Φιλοσοφία Γλωσσών Προγραμματισμού

Προγραμματιστικό Μοντέλο	**Γλώσσες**
Προστακτικός Προγραμματισμός (Imperative Programming)	FORTRAN, Algol, COBOL Pascal, C, Ada, Pazcal
Συναρτησιακός Προγραμματισμός (Functional Programming)	LISP, ML, Scheme, Haskell
Λογικός Προγραμματισμός (Logic Programming)	Prolog
Αντικειμενοστρεφής Προγραμματισμός (object-oriented programming)	Simula, Smalltalk, C++ Java, C#

Στάθης Ζάχος

Κεφάλαιο 1

Συναρτησιακός Προγραμματισμός

Ο συναρτησιακός προγραμματισμός τραβάει ιστορικά τις ρίζες του από τον λ-λογισμό του Church και την συνδυαστική λογική του Curry. Μία από τις πρώτες γλώσσες με μεγάλη διάδοση πού στηριζόταν στον λ-λογισμό είναι η LISP (John McCarthy, 1961). Στην δεκαετία του ΄80 ο συναρτησιακός προγραμματισμός (Functional Programming) πήρε νέα ώθηση από τις εργασίες των John Backus, David Turner, Robin Milner κτλ.

Αυτό που προσπαθούμε να κάνουμε τον συναρτησιακό προγραμματισμό είναι να ορίσουμε μία συνάρτηση που να επιλύει το ζητούμενο πρόβλημα. Αυτή η συνάρτηση μπορεί βέβαια να χρησιμοποιεί άλλες συναρτήσεις (ή ακόμα και τον εαυτό της).

Ο ρόλος του υπολογιστή είναι απλός: Του δίνουμε μία έκφραση και μας την αποτιμά. Αν μέσα στην έκφραση αυτή εμφανίζεται το όνομα μίας συνάρτησης που έχουμε ορίσει, τότε ο υπολογιστής χρησιμοποιεί τον ορισμό προκειμένου να 'ανάγει' την έκφραση σε μία μορφή που θα τον φέρει κοντύτερα στο επιδιωκόμενο αποτέλεσμα, που δεν είναι άλλο από το να αποτιμήσει την έκφραση που του δώσαμε.

Ας υποθέσουμε ότι έχουμε ανοικτό το τερματικό του υπολογιστή (terminal) και φορτώνουμε την γλώσσα FP (μια υποθετική συναρτησιακή γλώσσα). Βρισκόμαστε σε ένα διαλογικό περιβάλλον (interactive environment) που συγκρατεί και χρησιμοποιεί όλους τους ορισμούς που έχετε κάνει. Αμέσως εμφανίζεται στην οθόνη το prompt ? και ο υπολογιστής περιμένει να του δώσετε μια έκφραση για να την υπολογίσει. Έστω πως εσείς γράφετε:

```
?  2+2
```

ο υπολογιστής θα απαντήσει

```
4
```

Σε αυτήν την περίπτωση ανήγαγε την αρχική έκφραση '2+2' στην απλούστερη '4'. Η πιο ενδιαφέρουσα όμως περίπτωση είναι να δημιουργήσουμε ένα σύνολο ορισμών, που ονομάζεται script, και να το υποβάλλουμε στον υπολογιστή. Έστω το παρακάτων script:

```
double x = 2*x
min x y = x, if x <= y
          y, if x > y
```

Η σημασία είναι ότι ορίσαμε δύο συναρτήσεις την double και την min. Η double έχει μία παράμετρο, την x, και δίνει ώς αποτέλεσμα το διπλάσιο της x. Προσέξτε ότι ο ορισμός έχει την μορφή εξίσωσης που λέει ότι η έκφραση 'double x' μπορεί να αντικατασταθεί (να αναχθεί) από την '2*x'.

Μπορεί τώρα να γίνει ο εξής διάλογος με τον υπολογιστή:

```
? double (min 3 4)
6
```

1.1 Σειρά των αναγωγών (Reductions)

Ο υπολογιστής προκειμένου να παράγει το αποτέλεσμα 6 έκανε μια σειρά αναγωγών: χρησιμοποίησε την προκαθορισμένη σημασία του τελεστή * και το script που του δώσαμε. Η διαδικασία της εκτέλεσης του προγράμματος δεν είναι τίποτα περισσότερο από μία ακολουθία αντικαταστάσεων (αναγωγών) κάποιων εκφράσεων από άλλες ισοδύναμές τους. Για παράδειγμα, για την παραπάνω έκφραση υπάρχει η εξής ακολουθία αναγωγών:

$$\texttt{double}\ (3+4) \rightarrow \texttt{double 7} \rightarrow \texttt{2*7} \rightarrow \texttt{14}$$

αλλά υπάρχει και αυτή η ακολουθία αναγωγών:

$$\texttt{double}\ (3+4) \rightarrow \texttt{2*}(3+4) \rightarrow \texttt{2*7} \rightarrow \texttt{14}$$

Ας ονομάσουμε ΑΕ (αναγώγιμη έκφραση) κάθε έκφραση που εμφανίζεται στην πορεία των αναγωγών και υπάρχει η δυνατότητα να αναχθεί. Στο παραπάνω παράδειγμα ΑΕ είναι οι `double` $(3+4)$, `2*7`, `3+4` κτλ. Το 7 ή το 14 μόνα τους δεν είναι ΑΕ αφού δεν μπορούν να αναχθούν σε τίποτα άλλο.

Η πρώτη ακολουθία αναγωγών συμφωνεί με την τακτική των 'εσωτερικότερων' αναγωγών (innermost reductions) ενώ η δεύτερη με την τακτική των 'εξωτερικότερων' αναγωγών (outermost reductions). Συγκεκριμένα η τακτική των 'εσωτερικότερων' αναγωγών λέει ότι κάθε φορά αντικαθιστούμε μία ΑΕ η οποία δεν περιέχει άλλη ΑΕ. Για παράδειγμα στην αρχική έκφραση αντικαθιστούμε την ΑΕ $(3+4)$ που δεν περιέχει καμία ΑΕ.

Αντίθετα, η τακτική των 'εξωτερικότερων' αναγωγών λέει να αντικαθιστούμε κάθε φορά μία ΑΕ που δεν περιέχεται από άλλη ΑΕ. Οι 'εξωτερικότερες' αναγωγές έχουν το εξής πλεονέκτημα: Εφόσον υπάρχει η δυνατότητα η ζητούμενη έκφραση να αναχθεί σε κάποιο αποτέλεσμα, εφαρμόζοντας 'εξωτερικότερες' αναγωγές θα φτάσουμε σε αυτό το αποτέλεσμα. Αυτό δεν συμβαίνει πάντα με τις 'εσωτερικότερες' αναγωγές. Αυτή η διαπίστωση επιβεβαιώνεται από το παρακάτω παράδειγμα:

```
f x y = if (x == 0) then 1 else y ?  f 0 (1/0)
```

Με την τακτική των εσ. αν. ο υπολογιστής θα επιδιώξει να αναγάγει την ΑΕ 1/0 πράγμα που είναι αδύνατο. Κατά συνέπεια δεν θα βγάλει αποτέλεσμα. Με την τακτική των εξ. αν. θα δώσει αποτέλεσμα 1 χωρίς να απασχοληθεί καν με την αναγωγή του 1/0, αφού άλλωστε δεν πρόκειται το 1/0 να χρησιμοποιηθεί. Μάλιστα η τακτική των εξ. αν. λέγεται και **lazy evaluation (οκνηρή αποτίμηση)** αφού δεν αναγάγει κάποιο ΑΕ αν αυτή η αναγωγή δεν είναι απαραίτητη.

Ας δούμε μερικά παραδείγματα:

Παράδειγμα 1:

```
inc n = n + 1
f t = t * inc t

y = f (f 2)
```

Σημείωση: Οι τύποι των παραπάνω συναρτήσεων/τιμών είναι:

```
inc, f :: Int -> Int
x, y :: Int
```

Ας ανάγουμε την έκφραση y χρησιμοποιώντας εσωτ. αναγωγές:

$$\texttt{y} \rightarrow \texttt{f (f 2)} \rightarrow \texttt{f (2 * inc 2)} \rightarrow \texttt{f (2*(2+1))} \rightarrow \texttt{f (2*3)} \rightarrow \texttt{f 6} \rightarrow \texttt{6 * inc 6} \rightarrow \texttt{6*(6+1)} \rightarrow \texttt{6*7} \rightarrow \texttt{42}$$

Χρησιμοποιώντας εξωτ. αναγωγές βλέπουμε ότι καταλήγουμε στο ίδιο αποτέλεσμα αλλά με σημαντικά περισσότερο αριθμό βημάτων:

`y → f (f 2) → f 2 * inc (f 2) → (2 * inc 2) * inc (2 * inc 2) → (2*(2+1)) * inc (2*(2+1)`
`→ (2*(2+1)) * (2 * (2+1) + 1) → ... → 42`

Παράδειγμα 2:

```
f(x, y) = if (x == 0) then 1 else f(x-1, f(x, y))
```

Η οκνηρή αποτίμηση δίνει π.χ.:

$$f(2,0) \to f(1, f(2,0)) \to f(0, f(1, f(2,0))) \to 1\,.$$

ενώ με τις εσωτ. αναγωγές έχουμε:

$$f(2,0) \to f(1, f(2,0)) \to f(1, f(1, f(2,0))) \to \ldots$$

η οποία δεν σταματάει!

Πάντως ισχύει ότι εάν δύο διαφορετικοί τρόποι αποτίμησης σταματήσουν (καταλήξουν), τότε η τιμή θα είναι ίδια (δηλαδή μοναδική). Αυτή η τιμή λέγεται **κανονική μορφή** (normal form). Μπορεί όμως να διαφέρει σημαντικά ο αριθμός των αναγωγών ως την κανονική μορφή.

1.2 Συναρτήσεις

Στον συναρτησιακό προγραμματισμό, οι συναρτήσεις μπορεί να παίρνουν ως παραμέτρους άλλες συναρτήσεις και μπορεί και το αποτέλεσμα τους να είναι μία συνάρτηση. Για παράδειγμα μπορούμε να ορίσουμε μια συνάρτηση `twice` η οποία θα δέχεται ως πρώτο όρισμα μια συνάρτηση f και έναν ακέραιο x και επιστρέφει ακέραιο. Η f θα είναι μια συνάρτηση που δέχεται έναν ακέραιο και επιστρέφει ακέραιο (οι πληροφορίες για τους τύπους των μεταβλητών μπορούν να γραφούν στο πρόγραμμα Haskell όπως φαίνεται παρακάτω):

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

inc n = n + 1
plus2 x = twice (inc, x)
```

Άλλο ένα παράδειγμα που δείχνει ότι μια συνάρτηση μπορεί να επιστρέψει σαν αποτέλεσμα άλλη συνάρτηση:

```
plusN :: Int -> (Int -> Int)
plusN x = f
   where f y = x + y
```

1.2.1 Ανώνυμες Συναρτήσεις

Σε συναρτησιακές γλώσσες υπάρχει η δυνατότητα δήλωσης **ανώνυμων** συναρτήσεων όπως και στις προστακτικές γλώσσες μπορούμε να χρησιμοποιήσουμε αριθμητικές σταθερές χωρίς να τους δώσουμε όνομα (στις συναρτησιακές γλώσσες οι συναρτήσεις και οι αριθμητικές τιμές είναι αντικείμενα πρώτης τάξης).

Για παράδειγμα η συνάρτηση που απεικονίζει το n στο $n+1$ μπορεί να οριστεί ανώνυμα ως εξής:`\n -> n + 1`.

Άλλα παραδείγματα:

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

plus2 :: Int -> Int
plus2 x = twice (\n -> n + 1, x)

plusN :: Int -> (Int -> Int)
plusN x = \y -> x + y
```

1.3 Currying

Μια συνάρτηση με δύο παραμέτρους ισοδυναμεί με μια συνάρτηση που δέχεται την πρώτη παράμετρο και επιστρέφει μια συνάρτηση που δέχεται τη δεύτερη. Αυτή η ιδέα, η χρήση δηλαδή της δεύτερης ισοδύναμης μορφής των συναρτήσεων με πολλές παραμέτρους ονομάζεται currying. Ας δούμε μερικά παραδείγματα:

Παράδειγμα 1:

Η συνάρτηση max:

```
max :: (Int, Int) -> Int
max (x, y) = if x > y then x else y
```

Μπορεί να γραφεί διαφορετικά με την τεχνική currying ως εξής:

```
max' :: Int -> (Int -> Int)
max' x = \y -> (if x > y then x else y)
```

Η max παίρνει ένα όρισμα τύπου Int και δίνει μία συνάρτηση τύπου `Int -> Int`. Άρα μία έκφραση σαν την max 2 5 ισοδυναμεί με (max 2) 5 παρά με max(2, 5). Αυτή είναι και η ουσία του currying: Μπορούμε να αντικαταστήσουμε μια συνάρτηση n ορισμάτων με μία $(n-1)$ ορισμάτων αφού εφαρμόσουμε το πρώτο όρισμα, κατόπιν μια $(n-2)$ ορισμάτων αφού εφαρμόσουμε το δεύτερο όρισμα, κοκ.

Παράδειγμα 2:

Έστω η συνάρτηση add:

```
add :: (Int, Int) -> Int
add (x, y) = x +  y
```

Μπορεί να γραφεί διαφορετικά με την τεχνική currying ως εξής:

```
add' :: Int -> (Int -> Int)
add' x = \y -> x + y
```

Δηλαδή η `add'` δέχεται το πρώτο όρισμα x και επιστρέφει μια συνάρτηση (την οποία ορίζει ανώνυμα) η οποία δέχεται το δεύτερο όρισμα y και επιστρέφει το άθροισμα x + y. Ισχύει λοιπόν η παρακάτω ισοδυναμία:

```
add (x, y) == (add' x) y
```

Η Haskell προσφέρει κι έναν τρόπο να γραφούν με απλούστερο τρόπο οι add', max':

```
add' :: Int -> Int -> Int
add' x y = x + y

max' :: Int -> Int -> Int
max' x y = if x > y then x else y
```

Με τις curried συναρτήσεις επιτρέπεται η 'μερική εφαρμογή'. Ας δούμε ένα παράδειγμα αναγωγών της έκφρασης `twice (add 20) 2`:

```
twice (add 20) 2 → add 20 (add 20 2) → add 20 (20+2) → 20+(20+2) → 42
```

1.4 Παραδείγματα

Η παρακάτω συνάρτηση δέχεται έναν ακέραιο n και υπολογίζει το παραγοντικό $n!$ του n:

```
factorial n =
   if n <= 1 then 1
             else n * factorial (n-1)
```

Παρακάτω βλέπουμε πώς μπορεί να γραφεί η συνάρτηση που υπολογίζει τον Μέγιστου Κοινό Διαιρέτη με χρήση του αλγόριθμους του Ευκλείδη σε Haskell χρησιμοποιώντας αναδρομή:

```
gcd (n, 0) = n
gcd (n, m) = if (n < m) then gcd (m, n)
             else            gcd (m, n `mod` m)
```

1.5 Επίλογος

Όπως θα έχετε ήδη παρατηρήσει οι συναρτήσεις στον συναρτησιακό προγραμματισμό δεν έχουν παρενέργειες (side effects). Η σημασία κάθε έκφρασης δεν είναι τίποτα περισσότερο από την τιμή της. Ένα συναρτησιακό πρόγραμμα δεν είναι τίποτα παραπάνω από μία σειρά εξισώσεων ενώ η εκτέλεση ενός προγράμματος είναι μία σειρά αναγωγών. Συχνά η μαθηματική περιγραφή ενός προβλήματος είναι και το πρόγραμμα που το λύνει. Επιπλέον η απόδειξη της ορθότητας ενός προγράμματος μπορεί να γίνει εύκολα με έλεγχο των αντικαταστάσεων (αναγωγών) που έχουμε ορίσει, ενώ η απόδειξη ορθότητας αναδρομικών συναρτήσεων γίνεται με χρήση μαθηματικής επαγωγής.

Κεφάλαιο 2

Λογικός Προγραμματισμός (Prolog)

2.1 Ιστορία

Η γλώσσα Prolog, η οποία αναπτύχθηκε σε ορισμένα ερευνητικά ινστιτούτα της Γαλλίας και της Μ. Βρετανίας στα μέσα της δεκαετίας 1970-1980 με επικεφαλή τον Alain Colmerauer, και η οποία είναι η πιο πετυχημένη προσπάθεια για την επίτευξη του στόχου που καλείται 'Λογικός Προγραμματισμός'. Η Prolog (αρχικά τού PROgramming in LOGic) υιοθετεί μια τελείως διαφορετική φιλοσοφία υψηλού επιπέδου (ηιγη λεελ) από αυτήν των συνηθισμένων διαδικαστικών ή προστακτικών γλωσσών προγραμματισμού (Pascal, C, Basic, Fortran κτλ.), και γι΄ αυτό ακριβώς είναι ευκολότερο να μάθει να σκέφτεται σ΄ αυτή τη γλώσσα κάποιος που δεν έχει καμμία εμπειρία από προγραμματισμό, παρά ο έμπειρος προγραμματιστής.

Η Prolog υλοποιεί ένα υποσύνολο του κατηγορηματικού λογισμού πρώτης τάξης (first order logic or predicate calculus). Στο πρόγραμμα δεν υλοποιούμε τις ρουτίνες επίλυσης του προβλήματος, απλώς δίνουμε τα δεδομένα του προβλήματος και τους κανόνες που διέπουν το πρόβλημα (τους νόμους που επιδρούν πάνω στα δεδομένα) σε μορφή σύμφωνη με το συντακτικό της Prolog.

Έτσι, ένα πρόγραμμα Prolog δεν είναι τίποτα άλλο από μία συλλογή δεδομένων: γεγονότων (facts) και κανόνων (rules) που επιδρούν στα γεγονότα αυτά. Κατά τη διάρκεια της εκτέλεσης παράγονται νέα δεδομένα, αλλάζουν δεδομένα, ή και αποσύρονται (χάνονται) ορισμένα από αυτά.

Η Prolog είναι μια πολύ χρήσιμη γλώσσα για προβλήματα που περιγράφονται με κάποιους (συμβολικούς) περιορισμούς πάνω στις παραμέτρους (μεταβλητές) του προβλήματος. Εξ΄ αιτίας αυτού του γεγονότος η Prolog έχει επικρατήσει μαζί με τη Lisp στο χώρο της τεχνητής νοημοσύνης.

2.2 Η φιλοσοφία της γλώσσας Prolog

2.2.1 Δεδομένα-Τρόπος

Όπως έχει ήδη ειπωθεί, η φιλοσοφία και δομή της Prolog δεν είναι η ίδια με αυτή των διαδικαστικών ή πιο γενικά προστακτικών γλωσσών. Σε αυτές, ο προγραμματιστής που επιθυμεί να γράψει ένα πρόγραμμα για την επίλυση ενός προβλήματος, περιγράφει τη μέθοδο επίλυσης του προβλήματος και τα δεδομένα που χρειάζονται.

Αντίθετα στην Prolog ο προγραμματιστής περιγράφει μόνο τα δεδομένα, δηλαδή το 'χώρο' του προβλήματος και όχι τον τρόπο με τον οποίο θα βρεθούν σε αυτόν οι απαντήσεις (ο τρόπος όπως θα φανεί παρακάτω είναι ένας, καθώς κάθε πρόγραμμα Prolog μπορεί να θεωρηθεί τύπος της Λογικής, και έτσι ουσιαστικά υλοποιείται με μία μόνο διαδικασία (αλγόριθμο): την εξαντλητική αναζήτηση επαλήθευσης του τύπου μέσω του backtracking και του unification. Κατόπιν, ο προγραμματιστής 'ρωτάει' την Prolog συγκεκριμένα ερωτήματα τα οποία το σύστημα Prolog προσπαθεί να απαντήσει.

Για παράδειγμα, έστω ότι ο προγραμματιστής θέλει να φτιάξει ένα πρόγραμμα που δοκιμάζει και βρίσκει ποιός από κάποιους δυνατούς συνδυασμούς ανοίγει μία υποθετική κλεκτρονική κλειδαρια. Ο ψευδοκώδικας σε προστακτική γλώσσα θα ήταν κάπως έτσι:

```
begin
   repeat
      find the next combination;
      try the current combination to check if it is correct;
   until the right combination has been found
end
```

Σε αντίθεση, ο ψευδοκώδικας στην Prolog θα ήταν κάπως έτσι:

```
Ο Α είναι δυνατός συνδυασμός αν έχει αυτά και αυτά τα χαρακτηριστικά.
Ο Β είναι συνδυασμός που ανοίγει την κλειδαριά αν έχει αυτά τα χαρακτηριστικά.
```

Κατόπιν, ο προγραμματιστής θα 'ρωτούσε' το σύστημα:
`Ποιός συνδυασμός είναι δυνατός και ανοίγει την κλειδαριά?`

Τότε, το σύστημα βρίσκει ένα δυνατό συνδυασμό, ελέγχει αν είναι δεκτός (δηλαδή αν ανοίγει την κλειδαρία) και τότε σταματά, αλλιώς συνεχίζει μέχρι να βρεί κάποιον που να ικανοποιεί τα παραπάνω.

2.2.2 Δηλώσεις — κατηγορήματα και γεγονότα — κανόνες

Η βασική έννοια που χρησιμοποιείται στην περιγραφή των δεδομένων είναι το κατηγόρημα. Το κατηγόρημα (predicate) είναι ουσιαστικά μια ιδιότητα. Βλέπουμε, για παράδειγμα, ότι οι φράσεις:

```
Ο Χ είναι ψηλός.          % κατηγόρημα μιας θέσης
ή
Χ + 2 = 4.                % κατηγόρημα μιας θέσης
ή
Χ + Υ = 5.                % κατηγόρημα δύο θέσεων
```

δηλώνουν ιδιότητες κάποιων αντικειμένων.

Βλέπουμε ότι τοποθετώντας κάποια συγκεκριμένα αντικείμενα στις θέσεις ενός κατηγορήματος, κατασκευάζουμε μια πρόταση:

```
Ο Γιάννης είναι ψηλός.    % βάζουμε τη λέξη Γιάννης
ή
2 + 2 = 4.                % βάζουμε τον αριθμό 2
ή
2 + 2 = 5.                % βάζουμε τον αριθμό 2 στην πρώτη θέση
                            και τον αριθμό 2 στη δεύτερη θέση.
```

Οι προτάσεις που κατασκευάζονται με αυτόν τον τρόπο είναι ατομικές προτάσεις στον κατηγορηματικού λογισμό. Μεταφράζοντας τέτοιες προτάσεις σε μορφή κατανοητή από την Prolog κατασκευάζουμε τα γεγονότα (facts).

Το τελευταίο παράδειγμα δηλώσεων ('$2 + 2 = 5$') δείχνει ότι μια δήλωση που κάνουμε στην Prolog δεν είναι κατ' ανάγκη αληθής στον πραγματικό κόσμο, εφόσον η Prolog δεν ασχολείται (και δεν μπορεί να ασχοληθεί) με το εάν οι δηλώσεις που κάνουμε είναι αληθείς ή όχι. Η Prolog απλά θεωρεί αληθές ό,τι προκύπτει από τις δηλώσεις. Η λεπτομέρεια αυτή, δηλαδή ότι όλα τα ερωτήματα που έχουν νόημα για την Prolog, αφορούν το κατά πόσον κάποιες προτάσεις προκύπτουν ή όχι από τις δηλώσεις που έχουν γίνει και μόνο, αποτελεί ένα πολύ σημαντικό χαρακτηριστικό της Prolog που είναι γνωστό σαν 'υπόθεση κλειστού κόσμου' (closed world assumption).

Τα παραπάνω παραδείγματα θα μπορούσαν να γραφούν στην Prolog σαν γεγονότα ως εξής:

```
tall(john).
equals( plus(2, 2), 4 ).
equals( plus(2, 2), 5 ).
```

Οι κανόνες τώρα, είναι γενικά προτάσεις του κατηγορηματικού λογισμού που περιγράφουν μια συνεπαγωγή (implication). Δηλαδή, προσδιορίζουμε τις συνθήκες που εάν ισχύουν τότε ισχύει και το συμπέρασμα. Οι παρακάτω προτάσεις αποτελούν κανόνες:

```
Ο Χ είναι μηχανικός αν ο Χ έχει τελειώσει πολυτεχνείο και είναι έμπειρος.
Ο Χ είναι ρητός αν ο Χ μπορεί να γραφτεί στη μορφή Α/Β, όπου Α, Β φυσικοί
   και Β όχι ίσος με μηδέν.
```

Ας γράψουμε τους παραπάνω κανόνες πιο τυπικά στη γλώσσα της Συμβολικός Λογικής:

$$(\text{Χ είναι μηχανικός}) \Leftarrow (\text{Ο Χ έχει τελειώσει πολυτεχνείο}) \wedge (\text{Ο Χ είναι έμπειρος})$$
$$(X \in \mathbb{Q}) \Leftarrow ((X = A/B) \wedge (A \in \mathbb{Z}) \wedge (B \in \mathbb{Z}) \wedge (B \neq 0))$$

Σε μορφή κανόνων στην Prolog:

```
engineer(X) :- graduate(X), experience(X).
rational(X) :- X = A/B, integer(A), integer(B), not(B = 0).
```

Εν όψει των παραπάνω, ο ψευδοκώδικας σε Prolog για το παράδειγμα που αφορούσε την κλειδαριά θα μπορούσε να γραφτεί ως εξής:

$$(\text{Χ είναι δυνατός συνδυασμός}) \Leftarrow (\text{Χ έχει αυτό το χαρακτηριστικό}) \wedge (\text{Χ έχει το άλλο χαρακτηριστικό}).$$
$$(\text{Χ ανοίγει την κλειδαριά}) \Leftarrow (\text{Χ έχει αυτήν την ιδιότητα}) \wedge (\text{Χ έχει την άλλη ιδιότητα}).$$

και η ερώτηση ως εξής:

$$\text{Υπάρχει Χ τέτοιο ώστε (Χ είναι δυνατός συνδυασμός) και (Χ ανοίγει την κλειδαριά)?}$$

2.2.3 Εντολές

Θα πρέπει ήδη να έχει γίνει φανερό ότι άλλη μια σημαντική διαφορά μεταξύ Prolog και προστακτικών γλώσσών είναι η ανυπαρξία εντολών με την προστακτική έννοια, εφόσον η επίλυση προβλημάτων απαιτεί πλέον μόνο την περιγραφή των δεδομένων και των ερωτησεων. Στο πράδειγμα με την κλειδαριά είναι προφανής ο προστακτικός χαρακτήρας του προγράμματος σε προστακτική γλώσσα. Αντίθετα, σε Prolog δεν υπάρχει ένδειξη προσταγής, τουλάχιστον γλωσσική.

Ουσιαστικά, μπορεί να ειπωθεί ότι στις προστακτικές γλώσσες υπάρχουν δύο γενικές κατηγορίες εντολών (χωρίς αυτό να αποτελεί ένα εντελώς γενικό και ισχύον σχόλιο):

1. 'κάνε αυτό' (π.χ. WRITELN("...");) [assignment and procedure calls]

2. 'αν συμβαίνει αυτό και αυτό, κάνε αυτό' (π.χ. if (a == b) then WRITELN("...");) [control statements]

Αντίθετα, στην Prolog δεν υπάρχει καμία από τις παραπάνω κατηγορίες εντολών. Μάλιστα, αν εντολή ονομάζουμε την προσταγή προς τον υπολογιστή, φτάνουμε στο συμπέρασμα ότι στην Prolog δεν υπάρχουν καθόλου εντολές, παρά μόνον δηλωτικές προτάσεις (δηλαδή δηλώσεις γενονότων και κανόνων) και ερωτήσεις.

Με άλλα λόγια, εφόσον δεν περιγράφεται ο τρόπος επίλυσης στην Prolog, δεν υπάρχει η ανάγκη προσταγής του υπολογιστή, παρά μόνον ίσως στην διαδικασία διατύπωσης των ερωτημάτων, όπου θα μπορούσε να θεωρηθεί ότι οι ερωτήσεις αποτελούν εντολές της μορφής 'βρες την απάντηση της ερώτησης: ... '.

2.2.4 Παραδείγματα

Παράδειγμα 1:

Γεγονότα:

```
male(john).
male(george).
female(mary).
female(jenny).
parent(john,george).
parent(mary,george).
parent(john,jenny).
parent(mary,jenny).
```

Κανόνες:

```
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
human(X) :- male(X).
human(X) :- female(X).
brother(X,Y) :-
male(X), parent(Z,X), parent(Z,Y).
sister(X,Y) :-
female(X), parent(Z,X), parent(Z,Y).
```

Ερωτήσεις:

```
?- male(john).
yes

?- male(mary).
no

?- male(peter).
no

?- male(X).
X = john ;
X = george ;
no

?- human(X).
X = john;
X = george;
X = mary;
X = jenny;
no

?- brother(george,jenny).
yes

?- mother(X,george).
X = mary;
```

```
no

?- sister(X,Y).
X = jenny, Y = george;
X = jenny, Y = jenny;
X = jenny, Y = george;
X = jenny, Y = jenny;
no
```

Παράδειγμα 2 (Μέγιστος Κοινός Διαιρέτης):

```
gcd(X, 0, X).
gcd(0, X, X).
gcd(X, Y, D) :- X < Y,
                Y1 is Y mod X,
                gcd(X, Y1, D).
gcd(X, Y, D) :- Y < X,
                gcd(Y, X, D).
```

Παράδειγμα 3 (Παραγοντικό):

```
factorial(0, 1).
factorial(N, Result) :- N > 0,
                        N1 is N-1,
                        factorial(N1, Facmin),
                        Result is Facmin*N.
```

2.3 Επίλογος

Προσπαθήσαμε να δώσουμε τις αρχές που διέπουν την φιλοσοφία του προγραμματισμού στην Prolog, γλώσσας πάρα-πολύ-υψηλού επιπέδου, η οποία έχει χρησιμοποιηθεί με τεράστια επιτυχία τόσο στην τεχνητή νοημοσύνη, όσο και σε άλλες περιοχές της επιστήμης των υπολογιστών. Μέσα από αυτά που είπαμε ελπίζουμε να δώσουμε το ερέθισμα για μια πιο βαθιά μελέτη της γλώσσας αυτής, μιας γλώσσας που εκτός από την ευχέρειά της στον συμβολικό υπολογισμό (symbolic computation) και την δυνατότητά της να υποχρεώνει σχεδόν το χρήστη να γράφει ευανάγνωστα προγράμματα, που να φανερώνουν τη λογική τους, έχει και το μεγάλο πλεονέκτημα να είναι εύκολο να υλοποιηθεί σε παράλληλες μηχανές.

Κεφάλαιο 3

Αντικειμενοστρεφής Προγραμματισμός

Object Oriented Programming (Java)

Υπάρχουν τόσες διαφορετικές απόψεις για το τι είναι ο OOP, όσοι και οι μηχανικοί πληροφορικής. Ο όρος πρωτοχρησιμοποιήθηκε για να περιγράψει το προγραμματιστικό περιβάλλον της Smalltalk (72, 74, 76, 90, Xerox Parc). Το OOP δίνει έμφαση στη μοντελοποίηση προβλημάτων.

Ενα "πρόγραμμα" αποτελείται από πολλά αντικείμενα που συνεργάζονται για να πετύχουν το ζητούμενο αποτέλεσμα. Αντικείμενο είναι ενα ανεξάρτητο module, μια δομή που περιλαμβάνει δεδομένα αλλά και διαδικασίες, που αναφέρονται σ' αυτά, περιλαμβάνει δηλαδή **κατάσταση** και **συμπεριφορά**, είναι άρα έννοια υψηλότερου επιπέδου από τους τύπους δεδομένων ή συναρτήσεων που συναντήσαμε στις άλλες γλώσσες. Π.χ. στοίβες, ουρές, μεταγλωττιστές, λειτουργικά συστήματα είναι προγραμματιστικές δομές που μπορούν να θεωρηθούν αντικείμενα. Η ιδέα της **αφαίρεσης (abstraction)** είναι πολύ σημαντική (στη ζωή γενικά αλλά και) στον προγραμματισμό (ειδικά). Τα αντικείμενα αποτέλουν αφαιρέσεις (abstraction). Π.χ. μια δομή δέντρου με διαδικασίες για εύρεση στοιχείου, διαγραφή στοιχείου, καταγραφή στοιχείων, κ.τ.λ., αποτελεί ένα αντικείμενο. Ενα απλούστερο αντικείμενο είναι π.χ. ο ακέραιος τύπος (τύπος στοιχείου) με διαδικασίες για πρόσθεση, αφαίρεση, πολλαπλασιασμό και διαίρεση. Ενα πιο πολύπλοκο αντικείμενο είναι ενα λεξικό οργανωμένο σα δυαδικό δέντρο με διαδικασίες για ανεύρεση, διαγραφή, εισαγωγή λέξεων κ.ο.κ.

Η περιγραφή των ιδιοτήτων ενός αντικειμένου (π.χ. λεξικού) λέγεται **κλάση,** ενα λεξικό είναι απλώς ενα στιγμιότυπο της κλάσης "λεξικό". Οι κλάσεις είναι οργανωμένες ιεραρχικά, έτσι έχουμε **υποκλάσεις** και **υπερκλάσεις.** Τα αντικείμενα μιας κλάσεως **κληρονομούν (inherit)** όλες τις ιδιότητες της υπερκλάσης. Παράβαλε την ιεραρχική

ταξινόμηση των ζώων στη Ζωολογία π.χ.: Τα άλογα έχουν όλες τις ιδιότητες των θηλαστικών, τα θηλαστικά έχουν όλες τις ιδιότητες των ζώων, κ.ο.κ.

Ιστορικά πολλές από τις ιδέες του OOP προήλθαν από την Simula (1966, Oslo, Dahl &Nygard). Εννοίες όπως **μήνυμα, κλάση, αντικείμενο, ...,** πρωτοεμφανίστηκαν εκεί.

Μια γλώσσα πρέπει να έχει τέσσερα κύρια χαρακτηριστικά για να υποστηρίζει OOP: **Information Hiding, Data Abstraction, Dynamic Binding και Inheritance.**

Η Modula-2 και η Ada που δεν περιέχουν και τα τέσσερα χαρακτηριστικά δεν μπορούν να θεωρηθούν ΟΟ γλώσσες. Επίσης η C++ και η Turbo Pascal (προεκτάσεις της C και Pascal αντίστοιχα) απλώς υποστηρίζουν και ΟΟ χαρακτηριστικά, δεν είναι όμως αυθεντικές OOP γλώσσες. Το OOP δεν εξαρτάται απλώς από τη γλώσσα προγραμματισμού, αλλά βασικά και από το ελκυστικό προγραμματιστικό περιβάλλον.

ΜΗΝΥΜΑΤΑ ΑΝΤΙ ΔΙΑΔΙΚΑΣΙΩΝ

Οι περισσότερες γλώσσες προγραμματισμού υποστηρίζουν τις διαδικασίες επί δεδομένων (Data procedure paradigm): Ο διαδικασίες ενεργούν πάνω σε παθητικά δεδομένα . Π.χ. η sqr(x) παίρνει έναν αριθμό και επιστρέφει το τετράγωνό του.

Σε μια γλώσσα όπως η Pascal υπάρχουν διαφορετικές εκδόσεις της sqr(x) για κάθε τύπο του x, που συνήθως επιστρέφουν ως αποτέλεσμα ένα αριθμό ιδίου τύπου με το όρισμα (πολυμορφισμός). Η Lisp αντίθετα εξακριβώνει τον τύπο του x στο χρόνο εκτέλεσης και εκτελεί τις κατάλληλες πράξεις γι' αυτό τον τύπο δεδομένων. Αλλο παράδειγμα σε Pascal μπορεί να είναι η mult(a,b), όπου το είδος του πολλαπλασιασμού εξαρτάται σημαντικά από τον τύπο των a,b (εάν π.χ. a και b είναι πίνακες τότε mult(a,b) σημαίνει πολλαπλασιασμό πινάκων). Επίσης στην Pascal οι διαδικασίες read και write είναι πολυμορφικές.

Στις ΟΟ γλώσσες αντί να περνάμε δεδομένα σε διαδικασίες που θα τα επεξεργαστούν, αναθέτουμε στα αντικείμενα (δεδομένα) να τις εκτελέσουν μόνα τους. Π.χ. για την εύρεση του τετραγώνου του x έχουμε

x sqr

όπου το x εκτελεί την sqr διεργασία στον εαυτό του: το x είναι ο **παραλήπτης** (receiver) του μηνύματος sqr. Στο επόμενο παράδειγμα έχουμε το εσωτερικό γινόμενο δύο διανυσμάτων x και y:

x dot: y

όπου το x εκτελεί την πράξη dot με όρισμα το y. Μπορούμε να πάρουμε το τετράγωνο του εσωτερικού γινομένου και να την αναθέσουμε σε μια μεταβλητή z:

z:=(x dot:y)sqr.

Αλλα Παραδείγματα:

Receiver	Selector	Arguments
Frame	center	
45	+	count
printer	display	'Title'
frame	moveTo:	newLocation

Συνεπώς στο OOP υπολογισμός είναι η μετάδοση **μηνυμάτων** προς **αντικείμενα** που είναι υπεύθυνα για εκτέλεση κάποιας λειτουργίας. Ο **παραλήπτης** ενός μηνύματος χρησιμοποιεί κάποια **μέθοδο** για να ικανοποιήσει το αίτημα του **αποστολέα.**

ΟΡΟΛΟΓΙΑ

Τα x,y που αναφέραμε στο παραπάνω παράδειγμα είναι αντικείμενα, στιγμιότυπα μιας κλάσης. Η κλάση παρέχει όλες τις πληροφορίες που είναι απαραίτητες για την **κατασκευή** και χρήση αντικειμένων ενός ορισμένου τύπου (δηλ. των αντικειμένων της). Κάθε στιγμιότυπο ανήκει σε μια κλάση, κάθε κλάση μπορεί να έχει πολλά στιγμιότυπα.

Η κλάση περιέχει και τις **μεθόδους** (methods) που περιγράφουν πως συμπεριφέρονται τα στιγμιότυπα της κλάσης σε αντίστοιχα μηνύματα.

Κάθε στιγμιότυπο (εκτός από τις μεθόδους της κλάσης που μπορεί να χρησιμοποιεί) περιέχει και κάποιες ατομικές του μεταβλητές, τις Instance Variables. Οι μεταβλητές αυτές μπορεί να είναι π.χ. αριθμοί, χαρακτήρες, πίνακες κ.τ.λ., ή γενικά αντικείμενα άλλων κλάσεων.

Ο προγραμματισμός γίνεται με την αποστολή μηνυμάτων σε αντικείμενα τα οποία πυροδοτούν τις ομώνυμες μεθόδους στις κλάσεις των αντικείμενων αυτών. Η αποστολή ενός μηνύματος σε ενα αντικείμενο έχει σκοπό να μεταβάλει τις Instance Variables του, που σημαίνει ότι αυτόματα αποστέλλονται μηνύματα σε αυτές (αφού οι Instance Variables είναι και αυτές αντικείμενα) κ.ο.κ. μέχρι να φτάσουμε σε πρωταρχικές μεθόδους (Primitives Methods). Κάθε μήνυμα επιστρέφει τελικά το αποτέλεσμα στον αποστολέα. Δεν υπάρχει άλλος τρόπος μεταβολής των Instance Variables ενός αντικείμενου.

INFORMATION HIDING (ΑΠΟΚΡΥΨΗ ΠΛΗΡΟΦΟΡΙΑΣ)

Για να χρησιμοποιήσει κάποιος αποστολέας μηνύματος ενα αντικείμενο χρειάζεται απλά και μόνο να γνωρίζει ποιά μηνύματα μπορεί να λάβει αυτό και τι κάνει το καθένα. Δεν χρειάζεται να γνωρίζει λεπτομέρειες για τον τρόπο ικανοποίησης του αιτήματός του, π.χ. τις εσωτερικές μεταβλητές. Το Information Hiding είναι σημαντικό για έλεγχο ορθότητας και μετατρεψιμότητα ενός συστήματος λογισμικού. Η κατάσταση της μεταφραστικής μονάδας περιέχεται σε τοπικές μεταβλητές οι οποίες είναι ορατές μόνο από το εσωτερικό αυτής της μονάδας και όχι από το εξωτερικό της: έτσι μπορούν (με σωστό βέβαια σχεδιασμό του συστήματος) να αλλάξουν (π.χ. να βελτιστοποιηθούν) οι εσωτερικές διαδικασίες και μεταβλητές χωρίς να επηρεάζονται οι εξωτερικές σχέσεις (**interface**) αυτής της μονάδας με άλλες μονάδες. Βέβαια σχεδόν όλες οι μοντέρνες γλώσσες προγραμματισμού υποστηρίζουν το Information Hiding.

DATA ABSTRACTION (ΑΦΑΙΡΕΣΗ ΔΕΔΟΜΕΝΩΝ)

Data Abstraction είναι ένας συγκεκριμένος τρόπος πραγματοποίησης της αρχής του Information Hiding που έχει αναφερθεί στην περιγραφή της Modula-2 δηλαδή είναι ο πλήρης διαχωρισμός ορισμού και υλοποίησης ενός αντικειμένου, π.χ. μιας στοίβας.

DYNAMIC BINDING (ΔΥΝΑΜΙΚΗ ΔΕΣΜΕΥΣΗ)

Ενας βασικός περιορισμός της Modula-2 είναι ότι οι Abstract Data Types μπορούν να εφαρμοστούν μόνο σε έναν τύπο δεδομένων (στοίβα μόνο από π.χ. integers). Επίσης υπάρχει ο (δευτερεύων) περιορισμός ότι σε ένα module δεν μπορούμε να δηλώσουμε μεταβλητή με το ίδιο όνομα που έχει μια διαδικασία που εισάγεται (με IMPORT) απο άλλο module. Σε πολύ μεγάλα προγράμματα λοιπόν ο συσχετισμός ονομάτων μεταβλητών και διαδικασιών (ή και modules) δεν είναι εύκολος.

Στην Ada (μερικά) και στη Smalltalk (ολοκληρωτικά) αυτά τα προβλήματα λύνονται με χρήση operator overloading (υπερφόρτωση τελεστών) και generic program units. Η υπερφόρτωση του τελεστή επιτρέπει να χρησιμοποιούνται σε ένα πρόγραμμα διαφορετικοί τελεστές με το ίδιο όνομα. Ετσι ο τελεστής + μπορεί να χρησιμοποιηθεί τόσο για την πρόσθεση αριθμών όσο και διανυσμάτων κ.λ.π. Οι generic program units επιτρέπουν ορισμό ενός module με διαφορετικούς τύπους δεδομένων. Πώς όμως θα

αποθηκεύσουμε στο ίδιο stack διαφορετικούς τύπους δεδομένων; Η λύση είναι το Dynamic Binding.

Ας προσθέσουμε τη διαδικασία Print σε ενα StackHandler module που τυπώνει τα περιεχόμενα ενός stack. Αν ο stack αποθηκεύει ακέραιους, πραγματικούς, χαρακτήρες, κ.λ.π., τότε ο συμβατικός προγραμματισμός λέει ότι πρέπει να υπάρχει μια εντολή case (switch) που να ελέγχει στο χρόνο εκτέλεσης ποιά διαδικασία θα πρέπει να χρησιμοποιηθεί για κάθε τύπο δεδομένων. Αν τώρα εισάγουμε ένα νέο τύπο δεδομένων στο πρόγραμμά μας τότε θα πρέπει να μεταβάλλουμε όλες τις εντολές case και να κάνουμε ξανά compilation- μια χρονοβόρα διαδικασία που μπορεί τελικά να μας οδηγήσει και σε λάθη. Θα θέλαμε οι προσθέσεις νέων τύπων να απαιτούν μόνο προσθέσεις και όχι μετατροπές.

Στον OOP η ευθύνη για το τύπωμα (και όχι μόνο) των αντικειμένων ανήκει στα ίδια τα αντικείμενα. Σε κάθε αντικείμενο του stack αποστέλλεται το μήνυμα Print και αυτό το εκτελεί με τον κατάλληλο τρόπο. Αυτό είναι γνωστό σαν **Πολυμορφισμός** καθώς το ίδιο μήνυμα μπορεί να επιφέρει διαφορετικές αντιδράσεις ανάλογα με τους παραλήπτες. (Η υπερφόρτωση του τελεστή που γίνεται στην Ada δεν καλύπτει το πρόβλημα του Δυναμικού Πολυμορφισμού καθώς η διεύθυνση της διαδικασίας που καλείται καθορίζεται κατά το χρόνο μετάφρασης).

INHERITANCE (ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ)

Μήπως όμως χρειάζεται να ξαναγράψουμε εξ ολοκλήρου τη μέθοδο Print για κάθε αντικείμενο ακόμα και όταν αυτή είναι ίδια ή περίπου ίδια;

Η λύση είναι η κληρονομικότητα.

Μπορούμε δηλαδή να ορίσουμε τις κλάσεις ιεραρχικά και έτσι έχουμε αντικείμενα που είναι ειδικεύσεις άλλων αντικειμένων. Μπορούμε να κατασκευάσουμε ένα αντικείμενο Dictionary που είναι εξειδίκευση ενός αντικειμένου Set που είναι εξειδίκευση ενός γενικού αντικειμένου Collection. Προφανώς το Set κληρονομεί συμπεριφορά απο το Collection (ίσως και να κληρονομεί τη μέθοδο Print αυτούσια) και το Dictionary κληρονομεί συμπεριφορά απ΄ το Set. Ετσι δημιουργούμε **Υποκλάσεις** που είναι εξειδικεύσεις των **Υπερκλάσεών** τους. Η υποκλάση μπορεί να εμπλουτιστεί και με νέες Instance Variables και μεθόδους ή να διαφοροποιηθεί ως προς κάποιες από τις μεθόδους

που κληρονόμησε. Φυσικά αυτό δεν έχει καμία επίπτωση στην Υπερκλάση η οποία δεν κληρονομεί τίποτα από τις Υποκλάσεις της.

Η κληρονομικότητα επιτρέπει τον ορισμό κλάσεων αντικειμένων με τον καθορισμό των διαφορών τους από τις ήδη υπάρχουσες αντί για ξεκίνημα από το μηδέν κάθε φορά. Πολύς κώδικας εξοικονομείται με αυτόν τον τρόπο.

ΠΛΕΟΝΕΚΤΗΜΑΤΑ - ΜΕΙΟΝΕΚΤΗΜΑΤΑ ΑΝΤΙΚ/ΣΤΡΕΦΟΥΣ ΜΟΝΤΕΛΟΥ

Πλεονεκτήματα

- Το Information Hiding και ειδικότερα το Data Abstraction αυξάνουν την ασφάλεια και αξιοπιστία του κώδικα, διαχωρίζοντας την περιγραφή από την υλοποίηση.
- Το Dynamic Binding επιτρέπει εισαγωγή νέων κλάσεων χωρίς την ανάγκη μετατροπής του ήδη υπάρχοντος λογισμικού.
- Ο συνδυασμός του Inheritance και του Dynamic Binding επιτρέπει πολλαπλή χρήση του ίδιου κώδικα (προγραμματισμού) και σημαντική μείωση κώδικα για τις μεγάλες εφαρμογές.
- ΣΥΝΕΠΕΙΑ: εύκολη και γρήγορη ανάπτυξη πρωτότυπων συστημάτων λογισμικού. Εύκολη τροποποίησή και εύκολος εντοπισμός λαθών.

Μειονεκτήματα

- Απώλεια ταχύτητας: η αποστολή μηνυμάτων υλοποιείται πιο αργά από μια συμβατική κλήση συνάρτησης. Συνεπώς απώλεια αποδοτικότητας από πλευράς χρόνου αλλά και χώρου (υπάρχει σαφής διαφορά π.χ. στην αποδοτικότητα μεταξύ C και C++).
- Περίπλοκη υλοποίηση μιας ΟΟ γλώσσας. Υψηλό κόστος μεταγλωττιστού.
- Κόπος προγραμματιστού για εκμάθηση μεγάλου αριθμού κλάσεων που ήδη υπάρχουν σε ''βιβλιοθήκη''.