη, συνεπάγεται και την βελτίωση του αντίστοιχου τμήματος στον κώδικα της εφαρμογής.

Επίσης, κατά την κατανομή του χρόνου υλοποίησης της εφαρμογής, δεν διατέθηκε αρκετός χρόνος στην σχεδίαση των κατηγορημάτων βοήθειας. Αυτό είχε ως αποτέλεσμα, οι αναφορές τους στα σχετικά κατηγορήματα να είναι συνοπτικές. Θα ήταν λοιπόν χρήσιμο, το τμήμα βοήθειας του συστήματος να αναπτυχθεί ώστε η εφαρμογή να μπορεί να είναι προσιτή σε περισσότερους χρήστες.

Ένα ακόμα χαρακτηριστικό του συστήματός μας, είναι η δυνατότητα εισαγωγής του προγράμματος αντικειμένου από αρχείο. Μοναδική δέσμευση είναι ότι το αρχείο θα πρέπει να περιέχει το πρόγραμμα αντικείμενο σε μορφή λίστας και σε βασική αναπαράσταση. Επομένως, μια από τις σημαντικότερες βελτιώσεις, θα ήταν ένα τμήμα μετατροπής του προγράμματος από μη βασική σε βασική αναπαράσταση. Αυτό θα έχει σαν συνέπεια το σύστημα να είναι πιο ευέλικτο, καθώς θα δίνεται η δυνατότητα να εφαρμόζονται οι μετασχηματισμοί άμεσα σε προγράμματα Prolog, σε μη βασική μορφή. Είναι λογικό ότι αυτή η βελτίωση θα προσφέρει αισθητή ευελιξία στο σύστημα, καθώς απαλλάσσεται ο χρήστης από την δέσμευση προσαρμογής του αρχείου στα πρότυπα της εφαρμογής, αυξάνεται η λειτουργικότητα του συστήματος και περιορίζεται η πιθανότητα σφάλματος.

Εξίσου σημαντική επέκταση θα ήταν ο σχεδιασμός ενός προγράμματος επικοινωνίας σε κάποια άλλη γλώσσα, όπως για παράδειγμα την C++ με την οποία η Prolog συνεργάζεται πλήρως, το οποίο θα αναλαμβάνει το γραφικό τμήμα του συστήματος. Αυτό, θα είχε ως αποτέλεσμα την απαλλαγή του τμήματος μετασχηματισμού από τα γραφικά των χαρακτήρων ASCII, που οπωσδήποτε επιβραδύνουν την εξέλιξη της έρευνας κατά την αναζήτηση του στόχου. Επίσης, το περιβάλλον θα γινόταν περισσότερο λειτουργικό με αυτό τον τρόπο, καθώς η Prolog δεν προσφέρεται για τον σχεδιασμό σύγχρονου και λειτουργικού περιβάλλοντος εργασίας.

9. Βιβλιογραφία

[1] Bratko Ivan, *Prolog Programming for Artificial Intelligence*, Addison-Wesley, (2nd ed.).

[2] Charniak E., Riesbeck C.K., McDermott D.V., Erlbaum Lawrence, *Artificial Intelligence Programming*, 1987.

[3] Christopher John Hogger, *Essentials of Logic Programming,* , Clarendon Press – Oxford, 1990.

[4] Clocksin W., Mellish C., *Programming in Prolog*, Springer-Verlang.

[5] Εγχειρίδιο της Sicstus Prolog 3.7

[6] Gallagher J., *Program Analysis and Transformation*, Handout at Logic Programming Summer School LPSS'90, University of Zurich, August 1990.

[7] Hill P. M. and Lloyd J. W. *Meta-Programming for Dynamic Knowledge Bases*. Technical Report CS-88-18, Department of Computer Science, University of Bristol, 1988.

[8] Κατζουράκη Μ., Γεργατσούλης Μ., Κόκκοτος Σ., *PROγραμματίζοντας στη LOGική,* Ελληνική Εταιρία Επιστημόνων Η/Υ και Πληροφορικής, Αθήνα 1991.

[9] Μαρακάκης Μ., *Σημειώσεις Τεχνητής Νοημοσύνης*, Ηράκλειο, Ιούνιος 2002.

[10] O'Keefe R., *The Craft of Prolog,* , MIT Press.

[11] Partsch H. and Steinbrüggen R., *Computing Serveys*, Vol.15, No.3, September 1983.

[12] Pettorossi Alberto and Proietti Maurizio, *Automatic Derivation of Logic Programs by Transformation*, ESSLLI 2000.

[13] Richardson Julian and Fuchs Norbert, *Development of Correct Transformation Schemata for Prolog Programs*, Department of Artificial Intelligence, Edinburgh University, 80 South Bridge, Edinburgh EH1 1HN, Scotland, Department of Computer Science, University of Zurich, CH-8057 Zurich, Switzerland.

[14] Shapiro E. and Sterling L., *The Art of Prolog,* , MIT Press.

[15] Tamaki H. and Sato T., *Unfold/Fold Transformation of Logic Programs*. Proc. Of 2nd International Logic Programming Conference, pp 127-138, Uppsala, 1984.

Παράρτημα

Α. Πηγαίος Κώδικας του συστήματος σε Sicstus Prolog

Στο παράρτημα αυτό, παραθέτουμε τον πηγαίο κώδικα του συστήματος που περιγράφθηκε στα προηγούμενα κεφάλαια, όπως υλοποιήθηκε σε Prolog:

- **Kernel.pl**

```
%%%%%%%%%%%%UNFOLD MENU%%%%%%%%%%%%%%%%%%%%
% 1.Insert initial program                                      %
% 2.Ask for new Definition and check if it is already in program,    %
%   if not proceed                                              %
% 3.Get term to use for unfolding                               %
% 4.Unfold and display                                          %
% 5.Save and return to main menu                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%FOLD MENU%%%%%%%%%%%%%%%%%%%%
% 1.Insert Initial program                                      %
% 2.Ask for clause to apply folding transformation              %
% 3.Ask for clause to use on folding                            %
% 4.Ask the terms that can be uniffied                          %
% 5.Fold and display                                            %
% 6.Save and return to main menu                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Set date and version
%---------------------------%
date:-
        print(' 07/11/2002 ').
ver:-
        print(' ver(1.09)'),nl.
%---------------------------%

%Load files
%-----------------%
:- ['unify.pl'].
:- ['rename.pl'].
:- ['apply.pl'].
```

```
:- ['unfold.pl'].
:- ['fold.pl'].
%-----------------%

%Interface
%-------------------------------------------------------------------%
clear_lines(0).
clear_lines(X):-
        X1 is X-1,clear_lines(X1),nl.
space(0).
space(X):-
        X1 is X-1,space(X1),print(' ').



cls:-
        clear_lines(48).
string(Chr,X):-
        X<0.
string(Chr,0).
string(Chr,Length):-
        Length1 is Length-1,string(Chr,Length1),put(Chr).
interface:-
        cls,space(23),string(176,5),string(177,5),string(178,5),string(178,5),
        string(177,5),string(176,5),nl,
        space(9),string(176,5),string(177,5),string(178,11),
        print('  TEI of Crete  '),string(178,11),string(177,5),string(176,5),nl,
        space(9),string(176,5),string(177,5),
        print(' Department of Electrical Engineering '),string(177,5),
        string(176,5),nl,space(10),string(176,5),string(177,5),
        print(' Diploma work on Meta-programming '),string(177,5),
        string(176,5),nl,space(15),string(176,5),
        string(177,5),print(' by George Santipantakis '),
        string(177,5),string(176,5),nl,
        space(17),string(176,5),string(177,5),string(178,5),date,
        string(178,5),string(177,5),string(176,5),nl,
        space(23),string(176,5),string(177,5),string(178,5),
        string(178,5),string(177,5),string(176,5),nl,
        space(28),string(176,5),string(177,5),string(177,5),string(176,5),nl,
        space(70),ver,nl.
```

```
%--------------------------------------------------------------------%
%Help lines
%--------------------------------------------------------------------%
hlp2:-
        space(8),string(201,1),string(205,62),string(187,1),nl,
        space(8),string(186,1),print('                         Morpheus'),
        space(27),string(186,1),nl,
        space(8),string(199,1),string(196,62),string(182,1),nl,
        space(8),string(186,1),
        print('    This program transformates prolog programs, using'),
        space(6),string(186,1),nl,
        space(8),string(186,1),
        print('                          fold/unfold modifications'),
        space(6),string(186,1),nl,
        space(8),string(186,1),
        print(' Type: hlp. for help or main. to initialize shell '),
        space(12),string(186,1),nl,
        space(8),string(200,1),string(205,62),string(188,1),nl.

%--------------------------------------------------------------------%
hlp:-
        space(8),string(201,1),string(205,62),string(187,1),nl,
        space(8),string(186,1),print(' Help: type hlp to get this message '),
        space(26),string(186,1),nl,
        space(8),string(199,1),string(196,62),string(182,1),nl,
        space(8),string(186,1),print(' Existing Predicates: '),
        space(40),string(186,1),nl,
        space(8),string(186,1),print('       sort.'),
        space(50),string(186,1),nl,
        space(8),string(186,1),print('       unify(T1,T2,Subst).'),space(36),
        string(186,1),nl,
        space(8),string(186,1),print('       apply(Clauses,Theta,New_Clauses).'),
        space(22),string(186,1),nl,
        space(8),string(186,1),
        print('    unfold.'),space(48),
        string(186,1),nl,
        space(8),string(186,1),
        print('    fold.'),
        space(50),string(186,1),nl,
        space(8),string(186,1),
```

```
        print('    n_definition.'),
        space(42),string(186,1),nl,
        space(8),string(186,1),print('       main.'),space(50),
        string(186,1),nl,
        space(8),string(186,1),print('       store.'),space(49),
        string(186,1),nl,
        space(8),string(186,1),print('       list.'),space(50),
        string(186,1),nl,
        space(8),string(186,1),print('       ver.'),
        space(51),string(186,1),nl,
        space(8),string(186,1),print('       input_fl(X).'),
        space(43),string(186,1),nl,
        space(8),string(186,1),print('       store_fl(X).'),
        space(43),string(186,1),nl,
        space(8),string(186,1),print('       convert.'),
        space(47),string(186,1),nl,
        space(8),string(186,1),
        print('    help_on(Topic), where Topic can be the name of one of  '),
        string(186,1),nl,
        space(8),string(186,1),print('       the above terms.'),
        space(39),string(186,1),nl,
        space(8),string(200,1),string(205,62),string(188,1),nl.

help_on(sort):-
        space(6),string(201,1),string(205,63),string(187,1),nl,
        space(6),string(186,1),print('                    sort.'),
        space(49),string(186,1),nl,
        space(6),string(199,1),string(196,63),string(182,1),nl,
        space(6),string(186,1),
        print('renames and sorts variable indexes in program.'),
        space(17),string(186,1),nl,
        space(6),string(186,1),
        print('e.g. sort on program'),
        space(43),string(186,1),nl,
        space(6),string(186,1),
        print('     C1 : subseq(nil,v(7))'),
        space(37),string(186,1),nl,
        space(6),string(186,1),
print('     C2 : subseq([v(5)|v(6)],[v(7)|v(11)]),subseq(v(5),v(11))'),
        space(2),string(186,1),nl,
```

```
        space(6),string(186,1),print('will return'),
        space(52),string(186,1),nl,
        space(6),string(186,1),
        print('     C1 : subseq(nil,v(1))'),
        space(37),string(186,1),nl,
        space(6),string(186,1),
print('     C2 : subseq([v(2)|v(3)],[v(4)|v(5)]),subseq(v(2),v(5))'),
        space(4),string(186,1),nl,
        space(6),string(200,1),string(205,63),string(188,1),nl.

help_on(unify):-
        space(6),string(201,1),string(205,63),string(187,1),nl,
        space(6),string(186,1),
        print('                 unify(T1,T2,Subst).'),
        space(35),string(186,1),nl,
        space(6),string(199,1),string(196,63),string(182,1),nl,
        space(6),string(186,1),
        print('unifies list of terms T1 with another list of terms T2.'),
        space(8),string(186,1),nl,
        space(6),string(186,1),
        print('e.g. unify([a,b],[v(1),v(2)],Theta). will return'),
        space(15),string(186,1),nl,
        space(6),string(186,1),
        print('Theta=[subst(a,v(1),subst(b,v(2)] '),
        space(29),string(186,1),nl,
        space(6),string(200,1),string(205,63),string(188,1),nl.

help_on(apply):-
        space(6),string(201,1),string(205,67),string(187,1),nl,
        space(6),string(186,1),
        print('                 apply(Clauses,Theta,New_Clauses).'),
        space(25),string(186,1),nl,
        space(6),string(199,1),string(196,67),string(182,1),nl,
        space(6),string(186,1),
print('applies list of substitutions Theta on list of clauses Clauses to'),
        space(2),string(186,1),nl,
        space(6),string(186,1),print('derive New_Clauses'),
        space(49),string(186,1),nl,
        space(6),string(186,1),
print('e.g. apply([a(c,v(1))],[subst(v(1),b)],New_Clauses). will return'),
```

```
            space(3),string(186,1),nl,
            space(6),string(186,1),
print(' New_Clauses=[a(c,b)]'),space(46),string(186,1),nl,
            space(6),string(200,1),string(205,67),string(188,1),nl.

help_on(main):-
            space(6),string(201,1),string(205,66),string(187,1),nl,
            space(6),string(186,1),
print('                                    main.'),
            space(36),string(186,1),nl,
            space(6),string(199,1),string(196,66),string(182,1),nl,
            space(6),string(186,1),
print('              This predicate, will initialize the main menu '),
            space(11),string(186,1),nl,
            space(6),string(186,1),
            print('and will use as a shell the rest predicates.'),
            space(22),string(186,1),nl,
            space(6),string(186,1),
            print(' Help still functions while in shell.'),
            space(29),string(186,1),nl,
            space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(input_fl):-
            space(6),string(201,1),string(205,66),string(187,1),nl,
            space(6),string(186,1),
            print('                                    input_fl(X).'),
            space(29),string(186,1),nl,
            space(6),string(199,1),string(196,66),string(182,1),nl,
            space(6),string(186,1),
            print('                          Syntax: input_fl(filename). '),
            space(20),string(186,1),nl,
            space(6),string(186,1),
            print('        This predicate, sets as input the given file'),
            space(16),string(186,1),nl,
            space(6),string(186,1),
            print('    and executes the commands within it'),
            space(30),string(186,1),nl,
            space(6),string(186,1),
            print('    from linestop(.) to linestop(.).'),
            space(33),string(186,1),nl,
```

```
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(store_fl):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                                store_fl(X).'),
        space(29),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('                       Syntax: store_fl(filename). '),
        space(20),string(186,1),nl,
        space(6),string(186,1),
        print('       This predicate, reads the given input file'),
        space(18),string(186,1),nl,
        space(6),string(186,1),
        print('    and stores the program within it'),
        space(33),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(list):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                                list.'),
        space(36),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('       This predicate, displays the stored program'),
        space(17),string(186,1),nl,
        space(6),string(186,1),
        print('    and the new definition clause'),
        space(36),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(ver):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                                          ver.'),
        space(29),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
```

```
        print('       This predicate, shows current version'),
        space(23),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(store):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                                  store.'),
        space(27),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('       Use this predicate to store a new program'),
        space(19),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(fold):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                           fold (or f).'),
        space(29),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('       Fold is the transformation technique that will fold'),
        space(9),string(186,1),nl,
        space(6),string(186,1),
        print(' partially current program, by application of a selected'),
        space(10),string(186,1),nl,
        space(6),string(186,1),
        print(' term on another. Type fold (or f) to enter fold menu'),
        space(13),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(unfold):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                           unfold (or u).'),
        space(27),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('       Unfold is a program transformation technique,'),
```

```
        space(15),string(186,1),nl,
        space(6),string(186,1),
        print(' as applied by Tamaki - Sato. Transforms a logic program'),
        space(10),string(186,1),nl,
        space(6),string(186,1),
        print(' into a new one with more clauses.'),
        space(32),string(186,1),nl,
        space(6),string(186,1),
        print(' Type unfold (or u) to enter unfold menu'),
        space(26),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(n_definition):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                         n_definition.'),
        space(28),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('      New Definition is a program transformation technique.'),
        space(7),string(186,1),nl,
        space(6),string(186,1),
        print(' Adds a new clause in the Program and the New Definition set.'),
        space(5),string(186,1),nl,
        space(6),string(186,1),
        print(' It is necessary for performing the folding transformation.'),
        space(7),string(186,1),nl,
        space(6),string(186,1),
        print(' Type n_definition to start the new definition process.'),
        space(11),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

help_on(convert):-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                         convert.'),
        space(33),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print('      Use this predicate to convert stored program'),
```

```
        space(16),string(186,1),nl,
        space(6),string(186,1),
        print('    to a non-ground form'),
        space(45),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

about:-
        cls,interface,hlp2.

help_on(X):-
        print(X),print(' is not part of the help file. Type hlp for help'),
        nl.
%---------------------------------------------------------------------%
%print something in window
print_in_window(X):-
        string(201,1),string(205,72),nl,
        string(186,1),print(X),nl,
        string(200,1),string(205,72),nl.
print_in_window(X,Title):-
        string(201,1),string(205,72),nl,
        string(186,1),print(Title),nl,
        string(199,1),string(196,72),nl,
        string(186,1),print(X),nl,
        string(200,1),string(205,72),nl.
%---------------------------------------------------------------------%
main_menu:-
        space(6),string(201,1),string(205,66),string(187,1),nl,
        space(6),string(186,1),
        print('                Main Menu'),space(48),string(186,1),nl,
        space(6),string(199,1),string(196,66),string(182,1),nl,
        space(6),string(186,1),
        print(' * (u) or unfold. ______to enter unfold menu'),
        space(22),string(186,1),nl,
        space(6),string(186,1),
        print(' * (f) or fold. ________to enter fold menu'),
        space(24),string(186,1),nl,
        space(6),string(186,1),
        print(' * n_definition. ________to import a new definition'),
        space(15),string(186,1),nl,
        space(6),string(186,1),
```

```
        print(' *  about. ________to read information about this project'),
        space(4),string(186,1),nl,
        space(6),string(186,1),print(' *    hlp. __________for help'),
        space(34),string(186,1),nl,
        space(6),string(186,1),print(' *    halt. _________to quit'),
        space(35),string(186,1),nl,
        space(6),string(200,1),string(205,66),string(188,1),nl.

main:-
        main(Program).

main(Program):-
        main_menu,shell(Program,N_d).
shell(P,N_Def):-
        print('Ready'),nl,
        read(X),request(X,P,N_Def).

request(halt,P,New_D):-
        interface,
        print('thank you for using this program'),nl,
        halt.
request(main,P,New_D):-
        print('Already in shell. Type hlp. for help'),nl,main_menu,shell(P,New_D).

request(n_definition,P,New_D):-
        new_definition(P,New_D,P1,New_D1),!,main_menu,shell(P1,New_D1).

request(u,P,New_D):-
        unfold(P,New_D),shell(P,New_D).
request(f,P,New_D):-
        fold(P,New_D),shell(P,New_D).

request(unfold,P,New_D):-
        unfold(P,New_D),shell(P,New_D).
request(fold,P,New_D):-
        fold(P,New_D),shell(P,New_D).

request(list,P,New_D):-
        display_P(P,0,X),nl,
        print('New Definition:'),nl,
```

```
        display_D(New_D,X,X1),nl,
        shell(P,New_D).

request(store,X,Y):-
        print('type program to be stored:'),nl,
        read(X1),nl,
        shell(X1,Y).

%ERROR HANDLER
request(sort,[],New_D):-
        print('No variables found, program empty.'),nl,
        print(' Store a new program before using sort command.'),nl,
        shell(New_P,N_New_D).

request(sort,P,New_D):-
        rename_program(P,New_D,New_P,X),
        shell(New_P,New_D).

request(convert,P,New_D):-
        export1(P),shell(P,New_D).

request(about,P,New_D):-
        cls,interface,shell(P,New_D).

request(X,P,New_D):-
        \+exists(X),
        print('command does not exist in this menu'),nl,
        shell(P,New_D).
request(X,P,New_D):-
        exists(X),
        X,
        shell(P,New_D).
%-----------------------------------------------------------------
exists(X):-
comms(X,
%existing commands
        [hlp,
        help_on(A),
        input_fl(Filename),
        store_fl(Filename),
```

```
        cls,
        ver,
        unify(A,B,C),
        rename1(A,B,C),
        unfold(A,B,C),
        apply(A,B,C)
]).

comms(X,[X|T]).
comms(X,[Y|T]):-
        comms(X,T).
%----------------------------------------------------------------
%runs outside shell
unfold:-
        unfold_hlp(P,New_D).
%runs from shell
unfold(P,New_D):-
     unfold_hlp(P,New_D).

unfold_menu:-
        space(6),string(201,1),string(205,67),string(187,1),nl,
     space(6),string(186,1),
     print('        Unfold Menu'),space(47),string(186,1),nl,
     space(6),string(199,1),string(196,67),string(182,1),nl,
        space(6),string(186,1),
        print(' *       start. _______to start unfold process'),
        space(20),string(186,1),nl,
     space(6),string(186,1),
        print(' *       store. _______to store a new program'),
        space(21),string(186,1),nl,
     space(6),string(186,1),
        print(' *       list. ________to display the program'),
        space(21),string(186,1),nl,
        space(6),string(186,1),
     print(' *       exit. ________to return to main menu'),
     space(21),string(186,1),nl,
     space(6),string(200,1),string(205,67),string(188,1),nl.

unfold_hlp(P,New_D):-
     unfold_menu,
```

```
        u_shell(P,New_D,UniTerm).

u_shell(Program,Definition,UniTerm):-
        print('(Unfold)'),nl,print('Ready'),nl,
        read(X),u_request(X,Program,Definition,UniTerm).

u_request(u,Program,Definition,UniTerm):-
        print('Already in unfold menu'),nl,
        u_shell(Program,Definition,UniTerm).

u_request(unfold,Program,Definition,UniTerm):-
        print('Already in unfold menu'),nl,unfold_menu,
        u_shell(Program,Definition,UniTerm).

u_request(exit,Program,Definition,UniTerm):-
        print('Returned to main menu'),nl,
            main_menu,shell(Program,Definition).

u_request(halt,Program,Definition,UniTerm):-
            print('Halt not permitted in this menu.'),nl,
            print('Type exit to return to main menu.'),nl,
        u_shell(Program,Definition,UniTerm).

u_request(start,X,Y,Z):-
print_in_window(
'Enter program (type "stored" to use stored program):',
'Starting unfold process...'),
            program(X,Answer,X1),
            display_P(X1,0,_Counter),
            print_in_window(
            'Enter clauses number:',
            'Select clause to use for unfolding above program'),!,
            read_number(1,Number,_Counter),
            Number1 is Number -1,
            derive_term(X1,Number1,Clause),
            print_in_window(Clause,'Selected clause'),
            ask_term(Clause,UniTerm,Z),
            unfold(X1,_New_P,Clause,UniTerm),
            reply(Reply,_New_P,X1),nl,
            main_menu,shell(Reply,Y).
```

```
u_request(store,X,Y,Z):-
        print('type program to be stored:'),nl,
        read(X1),nl,
        u_shell(X1,Y,Z).




%----------------------------------------------------------------
read_number(X_low,X1,X_max):-
        read(X),
        check_mid(X,X_low,X_max,X1).

check_mid(X,X_low,X_max,X1):-
        X>X_low,
        X>X_max,
        print('Number must be between '),
        print(X_low),print(' and '),
        print(X_max),print('.'),nl,
        read_number(X_low,X1,X_max).

check_mid(X,X_low,X_max,X1):-
        X<X_low,
        X<X_max,
        print('Number must be between '),
        print(X_low),print(' and '),
        print(X_max),print('.'),nl,
        read_number(X_low,X1,X_max).

check_mid(X,X_low,X_max,X):-
        X>X_low,
        X<X_max.

check_mid(X,X,X_max,X).
check_mid(X,X_low,X,X).

%----------------------------------------------------------------
%this tells to use stored program as initial one
%program(Current_Program,Answer,Use_Program)
program(X,A,X1):-
```

```
            read(A),program1(X,A,X1).
program1(X,stored,X).
program1(X,Y,Y).

%--------------------------------------------------------------------
%check if clause is part of initial program
definition(Return,New_D,Temp,Z):-
        print('New Definition(type "stored" to use the stored one:'),nl,
            program(New_D,Answer,Return),nl,
            print('selected:'),nl,
            display_D(Return),nl,
            !,check2(Return,Temp,Z).
check2(Y,Temp,Z):-
            member(Z,Y),print(Z),print(' exists within initial program'),
            nl,!,definition(Return,New_D,Temp,Z1).
check2(Y,Z,Z):-
            \+member(Z,Y).

u_request(list,X,Y,z):-
            display_P(X,0,X1),
            print('New Definition:'),nl,
            display_D(Y,X1,X2),nl,
            u_shell(X,Y,Z).



%--------------------------------------------------------------------
%check if term is part of new clause
ask_term(Y,Temp,Z):-
        print('Unfold using term (of '),
        print(Y),print(')-No "[]":'),nl,read(Z),!,check1(Y,Temp,Z),cls.
check1(Y,Temp,Z):-
            \+member(Z,Y),print(Z),print(' is not part of '),
            print(Y),nl,!,ask_term(Y,Temp,Z1).
check1(Y,Z,Z):-
            member(Z,Y).

%--------------------------------------------------------------------
reply(Reply,New_P,Old_P):-
        print('Keep program?(y/n)'),read(A),ask111(Reply,New_P,Old_P,A).
ask111(P,P,X,y):-
```

```
        cls,
        print('program stored -'),
        print(' It is suggested that you use sort command after unfolding.'),
        nl.
ask111(X,P,X,n):-
        print('Keeping old program and returning to main menu').
ask111(Reply,_New_P,X,A):-
    print('Keep program?(y/n)'),read(A),
        \+A==y,
        \+A==n,
        ask111(Reply,_New_P,X,A).

%-----------------------------------------------------------------
u_request(X,Program,Definition,UniTerm):-
    \+exists(X),
        print('command does not exist in unfold menu'),nl,
    u_shell(Program,Definition,UniTerm).
u_request(X,Program,Definition,UniTerm):-
    exists(X),
    X,
    u_shell(Program,Definition,UniTerm).

u_request(X):-
    \+exists(X),
    print('unknown command'),nl,
    u_shell.
u_request(X),
        exists(X),
        X,
        u_shell.

%--------------------------------------------------------------------
%runs outside shell
fold:-
        fold_hlp(P,New_D).
%runs from shell
fold(P,New_D):-
        fold_hlp(P,New_D).

%--------------------------------------------------------------------
```

```
fold_menu:-
        space(6),string(201,1),string(205,67),string(187,1),nl,
    space(6),string(186,1),
    print('        Fold Menu'),space(49),string(186,1),nl,
    space(6),string(199,1),string(196,67),string(182,1),nl,
        space(6),string(186,1),
        print(' *       start. _______to start fold process'),
        space(22),string(186,1),nl,
    space(6),string(186,1),
        print(' *       list. ________to display the program'),
        space(21),string(186,1),nl,
    space(6),string(186,1),
    print(' *       exit. ________to return to main menu'),
    space(21),string(186,1),nl,
    space(6),string(200,1),string(205,67),string(188,1),nl.

fold_hlp(P,New_D):-
    fold_menu,
    f_shell(P,New_D,UniTerm).

f_shell(Program,Definition,UniTerm):-
    print('(Fold)'),nl,print('Ready'),nl,
    read(X),f_request(X,Program,Definition,UniTerm).

f_request(f,Program,Definition,UniTerm):-
    print('Already in fold menu'),nl,fold_menu,
    f_shell(Program,Definition,UniTerm).
f_request(fold,Program,Definition,UniTerm):-
    print('Already in fold menu'),nl,fold_menu,
    f_shell(Program,Definition,UniTerm).

f_request(list,X,Y,z):-
        display_P(X,0,X1),
        print('New Definition:'),nl,
        display_D(Y,X1,X2),nl,
        f_shell(X,Y,Z).

f_request(store,X,Y,Z):-
        print_in_window(
'Folding requires a New Definition Set. To store a new Program return to main menu',
```

```
            'Unable to import set of clauses in Fold menu'),
            f_shell(X,Y,Z).

f_request(exit,Program,Param1,Param2):-
        print('Returned to main menu'),nl,main_menu,shell(Program,Param1).

f_request(start,X,Y,Z):-
            print('Starting folding process...'),nl,
            print_in_window(
            'Enter program (type "stored." to use stored program):'),
            nl,program(X,Answer,X1),
            display_P(X1,0,_Counter),
%           fold first clause using second by unifying given terms
            print('No. of clause to apply fold transformation(1-'),
            print(_Counter),print(')'),nl,
%%%%%%%%%%%%%%%%%%%
%Get clause to fold
            read_number(1,Number,_Counter),
            Number1 is Number -1,
            derive_term(X1,Number1,Term_to_apply),
%%%%%%%%%%%%%%%%%%%
            print('********************************************************'),nl,
            print('Fold transformation will be applied on C'),
            print(Number),nl,
            print('********************************************************'),nl,
            nl,space(5),print('Clause after rename is:'),nl,
            var_up([Term_to_apply],[Clause1],Var_max1),!,
            display_Clause(Clause1,Number),
            print('********************************************************'),nl,
            print_in_window(
'Folding uses clauses from New definition set to fold clauses of program.'),
            print_in_window('Currently available are clauses:'),
            display_D(Y,0,_Counter2),
            print('fold using clause(No. 1-'),
            print(_Counter2),print(':'),nl,
%%%%%%%%%%%%%%%%%%
%Get clause to use
            read_number(1,Number2,_Counter),
            Number3 is Number2 -1,
            derive_term(Y,Number3,Term),
```

```
%%%%%%%%%%%%%%%%%%%
          print('***************************************************'),nl,nl,
          print('Selected clause from New Definition set:'),nl,
          print(Term),nl,
          print('***************************************************'),nl,nl,
          print('Term that will be unified to derive theta1'),nl,nl,
          prepare_f1(Number,Clause1,-17,Term,Theta1,1),
          prepare_f1(Number,Clause1,-17,Term,Theta2,2),
          print('Processing operation <Theta1>*<Theta2> - composition'),nl,
          print('***************************************************'),nl,
          print('*'),nl,
          compose_theta(Theta1,Theta2,Comp_theta),
          print('*'),nl,
          print('***************************************************'),nl,
          print('Composed theta:'),nl,
          print(Comp_theta),nl,
%         display new clause
          get_applied_B(Clause1,Term,Comp_theta,Head1,B,Applied_B,Folded_clause),
          print('B derived:'),
          print(B),nl,
          print('B*theta:'),nl,
          print(Applied_B),nl,
          print('***************************************************'),nl,
          print('Folded clause:'),nl,
          print(Folded_clause),nl,
%         ask before replace
          reply_f(Reply,Folded_clause,X1,Number1),
%         return to main menu
          main_menu,shell(Reply,Y).

reply_f(Reply,Folded_clause,Old_P,N1):-
      print('Keep folded clause?(y/n)'),read(A),ask112(Reply,Folded_clause,
                                                            N1,Old_P,A).
ask112(Z1,F,N1,X,y):-
          replace_clause(Z1,X,F,N1),
          cls,
          print('clause stored -'),
          print(' It is suggested that you use sort command after folding.'),nl.

ask112(X,F,N1,X,n):-
```

```
        print('Keeping old program and returning to main menu'),nl.
ask112(Reply,F,N1,X,A):-
        reply_f(Reply,F,X,N1).

%-------------------------------------------------------------
replace_clause(New_Program,Old_Program,Folded_clause,Position):-
        Position1 is Position +1,
        derive_term(Old_Program,Position,Term),
        remove_clause(Old_Program,Term,P1),
        apply_n_definition(P1,[Folded_clause],New_Program).

%replace(Old_P,New_P,Replace,Position)
replace(L1,L2,Z,X):-
        rep(L1,L2,Z,X,0).

%rep(Old_P,New_P,Replace_clause,Position,Cnt)

rep([],[],Z,Cnt1,Cnt2).
rep([Z|T],[H|T2],Z,Cnt,Cnt):-
        Cnt1 is Cnt+1,
        rep(T,T2,Z,Cnt,Cnt1).
rep([H|T],[H|T2],Z,X,Cnt):-
        Cnt1 is Cnt+1,
        rep(T,T2,Z,X,Cnt1).

%-------------------------------------------------------------
get_applied_B([H1|T1],[B|T2],Comp_theta,H1,B,Applied_B,Folded_clause):-
        apply([B],Comp_theta,Applied_B),
        get_folded_clause(H1,Applied_B,Folded_clause).

get_folded_clause(H1,[Applied_B],[H1,Applied_B]).

show_1(N1,-17):-
        print('***Will try to fold C'),
        print(N1),print(' with clause D'),
        print('***'),nl,
        print('Selected D'),print(': ').

show_1(N1,N2):-
        print('***Will try to fold C'),
```

```
            print(N1),print(' with clause C'),
            print(N2),
            print('***'),nl,
            print('Selected C'),print(N2),print(': ').

%------------------------------------------------------------
f_request(X,Program,Definition,UniTerm):-
        \+exists(X),
            print('command does not exist'),nl,
        f_shell(Program,Definition,UniTerm).
f_request(X,Program,Definition,UniTerm):-
        exists(X),
        X,
        f_shell(Program,Definition,UniTerm).

%prepare_f1(N1,Clause1,N2,Clause2,Theta1)
prepare_f1(N1,C1,N2,C2,T1,Sub):-
            %C1
            print('Select '),
            show_f_clause(N1,C1),
            %ask Term1
            ask_term2(C1,Term1,Z1),nl,
            %C2
            print('will be unified with term '),
            show_f_clause(N2,C2),
            %ask Term2
            ask_term2(C2,Term2,Z2),nl,!,
            %unify Term1 and Term2
            print('deriving theta '),print(Sub),print(':'),nl,nl,
            unf([Term1],[Term2],T1,N1,C1,N2,C2,Sub),
            print('Theta '),print(Sub),print(' :'),nl,
            print(T1),nl,nl.

unf(Term1,Term2,T1,N1,C1,N2,C2,Sub):-
            unify(Term2,Term1,T1).

unf(Term1,Term2,T1,N1,C1,N2,C2,Sub):-
            \+unify(Term1,Term2,T1),
            print('failed to derive theta '),print(Sub),
            print(', try again'),nl,
```

```
            prepare_f1(N1,C1,N2,C2,T1,Sub).

show_f_clause(-17,Clause):-
            %C1
            print('from D'),
            print(': '),
            nl.
show_f_clause(Number,Clause):-
            %C1
            print('from C'),
            print(Number),print(': '),
            nl.

f_merge([],[],[]).
f_merge([],T,T).
f_merge(T,[],T).
f_merge([H1|T1],[H2|T2],[H1,H2|T]):-
            f_merge(T1,T2,T).

%-----------------------------------------------------------------
%check if term is part of new clause
ask_term2(Y,Temp,Z):-
%       print('Fold using term (of '),
        print(Y),print(')-No "[]":'),nl,read(Z),!,check3(Y,Temp,Z),nl,nl.
check3(Y,Temp,Z):-
            \+member(Z,Y),print(Z),print(' is not part of '),
            print(Y),nl,!,ask_term2(Y,Temp,Z1).
check3(Y,Z,Z):-
            member(Z,Y).
%-----------------------------------------------------------------
%get clause from program through number
%Program : the given program
%Term    : the returning clause
%Number  : the numeric input
%N       : internal number, returns clause's number

get_term1(Program,Term,Number,N):-
            read(Number),
            N1 is Number-1,
            get_f_t1(Program,N1,Term,N).
```

```
get_f_t1(Program,N,Term,N1):-
	\+derive_term(Program,N,Term),
print('failed to derive item, number does not represent item in group'),
	get_term1(Program,Term,Temp,N1).

get_f_t1(Program,T,Term,T1):-
	T1 is T + 1,
	derive_term(Program,T,Term).

%use d to fold with new definition
get_term2(Program,Term,Number,N,Definition):-
	read(Number),
	get_f_t2(Program,Number,Term,N,Definition).

get_f_t2(Program,T,Term,T1,Definition):-
	derive_term(Program,T,Term).

get_f_t2(Program,d,Definition,-17,Definition).

get_f_t2(Program,N,Term,N1,Definition):-
	\+derive_term(Program,N,Term),
	nl,
	print('invalid input, try again:'),
	get_term2(Program,Term,Temp,N1,Definition).

%term to store object program in program from file
store_fl(X):-
	see(X),store_file1(D).
store_file1(D):-
	read(A),store_file(A,[]).
store_file(A,D):-
	A==end_of_file,!,seen,print('session closed'),nl,
	main(D).
store_file(A,D):-
	!,\+A==end_of_file,
	store_file1([D,A]).

%empty(X)	: gives to variable X the empty value.
empty([]).
```

```
%--------------------------------------------------------------------------%
%export
export1(P):-
        print('Converting program to non-ground form:'),nl,
        export_progr(P).

%--------------------------------------------------------------------------%
%New definition
new_definition(P,New_D,P1,New_D1):-
print_in_window(
'Type the new definition you wish to add, or type cancel to abort',
'New Definition Input'),!,
read(A),sub_n_def(P,New_D,[A],P1,New_D1).

sub_n_def(P,New_D,[cancel],P,New_D):-
        print_in_window('Action canceled.').

sub_n_def(P,New_D,D1,P1,New_D1):-
        apply_n_definition(P,D1,P1),
        apply_n_definition(New_D,D1,New_D1),
        print('Process completed'),nl.

apply_n_definition([],C,C).

apply_n_definition([H|P],C,[H|P1]):-
        apply_n_definition(P,C,P1).

%--------------------------------------------------------------------------%
%Main
%--------------------------------------------------------------------------%
:- interface,main(P).
%--------------------------------------------------------------------------%
```

- **Rename.pl**

```
%A1=[[a(x,v(1)),b(v(2),c),q(v(1),v(2))],[r(a,v(3)),r(v(2),v(1))]]
%"v" stands for variable
%rename(Clauses,New_Clauses)    -       uses variable count
%rename1(Clauses,New_Clauses,Max) - Max default Constant
```

```
%  Initialize rename by
%                   var_up(Program,Renamed_Program,Returns_Max_var_index)
%
%rename_program sorts variables
%                   rename_program(Program,New_D,Renamed,Return_Number_X)

rename_program(List1,New_D,List2,X):-
        print('Program variables will be pushed above (2x)'),
        %count(List1,X1),
        merge11(List1,[New_D],List01),
        var_count(List01,List_u,X),
print('max var_index is '),print(X),nl,
        X1 is X*2,
rename1(List1,List_up,X1),
        print(X),nl,nl,nl,
        print('Program moved up...'),nl,
        display_P(List_up,0,Y),
        nl,nl,
%rename starting from 1
        ren_program(List_up,List2,0),
        print('Program renamed'),nl,
        display_P(List2,0,Z).

%------------------------------------------------------
%move variables above current max (to prevent conflict of
%variables with new ones
var_count(List1,List2,X):-
%move variables to list
        mv_var_prg(List1,Var),
%convert multilist to single list
        pt(Var,L),
%find max number in list
        find_max(L,X).

var_up(List1,List2,X):-
%move variables to list
        mv_var_prg(List1,Var),
%convert multilist to single list
        pt(Var,L),
%find max number in list
```

```
        find_max(L,X),
%push variables above max to avoid conflict
        X1 is X*2,
        rename1(List1,List2,X1).

ren_program([],[],X).

ren_program([List1|T1],[List2|T2],X):-
        rename(List1,List2,X,Max),
        count(List1,X1),
        ren_program(T1,T2,Max).

%Renames a clause
%List1 is clause
%List2 is New Clause

rename(List1,List2,Last_max,Max):-
        mv_to_list(List1,List1_var),
        rename_list(List1_var,List2_var,Last_max),
        find_max(List2_var,Max),
        rename0(List1,List2,List2_var).

mv_var_prg([],[]).
mv_var_prg([Clause1|T],[Var1|T2]):-
        mv_to_list(Clause1,Var1),
        mv_var_prg(T,T2).



%------------------------------------------------------%
rename1([],[],Num).
%v(X) is an object variable
rename1([v(X)|T],[v(X2)|T2],Num):-
     X2 is X + Num,rename1(T,T2,Num).

%H is an atom
rename1([H|T],[H|T2],Num):-
     atom(H),rename1(T,T2,Num).

%H is Term
rename1([H|T],[H1|New],Num):-
```

```
        functor(H,Name,Arity),Arity>0,
        rename_term(H,H1,Num),
        rename1(T,New,Num).

%T1 is a term
rename_term(T1,T2,Num):-
        T1=..[F1|List_Terms],
        rename1(List_Terms,List2,Num),
        T2=..[F1|List2].

%--------------------------------------------------------
rename0([],[],Num).
%v(X) is an object variable
rename0([v(X)|T],[v(X2)|T2],[X2|VT]):-
        rename0(T,T2,VT).

%H is an atom
rename0([H|T],[H|T2],Num):-
        atom(H),rename0(T,T2,Num).

%H is Term
rename0([H|T],[H1|New],Num):-
        functor(H,Name,Arity),Arity>0,
        rename_term0(H,H1,Num),
            count([H],Vars),
            remove_vars(Num,Vars,New_Num,0),
        rename0(T,New,New_Num).

%T1 is a term
rename_term0(T1,T2,Num):-
        T1=..[F1|List_Terms],
        rename0(List_Terms,List2,Num),
        T2=..[F1|List2].

%--------------------------------------------------------
%remove from list Num, Vars items and return New_Num
remove_vars([H|T],X,T2,Counter):-
            Cnt1 is Counter+1,
            remove_vars(T,X,T2,Cnt1).
remove_vars(T,X,T,X).
```

```
%-----------------------------------------------------------%
%counts the number of variables within a program
%(not individuals)

count([],0).

%count individuals
%v(X) is an object variable,
count([v(X)|T],N1):-
	count(T,N),N1 is N+1.

%H is an atom
count([H|T],Num):-
	atom(H),count(T,Num).

%H is Term
count([H|T],Num):-
	functor(H,Name,Arity),Arity>0,
	count_term(H,N),
	count(T,Num1),
	    Num is Num1+N.

%T1 is a term
count_term(T1,Num):-
	T1=..[F1|List_Terms],
	count(List_Terms,Num).
%-----------------------------------------------------------%
%move variables to list - only one clause per time
mv_to_list([],[]).
%count individuals
%v(X) is an object variable,
mv_to_list([v(X)|T],[X|T1]):-
	mv_to_list(T,T1).
%H is an atom
mv_to_list([H|T],List):-
	atom(H),mv_to_list(T,List).
%H is Term
mv_to_list([H|T],List):-
	functor(H,Name,Arity),Arity>0,
```

```
        mv_to_list_term(H,List1),
        mv_to_list(T,List2),
            merge11(List1,List2,List).
%T1 is a term
mv_to_list_term(T1,Num):-
        T1=..[F1|List_Terms],
        mv_to_list(List_Terms,Num).
%-------------------------------------------------------------%
%rep_var(X,List1,Y,List2)
%replace variable X in List1 with Y and return List2
rep_var(X,[],Z,[]).
rep_var(X,[X|I],Int,[Int|I2]):-
            rep_var(X,I,Int,I2).
rep_var(X,[A|I],Int,[A|List]):-
            rep_var(X,I,Int,List).

%-------------------------------------------------------------%
%merge11(List1,List2,List3)
%merge list1 with list2 and return list3

merge11([],T,T).
merge11([H|T1],List,[H|List2]):-
            merge11(T1,List,List2).

%-------------------------------------------------------------%
%renames the initial var list to the ideal one
rename_var([],L,L2,L).

rename_var([H|T],List1,[N|T2],L3):-
            rep_var(H,List1,N,L2),
            rename_var(T,L2,T2,L3).

rename_list(List1,List2,Last_max):-
            individuals(List1,L_gs),
            cnt(L_gs,X),
            X1 is X+Last_max,
            create_list(List3,X1,Last_max),
            reverse(List3,List4),
            !,
            rename_var(L_gs,List1,List4,List2).
```

```
%-----------------------------------------------------------%
%count numbers in list
cnt([],0).
cnt([A|T],N):-
        cnt(T,N1),N is N1+1.

%create list of numbers starting from X.
%List,max+startpoint,max
create_list([],X,X).
create_list([A|T],A,X):-
        create_list(T,A1,X),
        A is A1+1.
%find individual items in list and return them in another
%list
individuals([],[]).
individuals([A|T],[A|T1]):-
        \+member(A,T),
        individuals(T,T1).
individuals([A|T],T1):-
        member(A,T),
        individuals(T,T1).
%find max number X from List
max([],T).
max([A|T],C):-
        C>=A,max(T,C).
find_max(L,X):-
        member(X,L),max(L,X).
%-------------------------------------------
%put numbers from multi-list to single list
put([],[],X).
pt(List1,H):-
        put(List1,List2,[]),
%reverse to get the last list that contains all vars
        reverse(List2,[H|T]).
put([H|T],[H1|T1],L):-
        merge11(H,L,H1),
        put(T,T1,H1).
put([H],[],L).
%-----------------------------------------------------------%
```

```
%export program to non - ground form
export_progr([]).
export_progr([C|T]):-
        export_clause(C),
        put(8),print('.'),nl,
        export_progr(T).

%export clause to non-ground form - only one clause per time
export_clause([H|T]):-
        export_head([H],T),
        export_tail(T).

export_head(H,[]):-
        export_tail(H).
export_head(H,T):-
        export_tail(H),
        put(8),print(':-'),nl,
        space(8).

export_tail(T):-
        export(T,44).

export([],ASCII).
export([[]],ASCII).

export([v(X)|T],ASCII):-
        name(X,Z),
        merge11([88],Z,Var),
        name(V,Var),
        print(V),
        put(ASCII),
        export(T,ASCII).

export([A|T],ASCII):-
        atom(A),
        print(A),put(ASCII),
        export(T,ASCII).

%functor is a list
export([H|T],ASCII):-
```

```
        functor(H,'.',Arity),Arity>0,
            print('['),
        export_term(H,124),
            put(8),
            print(']'),put(ASCII),
        export(T,ASCII).

%functor is a term
export([H|T],ASCII):-
        functor(H,Name,Arity),Arity>0,
            print(Name),print('('),
        export_term(H,ASCII),
            put(8),
            print(')'),put(ASCII),
        export(T,ASCII).

export_term(T1,ASCII):-
            T1=..[A|T],
            export(T,ASCII).
```

- **Unify.pl**

```
%               Ακολουθεί ο αλγόριθμος ταυτοποίησης (unify)
%       Η ερώτηση θα έχει τη μορφή
%               ?-unify([p(a,f(v(1),c),b)],[p(v(2),f(b,v(3)),v(1))],Theta).
%       και επιστρέφει
%       Theta=[subst(v(3),c),subst(v(1),b),subst(v(2),a)]

unify(T1, T2, Subst) :-
    unify1(T1, T2, [], Subst).
unify1([], [], Subst_acc, Subst_acc).

% TERM CASES FOR INDIVIDUAL VARIABLES
% if T1 and T2 are diffent INDIVIDUAL variables unify them
unify1([v(N1)|StackT1], [v(N2)|StackT2], Subst_acc, Subst) :-
   N1 \== N2,
   replace_occurrences_in_stacks(subst(v(N1),v(N2)), StackT1, StackT2,
                                 New_stackT1, New_stackT2),
   make_indempotent_subst(subst(v(N1),v(N2)), Subst_acc, Subst_acc1), !,
   unify1(New_stackT1, New_stackT2, [subst(v(N1),v(N2))|Subst_acc1],
```

```
                                        Subst).
% if T1 is an INDIVIDUAL variable and T2 is a term unify them
unify1([v(N1)|StackT1], [T2|StackT2], Subst_acc, Subst) :-
    \+ occurs(v(N1), T2),
    replace_occurrences_in_stacks(subst(v(N1),T2), StackT1, StackT2,
                                  New_stackT1, New_stackT2),
    make_indempotent_subst(subst(v(N1),T2), Subst_acc, Subst_acc1), !,
    unify1(New_stackT1, New_stackT2, [subst(v(N1),T2)|Subst_acc1], Subst).

% if T1 is a term and T2 is an INDIVIDUAL variable unify them
unify1([T1|StackT1], [v(N2)|StackT2], Subst_acc, Subst) :-
    \+ occurs(v(N2), T1),
    replace_occurrences_in_stacks(subst(v(N2),T1), StackT1, StackT2,
                                  New_stackT1, New_stackT2),
    make_indempotent_subst(subst(v(N2),T1), Subst_acc, Subst_acc1), !,
    unify1(New_stackT1, New_stackT2, [subst(v(N2),T1)|Subst_acc1], Subst).

% GENERAL TERM CASES
% if T1 and T2  are identical atomic or non-atomic terms or identical
% variables, continue
unify1([T1|StackT1], [T2|StackT2], Subst_acc, Subst) :-
    T1 == T2,
    unify1(StackT1, StackT2, Subst_acc, Subst).
% if T1 and T2  are non-atomic terms with same functor push subterms into stack
unify1([T1|StackT1], [T2|StackT2], Subst_acc, Subst) :-
    T1 =.. [F1|Subterms1],
    T2 =.. [F2|Subterms2],
    F1 == F2,
    functor(T1, F1, Arity1),
    functor(T2, F2, Arity2),
    Arity1 == Arity2,
    push_subterms(Subterms1, Subterms2, StackT1, StackT2, New_stackT1,
                                               New_stackT2), !,
    unify1(New_stackT1, New_stackT2, Subst_acc, Subst).


% =========================================================
% The terms Terms1 and Terms2 are pushed into stacks StackT1 and StackT2
% respectively after they are reversed.
```

```
push_subterms(Terms1, Terms2, StackT1, StackT2, New_stackT1, New_stackT2) :-
  reverse(Terms1, Terms1_rev),
  reverse(Terms2, Terms2_rev),
  push(Terms1_rev, Terms2_rev, StackT1, StackT2, New_stackT1, New_stackT2).

push([], [], StackT1, StackT2, StackT1, StackT2).
push([T1|Terms1], [T2|Terms2], StackT1, StackT2, New_stackT1, New_stackT2) :-
   push(Terms1, Terms2, [T1|StackT1], [T2|StackT2], New_stackT1, New_stackT2).

% ==========================================================
% Occur check: It checks if Var occurs in Term.
occurs(Var, Term) :-
    Var == Term.
occurs(Var, Term) :-
   \+ atomic(Term),
   \+ member(Term, [var(N1), tvar(N2)]),
   functor(Term, F, Arity),
   occurs(Arity, Var, Term).
occurs(Arity, Var, Term) :-
   Arity > 1,
   Arity1 is Arity-1,
   occurs(Arity1, Var, Term).
occurs(Arity, Var, Term) :-
   arg(Arity, Term, Subterm),
   occurs(Var, Subterm).
% ----------------------------------------------------------------------
member(X,[X|l]).
member(X,[A|l]):-member(X,l).
% ----------------------------------------------------------------------
reverse([],[]).
reverse([H|T],T2):-
        reverse(T,T3),
        add(T3,[H],T2).
% ----------------------------------------------------------------------
add([H|T],T2,[H|T3]):-
        add(T,T2,T3).
        add([],T,T).
% ==========================================================
replace_occurrences_in_stacks(subst(T1,T2), StackT1, StackT2,
                                            New_stackT1, New_stackT2) :-
```

```
  replace_occurrences(subst(T1,T2), StackT1, New_stackT1),
  replace_occurrences(subst(T1,T2), StackT2, New_stackT2).

replace_occurrences(subst(T1,T2), [], []).
replace_occurrences(subst(T1,T2), [Top|Rest_stack], [New_top|New_stack]) :-
  replace_term_occurrences(subst(T1,T2), Top, New_top),
  replace_occurrences(subst(T1,T2), Rest_stack, New_stack).

replace_term_occurrences(subst(T1,T2), Term, T2) :-
  T1 == Term.
replace_term_occurrences(subst(T1,T2), Term, Term) :-
  atomic(Term).
% Case for compound term
replace_term_occurrences(subst(T1,T2), Term, New_term) :-
  Term =.. [F|Subterms],
  replace_term_occurrences1(subst(T1,T2), Subterms, New_subterms),
  New_term =.. [F|New_subterms].

replace_term_occurrences1(subst(T1,T2), [], []).
replace_term_occurrences1(subst(T1,T2), [SubT|Rest_subterms],
                          [T2|Rest_new_subterms]) :-
  T1 == SubT,
  replace_term_occurrences1(subst(T1,T2), Rest_subterms, Rest_new_subterms).
replace_term_occurrences1(subst(T1,T2), [SubT|Rest_subterms],
                          [SubT|Rest_new_subterms]) :-
  atomic(SubT),
  replace_term_occurrences1(subst(T1,T2), Rest_subterms, Rest_new_subterms).
replace_term_occurrences1(subst(T1,T2), [SubT|Rest_subterms],
                          [SubT|Rest_new_subterms]) :-
  member(SubT, [v(N1), tv(N2)]),
  replace_term_occurrences1(subst(T1,T2), Rest_subterms, Rest_new_subterms).
% Case for compound subterm
replace_term_occurrences1(subst(T1,T2), [SubT|Rest_subterms],
                         [New_subT|Rest_new_subterms]) :-
  replace_term_occurrences(subst(T1,T2), SubT, New_subT),
  replace_term_occurrences1(subst(T1,T2), Rest_subterms, Rest_new_subterms).

% ======================================================
make_indempotent_subst(subst(T1,T2), [], []).
make_indempotent_subst(subst(T1,T2), [subst(Var, Term)|Rest_subst],
```

```
                    [subst(Var, New_Term)|Rest_New_Subst]) :-
  replace_term_occurrences(subst(T1,T2), Term, New_Term),
  make_indempotent_subst(subst(T1,T2), Rest_subst, Rest_New_Subst).
% ==========================================================
```

- **Apply.pl**

```
%A1=[[p(v(1),v(2)),q(a,v(1)),r(v(1),v(2))],[p(v(3),v(4)),q(b,v(4))]]
%Theta=[subst(v(1),b),subst(v(2),v(6)),subst(v(3),c),subst(v(4),v(7))]

%apply(Clauses,Theta,New_clauses)

apply([],Theta,[]).

%H is an atom
apply([H|T],Theta,[H|New]):-
      atom(H),
      apply(T,Theta,New).
apply([H|T],Theta,[Theta1|New]):-
      apply_Theta(H,Theta,Theta1),
      apply(T,Theta,New).

%H is an object variable
apply_Theta(v(X),[],v(X)).
apply_Theta(v(X),[subst(v(X),B)|T],B).
apply_Theta(v(X),[subst(A,B)|T],Theta1):-
      apply_Theta(v(X),T,Theta1).

%H is a term
apply_Theta(T1,Theta,New):-
      T1=..[F1|List],
      apply(List,Theta,New_arg),
      New=..[F1|New_arg].

%sub_apply(Theta1,Theta2,New_theta).
sub_apply([],Theta,[]).
%H is an atom
sub_apply([H|T],Theta,[H|New]):-
      atom(H),
      sub_apply(T,Theta,New).
```

```
sub_apply([H|T],Theta,[Theta1|New]):-
        sub_apply_Theta(H,Theta,Theta1),
        sub_apply(T,Theta,New).

%H is an object variable
sub_apply_Theta(v(X),[],v(X)).
sub_apply_Theta(v(X),[subst(v(X),B)|T],B).
sub_apply_Theta(v(X),[subst(A,B)|T],Theta1):-
        sub_apply_Theta(v(X),T,Theta1).

%H is a term
sub_apply_Theta(T1,Theta,New):-
        T1=..[subst,A|List],
        sub_apply(List,Theta,New_arg),
        New=..[subst,A|New_arg].
sub_apply_Theta(T1,Theta,New):-
        T1=..[F1|List],
        sub_apply(List,Theta,New_arg),
        New=..[F1|New_arg].
```

- **Unfold.pl**

```
%P: Initial program
%New_P: New program after unfold
%New_Clause: New definition clause
%UniTerm: Term to use for unifying during unfolding

unfold(P,New_P,Clause,UniTerm):-
    print_in_window(
    'Removing clause that will be used for unfolding from program'),
    remove_clause(P,Clause,P1),
    print_in_window('Renaming program to '),nl,
%find max var_index and rename above it
    var_up(P1,P0,Var_max),!,
    nl,print('variables moved above 2x'),print(Var_max),nl,
    display_P(P0,0,X),nl,
    !,
     unfold1(P0,New_P,Clause,UniTerm),

     display_old_program(P0,Clause,UniTerm),
```

```
    display_P(New_P,0,X1),
    space(6),string(200,1),string(205,63),nl.

%----------------------------------------------------
remove_clause([],X,[]).
remove_clause([X|T],X,T1):-
    remove_clause(T,X,T1).
remove_clause([X|T],Y,[X|T1]):-
    remove_clause(T,Y,T1).
%----------------------------------------------------

unfold1([],[],New_Clause,UniTerm):-
    space(6),string(201,1),string(205,63),nl.

%New clause can be unified with H1
%unfold1(P0,NEW_P,New_Clause,UniTerm).

unfold1([[H1|T1]|Tail],[[H1|T1],New_clause_on_C|New_P],
New_Clause,UniTerm ):-
     member(UniTerm,New_Clause),
    print('Proceeding with unify...'),nl,
     print(H1),print(' & '),print(UniTerm),nl,
    unify([H1],[UniTerm],Theta),
    space(6),string(205,63),nl,
    space(6),print('Theta:'),print(Theta),nl,
    space(6),print('of '),print(H1),nl,
    space(6),print(' and '),print(UniTerm),print(''),nl,

    print('Forming New Clause:'),nl,
    form_new_clause(T1,Tail,New_Clause,UniTerm,Formed_clause),
    print(Formed_clause),nl,nl,
    print('applying theta ('),print(Theta),print(')'),nl,nl,
    print('New clause is:'),nl,
%temporary line
    apply(Formed_clause,Theta,New_clause_on_C),
    print(New_clause_on_C),nl,nl,
    print('New clause is added to old program...'),nl,nl,
%old one
     unfold1(Tail,New_P,New_Clause,UniTerm).
```

```
%New clause cannot be unified with H1
unfold1([[H1|T1]|Tail],[[H1|T1]|New_P],
New_Clause,UniTerm):-
     member(UniTerm,New_Clause),
     \+unify([H1],[UniTerm],Theta),
    print('failed to unify '),print(H1),print(' with '),
    print(UniTerm),nl,
    space(6),string(205,63),nl,
    print('unify failure:Predicates '),
     print(H1),print(' and '),print(UniTerm),
    print(' do not match.'),nl,nl,
     unfold1(Tail,New_P,New_Clause,UniTerm).

%UniTerm is not part of new clause
unfold1([C1|Tail],[C1|New_P],New_Clause,UniTerm):-
     \+member(UniTerm,New_Clause),
    print(UniTerm),
    print(' is not member of '),
    print(New_Clause),nl,
    print('Proceeding to next clause'),nl,nl,
     unfold1(Tail,New_P,New_Clause,UniTerm).

%terms to display the process
display_old_program(P0,New_Clause,UniTerm):-
    space(6),string(186,1),print('          Old program'),nl,
    space(6),string(199,1),string(196,63),nl,
    display_P(P0,0,X),
    space(6),string(199,1),string(196,63),nl,
    space(6),string(186,1),print('Clause used for unfolding:'),nl,
    space(6),string(186,1),      print(New_Clause),nl,
    space(6),string(199,1),string(196,63),nl,
    space(6),string(186,1),print('Term used for unification:'),
    print(UniTerm),nl,
    space(6),string(199,1),string(196,63),nl,
    space(6),string(186,1),print('          New program'),nl,
    space(6),string(199,1),string(196,63),nl.

display_P([],X,X).

display_P([H|T],X,Z):-
```

```
    space(6),string(186,1),Y is X+1,print('C'),
    print(Y),print(': '),print(H),nl,
    display_P(T,Y,Z).

display_D([],X,X).

display_D([H|T],X,Z):-
    space(6),string(186,1),Y is X+1,print('D'),
    print(Y),print(': '),print(H),nl,
    display_D(T,Y,Z).

display_D_Clause(Clause,X):-
    space(6),string(186,1),print(' D'),print(X),print(': '),
    print(Clause),nl.

display_Clause(Clause,X):-
    space(6),string(186,1),print(' C'),print(X),print(': '),
    print(Clause),nl.

%term to read from file
input_fl(X):-
    see(X),read_file1.
read_file1:-
    read(A),read_file(A).
read_file(A):-
    A==end_of_file,seen,print('session closed'),nl.
read_file(A):-
    !,\+A==end_of_file,
    print('execute: '),print(A),nl,
    A,!,
    read_file1.

%form new clause BEFORE apply
form_new_clause(T1,Rest_Program,[H1|New_Clause],
    UniTerm,[H1|Formed_clause]):-
        form1(T1,New_Clause,UniTerm,Formed_clause).

form1([],[],UniTerm,[]).
form1([],[UniTerm|Tail],UniTerm,Tail2):-
    form1([],Tail,UniTerm,Tail2).
```

```
form1([],[Q1|Tail],UniTerm,[Q1|Tail2]):-
    form1([],Tail,UniTerm,Tail2).
form1(T1,[],UniTerm,[]).

form1([R1],[UniTerm|Tail],UniTerm,[R1|Formed_clause]):-
    form1([R1],Tail,UniTerm,Formed_clause).

form1(R1,[Q1|Tail],UniTerm,[Q1|Formed_clause]):-
        form1(R1,Tail,UniTerm,Formed_clause).
```

- **Fold.pl**

```
%-------------------------------------------------------------------
%fold(Init_Program,Term_to_apply_to_by#,Term_to_fold_by#,New_Program).
%Init_Program    : Initial clause
%New_Program     : Clause after folding
%Term_to_use_by# : Term by number in Initial program, that will be used
%                for folding the Initial program

%*******************************************************************
        %Term_to_apply : clause where transformation is applied
        %Term          : clause we use for folding
        %FoldTerm      : term used  for unifying above clauses
        %Folded_clause : folded clause returned from transformation
        %fold(Term_to_apply,Term,FoldTerm,Folded_clause)



%*******************************************************************
fold(Init_Program,New_Program,Term_by_No):-
        derive_term(Init_Program,Term_by_No,Term),!,
        derive_theta(Init_Program,Term,List_of_composed_thetas),!,
        fold1(Inti_Program,New_Program,Term,List_of_composed_thetas).

%derive theta1 from intitial program and add it to list1
derive_theta([],Term,[]).
derive_theta([A|T],Term,[Comp_theta|T2]):-
        derive_t1(A,Term,Theta1),
        derive_t2(A,Term,Theta2),
        compose_theta(Theta1,Theta2,Comp_theta),
```

```
	derive_theta(T,Term,T2).

derive_t1([],Term,[]).
derive_t1([H,A|T],Term,Theta1):-
	print('deriving theta1 from '),print(A),
	print(' and '),print(Term),nl,
	unify([A],Term,Theta1),
	print('theta1 derived...{'),
	print(Theta1),print('}'),nl.

derive_t1([H,A|T],Term,Theta1):-
	\+unify([A],Term,Theta1),
	print('unify on theta1 failed!'),nl.

derive_t2([],Term,[]).
derive_t2([H,A,B|T],Term,Theta2):-
	print('deriving theta2 from '),print(B),
	print(' and '),print(Term),nl,
	unify([B],[Term],Theta2),
	print('theta2 derived...{'),
	print(Theta2),print('}'),nl.

derive_t2([H,A,B|T],Term,Theta2):-
	\+unify([B],[Term],Theta2),
	print('unify on theta2 failed!'),nl.

%derive term from program using number (0 for first term)
derive_term(A,I,B):-
	derive1(A,I,B,0).
derive1([],0,[],0).
derive1([],I,[],J):-
	fail.
derive1([H|T],I,H,J):-
	J==I.
derive1([H|T],I,B,J):-
	\+ J==I,
	K is J+1,
	derive1(T,I,B,K).

%term used to remove undesired substitutions
```

```
remove_yl([],[],[]).
remove_yl([],T,T).
remove_yl(T,[],T).
remove_yl([subst(A,B)|T1],[subst(C,D)|Theta2],[subst(A,B),subst(C,D)|T]):-
        remove_yl(T1,Theta2,T).
remove_yl([subst(Ul,Yl)|T1],[subst(Vl,Yl)|Theta2],[subst(Ul,Yl)|T]):-
        remove_yl(T1,Theta2,T).
%term used to compose theta1*theta2
compose_theta(Theta1,Theta2,Comp_theta):-
        sub_apply(Theta1,Theta2,New_Theta),!,
        print('Theta2 applied on Theta1:'),nl,print('Theta1:'),
        print(Theta1),nl,print('Theta2:'),
        print(Theta2),nl,print('New_Theta:'),
        print(New_Theta),nl,
        print('-----'),nl,nl,
        remove_yl(New_Theta,Theta2,Temp),
        print('Thetas merged:'),nl,
        print(Temp),nl,
        print('-----'),nl,nl,
        remove_xx(Temp,Comp_Theta),
        print('removed any subst(x,x)'),nl,
        print(Comp_Theta),nl,
        remove_dbl(Comp_Theta,Comp_theta).
%term produces List1&List2 from List1,List2
merge_([],[],[]).
merge_([],List2,List2).
merge_(List1,[],List1).
merge_([subst(X1,Y1)|T1],[subst(X2,Y2)|T2],[subst(X1,Y1),subst(X2,Y2)|C]):-
        \+X1==X2,
        merge_(T1,T2,C).

%remove self substitutions
remove_xx([],[]).
remove_xx([subst(X,X)|T],B):-
        remove_xx(T,B).
remove_xx([subst(A,B)|T],[subst(A,B)|C]):-
        remove_xx(T,C).

%remove double items from list
remove_dbl([],[]).
```

```
remove_dbl([H1|T1],[H1|T2]):-
        \+member(H1,T1),
        remove_dbl(T1,T2).
remove_dbl([H1|T1],T2):-
        member(H1,T1),
        remove_dbl(T1,T2).
```