# Αρχές Γλωσσών Προγραμματισμού

## Εργασία 3

### Prolog Interpreter

Σκοπός της εργασίας αυτής είναι η υλοποίηση ενος απλουστευμένου διερμηνέα για την γλώσσα Prolog. Ο διερμηνέας αυτός θα υποστηρίζει δύο τρόπους λειτουργίας, οι οποίοι περιγράφονται αναλυτικά στη συνέχεια:

- Top-Down Evaluation
- Bottom-Up (Naive) Evaluation

Τα προγράμματα τα οποία θα φορτώνονται στον διερμηνέα θα αποτελούνται απο το βασικότερο υποσύνολο της γλώσσας Prolog, δηλαδή θα περιλαμβάνουν μονο *κανόνες* μεταξύ *κατηγορημάτων*, στα οποία θα συμμετέχουν *ελεύθερες μεταβλητές* και *όροι*, πιθανώς με συναρτησιακά σύμβολα. Για παράδειγμα:

```
sum(0,Y,Y).
sum(s(X),Y,Z) :- sum(X,s(Y),Z).
```

Ο διερμηνέας, αφού διαβάσει ενα αρχείο πηγαίου κώδικα Prolog, θα δέχεται ερωτήματα απο τον χρήστη στα οποία θα απαντά είτε με *true/false*, είτε δίνοντας κάποια κατάλληλη ανάθεση των ελεύθερων μεταβλητών.

Η εκπόνηση της εργασίας μπορεί να χωριστεί στα εξής λογικά μέρη:

### Parser (30%)

Σε πρώτη φάση, καλείστε να υλοποιήσετε εναν parser για την Prolog, ο οποίος θα δέχεται σαν είσοδο ένα string (τα περιεχόμενα ενός αρχείου) και θα επιστρέφει για κάθε σωστή, συντακτικά, περίπτωση το ενδιάμεσο representation, όπως παρουσιάζεται παρακάτω.

Στις περιπτώσεις όπου υπάρχει συντακτικό λάθος δεν χρειάζεται να το χειρίζεστε με συγκεκριμένο τρόπο, απλώς πρέπει αυτό να αναγνωρίζεται από τον κώδικα σας ώστε να μην συνεχίσει η εκτέλεση.

Παρακάτω ακολουθεί η context-free grammar σε BNF η οποία περιγράφει την γλώσσα της Prolog που θα κάνετε parse:

```
<Program> ::= <StmtList>

<StmtList> ::= <Statement> <StmtList>
             | ""

<Statement> ::= <Predicate> "."
              | <Predicate> ":-" <PredList> "."

<PredList> ::= <Predicate> <PredList2>

<PredList2> ::= "," <Predicate> <PredList2>
              | ""

<Predicate> ::= <Atom> "(" <TermList> ")"
              | <Atom>

<TermList> ::= <Term> <TermList2>

<TermList2> ::= "," <Term> <TermList2>
              | ""
```

```
<Term> ::= <Variable>
         | <Predicate>

<Atom> ::= "[a-z][a-zA-Z0-9]*"
         | <Integer>

<Variable> ::= "[a-zA-Z][a-zA-Z0-9]*"

<Integer> ::= "[1-9][0-9]*"
            | 0
```

Η έξοδος του parser είναι ένα Abstract Syntax Tree του εκάστοτε αρχείου που δόθηκε ως είσοδος. Το AST, είναι μία λίστα από ASTNode, ένα για κάθε statement του προγράμματος, ενω το ASTNode εμπίπτει στους εξής συντατικούς κανόνες της Haskell [1]:

```
data ASTNode
    |   Fact ASTNode
    |   Rule ASTNode [ASTNode]
    |   Predicate String [ASTNode]
    |   PredVariable String
```

- Facts είναι τα clauses της μορφής "*predicate* ." ενώ τα Rules έχουν την μορφή

  *predicate* :- *predicate*, *predicate*, ..., *predicate*.

  Στην περίπτωση του Fact, το ASTNode που αντιστοιχεί είναι αυτό που παράγει το Predicate.

  Στην περίπτωση του Rule, το πρώτο ASTNode είναι το head predicate και η λίστα είναι όλα τα predicates που βρίσκονται στο tail του clause.

- Κάθε predicate αποτελείται απο ένα string του ονόματος του και με μια λίστα από Predicates ή Variables, η οποία προφανώς μπορεί να είναι κενή.
- Τα Variables είναι τα strings που ξεκινάνε με κεφαλαίο γράμμα και περιγράφονται ως "PredVariable String".

Για παράδειγμα, για το ακόλουθο πρόγραμμα:

```
a(X) :- a(s(X)), b(Y).
b(15).
```

Το ενδιάμεσο representation θα είναι:

```
[
    Rule (PredVariable "X") [
        Predicate "a" [
            Predicate "s" [PredVariable "X"]
        ],
        Predicate "b" [
            PredVariable "Y"
        ]
    ],
    Fact (Predicate "15" [])
]
```

[1] Δεν είναι υποχρεωτικό να ακολουθήσετε τη συγκεκριμένη αναπαράσταση. Αυτή δίνεται για δική σας ευχέρεια.

## Pattern Matching Algorithm (40%)

Οι αλγόριθμοι εξαγωγής συμπερασμάτων Top-Down και Bottom-Up βασίζονται σε έναν μηχανισμό σύγκρισης εκφράσεων. Για παράδειγμα, στο απλό πρόγραμμα:

```
p(s(s(X)),Y).
```

Προκειμένου να απαντηθεί το ερώτημα `?- p(s(s(0)),1)` πρέπει να συγκριθούν οι δυο εκφράσεις, ώστε να αποφασιστεί εαν μπορούν να *ταιριάξουν*, δηλαδή εαν υπάρχει κάποια αντικατάσταση των ελεύθερων μεταβλητών, ώστε οι δύο εκφράσεις να ταυτίζονται.

Επομένως, ενα βασικό κομμάτι της υλοποίησης θα πρέπει να μοντελοποιεί την έννοια της *αντικατάστασης/ανάθεσης* σε μια έκφραση, όπως και την έννοια του *πιο γενικού ενοποιτή* (Most General Unifier) μεταξύ δύο ή περισσοτέρων εκφράσεων.

Ο πιο γενικός ενοποιητής μεταξύ δύο ή παραπάνω εκφράσεων είναι η αντικατάσταση η οποία:

- Εαν εφαρμοστεί επι των εκφράσεων, τότε αυτές θα ταυτιστούν μεταξύ τους
- Είναι η πιο *γενική* αντικατάσταση με αυτή την ιδιότητα

Έχοντας αυτά υπ'όψιν, καλείστε να υλοποιήσετε μια συνάρτηση η οποία θα δέχεται δύο *εκφράσεις* και θα επιστρέφει τον *πιο γενικό ενοποιτή* τους.

Στην πορεία πιθανώς να χρειαστεί να υλοποιήσετε και άλλες 'πράξεις' μεταξύ *αντικαταστάσεων* και *εκφράσεων*, όπως εφαρμογή αντικατάστασης σε έκφραση και σύνθεση αντικαταστάσεων.

## Top-Down Evaluation (30%)

Η αποτίμηση Top-Down είναι ένας αναδρομικός αλγόριθμος ο οποίος προσπαθεί να αναγάγει ένα ερώτημα σε γνωστά facts, χρησιμοποιώντας τους κανόνες. By default, οι περισσότεροι Prolog interpreters λειτουργούν σύμφωνα με αυτή την τεχνική.

Συγκεκριμένα, όταν η αποτίμηση Top-Down δέχεται ένα **ικανοποιήσιμο** ερώτημα, τότε επιστρέφει μια ανάθεση των ελεύθερων μεταβλητών για την οποία το ερώτημα ικανοποιείται. Η ανάθεση αυτή μπορεί να είναι και 'κενή' για ενα ικανοποιήσιμο ερώτημα.

Αντίθετα, όταν η αποτίμηση Top-Down δέχεται ενα **μη ικανοποιήσιμο** ερώτημα, τότε επιστρέφει οτι δεν υπάρχει κάποια κατάλληλη ανάθεση.

Συνοπτικά, η διαδικασία αποτίμησης λειτουργεί ως εξής:

- Για ένα δωθέν ερώτημα, ελέγχεται εαν αυτό μπορεί να ταιριάξει με την *κεφαλή* κάποιου κανόνα.
- Εάν βρεθεί *κεφαλή* κανόνα που ταιριάζει, τότε εφαρμόζεται η αντίστοιχη ανάθεση στην *ουρά* του κανόνα.

  Έπειτα, για κάθε κατηγόρημα της ουράς, σειριακά, καλείται αναδρομικά η διαδικασία αποτίμησης και οι αναθέσεις που προκύπτουν εφαρμόζονται στα κατηγορήματα που ακολουθούν.

  Εαν βρεθεί ανάθεση για όλα τα κατηγορήματα του κανόνα, τοτε το ερώτημα είναι ικανοποιήσιμο και η τελική ανάθεση προκύπτει απο τη **σύνθεση** των επί μέρους αναθέσεων.

  Εαν κάποια αναδρομική κλήση δεν επιστρέψει ανάθεση, τότε το αρχικό ερώτημα δεν μπορεί να ικανοποιηθεί βάσει του συγκεκριμένου κανόνα και αναζητείται άλλος κανόνας.
- Εάν τελικά δεν βρεθεί τέτοιος κανόνας, τότε το ερώτημα είναι μη ικανοποιήσιμο.

Να σημειωθεί οτι η σειρά με την οποία ελέγχονται οι κανόνες μπορεί να επηρεάσει καθοριστικά την εκτέλεση. Για τον λόγο αυτό, η σειρά ελέγχου θα είναι «απο πάνω προς τα κάτω», κατα σύμβαση.

Αφού υλοποιήσετε τον αλγόριθμο ώστε να βρίσκει την πρώτη ανάθεση κατά σειρά, δοκιμάστε να τον επεκτείνετε ώστε να μπορεί να συνεχίσει την αναζήτηση, κατ'επιλογή του χρήστη [2], κάνοντας backtrack, όμοια με τον swipl interpreter που έχουμε δεί στο μάθημα.

[2] Εδω θα χρειαστεί να χειριστείτε είσοδο-έξοδο

## Bottom-Up Evaluation (Bonus 30%)

Η αποτίμηση Bottom-Up είναι ένας επαναληπτικός αλγόριθμος ο οποίος σε κάθε βήμα διευρύνει το σύνολο των facts, βάσει των ήδη γνωστών facts.

Σε κάθε επανάληψη, για κάθε κανόνα επιλέγονται όλα τα facts που ταιριάζουν στην ουρά και η αντίστοιχη *ενοποιημένη* κεφαλή προστίθεται ως fact σε όλες τις επόμενες επαναλήψεις.

Η διαδικασία αυτη θα τερματίσει εαν σε κάποια επανάληψη δεν προστεθούν νεα facts στο σύνολο.

Ωστόσο, υπάρχουν προγράμματα Prolog τα οποία παράγουν καινούργια facts σε κάθε επανάληψη, με αποτέλεσμα αυτή η διαδικασία να μην τερματίζει ποτέ.

Παράδειγμα 1:

```
p(s(s(s(0)))).
p(X) :- p(s(X)).
```

Εκτέλεση:

```
S = { p(s(s(s(0)))) }
S = { p(s(s(s(0)))), p(s(s(0))) }
S = { p(s(s(s(0)))), p(s(s(0))), p(s(0)) }
S = { p(s(s(s(0)))), p(s(s(0))), p(s(0)) p(0) }
```

Μετά την τέταρτη επανάληψη δεν θα προστεθεί κανένα νεο fact στο σύνολο `S`, αρα ο αλγόριθμος θα τερματίσει.

Παράδειγμα 2:

```
p(X, Y) :- q(s(X)), q(Y).
q(X) :- q(s(X)).
q(s(s(0))).
```

Εκτέλεση:

```
S = { q(s(s(0))) }
S = { ... , q(s(0)) }
S = { ... , q(0) }
S = { ... , p(0,0) }
```

Όμοια εδώ, μετά την 4η επανάληψη ο αλγόριθμος τερματίζει.

Παράδειγμα 3:

```
p(s(X)) :- p(X).
p(0).
```

Εκτέλεση:

```
S = { p(0) }
S = { ... , p(s(0)) }
S = { ... , p(s(s(0))) }
S = { ... , p(s(s(s(0)))) }
...
```

Εδω ο αλγόριθμος δεν τερματίζει.

## Παραδοτέο

Για κάθε ενα απο τα μέρη της εργασίας, υλοποιήστε ενα αντίστοιχο πηγαίο πρόγραμμα Haskell με τις σχετικές συναρτήσεις.

Να συμπεριλάβετε και ενα `main.hs` αρχείο το οποίο να υλοποιεί ενα στοιχειώδες User Interface για τον interpreter.

Φροντίστε το πρόγραμμά σας να μεταγλωττίζεται με τον Glasgow Haskell Compiler (ghc) και δημιουργήστε και ενα Makefile.

Στην εργασία αυτή απαγορεύεται η χρήση external modules της Haskell.

Η εργασία μπορεί να υλοποιηθεί είτε ατομικά, είτε από ομάδες των δύο ατόμων.

Επιπλέον, περιγράψτε συνοπτικά τις όποιες σχεδιαστικές επιλογές σας σε ενα README.

Για οποιαδήποτε περαιτέρω διευκρίνηση, μπορείτε να απευθυνθείτε μέσω email σε κάποιον απο τους υπεύθυνους συνεργάτες:


Γιώργος Βασιλακόπουλος - sdi2000018 [at] di.uoa.gr
Μιχάλης Βιταντζάκης - sdi2000232 [at] di.uoa.gr
Δημήτρης Κωσταντινίδης - sdi1700065 [at] di.uoa.gr