1 Συναρτησιακός Προγραμματισμός

Ο συναρτησιακός προγραμματισμός έχει τις ρίζες του στον λάμβδα λογισμό (lambda calculus), ο οποίος αναπτύχθηκε στη δεκαετία του 1930 από τον Alonzo Church, ως μία προσπάθεια για να δοθεί μαθηματική υπόσταση στην έννοια του υπολογισμού.

Στο συναρτησιακό προγραμματισμό, το πρόγραμμα συνίσταται από ένα πλήθος συναρτήσεων, οι οποίες ορίζονται χρησιμοποιώντας σύνθεση και αναδρομή, ξεκινώντας από ένα σύνολο πρωταρχικών συναρτήσεων (π.χ. αριθμητικών πράξεων). Ο υπολογισμός του επιθυμητού αποτελέσματος επιτυγχάνεται μέσω της αποτίμησης μίας παράστασης, η οποία μπορεί να περιέχει προκαθορισμένους τελεστές καθώς και συναρτήσεις που έχουν οριστεί στο πρόγραμμα.

Αντίθετα με τις προστακτικές γλώσσες (C, Pascal, κλπ) οι συναρτήσεις στις καθαρά συναρτησιακές γλώσσες δεν προκαλούν παρενέργειες (αλλαγές σε τιμές μεταβλητών) και η τιμή που επιστρέφουν εξαρτάται αποκλειστικά από τις τιμές των ορισμάτων. Οι καθαρές συναρτησιακές γλώσσες δεν έχουν εντολή ανάθεσης, ούτε και εντολές επανάληψης. Η επανάληψη επιτυγχάνεται με χρήση αναδρομής.

Ορισμένα χαρακτηριστικά, η ύπαρξη των οποίων διευκολύνεται από τη φύση των συναρτησιακών γλωσσών είναι :

- αναδρομή
- πολυμορφισμός
- οκνηρή αποτίμηση
- συναρτήσεις υψηλής τάξης και συναρτήσεις ως τιμές
- λίστες ως προκαθορισμένοι τύποι
- σύνθετοι τύποι ως αποτελέσματα συναρτήσεων
- σταθερές σύνθετου τύπου

2 Η γλώσσα Haskell

Η Γλώσσα Haskell είναι μία καθαρή συναρτησιακή γλώσσα, η οποία εφαρμόζει οκνηρή αποτίμηση. Αναπτύχθηκε τη δεκαετία του 1990, με σκοπό να αποτελέσει μια βάση για την έρευνα και την εξέλιξη των γλωσσών αυτής της κατηγορίας. Ωφείλει το όνομά της στον αμερικάνο μαθηματικό και λογικό Haskell Curry.

Στη συνέχεια θα χρησιμοποιήσουμε τη γλώσσα Haskell για να κάνουμε μία εισαγωγή στις έννοιες του συναρτησιακού προγραμματισμού. Ο σκοπός αυτών των σημειώσεων δεν είναι σε καμμία περίπτωση το να περιγραφεί πλήρως η γλώσσα Haskell. Θα περιοριστούμε κυριώς σε παραδείγματα πάνω σε αριθμούς και λίστες, ώστε να γίνει κατανοητό το πώς η αναδρομή αρκεί για την υλοποίηση επανάληψης. Επίσης θα δούμε πως ορίζονται πολυμορφικές συναρτήσεις, συναρτήσεις που παίρνουν ως ορίσματα και επιστρέφουν ως αποτελέσμα άλλες συναρτήσεις και το πώς μπορούμε να σχηματίσουμε άπειρες λίστες οι οποίες μπορουν να συμμετέχουν σε πεπερασμένους υπολογισμούς. Αντίθετα δεν θα εξετάσουμε μεταξύ άλλων το πλήρες σύστημα τύπων τις Haskell και θέματα που αφορούν είσοδο και έξοδο για δημιουργία διαδραστικών εφαρμογών.

Στη συνέχεια θα κάνουμε την υπόθεση ότι όλες οι συναρτήσεις που θα ορίσουμε περιέχονται σε ένα μοναδικό αρχείο. Η Haskell δίνει τη δυνατότητα να υποδιαιρεθεί το πρόγραμμα σε ενότητες που περιέχονται σε ξεχωριστά αρχεία, ωστόσο αυτό το χαρακτηριστικό δεν θα βοηθούσε στην καλύτερη κατανόηση των χαρακτηριστικών του συναρτησιακού προγραμματισμού.

3 Ο διερμηνέας hugs της Haskell

Για να εκτελεστεί ο διερμηνέας hugs της Haskell σε σύστημα Unix γράφουμε στο τερματικό `hugs` και πατάμε `<ENTER>`. Στα windows μπορούμε να ανοίξουμε τον hugs από τη λίστα προγραμμάτων ή κάνοντας διπλό κλικ στο αντίστοιχο εικονίδιο, εφόσον υπάρχει.

Ο hugs εμφανίζει την prompt

```
Prelude> _
```

Το Prelude είναι το όνομα της βασικής βιβλιοθήκης της Haskell που περιέχει ένα πλήθος από προκαθορισμένες συναρτήσεις και τελεστές μεταξύ των οποίων οι βασικοί αριθμητικοί τελεστές.

Σε αυτή τη φάση μπορούμε να γράψουμε μία αριθμητική παράσταση (ή όποια άλλη παράσταση εμπλέκει μόνο προκαθορισμένες συναρτήσεις και τελεστές) και ο hugs θα την αποτιμήσει επιστρέφοντας μας το αποτέλεσμα (θα πρέπει μετά την ολοκλήρωση της παράστασης να πατήσουμε `<ENTER>`):

```
Prelude> 3+4
7
Prelude>
```

Μπορούμε να φορτώσουμε ένα αρχείο προγράμματος π.χ. το `example.hs`, το οποίο βρίσκεται στον κατάλογο από τον οποίο εκκινήσαμε τον hugs γράφοντας:

```
Prelude> :load example.hs
Main>
```

Η εντολή `:load` μπορεί πιο απλά να γραφτεί `:l`. Αν το αρχείο βρίσκεται σε διαφορετικό κατάλογο από αυτόν από τον οποίο εκτελέσαμε τον hugs, τότε θα πρέπει να γράψουμε το πλήρες μονοπάτι.

Παρατηρούμε ότι η prompt έχει αλλάξει σε `Main>`. Αυτό δηλώνει ότι το πρόγραμμα `example.hs` δεν είχε συντακτικά λάθη, και είναι πλέον φορτωμένο στη μνήμη. Έχοντας φορτώσει το πρόγραμμα μπορούμε να αποτιμήσουμε παραστάσεις που περιέχουν συναρτήσεις που ορίζονται σε αυτό, αλλά και συναρτήσεις από το Prelude, το οποίο παραμένει φορτωμένο στη μνήμη. Αν το πρόγραμμα δεν είναι συντακτικά ορθό (ή αν το αρχείο δεν υπάρχει), εμφανίζεται ένα μηνυμα λάθους και η prompt παραμένει `Prelude>`. Το `example.hs` παραμένει φορτωμένο στη μνήμη μέχρι να φορτώσουμε κάποιο άλλο αρχείο ή να γράψουμε την εντολή `:load` χωρίς να ακολουθείται από όνομα αρχείου.

Ο hugs είναι συνδεδεμένος με κάποιον editor τον οποίο μπορούμε να ανοίξουμε μέσα απο το περιβάλλον του hugs με την εντολή `:edit` ή πιο απλά `:e`. Αν η εντολή `:edit`. δεν ακολουθείται απο όνομα αρχείου τότε ανοίγει το τελευταίο αρχείο που προσπαθήσαμε να φορτώσουμε (επιτυχώς ή ανεπιτυχώς) με την εντολή `:load`.

Η εκτέλεση του `hugs` τερματίζεται με την εντολή `:quit` ή πιο απλά `:q`.

4 Βασικοί τύποι

Η Haskell διαθέτει ένα πλούσιο σύστημα τύπων. Στη συνέχεια θα περιγράψουμε μόνο ορισμένους βασικούς τύπους που θα χρησιμοποιήσουμε για να γίνουν κατανοητά τα κύρια χαρακτηριστικά του συναρτησιακού προγραμματισμού.

Ακέραιοι αριθμοί: η Haskell διαθέτει τον τύπο `Int` (ακέραιος με καθορισμένο μήκος) και τον τύπο `Integer` (ακέραιος με αυθαίρετα μεγάλο μήκος).

Για τους παραπάνω τύπους είναι προκαθορισμένοι μεταξύ άλλων οι τελεστές `+`, `-`, `*`, `^` (υψωση σε δύναμη) και οι συναρτήσεις `div` (ακέραια διαίρεση), `mod` (υπόλοιπο διαίρεσης), `max` (μέγιστο δύο στοιχείων), `min` (ελάχιστο δύο στοιχείων) `abs` (απόλυτη τιμή) και `negate` (αντίθετος).

```
> div 9 4
2
> mod 15 4
3
> abs (-4)
4
```

Παρατηρήσεις:

- Τα ορίσματα μίας συνάρτησης δίνονται μετά το όνομα της συνάρτησης, χωρισμένα με κενά. Ένας δυαδικός τελεστής γράφεται ανάμεσα στα ορίσματά του.
- Αν κάποιο όρισμα είναι αρνητικός αριθμος τότε πρέπει να γραφτεί μέσα σε παρένθεση.
- Μπορούμε να μετατρέψουμε μία συνάρτηση δύο μεταβλητών σε τελεστή γράφοντάς το ονομά της ανάμεσα σε δύο σύμβολα ‘ (τόνους) π.χ.

  ```
  > 15 ‘mod‘ 4
  3
  ```

- Αντίστροφα, μπορούμε να μετατρέψουμε έναν τελεστή σε συνάρτηση δύο μεταβλητών γράφοντάς τον μέσα σε παρενθέσεις, π.χ.

  ```
  > (+) 3 2
  5
  ```

Πραγματικοί αριθμοί: η Haskell διαθέτει τον τύπο `Float` (πραγματικός απλής ακρίβειας) και τον τύπο `Double` (πραγματικός διπλής ακρίβειας).

Για τους παραπάνω τύπους είναι προκαθορισμένοι μεταξύ άλλων οι τελεστές `+`, `-`, `*`, `/`, `^` (υψωση σε ακέραια δύναμη), `**` (υψωση σε πραγματική δύναμη) και οι συναρτήσεις `sqrt` (τετραγωνική ρίζα), `abs`, `negate`, `ceiling`, `floor`, `round`, `exp`, `log`, `sin`, `cos`, `tan` ...

Χαρακτήρες: Οι χαρακτήρες στη Haskell ανήκουν στον τύπο `Char`.

Οι συναρτήσεις `succ` και `pred` επιστρέφουν αντίστοιχα τον επόμενο και τον προηγούμενο χαρακτήρα με βάση την κωδικοποίηση χαρακτήρων ASCII. Η συνάρτηση `fromEnum` μετατρέπει έναν χαρακτήρα στον αντίστοιχο κωδικό ASCII, ενώ η `toEnum` κάνει την αντίστροφη μετατροπή (το `::Char` μετατρέπει το αποτέλεσμα της `toEnum`, η οποία μπορεί να επιστρέψει αποτέλεσμα διαφόρων τύπων, σε `Char`).

```
> succ 'a'
'b'
> pred 'a'
'`'
> fromEnum 'a'
97
> toEnum 100::Char
'd'
```

Αλφαριθμητικά: Τα αλφαριθμητικά στη Haskell ανήκουν στον τύπο `String` ο οποίος είναι ισοδύναμος με λίστα χαρακτήρων.

Συνεπώς όλες οι λειτουργίες που θα περιγράψουμε αργότερα για λίστες, λειτουργούν και για τον τύπο `String`.

Τιμές αλήθειας της λογικής: Ο τύπος `Bool` της Haskell έχει πεδίο τιμών {`True`, `False`}.

Οι τελεστές `&&`, `||`, `not`, υλοποιούν αντίστοιχα την σύζευξη (και), τη διάζευξη (ή) και την άρνηση (όχι).

Οι τελεστές σύγκρισης στη Haskell είναι οι `==`, `<`, `>`, `<=`, `>=` και `/=` (διάφορο), οι οποίοι δέχονται δυο ορίσματα του ίδιου διατεταγένου τύπου (π.χ. `Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool`) και επιστρέφουν αποτέλεσμα τύπου `Bool`:

```
> 2 /= 3
True
> (3+5) == (4+4)
True
> 'a' < 'A'
False
> "alice" <= "bob"
True
> True >= False
True
```

Σύνθετοι τύποι: Αν t_1 και t_2 είναι δύο τύποι της Haksell τότε ο (t_1, t_2) είναι επίσης τύπος με πεδίο τιμών όλα τα ζεύγη με πρώτη συνιστώσα τύπου t_1 και δεύτερη συνιστώσα τύπου t_2.

Για παράδειγμα, τα ζεύγη ακεραίων (π.χ το `(0,1)`) έχουν τύπο `(Int,Int)`, ενώ τα ζεύγη που αποτελούνται απο ένα αλφαρηθμητικό και μία λογική τιμή (π.χ το `("Chris",True)`) έχουν τύπο `(String,Bool)`.

Μπορούμε με ανάλογο τρόπο να έχουμε διάφορους τύπους τριάδων, τετράδων κλπ, σε καθέναν από τους οποίους κάθε συνιστώσα να ανήκει σε έναν συγκεκριμένο τύπο, π.χ. `(Int,Int,String,(Float,Float))`. Κάθε τύπος που ορίζεται με αυτόν τον τρόπο περιλαμβάνει πλειάδες με συγκεκριμένο μήκος.

Αν t είναι τύπος της Haksell τότε ο `[`t`]` είναι επίσης τύπος με πεδίο τιμών όλες τις λίστες με στοιχεία τύπου t.

Για παράδειγμα, οι λίστες ακεραίων έχουν τύπο `[Int]`, ενώ ο τύπος `String` είναι ισοδύναμος με `[Char]`.

Τα στοιχεία μίας λίστας δίνονται μέσα σε [και] χωρισμένα με κόμμα π.χ [1,2,3,4], ['a','b','c']. Η κενή λίστα (οποιουδήποτε τύπου) συμβολίζεται με [].

Οι πλειάδες και οι λίστες διαφέρουν σε δύο βασικά σημεία:

- Οι πλειάδες που ανήκουν στον ίδιο τύπο έχουν ίδιο πλήθος συνιστωσών, ενώ ένας τύπος λίστας περιέχει λίστες με πλήθος στοιχείων που μπορεί να είναι οποιοσδήποτε φυσικός αριθμός.
- Τα στοιχεία μίας λίστας είναι όλα του ίδιου τύπου, ενώ οι συνιστώσες μιας πλειάδας μπορεί να ανήκουν σε διαφορετικούς τύπους.

5 Σταθερές

Στη Haskell μπορούμε να αναθέσουμε τιμές σε συμβολικά ονόματα. Η τιμή που ανατίθεται σε ένα όνομα δεν μπορεί αλλάξει με νέα ανάθεση. Συνεπώς τα ονόματα αυτά αντιστοιχούν σε σταθερές.

```
zero :: Int
zero = 0

pi :: Float
pi = 3.14159

space :: Char
space = ' '
```

6 Συναρτήσεις

Ο ορισμός μίας συνάρτησης αποτελείται από μία δήλωση που καθορίζει τον τύπο της συνάρτησης (τύποι ορισμάτων - αποτελέσματος) και μία σειρά από ισότητες που καθορίζουν την τιμή της.

Παράδειγμα 1: Η συνάρτηση `sqrInt` υπολογίζει το τετράγωνο ενός ακεραίου:

```
sqrInt :: Int -> Int
sqrInt n = n * n
```

Η πρώτη γραμμή δηλώνει ότι η `sqrInt` είναι μία συνάρτηση από ακέραιους σε ακέραιους, δηλαδή παίρνει ως είσοδο έναν ακέραιο και επιστρέφει μία ακέραια τιμή. Η δεύτερη γραμμή καθορίζει ότι η τιμή της συνάρτησης για είσοδο `n` είναι το `n*n`.

Για να αποτιμήσουμε την τιμή μίας συνάρτησης, γράφουμε το όνομα της ακολουθούμενο από τις πραγματικές παραμέτρους:

```
> sqrInt 5
25
```

∎

Παράδειγμα 2: η συνάρτηση `avgInt` επιστρέφει το 'μέσο όρο' δύο ακεραίων αριθμών:

```
avgInt :: Int -> Int -> Int
avgInt a b = (a + b) `div` 2
```

∎

Οι συναρτήσεις που έχουν οριστεί μπορούν να χρησιμοποιηθούν για να ορίσουμε νέες συναρτήσεις:

Παράδειγμα 3: η συνάρτηση `avgSqrInt` επιστρέφει το 'μέσο όρο' τών τετραγώνων δύο ακεραίων αριθμών:

```
avgSqrInt :: Int -> Int -> Int
avgSqrInt a b = avgInt (sqrInt a) (sqrInt b)
```

∎

7 Συντακτικοί κανόνες

Τα ονόματα στη Haksell σχηματίζονται χρησιμοποιώντας χαρακτήρες Unicode που αντιστοιχούν σε κεφαλαία ή μικρά γράμματα, ψηφία και τους χαρακτήρες _ (χαρακτήρας υπογράμισης) και ' (απλό εισαγωγικό).

Τα ονόματα των σταθερών, των συναρτήσεων και των τυπικών παραμέτρων τους, όπως επίσης των μεταβλητών τύπων που θα δούμε αργότερα πρέπει να αρχίζουν με μικρό γράμμα ή το χαρακτήρα υπογράμισης _.

Τα ονόματα των τύπων, και κατασκευαστών τύπων (όπως είναι τα `True` και `False`) πρέπει να αρχίζουν με κεφαλαίο γράμμα.

Ενα τμήμα ορισμού (δήλωση ή ισότητα) μπορεί να εκτείνεται σε περισσότερες από μία γραμμές. Οι πρόσθετες γραμμές (αν υπάρχουν) πρέπει να ξεκινούν δεξιότερα από την πρώτη γραμμή του τμήματος ορισμού.

Αν συναντηθεί γραμμή που ξεκινάει στην ίδια στήλη ή αριστερότερα από την πρώτη γραμμή ενός τμήματος ορισμού, η Haskell θεωρεί ότι η γραμμή αυτή ανήκει σε νέο τμήμα ορισμού.

Η Haskell κατά τη συντακτική ανάλυση εισάγει αυτόματα το χαρακτήρα ; ανάμεσα σε διαδοχικά τμήματα ορισμών. Επίσης οριοθετεί με τους χαρακτήρες { και } ορισμένα τμήματα του προγράμματος (όπως για παράδειγμα τους τοπικούς ορισμούς που ακολουθούν τη λέξη `where`, τους οποίους θα περιγράψουμε παρακάτω).

Για αυτό το λόγο, αν δεν τηρούμε τους κανόνες στοιχισης η Haskell ενδέχεται να μας επιστρέψει μηνύματα λάθους της μορφής `unexpected ;` ή `unexpected }`, παρότι οι συγκεκριμένοι χαρακτήρες δεν εμφανίζονται στο πρόγραμμα.

Στη Haskell μπορούμε να γράψουμε σχόλια περικλείοντάς τα στα σύμβολα {- και -}. Εναλλακτικά τα σχόλια μπορούν να ξεκινήσουν με τα σύμβολα --. Σε αυτή την περίπτωση εκτείνονται μέχρι το τέλος της γραμμής.

8 Παραστάσεις υπό συνθήκη

Στη Haskell υπάρχουν διάφοροι τρόποι ώστε να ορίσουμε συναρτήσεις που οι τιμές τους εξαρτώνται από κάποια συνθήκη:

- `if - then - else`: αν αληθεύει συνθήκη που ακολουθεί το `if` λαμβάνεται η τιμή που ακολουθεί το `then`, αλλιώς λαμβάνεται η τιμή που ακολουθεί το `else`.

  Παράδειγμα 4: Η παρακάτω συνάρτηση υπολογίζει το ελάχιστο δύο ακεραίων:

  ```
  minInt :: Int -> Int -> Int
  minInt m n = if m<n then m else n
  ```

  ∎

- συνθήκες φρουροί: η τιμή της συνάρτησης καθορίζεται από μία ακολουθία περιπτώσεων της μορφής: | <συνθήκη> = <τιμή>

  Οι συνθήκες εξετάζονται μία προς μία μέχρι να βρεθεί κάποια που αληθεύει, οπότε και λαμβάνεται η αντίστοιχη τιμή.

  Παράδειγμα 5: Η παρακάτω συνάρτηση υπολογίζει την απόλυτη τιμή ενός ακέραιου:

  ```
  absInt :: Int -> Int
  absInt n
    | n > 0 = n
    | otherwise = negate n
  ```

Το `otherwise` αποτιμάται πάντα σε `True`.

∎

- πρότυπα: δίνεται μια σειρά ισοτήτων, που ορίζουν την τιμή της συνάρτησης, για διάφορα πρότυπα των τιμών των παραμέτρων. Τα πρότυπα εξετάζονται με τη σειρά μέχρι κάποιο να ταιριάζει με τις τιμές των πραγματικών παραμέτρων:

  Παράδειγμα 6: Η παρακάτω συνάρτηση επιστρέφει `True` αν και μόνο αν το όρισμά της είναι μηδέν:

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

∎

- συνδυασμός των παραπάνω:

  Παράδειγμα 7: Η παρακάτω συνάρτηση επιστρέφει 1,0 ή -1 αν το όρισμά της είναι αντίστοιχα θετικό, μηδέν ή αρνητικό:

```
sign :: Int -> Int
sign 0 = 0
sign n
    | n>0       = 1
    | otherwise = -1
```

∎

9 Τοπικοί ορισμοί

Η Haskell παρέχει τη δυνατότητα να ορίσουμε τοπικά, εσωτερικά σε μία συνάρτηση, συμβολικά ονόματα ή ακόμη και ολόκληρες συναρτήσεις. Αυτό βοηθάει στην αναγνωσιμότητα του προγράμματος και σε πολλές περιπτώσεις κάνει την εκτέλεση του πιο αποδοτική. Οι τοπικές δηλώσεις ξεκινούν με τη λέξη `where`.

Παράδειγμα 8: Έστω η συνάρτηση

$$r(x, y, z) = (x + y + z)^x + (x + y + z)^y + (x + y + z)^z.$$

Η παρακάτω υλοποίηση, υπολογίζει τό άθροισμα $x + y + z$. τρείς φορές, μία για κάθε εμφάνιση του στον τύπο υπολογισμού.

```
r' :: Int -> Int -> Int -> Int
r' x y z = (x+y+z)^x + (x+y+z)^y + (x+y+z)^z
```

Μία καλύτερη υλοποίηση προκύπτει αν ορίσουμε τοπικά το `w` να έχει την τιμή `x+y+z`. Με αυτόν τον τρόπο το άθροισμα υπολογίζεται μία μόνο φορά. Επίσης απολουστεύεται η παράσταση που δίνει το επιστρεφόμενο αποτέλεσμα.

```
r :: Int -> Int -> Int -> Int
r x y z = w^x + w^y + w^z
      where w = x + y + z
```

∎

10 Οκνηρή Αποτίμηση

Το επιθυμητό αποτέλεσμα στη Haskell (όπως και σε όλες τις καθαρά συναρτησιακές γλώσσες) επιτυγχάνεται μέσω της αποτίμησης μίας παράστασης. Στη συνέχεια θα δούμε πώς γίνεται η αποτίμηση μίας παράστασης στη Haskell, η οποία στηρίζεται στη στρατηγική της οκνηρής αποτίμησης (lazy evaluation).

Είναι χρήσιμο αρχικά να δούμε πώς μπορούμε να κατασκευάσουμε ένα δέντρο που να περιγράφει τη δομή μιας παράστασης.

Πρώτα θα πρέπει να αρθούν όλες οι αμφισημίες, ώστε να είναι μονοσήμαντα ορισμένη η δομή της παράστασης, δηλαδή να είναι ξεκάθαρο ποιες υποπαραστάσεις αποτελούν τα ορίσματα του κάθε τελεστή. Αυτό γίνεται με βάση του κανόνες προτεραιότητας και προσεταιρισμού, οι κυριότεροι από τους οποίους συνοψίζονται παρακάτω:

- Οι τελεστές με ένα όρισμα έχουν μεγαλύτερη προτεραιότητα από τους δυαδικούς τελεστές.
- Οι τελεστές `*`, `` `div` ``, `` `mod` `` και `/` έχουν μικρότερη προτεραιότητα από τους τελεστές `^` και `**` και μεγαλύτερη προτεραιότητα από τους τελεστές `+` και `-`.
- Οι τελεστές `+`, `-`, `*`, `` `div` ``, `` `mod` `` και `/` προσεταιρίζονται από αριστερά προς τα δεξιά.
- Οι τελεστές `^` και `**` προσεταιρίζονται από δεξιά προς τα αριστερά.
- Οι τελεστές σύγκρισης έχουν μικρότερη προτεραιότητα άπό τους δυαδικούς αριθμητικούς τελεστές και μεγαλύτεροι προτεραιότητα από τους δυαδικούς λογικούς τελεστές.
- Ο τελεστής `&&` έχει μεγαλύτερη προτεραιότητα από τον `||`
- Οι τελεστές `&&` και `||` προσεταιρίζονται από τα αριστερά προς τα δεξιά.
- Εφαρμογή συνάρτησης έχει μεγαλύτερη προτεραιότητα από όλους τους τελεστές και προσεταιρίζεται από αριστερά πρός τα δεξιά.

Με δεδομένη μία παράσταση της Haskell μπορούμε να προσθέσουμε κατάλληλα παρενθέσεις έτσι ώστε τα ορίσματα κάθε τελεστή η συνάρτησης να είναι είτε απλές παραστάσεις (σταθερές ή μεταβλητές) είτε σύνθετες παραστάσεις κλεισμένες σε παρενθέσεις.

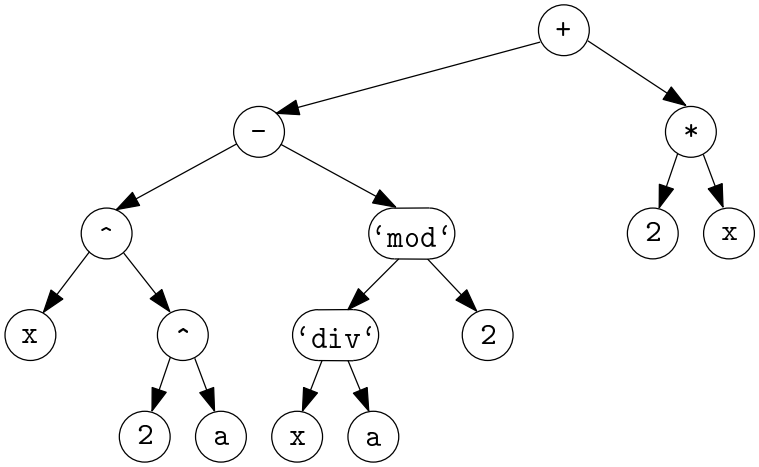
Η παραπάνω μετατροπή μας βοηθάει στο να περιγράψουμε την παράσταση με ένα δέντρο:

- Μία απλή παράσταση (συμβολική ή κυριολεκτική σταθερά ή μεταβλητή) παριστάνεται από ένα δέντρο αποτελούμενο από έναν κόμβο με ετικέτα τη σταθερά ή τη μεταβλητή.
- Μία σύνθετη παράσταση παριστάνεται από ένα δέντρο η ρίζα του οποίου έχει ετικέτα το όνομα της συνάρτησης ή του τελεστή που θα δώσει το τελικό αποτέλεσμα, ενώ τα παιδιά του από αριστερά προς τα δεξιά είναι οι ρίζες των δέντρων που αντιστοιχούν στις παραστάσεις που αποτελούν τα ορίσματα της συνάρτησης ή του τελεστή.

Παράδειγμα 9: Έστω η παράσταση `x^2^a-x‘div‘a‘mod‘2+2*x`

Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `((x^(2^a))-((x‘div‘a)‘mod‘2))+(2*x)`

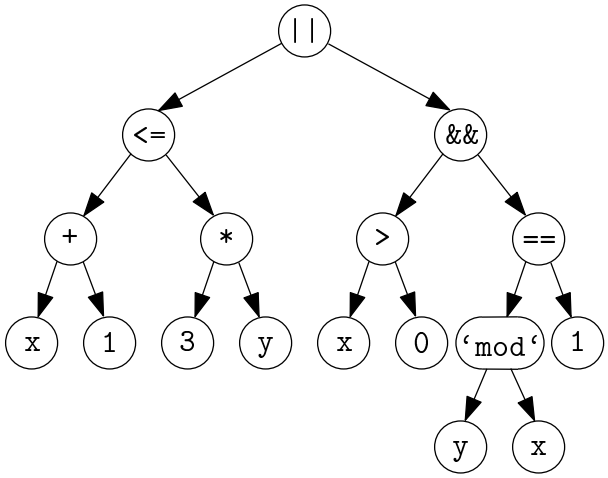
Το δέντρο που αντιστοιχεί στην παραπάνω παράσταση είναι:



Παράδειγμα 10: Έστω η παράσταση `x+1<=3*y||x>0&&y‘mod‘x==1`

Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `((x+1)<=(3*y))||((x>0)&&((y‘mod‘x)==1))`

Το δέντρο που αντιστοιχεί στην παραπάνω παράσταση είναι:



Σύμφωνα με την τη στρατηγική της οκνηρής αποτίμησης τα ορίσματα μίας συνάρτησης (τα οποία ενδέχεται να είναι σύνθετες παραστάσεις) δεν αποτιμούνται παρά μόνο αν και όταν χρειαστούν για τον υπολογισμό του τελικού αποτελέσματος.

Αν η τιμή κάποιου ορίσματος δεν επηρεάζει το αποτέλεσμα της συνάρτησης τότε το όρισμα αυτό δεν αποτιμάται. Επίσης αν κάποιο όρισμα έχει σύνθετο τύπο και αρκεί ένα μόνο τμήμα του για να γίνει ο υπολογισμός, τότε υπολογίζεται μόνο αυτό το τμήμα.

Τα πλεονεκτήματα της οκνηρή αποτίμηση είναι αφ'ενός ότι αποφεύγονται άσκοποι υπολογισμοι και αφ'ετέρου ότι δίνεται η δυνατότητα να επεξεργαζόμαστε άπειρες δομές.

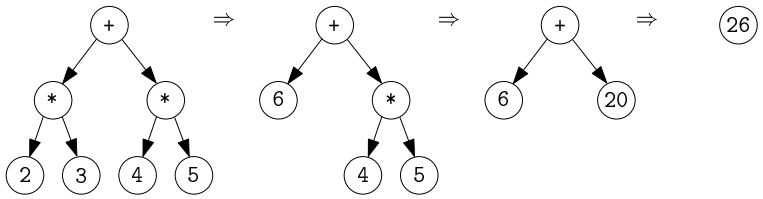
Αντίθετα με ότι συμβαίνει στις περισσότερες επιτακτικές γλώσσες, η αποτίμηση μίας παράστασης γίνεται από έξω προς τα μέσα: πρώτα εξετάζεται ο ορισμός της συνάρτησης (ή του τελεστή) που θα δώσει το τελικό αποτέλεσμα. Η συνάρτηση αυτή βρίσκεται στη ρίζα του δέντρου. Τα ορίσματα ενός τελεστή αποτιμούνται από αριστερά προς τα δεξιά.

Παράδειγμα 11: Έστω η παράσταση `2*3+4*5`. Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `(2*3)+(4*5)`. Η αποτίμηση της παράστασης δίνει διαδοχικά:

```
  (2*3)+(4*5)
= 6+(4*5)
= 6+20
= 26
```

Στο παράδειγμα αυτό η οκνηρή αποτίμηση δεν δημιουργεί καμία διαφορά, καθώς ο τελεστής + απαιτεί το υπολογισμό και των δύο ορισμάτων του.

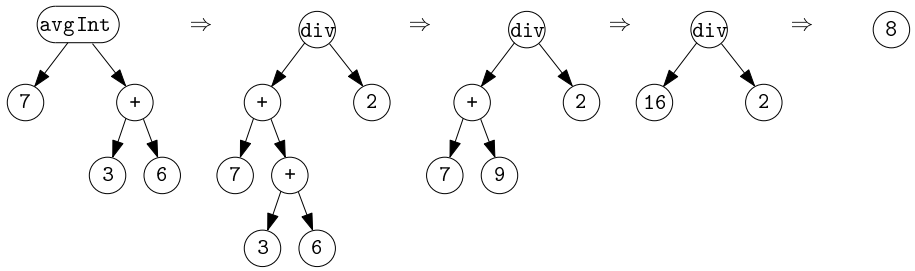
Σε μορφή δέντρων ο παραπάνω υπολογισμός παριστάνεται:



Παράδειγμα 12: Έστω η παράσταση `avgInt (3+5) 6`. Η αποτίμηση της δίνει διαδοχικά:

```
  avgInt 7 (3+6)
= (7 + (3+6)) `div` 2
= (7+9) `div` 2
= 16 `div` 2
= 8
```

Σε μορφή δέντρων:



Παρατηρούμε ότι λόγω της οκνηρής αποτίμησης, η επεξεργασία της παράστασης ξεκίνησε από τη ρίζα του δέντρου και το όρισμα `3+5` αποτιμήθηκε αφού πρώτα έγινε η αντικατάσταση με βάση τον ορισμό της `avgInt`, από την οποία προέκυψε η αναγκαιότητα της αποτίμησή του ώστε να ολοκληρωθεί ο υπολογισμός. ∎

Αν μία τυπική παράμετρος εμφανίζεται σε περισσότερα από ένα σημεία της παράστασης που ορίζει την τιμή μιας συνάρτησης, τότε το αντίστοιχο όρισμα (πραγματική παράμετρος) αποτιμάται μία μόνο φορά.

Παράδειγμα 13: Έστω η παρακάτω συνάρτηση `f`

```
f :: Int -> Int
f n = 2^n + n
```

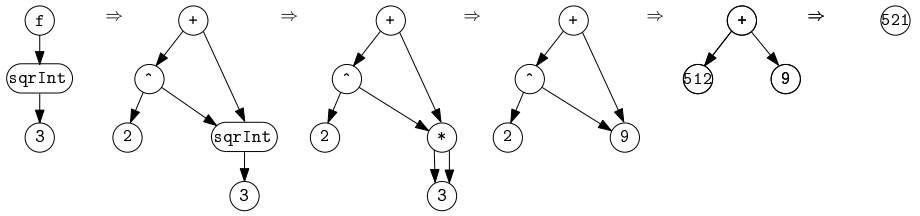
Η αποτίμηση της παράστασης `f (sqrInt 3)` δίνει διαδοχικά:

```
  f (sqrInt 3)
= (2^(sqrInt 3))+(sqrInt 3)
= (2^(3*3))+(3*3)
= (2^9)+9
= 512+9
= 521
```

Το `sqrInt 3` αποτιμάται μία μόνο φορά, παρότι εμφανίζεται σε δύο σημεία της ενδιάμεσης παράστασης.

Αυτό μπορεί να γίνει επειδή στην παραγματικότητα, οι ενδιάμεσες παραστάσεις δεν αντιστοιχούν σε δέντρα αλλά σε κατευθυνόμενα (πολυ)γραφήματα χωρίς κύκλους. Αυτό πρακτικά σημαίνει ότι ένας κόμβος μπορεί να είναι ταυτόχρονα παιδί δύο άλλων κόμβων.

Σε μορφή γραφημάτων ο παραπάνω υπολογισμός γράφεται:



Παράδειγμα 14: Έστω η παρακάτω συνάρτηση g

```
g :: Int -> Int -> Bool
g m n = n > 0 && m `mod` n == n-1
```

Η αποτίμηση της παράστασης `g 5 0` δίνει διαδοχικά:

```
  g 5 0
= (0 > 0) && ((5 `mod` 0) == (0-1))
= False && ((5 `mod` 0) == (0-1))
= False
```

Το πρώτο όρισμα του τελεστή `&&` αποτιμάται σε `False`, συνεπώς η τιμή της παράστασης είναι σίγουρα `False` ανεξάρτητα από την τιμή του δεύτερου ορίσματος και η Haskell δεν αποτιμά καθόλου το όρισμα αυτό. Σημειώνεται ότι αν η Haskell αποτιμούσε το δεύτερο όρισμα, τότε θα επέστρεφε μήνυμα λάθους λόγω διαίρεσης με το 0. Η μερική αποτίμηση των λογικών τελεστών αποτελεί ειδική περίπτωση της οκνηρής αποτίμησης και χρησιμοποιείται ακόμη και από γλώσσες που δεν υποστηρίζουν γενικά οκνηρή αποτίμηση (όπως η C). ∎

Αν ο ορισμός μίας συνάρτησης περιέχει πρότυπα, τότε τα ορίσματα υπολογίζονται μερικώς, στο βαθμό που απαιτείται ώστε να διαπιστωθεί αν ταιριάζουν

με το πρότυπο. Σε περίπτωση αποτυχίας εξετάζεται το επόμενο πρότυπο, το οποίο μπορεί να απαιτήσει αποτίμηση επιπλέον ορισμάτων.

Παράδειγμα 15: Έστω η παρακάτω συνάρτηση h

```
h :: Int -> Int -> Int
h m 0 = m
h 0 n = n
h m n = m*n
```

Έστω η παράσταση `h (3+4) (8 ‘div‘ 3)`

Η Haskell πρώτα θα εξετάσει αν μπορεί να εφαρμοστεί η πρώτη ισότητα. Για να το κάνει αυτό πρέπει να διαπιστώσει αν το `8 ‘div‘ 4` ταιριάζει με το πρότυπο 0, συνεπώς πρέπει να το αποτιμήσει. Η τιμή που προκύπτει είναι 2, οπότε η Haskell συνεχίζει εξετάζοντας αν η επόμενη ισότητα στον ορισμό της h μπορεί να εφαρμοστεί. Για να το κάνει αυτό πρέπει να διαπιστώσει αν το (3+4) ταιριάζει με το πρότυπο 0, συνεπώς πρέπει να το αποτιμήσει. Η τιμή που προκύπτει είναι 7, οπότε η Haskell συνεχίζει εξετάζοντας αν την τελευταία ισότητα στον ορισμό της h, η οποία μπορεί να εφαρμοστεί (αφού περιέχει μόνο μεταβλητές), επιστρέφοντας ως αποτέλεσμα το 14.

Συνοπτικά ο υπολογισμός φαίνεται παρακάτω. Το σύμβολο # δηλώνει την αρχή σύγκρισης με πρότυπο, η οποία απαιτείται για να συνεχιστεί υπολογισμός. Το arg `<<<` $pattern$ δηλώνει ότι εξετάζουμε αν το arg ταιριάζει με το πρότυπο $pattern$, δηλαδή ότι το πρότυπο είναι αρκετά γενικό ώστε να εμπεριέχει την τιμή του ορίσματος. Το αποτέλεσμα της διερεύνησης μπορεί να είναι `"yes"` ή `"no"`. (το `<<<` δεν αποτελεί τελεστή της Haskell παρα μόνο μία συντομογραφία που χρησιμοποιούμε εδώ για να περιγράψουμε τον υπολογισμό. Το ίδιο ισχύει και για τα `"yes"`,`"no"`,`<=>` και `#`). Σύγκρίσεις με πρότυπα που είναι μεταβλητές και συνεπώς επιτυγχάνουν αμέσως παραλείπονται.

```
  h (3+4) (8 ‘div‘ 3)
    #     8 ‘div‘ 3 <<< 0
     <=>  2 <<< 0
     <=>  "no"
    #     3+4 <<< 0
     <=>  7 <<< 0
     <=>  "no"
= 7 * 2
= 14
```

Στον παραπάνω υπολογισμό, αποτιμήθηκαν και τα δύο ορίσματα της h, ωστόσο πρώτα αποτιμήθηκε το δεύτερο και μετά το πρώτο. Αυτό ωφείλεται στη σειρά των ισοτήτων που ορίζουν την h και τα πρότυπα που αυτές περιέχουν. ∎

Αν ο ορισμός μίας συνάρτησης περιέχει συνθήκες φρουρούς τότε οι συνθήκες αποτιμούνται με τη σειρά, μέχρι κάποια να έχει τιμή `True`. Η τιμή της

συνάρτησης επιστρέφεται από την αντίστοιχη ισότητα.

Παράδειγμα 16: Έστω η συνάρτηση:

```
s :: Int -> Int -> Int -> Int
s a b c
       | a < 0 = b
       | a > 0 = c
       | otherwise = 0
```

Για να αποτιμηθεί η παράσταση `s (2*3) (5*6) (9*4)` πρώτα αποτιμάται η συνθήκη φρουρός `a<0`. Αυτό απαιτεί αποτίμηση του ορίσματος `(2*3)`. Επειδή η συνθήκη δεν ικανοποιείται εξετάζεται η επόμενη συνθήκη φρουρός `a>0`, η οποία αληθεύει, συνεπώς η τιμή της συνάρτηση είναι η τιμή του τρίτου ορίσματός της.

Συνοπτικά ο υπολογισμός φαίνεται παρακάτω. Το σύμβολο ? δηλώνει την αρχή της αποτίμησης συνθήκης φρουρού.

```
  s (2*3) (5*6) (9*4)
      ?   (2*3) < 0
        = 6 < 0
        = False
      ?   6 > 0
        = True
= 9*4
= 36
```

Παρατηρούμε ότι στον παραπάνω υπολογισμό το όρισμα `(5*6)` δεν αποτιμάται γιατί δεν χρειάζεται στον υπολογισμό του αποτελέσματος. Γενικότερα στην παραπάνω συνάρτηση αποτιμάται το πολύ μία από τις παραμέτρους `b,c`. ∎

Αν στον ορισμό μίας συνάρτησης υπάρχουν τιμές που ορίζονται τοπικά με το `where`, τότε αυτές υπολογίζονται μόνο μία φορά αν και όταν χρειαστεί (όπως ακριβώς συμβαίνει και με τις παραμέτρους της συνάρτησης).

Παράδειγμα 17: Έστω η συνάρτηση:

```
r :: Int -> Int -> Int -> Int
r x y z = w^x + w^y + w^z
      where w = x + y + z
```

Η αποτίμηση του `r 3 2 5` γίνεται με τον παρακάτω τρόπο:

```
  r 3 2 5
= ((w^3) + (w^2)) + (w^5) where w = (3 + 2) + 5
    {  (3 + 2) + 5
     = 5 + 5
     = 10
    }
= ((10^3) + (10^2)) + (10^5)
= (100 + (10^2)) + (10^5)
= (1000 + 100) + (10^5)
= 1100 + (10^5)
= 1100 + 100000
= 101100
```

Σε μορφή γραφημάτων ο παραπάνω υπολογισμός γράφεται: