Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Σχεδίαση και Υλοποίηση της Γλώσσας Προγραμματισμού lambda-cases

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΗΜΗΤΡΙΟΣ ΣΑΡΙΔΑΚΗΣ ΜΠΙΤΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Σχεδίαση και Υλοποίηση της Γλώσσας Προγραμματισμού lambda-cases

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΗΜΗΤΡΙΟΣ ΣΑΡΙΔΑΚΗΣ ΜΠΙΤΟΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Ιουλίου 2024.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

Δημήτριος Σαριδάκης Μπίτος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της παρούσας εργασίας είναι η σχεδίαση και υλοποίηση μιας συναρτησιακής γλώσσας προγραμματισμού, βασισμένης στη Haskell. Σε αυτή τη γλώσσα προσπάθησα να απομονώσω το μέρος της γλώσσας Haskell που χρησιμοποιώ πιο συχνά, αλλάζοντας παράλληλα τη σύνταξη κάποιων συστατικών ώστε να είναι πιο ευανάγνωστα. Τα παραπάνω υλοποιήθηκαν σε έναν μεταγλωττιστή από την νέα γλώσσα στην Haskell.

Λέξεις κλειδιά

Γλώσσα προγραμματισμού Haskell, Συναρτησιακός προγραμματισμός, Γλώσσες προγραμματισμού, Συστήματα τύπων, Μεταγλωττιστές, Γραμματικές, Συντακτική ανάλυση, Συντακτικό δέντρο, Εκφράσεις, Τελεστές.

Abstract

The purpose of this diploma dissertation is the design and implementation of a functional programming language, based on Haskell. In this language, I tried to isolate the part of Haskell that I use most often and to change the syntax of some language components, in order to make them more readable. I have implemented the above by means of a translator from the new language to Haskell.

Key words

Haskell programming language, Functional programming, Programming languages, Type systems, Compilers, Grammars, Syntax analysis, Syntax tree, Expressions, Operators.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νίκο Παπασπύρου, για τη συνεχή καθοδήγηση και εμπιστοσύνη του. Ευχαριστώ επίσης τη μητέρα μου, η οποία με υποστήριξε και έκανε δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Δημήτριος Σαριδάκης Μπίτος,

Αθήνα, 8η Ιουλίου 2024

Περιεχόμενα

Κατάλογος πινάκων

Πίνακες στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

Η Haskell είναι μια απολαυστική γλώσσα προγραμματισμού [1].

Ωστόσο, δεν φαίνεται να κατέχει τη θέση που της αξίζει ως προς το πόσο δημοφιλής είναι μεταξύ των προγραμματιστών. Γιατί συμβαίνει αυτό; Είναι εκ φύσεως δύσκολη στη μάθηση ή μήπως η σύνταξή της είναι περίπλοκη για τον αρχάριο; Πιστεύω ότι με κάποιες αλλαγές στη σύνταξη που θα την καταστήσουν πιο οικεία προς τον νέο χρήστη, δεν θα υπήρχε πιο ελκυστική γλώσσα από τη (νέα) Haskell. Σε αυτή την κατεύθυνση, παρουσιάζω κάποιες (ελπίζω χρήσιμες) νέες συντακτικές δομές, από τις οποίες μερικές είναι πιο κοντά στο προστακτικό/αντικειμενοστρεφή στυλ προγραμματισμού (για να προσελκύσουν περισσότερους ήδη έμπειρους προγραμματιστές από αυτές τις γλώσσες), μερικές είναι πιο κοντά στα μαθηματικά (στα οποία θα πρέπει να είναι έμπειροι οι περισσότεροι προγραμματιστές) και μερικές είναι πιο κοντά στη φυσική γλώσσα (στην οποία είμαστε όλοι ήδη έμπειροι).

Η Haskell είναι μακράν η αγαπημένη μου γλώσσα προγραμματισμού και ήταν έτσι από την πρώτη στιγμή που άρχισα να την καταλαβαίνω. Θυμάμαι να τη μαθαίνω στο μάθημα των προχωρημένων γλωσσών προγραμματισμού που δίδασκε ο επιβλέπων της διατριβής μου, ο κύριος Παπασπύρου. Καθώς άρχισα να καταλαβαίνω πώς συνδέονται όλες οι δομές μεταξύ τους με αυτόν τον όμορφα περιεκτικό τρόπο, άρχισα επίσης να αναρωτιέμαι, γιατί να χρησιμοποιήσω ποτέ οποιαδήποτε από αυτές τις άλλες γλώσσες που έχω μάθει; Γιατί να χάσω τον χρόνο μου γράφοντας κώδικα σε γλώσσες που είναι τόσο δύσκολες στην ανάγνωση ή/και τόσο δύσκολες στον εντοπισμό σφαλμάτων; Αυτό το συναίσθημα του να αφήνω τον μεταγλωττιστή να με καθοδηγεί σε όλα τα λάθη μου και μετά... τέλος! Απλά το πρόγραμμα λειτουργεί! (τις περισσότερες φορές) Πόσο εξαιρετικό! Ξαφνικά, ο κώδικας είναι ένας ευχάριστος γρίφος τύπων, που πρέπει απλώς να συνδυαστούν με τον σωστό τρόπο. Επιπλέον, υπάρχει και ένας πολύ χρήσιμος οδηγός!

Από την άλλη πλευρά, πριν φτάσω σε αυτό το σημείο κατανόησης, βρισκόμουν σε μεγάλη σύγχυση. Τι σημαίνει αγνός; Τι είναι αυτές οι μονάδες που φαίνεται να είναι τόσο σημαντικές; Γιατί αυτή η εντολή ”do” δεν με αφήνει να κάνω αυτό που θέλω να κάνω; Τι είναι αυτά τα πράγματα που ξεκινούν με κεφαλαίο γράμμα και μοιάζουν με συναρτήσεις; Γιατί υπάρχουν τόσα πολλά βέλη στους τύπους; Καθώς άρχισα να καταλαβαίνω σιγά σιγά όλα αυτά και χάρηκα πολύ με αυτό το γεγονός, είδα φίλους να εγκαταλείπουν το μάθημα. ”Δεν καταλαβαίνω και δεν πρόκειται ποτέ να καταλάβω”, μου είπαν, ”γιατί χρειαζόμαστε όλα αυτά τα πράγματα ούτως ή άλλως;”. Δεν μπορούσα να τους πείσω να μην εγκαταλείψουν το μάθημα. Είχα μόλις αρχίσει να καταλαβαίνω ο ίδιος και δεν ήμουν σε θέση να το εξηγήσω με σαφήνεια σε άλλους, ήταν περισσότερο μια αίσθηση σύνδεσης κομματιών παζλ και θαυμασμού για το πόσο εύκολο ήταν να γράψω κώδικα.

Καθώς πειθόμουν όλο και περισσότερο ότι θέλω να αποφύγω να γράφω κώδικα σε άλλες γλώσσες, αναρωτήθηκα επίσης, γιατί αυτός δεν είναι ο κυρίαρχος τρόπος γραφής κώδικα; Γιατί η Haskell δεν βρίσκεται ούτε στην πρώτη πεντάδα των πιο δημοφιλών γλωσσών, ενώ για μένα είναι ξεκάθαρα στο νούμερο ένα; Ως μηχανικός και επιστήμονας, πρέπει άλλωστε να εξετάσω τα δεδομένα και να δώσω την καλύτερη δυνατή εξήγηση. Φυσικά, δεν έχω ακόμα ιδέα, αλλά θα προσπαθήσω να μαντέψω με βάση τις εμπειρίες μου. Πρώτον, τα εργαλεία ανάπτυξης ίσως να αποτελούν πρόβλημα και σίγουρα πολλοί άνθρωποι (συμπεριλαμβανομένου και εμού) έχουν αντιμετωπίσει προβλήματα εξαιτίας τους. Αυτό όμως είναι ένα δύσκολο πρόβλημα και ξεπερνάει το σκοπό αυτής της εργασίας. Αντίθετα, θέλω να επικεντρωθώ σε μια άλλη άποψη στην οποία σιγά σιγά κατέληξα, η οποία είναι ότι η σημασιολογία της γλώσσας είναι πλήρης και θα παραμείνει για πάντα η ανώτερη σημασιολογία. Αντιθέτως, δεν

πιστεύω ότι μπορεί να ειπωθεί το ίδιο για τη σύνταξη.

Όσο αγαπάμε όλοι τις απαρχές της Haskell, τον λ-λογισμό, πιστεύω ότι η σύνταξη του λ-λογισμού έχει εξυπηρετήσει και εξυπηρετεί τον σκοπό της στον θεωρητικό τομέα και δεν πρέπει να επιβάλλεται στον μέσο μηχανικό λογισμικού, που απλώς θέλει να λύσει ένα πρόβλημα. Η εφαρμογή των συναρτήσεων στα μαθηματικά εδώ και αιώνες έχει τη σαφή σύνταξη όπου οι μεταβλητές βρίσκονται σε παρένθεση και ως εκ τούτου διαχωρίζονται οπτικά με σαφήνεια από τη συνάρτηση αυτή. Δεν είναι απαραίτητο να χρησιμοποιούμε τη σύνταξη του λ-λογισμού για να σκεφτόμαστε με τη σημασιολογία του λ-λογισμού.

Επιπλέον, παρόλο που είναι εκπληκτικό το γεγονός ότι μπορούμε να κάνουμε συμπερασμό τύπων, στην πράξη, εάν δεν χρησιμοποιηθούν επισημειώσεις τύπων, η Haskell χάνει την ομορφιά της και γίνεται πολύ δύσκολη στην ανάγνωση, όπως και όλες οι άλλες γλώσσες (και επίσης αρχίζει να έχει απαίσια μηνύματα σφάλματος). Έτσι, ίσως θα πρέπει να αναγκάσουμε τον χρήστη να χρησιμοποιεί επισημειώσεις τύπων (εκτός από τις σταθερές και άλλα προφανή πράγματα), προς όφελός του και προς όφελος όλων όσων θα διαβάσουν τον κώδικα στο μέλλον. Αυτό δίνει επίσης την ευκαιρία να συνδυάσουμε τις επισημειώσεις τύπων και τους ορισμούς σε ένα και να αποφύγουμε την επανάληψη του αναγνωριστικού για και τα δύο.

Ένα άλλο δύσκολο κομμάτι της εκμάθησης της Haskell ήταν η κατανόηση των λέξεων-κλειδιών. Οι λέξεις ”data”, ”type”, ”newtype”, ”class”, ”instance” δεν είναι όλες πολύ περιγραφικές. Ίσως πιο περιγραφικές και διαισθητικές λέξεις-κλειδιά θα μπορούσαν να βοηθήσουν τον νέο χρήστη να μάθει πιο γρήγορα και να θυμάται πώς να τις χρησιμοποιεί με μεγαλύτερη ευκολία.

Τέλος, βρέθηκα να χρησιμοποιώ την επέκταση LambdaCase και τη σύνταξη που τη συνοδεύει συνέχεια. Ήταν τόση χαρά που μπορούσα να αποφύγω να δίνω ονόματα στα πράγματα ακόμα περισσότερο από ό,τι το απέφευγα ήδη χρησιμοποιώντας τη σύνθεση συναρτήσεων και όλα τα άλλα εκπληκτικά εργαλεία της Haskell. Διαπίστωσα ότι αυτή η σύνταξη θα μπορούσε να αναπτυχθεί περαιτέρω για πολλαπλές παραμέτρους και μου φάνηκε πολύ χρήσιμη. Από εδώ προέρχεται και το όνομα της γλώσσας.

Ο σκοπός αυτής της εργασίας είναι να αντιμετωπιστούν όλα τα παραπάνω (και λίγο ακόμα) και να συνδυαστούν σε μια νέα γλώσσα.

Κεφάλαιο 2

Σύγκριση Μικρού Προγράμματος: C, Haskell, lcases

Σε αυτό το κεφάλαιο συγκρίνουμε υλοποιήσεις ενός μικρού προγράμματος σε C, Haskell και lcases. Το πρόγραμμα διαβάζει δύο ακεραίους από το stdin (πετάει σφάλμα αν δεν μπορέσει) και δείχνει ένα ωραίο μήνυμα με τον μέγιστο κοινό διαιρέτη. Αρχικά, συγκρίνουμε την υλοποίηση σε lcases με την υλοποίηση σε C και μετά με την υλοποίηση σε Haskell. Προφανώς, εφόσον οι υλοποιήσεις είναι γραμμένες από εμένα, είναι γραμμένες με το στιλ που γράφω εγώ σε αυτές τις γλώσσες. Ωστόσο, προσπάθησα να τις γράψω όσο το δυνατόν με τρόπο που αντιλαμβάνομαι εγώ ως στάνταρ τρόπο γραφής τους.

Σύγκριση με C

```c
#include<stdio.h>

int my_gcd(int a, int b) {
  if (b == 0)
    return a;
  else
    return my_gcd(b, a%b);
}

int main(){

  int x, y;

  printf("Please give me 2 ints\n");
  if (scanf("%d %d", &x, &y) != 2) {
    fprintf(stderr, "You didn't give me 2 ints\n");
    return 1;
  }

  printf("The GCD of %d and %d is %d\n", x, y, my_gcd(x, y));
  return 0;
}
```

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => my_gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print("Please give me 2 ints");
    get_line ;> split(_)to_words o> cases
      [s1, s2] => (from_string(s1), from_string(s2)) -> (_)from_io
      ... => throw_err("You didn't give me 2 ints")

ints(_, _, _)to_message: Int^3 => String
  = (x, y, gcd) => "The GCD of " + x + " and " + y + " is " + gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    ints(i1, i2, my_gcd_of(i1)and(i2))to_message -> print(_)
```

Μια βασική διαφορά μεταξύ της C και της lcases είναι ότι στην lcases εμφανίζεται ξεχωριστά το όνομα, ξεχωριστά η έκφραση του τύπου και ξεχωριστά η έκφραση της ανάθεσης. Αυτό φαίνεται συγκρίνοντας την συνάρτηση ”my_gcd” στο C κώδικα με την ”my_gcd_of(_)and(_)” στον lcases κώδικα. Συγκεκριμένα, συγκρίνοντας την γραμμή:

```
int my_gcd(int a, int b) {
```

με την γραμμή:

```
my_gcd_of(_)and(_): Int^2 => Int
```

βλέπουμε ότι το γεγονός ότι η συνάρτηση δέχεται δύο ακεραίους και επιστρέφει έναν ακέραιο γράφεται στην C δίνοντας τον τύπο **int** σε καθεμία από τις παραμέτρους και στο αποτέλεσμα, ενώ στην lcases όλα είναι γραμμένα σε μία ξεχωριστή έκφραση τύπου που δεν έχει ονόματα παραμέτρων ή συνάρτησης. Επίσης, στην ίδια γραμμή lcases βλέπουμε ότι το όνομα της συνάρτησης έχει παρένθεση στην μέση, κάτι το οποίο κάνει την συνάρτηση να δέχεται το πρώτο όρισμα σε εκείνο το σημείο. Η συνάρτηση καλείται αναδρομικά με ορίσματα στις παρενθέσεις στην γραμμή:

```
y => my_gcd_of(y)and((x)mod(y))
```

όπου μπορούμε επίσης να δούμε ότι η `(_)mod(_)` είναι ορισμένη έτσι ώστε να δέχεται το πρώτο όρισμά της στην αρχή. Αντίστοιχα, η `ints(_, _, _)to_message` δέχεται και τα τρία ορίσματά της στην ίδια παρένθεση στην μέση.

Άλλη μία διαφορά άξια αναφοράς είναι η χρήση της λέξης κλειδιού ”cases” στις γραμμές:

```
= (x, cases)
  0 => x
  y => my_gcd_of(y)and((x)mod(y))
```

αντί της εντολής ”if-then” στις γραμμές:

```
if (b == 0)
  return a;
else
  return my_gcd(b, a%b);
```

Χρησιμοποιώντας την λέξη κλειδί ”cases” στην θέση της δεύτερης παραμέτρου υποδεικνύουμε ότι θα πάρουμε περιπτώσεις για τις τιμές της. Αυτό μπορεί να γίνει για οποιοδήποτε υποσύνολο των παραμέτρων (βλέπε ενότητα 4.3.2).

Μια άλλη διαφορά είναι η χρήση του τελεστή πρόσθεσης για την ένωση συμβολοσειρών και για την αυτόματη μετατροπή των ακεραίων σε συμβολοσειρές ώστε να συνενωθούν στην γραμμή:

```
= (x, y, gcd) => ”The GCD of ” + x + ” and ” + y + ” is ” + gcd
```

αντί της χρήσης της ”printf” με ”%d” στην γραμμή:

```
printf(”The GCD of %d and %d is %d\n”, x, y, my_gcd(x, y));
```

Σύγκριση με Haskell

```haskell
my_gcd :: Int -> Int -> Int
my_gcd = \x y -> case y of
  0 -> x
  _ -> my_gcd y (mod x y)

read_two_ints :: IO (Int, Int)
read_two_ints = do
  putStrLn "Please give me 2 ints"
  s <- getLine
  case words s of
    [s1, s2] -> return (read s1, read s2)
    _ -> error "You didn't give me 2 ints"

ints_to_message :: Int -> Int -> Int -> String
ints_to_message = \x y gcd ->
  "The GCD of " ++ show x ++ " and " ++ show y ++ " is " ++ show gcd

main :: IO ()
main = do
  (i1, i2) <- read_two_ints
  putStrLn $ ints_to_message i1 i2 (my_gcd i1 i2)
```

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => my_gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print("Please give me 2 ints");
    get_line ;> split(_)to_words o> cases
      [s1, s2] => (from_string(s1), from_string(s2)) -> (_)from_io
      ... => throw_err("You didn't give me 2 ints")

ints(_, _, _)to_message: Int^3 => String
  = (x, y, gcd) => "The GCD of " + x + " and " + y + " is " + gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    ints(i1, i2, my_gcd_of(i1)and(i2))to_message -> print(_)
```

Μία σημαντική διαφορά μεταξύ της Haskell και της lcases είναι ότι η εφαρμογή συνάρτησης στην lcases γίνεται με παρενθέσεις αντί για κενά κάτι το οποίο επιτρέπει την τοποθέτηση ορισμάτων στην μέση του ονόματος της συνάρτησης. Αυτό φαίνεται στην σύγκριση της γραμμής:

`my_gcd :: Int -> Int -> Int`

με την γραμμή:

`my_gcd_of(_)and(_): Int^2 => Int`

Στις ίδιες γραμμές φαίνεται ότι χρησιμοποιούμε έναν τύπο εισόδου και έναν τύπο εξόδου (όπως στην Haskell), όμως στην είσοδο έχουμε το γινόμενο των τύπων των ορισμάτων. Αυτό ίσως φαίνεται περιοριστικό αλλά αντιθέτως είναι πιο γενικό εφόσον στην lcases είναι εφικτό να δώσουμε ένα υποσύνολο των ορισμάτων (βάζοντας κάτω παύλα στα υπόλοιπα) και το αποτέλεσμα είναι μία συνάρτηση που περιμένει τα υπόλοιπα ορίσματα (εκτός αν δώθηκαν όλα), βλέπε ενότητα 4.1.3. Στην Haskell τα ορίσματα πρέπει να δωθούν από αριστερά προς τα δεξιά και μπορούμε να μην δώσουμε τα n δεξιότερα για να έχουμε αποτέλεσμα μία συνάρτηση που δέχεται αυτά τα n. Ωστόσο, αν για παράδειγμα θέλουμε να δώσουμε μόνο το δεύτερο πρέπει να χρησιμοποιήσουμε την ”flip” πρώτα. Για πιο περίπλοκα αντίστοιχα πράγματα πρέπει να ορίσουμε συναρτήσεις παρόμοιες της ”flip” κάθε φορά με το χέρι.

Επίσης στις ίδιες γραμμές βλέπουμε την χρήση του τύπου δύναμης `Int^2` ο οποίος δεν υπάρχει στην Haskell.

Συγκρίνοντας τις γραμμές:

```
my_gcd :: Int -> Int -> Int
my_gcd = \x y -> case y of
```

με τις γραμμές:

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
```

βλέπουμε ότι στην lcases η επισημείωση τύπου και η ανάθεση ομαδοποιούνται για την αποφυγή της διπλής χρήσης του ονόματος.

Συγκρίνοντας τις γραμμές:

```
= (x, cases)
  0 => x
  y => my_gcd_of(y)and((x)mod(y))
```

με τις γραμμές:

```
my_gcd = \x y -> case y of
  0 -> x
  _ -> my_gcd y (mod x y)
```

βλέπουμε ότι στην lcases χρησιμοποιούμε κατευθείαν την λέξη κλειδί ”cases” στην θέση της παραμέτρου για την οποία θέλουμε να πάρουμε περιπτώσεις χωρίς να της δίνουμε όνομα. Το όνομα δίνεται (αν χρειάζεται) στην γενική/τελευταία περίπτωση (αν υπάρχει). Η λέξη κλειδί ”cases” μπορεί να χρησιμοποιηθεί σε οποιοδήποτε υποσύνολο των παραμέτρων για να πάρουμε περιπτώσεις για όλες μαζί ταυτόχρονα (βλέπε 4.3.2).

Συγκρίνοντας τις γραμμές:

```
= print(”Please give me 2 ints”);
  get_line ;> split(_)to_words o> cases
```

με τις γραμμές:

```
putStrLn ”Please give me 2 ints”
s <- getLine
case words s of
```

βλέπουμε ότι στην lcases δεν χρησιμοποιούμε do notation. Αντί αυτού χρησιμοποιούμε τους τελεστές περιβάλλοντος ”;” και ”;>” οι οποίοι ισοδυναμούν με τους τελεστές της Haskell ”>>” και ”>>=” αντίστοιχα.

Η τελευταία διαφορά που θα αναφέρουμε είναι η χρήση του τελεστή πρόσθεσης για την ένωση συμβολοσειρών και για την αυτόματη μετατροπή των ακεραίων σε συμβολοσειρές ώστε να συνενωθούν στην γραμμή:

```
= (x, y, gcd) => ”The GCD of ” + x + ” and ” + y + ” is ” + gcd
```

αντί της χρήσης του τελεστή ”++” και της ”show” στην γραμμή:

```
”The GCD of ” ++ show x ++ ” and ” ++ show y ++ ” is ” ++ show gcd
```

Κεφάλαιο 3

Περιγραφή της Γλώσσας: Γενικά

3.1 Δομή Προγράμματος

Ένα πρόγραμμα lcases αποτελείται από ένα σύνολο ορισμών, παρατσουκλιών τύπων και θεωρημάτων. Οι ορισμοί είναι χωρισμένοι σε ορισμούς τιμών, ορισμούς τύπων και ορισμούς προτάσεων τύπων. Τα θεωρήματα είναι αποδεδειγμένες προτάσεις τύπων. Οι συναρτήσεις καθώς και οι ”δράσεις περιβάλλοντος” θεωρούνται επίσης τιμές. Ο ορισμός της τιμής ”main” καθορίζει την συμπεριφορά του προγράμματος.

Παράδειγμα Προγράμματος: Ευκλείδιος Αλγόριθμος

```
gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print(”Please give me 2 ints”);
    get_line ;> split(_)to_words o> cases
      [x, y] => (from_string(x), from_string(y)) -> (_)from_io
      ... => throw_err(”You didn’t give me 2 ints”)

tuple_type NumsAndGcd
value (x, y, gcd):Int^3

nag(_)to_message: NumsAndGcd => String
  = nag => ”The GCD of ” + nag.x + ” and ” + nag.y + ” is ” + nag.gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    (i1, i2, gcd_of(i1)and(i2)) -> nag(_)to_message -> print(_)
```

3.2 Λέξεις κλειδιά

Οι λέξεις κλειδιά της lcases είναι:

```
cases all where tuple_type value or_type values
type_proposition needed equivalent type_theorem proof
```

Η χρήση της κάθεμίας περιγράφεται στην αντίστοιχη ενότητα του παρακάτω πίνακα:

Keyword	Section
`cases`	4.3.2 Εκφράσεις Συναρτήσεων ”cases”
`all where`	4.4 Ορισμοί Τιμών και Εκφράσεις ”where”
`tuple_type value or_type values type_nickname`	5.1 Τύποι
`type_proposition needed equivalent type_theorem proof`	5.2 Λογική Τύπων

Οι λέξεις κλειδιά ”`cases`” και ”`where`” είναι επίσης κατοχυρωμένες λέξεις. Επομένως, παρόλο που μπορούν να παραχθούν από την γραμματική τον ονομάτων, δεν μπορούν να χρησιμοποιηθούν σαν ονόματα.

Κεφάλαιο 4

Περιγραφή της Γλώσσας: Τιμές

4.1 Βασικές Εκφράσεις

4.1.1 Σταθερές και Ονόματα

Σταθερές

- *Παραδείγματα*

```
1  2  17  42  -100
1.62  2.72  3.14  -1234.567
'a'  'b'  'c'  'x'  'y'  'z'  '.'  ','  '\n'
"Hello World!"  "What's up, doc?"  "Alrighty then!"
```

- *Περιγραφή*

  Υπάρχουν σταθερές για τους τέσσερις βασικούς τύπους: `Int, Real, Char, String`.

Ονόματα

- *Παραδείγματα*

```
x y z
a1 a2 a3
(_)mod(_)
apply(_)to_all_in(_)
```

- *Περιγραφή*

  Τα ονόματα χρησιμοποιούνται για τιμές και παραμέτρους. Ένα όνομα τιμής χρησιμοποιείται στον ορισμό της τιμής και στις εκφράσεις που χρησιμοποιούν αυτή την τιμή. Ένα όνομα παραμέτρου χρησιμοποιείται στο σώμα της συνάρτησης της οποίας είναι παράμετρος.

  Ένα όνομα ξεκινάει με μικρό αγγλικό γράμμα και μπορεί να ακολουθηθεί από μικρά αγγλικά γράμματα ή κάτω παύλα και/ή να έχει ένα ψηφίο στο τέλος. Είναι επίσης δυνατόν να υπάρχουν κάτω παύλες μέσα σε παρένθεση πριν, μετά ή και κάπου στην μέση του ονόματος (η χρήση τους περιγράφεται στην ενότητα "Εφαρμογή Συνάρτησης με Παρενθέσεις" 4.1.3)

4.1.2 Παρενθέσεις, Πλειάδες και Λίστες

Παρενθέσεις

- *Παραδείγματα*

```
(1 + 2)
(((1 + 2) * 3)^4)
(n => 3*n + 1)
(get_line ;> line => print(”Line is: ” + line))
```

- *Περιγραφή*

  Μια έκφραση μπαίνει σε παρένθεση για να της δοθεί προτεραιότητα ή να απομονωθεί σε μια μεγαλύτερη έκφραση (τελεστών). Οι εκφράσεις μέσα σε παρένθεση είναι εκφράσεις τελεστών ή εκφράσεις συναρτήσεων.

  Οι εκφράσεις μέσα σε παρένθεση δεν μπορούν να επεκταθούν σε περισσότερες από μία γραμμές. Για εκφράσεις μεγαλύτερες από μία γραμμή, πρέπει να οριστεί νέα τιμή.

Πλειάδες

- *Παραδείγματα*

```
(1, ”What’s up, doc?”)
(2, ”Alrighty then!”, 3.14)
(x, y, z, w)
(1, my_function, (x, y, z) => sqrt(x^2 + y^2 + z^2))
```

- *Περιγραφή*

  Οι πλειάδες χρησιμοποιούνται για να ομαδοποιήσουν πολλές τιμές (που μπορούν να έχουν διαφορετικό τύπο) σε μία. Ο τύπος της πλειάδας είναι είτε το γινόμενο των τύπων των στοιχείων της είτε ένας ορισμένος τύπος `tuple_type` που είναι ισοδύναμος με τον προαναφερθέντα τύπο γινομένου. Για παράδειγμα, ο τύπος της δεύτερης πλειάδας στα παραδείγματα θα μπορούσε να είναι:

  ```
  Int x String x Real
  ```

  ή:

  ```
  MyType
  ```

  υποθέτοντας ότι ο τύπος ”`MyType`” έχει ορισμό παρόμοιο με τον παρακάτω:

  ```
  tuple_type MyType
  value (my_int, my_string, my_real) : Int x String x Real
  ```

- *Πλειάδες με κενά στοιχεία*

  Παραδείγματα

  ```
  (42, _)
  (_, 3.14, _)
  (_, _, ”Hello from 3rd field”)
  ```

  Περιγραφή

  Είναι εφικτό να μείνουν κενά κάποια στοιχεία μια πλειάδας χρησιμοποιώντας κάτω παύλα στην θέση τους. Η χρήση κενών στοιχείων δημιουργεί μία συνάρτηση που περιμένει τα κενά στοιχεία σαν ορίσματα για να επιστρέψει την συνολική πλειάδα. Αυτό φαίνεται καλύτερα από τους τύπους των παραπάνω παραδειγμάτων:

  ```
  (42, _) : T1 => Int x T1
  (_, 3.14, _) : T1 x T2 => T1 x Real x T2
  (_, _, ”Hello from 3rd field”) : T1 x T2 => T1 x T2 x String
  ```

  Ένα παράδειγμα σε μεγαλύτερη έκφραση είναι το παρακάτω:

  ```
  questions : ListOf(String)s
    = [”the Ultimate Question of Life”, ”the Universe”, ”Everything”]

  answers_to : ListOf(String)s
    = apply(”The answer to ” + _)to_all_in(questions)

  >> apply((42, _))to_all_in(answers_to)
    : ListOf(Int x String)s
    ==> [ (42, ”The answer to the Ultimate Question of Life”)
        , (42, ”The answer to the Universe”)
        , (42, ”The answer to Everything”)
        ]
  ```

Λίστες

- *Παραδείγματα*

  ```
  [1, 2, 17, 42, -100]
  [1.62, 2.72, 3.14, -1234.567]
  [”Hello World!”, ”What’s up, doc?”, ”Alrighty then!”]
  [x => x + 1, x => x + 2, x => x + 3]
  [x, y, z, w]
  ```

- *Περιγραφή*

  Οι λίστες χρησιμοποιούνται για την ομαδοποίηση πολλών τιμών του ίδιου τύπου σε μία. Ο τύπος της λίστας είναι `ListOf(T1)s` όπου `T1` είναι ο τύπος κάθε στοιχείου της λίστας. Επομένως, οι τύποι των τεσσάρων πρώτων παραδειγμάτων είναι:

  ```
  ListOf(Int)s
  ListOf(Real)s
  ListOf(String)s
  (@A)And(Int)Add_To(@B) --> ListOf(@A => @B)s
  ```

4.1.3 Εφαρμογή Συνάρτησης με Παρενθέσεις

- *Παραδείγματα*

```
f(x)
f(x, y, z)
(x)to_string
apply(f)to_all_in(l)
```

- *Περιγραφή*

  Η εφαρμογή συνάρτησης στην lcases μπορεί να γίνει με πολλούς τρόπους. Σε αυτή την ενότητα περιγράφονται οι τρόποι με τους οποίους γίνεται εφαρμογή συνάρτησης με παρενθέσεις.

  Στα πρώτα δύο παραδείγματα έχουμε την συνηθισμένη μαθηματική εφαρμογή συνάρτησης όπου τα ορίσματα είναι μετά το όνομα της συνάρτησης.

  Στην lcases επεκτείνεται αυτή η ιδέα επιτρέποντας να υπάρχουν ορίσματα πριν ή/και κάπου μέσα στο όνομα της συνάρτησης (παραδείγματα 3 και 4). Αυτό μπορεί να γίνει μόνο αν η συνάρτηση είναι ορισμένη με τις αντίστοιχες θέσεις όπου τοποθετούνται τα ορίσματα. Για παράδειγμα, ο ορισμός για την "`apply(_)to_all_in(_)`" ξεκινάει ως εξής:

```
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = <...definition>
```

  Το όνομα της συνάρτησης συμπεριλαμβάνει τις παρενθέσεις με τις κάτω παύλες οι οποίες λειτουργούν ως θέσεις για ορίσματα. Αυτό είναι πολύ χρήσιμο για τον ορισμό συναρτήσεων όπου τα ορίσματα μέσα ή πριν την συνάρτηση κάνουν την εφαρμογή της συνάρτησης να φαίνεται και να ακούγεται περισσότερο σαν φυσική γλώσσα.

- *Κενά Ορίσματα σε Εφαρμογή Συνάρτησης με Παρενθέσεις*

  Είναι εφικτό να δοθεί σε μία συνάρτηση ένα υποσύνολο των ορισμάτων της βάζοντας κάτω παύλα στα υπόλοιπα ορίσματα. Μια τέτοια έκφραση είναι μια συνάρτηση που περιμένει τα ορίσματα που λείπουν για να επιστρέψει το τελικό αποτέλεσμα. Παραδείγματα:

```
f(_, _, _) : Char x Int x Real => String
c, i, r : Char, Int, Real

f(c, i, r) : String

f(_, i, r) : Char => String
f(c, _, r) : Int => String
f(c, i, _) : Real => String

f(c, _, _) : Int x Real => String
f(_, i, _) : Char x Real => String
f(_, _, r) : Char x Int => String
```

4.1.4 Συναρτήσεις Προθήματος και Επιθήματος

Συναρτήσεις Προθήματος

- *Παραδείγματα*

```
the_value:1
error:e
result:r
apply(the_value:_)to_all_in(_)
```

- *Περιγραφή*

  Οι συναρτήσεις προθήματος δημιουργούνται αυτόματα από ορισμούς τύπων `or_type`. Είναι συναρτήσεις που μετατρέπουν μία τιμή ενός συγκεκριμένου τύπου σε μια τιμή που είναι περίπτωση ενός `or_type` και περιέχει τιμές του προαναφερθέντα τύπου εσωτερικά. Π.χ. για το πρώτο απ'τα παραπάνω παραδείγματα έχουμε:

  ```
  1 : Int
  the_value:1 : Possibly(Int)
  ```

  Όπου η συνάρτηση `the_value:_` δημιουργήθηκε αυτόματα από το ορισμού του τύπου `Possibly(_)`:

  ```
  or_type Possibly(T1)
  values the_value:T1 | no_value
  ```

  Και έχει τύπο `T1 => Possibly(T1)`.

  Οι συναρτήσεις αυτές αποκαλούνται συναρτήσεις προθήματος γιατί τοποθετούνται πριν από το όρισμά τους. Μπορούν όμως να χρησιμοποιηθούν και σαν ορίσματα άλλων συναρτήσεων τοποθετώντας μία κάτω παύλα στο όρισμά τους. Αυτό φαίνεται στο τελευταίο παράδειγμα, όπου η συνάρτηση `the_value:_` είναι όρισμα της συνάρτησης `apply(_)to_all_in(_)`.

Συναρτήσεις Επιθήματος

- *Παραδείγματα*

```
name.first_name
list.head
date.year
tuple.1st
apply(_.1st)to_all_in(_)
```

- *Περιγραφή*

  Οι συναρτήσεις επιθήματος δημιουργούνται αυτόματα από ορισμούς τύπων `tuple_type`. Είναι συναρτήσεις που δέχονται μία τιμή κάποιου `tuple_type` τύπου και επιστρέφουν κάποιο στοιχείο της τιμής (ήτοι συναρτήσεις προβολής). Π.χ. για το πρώτο από τα παραπάνω παραδείγματα έχουμε:

  ```
  name : Name
  name.first_name : String
  ```

  Όπου η συνάρτηση `_.first_name` δημιουργήθηκε αυτόματα από τον ορισμό του τύπου `Name`:

  ```
  tuple_type Name
  value (first_name, last_name) : String^2
  ```

  Και έχει τύπο `Name => String`.

  Υπάρχουν επίσης οι παρακάτω ειδικές συναρτήσεις προβολής που λειτουργούν σε όλες τις πλειάδες που είναι τύπου γινομένου: "`_.1st`", "`_.2nd`", "`_.3rd`", "`_.4th`", "`_.5th`". Για το τέταρτο παράδειγμα παραπάνω, υποθέτοντας:

  ```
  tuple : Int x String
  ```

  Έχουμε:

  ```
  tuple.1st : Int
  ```

  Οι γενικοί τύποι των συναρτήσεων αυτών είναι:

  ```
  _.1st : (@A)Is(@B)s_1st --> @B => @A
  _.2nd : (@A)Is(@B)s_2nd --> @B => @A
  ...
  ```

  Οι συναρτήσεις αυτές αποκαλούνται συναρτήσεις επιθήματος γιατί τοποθετούνται μετά από το όρισμά τους. Μπορούν όμως να χρησιμοποιηθούν και σαν ορίσματα άλλων συναρτήσεων τοποθετώντας μία κάτω παύλα στο όρισμά τους. Αυτό απεικονίζεται στο τελευταίο παράδειγμα, όπου η συνάρτηση `_.1st` είναι όρισμα της συνάρτησης `apply(_)to_all_in(_)`.

  Υπάρχει επίσης μία ειδική συνάρτηση επιθήματος που ονομάζεται "`_.change`" και περιγράφεται στην επόμενη παράγραφο.

The ”.change” Function

- *Παραδείγματα*

```
state.change{counter = counter + 1}
tuple.change{1st = 42, 3rd = 17}
point.change{x = 1.62, y = 2.72, z = 3.14}
apply(_.change{1st = 1st + 1})to_all_in(_)
x.change{1st = _, 3rd = _}
```

- *Περιγραφή*

  Οι συνάρτηση ”_.change” είναι μία ειδική συνάρτηση επιθήματος που λειτουργεί σε όλες τις πλειάδες. Επιστρέφει μία νέα πλειάδα η οποία είναι ίδια με την αρχική εκτός από κάποια στοιχεία που αλλάζουν. Το ποια στοιχεία αλλάζουν και σε ποια νέα τιμή προσδιορίζεται μέσα σε άγκιστρα μετά το ”.change”. Τα παρακάτω ειδικά ονόματα μπορούν επίσης να χρησιμοποιηθούν για αναφορά στα στοιχεία μιας πλειάδας τύπου γινομένου: ”1st”, ”2nd”, ”3rd”, ”4th”, ”5th” (παραδείγματα 2, 4 και 5). Αν η πλειάδα είναι κάποιου tuple_type τύπου, χρησιμοποιούνται τα ονόματα των στοιχείων που ορίστηκαν στον ορισμό του τύπου (παραδείγματα 1 και 3). Επομένως, έχουμε υποθέσει τα παρακάτω (ή παρόμοια) ώστε τα παραδείγματα να έχουν ορθούς τύπους:

```
tuple_type MyStateType
value (..., counter, ...) : ... x Int x ...

state : MyStateType


tuple : Int x SomeType x Int (x ...)


tuple_type Point
value (x, y, z) : Real^3

point : Point


apply(_.change{1st = 1st + 1})to_all_in(_)
  : (@A)And(Int)AddTo(@A), (@A)Is(@B)s_1st --> ListOf(@B)s => ListOf(@B)s


x : Int x Real x String
x.change{1st = _, 3rd = _} : Int x String => Int x Real x String
```

  Οι αλλαγές στοιχείων έχουν την μορφή: ”στοιχείο = <έκφραση νέας τιμής>” και χωρίζονται από κόμματα. Οι τιμές της αρχικής πλειάδας μπορούν να χρησιμοποιηθούν στις εκφράσεις των νέων τιμών και χρησιμοποιούνται τα ονόματα των στοιχείων για την αναφορά σε αυτές (παραδείγματα 1 και 4). Επίσης, μπορούν να χρησιμοποιηθούν κάτω παύλες στις νέες τιμές. Αυτό κάνει την έκφραση μία συνάρτηση που περιμένει τις κενές τιμές ως ορίσματα (τελευταίο παράδειγμα).

4.2 Τελεστές

4.2.1 Τελεστές Εφαρμογής Συνάρτησης και Σύνθεσης Συναρτήσεων

Τελεστές Εφαρμογής Συνάρτησης

Τελεστής	Τύπος
`->`	`T1 x (T1 => T2) => T2`
`<-`	`(T1 => T2) x T1 => T2`

Οι τελεστές εφαρμογής συνάρτησης "`->`" και "`<-`" είναι ένας διαφορετικός τρόπος εφαρμογής συνάρτησης από την εφαρμογή με παρένθεση. Είναι φτιαγμένοι ώστε να μοιάζουν με βέλη από το όρισμα προς την συνάρτηση. Οι τελεστές αυτοί είναι πολύ χρήσιμοι για την εφαρμογή πολλών συναρτήσεων στην σειρά χωρίς να χρειάζεται ο αντίστοιχος αριθμός ανοίγματος και κλεισίματος παρενθέσεων. Για παράδειγμα, υποθέτοντας ότι έχουμε τις παρακάτω συναρτήσεις με την συμπεριφορά που υπονοείται από τα ονόματα και τους τύπους τους:

```
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
str_len(_) : String => Int
filter(_)with(_) : ListOf(T1)s x (T1 => Bool) => ListOf(T1)s
(_)is_odd : Int => Bool
sum_ints(_) : ListOf(Int)s => Int
```

Και μια λίστα από συμβολοσειρές:

```
strings : ListOf(String)s
```

Ο παρακάτω είναι ένας απλός τρόπος να πάρουμε το συνολικό αριθμό χαρακτήρων σε όλες τις συμβολοσειρές που έχουν περιττό μήκος:

```
chars_in_odd_length_strings : Int
  = apply(str_len(_))to_all_in(strings) -> filter(_)with((_)is_odd) -> sum_ints(_)
```

Αυτό μπορεί να γίνει ισοδύναμα χρησιμοποιώντας τον τελεστή της αντίθετης κατεύθυνσης:

```
chars_in_odd_length_strings : Int
  = sum_ints(_) <- filter(_)with((_)is_odd) <- apply(str_len(_))to_all_in(strings)
```

Τελεστές Σύνθεσης Συναρτήσεων

Τελεστής	Τύπος
`o>`	`(T1 => T2) x (T2 => T3) => (T1 => T3)`
`<o`	`(T2 => T3) x (T1 => T2) => (T1 => T3)`

Οι τελεστές σύνθεσης συναρτήσεων ”o>” και ”<o” χρησιμοποιούνται για την σύνθεση συναρτήσεων, ο καθένας στην αντίστοιχη κατεύθυνση. Η χρήση του γράμματος ’o’ αποσκοπεί στην ομοιότητα με το σύμβολο σύνθεσης συναρτήσεων στα μαθηματικά ’∘’ και τα σύμβολα ’>’, ’<’ χρησιμοποιούνται ώστε οι τελεστές να ”δείχνουν” από την συνάρτηση που θα εφαρμοστεί πρώτη στην συνάρτηση που θα εφαρμοστεί δεύτερη. Ένα ωραίο παράδειγμα χρήσης των τελεστών σύνθεσης συναρτήσεων είναι το παρακάτω. Υποθέτοντας ότι έχουμε τις παρακάτω συναρτήσεις με την συμπεριφορά που υπονοείται από τα ονόματα και τους τύπους τους:

```
split(_)to_words : String => ListOf(String)s
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
reverse_str(_) : String => String
merge_words(_) : ListOf(String)s => String
```

Μπορούμε να αντιστρέψουμε όλες τις λέξεις σε μια συμβολοσειρά ως εξής:

```
reverse_words_in(_) : String => String
  = split(_)to_words o> apply(reverse_str(_))to_all_in(_) o> merge_words(_)
```

Αυτό μπορεί να γίνει ισοδύναμα χρησιμοποιώντας τον τελεστή της αντίθετης κατεύθυνσης:

```
reverse_words_in(_) : String => String
  = merge_words(_) <o apply(reverse_str(_))to_all_in(_) <o split(_)to_words
```

4.2.2 Αριθμητικοί, Σχεσιακοί και Λογικοί Τελεστές

Αριθμητικοί Τελεστές

Τελεστής	Τύπος
^	(@A)To_The(@B)Is(@C) --> @A x @B => @C
*	(@A)And(@B)Multiply_To(@C) --> @A x @B => @C
/	(@A)Divided_By(@B)Is(@C) --> @A x @B => @C
+	(@A)And(@B)Add_To(@C) --> @A x @B => @C
-	(@A)Minus(@B)Is(@C) --> @A x @B => @C

Οι συνηθισμένοι αριθμητικοί τελεστές λειτουργούν όπως στα μαθηματικά και στις άλλες γλώσσες προγραμματισμού για τους συνηθισμένους τύπους. Ωστόσο, είναι γενικευμένοι. Τα παρακάτω παραδείγματα δείχνουν την γενικότητά τους:

```
>> 1 + 1
: Int
==> 2
>> 1 + 3.14
: Real
==> 4.14
>> 'a' + 'b'
: String
==> "ab"
>> 'w' + "ord"
: String
==> "word"
>> "Hello " + "World!"
: String
==> "Hello World!"
>> 5 * 'a'
: String
==> "aaaaa"
>> 5 * "hi"
: String
==> "hihihihihi"
>> "1,2,3" - ','
: String
==> "123"
```

Ας αναλύσουμε περισσότερο το παράδειγμα της πρόσθεσης. Ο τύπος μπορεί να διαβαστεί ως εξής: ο τελεστής '+' έχει τύπο `@A x @B => @C`, δεδομένου ότι ισχύει η πρόταση `(@A)And(@B)Add_To(@C)`. Η αλήθεια της προτάσεως σημαίνει ότι η πρόσθεση έχει οριστεί για αυτούς τους τρεις τύπους. Επομένως, από τα παραπάνω παραδείγματα μπορούμε να συμπεράνουμε ότι οι παρακάτω προτάσεις είναι αληθείς:

```
(Int)And(Int)Add_To(Int)
(Int)And(Real)Add_To(Real)
(Char)And(Char)Add_To(String)
(Char)And(String)Add_To(String)
(Int)And(Char)Multiply_To(String)
(Int)And(String)Multiply_To(String)
(String)Minus(Char)Is(String)
```

Αυτό επιτρέπει την χρήση των γνώριμων αριθμητικών τελεστών σε τύπους οι οποίοι δεν είναι απαραίτητα αριθμοί αλλά είναι αρκετά προφανές το τι συμπεριφορά θα πρέπει να έχουν για αυτούς τους τύπους. Επιπλέον, οι συμπεριφορά τους μπορεί να οριστεί από τον χρήστη για τύπους ορισμένους επίσης από το χρήστη!

Σχεσιακοί, Λογικοί και bitwise τελεστές

Τελεστής	Τύπος
==	(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
!=	(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool
>	(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool
<	(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool
>=	(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool
<=	(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool
&	(@A)Has_And --> @A^2 => @A
|	(@A)Has_Or --> @A^2 => @A

Οι σχεσιακοί τελεστές είναι επίσης γενικευμένοι. Ο βασικός λόγος για την γενίκευσή τους είναι η σύγκριση αριθμών διαφορετικών τύπων όπως φαίνεται στο παρακάτω παράδειγμα:

```
>> 1
  : Int
  ==> 1
>> 1.1
  : Real
  ==> 1.1
>> 1.1 == 1
  : Bool
  ==> false
>> 1.0 == 1
  : Bool
  ==> true
```

Για να λειτουργήσει το παράδειγμα πρέπει να μπορούμε να συγκρίνουμε ακεραίους με πραγματικούς. Αντίστοιχα, όλοι οι σχεσιακοί τελεστές πρέπει να μπορούν να λειτουργήσουν με ορίσματα διαφορετικών τύπων.

Το λογικό ”και” και το bitwise ”και” έχουν συγχωνευτεί σε ένα γενικό τελεστή ”και” (&). Το ίδιο ισχύει και για τον τελεστή ”ή” (|).

4.2.3 Τελεστές Δράσεων σε Περιβάλλον

Τελεστής	Τύπος
;>	(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)
;	(@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)

Απλό Παράδειγμα Προγράμματος

```
main: (EmptyVal)FromIO
  = print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)
```

Στο παραπάνω παράδειγμα φαίνεται η χρήση των τελεστών δράσεων σε περιβάλλον με τον τύπο περιβάλλοντος `(_)FromIO`, ο οποίος είναι ο τύπος με τον οποίο γίνεται είσοδος/έξοδος στην lcases. Το πως ακριβώς γίνεται αυτό μπορεί να φανεί αν ρίξουμε μια ματιά στους τύπους (όπως πάντα!):

```
print_string(_): String => (EmptyVal)FromIO
print_string(”I’ll repeat the line”): (EmptyVal)FromIO
get_line: (String)FromIO
```

Για τον τελεστή ”έπειτα” έχουμε: `; : (@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)`

Στην παρακάτω έκφραση: `print_string(”I’ll repeat the line”) ; get_line`
ο τελεστής ”έπειτα” έχει τύπο: `(EmptyVal)FromIO x (String)FromIO => Κάποιο Τύπο T`

Ο μόνος τρόπος να ταιριάξουν οι τύποι είναι αν:
`@E(_) = (_)FromIO, T1 = EmptyVal, T2 = String και T = (String)FromIO`
Και λειτουργεί γιατί: `((_)FromIO)Has_Then`

Για την συνολική έκφραση έχουμε:

```
print_string(”I’ll repeat the line”) ; get_line
  : (String)FromIO
```

Για τον τελεστή ”χρησιμοποίησε” έχουμε:
`;> : (@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)`

Στην παρακάτω έκφραση:
`print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)`
ο τελεστής ”χρησιμοποίησε” έχει τύπο:
`(String)FromIO x (String => (EmptyVal)FromIO) => Κάποιο Τύπο T`

Ο μόνος τρόπος να ταιριάξουν οι τύποι είναι αν:
`@E(_) = (_)FromIO, T1 = String, T2 = EmptyVal και T = (EmptyVal)FromIO`
Και λειτουργεί γιατί: `((_)FromIO)Has_Use`

Για την συνολική έκφραση έχουμε:

```
print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)
  : (EmptyVal)FromIO
```

Ακόμα ένα Παράδειγμα Προγράμματος

```
main: (EmptyVal)FromIO
  = print_string(”Hello! What’s your name?”) ; get_line ;> name =>
    print_string(”And how old are you?”) ; get_line ;> age =>
    print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)

print_string(_): String => (EmptyVal)FromIO

print_string(”Hello! What’s your name?”) : (EmptyVal)FromIO

print_string(”Hello! What’s your name?”); get_line
  : (String)FromIO

print_string(”And how old are you?”); get_line
  : (String)FromIO

print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO

age => print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : String => (EmptyVal)FromIO

print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO

name =>
print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : String => (EmptyVal)FromIO

print_string(”Hello! What’s your name?”) ; get_line ;> name =>
print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO
```

Περιγραφή

Οι τελεστές δράσεων σε περιβάλλον χρησιμοποιούνται για να συνδυάσουν τιμές που είναι δράσεις σε περιβάλλον σε τιμές που είναι πιο περίπλοκες δράσεις σε περιβάλλον. Οι τύποι περιβάλλοντος είναι συναρτήσεις τύπων που δέχονται ένα τύπο σαν όρισμα και δημιουργούν έναν άλλον τύπο (όπως ο τύπος `ListOf(_)s`). Οι τελεστές ”έπειτα” (`;`) και ”χρησιμοποίησε” (`;>`) είναι ορισμένοι για τους τύπους περιβάλλοντος. Μια τιμή τύπου `@E(T1)` όπου ο `@E(_)` είναι τύπος περιβάλλοντος κάνει μια δράση στο περιβάλλον `@E(_)` και δημιουργεί μια τιμή τύπου `T1`. Αυτή η δράση μπορεί να συνδυαστεί με τον τελεστή ”έπειτα” με μία άλλη ώστε να γίνει η πρώτη και έπειτα η δεύτερη. Αντίστοιχα, με τον τελεστή ”χρησιμοποίησε” οι τιμή που δημιουργήθηκε από την πρώτη δράση, μπορεί να χρησιμοποιηθεί σαν όρισμα σε συνάρτηση που επιστρέφει μια δεύτερη δράση.

4.2.4 Εκφράσεις Τελεστών

- *Παραδείγματα*

```
1 + 2
1 + x * 3^y
"Hello " + "World!"
x -> f -> g
f o> g o> h
x == y
x >= y - z & x < 2 * y
get_line ; get_line ;> line => print("Second line: " + line)
2 * _
_ - 1
"Hello " + "it's me, " + _
"Hi, I am " + _ + " and I am " + _ + " years old"
```

- *Περιγραφή*

  Οι εκφράσεις τελεστών είναι εκφράσεις που αποτελούνται από τελεστές και τελεστέους. Οι τελεστές συμπεριφέρονται σαν συναρτήσεις δύο ορισμάτων που βρίσκονται ανάμεσα στα ορίσματά τους (τελεστέους). Επομένως, έχουν τύπους συνάρτησης και δρουν όπως περιγράφεται στην αντίστοιχη ενότητα του καθενός.

  Οι εκφράσεις τελεστών μπορεί να έχουν πολλούς τελεστές. Η σειρά με την οποία δρουν περιγράφεται στην επόμενη ενότητα.

  Όπως στις συναρτήσεις, οι τελεστέοι ενός τελεστή πρέπει να έχουν τύπους που ταιριάζουν με τους τύπους που περιμένει ο τελεστής.

  Είναι δυνατόν ο δεύτερος τελεστέος να είναι έκφραση συνάρτησης.

  Είναι επίσης δυνατόν να χρησιμοποιηθούν κάτω παύλες στην θέση τελεστέων. Μια έκφραση τελεστών που έχει τελεστέους κάτω παύλες λειτουργεί ως μία συνάρτηση που περιμένει τους κενούς τελεστέους ως ορίσματα. Αυτό φαίνεται καλύτερα από τους τύπους των τελευταίων τεσσάρων παραδειγμάτων:

```
2 * _ : Int => Int
_ - 1 : Int => Int
"Hello " + "it's me, " + _ : String => String
"Hi, I am " + _ + " and I am " + _ + " years old" : String^2 => String
```

  Σημείωση: Αυτοί δεν είναι οι πιο γενικοί τύποι για τα παραδείγματα αλλά είναι συμβατοί.

4.2.5 Πλήρης Πίνακας Τελεστών, Προτεραιότητα και Προσεταιριστικότητα

Πίνακας 4.1: Ο πλήρης πίνακας τελεστών της lcases μαζί με τους τύπους και της συνοπτικές περιγραφές τους.

Τελεστής	Τύπος	Περιγραφή
`->`	`T1 x (T1 => T2) => T2`	Δεξία Εφαρμογή Συνάρτησης
`<-`	`(T1 => T2) x T1 => T2`	Αριστερή Εφαρμογή Συνάρτησης
`o>`	`(T1 => T2) x (T2 => T3) => (T1 => T3)`	Δεξιά Σύνθεση Συναρτήσεων
`<o`	`(T2 => T3) x (T1 => T2) => (T1 => T3)`	Αριστερή Σύνθεση Συναρτήσεων
`^`	`(@A)To_The(@B)Is(@C) --> @A x @B => @C`	Γενική Ύψωση σε Δύναμη
`*`	`(@A)And(@B)Multiply_To(@C) --> @A x @B => @C`	Γενικός Πολλαπλασιασμός
`/`	`(@A)Divided_By(@B)Is(@C) --> @A x @B => @C`	Γενική Διαίρεση
`+`	`(@A)And(@B)Add_To(@C) --> @A x @B => @C`	Γενική Πρόσθεση
`-`	`(@A)Minus(@B)Is(@C) --> @A x @B => @C`	Γενική Αφαίρεση
`==`	`(@A)And(@B)Can_Be_Equal --> @A x @B => Bool`	Γενική Ισότητα
`!=`	`(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool`	Γενική Ανισότητα
`>`	`(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool`	Γενικό Μεγαλύτερο
`<`	`(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool`	Γενικό Μικρότερο
`>=`	`(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool`	Γενικό Μεγαλύτερο Ίσο
`<=`	`(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool`	Γενικό Μικρότερο Ίσο
`&`	`(@A)Has_And --> @A^2 => @A`	Γενικό Και
`|`	`(@A)Has_Or --> @A^2 => @A`	Γενικό Ή
`;>`	`(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)`	Τελεστής Δράσης ”Χρησιμοποίησε”
`;`	`(@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)`	Τελεστής Δράσης ”Έπειτα”

Η σειρά δράσης των τελεστών γίνεται από μεγαλύτερη σε μικρότερη προτεραιότητα. Στο ίδιο επίπεδο προτεραιότητας η σειρά δράσης είναι από αριστερά προς τα δεξιά αν η προσεταιριστικότητα είναι ”Αριστερή” και από δεξιά προς τα αριστερά αν η προσεταιριστικότητα είναι ”Δεξιά”. Για τους τελεστές όπου ”Δεν Υπάρχει” προσεταιριστικότητα, δεν επιτρέπεται η χρήση τους στην ίδια έκφραση τελεστών. Η προτεραιότητα και προσεταιριστικότητα των τελεστών φαίνεται στον παρακάτω πίνακα.

Πίνακας 4.2: Πίνακας προτεραιότητας και προσεταιριστικότητας των τελεστών της lcases

Τελεστής	Προτεραιότητα	Προσεταιριστικότητα
`->`	10 (highest)	Αριστερή
`<-`	9	Δεξιά
`o> <o`	8	Αριστερή
`^`	7	Δεξιά
`* /`	6	Αριστερή
`+ -`	5	Αριστερή
`== != > < >= <=`	4	Δεν Υπάρχει
`&`	3	Αριστερή
`|`	2	Αριστερή
`;> ;`	1	Αριστερή

4.3 Εκφράσεις Συναρτήσεων

Οι εκφράσεις συναρτήσεων χωρίζονται σε **κανονικές εκφράσεις συναρτήσεων** και **εκφράσεις συναρτήσεων "cases"** και η κάθε κατηγορία περιγράφεται στην αντίστοιχη από τις παρακάτω ενότητες.

4.3.1 Κανονικές Εκφράσεις Συναρτήσεων

- *Παραδείγματα*

```
a => 17 * a + 42
(a, b) => a + 2*b
(x, y, z) => sqrt(x^2 + y^2 + z^2)
* => 42
(x, *, z) => x + z
((x1, y1), (x2, y2)) => (x1 + x2, y1 + y2)
```

- *Περιγραφή*

  Οι κανονικές εκφράσεις συναρτήσεων χρησιμοποιούνται για να ορίσουν συναρτήσεις ή είναι μέρος μιας μεγαλύτερης έκφρασης ως ανώνυμες συναρτήσεις. Αποτελούνται από τις παραμέτρους και το σώμα τους.

  Οι παράμετροι είναι ονόματα. Είτε υπάρχει μόνο μία παράμετρος, στην οποία περίπτωση δεν υπάρχει παρένθεση, είτε υπάρχουν πολλές, στην οποία περίπτωση είναι σε παρένθεση χωρισμένες από κόμματα. Αν μία παράμετρος δεν χρειάζεται μπορεί να μείνει χωρίς όνομα, βάζοντας ένα αστερίσκο στην θέση της (παραδείγματα 4 και 5). Αν μία παράμετρος είναι πλειάδα τύπου γινομένου μπορεί να ταιριαστεί περαιτέρω (να δοθούν ονόματα στα στοιχεία της) ανοίγοντας επιπλέον παρενθέσεις στην θέση της (παράδειγμα 6).

  Οι παράμετροι και το σώμα χωρίζονται από το βέλος συνάρτησης (”=>”).

4.3.2 Εκφράσεις Συναρτήσεων ”cases”

- *Παραδείγματα*

```
print_sentimental_bool(_): Bool => IO
  = cases
    true => print(”It’s true!! :)”)
    false => print(”It’s false... :(”)

or_type TrafficLight
values green | amber | red

(_)is_not_red: TrafficLight => Bool
  = cases
    green => true
    amber => true
    red => false

(_)is_seventeen_or_forty_two: Int => Bool
  = cases
    17 => true
    42 => true
    ... => false

traffic_lights_match(_, _): TrafficLight^2 => Bool
  = (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false

gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => gcd_of(y)and((x)mod(y))

apply(_)to_all_in(_): (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = (f(_), cases)
    [] => []
    [head, tail = ...] => f(head) + apply(f(_))to_all_in(tail)

(_)is_sorted: (@A)Has_Less_Than_Or_Equal --> ListOf(@A)s => Bool
  = cases
    [x1, x2, xs = ...] => (x1 <= x2) & (x2 + xs)is_sorted
    ... => true
```

- *Περιγραφή*

  Η λέξη κλειδί ”cases” λειτουργεί ως ειδική παράμετρος. Αντί να δίνει το όνομα ”cases”, χρησιμοποιείται για να τοποθετήσει την παράμετρο σε μία πλειάδα παραμέτρων που θα ορίσουν περιπτώσεις ή για να οριστούν περιπτώσεις για αυτή την παράμετρο μόνο. Για κάθε ξεχωριστή περίπτωση ορίζεται το αντίστοιχο αποτέλεσμα της συνάρτησης.

  Οι τελευταία περίπτωση μπορεί να είναι ”... => <σώμα γενικής περίπτωσης>” δίνοντας το αποτέλεσμα για όλες τις υπόλοιπες περιπτώσεις χωρίς να δοθεί όνομα στην τιμή (παράδειγμα ”is_seventeen_or_forty_two”), ή μπορεί να είναι
  ”<κάποιο όνομα> => <σώμα γενικής περίπτωσης>” για να δοθεί όνομα στην τιμή και να μπορεί να χρησιμοποιηθεί (”y” στο παράδειγμα ”gcd_of(_)and(_)”).

  Μια έκφραση ”where” μπορεί να τοποθετηθεί κάτω από μία περίπτωση δεδομένου ότι είναι στοιχισμένη δύο κενά πιο μέσα.

  Μπορεί επίσης να χρησιμοποιηθεί η παρακάτω σύνταξη για να δοθεί όνομα σε οσαδήποτε από τα πρώτα στοιχεία μιας λίστας χρειάζεται και προαιρετικά να δοθεί όνομα και στο υπόλοιπο της λίστας.

```
# υπόλοιπο λίστας δεν έχει όνομα
[x1, ...] => <σώμα περίπτωσης>
[x1, x2, ...] => <σώμα περίπτωσης>
[x1, x2, x3, ...] => <σώμα περίπτωσης>

# υπόλοιπο λίστας έχει όνομα
[x1, xs = ...] => <σώμα περίπτωσης>
[x1, x2, xs = ...] => <σώμα περίπτωσης>
[x1, x2, x3, xs = ...] => <σώμα περίπτωσης>
```

4.4 Ορισμοί Τιμών και Εκφράσεις ”where”

4.4.1 Ορισμοί Τιμών

- *Παραδείγματα*

```
foo: Int
  = 42

f(_, _, _): Int^3 => Int
  = (a, b, c) => a + b * c

val1, val2, val3: Int, Bool, Char
  = 42, true, 'a'

int1, int2, int3: all Int
  = 1, 2, 3
```

- *Περιγραφή*

  Οι ορισμοί τιμών είναι το βασικό συστατικό ενός προγράμματος lcases. Για να οριστεί μια νέα τιμή πρέπει να της δοθεί ένα όνομα, ένας τύπος και μια έκφραση. Το όνομα ακολουθείται από το σύμβολο ”έχει τύπο” (’:’) και την έκφραση του τύπου. Η επόμενη γραμμή είναι στοιχισμένη δύο κενά πιο μέσα, ξεκινάει με ίσον και συνεχίζεται με την έκφραση της τιμής (η οποία μπορεί να εκτείνεται σε οσεσδήποτε γραμμές χρειάζεται).

  Ένας ορισμός τιμής μπορεί να ξεκινάει είτε στην πρώτη στήλη όπου μπορούν να τον ”δουν” όλοι οι άλλοι ορισμοί τιμών, είτε είναι μέσα σε μια έκφραση ”`where`” (βλέπε επόμενη ενότητα), όπου μπορεί να τον ”δει” η έκφραση πριν το ”`where`” και όλοι οι άλλοι ορισμοί μέσα στην ίδια έκφραση ”`where`”.

  Ένας ορισμός τιμής μπορεί να ακολουθηθεί από μια έκφραση ”`where`” όπου ορίζονται ενδιάμεσες τιμές που χρησιμοποιούνται στον ορισμό. Στην περίπτωση αυτή η έκφραση ”`where`” πρέπει να στοιχιθεί δύο κενά πιο μέσα από το ίσον του ορισμού.

  Υπάρχει η δυνατότητα ομαδοποίησης ορισμών τιμών χωρίζοντας τα ονόματα, τους τύπους και τις εκφράσεις με κόμματα. Αυτό είναι πολύ χρήσιμο για να μην γεμίζει το πρόγραμμα με μικρούς ορισμούς άσκοπα (π.χ. ορισμούς σταθερών). Σε μια ομάδα ορισμών μπορεί να χρησιμοποιηθεί η λέξη κλειδί ”`all`” για να δοθεί ο ίδιος τύπος σε όλες τις τιμές.

4.4.2 Εκφράσεις ”where”

- *Παραδείγματα*

```
sort(_): ListOf(Int)s => ListOf(Int)s
  = cases
    [] => []
    [head, tail = ...] =>
      sort(less_l) + head + sort(greater_l)
      where
      less_l, greater_l: all ListOf(Int)s
        = filter(tail)with(_ < head), filter(tail)with(_ >= head)

sum_nodes(_): TreeOf(Int)s => Int
  = tree =>
    tree.root + apply(sum_nodes(_))to_all_in(tree.subtrees) -> sum_list(_)
    where
    sum_list(_) : ListOf(Int)s => Int
      = cases
        [] => 0
        [head, tail = ...] => head + sum_list(tail)

big_string : String
  = s1 + s2 + s3 + s4
    where
    s1, s2, s3, s4 : all String
      = ”Hello, my name is Struggling Programmer.”
      , ” I have tried way too many times to fit a big chunk of text”
      , ” inside my program, without it hitting the half-screen mark!”
      , ” I am so glad I finally discovered lcases!”
```

- *Περιγραφή*

  Οι εκφράσεις ”where” επιτρέπουν στον προγραμματιστή να χρησιμοποιήσει τιμές μέσα σε μια έκφραση και να τις ορίσει από κάτω. Είναι πολύ χρήσιμες για την επαναχρησιμοποίηση ή την σύντμηση εκφράσεων που χρησιμοποιούνται σε ένα συγκεκριμένο ορισμό ή σε μια συγκεκριμένη περίπτωση συνάρτησης.

  Μια έκφραση ”where” ξεκινάει με μία γραμμή που έχει μόνο την λέξη ”where”. Στοιχίζεται όπως αναφέρεται στις ενότητες ”Ορισμοί Τιμών” και ”Εκφράσεις Συναρτήσεων ’cases’”. Οι ορισμοί τοποθετούνται κάτω από την γραμμή αυτή και έχουν την ίδια στοίχιση.

Κεφάλαιο 5

Περιγραφή της Γλώσσας: Τύποι και Λογική Τύπων

5.1 Τύποι

Οι έννοιες που αφορούν τους τύπους είναι οι **εκφράσεις τύπων**, οι **ορισμοί τύπων** και τα **παρατσούκλια τύπων** και περιγράφονται στις αντίστοιχες ενότητες παρακάτω.

5.1.1 Εκφράσεις Τύπων

Οι εκφράσεις τύπων χωρίζονται στις εξής κατηγορίες:

- Ονόματα Τύπων
- Μεταβλητές Τύπων
- Τύποι Εφαρμογής Τύπου
- Τύποι Γινόμενα
- Τύποι Συναρτήσεων
- Τύποι Με Προϋπόθεση

που περιφράφονται στις παρακάτω παραγράφους.

Ονόματα Τύπων

- *Παραδείγματα*

```
Int     Real     Char     String     SelfReferencingType
```

- *Περιγραφή*

  Ένα όνομα τύπου είναι είτε το όνομα ενός εκ των βασικών τύπων (Int, Real, Char, String) ή το όνομα ενός ορισμένου τύπου που δεν έχει παραμέτρους τύπων. Ξεκινάει με κεφαλαίο αγγλικό γράμμα και ακολουθείται από κεφαλαία ή μικρά αγγλικά γράμματα.

Μεταβλητές Τύπων

Οι μεταβλητές τύπων βρίσκονται μέσα σε εκφράσεις τύπων και μπορούν να αντικατασταθούν με συγκεκριμένους τύπους ανάλογα με την περίπτωση. Αυτό κάνει τις ευρύτερες εκφράσεις τύπων (στις οποίες εμφανίζονται οι μεταβλητές τύπων) **πολυμορφικούς** τύπους. Οι τύποι πολυμορφισμού που υπάρχουν στην lcases είναι ο **παραμετρικός πολυμορφισμός** και ο **ad hoc πολυμορφισμός**. Οι μεταβλητές τύπων για τον καθένα από τους δύο πολυμορφισμούς έχουν διαφορετική σύνταξη και περιγράφονται στις επόμενες παραγράφους.

Παραμετρικές Μεταβλητές Τύπων

- *Παραδείγματα*

```
T1     T2     T3
```

- *Παραδείγματα παραμετρικών μεταβλητών τύπων μέσα σε μεγαλύτερες εκφράσεις τύπων*

```
T1 => T1
(T1 => T2) x (T2 => T3) => (T1 => T3)
(T1^2 => T1) x T1 x ListOf(T1)s => T1
```

- *Περιγραφή*

  Οι παραμετρικές μεταβλητές τύπων μπορούν να αντικατασταθούν από οποιονδήποτε τύπο και θα λειτουργήσει το πρόγραμμα (δεδομένου ότι είναι ο ίδιος παντού). Το πιο απλό παράδειγμα πολυμορφικού τύπου με παραμετρική μεταβλητή τύπου είναι ο τύπος της ταυτοτικής συνάρτησης:

```
id(_): T1 => T1
  = x => x

id(1): Int
  όπου ο T1 αντικαθίσταται από τον Int και η id έχει τύπο Int => Int

id(”Hello”): String
  όπου ο T1 αντικαθίσταται από τον String και η id έχει τύπο String => String
```

  Μια παραμετρική μεταβλητή τύπου γράφεται με κεφαλαίο αγγλικό ”T” και ακολουθείται από ένα ψηφίο.

Μεταβλητές Τύπων Ad Hoc

- *Παραδείγματα*

```
@A @B @C @T
```

- *Παραδείγματα μεταβλητών τύπων ad hoc μέσα σε μεγαλύτερες εκφράσεις τύπων*

```
(@T)Has_Str_Rep --> @T => String
(@A)Is(@B)s_First --> @B => @A
(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
(@A)And(@B)Add_To(@C) --> @A x @B => @C
```

- *Περιγραφή*

  Οι μεταβλητές τύπων ad hoc μπορούν να αντικατασταθούν μόνο από τύπους που ικανοποιούν μία προϋπόθεση. Οι προϋπόθεση έχει τη μορφή μιας πρότασης τύπων (βλέπε ενότητα Λογική Τύπων 5.2). Επομένως, κάθε μεταβλητή τύπου ad hoc πρέπει να εμφανίζεται και στην προϋπόθεση του τύπου.

  Μια μεταβλητή τύπου ad hoc γράφεται με '@' και ακολουθείται από κάποιο κεφαλαίο αγγλικό γράμμα.

Τύποι Εφαρμογής Τύπου

- *Παραδείγματα*

```
Possibly(Int)
ListOf(Real)s
TreeOf(String)s
Result(Int)OrError(String)
ListOf(Int => Int)s
ListOf(T1)s
```

- *Περιγραφή*

  Οι τύποι εφαρμογής τύπου είναι τύποι που δημιουργούνται δίνοντας τύπους ορίσματα σε συναρτήσεις τύπων που έχουν δημιουργηθεί από έναν ορισμό `tuple_type`, έναν ορισμό `or_type` ή ένα παρατσούκλι τύπου. Για παράδειγμα, δεδομένου του ορισμού του `Possibly(T1)`:

```
or_type Possibly(T1)
values the_value:T1 | no_value
```

  Έχουμε σαν συνάρτηση τύπων την `Possibly(_)`, η οποία δέχεται ένα τύπο σαν όρισμα. Π.χ. ο `Possibly(Int)` είναι ο τύπος αποτέλεσμα της εφαρμογής της `Possibly(_)` στον τύπο `Int`.

  Οι τύποι εφαρμογής τύπου έχουν την ίδια μορφή με το όνομα που ορίστηκε στον ορισμό τύπου ή στο παρατσούκλι από το οποίο προήλθε η συνάρτηση τύπου με την διαφορά ότι οι παράμετροί της έχουν αντικατασταθεί με τις εκφράσεις των τύπων ορισμάτων.

Τύποι Γινόμενα

- *Παραδείγματα*

```
Int x Real x String
ListOf(Int)s x Int x ListOf(String)s
(Int => Int) x (Int x Real) x (Real => String)
Int^2 x Int^2
Real^3 x Real^3
```

- *Περιγραφή*

  Οι τύποι γινόμενα είναι οι τύποι των πλειάδων που δεν ανήκουν σε κάποιο ορισμένο τύπο `tuple_type`. Αποτελούνται από τις εκφράσεις των τύπων των στοιχείων της πλειάδας, οι οποίοι χωρίζονται από την συμβολοσειρά ” x ” (κενό ’x’ κενό) λόγω της ομοιότητας με το σύμβολο του καρτεσιανού γινομένου. Αν κάποιο από τα στοιχεία έχει τύπο γινομένου ή συνάρτησης τότε η έκφραση του τύπου αυτού πρέπει να μπει μέσα σε παρένθεση. Ένας τύπος γινόμενο όπου όλα τα στοιχεία έχουν τον ίδιο τύπο μπορεί να συντμηθεί με μία έκφραση τύπου δύναμης η οποία αποτελείται από τον τύπο των στοιχείων, το σύμβολο της δύναμης ’^’ και το πλήθος των στοιχείων.

Τύποι Συναρτήσεων

- *Παραδείγματα*

```
String => String
Real => Int
T1 => T1
Int^2 => Int
Real^3 => Real
(T1 => T2) x (T2 => T3) => (T1 => T3)
(Int => Int) => (Int => Int)
```

- *Περιγραφή*

  Μία έκφραση τύπου συνάρτησης αποτελείται από την έκφραση του τύπου εισόδου και την έκφραση του τύπου εξόδου χωρισμένες από το βέλος συνάρτησης (”=>”). Οι εκφράσεις των τύπων εισόδου και εξόδου μπαίνουν σε παρένθεση αν είναι επίσης εκφράσεις τύπων συνάρτησης.

Τύποι Με Προϋπόθεση

- *Παραδείγματα*

```
(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
(@A)And(@B)Add_To(@C) --> @A x @B => @C
(@A)Is(@B)s_First --> @B => @A
(@T)Has_Str_Rep --> @T => String
(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)
```

- *Περιγραφή*

  Οι τύποι με προϋπόθεση είναι οι τύποι των τιμών που είναι ad hoc πολυμορφικές, δηλαδή έχουν οριστεί για πολλούς διαφορετικούς συνδυασμούς τύπων. Αποτελούνται από την προϋπόθεση και τον ”απλό” τύπο (ήτοι τον τύπο χωρίς την προϋπόθεση) και χωρίζονται από το βέλος προϋπόθεσης (” `-->` ”). Η προϋπόθεση είναι μία πρόταση τύπων η οποία αναφέρεται σε ad hoc μεταβλητές τύπων μέσα στον ”απλό τύπο” και πρέπει να ισχύει για να χρησιμοποιηθεί μια πολυμορφική τιμή που έχει αυτόν τον τύπο. Για παράδειγμα η τιμή:

  ```
  (_)first: (@A)Is(@B)s_First --> @B => @A
  ```

  μπορεί να χρησιμοποιηθεί ως εξής:

  ```
  pair, triple, list
    : Int x String, Real x Char x Int, ListOf(String)s
    = (42, ”The answer to everything”), (3.14, ’a’, 1), [”Hi!”, ”Hello”, Heeey”]

  >> (pair)first
    : Int
    ==> 42
  >> (triple)first
    : Real
    ==> 3.14
  >> (list)first
    : String
    ==> ”Hi!”
  ```

  και αυτό διότι οι παρακάτω προτάσεις ισχύουν:

  ```
  (Int)Is(Int x String)s_First
  (Real)Is(Real x Char x Int)s_First
  (String)Is(ListOf(String)s)s_First
  ```

  το οποίο με την σειρά του σημαίνει ότι η συνάρτηση ”`(_)first`” έχει οριστεί για αυτούς τους συνδυασμούς τύπων.

5.1.2 Ορισμοί Τύπων

Οι ορισμοί τύπων χωρίζονται σε ορισμούς `tuple_type` και ορισμούς `or_type` και περιγράφονται στις παραγράφους που ακολουθούν.

Ορισμοί `tuple_type`

- *Παραδείγματα Ορισμών*

```
tuple_type Name
value (first_name, last_name) : String^2

tuple_type Date
value (day, month, year) : Int^3

tuple_type MathematicianInfo
value (name, nationality, date_of_birth) : Name x String x Date

tuple_type TreeOf(T1)s
value (root, subtrees) : T1 x ListOf(TreeOf(T1)s)s

tuple_type Indexed(T1)
value (index, val) : Int x T1
```

- *Παραδείγματα Χρήσης*

```
euler_info: MathematicianInfo
  = ((”Leonhard”, ”Euler”), ”Swiss”, (15, 4, 1707))

name(_)to_string: Name => String
  = n => ”\nFirst Name: ” + n.first_name + ”\nLast Name: ” + n.last_name

print_name_and_nat(_): MathematicianInfo => IO
  = ci => print(name(ci.name)to_string + ”\nNationality: ” + ci.nationality)

sum_nodes(_): TreeOf(Int)s => Int
  = tree => tree.root + apply(sum_nodes(_))to_all_in(tree.subtrees) -> sum_list(_)
```

- *Περιγραφή*

  Ένας ορισμός `tuple_type` (τύπος πλειάδας) ορίζει ένα νέο τύπο που είναι ισοδύναμος με κάποιο τύπο γινόμενο δίνοντας του όμως νέο όνομα και ονόματα στα στοιχεία για ευκολία. Ένας ορισμός τύπου πλειάδας δημιουργεί αυτόματα συναρτήσεις επιθήματος για καθένα από τα στοιχεία οι οποίες γράφονται με μια τελεία και το όνομα του στοιχείου. Π.χ. το παράδειγμα του ορισμού του τύπου ”`MathematicianInfo`” δημιουργεί τις παρακάτω συναρτήσεις:

```
_.name : MathematicianInfo => Name
_.nationality : MathematicianInfo => String
_.date_of_birth : MathematicianInfo => Date
```

Ορισμοί `or_type`

- *Παραδείγματα Ορισμών*

```
or_type Bool
values true | false

or_type TrafficLight
values green | amber | red

or_type Possibly(T1)
values the_value:T1 | no_value

or_type Result(T1)OrError(T2)
values result:T1 | error:T2
```

- *Παραδείγματα Χρήσης*

```
traffic_lights_match(_, _): TrafficLight^2 => Bool
  = (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false

err_if(_)is_no_value : Possibly(T1) => Result(T1)OrError(String)
  = cases
    no_value => error:"There is no value!"
    the_value:val => result:val

print_err(_)or_res(_): (@A)Has_Str_Rep --> Result(@A)OrError(String) => IO
  = cases
    result:r => print("All good! The result is: " + (r)to_string)
    error:e => print("Error occurred: " + e)
```

- *Περιγραφή*

  Οι τιμές ενός `or_type` χωρίζονται σε διαφορετικές περιπτώσεις. Κάποιες περιπτώσεις έχουν άλλες τιμές εσωτερικά. Οι περιπτώσεις που έχουν εσωτερικές τιμές ακολουθούνται από άνω κάτω τελεία και των τύπο της εσωτερικής τιμής. Αντίστοιχη σύνταξη μπορεί να χρησιμοποιηθεί για να διαλέξουμε την κάθε περίπτωση σε μια συνάρτηση "cases". Ένας ορισμός `or_type` δημιουργεί αυτόματα συναρτήσεις προθήματος για κάθε μία από τις περιπτώσεις που έχουν εσωτερική τιμή. Οι συναρτήσεις αυτές έχουν ως χρήση την μετατροπή μιας τιμής που έχει τύπο των εσωτερικό τύπο μιας περίπτωσης σε μια τιμή που είναι αυτή η περίπτωση και έχει τύπο τον ορισμένο τύπο. Για παράδειγμα, για την περίπτωση "`the_value`" του "`Possibly(T1)`" η συνάρτηση "`the_value:_`" δημιουργήθηκε αυτόματα από το ορισμό και έχει τύπο:

  ```
  the_value:_ : T1 => Possibly(T1)
  ```

  Αυτές οι συναρτήσεις μπορούν να χρησιμοποιηθούν όπως κάθε συνάρτηση σαν ορίσματα άλλων συναρτήσεων. Για παράδειγμα:

  ```
  (_)to_possiblies : ListOf(T1)s => ListOf(Possibly(T1))s
    = apply(the_value:_)to_all_in(_)
  ```

5.1.3 Παρατσούκλια Τύπων

- *Παραδείγματα*

```
type_nickname Ints = ListOf(Int)s
type_nickname IntStringPairs = ListOf(Int x String)s
type_nickname IO = (EmptyVal)FromIO
type_nickname Res(T1)OrErr = Result(T1)OrError(String)
```

- *Περιγραφή*

  Τα παρατσούκλια τύπων χρησιμοποιούνται για την σύντμηση ή για να δοθεί ένα πιο περιγραφικό όνομα σε κάποιον υπάρχον τύπο. Ξεκινούν με την λέξη κλειδί ”`type_nickname`” που ακολουθείται από το παρατσούκλι, ένα ίσον και τον τύπο για τον οποίο δημιουργείται παρατσούκλι. Στο παρατσούκλι μπορούν να χρησιμοποιηθούν παραμετρικές μεταβλητές τύπων.

5.2 Λογική Τύπων

Η λογική τύπων είναι ο μηχανισμός για ad hoc πολυμορφισμό στην lcases. Οι κεντρική έννοια της **λογικής τύπων** είναι η **πρόταση τύπων**. Μια πρόταση τύπων είναι μία πρόταση που δέχεται τύπους ως ορίσματα και είναι αληθής η ψευδής για κάθε συνδυασμό ορισμάτων.

Οι προτάσεις τύπων μπορούν είτε να οριστούν είτε να αποδειχτούν. Οι έννοιες των **ορισμών προτάσεων τύπων** και **θεωρημάτων τύπων** υπάρχουν για τον αντίστοιχο από τους προαναφερθέντες σκοπούς.

Από εδώ και στο εξής δεν θα αναφέρεται η λέξη ”τύπων”, κάθε πρόταση είναι πρόταση τύπων και κάθε θεώρημα είναι θεώρημα τύπων.

5.2.1 Ορισμοί Προτάσεων

Οι ορισμοί προτάσεων χωρίζονται σε ορισμούς **ατομικών προτάσεων** και ορισμούς **προτάσεων μετονομασίας** οι οποίοι περιγράφονται στις παραγράφους που ακολουθούν.

Ατομικές Προτάσεις

- *Παραδείγματα*

```
type_proposition (@A)Is(@B)s_First
needed (_)first: @B => @A

type_proposition (@T)Has_Str_Rep
needed (_)to_string: @T => String

type_proposition (@T)Has_A_Wrapper
needed wrap(_): T1 => @T(T1)

type_proposition (@T)Has_Internal_App
needed apply(_)inside(_) : (T1 => T2) x @T(T1) => @T(T2)
```

Τα παραπάνω παραδείγματα ορίζουν τις παρακάτω ad hoc πολυμορφικές συναρτήσεις οι οποίες έχουν τους αντίστοιχους τύπους με προϋπόθεση:

```
(_)first: (@A)Is(@B)s_First --> @B => @A

(_)to_string: (@T)Has_Str_Rep --> @T => String

wrap(_): (@T)Has_A_Wrapper --> T1 => @T(T1)

apply(_)inside(_): (@T)Has_Internal_App --> (T1 => T2) x @T(T1) => @T(T2)
```

- *Περιγραφή*

  Ένας ορισμός ατομικής πρότασης ορίζει ταυτόχρονα την ίδια την **ατομική πρόταση** και την αντίστοιχη **πολυμορφική τιμή** ορίζοντας την μορφή του τύπου της τιμής δεδομένων των παραμέτρων τύπων της πρότασης. Η πρόταση είναι αληθής ή ψευδής όταν οι παράμετροι τύπων αντικατασταθούν από συγκεκριμένους τύπους ορίσματα ανάλογα με το αν έχει οριστεί η τιμή για αυτούς τους τύπους ορίσματα. Η τιμή αληθείας της πρότασης ορίζει αν η πολυμορφική τιμή έχει χρησιμοποιηθεί σωστά στο πρόγραμμα και επομένως αν θα περάσει ή όχι τον ελεγκτή τύπων. Για να προστεθούν περισσότεροι συνδυασμοί τύπων ορισμάτων για τους οποίους λειτουργεί η τιμή, ήτοι να οριστεί η τιμή για αυτούς τους τύπους, ήτοι να ισχύει η πρόταση για αυτούς τους τύπους, πρέπει να αποδειχτεί το αντίστοιχο θεώρημα. Τα θεωρήματα περιγράφονται σε μεγαλύτερο βάθος στην επόμενη ενότητα. Για τώρα, ως δούμε όσα αναφέρθηκαν σε αυτή την παράγραφο για την πρόταση ”`(@A)Is(@B)s_First`”:

  – Ορισμός πρότασης:

    ```
    type_proposition (@A)Is(@B)s_First
    needed (_)first: @B => @A
    ```

  – Τιμή (συνάρτηση) που ορίστηκε και ο τύπος της:

    ```
    (_)first: (@A)Is(@B)s_First --> @B => @A
    ```

  – Θεωρήματα για συγκεκριμένους τύπους:

    ```
    type_theorem (T1)Is(T1 x T2)s_First
    proof (_)first = _.1st

    type_theorem (T1)Is(ListOf(T1)s)s_First
    proof
      (_)first =
        cases
          [] => throw_err(”Tried to take the first element of an empty list”)
          [head, ...] => head
    ```

  – Χρήση της συνάρτησης:

    ```
    pair, list
      : Int x String, ListOf(String)s
      = (42, ”The answer to everything”), [”Hi!”, ”Hello”, Heeey”]

    >> (pair)first
      : Int
      ==> 42
    >> (list)first
      : String
      ==> ”Hi!”
    ```

  Ένας ορισμός ατομικής πρότασης ξεκινάει με την λέξη κλειδί ”`type_proposition`” την οποία ακολουθεί το όνομα της πρότασης (συμπεριλαμβανομένων και των παραμέτρων τύπων) στην πρώτη γραμμή. Η δεύτερη γραμμή ξεκινάει με την λέξη κλειδί ”`needed`” την οποία ακολουθεί το όνομα της πολυμορφικής τιμής και η έκφραση του τύπου της, χωρισμένοι από το σύμβολο ”έχει τύπο” (’`:`’).

Προτάσεις Μετονομασίας

- *Παραδείγματα*

```
type_proposition (@T)Has_Equality
equivalent (@T)And(@T)Can_Be_Equal

type_proposition (@A)And(@B)Are_Comparable
equivalent
  (@A)Can_Be_Less_Than(@B), (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_proposition (@T)Has_Comparison
equivalent (@T)And(@T)Are_Comparable
```

- *Περιγραφή*

  Οι ορισμοί προτάσεων μετονομασίας χρησιμοποιούνται για τη σύντμηση μίας πρότασης ή την σύζευξη πολλών προτάσεων σε μία νέα πρόταση.

  Ένας ορισμός πρότασης μετονομασίας ξεκινάει με την λέξη κλειδί ”`type_proposition`” την οποία ακολουθεί το όνομα της πρότασης (συμπεριλαμβανομένων και των παραμέτρων τύπων) στην πρώτη γραμμή. Η δεύτερη γραμμή ξεκινάει με την λέξη κλειδί ”`equivalent`” την οποία ακολουθεί μία πρόταση ή πολλές προτάσεις χωρισμένες με κόμματα (σε περίπτωση σύζευξης).

5.2.2 Θεωρήματα

Τα θεωρήματα χωρίζονται σε θεωρήματα **ατομικών προτάσεων** και θεωρήματα **προτάσεων συνεπαγωγής** τα οποία περιγράφονται στις παραγράφους που ακολουθούν.

Ατομικές Προτάσεις

- *Παραδείγματα*

```
type_theorem (Possibly(_))Has_A_Wrapper
proof wrap(_) = the_value:_

type_theorem (ListOf(_)s)Has_A_Wrapper
proof wrap(_) = [_]

type_theorem (Possibly(_))Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      no_value => no_value
      the_value:x => the_value:f(x)

type_theorem (ListOf(_)s)Has_Internal_App
proof apply(_)inside(_) = apply(_)to_all_in(_)
```

- *Χρήση*

```
a, b : all Possibly(Int)
  = wrapper(1), no_value

l1, l2, l3 : all ListOf(Int)s
  = wrapper(1), [], [1, 2, 3]

>> a
  : Possibly(Int)
  ==> the_value:1
>> b
  : Possibly(Int)
  ==> no_value
>> l1
  : ListOf(Int)s
  ==> [1]
>> l2
  : ListOf(Int)s
  ==> []
```

```
>> apply(_ + 1)inside(a)
  : Possibly(Int)
  ==> the_value:2
>> apply(_ + 1)inside(b)
  : Possibly(Int)
  ==> no_value
>> apply(_ + 1)inside(l1)
  : ListOf(Int)s
  ==> [2]
>> apply(_ + 1)inside(l2)
  : ListOf(Int)s
  ==> []
>> apply(_ + 1)inside(l3)
  : ListOf(Int)s
  ==> [2, 3, 4]
```

- *Περιγραφή*

  Ένα θεώρημα ατομικής πρότασης αποδεικνύει την πρόταση για συγκεκριμένους τύπους ορίσματα, υλοποιώντας την τιμή που αντιστοιχεί στην πρόταση για αυτούς τους τύπους. Επομένως, η τιμή που αντιστοιχεί στην πρόταση μπορεί να χρησιμοποιηθεί για όλους τους τύπους ορίσματα για τους όποιους ισχύει η πρόταση, δηλαδή τους τύπους ορίσματα για τους οποίους έχει υλοποιηθεί.

  Μια απόδειξη ενός θεωρήματος ατομικής πρότασης είναι ορθή αν η υλοποίηση της τιμής που αντιστοιχεί στην πρόταση έχει τύπο που έχει την μορφή που δόθηκε στον ορισμό της πρότασης αντικαθιστώντας τις παραμέτρους τύπων με τους τύπους ορίσματα του θεωρήματος.

  Ένα θεώρημα ατομικής πρότασης ξεκινάει με την λέξη κλειδί "`type_theorem`" η οποία ακολουθείται από το όνομα της πρότασης με τις παραμέτρους τύπων αντικατεστημένες από τους τύπους ορίσματα για τους οποίους θα αποδειχθεί η πρόταση. Η δεύτερη γραμμή είναι η λέξη κλειδί "`proof`". Η τρίτη γραμμή είναι στοιχισμένη δύο κενά πιο μέσα και είναι η γραμμή στην οποία ξεκινάει η υλοποίηση. Η υλοποίηση ξεκινάει με το όνομα της τιμής που αντιστοιχεί στην πρόταση και ακολουθείται από ίσον και την έκφραση που υλοποιεί την τιμή.

Προτάσεις Συνεπαγωγής

- *Παραδείγματα*

```
type_theorem (@A)And(@B)Can_Be_Equal --> (@A)And(@B)Can_Be_Unequal
proof a \= b = not(a == b)

type_theorem (@A)Can_Be_Greater_Than(@B) --> (@A)Can_Be_Le_Or_Eq_To(@B)
proof a <= b = not(a > b)

type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@A)And(@B)Have_Eq_And_Gr --> (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
```

- *Περιγραφή*

  Ένα θεώρημα πρότασης συνεπαγωγής είναι παρόμοιο με ένα θεώρημα ατομικής πρότασης με την διαφορά ότι χρησιμοποιούνται ad hoc πολυμορφικές τιμές και στην υλοποίηση. Επομένως, η υλοποίηση δεν αποδεικνύει ότι η πρόταση που αντιστοιχεί στην τιμή ισχύει πάντα για τους τύπους ορίσματα αλλά μόνο όταν ισχύουν και οι προτάσεις που αντιστοιχούν στις πολυμορφικές τιμές που χρησιμοποιούνται στην υλοποίηση για τους αντίστοιχους τύπους. Επομένως, έχουμε συνεπαγωγή, η οποία συμβολίζεται με το βέλος συνεπαγωγής (” `-->` ”). Η υπόθεση της συνεπαγωγής είναι μία πρόταση μετονομασίας που ισοδυναμεί με την σύζευξη όλων των προτάσεων που αντιστοιχούν στις ad hoc τιμές που χρησιμοποιούνται ή αν χρησιμοποιείται μόνο μία ad hoc τιμή τότε είναι η αντίστοιχη πρόταση. Το συμπέρασμα είναι η πρόταση που αντιστοιχεί στην τιμή που υλοποιείται.

Κεφάλαιο 6

Υλοποίηση Συντακτικού Αναλυτή

6.1 Πλήρης Γραμματική και Σύστημα Στοίχισης

Η πλήρης γραμματική βρίσκεται στο αντίστοιχο κεφάλαιο του αγγλικού κειμένου 7.1.1.

Σύστημα Στοίχισης Το μη-τερματικό σύμβολο <*indent*> στην γραμματική δεν είναι κανονικό BNF μη-τερματικό σύμβολο, είναι ένα σύμβολο που εξαρτάται απ΄τα συμφραζόμενα και ακολουθεί τους κανόνες στοίχισης της lcases. Εξαρτάται από μια τιμή που ονομάζεται ”επίπεδο στοίχισης” (il). Το <*indent*> αντιστοιχεί σε $2 * il$ κενά. Το επίπεδο στοίχισης ακολουθεί τους παρακάτω κανόνες:

1. Στην αρχή: $il = 0$
2. Στους ορισμούς τιμής:
   (a) Στο τέλος της πρώτης γραμμής: $il \leftarrow il + 1$
   (b) Στο τέλος της γραμμής με το ίσον: $il \leftarrow il + 1$
   (c) Στο τέλος του ορισμού: $il \leftarrow il - 2$
3. Στους ορισμούς μια ομάδας τιμών:
   (a) Στο τέλος της πρώτης γραμμής: $il \leftarrow il + 1$
   (b) Στο τέλος των ορισμών: $il \leftarrow il - 1$
4. Στις περίπτωσεις μίας έκφρασης συνάρτησης ”cases”:
   (a) Μετά την γραμμή με το βέλος συνάρτησης (”=>”): $il \leftarrow il + 1$.
   (b) Στο τέλος της περίπτωσης: $il \leftarrow il - 1$
5. Στα θεωρήματα τύπων:
   (a) Μετά την γραμμή με το ίσον: $il \leftarrow il + 2$.
   (b) Στο τέλος του θεωρήματος: $il \leftarrow il - 2$.
6. Στις εκφράσεις συναρτήσεων ”cases” που δεν ξεκινούν στην γραμμή με το ίσον ενός ορισμού τιμής:
   (a) Μετά την γραμμή των παραμέτρων: $il \leftarrow il + 1$.
   (b) Στο τέλος την έκφρασης ”cases”: $il \leftarrow il - 1$.

6.2 Δομή Υψηλού Επιπέδου

6.2.1 Βιβλιοθήκη Parsec

Ο συντακτικός αναλυτής υλοποιήθηκε χρησιμοποιώντας την βιβλιοθήκη **parsec** [2]. Η Parsec είναι μία βιομηχανικής δύναμης βιβλιοθήκη από monadic parser combinators. Μπορεί να αναλύσει γραμματικές με συμφραζόμενα και άπειρο look-ahead. Τα καταφέρνει αυτά χρησιμοποιώντας ένα πολυμορφικό τύπο συντακτικής ανάλυσης με τις παρακάτω παραμέτρους τύπων:

- *stream type:* Ο τύπος εισόδου του αναλυτή.
- *user state type:* Τύπος κατάστασης που επιθυμεί ο προγραμματιστής.
- *underlying monad type:* Επιπλέον εσωτερικός τύπος Monad αν χρειάζεται.
- *return type:* Ο τύπος του συντακτικού δέντρου του αναλυτή.

Η περιγραφή του package στο hackage βρίσκεται στο εξής url:
`https://hackage.haskell.org/package/parsec`

Στον Συγκεκριμένο Αναλυτή

- *stream type:* String
- *state type:*
  ParserState: Ορίζεται στον συγκεκριμένο αναλυτή και ακολουθεί παράγραφος που τον περιγράφει.
- *underlying monad:*
  Identity: Δηλαδή δεν χρησιμοποιείται κάποιος ενδιαφέρον τύπος.
- *return type:*
  Αυτός ο τύπος είναι διαφορετικός σε κάθε επιμέρους αναλυτή και είναι ο τύπος του αντίστοιχου υποδέντρου του συνολικού συντακτικού δέντρου.

Τύπος κατάστασης του αναλυτή: ParserState

Ο κώδικας που τον ορίζει:

```
type IndentationLevel = Int
type InEqualLine = Bool
type ParserState = (IndentationLevel, InEqualLine)
```

Χρειαζόμαστε αυτό τον τύπο κατάστασης για να υλοποιήσουμε του κανόνες στοίχισης.

6.2.2 Δομή Αρχείων

Ο κώδικας του αναλυτή χωρίζεται στα παρακάτω αρχεία:

- `ASTTypes.hs`: Ορισμοί των τύπων του συντακτικού δέντρου
- `ShowInstances.hs`: Αναπαράσταση του κάθε τύπου του συντακτικού δέντρου σε String για εκτύπωση.
- `Parsers.hs`: Επιμέρους αναλυτές για κάθε επιμέρους τύπο του συντακτικού δέντρου αλλά και για τον τύπο του συνολικού συντακτικού δέντρου.

Τα παραπάνω γράφτηκαν χρησιμοποιώντας την πλήρη γραμματική. Οι τύποι αντιστοιχούν σε μητερματικά σύμβολα και οι αναλυτές προσπαθούν να αναλύσουν την είσοδο τους και να την μετατρέψουν σε τιμή του αντίστοιχου τύπου του συντακτικού δέντρου.

6.3 Παραδείγματα Συντακτικών Αναλυτών

Σ'αυτή την ενότητα αναλύεται ο τρόπος με τον οποίο οι τύποι και οι αναλυτές γράφτηκαν με βάση την γραμματική, με μερικά παραδείγματα. Ξεκινάμε με έναν κανόνα γραμματικής, μετά φτιάχνουμε τον αντίστοιχο τύπο του συντακτικού δέντρου και μετά τον αναλυτή που αναλύει με βάση τον κανόνα.

6.3.1 Κλάση Parser και Παράδειγμα 0: Σταθερές

Έχουμε τον τύπο Parser ο οποίος είναι πολυμορφικός ως προς τον τύπο του συντακτικού δέντρου στον οποίο αναλύει με τύπο εισόδου String και τύπο κατάστασης ParserState:

```
type Parser = Parsec String ParserState
```

Φτιάχνουμε μία πολυμορφική τιμή "parser" με την κλάση τύπων "HasParser" έτσι ώστε όλοι οι αναλυτές να έχουν το όνομα "parser" ανεξαρτήτως του τύπου του συντακτικού δέντρου στον οποίο αντιστοιχούν:

```
class HasParser a where
  parser :: Parser a
```

Ξεκινάμε με το απλό παράδειγμα του literal με το παρακάτω κανόνα:

⟨*literal*⟩ ::= ⟨*int-lit*⟩ | ⟨*real-lit*⟩ | ⟨*char-lit*⟩ | ⟨*string-lit*⟩

Ο τύπος του συντακτικού δέντρου για το literal είναι:

```
data Literal =
  Int Integer | R Double | Ch Char | S String
```

Και παρακάτω είναι ο συντακτικός αναλυτής του literal ο οποίος είναι ορισμένος ως instance της κλάσης HasParser. Εσωτερικά χρησιμοποιούμε του αναλυτές για κάθε συγκεκριμένο literal οι οποίοι είναι ορισμένοι ξεχωριστά:

```
instance HasParser Literal where
  parser =
    R <$> try parser <|> Int <$> parser <|> Ch <$> parser <|> S <$> parser <?>
    "Literal"
```

Ο τύπος Parser είναι Functor, έτσι ο τελεστής "<$>" (fmap) περνάει εσωτερικά τον constructor σε κάθε συγκεκριμένο αναλυτή. Ο "try" είναι parser combinator που κάτι backtracking αν οι αναλυτής αποτύχει. Ο τελεστής "<|>" σημαίνει "αυτός ο αναλυτής ή ο άλλος αναλυτής". Τέλος, ο τελεστής "<?>" σημαίνει "αν αποτύχουν όλοι οι αναλυτές τότε δείξε αυτό στο μήνυμα σφάλματος".

6.3.2 Παράδειγμα 1: Λίστες

Ο κανόνας γραμματικής για την λίστα είναι:

⟨*list*⟩ ::= ‘`[`’ [‘ ’] [⟨*line-expr-or-unders*⟩] [‘ ’] ‘`]`’

Ο τύπος του συντακτικού δέντρου για την λίστα είναι:

```
newtype List = L (Maybe LineExprOrUnders)
```

Και ο αναλυτής για την λίστα είναι:

```
instance HasParser List where
  parser =
    L <$> (char '[' *> opt_space_around (optionMaybe parser) <* char ']')
```

Εδώ χρησιμοποιούμε τους τελεστές ”*>” και ”<*” οι οποίοι χρησιμοποιούν και τους δύο αναλυτές που έχουν ως τελεστέους (πρώτα τον αριστερό) αλλά επιστρέφουν μόνο αυτό που ανέλυσε ο αναλυτής στον οποίο ”δείχνουν”. Το ”opt_space_around” αναλύει προαιρετικά ένα κενό από κάθε μεριά αυτών που ανέλυσε ο αναλυτής όρισμα. Το ”optionMaybe” είναι ορισμένο στην βιβλιοθήκη. Χρησιμοποιεί τον αναλυτή όρισμα και αν αυτός πετύχει τότε γυρνάει Just <αυτό που ανέλυσε ο αναλυτής όρισμα>, αλλιώς γυρνάει Nothing.

6.3.3 Παράδειγμα 2: Change

Ο κανόνας γραμματικής για την έκφραση ”change” είναι:

⟨*dot-change*⟩ ::= ‘`.change{`’ [‘ ’] ⟨*field-change*⟩ (⟨*comma*⟩ ⟨*field-change*⟩)* [‘ ’] ‘`}`’

Ο τύπος του συντακτικού δέντρου για την έκφραση ”change” είναι:

```
newtype DotChange = DC (FieldChange, [FieldChange])
```

Και ο αναλυτής για την έκφραση ”change” είναι:

```
instance HasParser DotChange where
  parser =
    DC <$>
      (try (string ".change{") *> opt_space_around field_changes_p <* char '}')
    where
    field_changes_p :: Parser (FieldChange, [FieldChange])
    field_changes_p = field_change_p +++ many (comma *> field_change_p)
```

Εδώ χρησιμοποιούμε τον τελεστή ”+++” ο οποίος χρησιμοποιεί τους αναλυτές τελεστεούς τους (πρώτα τον αριστερό) και βάζει τα αποτελέσματά τους σε μία πλειάδα. Ο ”field_change_p” αναλύει μία αλλαγή στοιχείου. Ο ”many” είναι ορισμένος στην βιβλιοθήκη και αναλύει ότι αναλύει ο αναλυτής όρισμα όσες φορές είναι εφικτό και βάζει τα αποτελέσματα σε μία λίστα (το άστρο Kleene των αναλυτών).

6.3.4 Παράδειγμα 3: Ορισμοί Τιμών

Ο κανόνας γραμματικής για τον ορισμό τιμής είναι:

⟨*value-def*⟩ ::=
⟨*indent*⟩ ⟨*identifier*⟩ ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’) ⟨*type*⟩
⟨*nl*⟩ ⟨*indent*⟩ ‘= ’ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

Ο τύπος του συντακτικού δέντρου για τον ορισμό τιμής είναι:

```
newtype ValueDef = VD (Identifier, Type, ValueExpr, Maybe WhereExpr)
```

Και ο αναλυτής για τον ορισμό τιμής είναι:

```
instance HasParser ValueDef where
  parser =
    indent *> parser >>= \identifier ->

    increase_il_by 1 >>

    has_type_symbol *> parser >>= \type_ ->
    nl_indent *> string "= " *>

    increase_il_by 1 >> we_are_in_equal_line >>

    parser >>= \value_expr ->

    we_are_not_in_equal_line >>

    optionMaybe (try parser) >>= \maybe_where_expr ->

    decrease_il_by 2 >>

    return (VD (identifier, type_, value_expr, maybe_where_expr))
```

Σε αυτό το παράδειγμα βλέπουμε το πως η κατάσταση του αναλυτή χρησιμοποιείται για να εφαρμοστούν οι κανόνες του συστήματος στοίχισης. Ο αναλυτής "indent" αναλύει <2 * επίπεδο στοίχισης> κενά διαβάζοντας το επίπεδο στοίχισης από την κατάσταση. Οι "`increase_il_by`" και "`decrease_il_by`" έχουν τύπο Parser αλλά δεν αναλύουν τίποτα, είναι "αναλυτές" οι οποίοι χρημοποιούνται μόνο για να αλλάξουν την κατάσταση και να συνδυαστούν με άλλους αναλυτές. Χρησιμοποιούνται όπως ορίζεται στον κανόνα 2 του Συστήματος Στοίχισης (6.1). Ο "`has_type_symbol`" αναλύει με το εξής κομμάτι του κανόνα: ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’). Οι "`we_are_in_equal_line`" και "`we_are_not_in_equal_line`" αλλάζουν την κατάσταση για να εφαρμόσουν τον κανόνα 6 του Συστήματος Στοίχισης.

Κεφάλαιο 7

Μετάφραση σε Haskell

7.1 Περιγραφή Υψηλού Επιπέδου

Για να μην χρειάζεται να ξαναγραφτεί όλο το σύστημα τύπων της Haskell, η lcases μεταφράζεται κατευθείαν σε Haskell και δεν γίνεται πλήρης σημασιολογική ανάλυση αλλά η ελάχιστη δυνατή ώστε να μπορεί να γίνει η μετάφραση. Οι φάσεις υψηλού επιπέδου για την μετάφραση είναι:

- **Συλλογή**

  Σε αυτήν την φάση περνάμε από το συντακτικό δέντρο και συλλέγουμε τα παρακάτω:

  - Όλα τα ονόματα ”γυμνών περιπτώσεων” όλων των `or_types`, όπου μια ”γυμνή περίπτωση” είναι μία περίπτωση που δεν έχει εσωτερική τιμή (π.χ. `no_value` σε αντίθεση με `the_value:_`)
  - Όλα τα ονόματα των fields όλων των `tuple_types`.
  - Όλες οι προτάσεις μετονομασίας.

  Τα συλλεγμένα χρησιμοποιούνται στην φάση προεπεξεργασίας.

- **Προεπεξεργασία**

  Σε αυτή την φάση το συντακτικό δέντρο που έχει δημιουργηθεί από τον συντακτικό αναλυτή μετατρέπεται σε ένα λίγο διαφορετικό ακολουθόντας του παρακάτω κανόνες:

  - Αν ένα όνομα γυμνής περίπτωσης βρεθεί σε μία έκφραση τιμής, του προσθέτουμε μπροστά ένα ’C’ (από το constructor). Αυτό χρειάζεται γιατί οι περιπτώσεις των `or_types` μετατρέπονται σε Haskell data constructors, οι οποίοι πρέπει να ξεκινούν με κεφαλαίο γράμμα ενώ οι περιπτώσεις `or_type` στην lcases ξεκινούν με μικρό.
  - Αν ένα όνομα ενός field βρεθεί μέσα σε μια έκφραση τιμής η οποία είναι υποέκφραση μίας έκφραφασης ”`.change`”, το όνομα μετατρέπεται σε μία συνάρτηση επιθήματος η οποία έχει το ίδιο όρισμα με την συνάρτηση ”`.change`”. Για παράδειγμα, αν ένα υποδέντρο του συντακτικού δέντρου αντιστοιχεί στην παρακάτω έκφραση, θα μετατρεπεί ως εξής:
    `x.change{f1 = f1 + 1}` $\Longrightarrow$ `x.change{f1 = x.f1 + 1}`

    Το ίδιο ισχύει και για τα ειδικά ονόματα:
    `x.change{1st = 1st + 1}` $\Longrightarrow$ `x.change{1st = x.1st + 1}`

  - Αν μία πρόταση μετονομασίας εμφανίζεται σε ένα θεώρημα συνεπαγωγής πριν το βέλος, αντικαθίσταται από την σύζευξη στην οποία αντιστοιχεί. Όλες οι αντικαταστάσεις που έχουν γίνει στις μεταβλητές τύπων της, μεταφέρονται και στις μεταβλητές τύπων της σύζευξης. Για παράδειγμα, αν τα παρακάτω εμφανίζονται στο πρόγραμμα:

```
type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@C)And(@D)Have_Eq_And_Gr --> (@C)Can_Be_Gr_Or_Eq_To(@D)
proof a >= b = a == b | a > b
```

το θεώρημα μετατρέπεται ως εξής:

```
type_theorem
  [(@C)And(@D)Can_Be_Equal, (@C)Can_Be_Greater_Than(@D)] -->
  (@C)Can_Be_Gr_Or_Eq_To(@D)
proof a >= b = a == b | a > b
```

στο προεπεξεργασμένο δέντρο.

- **Μετάφραση**

  Σε αυτή την φάση το προεπεξεργασμένο δέντρο μεταφράζεται κατευθείαν σε Haskell. Οι λεπτομέρειες αυτής της διαδικασίας περιγράφονται στις επόμενες ενότητες.

7.2 Φάση Μετάφρασης: Γενικά

Για την φάση μετάφρασης υπάρχει η υλοποίηση μίας εκ των παρακάτω τριών πολυμορφικών συναρτήσεων για κάθε τύπο του συντακτικού δέντρου:

- `to_haskell`:

  Αυτή η συνάρτηση είναι για τους τύπους που δεν χρειάζονται κάποιου είδους κατάσταση και μπορούν να μεταφραστούν κατευθείαν σε Haskell.

- `to_hs_wpn`: (`to_haskell` with parameter number)

  Αυτή η συνάρτηση είναι για τους τύπους που προσθέτουν καινούργιες παραμέτρους κατά την μετάφραση και άρα χρειάζονται μία κατάσταση με μετρητή παραμέτρων.

- `to_hs_wil`: (`to_haskell` with indentation level)

  Αυτή η συνάρτηση είναι για τους τύπους που χρειάζονται πληροφορία για την στοίχιση για να μεταφραστούν σωστά και άρα χρειάζονται μία κατάσταση που περιέχει το επίπεδο στοίχισης.

Παρακάτω έχουμε τα classes που ορίζουν τις παραπάνω συναρτήσεις:

- ```
  type Haskell = String

  class ToHaskell a where
    to_haskell :: a -> Haskell
  ```

- ```
  type WithParamNum = State Int

  class ToHsWithParamNum a where
    to_hs_wpn :: a -> WithParamNum Haskell
  ```

- ```
  type WithIndentLvl = State Int

  class ToHsWithIndentLvl a where
    to_hs_wil :: a -> WithIndentLvl Haskell
  ```

Στις επόμενες ενότητες μπαίνουμε σε περισσότερη λεπτομέρεια για συγκεκριμένους τύπους.
  ```

7.3 Φάση Μετάφρασης: Βασικές Εκφράσεις

7.3.1 Σταθερές και Ονόματα

- **Σταθερές**

  Οι σταθερές παραμένουν όπως είναι μετά την συντακτική ανάλυση εκτός από τις σταθερές αριθμών, στις οποίες προστίθεται επισημείωση τύπου όταν βρίσκονται σε μία έκφραση τιμής (δεν προστίθεται αν είναι περίπτωση συνάρτησης ”cases”).

- **Ονόματα**

  – *Παραδείγματα*

  `x1` $\Longrightarrow$ `x1`
  `apply(_)to_all_in(_)` $\Longrightarrow$ `apply'to_all_in'`
  `(_)to_string` $\Longrightarrow$ `a'to_string`
  `f(_, _, _)` $\Longrightarrow$ `f'''`

  – *Περιγραφή*

  Για τα ονόματα, όλες οι παρανθέσεις με κάτω παύλες αντικαθιστώνται από αποστρόφους ίσου πλήθους με τις παύλες. Αν υπάρχει παρένθεση στην αρχή του ονόματος, ένα ’a’ τοποθετείται πριν τις αντίστοιχες αποστρόφους για να είναι σωστό όνομα Haskell.

7.3.2 Παρενθέσεις, Πλειάδες και Λίστες

- **Παρένθεση**

  Οι μετάφραση μια έκφρασης που είναι σε παρένθεση μπαίνει επίσης σε παρένθεση.

- **Πλειάδες**

  – *Παραδείγματα*

  `(x, y)` $\Longrightarrow$ `ft2(x, y)`
  `(_, 3.14, _)` $\Longrightarrow$ `(\(pA0, pA1) -> ft3(pA0, 3.14, pA1))`

  ```
  (_, _, ”Hello from 3rd field”)
  ⟹
  (\(pA0, pA1) -> ft3(pA0, pA1, ”Hello from 3rd field”))
  ```

  – *Περιγραφή*

  Συνάρτηση ft*n*

  Στην μετάφραση, κάθε πλειάδα δίνεται ως όρισμα στην συνάρτηση **ft*n*** όπου ***n*** είναι το μέγεθος της πλειάδας. Η **ft*n*** είναι μια πολυμορφική συνάρτηση που ορίζεται από το παρακάτω class (για ***n*** = 2 και αντίστοιχα classes για 3,4 και 5):

  ```
  class FromTuple2 a b c | c -> a b where
    ft2 :: (a, b) -> c
  ```

Αυτό γίνεται διότι η ίδια πλειάδα μπορεί να έχει τύπο γινόμενο ή τύπο `tuple_type` ανάλογα με το που εμφανίζεται στο πρόγραμμα. Για τύπους γινόμενα έχουμε το παρακάτω instance:

```
instance FromTuple2 a b (a, b) where
  ft2 = id
```

Και ο μεταφραστής φτιάχνει αυτόματα ένα καινούργιο instance για κάθε ορισμό `tuple_type`. Για παράδειγμα, για τον παρακάτω ορισμό:

```
tuple_type Name
value (first_name, last_name) : String^2
```

μετά την μετάφραση του ίδιου του ορισμού θα έχουμε το παρακάτω instance:

```
instance FromTuple2 String String Name where
  ft2 = \(x1, x2) -> Name' x1 x2
```

Με αυτό το μηχανισμό το πρόγραμμα περνάει τον ελεγκτή τύπων και στις δύο περιπτώσεις.

Παράμετροι για κάτω παύλες

Για όλα τα στοιχεία μίας πλειάδας που περιέχουν κάτω παύλα, δημιουργείται μία νέα παράμετρος στην θέση της (”pA<*n - 1*>” για την n-οστή κάτω παύλα). Αφού τελειώσει η μετάφραση της πλειάδας προσθέτουμε στην αρχή της όλες τις παραμέτρους που δημιουργήθηκαν για να την κάνουμε έκφραση συνάρτησης. Το ”pA” είναι από το ”parameter” και το ’A’ είναι κεφαλαίο για να αποφευχθούν πιθανές συγκρούσεις με άλλα ονόματα του προγράμματος (στην lcases δεν έχουμε κεφαλαία στα ονόματα).

- **Λίστες**

  – *Παραδείγματα*

  ```
  [1.61, 2.71, 3.14]
  ⟹
  [(1.61 :: Double), (2.71 :: Double), (3.14 :: Double)]

  [_, x, _] ⟹ (\(pA0, pA1) -> [pA0, x, pA1])
  ```

  – *Περιγραφή*

  Οι λίστες λειτουργούν με τον ίδιο τρόπο που λειτουργούν οι πλειάδες με την διαφορά ότι δεν χρειάζονται την συνάρτηση `ft`*n*.

7.3.3 Εφαρμογή Συνάρτησης με Παρενθέσεις

- *Παραδείγματα*

  `f(x, y, z)` $\Longrightarrow$ `f'''(x, y, z)`
  `f(x, _, _)` $\Longrightarrow$ `(\(pA0, pA1) -> f'''(x, pA0, pA1))`
  `(x)to_str` $\Longrightarrow$ `a'to_str(x)`
  `apply(f)to_all_in(_)` $\Longrightarrow$ `(\pA0 -> apply'to_all_in'(f, pA0))`

- *Περιγραφή*

  Για την εφαρμογή συνάρτησης με παρενθέσεις, διαχωρίζουμε το όνομα της συνάρτησης από το ορίσματά της αντικαθιστώντας κάθε παρένθεση με αποστρόφους ίδιου πλήθους με τα ορίσματα της παρένθεσης (και βάζοντας ένα 'a' μπροστά αν υπάρχει παρένθεση στην αρχή). Παράλληλα, συλλέγουμε όλα τα ορίσματα σε μία πλειάδα την οποία μετά βάζουμε σαν όρισμα στο τέλος του ονόματος της συνάρτησης. Αν στην θέση κάποιου ορίσματος έχουμε κάτω παύλα, κάνουμε το ίδιο που κάνουμε για τις κάτω παύλες σε πλειάδες ώστε να δημιουργηθεί μία νέα παράμετρος.

7.3.4 Συναρτήσεις Προθήματος και Επιθήματος

- **Συναρτήσεις Προθήματος**
  - *Παραδείγματα*

    `the_value:42` $\Longrightarrow$ `Cthe_value((42 :: Integer))`
    `error:"this is an error message"` $\Longrightarrow$ `Cerror("this is an error message")`
    `the_value:_` $\Longrightarrow$ `(\pA0 -> Cthe_value(pA0))`
    `the_value:result:true` $\Longrightarrow$ `Cthe_value(Cresult(True))`

  - *Περιγραφή*

    Οι συναρτήσεις προθήματος είναι data constructors στην Haskell, οι οποίοι έχουν εισαχθεί από την μετάφραση του αντίστοιχου ορισμού `or_type`. Στην μετάφραση τους προσθέτουμε μπροστά ένα 'C' (από το "constructor") ώστε να είναι σωστοί Haskell constructors. Το όρισμα τοποθετείται σε παρένθεση η οποία τοποθετείται μετά την μετάφραση της συνάρτησης προθήματος (στην Haskell δεν έχουμε άνω κάτω τελεία). Αν το όρισμα είναι κάτω παύλα τότε στην παρένθεση τοποθετούμε μία νέα παράμετρο αντίστοιχα με το τρόπο που γίνεται αυτό στις πλειάδες και η συνολική μετάφραση μπαίνει σε παρένθεση για να κόψουμε την εμβέλεια της νέας παραμέτρου.

- **Συναρτήσεις Επιθήματος**
  - *Παραδείγματα*

    `date.year` $\Longrightarrow$ `year(date)`
    `tuple.1st` $\Longrightarrow$ `p1st(tuple)`
    `info.date.year` $\Longrightarrow$ `year(date(info))`
    `tuple.1st.2nd` $\Longrightarrow$ `p2nd(p1st(tuple))`

– *Περιγραφή*

Οι συναρτήσεις επιθήματος δημιουργούνται αυτόματα από την μετάφραση του αντίστοιχου ορισμού `tuple_type` ή είναι οι ειδίκες συναρτήσεις επιθήματος τύπων γινομένων (`_.1st _.2nd` κτλ). Μεταφράζονται σε κανονικές Haskell συναρτήσεις (όπως είναι οι συναρτήσεις προβολής στην Haskell) με το όρισμά τους σε παρένθεση. Για τις ειδικές συναρτήσεις τύπων γινομένων προσθέτουμε στην αρχή ένα 'p' (από το "projection") ώστε να είναι σωστό όνομα Haskell.

Οι συναρτήσεις των τύπων γινομένων είναι πολυμορφικές και λειτουργούν σε πλειάδες οποιουδήποτε μήκους (για τώρα ≤ 5). Αυτό το καταφέρνουμε με τον ορισμό του παρακάτω class (για το p1st και αντίστοιχων classes για τα υπόλοιπα):

```
class IsFirst' a b | b -> a where
  p1st :: b -> a
```

Και των παρακάτω instances:

```
instance IsFirst' a (a, b) where
  p1st = fst

instance IsFirst' a (a, b, c) where
  p1st = \(a, _, _) -> a

instance IsFirst' a (a, b, c, d) where
  p1st = \(a, _, _, _) -> a

...
```

- **Συνάρτηση ".change"**

  – *Παραδείγματα*

  `state.change{counter = counter + 1}`
  $\xrightarrow{\text{preprocessing}}$ `state.change{counter = state.counter + 1}`
  $\Longrightarrow$ `c0counter(counter(state) !+ (1 :: Integer)) state`

  `tuple.change{1st = 1.61, 3rd = 3.14}` $\Longrightarrow$ `(c1st(1.61) .> c3rd(3.14)) tuple`

  `tuple.change{x = _, y = _}` $\Longrightarrow$ `(\(pA0, pA1) -> (c0x(pA0) .> c0y(pA1)) tuple)`

  – *Περιγραφή*

  Για την συνάρτηση ".change" δημιουργείται μία συνάρτηση αλλαγής για κάθε field κάθενος εκ των `tuple_type` τύπων, αντίστοιχα με τις συναρτήσεις προβολής. Αυτή η συνάρτηση έχει όνομα το όνομα του field όπου έχει προστεθεί μπροστά το string "c0". Το 'c' είνα από το "change" και το '0' το βάζουμε για την αποφυγή συγκρούσεων με άλλα ονόματα (στην lcases τα ψηφία μπορούν να μπουν μόνο στο τέλος ενός ονόματος). Για τα fields τύπων γινομένων οι συναρτήσεις αλλαγής είναι "`c1st`", "`c2nd`", κτλ. Ο τύπος κάθε συνάρτησης αλλαγής έχει την μορφή:

  ```
  FieldType -> TupleOrProdType -> TupleOrProdType
  ```

Για κάθε ανάθεση χρησιμοποιούμε την συνάρτηση αλλαγής του αντίστοιχου field με την έκφραση της ανάθεσης ως όρισμα τύπου `FieldType`. Με αυτό τον τρόπο έχουμε όλες τις συναρτήσεις αλλαγής με δοσμένο το πρώτο όρισμα να έχουν πλέον τύπο τις μορφής:

```
TupleOrProdType -> TupleOrProdType
```

Τις συνθέτουμε όλες με τον τελεστή σύνθεσης "`.>`" και έχουμε την συνολική συνάρτηση όλων των αλλαγών, τύπου:

```
TupleOrProdType -> TupleOrProdType
```

Τελειώνουμε εφαρμόζοντας την στην μετάφραση του ορίσματος της "`.change`".

Οι συναρτήσεις αλλαγής για τύπους γινόμενα λειτουργούν σε πλειάδες κάθε μεγέθους (για τώρα ≤ 5). Αυτό λειτουργεί με παρόμοια classes και instances με αυτά των `p1st`, `p2nd`, κτλ. Δυστυχώς λόγω αυτού δεν μπορούμε να χρησιμοποιήσουμε το συμβολισμό της Haskell για αυτές τις αλλαγές γιατί όπως λέει το σφάλμα της Haskell αυτές οι πολυμορφικές συναρτήσεις δεν είναι "record selectors".

Αν υπάρχει κάτω παύλα κάπου, δημιουργείται νέα μεταβλητή παρόμοια με τον τρόπο που γίνεται αυτό στις πλειάδες.

7.4 Φάση Μετάφρασης: Τελεστές

Τελεστές Για καθένα από τους τελεστές της lcases, έχει οριστεί ένας νέος Haskell τελεστής στον οποίο μεταφράζεται. Αυτό επιτρέπει την χρήση του μηχανισμού προτεραιότητας και προσεταιριστικότητας της Haskell για νέους τελεστές όρισμένους από τον χρήστη. Έτσι, δεν χρειάστηκε να υλοποιηθούν στον συντακτικό αναλυτή, κάτι το οποίο θα ήταν αρκετά πιο χρονοβόρο και θα χρειαζόντουσαν αρκετά περισσότερες παρενθέσεις στην μετάφραση. Οι τελεστές εφαρμογής συνάρτησης και σύνθεσης συναρτήσεων ορίζονται όπως ορίζονται παρακάτω. Κάθε άλλος τελεστής ορίζεται με ένα type class. Η υλοποίηση του τελεστή για κάθε συνδιασμό τύπων ορίζεται με ένα αντίστοιχο instance. Τα παρακάτω παραδείγματα για τον τελεστή της πρόσθεσης δείχνουν την γενική μορφή αυτών των classes και instances.

lcases τελεστές	Haskell τελεστές
->	&>
<-	<&
o>	.>
<o	<.
^	!^
*	!*
/	!/
+	!+
-	!-
==	!==
!=	!!=
>	!>
<	!<
>=	!>=
<=	!<=
&	!&
|	!|
;>	!>>=
;	!>>

Προτεραιότητα και προσεταιριστικότητα με Haskell:

```
infixl 9 &>
infixr 8 <&
infixl 7 .>, <.
infixr 6 !^
infixl 5 !*, !/
infixl 4 !+, !-
infix 3 !==, !!=, !>, !<, !>=, !<=
infixr 2 !&
infixr 1 !|
infixr 0 !>>=, !>>
```

Τελεστές εφαρμογής συνάρτησης και σύνθεσης συναρτήσεων:

```
(&>) :: a -> (a -> b) -> b
x &> f = f x

(<&) :: (a -> b) -> a -> b
f <& x = f x

(.>) :: (a -> b) -> (b -> c) -> a -> c
(.>) = flip (.)

(<.) :: (b -> c) -> (a -> b) -> a -> c
(<.) = (.)
```

```
# Type class για πρόσθεση:
class A'And'Add_To' a b c where
  (!+) :: a -> b -> c

# Μερικά από τα instances

# Πρόσθεση δύο λιστών:
instance b ~ [a] => A'And'Add_To' [a] [a] b where
  (!+) = (++)

# Πρόσθεση ενός τύπου 'a' με μία λίστα από 'a'
instance b ~ [a] => A'And'Add_To' a [a] b where
  (!+) = (:)

# Πρόσθεση ενός String με ένα τύπο 'a' που έχει Show instance
instance (Show a, b ~ String) => A'And'Add_To' String a b where
  str !+ x = str ++ show x
```

Εκφράσεις Τελεστών

- *Παραδείγματα*

  `5 * 'a'` $\Longrightarrow$ `(5 :: Integer) !* 'a'`

  `"Hello " + "World!"` $\Longrightarrow$ `"Hello " !+ "World!"`

  `_ - 1` $\Longrightarrow$ `(\pA0 -> pA0 !- (1 :: Integer))`

  `_ + "string in the middle of the arguments" + _`
  $\Longrightarrow$ `(\(pA0, pA1) -> pA0 !+ "string in the middle of the arguments" !+ pA1)`

- *Περιγραφή*

  Στις εκφράσεις τελεστών, οι τελεστέοι μεταφράζονται αντίστοιχα με το τι τελεστέοι είναι και οι τελεστές αντικαθιστώνται με τους αντίστοιχους προαναφερόμενους τελεστές. Αν στην θέση κάποιου τελεστέου υπάρχει κάτω παύλα, μία νέα παράμετρος δημιουργείται παρόμοια με τον τρόπο που γίνεται αυτό στις πλειάδες.

7.5 Φάση Μετάφρασης: Εκφράσεις Συναρτήσεων

Κανονικές Εκφράσεις Συναρτήσεων

- *Παραδείγματα*

```
x => 17 * x + 42 ⟹ \x -> (17 :: Integer) !* x !+ (42 :: Integer)

* => 42 ⟹ \_ -> (42 :: Integer)

(x, *, z) => x + z ⟹ \(x, _, z) -> x !+ z

((x1, y1), (x2, y2)) => (x1 + x2, y1 + y2)
⟹ \((x1, y1), (x2, y2)) -> ft2(x1 !+ x2, y1 !+ y2)
```

- *Περιγραφή*

  Για την μετάφραση των παραμέτρων γίνονται τα παρακάτω:

  – Τοποθετείται μπροστά ο χαρακτήρας `'\'`
  – Το βέλος `"=>"` αντικαθίσταται από το βέλος `"->"`
  – Οι παράμετροι με αστερίσκο μετατρέπονται σε παραμέτρους με κάτω παύλα

  Το σώμα της συνάρτησης μεταφράζεται ανάλογα με το τι έκφραση είναι.

  Είναι δυνατόν η έκφραση συνάρτησης να ακολουθείται από μία έκφραση ”`where`” όπως φαίνεται στο παρακάτω παράδειγμα:

```
gac => print(message)
where
message: String
  = "Gcd: " + gac.gcd + "\nCoefficients: a = " + gac.a + ", b = " + gac.b
```

  Σε αυτή την περίπτωση, η έκφραση ”where” μετατρέπεται σε μια έκφραση ”`let-in`” όπως περιγράφεται στην αντίστοιχη ενότητα. Αυτή η έκφραση ”`let-in`” τοποθετείται ανάμεσα στις παραμέτρους και στο σώμα της συνάρτησης, ώστε οι παράμετροι να μπορούν να χρησιμοποιηθούν στους ορισμούς της ”`let-in`” έκφρασης όπως φαίνεται παρακάτω:

```
\gac ->
let
message :: String
message =
  "Gcd: " !+ gcd(gac) !+ "\nCoefficients: a = " !+ a(gac) !+ ", b = " !+ b(gac)
in
print'(message)
```

  όπου η παράμετρος ”`gac`” χρησιμοποιείται στο ”`message`”.

Εκφράσεις Συναρτήσεων ”cases”

- *Παραδείγματα*

```
cases
  true => print(”It’s true!! :)”)
  false => print(”It’s false... :(”)

⟹

\pA0 ->
case pA0 of
  True -> print’(”It’s true!! :)”)
  False -> print’(”It’s false... :(”)
```

```
(x, cases)
  0 => x
  y => gcd(y, (x)mod(y))

⟹

\(x, pA0) ->
case pA0 of
  0 -> x
  y -> gcd’’(y, a’mod’(x, y))
```

```
(cases, cases)
  (green, green) => true
  (amber, amber) => true
  (red, red) => true
  ... => false

⟹

\(pA0, pA1) ->
case (pA0, pA1) of
  (green, green) -> True
  (amber, amber) -> True
  (red, red) -> True
  _ -> False
```

```
cases
  [x1, x2, xs = ...] =>
    (x1 < x2) & (x2 + xs)is_sorted
  ... => true

⟹

\pA0 ->
case pA0 of
  x1 : x2 : xs ->
    (x1 !< x2) !& a’is_sorted(x2 !+ xs)
  _ -> True
```

- *Περιγραφή*

Οι παράμετροι μεταφράζονται παρόμοια με τον τρόπο που μεταφράζονται στις κανονικές εφκράσεις συναρτήσεων με την διαφορά ότι όλες οι παράμετροι που περιέχουν την λέξη κλειδί ”`cases`” μεταφράζονται σε νέες παραμέτρους. Κάθε τέτοια νέα παράμετρος συλλέγεται σε μία πλειάδα στην οποία κάνουμε pattern matching με την εξής γραμμή:
”`case <πλειάδα νέων παραμέτρων> of`”.
Για την τελυταία περίπτωση, αν έχουμε ”. . .” αυτό μεταφράζεται σε ”_”. Όταν κάνουμε pattern matching στα πρώτα στοιχεία μίας λίστα η μετάφραση γίνεται ως εξής:

```
[x1, x2, ...] => <σώμα περίπτωσης>
⟹
x1 : x2 : _ -> <μετάφραση σώματος περίπτωσης>

[x1, x2, xs = ...] => <σώμα περίπτωσης>
⟹
x1 : x2 : xs -> <μετάφραση σώματος περίπτωσης>
```

όπου αφαιρούνται οι αγκύλες και τα κόμματα γίνονται άνω κάτω τελείες. Αν το υπόλοιπο της λίστας έχει όνομα τότε αυτό είναι το μόνο που παραμένει μετά την τελευταία άνω κάτω τελεία, αλλιώς έχουμε κάτω παύλα.

7.6 Φάση Μετάφρασης: Ορισμοί Τιμών και Εκφράσεις ”where”

- *Παραδείγματα*

```
foo: Int
  = 42

⟹

foo :: Integer
foo =
  (42 :: Integer)
```

```
dfs_on_tree(_) : (T1)Tree => (Int x T1)Tree
  = dfs_on_tree(_)with_num(1) o> _.tree
    where
    dfs_on_tree(_)with_num(_) : (T1)Tree x Int => (T1)ResultTreeAndNum
      = <irrelevant stuff>

    <irrelevant stuff>

dfs_on_tree' :: forall a1. A'Tree a1 -> A'Tree (Integer, a1)
dfs_on_tree' =
  let
  dfs_on_tree'with_num' :: (A'Tree a1, Integer) -> A'ResultTreeAndNum a1
  dfs_on_tree'with_num' = <irrelevant stuff>

  <irrelevant stuff>
  in
  (\pA0 -> dfs_on_tree'with_num'(pA0, (1 :: Integer))) .> (\x' -> tree(x'))
```

```
val1, val2, val3 : Int, Bool, Char
  = 42, true, 'a'

val1 :: Integer
val1 =
  (42 :: Integer)

val2 :: Bool
val2 =
  True

val3 :: Char
val3 =
  'a'
```

```
print_gcd_and_coeffs_of(_): GcdAndCoeffs => IO
  = gac => print(message)
    where
    message: String
      = ″Gcd: ″ + gac.gcd_ + ″\nCoefficients: a = ″ + gac.a + ″, b = ″ + gac.b
```

$\Longrightarrow$

```
print_gcd_and_coeffs_of′ :: GcdAndCoeffs -> IO
print_gcd_and_coeffs_of′ =
  \gac ->
  let
  message :: String
  message =
    ″Gcd: ″ !+ gcd_(gac) !+ ″\nCoefficients: a = ″ !+ a(gac) !+ ″, b = ″ !+ b(gac)
  in
  print′(message)
```

Περιγραφή

Για την μετάφραση των ορισμών τιμών γίνονται τα εξής:

- Το σύμβολο ”έχει τύπο” μεταφράζεται βάζοντας μία δεύτερη άνω κάτω τελεία.
- Το όνομα της τιμής επαναχρησιμοποιείται πριν το ίσον.
- Αν ο ορισμός τιμής είναι στο επίπεδο στοίχισης 0 τότε γίνονται τα παρακάτω:
  - Συλλέγουμε όλες τις παραμετρικές μεταβλητές τύπων του τύπου.
  - Προσθέτουμε το παρακάτω στην αρχή της μετάφρασης του τύπου:
    `″forall ″ <παραμ. μετάβ. τύπου μεταφρασμένες και χωρισμένες με κενά> ′.′`

  Αυτό επιτρέπει την χρήση των ίδιων μεταβλητών τύπων και στους τύπους μέσα στην έκφραση ”`where`” αν υπάρχει. Αυτό φείνεται στο 2ο παράδειγμα όπου ο `T1` (πριν την μετάφραση ή `a1` μετά) χρησιμοποιείται και στην επισημείωση τύπου του `dfs_on_tree(_)with_num(_)`. Αν δεν είχαν γίνει τα παραπάνω τότε τα ”`a1`” δεν θα αναφερόντουσαν στην ίδια μεταβλητή τύπου παρόλο που έχουν το ίδιο όνομα. Για να λειτουργήσουν τα παραπάνω χρειάζεται επίσης το compiler extension `ScopedTypeVariables`.
- Αν η έκφραση της τιμής ακολουθείται από μία έκφραση ”`where`”, η έκφραση ”`where`” μεταφράζεται σε μία έκφραση ”`let-in`” η οποία τοποθετείται πάνω από την μετάφραση της έκφρασης τιμής (2ο παράδειγμα). Η μόνη εξαίρεση σ΄αυτόν τον κανόνα είναι όταν η έκφραση τιμής είναι έκφραση κανονικής συνάρτησης όπου η έκφραση ”`let-in`” τοποθετείται μεταξύ των παραμέτρων και του σώματος της συνάρτησης (τελευταίο παράδειγμα).

7.7 Φάση Μετάφρασης: Τύποι

7.7.1 Εκφράσεις Τύπων

Ονόματα Τύπων

- *Παραδείγματα*

  `Int` $\Longrightarrow$ `Integer`

  `Real` $\Longrightarrow$ `Double`

  `String` $\Longrightarrow$ `String`

  `SelfReferencingType` $\Longrightarrow$ `SelfReferencingType`

- *Περιγραφή*

  Τα ονόματα τύπων παραμένουν τα ίδια κατά την μετάφραση εκτός των Int και Real, τα οποία μεταφράζονται σε Integer και Double αντίστοιχα.

Μεταβλητές Τύπων

- Παραμετρικές Μεταβλητές Τύπων
  - *Παραδείγματα*

    `T1` $\Longrightarrow$ `a1`

    `T2` $\Longrightarrow$ `a2`

    `T3` $\Longrightarrow$ `a3`

  - *Περιγραφή*

    Το 'T' γίνεται 'a'.

- Ad Hoc Μεταβλητές Τύπων
  - *Παραδείγματα*

    `@A` $\Longrightarrow$ `b0`

    `@B` $\Longrightarrow$ `b1`

    `@C` $\Longrightarrow$ `b2`

  - *Περιγραφή*

    Το '@' γίνεται 'b' και τα κεφαλαία αγγλικά γράμματα μεταφράζονται ως εξής:
    A $\Longrightarrow$ 0, B $\Longrightarrow$ 1, κτλ

Τύποι Εφαρμογής Τύπου

- *Παραδείγματα*

  `ListOf(Int)s` $\Longrightarrow$ `ListOf's Integer`

  `Error(String)OrResult(Int)` $\Longrightarrow$ `Error'OrResult' String Integer`

  `(Int)Tree` $\Longrightarrow$ `A'Tree Integer`

  `ListOf(Int => Int)s` $\Longrightarrow$ `ListOf's (Integer -> Integer)`

  `Before(B,C)After` $\Longrightarrow$ `Before''After B C`

  `A(B(C))` $\Longrightarrow$ `A' (B' C)`

- *Περιγραφή*

  Για τους τύπους εφαρμογής τύπου, το Haskell όνομα απομονώνεται αντικαθιστώντας κάθε παρένθεση με αποστρόφους ίσου πλήθους με τα ορίσματα της παρένθεσης. Αν υπάρχει παρένθεση στην αρχή τότε προστίθεται ένα 'A' (από το "argument") πριν από το όνομα για να είναι σωστό όνομα Haskell. Οι τύποι ορίσματα συλλέγονται, μεταφράζονται και για όποιον χρειάζεται τοποθετείται σε παρένθεση και όλοι μαζί τοποθετούνται μετά το όνομα χωρισμένοι από κενά.

Τύποι Γινόμενα

- *Παραδείγματα*

  `Int x Real x String` $\Longrightarrow$ `(Integer, Double, String)`

  `Int^2 x Int^2` $\Longrightarrow$ `((Integer, Integer), (Integer, Integer))`

  `(A^2 => A) x A x ListOf(A)s` $\Longrightarrow$ `((A, A) -> A, A, ListOf's A)`

- *Περιγραφή*

  Για τους τύπους γινόμενα μεταφράζονται όλοι οι τύποι των fields, χωρίζονται από κόμματα και μπαίνουν σε παρένθεση.

Τύποι Συναρτήσεων

- *Παραδείγματα*

  `T1 => T1` $\Longrightarrow$ `a1 -> a1`

  `Int^2 => Int` $\Longrightarrow$ `(Integer, Integer) -> Integer`

  `(A^2 => A) x A x ListOf(A)s => A` $\Longrightarrow$ `((A, A) -> A, A, ListOf's A) -> A`

- *Περιγραφή*

  Για τους τύπους συναρτήσεων μεταφράζονται οι τύποι εισόδου και εξόδου και το βέλος μεταξύ τους μετατρέπεται από "=>" σε "->".

Τύποι Με Προϋπόθεση

- *Παραδείγματα*

  `(@T)Has_Str_Rep --> @T => String` $\Longrightarrow$ `A'Has_Str_Rep b19 => b19 -> String`

  `(@A)Is(@B)s_First --> @B => @A` $\Longrightarrow$ `A'Is's_First b0 b1 => b1 -> b0`

  `(@T)Has_Internal_App --> (T1 => T2) x @T(T1) => @T(T2)`
  $\Longrightarrow$
  `A'Has_Internal_App b19 => (a1 -> a2, b19 a1) -> b19 a2`

- *Περιγραφή*

  Για τους τύπους με προϋπόθεση, η προϋπόθεση μεταφράζεται παρόμοια με την μετάφραση των τύπων εφαρμογής τύπου, όπου οι παρενθέσεις αντικαθιστώνται από αποστρόφους και οι μεταβλητές τοποθετούνται μετά το όνομα χωρισμένες από κενά. Ο τύπος μετά την προϋπόθεση μεταφράζεται ανάλογα με το τι τύπος είναι και το βέλος μεταξύ της προϋπόθεσης και του τύπου μετατρέπεται από "-->" σε "=>".

7.7.2 Ορισμοί Τύπων

Ορισμοί `tuple_type`

- *Παραδείγματα*

```
tuple_type Date
value (day, month, year) : Int^3

⟹

data Date =
  Date' { day :: Integer, month :: Integer, year :: Integer }

instance FromTuple3 Integer Integer Integer Date where
  ft3 = \(x1, x2, x3) -> Date' x1 x2 x3

c0day :: Integer -> Date -> Date
c0month :: Integer -> Date -> Date
c0year :: Integer -> Date -> Date
c0day = \new x -> x { day = new }
c0month = \new x -> x { month = new }
c0year = \new x -> x { year = new }
```

```
tuple_type Edge
value (u, v) : Node^2

⟹

data Edge =
  Edge' { u :: Node, v :: Node }

instance FromTuple2 Node Node Edge where
  ft2 = \(x1, x2) -> Edge' x1 x2

c0u :: Node -> Edge -> Edge
c0v :: Node -> Edge -> Edge
c0u = \new x -> x { u = new }
c0v = \new x -> x { v = new }
```

```
tuple_type (T1)Tree
value (root, subtrees) : T1 x (T1)Trees
```

$\Longrightarrow$

```
data A'Tree a1 =
  A'Tree' { root :: a1, subtrees :: A'Trees a1 }

instance FromTuple2 a1 (A'Trees a1) (A'Tree a1) where
  ft2 = \(x1, x2) -> A'Tree' x1 x2

c0root :: a1 -> A'Tree a1 -> A'Tree a1
c0subtrees :: A'Trees a1 -> A'Tree a1 -> A'Tree a1
c0root = \new x -> x { root = new }
c0subtrees = \new x -> x { subtrees = new }
```

- *Περιγραφή*

Για τους ορισμούς `tuple_type` η μετάφραση γίνεται με τα παρακάτω βήματα:

1. "`tuple_type`" $\Longrightarrow$ "`data`"
2. Από το όνομα του τύπου διαχωρίζουμε το όνομα της συνάρτησης τύπων αντικαθιστώντας τις παρενθέσεις με αποστρόφους ίσου πλήθους με τις μεταβλητές τύπων που υπάρχουν στην κάθε παρένθεση. Αν υπήρχε παρένθεση στην αρχή βάζουμε ένα 'A' πριν από τις πρώτες αποστρόφους. Ύστερα, προσθέτουμε τις μεταφράσεις των μεταβλητών χωρισμένες από κενά και στο τέλος της γραμμής βάζουμε ίσον.
3. Οι λέξη κλειδί "`value`" δεν υπάρχει στην μετάφραση και η δεύτερη γραμμή στοιχίζεται πιο μέσα και ξεκινάει με τον data constructor, ο οποίος είναι ίδιος με την συνάρτηση τύπων του ονόματος με μία επιπλέον απόστροφο στο τέλος.
4. Προσθέτουμε το record syntax της Haskell βάζοντας το κάθε field με την επισημείωση του τύπου του, χωρισμένα με κόμματα και μέσα σε άγκιστρα. Μ'αυτό κλείνει ο ορισμός του ίδιου του τύπου στην Haskell.
5. Σε αυτό το βήμα ορίζουμε το `ft`*n* instance του τύπου που ορίσαμε, όπου *n* είναι το μήκος των πλειάδων του. Ο ορισμός αυτός χρειάζεται για την μετάφραση των πλειάδων για λόγους που περιγράφονται στην αντίστοιχη ενότητα και χωρίζεται στα παρακάτω βήματα:
   (a) "`instance FromTuple`*n* "
   (b) Προσθέτουμε τις μεταφράσεις των field types (σε παρένθεση όποια χρειάζεται) και την μετάφραση του ονόματος του `tuple_type` (με αυτή τη σειρά) χωρισμένες από κενά. Ακολουθεί το string " `where`".
   (c) Η δεύτερη γραμμή στοιχίζεται πιο μέσα και ξεκινάει με "`ft`*n* `=`".
   (d) Συνεχίζει με ένα '`\`', την πλειάδα των παραμέτρων `x1` έως `x`*n* και το βέλος " `->` ".
   (e) Στο σώμα της συνάρτησης έχουμε τον data constructor του βήματος 3 ακολουθούμενο από τις παραμέτρους `x1` έως `x`*n* χωρισμένες από κενά.
6. Σ'αυτό το βήμα επισημειώνουμε τις συναρτήσεις αλλαγής κάθενος field με τους τύπους τους ως εξής:
   – Το όνομα της συνάρτησης αλλαγής είναι "`c0`" ακολουθούμενο από το όνομα του field.
   – Προσθέτουμε το σύμβολο "έχει τύπο" (" `::` ") ακολουθούμενο από τον τύπο, ο οποίος έχει την παρακάτω μορφή:

```
<μετάφραση field type> -> <όνομα tuple_type> -> <όνομα tuple_type>
```

7. Σ'αυτό το βήμα δίνουμε τους ορισμούς των συναρήσεων αλλαγής για κάθε field ως εξής:
   - Το όνομα της συνάρτησης αλλαγής είναι "`c0`" ακολουθούμενο από το όνομα του field.
   - Ακουθεί το string "`= \new x -> x { <όνομα του field> = new }`"

Ορισμοί `or_type`

- *Παραδείγματα*

```
or_type Bool
values true | false

⟹

data Bool =
  Ctrue |
  Cfalse
```

```
or_type Possibly(T1)
values the_value:T1 | no_value

⟹

data Possibly' a1 =
  Cthe_value a1 |
  Cno_value
```

```
or_type Error(T1)OrResult(T2)
values error:T1 | result:T2

⟹

data Error'OrResult' a1 a2 =
  Cerror a1 |
  Cresult a2
```

```
or_type Comparison
values lesser | equal | greater

⟹

data Comparison =
  Clesser |
  Cequal |
  Cgreater
```

- *Περιγραφή*

Για τους ορισμούς `or_type` η μετάφραση γίνεται με τα παρακάτω βήματα:

1. "`tuple_type`" $\Longrightarrow$ "`data`"
2. Το όνομα του τύπου μεταφράζεται με τον ίδιο τρόπο που μεταφράζεται στους ορισμούς `tuple_type` και ακολουθείται από ίσον.
3. Η λέξη κλειδί "`values`" δεν υπάρχει στην μετάφραση και από την δεύτερη γραμμή και μετά έχουμε τους data constructors για κάθε μία από τις περιπτώσεις του `or_type`, έναν σε κάθε γραμμή. Για την κάθε περίπτωση γίνονται τα παρακάτω:
   (a) Βάζουμε ένα 'C' πριν το όνομα της περίπτωσης για να την κάνουμε σωστό data constructor.
   (b) Αν υπάρχει εσωτερική τιμή, ο τύπος μεταφράζεται και τοποθετείται μετά τον data constructor (διαχωρίζονται με κενό).
   (c) Αν δεν είναι η τελευταία περίπτωση στο τέλος της γραμμής έχουμε "` |`".

Παρατσούκλια Τύπων

- *Παραδείγματα*

  `type_nickname Ints = ListOf(Int)s` $\Longrightarrow$ `type Ints = ListOf's Integer`

  `type_nickname ErrOrRes(T1) = Error(String)OrResult(T1)`
  $\Longrightarrow$ `type ErrOrRes' a1 = Error'OrResult' String a1`

- *Περιγραφή*

  Για τα παρατσούκλια τύπων αντικαθιστούμε το "`type_nickname`" με "`type`", το όνομα του παρατσουκλιού με την μετάφραση του και τον τύπο στον οποίο αναφέρεται με την δικιά του μετάφραση.

7.8 Φάση Μετάφρασης: Λογική Τύπων

Ορισμοί Προτάσεων Οι ορισμοί προτάσεων μετονονομασίας δεν έχουν ξεχωριστή μετάφραση. Χρησιμοποίουνται μόνο στην φάση προεπεξεργασίας όπου αλλάζουν τα θεωρήματα στα οποία εμφανίζονται οι ορισμένες προτάσεις. Για τους ορισμούς ατομικών προτάσεων έχουμε τα παρακάτω:

- *Παραδείγματα*

```
type_proposition (@A)Is(@B)s_First
needed (_)first : @B => @A

⟹

class A'Is's_First b0 b1 where
  a'first :: b1 -> b0
```

```
type_proposition (@T)Has_Internal_App
needed
  apply(_)inside(_)
    : (T1 => T2) x @T(T1) => @T(T2)

⟹

class A'Has_Internal_App b19 where
  apply'inside'
    :: (a1 -> a2, b19 a1) -> b19 a2
```

```
type_proposition (@T)Has_A_Wrapper
needed wrap(_) : T1 => @T(T1)

⟹

class A'Has_A_Wrapper b19 where
  wrap' :: a1 -> b19 a1
```

```
type_proposition (@T)Has_String_Repr
needed (_)to_string : @T => String

⟹

class A'Has_String_Repr b19 where
  a'to_string :: b19 -> String
```

```
type_proposition (@A, @B)To(@C)
needed ab_to_c: @A x @B => @C#

⟹

class A''To' b0 b1 b2 where
  ab_to_c :: (b0, b1) -> b2
```

- *Περιγραφή*

  Οι ορισμοί ατομικών προτάσεων μεταφράζονται ως εξής:

  - ”`type_proposition`” $\Longrightarrow$ ”`class`”.
  - Το όνομα της πρότασης μεταφράζεται αντίστοιχα με το όνομα του τύπου σε έναν ορισμό τύπου. Κάθε παρένθεση αντικαθίσταται από αποστρόφους ίσου πλήθους με της μεταβλητές που βρίσκονται στην παρένθεση και αν υπήρχε παρένθεση στην αρχή τότε προσθέτουμε ένα 'A' πριν από τις αντίστοιχες αποστρόφους για να είναι σωστό όνομα class της Haskell. Ύστερα, τοποθετούμε τις μεταφράσεις των ad hoc μεταβλητών τύπων χωρισμένες από κενά. Τέλος, κλείνουμε την πρώτη γραμμή με ” `where`”.
  - Η λέξη κλειδί ”`needed`” δεν εμφανίζεται στην μετάφραση. Η δεύτερη γραμμή είναι στοιχισμένη πιο μέσα και ξεκινάει με το όνομα της πολυμορφικής τιμής που ορίζουμε ακολουθούμενο από ” `::` ” και την μετάφραση του τύπου της.

Θεωρήματα

- *Παραδείγματα*

```
type_theorem
  (Possibly(_))Has_A_Wrapper
proof wrap(_) = the_value:_

⟹

instance
  A'Has_A_Wrapper Possibly' where
  wrap' = (\pA0 -> Cthe_value(pA0))
```

```
type_theorem
  (ListOf(_)s)Has_A_Wrapper
proof wrap(_) = [_]

⟹

instance
  A'Has_A_Wrapper ListOf's where
  wrap' = (\pA0 -> [pA0])
```

```
type_theorem (Possibly(_))Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      no_value => no_value
      the_value:x => the_value:f(x)

⟹

instance A'Has_Internal_App Possibly' where
  apply'inside' =
    \(f', pA0) ->
    case pA0 of
      Cno_value -> Cno_value
      Cthe_value x -> Cthe_value(f'(x))
```

```
type_theorem (ListOf(_)s)Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      [] => []
      [head, tail = ...] => f(head) + apply(f(_))inside(tail)

⟹

instance A'Has_Internal_App ListOf's where
  apply'inside' =
    \(f', pA0) ->
    case pA0 of
      [] -> []
      head : tail -> f'(head) !+ apply'inside'((\pA0 -> f'(pA0)), tail)
```

```
type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@A)And(@B)Have_Eq_And_Gr --> (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
```

προεπεξεργασία ⟹

```
type_theorem
  [(@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)] -->
  (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
```

μετάφραση ⟹

```
instance
  (A'And'Can_Be_Equal b0 b1, A'Can_Be_Greater_Than' b0 b1) =>
  A'Can_Be_Gr_Or_Eq_To' b0 b1
  where
  a !>= b = a !== b !| a !> b
```

- *Περιγραφή*

  Η μετάφραση των θεωρημάτων γίνεται με τα παρακάτω βήματα:

  - ”`type_theorem`” $\Longrightarrow$ ”`instance`”.
  - Μεταφράζονται οι προτάσεις και το βέλος μετατρέπεται από ”`->`” σε ”`=>`”. Οι προτάσεις μεταφράζονται αντίστοιχα με τα ονόματα προτάσεων στους ορισμούς ατομικών προτάσεων με την διαφορά ότι αντί για ad hoc μεταβλητές τύπων έχουμε τύπους ορίσματα. Αν μία πρόταση πριν από βέλος αντιστοιχεί σε πρόταση μετονομασίας, τότε στο βήμα προεπεξεργασίας αυτή έχει αντικατασταθεί από την αντίστοιχη πρόταση ή σύζευξη όπως φαίνεται στο τελευταίο παράδειγμα. Τα παραπάνω κλείνουν με ” `where`”.
  - Η λέξη κλειδί ”`proof`” δεν υπάρχει υπάρχει στην μετάφραση. Η επόμενη γραμμή είναι στοιχισμένη πιο μέσα όπου έχουμε την μετάφραση τις τιμής ή του τελεστή με τους τελεστέους παραμέτρους, ακολουθούμενους από ίσον και την μετάφραση της έκφρασης τιμής με την οποία ισούνται.

Κεφάλαιο 8

Συμπεράσματα

Υλοποιήθηκε ένας μεταφραστής από lambda-cases σε Haskell, ο οποίος είναι επίσης γραμμένος σε Haskell. Οι βασικές διαφορές μεταξύ της lambda-cases και της Haskell είναι:

- Η εφαρμογή συνάρτησης γίνεται με παρενθέσεις αντί για κενά.
  Παράδειγμα: ”`f(x,y,z)`” αντί για ”`f x y z`”
- Οι συναρτήσεις μπορούν να οριστούν ώστε να δέχονται ορίσματα πριν ή στην μέση του ονόματός τους.
  Παράδειγμα: ”`apply(f)to_all_in(list)`” αντί για ”`map f list`”
- Μερική εφαρμογή συνάρτησης μπορεί να γίνει για οποιαδήποτε από τα ορίσματα.

  Παραδείγματα:
  ”`f(x, y, _)`” αντί για ”`f x y`” για την συνηθισμένη χρήση μίας συνάρτησης που περιμένει τρία ορίσματα στην Haskell και τις δίνουμε τα δύο πρώτα.
  Επιπλέον, είναι εφικτά τα παρακάτω:

  - ”`f(x, _, z)`” αντί για ”`\y -> f x y z`”
  - ”`f(_, y, z)`” αντί για ”`\x -> f x y z`”
  - ”`f(_, _, z)`” αντί για ”`\x y -> f x y z`”
  - κτλ

- Η παραπάνω σύνταξη όπου περιμένουμε τις κάτω παύλες σαν ορίσματα μπορεί να χρησιμοποιηθεί και για τελεστέους σε εκφράσεις τελεστών.

  Παραδείγματα:
  ”`_ + 1`” αντί για ”`(+ 1)`” για την μετατροπή του αριστερού τελεστέου σε παράμετρο.
  Επιπλέον, είναι εφικτά τα παρακάτω:

  - ”`”Hello ” + _ + ”! You look much younger than ” + _ + ”!”`”
    αντί για
    ”`\name age -> ”Hello ” ++ name ++ ”! You look much younger than ” + age + ”!”`”
  - ”`_^2 + _^2`” αντί για ”`\x y -> x^2 + y^2`”
  - κτλ

- Η παραπάνω σύνταξη όπου περιμένουμε τις κάτω παύλες σαν ορίσματα μπορεί να χρησιμοποιηθεί και για στοιχεία πλειάδων ή λιστών.

  Παραδείγματα:

  - ”`(17, _, 42)`” αντί για ”`\x -> (17, x, 42)`”
  - ”`[17, _, 42]`” αντί για ”`\x -> [17, x, 42]`”

- Δεν χρειάζονται ονόματα για να γίνει pattern matching σε παραμέτρους, η λέξη κλειδί ”cases” χρησιμοποιείται αντί αυτού.

  Παραδείγματα:

  –
  ```
  cases
    true => print(”It’s true!! :)”)
    false => print(”It’s false... :(”)
  ```
  αντί για
  ```
  \b -> case b of
    True -> putStrLn ”It’s true!! :)”
    False -> putStrLn ”It’s false... :(”
  ```
  Αυτό γίνεται και στην Haskell με το LambdaCase extension ως εξής:
  ```
  \case
    True -> putStrLn ”It’s true!! :)”
    False -> putStrLn ”It’s false... :(”
  ```
  Το όνομα της γλώσσας προέρχεται από την πολύ συχνή χρήση του LambdaCase. Αυτή η ιδέα επεκτείνεται σε οποιοδήποτε υποσύνολο των παραμέτρων όπως φαίνεται στο παρακάτω παράδειγμα (και στα παραδείγματα της ενότητας Εκφράσεις Συναρτήσεων ”cases” 4.3.2).
  –
  ```
  (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false
  ```
  αντί για
  ```
  \x y -> case (x, y) of
    (Green, Green) -> True
    (Amber, Amber) -> True
    (Red, Red) -> True
    _ -> False
  ```

- Στους τύπους μπορούν να χρησιμοποιηθούν δυνάμεις. Για παράδειγμα:

  ”`Int^2`” αντί για ”`Int x Int`” στην lcases ή ”`(Integer, Integer)`” στην Haskell.

- Οι τελεστές είναι γενικευμένοι.

  Παραδείγματα:

  – ”`”Hello ” + ”World!”`” αντί για ”`”Hello ” ++ ”World!”`”
  – ”`5 * ’a’`” αντί για ”`replicate 5 ’a’`”

Μελλοντική έρευνα

Πολύ λίγη σημασιολογική ανάλυση έχει γίνει και οποιοδήποτε σφάλμα δεν είναι συντακτικό θα απορριφθεί από τον μεταγλωττιστή της Haskell. Για έναν πιο ολοκληρωμένο μεταγλωττιστή/μεταφραστή, απαιτείται ένα βήμα σημασιολογικής ανάλυσης και τα μηνύματα σφάλματος θα πρέπει να αναφέρονται στο πρόγραμμα lambda-cases και όχι στη μετάφραση της Haskell. Αυτό δεν έχει υλοποιηθεί λόγω της πολυπλοκότητας του συστήματος τύπων της Haskell και της τεράστιας δουλειάς που θα απαιτούσε η επαναδιατύπωσή του. Ωστόσο, επειδή οι lambda-cases αναγκάζουν τον χρήστη να χρησιμοποιεί σχόλια τύπων, ίσως να είναι δυνατή η υλοποίηση ενός απλούστερου συστήματος τύπων που θα αρκεί για τις ανάγκες των lambda-cases ή/και η εμφάνιση μηνύματος σφάλματος εάν δεν είναι δυνατός ο έλεγχος τύπων, για να αναγκάσει τον χρήστη να εισαγάγει επισημειώσεις τύπων. Αυτά είναι οι σημαντικότερες κατευθύνσεις μελλοντικής έρευνας.

Κείμενο στα αγγλικά

Chapter 1

Introduction

Haskell is a delightful language [1]. Yet, it doesn't seem to have its rightful place in terms of popularity in industry. Why is it so? Is it inherently hard to learn or could it be that the syntax is perplexing to the amateur eye? It is my belief that with some syntax changes that give a greater familiarity to the new user, there would be no language more compelling than (the new) Haskell. In an attempt to achieve that familiarity, I present some (hopefully useful) new syntax, of which some is closer to the imperative/OOP style (to attract more already experienced programmers from these languages), some is closer to mathematics (in which most programmers should be experienced) and some is closer to natural language (in which we are all already experienced).

Haskell is by far my favourite language and it has been ever since I first started to grasp it. I remember learning it in the advanced programming languages course taught by the supervisor of my thesis, Mr. Papaspyrou. As I started to understand how all the constructs fit together in this beautifully concise manner, I also started to wonder, why would I ever use any of these other languages that I've learned? Why would I waste my time on writing code with languages that are so difficult to read and/or so difficult to debug? This feeling of letting the compiler guide me through all my mistakes and then... done! It just works!! (most of the time) How extraordinary! Suddenly, coding is a joyful puzzle of types, that must simply be put together in the correct way. On top of that, there's also a very helpful guide!

On the other hand, before all of that, there was a lot of confusion. What does it mean to be pure? What is this monad thing that seems to be so important? Why is this "do notation" not letting me do what I want to do? What are those things starting with a capital letter that look like functions? Why are there so many arrows in the types? As I started to slowly understand all those things and be very glad that I did, I saw friends dropping the class, "I don't understand and I'm never going to understand" they told me, "why do we need all those things anyway?". I could not persuade them not to drop the class. I had only just started to understand myself and I was not in the position to explain it clearly to others, it was more of a feeling of connecting puzzle pieces and being amazed by how easy it was to write the code.

As I got more and more convinced that I want to avoid writing code in other languages, I also wondered, why isn't this the mainstream way of writing code? Why is Haskell not even in the top five most popular languages when for me it's clearly number one? As an engineer and scientist, I must after all look at the facts and provide my best possible explanation. Of course, I still have no clue, but I'm going to try to guess based on my experiences. Firstly, the tooling might be a problem and certainly lots of people (myself included) have had problems because of it. This is however a hard problem and one beyond the scope of this thesis. Instead, I want to focus on another opinion that I slowly gravitated towards, which is that the semantics of the language is complete and will remain the superior semantics for centuries to come. In contrast, I don't think the same could be said about the syntax.

As much as we all love the origins, the lambda calculus, I think that the syntax of the lambda calculus has served and continues to serve its purpose in the theoretical domain and it is not to be forced upon the average software engineer that just wants to solve a problem. Function application in mathematics has had for centuries the clear syntax whereby the variables are in parenthesis and because of that, they are clearly separated visually from the function itself. It is not necessary to have

the lambda calculus syntax to be thinking in lambda calculus semantics.

In addition, even though the fact that we can do type inference is amazing, in practice, if type annotations are not used Haskell loses its beauty and becomes really hard to read, much like all the other languages (and also starts to have terrible error messages). So maybe we should force the user to use type annotations (except for literals and other obvious things), for his own benefit and for the benefit of everybody who's going to read the code in the future. This also gives the opportunity to combine the type annotation and the definition into one and avoid having to repeat the identifier for both.

One other difficult part of learning Haskell was understanding the keywords. "data", "type", "newtype", "class", "instance" are all not very descriptive. Maybe more descriptive and intuitive keywords could help the new user learn faster and remember how to use them with more ease.

Lastly, I found myself using the LambdaCase extension and the syntax that comes with it all the time. It was such a joy to be able to avoid giving identifiers to things even more than I was already avoiding it by using function composition and all the other amazing Haskell tools. I found that this syntax could be further developed for multiple parameters and it seemed to be very useful to me. This is also where the name of the language comes from.

The purpose of the thesis is to address all of the above (and a bit more) and combine them into a new language.

Chapter 2

Small Program Comparison: C, Haskell, lcases

In this chapter we compare implementations of a small program in C, Haskell and lcases. The program reads two integers from stdin (it throws an error if it can't) and returns a nice message with their gcd. We firstly compare the lcases implementation with an implementation in C and secondly with an implementation in Haskell. Of course, the implementations are written by me and therefore they are written in the style I use in these languages. However, I've tried to write them in a way which I perceive to be a relatively standard way to write them.

Comparison with C

```c
#include<stdio.h>

int my_gcd(int a, int b) {
  if (b == 0)
    return a;
  else
    return my_gcd(b, a%b);
}

int main(){

  int x, y;

  printf("Please give me 2 ints\n");
  if (scanf("%d %d", &x, &y) != 2) {
    fprintf(stderr, "You didn't give me 2 ints\n");
    return 1;
  }

  printf("The GCD of %d and %d is %d\n", x, y, my_gcd(x, y));
  return 0;
}
```

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => my_gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print("Please give me 2 ints");
    get_line ;> split(_)to_words o> cases
      [s1, s2] => (from_string(s1), from_string(s2)) -> (_)from_io
      ... => throw_err("You didn't give me 2 ints")

ints(_, _, _)to_message: Int^3 => String
  = (x, y, gcd) => "The GCD of " + x + " and " + y + " is " + gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    ints(i1, i2, my_gcd_of(i1)and(i2))to_message -> print(_)
```

An important difference between C and lcases is the fact that in lcases the identifier, the expression of its type and the expression that is assigned to it are all separated from each other. This is demonstrated by comparing the ”`my_gcd`” function in the C code with the ”`my_gcd_of(_)and(_)`” in lcases. Specifically comparing the line:

```
int my_gcd(int a, int b) {
```

with the line:

```
my_gcd_of(_)and(_): Int^2 => Int
```

we can see that the fact that the function receives two integers and returns an integer is written in C by giving the type **`int`** to each of the parameters and to the result of the function, while in lcases this is all written in a single type expression that does not have any identifiers in it.

Also in the same lcases line, we can observe that the identifier has a parenthesis in the middle which makes it expect the first argument in that spot. The function is then called recursively with arguments inside the parentheses in the line:

```
y => my_gcd_of(y)and((x)mod(y))
```

where we can also see that `(_)mod(_)` is defined expecting the first argument in the beginning. Similarly, `ints(_, _, _)to_message` expects all three arguments in the same parenthesis in the middle.

Another notable difference is the use of the ”cases” keyword in the lines:

```
= (x, cases)
  0 => x
  y => my_gcd_of(y)and((x)mod(y))
```

instead of the ”if-then” statement of the lines:

```
if (b == 0)
  return a;
else
  return my_gcd(b, a%b);
```

By using the ”cases” keyword in the spot of the second parameter we indicate that we are going to pattern match on it in the cases. That can be done for any subset of the parameters (see section 4.3.2).

Another difference worth mentioning is the use of the plus operator for string concatenation and also for automatically converting the integers to strings for them to be concatenated in the line:

```
= (x, y, gcd) => ”The GCD of ” + x + ” and ” + y + ” is ” + gcd
```

as opposed to using ”printf” with ”%d” in the line:

```
printf(”The GCD of %d and %d is %d\n”, x, y, my_gcd(x, y));
```

Finally, another difference is the fact the semicolon is actually an operator in lcases with environment actions as operands as described in section 4.2.3.

Comparison with Haskell

```haskell
my_gcd :: Int -> Int -> Int
my_gcd = \x y -> case y of
  0 -> x
  _ -> my_gcd y (mod x y)

read_two_ints :: IO (Int, Int)
read_two_ints = do
  putStrLn "Please give me 2 ints"
  s <- getLine
  case words s of
    [s1, s2] -> return (read s1, read s2)
    _ -> error "You didn't give me 2 ints"

ints_to_message :: Int -> Int -> Int -> String
ints_to_message = \x y gcd ->
  "The GCD of " ++ show x ++ " and " ++ show y ++ " is " ++ show gcd

main :: IO ()
main = do
  (i1, i2) <- read_two_ints
  putStrLn $ ints_to_message i1 i2 (my_gcd i1 i2)
```

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => my_gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print("Please give me 2 ints");
    get_line ;> split(_)to_words o> cases
      [s1, s2] => (from_string(s1), from_string(s2)) -> (_)from_io
      ... => throw_err("You didn't give me 2 ints")

ints(_, _, _)to_message: Int^3 => String
  = (x, y, gcd) => "The GCD of " + x + " and " + y + " is " + gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    ints(i1, i2, my_gcd_of(i1)and(i2))to_message -> print(_)
```

An important difference between Haskell and lcases is the fact that in lcases function application is done with parenthesis instead of spaces which also gives the opportunity to have arguments inside the function identifier as can be seen by comparing the line:

```
my_gcd :: Int -> Int -> Int
```

with the line:

```
my_gcd_of(_)and(_): Int^2 => Int
```

In the same lines we can also see that we use one input and one output (as in Haskell) but the input is the product type of all the argument types. This might seem restrictive but in fact it is more general as in lcases it is possible to provide any subset of the arguments (by putting an underscore in the rest) and the result is a function that expects the rest of the arguments (unless they were all provided), see section 4.1.3. In Haskell the arguments must be provided from left to right and we can omit the n rightmost arguments for the result to be a function that expects these n arguments. However, if for example we want to omit the first and provide the second (from left to right) we must use ”flip” first. For more complicated similar stuff we must define functions similar to ”flip” every time by hand.

Also in the same lines we can see the use of the power type `Int^2` which does not exist in Haskell.

By comparing the lines:

```
my_gcd :: Int -> Int -> Int
my_gcd = \x y -> case y of
```

with the lines:

```
my_gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
```

we see that in lcases the type annotation and the assignment are grouped into one to avoid using the identifier twice.

By comparing the lines:

```
= (x, cases)
  0 => x
  y => my_gcd_of(y)and((x)mod(y))
```

with the lines:

```
my_gcd = \x y -> case y of
  0 -> x
  _ -> my_gcd y (mod x y)
```

we see that in lcases we immediately use the ”cases” keyword in the parameter we want to pattern match on without giving it a name. The name is given (if it is necessary) on the default/last case (if there is one). The ”cases” keyword can be used on any subset of the parameters to pattern match on all of them at the same time (see section 4.3.2).

By comparing the lines:

```
= print(”Please give me 2 ints”);
  get_line ;> split(_)to_words o> cases
```

with the lines:

```
putStrLn ”Please give me 2 ints”
s <- getLine
case words s of
```

we see that in lcases we don’t use do notation, instead we use the environment operators ”;” and ”;>” which are the equivalent to the Haskell operators ”>>” and ”>>=” respectively (see section 4.2.3).

Finally, the last difference worth mentioning is the use of the plus operator for string concatenation and also for automatically converting the integers to strings for them to be concatenated in the line:

```
= (x, y, gcd) => ”The GCD of ” + x + ” and ” + y + ” is ” + gcd
```

as opposed to using the ”++” operator and ”show” in the line:

```
”The GCD of ” ++ show x ++ ” and ” ++ show y ++ ” is ” ++ show gcd
```

Chapter 3

Language Description: General

3.1 Program Structure

An lcases program consists of a set of definitions, type nicknames and theorems. Definitions are split into value definitions, type definitions and type proposition definitions. Theorems are proven type propositions. Functions as well as ”Environment Actions” (see section 4.2.3) are also considered values. The definition of the ”main” value determines the program’s behaviour.

Program example: Euclidean Algorithm

```
gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => gcd_of(y)and((x)mod(y))

read_two_ints: (Int^2)FromIO
  = print(”Please give me 2 ints”);
    get_line ;> split(_)to_words o> cases
      [x, y] => (from_string(x), from_string(y)) -> (_)from_io
      ... => throw_err(”You didn’t give me 2 ints”)

tuple_type NumsAndGcd
value (x, y, gcd):Int^3

nag(_)to_message: NumsAndGcd => String
  = nag => ”The GCD of ” + nag.x + ” and ” + nag.y + ” is ” + nag.gcd

main: IO
  = read_two_ints ;> (i1, i2) =>
    (i1, i2, gcd_of(i1)and(i2)) -> nag(_)to_message -> print(_)
```

Program grammar

⟨*program*⟩ ::= ⟨*nl*⟩* ⟨*program-part*⟩ (⟨*nl*⟩ ⟨*nl*⟩ ⟨*program-part*⟩)* ⟨*nl*⟩*

⟨*program-part*⟩ ::=
⟨*value-def*⟩ | ⟨*grouped-value-defs*⟩ | ⟨*type-def*⟩ | ⟨*t-nickname*⟩ | ⟨*type-prop-def*⟩ | ⟨*type-theo*⟩

⟨*nl*⟩ :: (‘ ’ | ‘\t’)* ‘\n’

3.2 Keywords

The lcases keywords are the following:

```
cases all where tuple_type value or_type values
type_proposition needed equivalent type_theorem proof
```

Each keyword's functionality is described in the respective section shown in the table below:

Keyword	Section
`cases`	4.3.2 "cases" Function Expressions
`all where`	4.4 Value Definitions and "where" Expressions
`tuple_type value or_type values type_nickname`	5.1 Types
`type_proposition needed equivalent type_theorem proof`	5.2 Type Logic

The "`cases`" and "`where`" keywords are also reserved words. Therefore, even though they can be generated by the "identifiers" grammar, they cannot be used as identifiers (see "Literals and Identifiers" section 4.1.1).

Chapter 4

Language Description: Values

4.1 Basic Expressions

4.1.1 Literals and Identifiers

Literals

- *Examples*

```
1  2  17  42  -100
1.62  2.72  3.14  -1234.567
'a'  'b'  'c'  'x'  'y'  'z'  '.'  ','  '\n'
"Hello World!"  "What's up, doc?"  "Alrighty then!"
```

- *Description*

  There are literals for the four basic types: `Int, Real, Char, String`.

- *Grammar*

  ⟨*literal*⟩ ::= ⟨*int-lit*⟩ | ⟨*real-lit*⟩ | ⟨*char-lit*⟩ | ⟨*string-lit*⟩

Identifiers

- *Examples*

```
x y z
a1 a2 a3
(_)mod(_)
apply(_)to_all_in(_)
```

- *Description*

  An identifier is the name of a value or a parameter. It is used in the definition of a value and in expressions that use that value, or in the parameters of a function and in the body of that function.

  An identifier starts with a lower case letter, which can be followed by lower case letters or underscores and/or ended with a digit. It is also possible to have underscores in parenthesis before, after or in the middle of an identifier (see "Parenthesis Function Application" section 4.1.3 for why this can be useful).

  A simple identifier is an identifier without any underscores in parenthesis. It used in expressions where underscores in parenthesis don't make sense (e.g. "Prefix and Postfix Functions" 4.1.4).

- *Grammar*

  ⟨*identifier*⟩ ::= [⟨*unders-in-paren*⟩] ⟨*id-start*⟩ ⟨*id-cont*⟩* [[0-9]] [⟨*unders-in-paren*⟩]

  ⟨*simple-id*⟩ ::= ⟨*id-start*⟩ [[0-9]]

  ⟨*id-start*⟩ ::= [a-z] [a-z_]*

  ⟨*id-cont*⟩ ::= ⟨*unders-in-paren*⟩ [a-z_]+

  ⟨*unders-in-paren*⟩ ::= ‘(_’ (⟨*comma*⟩ ‘_’)* ‘)’

  ⟨*comma*⟩ ::= ‘,’ [‘ ’]

  Even though the ”`cases`” and ”`where`” keywords can be generated by these grammar rules, they cannot be used as identifiers.

4.1.2 Parenthesis, Tuples and Lists

Parenthesis

- *Examples*

```
(1 + 2)
(((1 + 2) * 3)^4)
(n => 3*n + 1)
(get_line ;> line => print(”Line is: ” + line))
```

- *Description*

  An expression is put in parenthesis to prioritize it or isolate it in a bigger (operator) expression. The expressions inside parenthesis are operator or function expressions.

  Parenthesis expressions cannot extend over multiple lines. For expressions that extend of over multiple lines new values must be defined.

- *Grammar*

  ⟨*paren-expr*⟩ ::= ‘(’ [‘ ’] ⟨*line-op-expr*⟩ | ⟨*line-func-expr*⟩ [‘ ’] ‘)’

Tuples

- *Examples*

```
(1, ”What’s up, doc?”)
(2, ”Alrighty then!”, 3.14)
(x, y, z, w)
(1, my_function, (x, y, z) => sqrt(x^2 + y^2 + z^2))
```

- *Description*

  Tuples are used to group many values (of possibly different types) into one. The type of a tuple can be either the product of the types of the fields or a defined `tuple_type` which is equivalent to the aforementioned product type (see ”Tuple Types” in section 5.1.2 for details). For example, the type of the second tuple above could be:

```
Int x String x Real
```

  or:

```
MyType
```

  assuming ”`MyType`” has been defined in a similar way to the following:

```
tuple_type MyType
value
  (my_int, my_string, my_real) : Int x String x Real
```

- *Big Tuples*

  Example

```
my_big_tuple
  : String x Int x Real x String x String x (String x Real x Real)
  = ( "Hey, I'm the first field and I'm also a relatively big string."
    , 42, 3.14, "Hey, I'm the first small string", "Hey, I'm the second small string"
    , ("Hey, I'm a string inside the nested tuple", 2.72, 1.62)
    )
```

  Description

  It is possible to stretch a (big) tuple expression over multiple lines (only) in a separate value definition (see "Value Definitions" section 4.4.1). In that case:

  – The character '`(`' is after the "= " part of the value definition and the first field must be in the same line.
  – The tuple can split in a new line only at a '`,`' character. Every such line must be indented so that the '`,`' is in same column where the '`(`' character was in the first line.
  – The tuple must be ended by a line that only contains the '`)`' character and is also indented so that the '`)`' is in same column where the '`(`' character was in the first line.
  – The precise indentation rules are described in the section "Indentation System" 7.1.2.

- *Tuples with empty fields*

  Examples

```
(42, _)
(_, 3.14, _)
(_, _, "Hello from 3rd field")
```

  Description

  It is possible to leave some fields empty in a tuple by having an underscore in their position. This creates a function that expects the empty fields and returns the whole tuple. This is best demonstrated by the types of the examples above:

```
(42, _) : T1 => Int x T1
(_, 3.14, _) : T1 x T2 => T1 x Real x T2
(_, _, "Hello from 3rd field") : T1 x T2 => T1 x T2 x String
```

  An example in a bigger expression is the following:

```
questions : ListOf(String)s
  = ["the Ultimate Question of Life", "the Universe", "Everything"]

answers_to : ListOf(String)s
  = apply("The answer to " + _)to_all_in(questions)
```

```
>> apply((42, _))to_all_in(answers_to)
  : ListOf(Int x String)s
  ==> [ (42, "The answer to the Ultimate Question of Life")
      , (42, "The answer to the Universe")
      , (42, "The answer to Everything")
      ]
```

- *Grammar*

  ⟨*tuple*⟩ ::= '(' [' '] ⟨*line-expr-or-under*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩ [' '] ')'

  ⟨*line-expr-or-unders*⟩ ::= ⟨*line-expr-or-under*⟩ (⟨*comma*⟩ ⟨*line-expr-or-under*⟩)*

  ⟨*line-expr-or-under*⟩ ::= ⟨*line-expr*⟩ | '_'

  ⟨*line-expr*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ | ⟨*line-func-expr*⟩

  ⟨*basic-or-app-expr*⟩ ::= ⟨*basic-expr*⟩ | ⟨*pre-func-app*⟩ | ⟨*post-func-app*⟩

  ⟨*basic-expr*⟩ ::= ⟨*literal*⟩ | ⟨*paren-func-app-or-id*⟩ | ⟨*special-id*⟩ | ⟨*tuple*⟩ | ⟨*list*⟩

  ⟨*big-tuple*⟩ ::=
  '(' [' '] ⟨*line-expr-or-under*⟩ [⟨*nl*⟩ ⟨*indent*⟩] ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩
  (⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩)*
  ⟨*nl*⟩ ⟨*indent*⟩ ')'

Lists

- *Examples*

```
[1, 2, 17, 42, -100]
[1.62, 2.72, 3.14, -1234.567]
["Hello World!", "What's up, doc?", "Alrighty then!"]
[x => x + 1, x => x + 2, x => x + 3]
[x, y, z, w]
```

- *Description*

  Lists are used to group many values of the same type into one. The type of the list is `ListOf(T1)s` where `T1` is the type of every value inside. Therefore, the types of the first four examples are:

```
ListOf(Int)s
ListOf(Real)s
ListOf(String)s
(@A)And(Int)Add_To(@B) --> ListOf(@A => @B)s
```

  And the last list is only legal if x, y, z and w all have the same type. Assuming they do and it's the type T, the type of the list is:

```
ListOf(T)s
```

- *Big Lists*

  Example

```
my_big_list: ListOf(Int => IO)s
  = [ x => print(”I’m the first function and x + 1 is: ” + (x + 1))
    , x => print(”I’m the second function and x + 2 is: ” + (x + 2))
    , x => print(”I’m the third function and x + 3 is: ” + (x + 3))
    ]
```

  Description

  It is possible to stretch a (big) list expression over multiple lines (only) in a separate value definition (see ”Value Definitions” section 4.4.1). In that case:

  – The character ’`[`’ is after the ”= ” part of the value definition and the first element must be in the same line.
  – The list can split in a new line only at a ’`,`’ character. Every such line must be indented so that the ’`,`’ is in same column where the ’`[`’ character was in the first line.
  – The list must be ended by a line that only contains the ’`]`’ character and is also indented so that the ’`]`’ is in same column where the ’`[`’ character was in the first line.
  – The precise indentation rules are described in the section ”Indentation System” 7.1.2.

- *Grammar*

  ⟨*list*⟩ ::= ‘`[`’ [‘ ’] [⟨*line-expr-or-unders*⟩] [‘ ’] ‘`]`’

  ⟨*big-list*⟩ ::=
  ‘`[`’ [‘ ’] ⟨*line-expr-or-unders*⟩
  (⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩)*
  ⟨*nl*⟩ ⟨*indent*⟩ ‘`]`’

4.1.3 Parenthesis Function Application

- *Examples*

```
f(x)
f(x, y, z)
(x)to_string
apply(f)to_all_in(l)
```

- *Description*

  Function application in lcases can be done in many different ways. In this section, we discuss the ways function application can be done with parenthesis.

  In the first two examples, the usual mathematical function application is used which is also used in most programming languages and should be familiar to the reader, i.e. function application is done with the arguments of the function in parenthesis separated by commas and **appended** to the function identifier.

  This idea can be extended by allowing the arguments to be **prepended** or to be **inside** to the function identifier (examples 3 and 4). This is only valid if the function has been **defined with these parentheses in the identifier**. For example, the definition for "`apply(_)to_all_in(_)`" starts like so:

```
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = <...definition>
```

  The identifier is "`apply(_)to_all_in(_)`" with the parentheses **included**. This is very useful for defining functions where the argument in the middle or before makes the function application look and sound more like natural language.

  It is possible to have many parentheses in a single function application (last example). The arguments are always inserted to the function from **left to right**. Therefore, when multiple parentheses are present the arguments of the leftmost parentheses are inserted first then the next ones to the right and so on.

- *Empty arguments in Parenthesis Function Application*

  It is possible to provide a function with a subset its arguments by putting an underscore to all the missing arguments. The resulting expression is a function that expects the missing arguments to return the final result. Let's see this in action:

```
f(_, _, _) : Char x Int x Real => String
c, i, r : Char, Int, Real

f(c, i, r) : String

f(_, i, r) : Char => String
f(c, _, r) : Int => String
f(c, i, _) : Real => String

f(c, _, _) : Int x Real => String
f(_, i, _) : Char x Real => String
f(_, _, r) : Char x Int => String
```

- *Grammar*

  ⟨*paren-func-app-or-id*⟩ ::=
  [⟨*arguments*⟩] ⟨*id-start*⟩ (⟨*arguments*⟩ [a-z_]+)* [[0-9]] [⟨*arguments*⟩]

  ⟨*arguments*⟩ ::= '(' [' '] ⟨*line-expr-or-unders*⟩ [' '] ')'

4.1.4 Prefix and Postfix Functions

Prefix Functions

- *Examples*

```
the_value:1
error:e
result:r
apply(the_value:_)to_all_in(_)
```

- *Description*

  Prefix functions are automatically generated from `or_type` definitions (see ”Or Types” in section 5.1.2). They are functions that convert a value of a particular type to a value that is a case of an `or_type` and has values of the first type inside. For example in the first example above we have:

```
1 : Int
the_value:1 : Possibly(Int)
```

  Where the function `the_value:_` is automatically generated from the definition of the `Possibly(_)` type:

```
or_type Possibly(T1)
values the_value:T1 | no_value
```

  And it has the type `T1 => Possibly(T1)`.

  These functions are called prefix functions because they are prepended to their argument. However, they can also be used as arguments to other functions with an underscore in their argument. This is illustrated in the last example, where the function `the_value:_` is an argument of the function `apply(_)to_all_in(_)`.

- *Grammar*

  ⟨*pre-func*⟩ ::= ⟨*simple-id*⟩ ‘`:`’

  ⟨*pre-func-app*⟩ ::= ⟨*pre-func*⟩ ⟨*operand*⟩

Postfix Functions

- *Examples*

```
name.first_name
list.head
date.year
tuple.1st
apply(_.1st)to_all_in(_)
```

- *Description*

  Postfix functions are automatically generated from `tuple_type` definitions (see "Tuple Types" in section 5.1.2). They are functions that take a `tuple_type` value and return a particular field (i.e. projection functions). For example in the first example above we have:

```
name : Name
name.first_name : String
```

  Where the function `_.first_name` is automatically generated from the definition of the `Name` type:

```
tuple_type Name
value (first_name, last_name) : String^2
```

  And it has the type `Name => String`.

  There are also the following special projection functions that work on all tuples that are of a product type: "`_.1st`", "`_.2nd`", "`_.3rd`", "`_.4th`", "`_.5th`" . For the 4th example above, assuming:

```
tuple : Int x String
```

  We have:

```
tuple.1st : Int
```

  The general types of these functions are:

```
_.1st : (@A)Is(@B)s_1st --> @B => @A
_.2nd : (@A)Is(@B)s_2nd --> @B => @A
...
```

  These functions are called postfix functions because they are appended to their argument. However, they can also be used as arguments to other functions with an underscore in their argument. This is illustrated in the last example, where the function "`_.1st`" is an argument of the function "`apply(_)to_all_in(_)`".

  There is a special postfix function called "`_.change`" which is described in the following paragraph.

- *Grammar*

  ⟨*post-func*⟩ ::= '`.`' (⟨*simple-id*⟩ | ⟨*special-id*⟩)

  ⟨*special-id*⟩ ::= '`1st`' | '`2nd`' | '`3rd`' | '`4th`' | '`5th`'

  ⟨*post-func-app*⟩ ::=
  (⟨*basic-expr*⟩ | ⟨*paren-expr*⟩ | '`_`') (⟨*dot-change*⟩ | ⟨*post-func*⟩+ [⟨*dot-change*⟩])

The ".change" Function

- *Examples*

```
state.change{counter = counter + 1}
tuple.change{1st = 42, 3rd = 17}
point.change{x = 1.62, y = 2.72, z = 3.14}
apply(_.change{1st = 1st + 1})to_all_in(_)
x.change{1st = _, 3rd = _}
```

- *Description*

  The "`_.change`" function is a special postfix function that works an all tuples. It returns a new tuple that is the same as the input tuple except for some changed fields. Which fields change and to what new value is specified inside curly brackets after the "`.change`". The following special identifiers can be used for referring to the fields of product type tuples: "`1st`", "`2nd`", "`3rd`", "`4th`", "`5th`" (2nd, 4th and 5th example). If the tuple is of a tuple type, the identifiers of the fields specified in the type definition are used (1st and 3rd example). Therefore, we are assuming the following (or similar) if the examples are to type check:

```
tuple_type MyStateType
value (..., counter, ...) : ... x Int x ...

state : MyStateType


tuple : Int x SomeType x Int (x ...)


tuple_type Point
value (x, y, z) : Real^3

point : Point


apply(_.change{1st = 1st + 1})to_all_in(_)
  : (@A)And(Int)AddTo(@A), (@A)Is(@B)s_1st --> ListOf(@B)s => ListOf(@B)s


x : Int x Real x String
x.change{1st = _, 3rd = _} : Int x String => Int x Real x String
```

The changes of the fields have the following structure: "`field = <expression of new value>`" and they are separated by commas. The input tuple's fields (i.e. the "old" values) can be used inside the expression of a new value and they are referred to by the field identifier (1st and 4th example). Underscores can be used as the expressions of some new values which makes the whole expression a function that expects those new values as arguments (last example).

- *Grammar*

  ⟨*dot-change*⟩ ::= '`.change{`' [' '] ⟨*field-change*⟩ (⟨*comma*⟩ ⟨*field-change*⟩)* [' '] '`}`'

  ⟨*field-change*⟩ ::= (⟨*simple-id*⟩ | ⟨*special-id*⟩) ⟨*equals*⟩ ⟨*line-expr-or-under*⟩

  ⟨*equals*⟩ ::= [' '] '`=`' [' ']

4.2 Operators

4.2.1 Function Application and Function Composition Operators

Function Application Operators

Operator	Type
`->`	`T1 x (T1 => T2) => T2`
`<-`	`(T1 => T2) x T1 => T2`

The function application operators "`->`" and "`<-`" are a different way to apply functions to arguments than the usual parenthesis function application. They are meant to look like arrows that point from the argument to the function. These operators are very useful for chaining many function applications without the clutter of having to open and close parentheses for each one of the functions. For example, assuming we have the following functions with the behaviour suggested by their names and types:

```
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
str_len(_) : String => Int
filter(_)with(_) : ListOf(T1)s x (T1 => Bool) => ListOf(T1)s
(_)is_odd : Int => Bool
sum_ints(_) : ListOf(Int)s => Int
```

And a list of strings:

```
strings : ListOf(String)s
```

Here is a simple way to get the total number of characters in all the strings that have odd length:

```
chars_in_odd_length_strings : Int
  = apply(str_len(_))to_all_in(strings) -> filter(_)with((_)is_odd) -> sum_ints(_)
```

This can be done equivalently using the other operator:

```
chars_in_odd_length_strings : Int
  = sum_ints(_) <- filter(_)with((_)is_odd) <- apply(str_len(_))to_all_in(strings)
```

Function Composition Operators

Operator	Type
`o>`	`(T1 => T2) x (T2 => T3) => (T1 => T3)`
`<o`	`(T2 => T3) x (T1 => T2) => (T1 => T3)`

The function composition operators ”o>” and ”<o” are used to compose functions, each one in the corresponding direction. The use of the letter ’o’ is meant to be similar to the mathematical function composition symbol ’∘’ and the symbols ’>’, ’<’ are used so that the operator points from the function which is applied first to the function which is applied second. A neat example using function composition is the following. Assuming we have the following functions with the behaviour suggested by their names and types:

```
split(_)to_words : String => ListOf(String)s
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
reverse_str(_) : String => String
merge_words(_) : ListOf(String)s => String
```

We can reverse the all the words in a string like so:

```
reverse_words_in(_) : String => String
  = split(_)to_words o> apply(reverse_str(_))to_all_in(_) o> merge_words(_)
```

This can be done equivalently using the other operator:

```
reverse_words_in(_) : String => String
  = merge_words(_) <o apply(reverse_str(_))to_all_in(_) <o split(_)to_words
```

4.2.2 Arithmetic, Comparison and Boolean Operators

Arithmetic Operators

Operator	Type
^	(@A)To_The(@B)Is(@C) --> @A x @B => @C
*	(@A)And(@B)Multiply_To(@C) --> @A x @B => @C
/	(@A)Divided_By(@B)Is(@C) --> @A x @B => @C
+	(@A)And(@B)Add_To(@C) --> @A x @B => @C
-	(@A)Minus(@B)Is(@C) --> @A x @B => @C

The usual arithmetic operators work as they are expected, similarly to mathematics and other programming languages for the usual types. However, they are generalized. The examples below show their generality:

```
>> 1 + 1
  : Int
  ==> 2
>> 1 + 3.14
  : Real
  ==> 4.14
>> 'a' + 'b'
  : String
  ==> "ab"
>> 'w' + "ord"
  : String
  ==> "word"
>> "Hello " + "World!"
  : String
  ==> "Hello World!"
>> 5 * 'a'
  : String
  ==> "aaaaa"
>> 5 * "hi"
  : String
  ==> "hihihihihi"
>> "1,2,3" - ','
  : String
  ==> "123"
```

Let's analyze further the example of addition. The type can be read as such: the '+' operator has the type
`@A x @B => @C`, provided that the type proposition `(@A)And(@B)Add_To(@C)` holds. This proposition being true, means that addition has been defined for these three types (see section "Type Logic" 5.2 for more on type propositions). Therefore, by the examples above we can deduce that the following propositions are true (in the order of the examples):

```
(Int)And(Int)Add_To(Int)
(Int)And(Real)Add_To(Real)
(Char)And(Char)Add_To(String)
(Char)And(String)Add_To(String)
(Int)And(Char)Multiply_To(String)
(Int)And(String)Multiply_To(String)
(String)Minus(Char)Is(String)
```

This allows us to use the familiar arithmetic operators in types that are not necessarily numbers but it is somewhat intuitively obvious what they should do in those other types. Furthermore, their behaviour can be defined by the user for new user defined types!

Comparison, Boolean and Bitwise Operators

Operator	Type
==	(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
!=	(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool
>	(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool
<	(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool
>=	(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool
<=	(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool
&	(@A)Has_And --> @A^2 => @A
|	(@A)Has_Or --> @A^2 => @A

Comparison operators are also generalized. The main reason for the generalization is to be able to compare numbers of different types. Consider the following example:

```
>> 1
  : Int
  ==> 1
>> 1.1
  : Real
  ==> 1.1
>> 1.1 == 1
  : Bool
  ==> false
>> 1.0 == 1
  : Bool
  ==> true
```

In order for the example to work we need to be able to compare integers and reals. Similarly, all the comparison operators need to be able to work on arguments of different types.

Boolean ”and” and bitwise ”and” are combined into one general ”and” operator (`&`). The same applies to the ”or” operator (`|`).

4.2.3 Environment Action Operators

Operator	Type
;>	(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)
;	(@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)

Simple Example Program

```
main: (EmptyVal)FromIO
  = print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)
```

The example above demonstrates the use of the environment action operators with the `(_)FromIO` environment type, which is how IO is done in lcases. Some light can be shed on how this is done, if we take a look at the types (as always!):

```
print_string(_): String => (EmptyVal)FromIO
print_string(”I’ll repeat the line”): (EmptyVal)FromIO
get_line: (String)FromIO
```

For the ”then” operator we have: `; : (@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)`

In the following expression: `print_string(”I’ll repeat the line”) ; get_line`
the ”then” operator has type: `(EmptyVal)FromIO x (String)FromIO => SomeType`

The only way to match the types is:
`@E(_) = (_)FromIO, T1 = EmptyVal, T2 = String and SomeType = (String)FromIO`
and it type checks because: `((_)FromIO)Has_Then`

For the whole expression we have:

```
print_string(”I’ll repeat the line”) ; get_line
  : (String)FromIO
```

For the ”use” operator we have: `;> : (@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)`

In the following expression: `print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)`
the ”use” operator has type: `(String)FromIO x (String => (EmptyVal)FromIO) => SomeType`

The only way to match the types is:
`@E(_) = FromIO, T1 = String, T2 = EmptyVal and SomeType = (EmptyVal)FromIO`
and it type checks because: `((_)FromIO)Has_Use`

For the whole expression we have:

```
print_string(”I’ll repeat the line”) ; get_line ;> print_string(_)
  : (EmptyVal)FromIO
```

Another Example Program

```
main: (EmptyVal)FromIO
  = print_string(”Hello! What’s your name?”) ; get_line ;> name =>
    print_string(”And how old are you?”) ; get_line ;> age =>
    print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)

print_string(_): String => (EmptyVal)FromIO

print_string(”Hello! What’s your name?”) : (EmptyVal)FromIO

print_string(”Hello! What’s your name?”); get_line
  : (String)FromIO

print_string(”And how old are you?”); get_line
  : (String)FromIO

print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO

age => print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : String => (EmptyVal)FromIO

print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO

name =>
print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : String => (EmptyVal)FromIO

print_string(”Hello! What’s your name?”) ; get_line ;> name =>
print_string(”And how old are you?”) ; get_line ;> age =>
print_string(”Oh! You don’t look ” + age + ” ” + name + ”!”)
  : (EmptyVal)FromIO
```

Description

The environment action operators are used to combine values that do environment actions into values that do more complicated environment actions. Environment types are type functions that take a type argument and produce a type (just like `ListOf(_)s`). These types have the ”then” operator (`;`) and the ”use” operator (`;>`) defined for them. A value of the type `@E(T1)` where `(@E)Has_Then` does an environment action of type `@E(_)` that produces a value of type `T1` which can then be combined with another one with the ”then” operator. Similarly, with the ”use” operator the produced value of an action can be used by a function that returns another action.

The effect of the ”`;`” operator described in words is the following: given a value of type `@E(T1)` and a value of type `@E(T2)` (which are environment actions that produce values of type `T1` and `T2` respectively), create a new value the does both actions (provided the first did not result in an error). The overall effect is a value which is an environment action of type `@E(_)` (the combination of the ”smaller” actions) which produces a value of type `T2` (the one produced by the second action) and therefore it is of type `@E(T2)`.

Note that the value of type `T1` produced by the first action is not used anywhere. This happens mostly when `T1 = EmptyVal` and it is because values of type `@E(EmptyVal)` are used for their environment action only
(e.g. `print_string(...): (EmptyVal)FromIO`).

How the two environment actions of the `@E(T1)` and `@E(T2)` values are combined to produce the new environment action is specific to the environment action type `@E(_)`.

The effect of the ”`;>`” operator described in words is the following: given a value of type `@E(T1)` (which is an environment action of type `@E(_)` that produces a value of type `T1`) and a value of type `T1 => @E(T2)` (which is a function that takes a value of type `T1` and returns an environment action of type `@E(_)` that produces a value of type `T2`), combine those two values by creating a value that does the following:

- Performs the first action that produces a value of type `T1`
- Takes the value of type `T1` produced (provided there was no error) and passes it to the function of type `T1 => @E(T2)` that then returns an action
- Performs the resulting action

The overall effect is an environment action of type `@E(_)` that produces a value is of type `T2` and therefore the new value is of type `@E(T2)`.

4.2.4 Operator Expressions

- *Examples*

```
1 + 2
1 + x * 3^y
"Hello " + "World!"
x -> f -> g
f o> g o> h
x == y
x >= y - z & x < 2 * y
get_line ; get_line ;> line => print("Second line: " + line)
2 * _
_ - 1
"Hello " + "it's me, " + _
"Hi, I am " + _ + " and I am " + _ + " years old"
```

- *Description*

  Operator expressions are expressions that are comprised of operators and operands. Operators act like two-argument-functions that are placed in between their arguments (operands). Therefore, they have function types and they act as it is described in their respective sections above this one.

  An operator expression might have multiple operators. The order of operations is explained in the next section ("Complete Operator Table, Precedence and Associativity") in Table 4.2.

  Just like functions, the operands of an operator, must have types that match the types expected by the operator.

  It is possible for the second operand of an operator to be a function expression. This is mostly useful with the ";>" operator (see previous section: "Environment Operators").

  It is possible to use an underscore as an operand. An operator expression with underscore operands becomes a function that expects those operands as arguments. This is best demonstrated by the types of the last four examples:

```
2 * _ : Int => Int
_ - 1 : Int => Int
"Hello " + "it's me, " + _ : String => String
"Hi, I am " + _ + " and I am " + _ + " years old" : String^2 => String
```

  Note: These are not the most general types for these examples but they are compatible.

- *Big Operator Expressions*

  Example

```
"Hello, I'm a big string that's going to contain multiple values from " +
"inside the imaginary program that I'm a part of. Here they are:\n" +
"value1 = " + value1 + ", value2 = " + value2 + ", value3 = " + value3 +
", value4 = " + value4 + ", value5 = " + value5
```

  Description

  It is possible to stretch a (big) operator expression over multiple lines. In that case:

  – The operator expression must split in a new line after an operator (not an operand).
  – Every line after the first must be indented so that in begins at the column where the first line of the operator expression begun.
  – The precise indentation rules are described in the section "Indentation System" 7.1.2.

- *Grammar*

  ⟨*op-expr*⟩ ::= ⟨*line-op-expr*⟩ | ⟨*big-op-expr*⟩

  ⟨*op-expr-start*⟩ ::= (⟨*operand*⟩ ⟨*op*⟩)+

  ⟨*line-op-expr*⟩ ::= ⟨*op-expr-start*⟩ (⟨*operand*⟩ | ⟨*line-func-expr*⟩)

  ⟨*big-op-expr*⟩ ::= ⟨*big-op-expr-op-split*⟩ | ⟨*big-op-expr-func-split*⟩

  ⟨*big-op-expr-op-split*⟩ ::= ⟨*op-split-line*⟩+ [⟨*op-expr-start*⟩] (⟨*operand*⟩ | ⟨*func-expr*⟩)

  ⟨*op-split-line*⟩ ::= (⟨*op-expr-start*⟩ (⟨*nl*⟩ | ⟨*oper-fco*⟩) | ⟨*oper-fco*⟩) ⟨*indent*⟩

  ⟨*oper-fco*⟩ ::= ⟨*operand*⟩ ' ' ⟨*func-comp-op*⟩ '\n'

  ⟨*big-op-expr-func-split*⟩ ::= ⟨*op-expr-start*⟩ (⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩)

  ⟨*operand*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*paren-expr*⟩ | '_'

  ⟨*op*⟩ ::= ' ' ⟨*func-comp-op*⟩ ' ' | [' '] ⟨*optional-spaces-op*⟩ [' ']

  ⟨*func-comp-op*⟩ ::= 'o>' | '<o'

  ⟨*optional-spaces-op*⟩ ::=
  '->' | '<-' | '^' | '*' | '/' | '+' | '-' | '==' | '!=' | '>' | '<' | '>=' | '<=' | '&' | '|' | ';>' | ';'

4.2.5 Complete Operator Table, Precedence and Associativity

Table 4.1: The complete operator table of lcases operators along with their types and their short descriptions.

Op	Type	Description
`->`	`T1 x (T1 => T2) => T2`	Right Function Application
`<-`	`(T1 => T2) x T1 => T2`	Left Function Application
`o>`	`(T1 => T2) x (T2 => T3) => (T1 => T3)`	Right Function Composition
`<o`	`(T2 => T3) x (T1 => T2) => (T1 => T3)`	Left Function Composition
`^`	`(@A)To_The(@B)Is(@C) --> @A x @B => @C`	General Exponentiation
`*`	`(@A)And(@B)Multiply_To(@C) --> @A x @B => @C`	General Multiplication
`/`	`(@A)Divided_By(@B)Is(@C) --> @A x @B => @C`	General Division
`+`	`(@A)And(@B)Add_To(@C) --> @A x @B => @C`	General Addition
`-`	`(@A)Minus(@B)Is(@C) --> @A x @B => @C`	General Subtraction
`==`	`(@A)And(@B)Can_Be_Equal --> @A x @B => Bool`	General Equality
`!=`	`(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool`	General Inequality
`>`	`(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool`	General Greater Than
`<`	`(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool`	General Less Than
`>=`	`(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool`	General Greater Than or Equal To
`<=`	`(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool`	General Less Than or Equal To
`&`	`(@A)Has_And --> @A^2 => @A`	General And
`|`	`(@A)Has_Or --> @A^2 => @A`	General Or
`;>`	`(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)`	”Use” Environment Action
`;`	`(@E(_))Has_Then --> @E(T1) x @E(T2) => @E(T2)`	”Then” Environment Action

The order of operations is done from highest to lowest precedence. In the same level of precedence the order is done from left to right if the associativity is ”Left” and from right to left if the associativity is ”Right”. For the operators that have associativity ”None” it is not allowed to place them in the same operator expression. The precedence and associativity of the operators is shown in the table below.

Table 4.2: The table of precedence and associativity of the lcases operators.

Operator	Precedence	Associativity
`->`	10 (highest)	Left
`<-`	9	Right
`o> <o`	8	Left
`^`	7	Right
`* /`	6	Left
`+ -`	5	Left
`== != > < >= <=`	4	None
`&`	3	Left
`|`	2	Left
`;> ;`	1	Left

4.3 Function Expressions

Function expressions are divided into **regular function expressions** and **"cases" function expressions** which are described in the following sections.

⟨*func-expr*⟩ ::= ⟨*line-func-expr*⟩ | ⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩

4.3.1 Regular Function Expressions

- *Examples*

```
a => 17 * a + 42
(a, b) => a + 2*b
(x, y, z) => sqrt(x^2 + y^2 + z^2)
* => 42
(x, *, z) => x + z
((x1, y1), (x2, y2)) => (x1 + x2, y1 + y2)
```

- *Description*

  Regular function expressions are used to define functions or be part of bigger expressions as anonymous functions. They are comprised by their parameters and their body.

  Parameters have identifiers. There is either only one parameter, in which case there is no parenthesis, or there are many, in which case they are in parentheses, separated by commas. If a parameter is not needed it can be left empty by having an asterisk instead of an identifier (4th and 5th example). If a parameter is a tuple itself it can be matched further by using parentheses and giving identifiers to its fields (6th example).

  The parameters and the body are separated by the function arrow ("=>"). The body is a basic expression, an operator expression or a function expression in parenthesis.

- *Big Function Expressions*

  Example

```
(value1, value2, value3, value4, value5, value6, value7) =>
print("value1 = " + value1 + ", value2 = " + value2 + ", value3 = " + value3) ;
print("value4 = " + value4 + ", value5 = " + value5 + ", value6 = " + value6) ;
print("value7 = " + value7)
```

  Description

  It is possible to stretch a (big) function expression over multiple lines. In that case:

  – The function expression must split in a new line after the function arrow ("=>").
  – Every line after the first must be indented so that in begins at the column where the first character of the parameters was in the first line.
  – The precise indentation rules are described in the section "Indentation System" 7.1.2.

- *Grammar*

  ⟨*line-func-expr*⟩ ::= ⟨*parameters*⟩ [‘ ’] ‘=>’ ⟨*line-func-body*⟩

  ⟨*big-func-expr*⟩ ::= ⟨*parameters*⟩ [‘ ’] ‘=>’ ⟨*big-func-body*⟩

  ⟨*parameters*⟩ ::=
  ⟨*identifier*⟩ | ‘*’ | ‘(’ [‘ ’] ⟨*parameters*⟩ (⟨*comma*⟩ ⟨*parameters*⟩)+ [‘ ’] ‘)’

  ⟨*line-func-body*⟩ ::=
  [‘ ’] (⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ | ‘(’ [‘ ’] ⟨*line-func-expr*⟩ [‘ ’] ‘)’)

  ⟨*big-func-body*⟩ ::=
  ⟨*nl*⟩ ⟨*indent*⟩ (⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ | ‘(’ [‘ ’] ⟨*line-func-expr*⟩ [‘ ’] ‘)’)

4.3.2 ”cases” Function Expressions

- *Examples*

```
print_sentimental_bool(_): Bool => IO
  = cases
    true => print(”It’s true!! :)”)
    false => print(”It’s false... :(”)

or_type TrafficLight
values green | amber | red

(_)is_not_red: TrafficLight => Bool
  = cases
    green => true
    amber => true
    red => false

(_)is_seventeen_or_forty_two: Int => Bool
  = cases
    17 => true
    42 => true
    ... => false

traffic_lights_match(_, _): TrafficLight^2 => Bool
  = (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false

gcd_of(_)and(_): Int^2 => Int
  = (x, cases)
    0 => x
    y => gcd_of(y)and((x)mod(y))

apply(_)to_all_in(_): (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = (f(_), cases)
    [] => []
    [head, tail = ...] => f(head) + apply(f(_))to_all_in(tail)

(_)is_sorted: (@A)Has_Less_Than_Or_Equal --> ListOf(@A)s => Bool
  = cases
    [x1, x2, xs = ...] => (x1 < x2) & (x2 + xs)is_sorted
    ... => true
```

- *Description*

  ”`cases`” is a keyword that works as a special parameter. Instead of giving the name ”`cases`” to that parameter, it is used to pattern match on the possible values of that parameter and return a different result for each particular case.

  The last case can be ”`... => (body of default case)`” to capture all remaining cases while dismissing the value (e.g. ”`is_seventeen_or_forty_two`” example), or it can be ”`some_id => (body of default case)`” to capture all remaining cases while being able to use the value with the name ”`some_id`” (e.g. ”`y`” in ”`gcd`” example).

  It is possible to use ”`cases`” in multiple parameters to match on all of them combined. By doing that, each case represents a particular combination of values for the ”`cases`” parameters involved
  (e.g. `traffic_lights_match` example).

  It is also possible to use a ”`where`” expression below a particular case. The ”`where`” expression must be indented two spaces more than than the line where that particular case begins.

  It is also possible to use the following syntax to match on as many elements of a list as needed and optionally give a name to the rest of the list:

```
# no name for the rest of the list
[x1, ...] => <case body>
[x1, x2, ...] => <case body>
[x1, x2, x3, ...] => <case body>

# name for the rest of the list
[x1, xs = ...] => <case body>
[x1, x2, xs = ...] => <case body>
[x1, x2, x3, xs = ...] => <case body>
```

- *Grammar*

⟨*cases-func-expr*⟩ ::= ⟨*cases-params*⟩ ⟨*case*⟩+ [⟨*end-case*⟩]

⟨*cases-params*⟩ ::=
    ⟨*identifier*⟩ | ‘`cases`’ | ‘`*`’ |
    ‘`(`’ [‘ ’] ⟨*cases-params*⟩ (⟨*comma*⟩ ⟨*cases-params*⟩)+ [‘ ’] ‘`)`’

⟨*case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ⟨*outer-matching*⟩ [‘ ’] ‘`=>`’ ⟨*case-body*⟩

⟨*end-case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ (‘`...`’ | ⟨*identifier*⟩) [‘ ’] ‘`=>`’ ⟨*case-body*⟩

⟨*outer-matching*⟩ ::= ⟨*simple-id*⟩ | ⟨*matching*⟩

⟨*matching*⟩ ::= ⟨*literal*⟩ | ⟨*pre-func*⟩ ⟨*inner-matching*⟩ | ⟨*tuple-matching*⟩ | ⟨*list-matching*⟩

⟨*inner-matching*⟩ ::= ‘`*`’ | ⟨*identifier*⟩ | ⟨*matching*⟩

⟨*tuple-matching*⟩ ::= ‘`(`’ [‘ ’] ⟨*inner-matching*⟩ (⟨*comma*⟩ ⟨*inner-matching*⟩)+ [‘ ’] ‘`)`’

⟨*list-matching*⟩ ::=
    ‘`[`’ [‘ ’] [⟨*inner-matching*⟩ (⟨*comma*⟩ ⟨*inner-matching*⟩)* [⟨*rest-list-matching*⟩]]
    [‘ ’] ‘`]`’

⟨*rest-list-matching*⟩ ::= ⟨*comma*⟩ [⟨*simple-id*⟩ ⟨*equals*⟩] ‘`...`’

⟨*case-body*⟩ ::= ⟨*line-func-body*⟩ | ⟨*big-func-body*⟩ [⟨*where-expr*⟩]

4.4 Value Definitions and ”where” Expressions

4.4.1 Value Definitions

- *Examples*

```
foo: Int
  = 42

f(_, _, _): Int^3 => Int
  = (a, b, c) => a + b * c

val1, val2, val3: Int, Bool, Char
  = 42, true, 'a'

int1, int2, int3: all Int
  = 1, 2, 3
```

- *Description*

  Value definitions are the main building block of lcases programs. To define a new value you give it a name, a type and an expression. The name is an identifier which is followed by the ”has type” symbol (’`:`’) and the expression of the type of the value. The line below is indented two spaces and begins with the equal sign and continues with the expression of the value (which extends to as many lines as needed).

  A value definition begins either in the first column, where it can be ”seen” by all other value definitions, or it is inside a ”`where`” expression (see section below), where it can be ”seen” by the expression above the ”`where`” and all the other definitions in the same ”`where`” expression.

  A value definition can be followed by a ”`where`” expression where intermediate values used in the value expression are defined. In that case, the ”`where`” expression must be indented two spaces more than the ”=” line of the value definition.

  It is possible to group value definitions together by separating the names, the types and the expressions with commas. This is very useful for not cluttering the program with many definitions for values with small expressions (e.g. constants). When grouping definitions together it is also possible to use the keyword ”`all`” to give the same type to all the values.

- *Grammar*

  ⟨*value-def*⟩ ::=
  ⟨*indent*⟩ ⟨*identifier*⟩ ([‘ ’] ‘`:`’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘`: `’) ⟨*type*⟩
  ⟨*nl*⟩ ⟨*indent*⟩ ‘`= `’ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

  ⟨*value-expr*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ | ⟨*func-expr*⟩ | ⟨*big-tuple*⟩ | ⟨*big-list*⟩

  ⟨*grouped-value-defs*⟩ ::=
  ⟨*indent*⟩ ⟨*identifier*⟩ (⟨*comma*⟩ ⟨*identifier*⟩)+
  ([‘ ’] ‘`:`’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘`: `’) (⟨*type*⟩ (⟨*comma*⟩ ⟨*type*⟩)+ | ‘`all `’ ⟨*type*⟩)
  ⟨*nl*⟩ ⟨*indent*⟩ ‘`= `’ ⟨*line-exprs*⟩ (⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-exprs*⟩)*

  ⟨*line-exprs*⟩ ::= ⟨*line-expr*⟩ (⟨*comma*⟩ ⟨*line-expr*⟩)*

4.4.2 ”where” Expressions

- *Examples*

```
sort(_): ListOf(Int)s => ListOf(Int)s
  = cases
    [] => []
    [head, tail = ...] =>
      sort(less_l) + head + sort(greater_l)
      where
      less_l, greater_l: all ListOf(Int)s
        = filter(tail)with(_ < head), filter(tail)with(_ >= head)

sum_nodes(_): TreeOf(Int)s => Int
  = tree =>
    tree.root + apply(sum_nodes(_))to_all_in(tree.subtrees) -> sum_list(_)
    where
    sum_list(_) : ListOf(Int)s => Int
      = cases
        [] => 0
        [head, tail = ...] => head + sum_list(tail)

big_string : String
  = s1 + s2 + s3 + s4
    where
    s1, s2, s3, s4 : all String
      = ”Hello, my name is Struggling Programmer.”
      , ” I have tried way too many times to fit a big chunk of text”
      , ” inside my program, without it hitting the half-screen mark!”
      , ” I am so glad I finally discovered lcases!”
```

- *Description*

  ”`where`” expressions allow the programmer to use values inside an expression and define them below it. They are very useful for reusing or abbreviating expressions that are specific to a particular definition or case.

  A ”`where`” expression begins by a line that only has the word ”`where`” in it. It is indented as described in the ”Value Definitions” (4.4.1) or ”’cases’ Function Expressions” (4.3.2) sections. The definitions are placed below the ”`where`” line and must have the same indentation.

- *Grammar*

  ⟨*where-expr*⟩ ::=
  ⟨*nl*⟩ ⟨*indent*⟩ ‘`where`’ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩ (⟨*nl*⟩ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩)*

  ⟨*value-def-or-defs*⟩ ::= ⟨*value-def*⟩ | ⟨*grouped-value-defs*⟩

Chapter 5

Language Description: Types and Type Logic

5.1 Types

The constructs regarding types are **type expressions**, **type definitions** and **type nicknames** and they are described in the following sections.

5.1.1 Type Expressions

Type expressions are divided into the following categories:

- Type Identifiers
- Type Variables
- Type Application Types
- Product Types
- Function Types
- Conditional Types

which are described in the following paragraphs.

The grammar of a type expression is:

⟨*type*⟩ ::= [⟨*condition*⟩] ⟨*simple-type*⟩

⟨*simple-type*⟩ ::= ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | ⟨*func-type*⟩

Type Identifiers

- *Examples*

```
Int     Real     Char     String     SelfReferencingType
```

- *Description*

  A type identifier is either the name of a basic type (Int, Real, Char, String) or the name of some defined type that has no type parameters. It begins with a capital letter and is followed by capital or lowercase letters.

- *Grammar*

  ⟨*type-id*⟩ ::= [A-Z] [A-Za-z]*

Type Variables

Type Variables are placeholders inside bigger type expressions that can be substituted with various types. This makes the bigger type expression an expression of a **polymorphic** type. The types of polymorphism that exist in lcases are **parametric polymorphism** and **ad hoc polymorphism**. Type variables for each of the two types have different syntax and they are described in the following paragraphs.

Parametric Type Variables

- *Examples*

```
T1     T2     T3
```

- *Examples of parametric type variables inside bigger type expressions*

```
T1 => T1
(T1 => T2) x (T2 => T3) => (T1 => T3)
(T1^2 => T1) x T1 x ListOf(T1)s => T1
```

- *Description*

  Parametric type variables can be substituted with any type and the program will type check. The simplest example of a polymorphic type with a parametric type variable is the type of the identity function where we have:

```
id(_): T1 => T1
  = x => x

id(1): Int
  where T1 is substituted by Int and id gets the type Int => Int

id(”Hello”): String
  where T1 is substituted by String and id gets the type String => String
```

  A parametric type variable is written with capital ”T” followed by a digit.

- *Grammar*

  ⟨*param-t-var*⟩ ::= ‘`T`’ [0-9]

Ad Hoc Type Variables

- *Examples*

```
@A @B @C @T
```

- *Examples of ad hoc type variables inside bigger type expressions*

```
(@T)Has_Str_Rep --> @T => String
(@A)Is(@B)s_First --> @B => @A
(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
(@A)And(@B)Add_To(@C) --> @A x @B => @C
```

- *Description*

  Ad hoc type variables are type variables that can be substituted only by types that satisfy a condition. This condition comes in the form of a type proposition (see Type Logic section 5.2). Therefore, any ad hoc type variable must also appear in the condition as shown in the examples.

  An ad hoc type variable is written with an '@' followed by any capital letter.

- *Grammar*

  ⟨*ad-hoc-t-var*⟩ ::= '`@`' [A-Z]

Type Application Types

- *Examples*

```
Possibly(Int)
ListOf(Real)s
TreeOf(String)s
Result(Int)OrError(String)
ListOf(Int => Int)s
ListOf(T1)s
```

- *Description*

  Type application types are types that are produced by passing type arguments to a type function generated by a `tuple_type` definition, an `or_type` definition or a `type_nickname`. For example, given the definition of `Possibly(T1)`:

```
or_type Possibly(T1)
values the_value:T1 | no_value
```

  We have that `Possibly(_)` is a type function that receives one type parameter and returns a resulting type. For example `Possibly(Int)` is the result of passing the type argument `Int` to `Possibly(_)`.

  Type application types have the same form as the name in the `tuple_type` definition, `or_type` definition or `type_nickname`, with the difference that type parameters are substituted by the expressions of the type arguments.

- *Grammar*

  ⟨*type-app-id-or-ahtv*⟩ ::= [⟨*types-in-paren*⟩] ⟨*taioa-middle*⟩ [⟨*types-in-paren*⟩]

  ⟨*taioa-middle*⟩ ::= ⟨*type-id*⟩ (⟨*types-in-paren*⟩ [A-Za-z]+)* | ⟨*ad-hoc-t-var*⟩

  ⟨*types-in-paren*⟩ ::= '`(`' [' '] ⟨*simple-type*⟩ (⟨*comma*⟩ ⟨*simple-type*⟩)* [' '] '`)`'

Product Types

- *Examples*

```
Int x Real x String
ListOf(Int)s x Int x ListOf(String)s
(Int => Int) x (Int x Real) x (Real => String)
Int^2 x Int^2
Real^3 x Real^3
```

- *Description*

  Product types are the types of tuples. They are comprised of the expressions of the types of the fields separated by the string " x " (space 'x' space) to resemble the cartesian product. If any of the fields is of a product or a function type then the corresponding type expression must be inside parentheses. A product type where all the fields are of the same type can be abbreviated with a power type expression which is comprised of the type, the power symbol '^' and the number of times the type is repeated.

- *Grammar*

  ⟨*prod-type*⟩ ::= ⟨*field-type*⟩ (' x ' ⟨*field-type*⟩)+

  ⟨*field-type*⟩ ::= ⟨*power-base-type*⟩ | ⟨*power-type*⟩

  ⟨*power-base-type*⟩ ::=
  ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | '(' [' '] (⟨*prod-type*⟩ | ⟨*func-type*⟩) [' '] ')'

  ⟨*power-type*⟩ ::= ⟨*power-base-type*⟩ '^' ⟨*int-greater-than-one*⟩

Function Types

- *Examples*

```
String => String
Real => Int
T1 => T1
Int^2 => Int
Real^3 => Real
(T1 => T2) x (T2 => T3) => (T1 => T3)
(Int => Int) => (Int => Int)
```

- *Description*

  A function type expression is comprised of the input type expression and the output type expression separated by the function arrow ("=>"). The input and output type expressions are type expressions which are put in parentheses only if they are function type expressions.

- *Grammar*

  ⟨*func-type*⟩ ::= ⟨*in-or-out-type*⟩ ' => ' ⟨*in-or-out-type*⟩

  ⟨*in-or-out-type*⟩ ::=
  ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ |
  '(' [' '] ⟨*func-type*⟩ [' '] ')'

Conditional Types

- *Examples*

```
(@A)And(@B)Can_Be_Equal --> @A x @B => Bool
(@A)And(@B)Add_To(@C) --> @A x @B => @C
(@A)Is(@B)s_First --> @B => @A
(@T)Has_Str_Rep --> @T => String
(@E(_))Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)
```

- *Description*

  Conditional types are the types of values that are polymorphic not because of their structure but because they have been defined (separately) for many different combinations of types (i.e. they are ad hoc polymorphic). They are comprised of a condition and a ”simple” type (i.e. a type without a condition) which are separated by the condition arrow (” `-->` ”). The condition is a type proposition which refers to type variables inside the ”simple” type and it must hold whenever the polymorphic value of that type is used. For example:

```
(_)first: (@A)Is(@B)s_First --> @B => @A
```

  can be used as follows:

```
pair, triple, list
  : Int x String, Real x Char x Int, ListOf(String)s
  = (42, ”The answer to everything”), (3.14, ’a’, 1), [”Hi!”, ”Hello”, Heeey”]

>> (pair)first
  : Int
  ==> 42
>> (triple)first
  : Real
  ==> 3.14
>> (list)first
  : String
  ==> ”Hi!”
```

  and that is because the following propositions hold:

```
(Int)Is(Int x String)s_First
(Real)Is(Real x Char x Int)s_First
(String)Is(ListOf(String)s)s_First
```

  which it turn means that the function ”`(_)first`” has been defined for these combinations of types. For more on how conditions, propositions and ad hoc polymorphism works, see the ”Type Logic” section (5.2).

- *Grammar*

  ⟨*condition*⟩ ::= ⟨*prop-name*⟩ ‘ `->` ’

5.1.2 Type Definitions

Type definitions are divided into `tuple_type` definitions and `or_type` definitions which are described in the following paragraphs.

The grammar of a type definition is:

⟨*type-def*⟩ ::= ⟨*tuple-type-def*⟩ | ⟨*or-type-def*⟩

Tuple Types

- *Definition Examples*

```
tuple_type Name
value (first_name, last_name) : String^2

tuple_type Date
value (day, month, year) : Int^3

tuple_type MathematicianInfo
value (name, nationality, date_of_birth) : Name x String x Date

tuple_type TreeOf(T1)s
value (root, subtrees) : T1 x ListOf(TreeOf(T1)s)s

tuple_type Indexed(T1)
value (index, val) : Int x T1
```

- *Usage Examples*

```
euler_info: MathematicianInfo
  = (("Leonhard", "Euler"), "Swiss", (15, 4, 1707))

name(_)to_string: Name => String
  = n => "\nFirst Name: " + n.first_name + "\nLast Name: " + n.last_name

print_name_and_nat(_): MathematicianInfo => IO
  = ci => print(name(ci.name)to_string + "\nNationality: " + ci.nationality)

sum_nodes(_): TreeOf(Int)s => Int
  = tree => tree.root + apply(sum_nodes(_))to_all_in(tree.subtrees) -> sum_list(_)
```

- *Description*

  A tuple type is equivalent to a product type with a new name and names for the fields for convenience. A tuple type generates postfix functions for all of the fields by using a '.' before the name of the field. For example the "`MathematicianInfo`" type above generates the following functions:

```
_.name : MathematicianInfo => Name
_.nationality : MathematicianInfo => String
_.date_of_birth : MathematicianInfo => Date
```

- *Grammar*

⟨*tuple-type-def*⟩ ::=
    ‘`tuple_type ’` ⟨*type-name*⟩ ⟨*nl*⟩
    ‘`value’` (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*id-tuple*⟩ [‘ ’] ‘`:`’ [‘ ’] (⟨*prod-type*⟩ | ⟨*power-type*⟩)

⟨*type-name*⟩ ::=
    [⟨*param-vars-in-paren*⟩] ⟨*type-id*⟩ (⟨*param-vars-in-paren*⟩ [A-Za-z]+)*
    [⟨*param-vars-in-paren*⟩]

⟨*param-vars-in-paren*⟩ ::= ‘`(`’ [‘ ’] ⟨*param-t-var*⟩ (⟨*comma*⟩ ⟨*param-t-var*⟩)* [‘ ’] ‘`)`’

⟨*id-tuple*⟩ ::= ‘`(`’ [‘ ’] ⟨*simple-id*⟩ (⟨*comma*⟩ ⟨*simple-id*⟩)+ [‘ ’] ‘`)`’

Or Types

- *Definition Examples*

```
or_type Bool
values true | false

or_type TrafficLight
values green | amber | red

or_type Possibly(T1)
values the_value:T1 | no_value

or_type Result(T1)OrError(T2)
values result:T1 | error:T2
```

- *Usage Examples*

```
traffic_lights_match(_, _): TrafficLight^2 => Bool
  = (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false

err_if(_)is_no_value : Possibly(T1) => Result(T1)OrError(String)
  = cases
    no_value => error:”There is no value!”
    the_value:val => result:val

print_err(_)or_res(_): (@A)Has_Str_Rep --> Result(@A)OrError(String) => IO
  = cases
    result:r => print(”All good! The result is: ” + (r)to_string)
    error:e => print(”Error occured: ” + e)
```

- *Description*

  The values of an `or_type` are split into cases. Some cases have other values inside. The cases which have other values inside are followed by a colon and the type of the internal value. Similar syntax can be used for matching that particular case in a function using the ”cases” syntax. An `or_type` definition automatically creates prefix functions for each case with an internal value (which are simply conversions from the type of the internal value to the `or_type`). For example, for the case ”`the_value`” of a ”`Possibly(T1)`” the function ”`the_value:_`” is automatically created from the definition for which we can say:

```
the_value:_ : T1 => Possibly(T1)
```

  These functions can be used like any other function as arguments to other functions. For example:

```
(_)to_possiblies : ListOf(T1)s => ListOf(Possibly(T1))s
  = apply(the_value:_)to_all_in(_)
```

- *Grammar*

  ⟨*or-type-def*⟩ ::=
  ‘`or_type` ’ ⟨*type-name*⟩ ⟨*nl*⟩
  ‘`values`’ (‘ ’ | ⟨*nl*⟩ ‘ ’)
  ⟨*simple-id*⟩ [‘`:`’ ⟨*simple-type*⟩] ([‘ ’] ‘|’ [‘ ’] ⟨*simple-id*⟩ [‘`:`’ ⟨*simple-type*⟩])*

5.1.3 Type Nicknames

- *Examples*

```
type_nickname Ints = ListOf(Int)s
type_nickname IntStringPairs = ListOf(Int x String)s
type_nickname IO = (EmptyVal)FromIO
type_nickname Res(T1)OrErr = Result(T1)OrError(String)
```

- *Description*

  Type nicknames are used to abbreviate or give a more descriptive name to a type. They start with the keyword ”`type_nickname`”, followed by the nickname, then an equal sign and the type to be nicknamed. Parametric type variables can be used in the nickname.

- *Grammar*

  ⟨*t-nickname*⟩ ::= ‘`type_nickname` ’ ⟨*type-name*⟩ ⟨*equals*⟩ ⟨*simple-type*⟩

5.2 Type Logic

Type logic is the mechanism for ad hoc polymorphism in lcases. The central notion of **type logic** is the **type proposition**. A type proposition is a proposition that has types as parameters and is true or false for particular type arguments.

Type propositions can either be defined or proven (for certain type arguments). Therefore, the following constructs exist and accomplish the aforementioned respectively: **type proposition definitions** and **type theorems**. These constructs are described in detail in the following sections. From this point onwards the ”type” part will be omitted, i.e. propositions are always type propositions and theorems are always type theorems.

5.2.1 Proposition Definitions

Proposition definitions are split into definitions of **atomic propositions** and definitions of **renaming propositions** which are described in the following paragraphs.

Atomic Propositions

- *Examples*

```
type_proposition (@A)Is(@B)s_First
needed (_)first: @B => @A

type_proposition (@T)Has_Str_Rep
needed (_)to_string: @T => String

type_proposition (@T)Has_A_Wrapper
needed wrap(_): T1 => @T(T1)

type_proposition (@T)Has_Internal_App
needed apply(_)inside(_) : (T1 => T2) x @T(T1) => @T(T2)
```

The examples above define the following (ad hoc) polymorphic functions which have the respective (conditional) types:

```
(_)first: (@A)Is(@B)s_First --> @B => @A

(_)to_string: (@T)Has_Str_Rep --> @T => String

wrap(_): (@T)Has_A_Wrapper --> T1 => @T(T1)

apply(_)inside(_): (@T)Has_Internal_App --> (T1 => T2) x @T(T1) => @T(T2)
```

- *Description*

  An atomic proposition definition defines simultaneously the **atomic proposition** itself and a **polymorphic value** (usually, but not necessarily, a function), by defining the form of the type of the value given the type parameters of the proposition. The proposition is true or false when the type parameters are substituted by specific type arguments depending on whether the implementation of the value has been defined for these type arguments. The aforementioned truth value determines whether the value is used correctly inside the program and therefore whether the program will typecheck. In order to add more types for which the function works, i.e. define the function for these types, i.e. make the proposition true for these types, one must prove a theorem. The specifics of theorems are described in the next section. For now, we'll show the example for everything mentioned in this paragraph for the proposition "`(@A)Is(@B)s_First`":

  – Proposition Definition:

  ```
  type_proposition (@A)Is(@B)s_First
  needed (_)first: @B => @A
  ```

  – Function defined and its type:

  ```
  (_)first: (@A)Is(@B)s_First --> @B => @A
  ```

  – Theorems for specific types:

  ```
  type_theorem (T1)Is(T1 x T2)s_First
  proof (_)first = _.1st

  type_theorem (T1)Is(ListOf(T1)s)s_First
  proof
    (_)first =
      cases
        [] => throw_err("Tried to take the first element of an empty list")
        [head, ...] => head
  ```

  – Usage of the function

  ```
  pair, list
    : Int x String, ListOf(String)s
    = (42, "The answer to everything"), ["Hi!", "Hello", Heeey"]

  >> (pair)first
    : Int
    ==> 42
  >> (list)first
    : String
    ==> "Hi!"
  ```

  An atomic proposition definition begins with the keyword "`type_proposition`" followed by the name of the proposition (including the type parameters) in the first line. The second line begins with the keyword "`needed`" which is followed by the identifier and the type expression of the value separated by the "has type" symbol ('`:`').

Renaming Propositions

- *Examples*

```
type_proposition (@T)Has_Equality
equivalent (@T)And(@T)Can_Be_Equal

type_proposition (@A)And(@B)Are_Comparable
equivalent
  (@A)Can_Be_Less_Than(@B), (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_proposition (@T)Has_Comparison
equivalent (@T)And(@T)Are_Comparable
```

- *Description*

  A renaming proposition definition is used to abbreviate one or the conjunction of many propositions into a new proposition.

  A renaming proposition definition begins with the keyword ”`type_proposition`” followed by the name of the proposition (including the type parameters) in the first line. The second line begins with the keyword ”`equivalent`” followed by either one proposition or (if it is a conjunction) many propositions separated by commas (where the commas mean ”and”).

Grammar for Proposition Definitions

⟨*type-prop-def*⟩ ::= ⟨*atom-prop-def*⟩ | ⟨*renaming-prop-def*⟩

⟨*atom-prop-def*⟩ ::=
⟨*prop-name-line*⟩ ⟨*nl*⟩ ‘`needed`’ (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*identifier*⟩ [‘ ’] ‘`:`’ [‘ ’] ⟨*simple-type*⟩

⟨*renaming-prop-def*⟩ ::=
⟨*prop-name-line*⟩ ⟨*nl*⟩ ‘`equivalent`’ (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*prop-name*⟩ (⟨*comma*⟩ ⟨*prop-name*⟩)*

⟨*prop-name-line*⟩ ::= ‘`type_proposition `’ ⟨*prop-name*⟩

⟨*prop-name*⟩ ::=
[A-Z] (⟨*name-part*⟩ ⟨*types-in-paren*⟩)+ [⟨*name-part*⟩]
| (⟨*types-in-paren*⟩ ⟨*name-part*⟩)+ [⟨*types-in-paren*⟩]

⟨*name-part*⟩ ::= ([A-Za-z] | ‘_’[A-Z])+

5.2.2 Theorems

Theorems are split into theorems of **atomic propositions** and theorems of **implication propositions** which are described in the following paragraphs.

Atomic Propositions

- *Examples*

```
type_theorem (Possibly(_))Has_A_Wrapper
proof wrap(_) = the_value:_

type_theorem (ListOf(_)s)Has_A_Wrapper
proof wrap(_) = [_]

type_theorem (Possibly(_))Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      no_value => no_value
      the_value:x => the_value:f(x)

type_theorem (ListOf(_)s)Has_Internal_App
proof apply(_)inside(_) = apply(_)to_all_in(_)
```

- *Usage*

```
a, b : all Possibly(Int)
  = wrapper(1), no_value

l1, l2, l3 : all ListOf(Int)s
  = wrapper(1), empty_l, [1, 2, 3]

>> a
  : Possibly(Int)
  ==> the_value:1
>> b
  : Possibly(Int)
  ==> no_value
>> l1
  : ListOf(Int)s
  ==> [1]
>> l2
  : ListOf(Int)s
  ==> []
```

```
>> apply(_ + 1)inside(a)
  : Possibly(Int)
  ==> the_value:2
>> apply(_ + 1)inside(b)
  : Possibly(Int)
  ==> no_value
>> apply(_ + 1)inside(l1)
  : ListOf(Int)s
  ==> [2]
>> apply(_ + 1)inside(l2)
  : ListOf(Int)s
  ==> []
>> apply(_ + 1)inside(l3)
  : ListOf(Int)s
  ==> [2, 3, 4]
```

- *Description*

  A theorem of an atomic proposition proves the proposition for specific type arguments, by implementing the value associated to the proposition for these type arguments. Therefore, the value associated with the proposition can be used with all the combinations of type arguments for which the proposition is true, i.e. the combinations of type arguments for which the value has been implemented.

  A proof of a theorem of an atomic proposition is correct when the implementation of the value associated with the proposition follows the form of the type given to the value by the definition of the proposition, i.e. the only difference between the type of the value in the theorem and the type of the value in the definition is that the type parameters of the proposition are substituted by the type arguments of the theorem.

  A theorem of an atomic proposition begins with the keyword ”`type_theorem`” followed by the name of the proposition with the type parameters substituted by the specific types for which the proposition will be proven. The second line is the keyword ”`proof`”. The third line is indented once and it is the line in which the proof begins. The proof begins with the identifier of the value associated with the proposition and is followed by an equal sign and the value expression which implements the value.

Implication Propositions

- *Examples*

```
type_theorem (@A)And(@B)Can_Be_Equal --> (@A)And(@B)Can_Be_Unequal
proof a \= b = not(a == b)

type_theorem (@A)Can_Be_Greater_Than(@B) --> (@A)Can_Be_Le_Or_Eq_To(@B)
proof a <= b = not(a > b)

type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@A)And(@B)Have_Eq_And_Gr --> (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
```

- *Description*

  A theorem of an implication proposition is similar to a theorem of an atomic proposition but it also uses other ad hoc polymorphic values in the implementation. Therefore, the implementation does not prove the proposition associated to the value it implements unless the polymorphic values used in the implementation are already defined in their own theorems. In other words it proves the following: ”if these ad hoc polymorphic values are defined then we can also define this other one”. This can be translated into the following implication proposition: ”if the propositions associated to the values we are using are true then the proposition associated to the value we are defining is true”, which can be condensed to the notation with the condition arrow (” `-->` ”) used in the examples.

  When we are using many ad hoc values in the implementation we need to group all their propositions into one conjunction proposition with a renaming proposition definition as it is done in the example of
  `(@A)And(@B)Have_Eq_And_Gr`.

  The proof of an implication proposition allows the compiler to automatically create the definition for an ad hoc polymorphic value for a particular combination of types given the definitions of the ad hoc polymorphic values used in the implementation for this same combination of types. This mechanism essentially gives definitions for free, that is in the sense that when you define a set of ad hoc polymorphic values for a particular set of types you get for free all the ad hoc polymorphic values that can be defined using a subset of the defined ones.

  A theorem of an implication proposition is grammatically the same as a theorem of an atomic proposition with the only difference being that an implication proposition is comprised by two atomic propositions separated by the condition arrow (” `-->` ”) arrow.

Grammar for Theorems

⟨*type-theo*⟩ ::=
    ‘`type_theorem` ’ ⟨*prop-name-with-subs*⟩ [‘ `-->` ’ ⟨*prop-name-with-subs*⟩] ⟨*nl*⟩ ‘`proof`’ ⟨*proof*⟩

⟨*prop-name-with-subs*⟩ ::=
    [A-Z] (⟨*name-part*⟩ ⟨*subs-in-paren*⟩)+ [⟨*name-part*⟩]
  | (⟨*subs-in-paren*⟩ ⟨*name-part*⟩)+ [⟨*subs-in-paren*⟩]

⟨*subs-in-paren*⟩ ::= ‘(’ [‘ ’] ⟨*t-var-sub*⟩ (⟨*comma*⟩ ⟨*t-var-sub*⟩)* [‘ ’] ‘)’

⟨*t-var-sub*⟩ ::= ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩

⟨*type-app-id-or-ahtv-sub*⟩ ::=
    [⟨*subs-or-unders-in-paren*⟩] ⟨*taioas-middle*⟩ [⟨*subs-or-unders-in-paren*⟩]

⟨*taioas-middle*⟩ ::= ⟨*type-id*⟩ (⟨*subs-or-unders-in-paren*⟩ [A-Za-z]+)* | ⟨*ad-hoc-t-var*⟩

⟨*subs-or-unders-in-paren*⟩ ::= ‘(’ [‘ ’] ⟨*sub-or-under*⟩ (⟨*comma*⟩ ⟨*sub-or-under*⟩)* [‘ ’] ‘)’

⟨*sub-or-under*⟩ ::= ⟨*t-var-sub*⟩ | ‘_’

⟨*power-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ ‘^’ ⟨*int-greater-than-one*⟩

⟨*power-base-type-sub*⟩ ::=
    ‘_’ | ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ |
    ‘(’ [‘ ’] (⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩) [‘ ’] ‘)’

⟨*prod-type-sub*⟩ ::= ⟨*field-type-sub*⟩ (‘ `x` ’ ⟨*field-type-sub*⟩)+

⟨*field-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ | ⟨*power-type-sub*⟩

⟨*func-type-sub*⟩ ::= ⟨*in-or-out-type-sub*⟩ ‘ `=>` ’ ⟨*in-or-out-type-sub*⟩

⟨*in-or-out-type-sub*⟩ ::=
    ‘_’ | ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ |
    ‘(’ [‘ ’] ⟨*func-type-sub*⟩ [‘ ’] ‘)’

⟨*proof*⟩ ::= ‘ ’ ⟨*id-or-op-eq*⟩ ‘ ’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ‘ ’ ⟨*id-or-op-eq*⟩ ⟨*tt-value-expr*⟩

⟨*id-or-op-eq*⟩ ::= ⟨*identifier*⟩ [⟨*op*⟩ ⟨*identifier*⟩] ‘ =’

⟨*tt-value-expr*⟩ ::= ‘ ’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ⟨*indent*⟩ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

Chapter 6

Language Description: Predefined

6.1 Values

- Constants: `undefined, pi, empty_val`
- Functions
  - Miscellaneous: `not(_), id(_), throw_err(_)`
  - Numerical:
    - Miscellaneous:
      `sqrt_of(_), abs_val_of(_), max_of(_)and(_), min_of(_)and(_)`
    - Trigonometric:
      `sin(_), cos(_), tan(_), asin(_), acos(_), atan(_)`
    - Division related:
      `(_)div(_), (_)mod(_), gcd_of(_)and(_), lcm_of(_)and(_),`
      `(_)is_even, (_)is_odd`
    - Rounding:
      `truncate(_), round(_), floor(_), ceiling(_)`
    - e and log:
      `exp(_), ln(_), log_of(_)base(_)`
  - List:
    `(_)length, (_)is_in(_), apply(_)to_all_in(_), filter(_)with(_),`
    `take(_)from(_), ignore(_)from(_), split(_)at(_),`
    `zip(_)with(_), unzip(_), apply(_)to_all_in_zipped(_,_)`
  - IO:
    - Input: `get_char, get_line, get_input, read_file(_)`
    - Output: `print(_), print_string(_), write(_)in_file(_)`
  - Ad Hoc Polymorphic:
    `wrap(_), (_)to_string, from_string(_), apply(_)inside(_),`

6.2 Types

- Basic: `Int, Real, Char, String, (_)FromIO, (_)FState(_)Man`
- Or Types: `EmptyVal, Bool, Possibly(_), ListOf(_)s, Result(_)OrError(_)`
- Type Nicknames: `IO, State(_)Man, Z, R`

6.3 Type Propositions

- Operator Propositions:
  - `(@A)To_The(@B)Is(@C)`
  - `(@A)And(@B)Multiply_To(@C)`
  - `(@A)Divided_By(@B)Is(@C)`
  - `(@A)And(@B)Add_To(@C)`
  - `(@A)Minus(@B)Is(@C)`
  - `(@A)And(@B)Can_Be_Equal`
  - `(@A)And(@B)Can_Be_Unequal`
  - `(@A)Can_Be_Gr_Or_Eq_To(@B)`
  - `(@A)Can_Be_Le_Or_Eq_To(@B)`
  - `(@A)Can_Be_Greater_Than(@B)`
  - `(@A)Can_Be_Less_Than(@B)`
  - `(@A)Has_And`
  - `(@A)Has_Or`
  - `(@T)Has_Use`
  - `(@T)Has_Then`
- Function Propositions:
  - `(@T)Has_A_Wrapper`
  - `(@T)Has_Str_Rep`
  - `(@T)Has_Internal_App`

- Theorems:
  - `(Possibly(_))Has_A_Wrapper`
  - `(ListOf(_)s)Has_A_Wrapper`
  - `((_)FState(T1)Man)Has_A_Wrapper`
  - `(Result(_)OrError(T1))Has_A_Wrapper`
  - `(Int)Has_Str_Rep`
  - `(Char)Has_Str_Rep`
  - `(Real)Has_Str_Rep`
  - `(@A)Has_Str_Rep --> (ListOf(@A)s)Has_Str_Rep`
  - `(Possibly(_))Has_Internal_App`
  - `(ListOf(_)s)Has_Internal_App`
  - `((_)FState(T1)Man)Has_Internal_App`
  - `(Result(_)OrError(T1))Has_Internal_App`
  - `((_)FromIO)Has_Use`
  - `((_)FState(T1)Man)Has_Use`
  - `((_)FromIO)Has_Then`
  - `((_)FState(T1)Man)Has_Then`

Chapter 7

Parser Implementation

7.1 Full Grammar and Indentation System

7.1.1 Full Grammar

⟨*literal*⟩ ::= ⟨*int-lit*⟩ | ⟨*real-lit*⟩ | ⟨*char-lit*⟩ | ⟨*string-lit*⟩

⟨*identifier*⟩ ::= [⟨*unders-in-paren*⟩] ⟨*id-start*⟩ ⟨*id-cont*⟩* [[0-9]] [⟨*unders-in-paren*⟩]

⟨*simple-id*⟩ ::= ⟨*id-start*⟩ [[0-9]]

⟨*id-start*⟩ ::= [a-z] [a-z_]*

⟨*id-cont*⟩ ::= ⟨*unders-in-paren*⟩ [a-z_]+

⟨*unders-in-paren*⟩ ::= ‘(_’ (⟨*comma*⟩ ‘_’)* ‘)’

⟨*comma*⟩ ::= ‘,’ [‘ ’]

⟨*paren-expr*⟩ ::= ‘(’ [‘ ’] ⟨*line-op-expr*⟩ | ⟨*line-func-expr*⟩ [‘ ’] ‘)’

⟨*tuple*⟩ ::= ‘(’ [‘ ’] ⟨*line-expr-or-under*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩ [‘ ’] ‘)’

⟨*line-expr-or-unders*⟩ ::= ⟨*line-expr-or-under*⟩ (⟨*comma*⟩ ⟨*line-expr-or-under*⟩)*

⟨*line-expr-or-under*⟩ ::= ⟨*line-expr*⟩ | ‘_’

⟨*line-expr*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ | ⟨*line-func-expr*⟩

⟨*basic-or-app-expr*⟩ ::= ⟨*basic-expr*⟩ | ⟨*pre-func-app*⟩ | ⟨*post-func-app*⟩

⟨*basic-expr*⟩ ::= ⟨*literal*⟩ | ⟨*paren-func-app-or-id*⟩ | ⟨*special-id*⟩ | ⟨*tuple*⟩ | ⟨*list*⟩

⟨*big-tuple*⟩ ::=
‘(’ [‘ ’] ⟨*line-expr-or-under*⟩ [⟨*nl*⟩ ⟨*indent*⟩] ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩
(⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩)* ⟨*nl*⟩ ⟨*indent*⟩ ‘)’

⟨*list*⟩ ::= ‘[’ [‘ ’] [⟨*line-expr-or-unders*⟩] [‘ ’] ‘]’

⟨*big-list*⟩ ::=
‘[’ [‘ ’] ⟨*line-expr-or-unders*⟩ (⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-expr-or-unders*⟩)* ⟨*nl*⟩ ⟨*indent*⟩ ‘]’

⟨*paren-func-app-or-id*⟩ ::=
[⟨*arguments*⟩] ⟨*id-start*⟩ (⟨*arguments*⟩ [a-z_]+)* [[0-9]] [⟨*arguments*⟩]

⟨*arguments*⟩ ::= ‘(’ [‘ ’] ⟨*line-expr-or-unders*⟩ [‘ ’] ‘)’

⟨*pre-func*⟩ ::= ⟨*simple-id*⟩ ‘:’

⟨*pre-func-app*⟩ ::= ⟨*pre-func*⟩ ⟨*operand*⟩

⟨*post-func*⟩ ::= ‘.’ (⟨*simple-id*⟩ | ⟨*special-id*⟩)

⟨*special-id*⟩ ::= ‘1st’ | ‘2nd’ | ‘3rd’ | ‘4th’ | ‘5th’

⟨*post-func-app*⟩ ::= (⟨*basic-expr*⟩ | ⟨*paren-expr*⟩ | ‘_’) (⟨*dot-change*⟩ | ⟨*post-func*⟩+ [⟨*dot-change*⟩])

⟨*dot-change*⟩ ::= ‘.change{’ [‘ ’] ⟨*field-change*⟩ (⟨*comma*⟩ ⟨*field-change*⟩)* [‘ ’] ‘}’

⟨*field-change*⟩ ::= (⟨*simple-id*⟩ | ⟨*special-id*⟩) ⟨*equals*⟩ ⟨*line-expr-or-under*⟩

⟨*equals*⟩ ::= [‘ ’] ‘=’ [‘ ’]

⟨*op-expr*⟩ ::= ⟨*line-op-expr*⟩ | ⟨*big-op-expr*⟩

⟨*op-expr-start*⟩ ::= (⟨*operand*⟩ ⟨*op*⟩)+

⟨*line-op-expr*⟩ ::= ⟨*op-expr-start*⟩ (⟨*operand*⟩ | ⟨*line-func-expr*⟩)

⟨*big-op-expr*⟩ ::= ⟨*big-op-expr-op-split*⟩ | ⟨*big-op-expr-func-split*⟩

⟨*big-op-expr-op-split*⟩ ::= ⟨*op-split-line*⟩+ [⟨*op-expr-start*⟩] (⟨*operand*⟩ | ⟨*func-expr*⟩)

⟨*op-split-line*⟩ ::= (⟨*op-expr-start*⟩ (⟨*nl*⟩ | ⟨*oper-fco*⟩) | ⟨*oper-fco*⟩) ⟨*indent*⟩

⟨*oper-fco*⟩ ::= ⟨*operand*⟩ ‘ ’ ⟨*func-comp-op*⟩ ‘\n’

⟨*big-op-expr-func-split*⟩ ::= ⟨*op-expr-start*⟩ (⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩)

⟨*operand*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*paren-expr*⟩ | ‘_’

⟨*op*⟩ ::= ‘ ’ ⟨*func-comp-op*⟩ ‘ ’ | [‘ ’] ⟨*optional-spaces-op*⟩ [‘ ’]

⟨*func-comp-op*⟩ ::= ‘o>’ | ‘<o’

⟨*optional-spaces-op*⟩ ::=
‘->’ | ‘<-’ | ‘^’ | ‘*’ | ‘/’ | ‘+’ | ‘-’ | ‘==’ | ‘!=’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘&’ | ‘|’ | ‘;>’ | ‘;’

⟨*func-expr*⟩ ::= ⟨*line-func-expr*⟩ | ⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩

⟨*line-func-expr*⟩ ::= ⟨*parameters*⟩ [‘ ’] ‘=>’ ⟨*line-func-body*⟩

⟨*big-func-expr*⟩ ::= ⟨*parameters*⟩ [‘ ’] ‘=>’ ⟨*big-func-body*⟩

⟨*parameters*⟩ ::= ⟨*identifier*⟩ | ‘*’ | ‘(’ [‘ ’] ⟨*parameters*⟩ (⟨*comma*⟩ ⟨*parameters*⟩)+ [‘ ’] ‘)’

⟨*line-func-body*⟩ ::=
[‘ ’] (⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ | ‘(’ [‘ ’] ⟨*line-func-expr*⟩ [‘ ’] ‘)’)

⟨*big-func-body*⟩ ::=
⟨*nl*⟩ ⟨*indent*⟩ (⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ | ‘(’ [‘ ’] ⟨*line-func-expr*⟩ [‘ ’] ‘)’)

⟨*cases-func-expr*⟩ ::= ⟨*cases-params*⟩ ⟨*case*⟩+ [⟨*end-case*⟩]

⟨*cases-params*⟩ ::=
⟨*identifier*⟩ | ‘cases’ | ‘*’ | ‘(’ [‘ ’] ⟨*cases-params*⟩ (⟨*comma*⟩ ⟨*cases-params*⟩)+ [‘ ’] ‘)’

⟨*case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ⟨*outer-matching*⟩ [‘ ’] ‘=>’ ⟨*case-body*⟩

⟨*end-case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ (‘...’ | ⟨*identifier*⟩) [‘ ’] ‘=>’ ⟨*case-body*⟩

⟨*outer-matching*⟩ ::= ⟨*simple-id*⟩ | ⟨*matching*⟩

⟨*matching*⟩ ::= ⟨*literal*⟩ | ⟨*pre-func*⟩ ⟨*inner-matching*⟩ | ⟨*tuple-matching*⟩ | ⟨*list-matching*⟩

⟨*inner-matching*⟩ ::= ‘*’ | ⟨*identifier*⟩ | ⟨*matching*⟩

⟨*tuple-matching*⟩ ::= ‘(’ [‘ ’] ⟨*inner-matching*⟩ (⟨*comma*⟩ ⟨*inner-matching*⟩)+ [‘ ’] ‘)’

⟨*list-matching*⟩ ::=
‘[’ [‘ ’] [⟨*inner-matching*⟩ (⟨*comma*⟩ ⟨*inner-matching*⟩)* [⟨*rest-list-matching*⟩]] [‘ ’] ‘]’

⟨*rest-list-matching*⟩ ::= ⟨*comma*⟩ [⟨*simple-id*⟩ ⟨*equals*⟩] ‘...’

⟨*case-body*⟩ ::= ⟨*line-func-body*⟩ | ⟨*big-func-body*⟩ [⟨*where-expr*⟩]

⟨*value-def*⟩ ::=
    ⟨*indent*⟩ ⟨*identifier*⟩ ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’) ⟨*type*⟩
    ⟨*nl*⟩ ⟨*indent*⟩ ‘= ’ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

⟨*value-expr*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ | ⟨*func-expr*⟩ | ⟨*big-tuple*⟩ | ⟨*big-list*⟩

⟨*grouped-value-defs*⟩ ::=
    ⟨*indent*⟩ ⟨*identifier*⟩ (⟨*comma*⟩ ⟨*identifier*⟩)+
    ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’) (⟨*type*⟩ (⟨*comma*⟩ ⟨*type*⟩)+ | ‘all ’ ⟨*type*⟩)
    ⟨*nl*⟩ ⟨*indent*⟩ ‘= ’ ⟨*line-exprs*⟩ (⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-exprs*⟩)*

⟨*line-exprs*⟩ ::= ⟨*line-expr*⟩ (⟨*comma*⟩ ⟨*line-expr*⟩)*

⟨*where-expr*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ‘where’ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩ (⟨*nl*⟩ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩)*

⟨*value-def-or-defs*⟩ ::= ⟨*value-def*⟩ | ⟨*grouped-value-defs*⟩

⟨*type*⟩ ::= [⟨*condition*⟩] ⟨*simple-type*⟩

⟨*simple-type*⟩ ::= ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | ⟨*func-type*⟩

⟨*type-id*⟩ ::= [A-Z] [A-Za-z]*

⟨*param-t-var*⟩ ::= ‘T’ [0-9]

⟨*ad-hoc-t-var*⟩ ::= ‘@’ [A-Z]

⟨*type-app-id-or-ahtv*⟩ ::= [⟨*types-in-paren*⟩] ⟨*taioa-middle*⟩ [⟨*types-in-paren*⟩]

⟨*taioa-middle*⟩ ::= ⟨*type-id*⟩ (⟨*types-in-paren*⟩ [A-Za-z]+)* | ⟨*ad-hoc-t-var*⟩

⟨*types-in-paren*⟩ ::= ‘(’ [‘ ’] ⟨*simple-type*⟩ (⟨*comma*⟩ ⟨*simple-type*⟩)* [‘ ’] ‘)’

⟨*prod-type*⟩ ::= ⟨*field-type*⟩ (‘ x ’ ⟨*field-type*⟩)+

⟨*field-type*⟩ ::= ⟨*power-base-type*⟩ | ⟨*power-type*⟩

⟨*power-base-type*⟩ ::=
    ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | ‘(’ [‘ ’] (⟨*prod-type*⟩ | ⟨*func-type*⟩) [‘ ’] ‘)’

⟨*power-type*⟩ ::= ⟨*power-base-type*⟩ ‘^’ ⟨*int-greater-than-one*⟩

⟨*func-type*⟩ ::= ⟨*in-or-out-type*⟩ ‘ => ’ ⟨*in-or-out-type*⟩

⟨*in-or-out-type*⟩ ::=
    ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | ‘(’ [‘ ’] ⟨*func-type*⟩ [‘ ’] ‘)’

⟨*condition*⟩ ::= ⟨*prop-name*⟩ ‘ `->` ’

⟨*type-def*⟩ ::= ⟨*tuple-type-def*⟩ | ⟨*or-type-def*⟩

⟨*tuple-type-def*⟩ ::=
‘`tuple_type` ’ ⟨*type-name*⟩ ⟨*nl*⟩
‘`value`’ (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*id-tuple*⟩ [‘ ’] ‘`:`’ [‘ ’] (⟨*prod-type*⟩ | ⟨*power-type*⟩)

⟨*type-name*⟩ ::=
[⟨*param-vars-in-paren*⟩] ⟨*type-id*⟩ (⟨*param-vars-in-paren*⟩ [A-Za-z]+)*
[⟨*param-vars-in-paren*⟩]

⟨*param-vars-in-paren*⟩ ::= ‘`(`’ [‘ ’] ⟨*param-t-var*⟩ (⟨*comma*⟩ ⟨*param-t-var*⟩)* [‘ ’] ‘`)`’

⟨*id-tuple*⟩ ::= ‘`(`’ [‘ ’] ⟨*simple-id*⟩ (⟨*comma*⟩ ⟨*simple-id*⟩)+ [‘ ’] ‘`)`’

⟨*or-type-def*⟩ ::=
‘`or_type` ’ ⟨*type-name*⟩ ⟨*nl*⟩
‘`values`’ (‘ ’ | ⟨*nl*⟩ ‘ ’)
⟨*simple-id*⟩ [‘`:`’ ⟨*simple-type*⟩] ([‘ ’] ‘|’ [‘ ’] ⟨*simple-id*⟩ [‘`:`’ ⟨*simple-type*⟩])*

⟨*t-nickname*⟩ ::= ‘`type_nickname` ’ ⟨*type-name*⟩ ⟨*equals*⟩ ⟨*simple-type*⟩

⟨*type-prop-def*⟩ ::= ⟨*atom-prop-def*⟩ | ⟨*renaming-prop-def*⟩

⟨*atom-prop-def*⟩ ::=
⟨*prop-name-line*⟩ ⟨*nl*⟩ ‘`needed`’ (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*identifier*⟩ [‘ ’] ‘`:`’ [‘ ’] ⟨*simple-type*⟩

⟨*renaming-prop-def*⟩ ::=
⟨*prop-name-line*⟩ ⟨*nl*⟩ ‘`equivalent`’ (‘ ’ | ⟨*nl*⟩ ‘ ’) ⟨*prop-name*⟩ (⟨*comma*⟩ ⟨*prop-name*⟩)*

⟨*prop-name-line*⟩ ::= ‘`type_proposition` ’ ⟨*prop-name*⟩

⟨*prop-name*⟩ ::=
[A-Z] (⟨*name-part*⟩ ⟨*types-in-paren*⟩)+ [⟨*name-part*⟩]
| (⟨*types-in-paren*⟩ ⟨*name-part*⟩)+ [⟨*types-in-paren*⟩]

⟨*name-part*⟩ ::= ([A-Za-z] | ‘_’[A-Z])+

⟨*type-theo*⟩ ::=
    ‘`type_theorem` ’ ⟨*prop-name-with-subs*⟩ [‘ `->` ’ ⟨*prop-name-with-subs*⟩] ⟨*nl*⟩ ‘`proof`’ ⟨*proof*⟩

⟨*prop-name-with-subs*⟩ ::=
    [A-Z] (⟨*name-part*⟩ ⟨*subs-in-paren*⟩)+ [⟨*name-part*⟩]
  | (⟨*subs-in-paren*⟩ ⟨*name-part*⟩)+ [⟨*subs-in-paren*⟩]

⟨*subs-in-paren*⟩ ::= ‘`(`’ [‘ ’] ⟨*t-var-sub*⟩ (⟨*comma*⟩ ⟨*t-var-sub*⟩)* [‘ ’] ‘`)`’

⟨*t-var-sub*⟩ ::= ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩

⟨*type-app-id-or-ahtv-sub*⟩ ::=
    [⟨*subs-or-unders-in-paren*⟩] ⟨*taioas-middle*⟩ [⟨*subs-or-unders-in-paren*⟩]

⟨*taioas-middle*⟩ ::= ⟨*type-id*⟩ (⟨*subs-or-unders-in-paren*⟩ [A-Za-z]+)* | ⟨*ad-hoc-t-var*⟩

⟨*subs-or-unders-in-paren*⟩ ::= ‘`(`’ [‘ ’] ⟨*sub-or-under*⟩ (⟨*comma*⟩ ⟨*sub-or-under*⟩)* [‘ ’] ‘`)`’

⟨*sub-or-under*⟩ ::= ⟨*t-var-sub*⟩ | ‘_’

⟨*power-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ ‘`^`’ ⟨*int-greater-than-one*⟩

⟨*power-base-type-sub*⟩ ::=
    ‘_’ | ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ |
    ‘`(`’ [‘ ’] (⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩) [‘ ’] ‘`)`’

⟨*prod-type-sub*⟩ ::= ⟨*field-type-sub*⟩ (‘ `x` ’ ⟨*field-type-sub*⟩)+

⟨*field-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ | ⟨*power-type-sub*⟩

⟨*func-type-sub*⟩ ::= ⟨*in-or-out-type-sub*⟩ ‘ `=>` ’ ⟨*in-or-out-type-sub*⟩

⟨*in-or-out-type-sub*⟩ ::=
    ‘_’ | ⟨*param-t-var*⟩ | ⟨*type-app-id-or-ahtv-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ |
    ‘`(`’ [‘ ’] ⟨*func-type-sub*⟩ [‘ ’] ‘`)`’

⟨*proof*⟩ ::= ‘ ’ ⟨*id-or-op-eq*⟩ ‘ ’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ‘ ’ ⟨*id-or-op-eq*⟩ ⟨*tt-value-expr*⟩

⟨*id-or-op-eq*⟩ ::= ⟨*identifier*⟩ [⟨*op*⟩ ⟨*identifier*⟩] ‘ `=`’

⟨*tt-value-expr*⟩ ::= ‘ ’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ⟨*indent*⟩ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

⟨*program*⟩ ::= ⟨*nl*⟩* ⟨*program-part*⟩ (⟨*nl*⟩ ⟨*nl*⟩ ⟨*program-part*⟩)* ⟨*nl*⟩*

⟨*program-part*⟩ ::=
    ⟨*value-def*⟩ | ⟨*grouped-value-defs*⟩ | ⟨*type-def*⟩ | ⟨*t-nickname*⟩ | ⟨*type-prop-def*⟩ | ⟨*type-theo*⟩

⟨*nl*⟩ :: (‘ ’ | ‘`\t`’)* ‘`\n`’

7.1.2 Indentation System

The <*indent*> non-terminal in not a normal BNF non-terminal. It is a context sensitive construct that enforces the indentation rules of lcases. It depends on a integer value called the "indentation level" (il). The <*indent*> non-terminal corresponds to $2 * il$ space characters. The indentation level follows the rules below:

Indentation Rules

1. At the beginning: $il = 0$
2. In a single value definition:
   (a) At the end of the first line: $il \leftarrow il + 1$
   (b) At the end of the "=" line: $il \leftarrow il + 1$
   (c) At the end: $il \leftarrow il - 2$
3. In a group of value definitions:
   (a) At the end of the first line: $il \leftarrow il + 1$
   (b) At the end: $il \leftarrow il - 1$
4. In a case (of a cases function expression):
   (a) After the arrow ("=>") line: $il \leftarrow il + 1$.
   (b) At the end: $il \leftarrow il - 1$.
5. In a type theorem:
   (a) After "=" line: $il \leftarrow il + 2$.
   (b) At the end: $il \leftarrow il - 2$.
6. In a cases function expression which does not begin at the "=" line of a value definition:
   (a) After the parameters line: $il \leftarrow il + 1$.
   (b) At the end of the cases function expression: $il \leftarrow il - 1$.

7.2 High Level Structure

7.2.1 Parsec Library

The parser was implemented using the **parsec** library [2]. Parsec is an industrial strength, monadic parser combinator library for Haskell. It can parse context-sensitive, infinite look-ahead grammars. It achieves this with a polymorphic parser type with the following parameter types:

- *stream type:* The input type to the parser.
- *user state type:* Type of custom state added by the parser developer.
- *underlying monad type:* A custom monad type in case it is needed.
- *return type:* This is the type of the value that is built by the parser.

The library has a lot of very nice parsers and parser combinators. The package description in hackage is in the following url: `https://hackage.haskell.org/package/parsec`

In this parser

- *stream type:*
  In this parser this is String.
- *state type:*
  In this parser this is ParserState. It is defined in the parser. A paragraph explaining what it is follows.
- *underlying monad:*
  In this parser this is not used interestingly (Identity is the underlying monad).
- *return type:*
  This is the type of the value that is built during parsing. Every AST type is the return type of the corresponding (sub)parser.

State type of the parser: ParserState

Here's the actual code for it:

```
type IndentationLevel = Int
type InEqualLine = Bool
type ParserState = (IndentationLevel, InEqualLine)
```

We need this state to enforce the indentation rules (of 7.1.2).

7.2.2 File Structure

The parser code is split into the following files:

- `ASTTypes.hs`: Definitions of abstract syntax tree types
- `ShowInstances.hs`: String representations for each AST type
- `Parsers.hs`: Parsers for each AST type

All of the above are written using the full grammar. The types correspond to non-terminal symbols. The parsers try to parse a string into the corresponding AST type. If the string is valid every terminal symbol is discarded unless it's part of a literal or an identifier.

7.3 Parser Examples

In this section we show how the types and the parsers are derived from the grammar with some examples. We begin with a grammar rule and we create the AST type and the parser that parses it.

7.3.1 Parser Class and Example 0: Literal

We have the Parser type which is polymorphic in the return type with a stream type of String and a state type of ParserState:

```
type Parser = Parsec String ParserState
```

We create the polymorphic value ”parser” with the ”HasParser” class so that all the parsers have the name ”parser” irrespective of the particular type they are parsing:

```
class HasParser a where
  parser :: Parser a
```

We begin with the very simple example of the literal with the following grammar rule:

⟨*literal*⟩ ::= ⟨*int-lit*⟩ | ⟨*real-lit*⟩ | ⟨*char-lit*⟩ | ⟨*string-lit*⟩

The AST type for the literal is:

```
data Literal =
  Int Integer | R Double | Ch Char | S String
```

And here is the parser for the literal which is defined as an instance of the HasParser class. Inside we use the parsers for each particular literal which are defined separately:

```
instance HasParser Literal where
  parser =
    R <$> try parser <|> Int <$> parser <|> Ch <$> parser <|> S <$> parser <?>
    ″Literal″
```

The Parser type is a Functor so the ”<$>” (fmap) operator passes each constructor inside the particular parser. The ”try” parser combinator does backtracking if the parser fails so that it does not consume any input. This is used if two parsers can parse the same input up to a point but the second one is the correct one. In this case a real number (a number with a decimal point) will parse an integer up to before the decimal point but will fail if there isn’t one. In this case we need to backtrack and let the integer parser consume the input. The ”<|>” operator means ”this parser or that parser”. Finally, the ”<?>” operator means ”if all the parsers fail show this error message”.

7.3.2 Example 1: List

The grammar rule for the list is the following:

⟨*list*⟩ ::= ‘`[`’ [‘ ’] [⟨*line-expr-or-unders*⟩] [‘ ’] ‘`]`’

The AST type for the list is:

```
newtype List = L (Maybe LineExprOrUnders)
```

And the parser for the list is:

```
instance HasParser List where
  parser =
    L <$> (char '[' *> opt_space_around (optionMaybe parser) <* char ']')
```

Here we use the ”`*>`” and ”`<*`” operators which parse both of the parsers that they have as operands (from left to right) but only keep the result of the parser that they ”point” to. ”opt_space_around” parses one space optionally on each side of the text parsed by the argument parser and returns what was parsed by the argument parser. ”optionMaybe” is defined in the library and it optionally parses what its argument parser parses. If the argument parser succeeds at parsing then it returns a Just <whatever was parsed>, whereas if it fails it returns Nothing.

7.3.3 Example 2: Change

The grammar rule for the ”change” expression is the following:

⟨*dot-change*⟩ ::= ‘`.change{`’ [‘ ’] ⟨*field-change*⟩ (⟨*comma*⟩ ⟨*field-change*⟩)* [‘ ’] ‘`}`’

The AST type for the ”change” expression is:

```
newtype DotChange = DC (FieldChange, [FieldChange])
```

And the parser for the ”change” expression is:

```
instance HasParser DotChange where
  parser =
    DC <$>
      (try (string ".change{") *> opt_space_around field_changes_p <* char '}')
    where
    field_changes_p :: Parser (FieldChange, [FieldChange])
    field_changes_p = field_change_p +++ many (comma *> field_change_p)
```

Here we use the ”`+++`” operator which is defined in the parser. It takes two parsers as operands and creates a parser that uses them sequentially and puts the two results in a tuple. ”field_change_p” parses a single field change. ”many” is defined in the library, it parses with the argument parser as many times as possible and puts the results in a list (the Kleene star of the parser world).

7.3.4 Example 3: Value Definition

The grammar rule for the value definition is the following:

⟨*value-def*⟩ ::=
    ⟨*indent*⟩ ⟨*identifier*⟩ ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’) ⟨*type*⟩
    ⟨*nl*⟩ ⟨*indent*⟩ ‘= ’ ⟨*value-expr*⟩ [⟨*where-expr*⟩]

The AST type for the value definition is:

```
newtype ValueDef = VD (Identifier, Type, ValueExpr, Maybe WhereExpr)
```

And the parser for the value definition is:

```
instance HasParser ValueDef where
  parser =
    indent *> parser >>= \identifier ->

    increase_il_by 1 >>

    has_type_symbol *> parser >>= \type_ ->
    nl_indent *> string "= " *>

    increase_il_by 1 >> we_are_in_equal_line >>

    parser >>= \value_expr ->

    we_are_not_in_equal_line >>

    optionMaybe (try parser) >>= \maybe_where_expr ->

    decrease_il_by 2 >>

    return (VD (identifier, type_, value_expr, maybe_where_expr))
```

In this example we see how the state of the parser is used to enforce the indentation rules. The "indent" parser parses <2 * the indentation level> spaces, getting the indentation level from the state (see Indentation System 7.1.2). "`increase_il_by`" and "`decrease_il_by`" have a Parser type but they don't actually parse anything, they are "parsers" that only update the indentation level (but can also be combined with other parsers). They are used as described in rule 2 of the Indentation System (7.1.2). "`has_type_symbol`" parses the following part of the grammar rule: ([‘ ’] ‘:’ [‘ ’] | ⟨*nl*⟩ ⟨*indent*⟩ ‘: ’). "`we_are_in_equal_line`" and "`we_are_not_in_equal_line`" change the state to enforce rule 6 of the Indentation System.

Chapter 8

Translation to Haskell

8.1 High Level Overview

To avoid rewriting the whole Haskell type system, lcases is translated directly to Haskell without any semantic analysis, except for the bare minimum that is required for the translation. The high level phases for the translation are the following:

- **Collect**

  In this phase we traverse the AST and the following are collected:

  - All the ”`naked case`” identifiers of all the `or_types` in the program, where a naked case is a case which does not contain an internal value (e.g. `no_value` as opposed to `the_value:_`).
  - All the field identifiers of the all the `tuple_types` in the program.
  - All the renaming type propositions.

  The above are used in the preprocess phase.

- **Preprocess**

  In this phase the AST generated by the parser is traversed and tweaked according to the following rules:

  - If an identifier of a naked case is found in a value expression, add a ’C’ (for constructor) in the front. This is going to be needed because `or_type` cases are translated to data constructors in Haskell. Data constructors need to start with an upper case letter whereas `or_type` cases don’t (this is not needed for the `or_type` cases with internal values because we can identify them on the spot from the colon and handle them appropriately in the translation phase).

  - If a field identifier is found in a value expression which is a subexpression of a ”`.change`” expression than the identifier is converted to a postfix function with the same argument as the ”`.change`”. For example, if the AST has a part representing the expression below, it is going be converted as follows:
    `x.change{f1 = f1 + 1}` $\Longrightarrow$ `x.change{f1 = x.f1 + 1}`

    The same applies to special identifiers:
    `x.change{1st = 1st + 1}` $\Longrightarrow$ `x.change{1st = x.1st + 1}`

– If a renaming type proposition appears in a theorem of an implication before the arrow, it is substituted with the conjunction defined in its definition. All the substitutions for its ad hoc type variables are also propagated to all of the propositions in the conjunction. For example, if the following appear in the program:

```
type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@C)And(@D)Have_Eq_And_Gr --> (@C)Can_Be_Gr_Or_Eq_To(@D)
proof a >= b = a == b | a > b
```

the theorem becomes:

```
type_theorem
  [(@C)And(@D)Can_Be_Equal, (@C)Can_Be_Greater_Than(@D)] -->
  (@C)Can_Be_Gr_Or_Eq_To(@D)
proof a >= b = a == b | a > b
# this is a representation of the AST after preprocessing not lcases syntax
```

in the preprocessed AST.

- **Translate**

  In this phase the preprocessed AST is directly translated to Haskell. The details of how this is done are described in the following section.

8.2 Translation Phase: General

For the translation phase every type of the AST has its implementation of one of the following 3 polymorphic functions:

- `to_haskell`:

  This function is for the AST types that need no state and can be directly translated to Haskell.

- `to_hs_wpn`: (to `to_haskell` with parameter number)

  This function is for the AST types that implicitly introduce extra parameters when translated to Haskell and therefore need a state to keep track of how many parameters have been introduced.

- `to_hs_wil`: (to `to_haskell` with indentation level)

  This function is for the AST types that need indentation level information to be correctly indented when translated to Haskell. Therefore, a state to keep track of the indentation level is needed.

Here are the corresponding classes that define the above functions:

- ```
  type Haskell = String

  class ToHaskell a where
    to_haskell :: a -> Haskell
  ```

- ```
  type WithParamNum = State Int

  class ToHsWithParamNum a where
    to_hs_wpn :: a -> WithParamNum Haskell
  ```

- ```
  type WithIndentLvl = State Int

  class ToHsWithIndentLvl a where
    to_hs_wil :: a -> WithIndentLvl Haskell
  ```

In the following sections we dive into more detail for particular types.
  ```

8.3 Translation Phase: Basic Expressions

8.3.1 Literals and Identifiers

- **Literals**

  Literals are kept the same as they were parsed except for number literals. Number literals have explicit type annotations (when they are not a case of a ”cases” function expression), so that they have type `Integer` or `Double` and not `Num a => a` or `Fractional p => p` for the Haskell compiler.

- **Identifiers**

  – *Examples*

  `x1` $\Longrightarrow$ `x1`
  `apply(_)to_all_in(_)` $\Longrightarrow$ `apply'to_all_in'`
  `(_)to_string` $\Longrightarrow$ `a'to_string`
  `f(_, _, _)` $\Longrightarrow$ `f'''`

  – *Description*

  For identifiers all underscores in parenthesis are replaced by an equal amount of single quotes. If there is a parenthesis in the beginning of the identifier, the letter 'a' is prepended to make it a valid Haskell identifier.

8.3.2 Parenthesis, Tuples and Lists

- **Parenthesis**

  The translation of the internal expression is put in parenthesis.

- **Tuples**

  – *Examples*

  ```
  (x, y) ⟹ ft2(x, y)
  (_, 3.14, _) ⟹ (\(pA0, pA1) -> ft3(pA0, 3.14, pA1))

  (_, _, ”Hello from 3rd field”)
  ⟹
  (\(pA0, pA1) -> ft3(pA0, pA1, ”Hello from 3rd field”))
  ```

– *Description*

ft*n* function

Every tuple is passed through the **ft*n*** function where ***n*** is the size of the tuple. This is a polymorphic function defined by the following class (for ***n*** = 2 and with similar classes for 3, 4 and 5):

```
class FromTuple2 a b c | c -> a b where
  ft2 :: (a, b) -> c
```

This is because the same tuple may be of a `product type` or of a `tuple_type` depending on where it appears on the program. For product types we have the following instance:

```
instance FromTuple2 a b (a, b) where
  ft2 = id
```

And there is also a new instance automatically generated for every `tuple_type` definition. For example, for the following definition:

```
tuple_type Name
value (first_name, last_name) : String^2
```

Along with the definition itself which is described in section 8.7.2 there will be the following instance:

```
instance FromTuple2 String String Name where
  ft2 = \(x1, x2) -> Name' x1 x2
```

With this mechanism the program will type check correctly on both cases.

Parameters for the underscores

For all the fields of the tuple that contain an underscore, we generate a new parameter in its place
("`pA<n - 1>`" for the n-th underscore) and at the end we prepend the parameters to make it a function expression. "pA" stands for parameter and the A is uppercase to avoid collisions with regular lcases identifiers.

Big Tuples

For big tuples, the original lines are kept and split in the same way. The `ftn` function and the implicitly introduced parameters have their own separate lines at the top as shown in the following example:

```
( "Hey, I'm the first field and I'm also a pretty big string."
, "Hey, I'm the second field and I'm a smaller string."
, _
)
```

Becomes:

```
\pA0 ->
ft3
( "Hey, I'm the first field and I'm also a pretty big string."
, "Hey, I'm the second field and I'm a smaller string"
, pA0
)
```

All of these lines are also indented to the same column according to the indentation level.

- **Lists**
  - *Examples*

```
[1.61, 2.71, 3.14]
⟹
[(1.61 :: Double), (2.71 :: Double), (3.14 :: Double)]

[_, x, _] ⟹ (\(pA0, pA1) -> [pA0, x, pA1])
```

  - *Description*

    Lists work the same way as tuples except for the fact that they don't need the `ft`*n* function.

8.3.3 Parenthesis Function Application

- *Examples*

```
f(x, y, z) ⟹ f'''(x, y, z)
f(x, _, _) ⟹ (\(pA0, pA1) -> f'''(x, pA0, pA1))
(x)to_str ⟹ a'to_str(x)
apply(f)to_all_in(_) ⟹ (\pA0 -> apply'to_all_in'(f, pA0))
```

- *Description*

  For the parenthesis function application, we separate the underlying function identifier by putting quotes in the place of the arguments (and prepending an 'a' if needed) and we also collect the arguments in a tuple. From there, we can apply the function to the argument tuple. If any argument is an underscore, it implicitly introduces a parameter and it is treated similarly to how it is treated in tuples (section 8.3.2).

8.3.4 Prefix and Postfix Functions

- **Prefix functions**
  - *Examples*

```
the_value:42 ⟹ Cthe_value((42 :: Integer))
error:"this is an error message" ⟹ Cerror("this is an error message")
the_value:_ ⟹ (\pA0 -> Cthe_value(pA0))
the_value:result:true ⟹ Cthe_value(Cresult(True))
```

  - *Description*

    Prefix functions are data constructors in Haskell which are introduced by the translation of the relevant `or_type` definition. They are prepended with an upper case 'C' to be valid data constructors and their argument is placed in parenthesis. The argument can be an underscore, in which case a new parameter name is put in its place, the appropriate lambda abstraction is prepended and the whole expression is put in parenthesis to limit the scope of the abstraction.

- **Postfix functions**
  - *Examples*

```
date.year ⟹ year(date)
tuple.1st ⟹ p1st(tuple)
info.date.year ⟹ year(date(info))
tuple.1st.2nd ⟹ p2nd(p1st(tuple))
```

  - *Description*

    Postfix functions are projection functions that are generated automatically by the translation of the relevant `tuple_type` definition or they are the projection functions for product types (`_.1st _.2nd etc`). They are translated into regular Haskell functions (as are the record accessor functions of Haskell) with their argument in parenthesis. For

`_.1st _.2nd etc` a 'p' for "projection" is prepended to make it a valid Haskell function.

The projection functions for product types are polymorphic and work on tuples of any size (in principle, for size ≤ 5 for now). This is achieved by making them polymorphic through the following class (for p1st and similar classes for the rest):

```
class IsFirst' a b | b -> a where
  p1st :: b -> a
```

And the following instances:

```
instance IsFirst' a (a, b) where
  p1st = fst

instance IsFirst' a (a, b, c) where
  p1st = \(a, _, _) -> a

instance IsFirst' a (a, b, c, d) where
  p1st = \(a, _, _, _) -> a

...
```

- **The ".change" Function**
  - *Examples*

```
state.change{counter = counter + 1}
  ⟶ (preprocessing) state.change{counter = state.counter + 1}
  ⟹ c0counter(counter(state) !+ (1 :: Integer)) state

tuple.change{1st = 1.61, 3rd = 3.14} ⟹ (c1st(1.61) .> c3rd(3.14)) tuple

tuple.change{x = _, y = _} ⟹ (\(pA0, pA1) -> (c0x(pA0) .> c0y(pA1)) tuple)
```

  - *Description*

For the ".change" function, similarly to the projection functions, a change function (in Haskell) is introduced for every field of every `tuple_type`. This function is the name of the field prepended by "c0" where the 'c' stands for "change" and the '0' is there so that there can be no collisions with other identifiers (numbers can only be at the end for lcases identifiers). For the product type fields, the change functions are "`c1st`", "`c2nd`", etc. The type of every change function has the form:

```
FieldType -> TupleOrProdType -> TupleOrProdType
```

For every assignment inside the braces we use the change function of the assigned field with the expression of the assignment as the `FieldType` argument. By doing this we have all the functions of type:

```
TupleOrProdType -> TupleOrProdType
```

we need to make the changes. We compose them all with the ".>" operator, which is a predefined operator for function composition from left to right (although the regular "." should also do). Now we have one function of type:

`TupleOrProdType -> TupleOrProdType`

that does all the changes. We apply it on the argument of the "`.change`" function and we are done.

The change functions for product types work on tuples of any size (in principle, for size ≤ 5 for now). This works with appropriate type classes and instances, similarly to the way the `p1st`, `p2nd`, etc functions work. Unfortunately, because of this fact the regular braces notation of Haskell for the same purpose cannot work, since `p1st`, `p2nd`, etc are polymorphic functions and not "record selectors".

If an underscore is assigned to a variable, new parameters are introduced similarly to the way they are introduced for tuples.

8.4 Translation Phase: Operators

8.4.1 Operators

For each one of the lcases operators, a new Haskell operator is defined, for it to be translated to. This allows for the use of the precedence and associativity mechanism for new user defined operators in Haskell instead of having to implement them from scratch in the parser, while also avoiding a lot of extra parentheses that the latter approach would need. Function application and function composition operators are defined as shown below. Every other operator is defined by a type class that matches the type described in table 4.1. The implementation of the operator for every combination of types is defined by an appropriate instance. The examples for the addition operator in the next page show the general structure.

lcases operator	Haskell operator
->	&>
<-	<&
o>	.>
<o	<.
^	!^
*	!*
/	!/
+	!+
-	!-
==	!==
!=	!!=
>	!>
<	!<
>=	!>=
<=	!<=
&	!&
|	!|
;>	!>>=
;	!>>

Precedence and associativity using Haskell:

```
infixl 9 &>
infixr 8 <&
infixl 7 .>, <.
infixr 6 !^
infixl 5 !*, !/
infixl 4 !+, !-
infix 3 !==, !!=, !>, !<, !>=, !<=
infixr 2 !&
infixr 1 !|
infixr 0 !>>=, !>>
```

Function application/composition operators

```
(&>) :: a -> (a -> b) -> b
x &> f = f x

(<&) :: (a -> b) -> a -> b
f <& x = f x

(.>) :: (a -> b) -> (b -> c) -> a -> c
(.>) = flip (.)

(<.) :: (b -> c) -> (a -> b) -> a -> c
(<.) = (.)
```

```
# Type class for addition:
class A'And'Add_To' a b c where
  (!+) :: a -> b -> c

# Some of the instances

# Adding 2 lists:
instance b ~ [a] => A'And'Add_To' [a] [a] b where
  (!+) = (++)

# Adding any type 'a' with a list of 'a' s
instance b ~ [a] => A'And'Add_To' a [a] b where
  (!+) = (:)

# Adding a String to a type 'a' with a Show instance without needing to call show
instance (Show a, b ~ String) => A'And'Add_To' String a b where
  str !+ x = str ++ show x
```

8.4.2 Operator Expressions

- *Examples*

  ```
  5 * 'a' ⟹ (5 :: Integer) !* 'a'

  "Hello " + "World!" ⟹ "Hello " !+ "World!"

  _ - 1 ⟹ (\pA0 -> pA0 !- (1 :: Integer))

  _ + "string in the middle of the arguments" + _
  ⟹ (\(pA0, pA1) -> pA0 !+ "string in the middle of the arguments" !+ pA1)
  ```

- *Description*

  In operator expressions, the operands are translated according to what operands they are (described in their respective section) and the operators are substituted by their respective Haskell operators. If an operand is an underscore than a new lambda abstraction is introduced, similarly to how this is done for tuples in 8.3.2.

  For big operator expression that span multiple lines, the lines are split the same way they are split in the source file and they are indented all the same according to the indentation level. If new lambda abstractions are introduced, they are all placed in a new line on the top, also indented the same. Again, all of the above are done similarly to how they are done for tuples in 8.3.2.

8.5 Translation Phase: Function Expressions

8.5.1 Regular Function Expressions

- *Examples*

```
x => 17 * x + 42 ⟹ \x -> (17 :: Integer) !* x !+ (42 :: Integer)

* => 42 ⟹ \_ -> (42 :: Integer)

(x, *, z) => x + z ⟹ \(x, _, z) -> x !+ z

((x1, y1), (x2, y2)) => (x1 + x2, y1 + y2)
⟹ \((x1, y1), (x2, y2)) -> ft2(x1 !+ x2, y1 !+ y2)
```

- *Description*

  The following are done to translate the parameters:

  – A `'\'` character is prepended
  – The `"=>"` arrow is replaced by the `"->"` arrow
  – A `'*'` parameter becomes a `'_'` parameter

  The body of the function is translated according to expression it is.

  It is possible that the function expression is accompanied by a ”`where`” expression below it. As shown in the example below:

```
gac => print(message)
where
message: String
  = "Gcd: " + gac.gcd + "\nCoefficients: a = " + gac.a + ", b = " + gac.b
```

  In that case, the ”`where`” expression becomes a ”`let-in`” expression as described in section 8.6. This ”`let-in`” expression is placed between the parameters and the body of the function, to make the parameters ”visible” to the expressions in the ”`let-in`” expression as shown below:

```
\gac ->
let
message :: String
message =
  "Gcd: " !+ gcd(gac) !+ "\nCoefficients: a = " !+ a(gac) !+ ", b = " !+ b(gac)
in
print'(message)
```

  where the ”`gac`” parameter is ”visible” to the expression of ”`message`”.

8.5.2 ”cases” Function Expressions

- *Examples*

```
cases
  true => print(”It’s true!! :)”)
  false => print(”It’s false... :(”)

⟹

\pA0 ->
case pA0 of
  True -> print’(”It’s true!! :)”)
  False -> print’(”It’s false... :(”)
```

```
(x, cases)
  0 => x
  y => gcd(y, (x)mod(y))

⟹

\(x, pA0) ->
case pA0 of
  0 -> x
  y -> gcd’’(y, a’mod’(x, y))
```

```
(cases, cases)
  (green, green) => true
  (amber, amber) => true
  (red, red) => true
  ... => false

⟹

\(pA0, pA1) ->
case (pA0, pA1) of
  (green, green) -> True
  (amber, amber) -> True
  (red, red) -> True
  _ -> False
```

```
cases
  [x1, x2, xs = ...] =>
    (x1 < x2) & (x2 + xs)is_sorted
  ... => true

⟹

\pA0 ->
case pA0 of
  x1 : x2 : xs ->
    (x1 !< x2) !& a’is_sorted(x2 !+ xs)
  _ -> True
```

- *Description*

The parameters are translated similarly to how they are translated in regular function expressions, with the one difference being that all parameters that contain the word ”`cases`” are translated to newly generated parameters. Every such parameter is then collected in a tuple that is pattern matched on by creating a new line of the form ”`case <tuple of new parameters> of`”. For the last case, if we have ”...” then it is translated to ”_”. When matching on the first few elements of a list the translation is done as follows:

`[x1, x2, ...] => <case body>` $\Longrightarrow$ `x1 : x2 : _ -> <case body translation>`

`[x1, x2, xs = ...] => <case body>` $\Longrightarrow$ `x1 : x2 : xs -> <case body translation>`

where the square brackets are removed and the commas become colons. If the rest of the list has a name that it is the only thing that is kept after the last colon and if it doesn’t an underscore placed instead.

8.6 Translation Phase: Value Definitions and ”where” Expressions

- *Examples*

```
foo: Int
  = 42

⟹

foo :: Integer
foo =
  (42 :: Integer)
```

```
dfs_on_tree(_) : (T1)Tree => (Int x T1)Tree
  = dfs_on_tree(_)with_num(1) o> _.tree
    where
    dfs_on_tree(_)with_num(_) : (T1)Tree x Int => (T1)ResultTreeAndNum
      = <irrelevant stuff>

    <irrelevant stuff>

dfs_on_tree' :: forall a1. A'Tree a1 -> A'Tree (Integer, a1)
dfs_on_tree' =
  let
  dfs_on_tree'with_num' :: (A'Tree a1, Integer) -> A'ResultTreeAndNum a1
  dfs_on_tree'with_num' = <irrelevant stuff>

  <irrelevant stuff>
  in
  (\pA0 -> dfs_on_tree'with_num'(pA0, (1 :: Integer))) .> (\x' -> tree(x'))
```

```
val1, val2, val3 : Int, Bool, Char
  = 42, true, 'a'

val1 :: Integer
val1 =
  (42 :: Integer)

val2 :: Bool
val2 =
  True

val3 :: Char
val3 =
  'a'
```

```
print_gcd_and_coeffs_of(_): GcdAndCoeffs => IO
  = gac => print(message)
    where
    message: String
      = ″Gcd: ″ + gac.gcd_ + ″\nCoefficients: a = ″ + gac.a + ″, b = ″ + gac.b
```

$\Longrightarrow$

```
print_gcd_and_coeffs_of' :: GcdAndCoeffs -> IO
print_gcd_and_coeffs_of' =
  \gac ->
  let
  message :: String
  message =
    ″Gcd: ″ !+ gcd_(gac) !+ ″\nCoefficients: a = ″ !+ a(gac) !+ ″, b = ″ !+ b(gac)
  in
  print'(message)
```

Description

The following are done to translate value definitions:

- The ”has type” symbol is translated by doubling the colon
- The identifier is reused before the equal sign
- If the value definition is on indentation level 0 the following steps are taken:
  - Collect all the parametric type variables of the type
  - Prepend the following to the translation of the type:
    `″forall ″ <parametric type vars translated and space separated> '.'`

  This allows the use of the same type variable in the types inside the ”`where`” expression if there is one. This is demonstrated in the 2nd example where `T1` (before translation or `a1` after) is used also in the type annotation of as `dfs_on_tree(_)with_num(_)`. By default the ”`a1`”s do not refer to the same type even if they have the same name. The compiler extension `ScopedTypeVariables` is also needed for this to work.
- If the value expression is followed by a ”`where`” expression, the ”`where`” expression is translated to a ”`let-in`” expression that is placed above the translation of the value expression (2nd example). The one exception to this rule is when the value expression is a regular function expression where the ”`let-in`” expression is placed between the parameters and the body (last example).

8.7 Translation Phase: Types

8.7.1 Type Expressions

Type Identifiers

- *Examples*

  `Int` $\Longrightarrow$ `Integer`

  `Real` $\Longrightarrow$ `Double`

  `String` $\Longrightarrow$ `String`

  `SelfReferencingType` $\Longrightarrow$ `SelfReferencingType`

- *Description*

  Type ids remain the same except for Int and Real, which are translated to Integer and Double respectively.

Type Variables

- Parametric Type Variables
  - *Examples*

    `T1` $\Longrightarrow$ `a1`

    `T2` $\Longrightarrow$ `a2`

    `T3` $\Longrightarrow$ `a3`

  - *Description*

    The 'T' becomes an 'a'.

- Ad Hoc Type Variables
  - *Examples*

    `@A` $\Longrightarrow$ `b0`

    `@B` $\Longrightarrow$ `b1`

    `@C` $\Longrightarrow$ `b2`

  - *Description*

    The '@' becomes a 'b' and the capital letters map like so: A $\Longrightarrow$ 0, B $\Longrightarrow$ 1, etc

Type Application Types

- *Examples*

  `ListOf(Int)s` $\Longrightarrow$ `ListOf's Integer`

  `Error(String)OrResult(Int)` $\Longrightarrow$ `Error'OrResult' String Integer`

  `(Int)Tree` $\Longrightarrow$ `A'Tree Integer`

  `ListOf(Int => Int)s` $\Longrightarrow$ `ListOf's (Integer -> Integer)`

  `Before(B,C)After` $\Longrightarrow$ `Before''After B C`

  `A(B(C))` $\Longrightarrow$ `A' (B' C)`

- *Description*

  For type application types, the type id for Haskell is extracted by replacing every parenthesis with as many single quotes as the number of type arguments it has. If the parenthesis is in the beginning, an 'A' (for argument) is prepended to make it a valid Haskell identifier. The type arguments of all the parentheses are collected, translated, parenthesized if needed and appended to the type id separated by spaces.

Product Types

- *Examples*

  `Int x Real x String` $\Longrightarrow$ `(Integer, Double, String)`

  `Int^2 x Int^2` $\Longrightarrow$ `((Integer, Integer), (Integer, Integer))`

  `(A^2 => A) x A x ListOf(A)s` $\Longrightarrow$ `((A, A) -> A, A, ListOf's A)`

- *Description*

  For the product types, all the field types and translated, separated by commas and put in parenthesis.

Function Types

- *Examples*

```
T1 => T1 ⟹ a1 -> a1

Int^2 => Int ⟹ (Integer, Integer) -> Integer

(A^2 => A) x A x ListOf(A)s => A ⟹ ((A, A) -> A, A, ListOf's A) -> A
```

- *Description*

  For the function types, the input and output types are translated and the arrow between them changes from "`=>`" to "`->`".

Conditional Types

- *Examples*

```
(@T)Has_Str_Rep --> @T => String ⟹ A'Has_Str_Rep b19 => b19 -> String

(@A)Is(@B)s_First --> @B => @A ⟹ A'Is's_First b0 b1 => b1 -> b0

(@T)Has_Internal_App --> (T1 => T2) x @T(T1) => @T(T2)
⟹
A'Has_Internal_App b19 => (a1 -> a2, b19 a1) -> b19 a2
```

- *Description*

  For conditional types, the condition is translated similarly to how type application types are translated, with quotes replacing the parentheses and the type variables appended to the condition name and separated by spaces. The simple type is translated according to what type it is and the arrow between the condition and the simple type changes from "`-->`" to "`=>`".

8.7.2 Type Definitions

Tuple Types

- *Examples*

```
tuple_type Date
value (day, month, year) : Int^3

⟹

data Date =
  Date' { day :: Integer, month :: Integer, year :: Integer }

instance FromTuple3 Integer Integer Integer Date where
  ft3 = \(x1, x2, x3) -> Date' x1 x2 x3

c0day :: Integer -> Date -> Date
c0month :: Integer -> Date -> Date
c0year :: Integer -> Date -> Date
c0day = \new x -> x { day = new }
c0month = \new x -> x { month = new }
c0year = \new x -> x { year = new }
```

```
tuple_type Edge
value (u, v) : Node^2

⟹

data Edge =
  Edge' { u :: Node, v :: Node }

instance FromTuple2 Node Node Edge where
  ft2 = \(x1, x2) -> Edge' x1 x2

c0u :: Node -> Edge -> Edge
c0v :: Node -> Edge -> Edge
c0u = \new x -> x { u = new }
c0v = \new x -> x { v = new }
```

```
tuple_type (T1)Tree
value (root, subtrees) : T1 x (T1)Trees
```

$\Longrightarrow$

```
data A'Tree a1 =
  A'Tree' { root :: a1, subtrees :: A'Trees a1 }

instance FromTuple2 a1 (A'Trees a1) (A'Tree a1) where
  ft2 = \(x1, x2) -> A'Tree' x1 x2

c0root :: a1 -> A'Tree a1 -> A'Tree a1
c0subtrees :: A'Trees a1 -> A'Tree a1 -> A'Tree a1
c0root = \new x -> x { root = new }
c0subtrees = \new x -> x { subtrees = new }
```

- *Description*

  For tuple types the translation has the following steps:

  1. "`tuple_type`" $\Longrightarrow$ "`data`"
  2. From the type name the Haskell type id is extracted by replacing the parametric type variables in the parenthesis with single quotes and the parametric type variables are appended separated by spaces, similarly to how it is done in Type Application Types. An equal sign is appended to the above.
  3. "`value`" is discarded and the second line is indented and starts with the data constructor, which is the Haskell type id ended with a single quote.
  4. We add the Haskell record syntax with the fields separated by commas and annotated with their respective types, which are translated from the product type that ends the "`tuple_type`" definition.
  5. In this step the instance for the `ft`*n* function is defined where *n* is the size of the tuple of the `tuple_type`. This is required for the translation of tuples for reasons explained in section 8.3.2. This can be divided into the following substeps (where *n* is the size of the tuple of the "`tuple_type`", not 'n' verbatim):
     (a) "`instance FromTuple`*n* "
     (b) We append the translations (parenthesized if needed) of all the field types and the `tuple_type` name (in that order) separated by spaces and followed by " `where`".
     (c) The second line is indented and starts with `ft`*n* `=`.
     (d) A `\` and a tuple parameter with internal parameters `x1` to `x`*n* followed by " `->` " is appended.
     (e) The data constructor of step 3 followed by the parameters `x1` to `x`*n* separated by spaces are appended.
  6. The type annotations for the change function of each field (described in the "`.change`" section of 8.3.4) are generated by the following substeps for each change function:
     – Prepend "`c0`" to the field identifier
     – Append " `::` " followed by the type, which is of the following form:

       ```
       <translation of field type> ->
         <name of the tuple_type> -> <name of the tuple_type>
       ```

7. The definitions for the change function of each field are generated by the following sub-steps for each change function:
   - Prepend "`c0`" to the field identifier
   - Append " `= \new x -> x { <field id> = new }`"

Or Types

- *Examples*

```
or_type Bool
values true | false

⟹

data Bool =
  Ctrue |
  Cfalse
```

```
or_type Possibly(T1)
values the_value:T1 | no_value

⟹

data Possibly' a1 =
  Cthe_value a1 |
  Cno_value
```

```
or_type Error(T1)OrResult(T2)
values error:T1 | result:T2

⟹

data Error'OrResult' a1 a2 =
  Cerror a1 |
  Cresult a2
```

```
or_type Comparison
values lesser | equal | greater

⟹

data Comparison =
  Clesser |
  Cequal |
  Cgreater
```

- *Description*

For `or_types` the translation has the following steps:

1. "`tuple_type`" ⟹ "`data`"
2. The type name is translated the same way as it is in tuple types.
3. " =" is appended.
4. "`values`" is discarded and the lines from the 2nd onwards have the data constructors for each of the values of the `or_type` (one in each). This is done as follows:
   (a) 'C' is prepended to the identifier of the value to make it a data constructor.
   (b) If there is an internal value, the type is translated and appended (separated by a space).
   (c) If it is not the last value, " |" is appended.

Type Nicknames

- *Examples*

```
type_nickname Ints = ListOf(Int)s ⟹ type Ints = ListOf's Integer

type_nickname ErrOrRes(T1) = Error(String)OrResult(T1)
⟹ type ErrOrRes' a1 = Error'OrResult' String a1
```

- *Description*

  For type nicknames we replace "`type_nickname`" with "`type`", the type name (before the equal sign) with its translation and the type (after the equal sign) with its translation.

8.8 Translation Phase: Type Logic

Proposition Definitions Renaming proposition definitions do not have a translation as they only affect theorems during the preprocessing phase. For atomic proposition definitions we have the following:

- *Examples*

```
type_proposition (@A)Is(@B)s_First
needed (_)first : @B => @A

⟹

class A'Is's_First b0 b1 where
  a'first :: b1 -> b0
```

```
type_proposition (@T)Has_Internal_App
needed
  apply(_)inside(_)
    : (T1 => T2) x @T(T1) => @T(T2)

⟹

class A'Has_Internal_App b19 where
  apply'inside'
    :: (a1 -> a2, b19 a1) -> b19 a2
```

```
type_proposition (@T)Has_A_Wrapper
needed wrap(_) : T1 => @T(T1)

⟹

class A'Has_A_Wrapper b19 where
  wrap' :: a1 -> b19 a1
```

```
type_proposition (@T)Has_String_Repr
needed (_)to_string : @T => String

⟹

class A'Has_String_Repr b19 where
  a'to_string :: b19 -> String
```

```
type_proposition (@A, @B)To(@C)
needed ab_to_c: @A x @B => @C#

⟹

class A''To' b0 b1 b2 where
  ab_to_c :: (b0, b1) -> b2
```

- *Description*

  The translation of atomic type proposition definitions is done with the following steps:

  - "`type_proposition`" is replaced by "`class`".
  - The name of the proposition is translated similarly to the type name in a type definition. All the parentheses are replaced by the same amount of single quotes as there are ad hoc type variables inside the parenthesis and if there is a parenthesis at the beginning we prepend an 'A' to make it a valid type class name in Haskell. The translations of the ad hoc type variables are then appended and separated by spaces (in the same order in which they appear in the proposition).
  - "` where`" is appended to the first line.
  - "`needed`" is discarded and the second line is indented
  - The identifier of the value needed is translated and followed by "` :: `" and the translation of its type.

Theorems

- *Examples*

```
type_theorem
  (Possibly(_))Has_A_Wrapper
proof wrap(_) = the_value:_

⟹

instance
  A'Has_A_Wrapper Possibly' where
  wrap' = (\pA0 -> Cthe_value(pA0))
```

```
type_theorem
  (ListOf(_)s)Has_A_Wrapper
proof wrap(_) = [_]

⟹

instance
  A'Has_A_Wrapper ListOf's where
  wrap' = (\pA0 -> [pA0])
```

```
type_theorem (Possibly(_))Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      no_value => no_value
      the_value:x => the_value:f(x)

⟹

instance A'Has_Internal_App Possibly' where
  apply'inside' =
    \(f', pA0) ->
    case pA0 of
      Cno_value -> Cno_value
      Cthe_value x -> Cthe_value(f'(x))
```

```
type_theorem (ListOf(_)s)Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      [] => []
      [head, tail = ...] => f(head) + apply(f(_))inside(tail)

⟹

instance A'Has_Internal_App ListOf's where
  apply'inside' =
    \(f', pA0) ->
    case pA0 of
      [] -> []
      head : tail -> f'(head) !+ apply'inside'((\pA0 -> f'(pA0)), tail)
```

```
type_proposition (@A)And(@B)Have_Eq_And_Gr
equivalent (@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)

type_theorem (@A)And(@B)Have_Eq_And_Gr --> (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
```

preprocessing ⟹

```
type_theorem
  [(@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)] -->
  (@A)Can_Be_Gr_Or_Eq_To(@B)
proof a >= b = a == b | a > b
# this is a representation of the AST after preprocessing
# not lcases syntax
```

translation ⟹

```
instance
  (A'And'Can_Be_Equal b0 b1, A'Can_Be_Greater_Than' b0 b1) =>
  A'Can_Be_Gr_Or_Eq_To' b0 b1
  where
  a !>= b = a !== b !| a !> b
```

- *Description*

  The translation of type theorems is done with the following steps:

  - ”`type_theorem`” is replaced by ”`instance`”.
  - All the type propositions with substitutions (before and after the arrow if there is one), are translated and the arrow changes from ”`->`” to ”`=>`”. The propositions with substitutions are translated the same way propositions are translated in proposition definitions. The only difference is that instead of ad hoc type variables there are type substitutions. These come in the form of type expressions where there might be underscores instead of type arguments (e.g `ListOf(_)s`). These type expressions are translated like regular type expressions except for the underscore type arguments which are not translated at all, so that the type constructor itself acts as a higher kinded type (e.g `ListOf's`).
  - ” `where`” is appended to the above.
  - ”`proof`” is discarded and the second line is indented.
  - The identifier of the value or the operator surrounded by two identifiers are translated, followed by
    ” `=` ” and the translation of the value expression.

Chapter 9

Running

- Create compiler executable

  ```
  make lcc
  ```

- Compile lcases program to executable

  ```
  ./lcc hello_world.lc
  ```

  Creates executable ”hello_world”

  ```
  ./hello_world
  ```

- Compile lcases program to Haskell

  ```
  ./lcc -h hello_world.lc
  ```

  Creates executable ”hello_world.hs”

  ```
  ghci hello_world.hs
  ```

- Run test_inputs

  ```
  make
  ```

  Creates ”test_outputs” directory where

  - `test_outputs/compiled_progs`
    contains the compiled executables of all ”`test_inputs/programs`”
  - `test_outputs/programs`
    contains the Haskell translation of all ”`test_inputs/programs`”
  - `test_outputs/grammar_rules`
    contains the Haskell translation of various examples for each grammar rule from ”`test_inputs/grammar_rules`”

  also create ”grules” executable that is run for ”`test_outputs/grammar_rules`”

- Clean

  ```
  make clean
  ```

  Removes: `test_outputs`, `lcc`, `grules`, `hello_world`, `hello_world.hs`

Chapter 10

Conclusion

A translator from lambda-cases to Haskell has been implemented and it is written in Haskell. The main differences between lambda-cases and Haskell are:

- Function application is done with parentheses instead of spaces.
  Example: "`f(x,y,z)`" instead of "`f x y z`"
- Functions can be defined to expect arguments before or in the middle of the function identifier.
  Example: "`apply(f)to_all_in(list)`" instead of "`map f list`"
- Partial function application can be done for any of the arguments.

  Examples:
  "`f(x, y, _)`" instead of "`f x y`" for the regular Haskell use for a function that expects 3 arguments.
  In addition, the following are possible:

  - "`f(x, _, z)`" instead of "`\y -> f x y z`"
  - "`f(_, y, z)`" instead of "`\x -> f x y z`"
  - "`f(_, _, z)`" instead of "`\x y -> f x y z`"
  - etc
- The above syntax for expected arguments with underscores is also possible for operands in operator expressions.

  Examples:
  "`_ + 1`" instead of "`(+ 1)`" for the regular Haskell use for expecting one operand.
  In addition, the following are possible:

  - "`"Hello " + _ + "! You look much younger than " + _ + "!"`"
    instead of
    "`\name age -> "Hello " ++ name ++ "! You look much younger than " + age + "!"`"
  - "`_^2 + _^2`" instead of "`\x y -> x^2 + y^2`"
  - etc
- The above syntax for expected arguments with underscores is also possible for elements of tuples and lists.

  Examples:

  - "`(17, _, 42)`" instead of "`\x -> (17, x, 42)`"
  - "`[17, _, 42]`" instead of "`\x -> [17, x, 42]`"

- No identifiers are needed for pattern matching on function parameters, the ”`cases`” keyword is used instead.

  Examples:

  –
  ```
  cases
    true => print(”It’s true!! :)”)
    false => print(”It’s false... :(”)
  ```
  instead of
  ```
  \b -> case b of
    True -> putStrLn ”It’s true!! :)”
    False -> putStrLn ”It’s false... :(”
  ```
  This is also possible in Haskell with the LambdaCase extension like so:
  ```
  \case
    True -> putStrLn ”It’s true!! :)”
    False -> putStrLn ”It’s false... :(”
  ```
  This is where the name of the language comes from, the very frequent use of the LambdaCase. This idea is extended to any subset of any number of parameters as shown in the following example (and in the examples of the ”cases” function expressions section 4.3.2)
  –
  ```
  (cases, cases)
    (green, green) => true
    (amber, amber) => true
    (red, red) => true
    ... => false
  ```
  instead of
  ```
  \x y -> case (x, y) of
    (Green, Green) -> True
    (Amber, Amber) -> True
    (Red, Red) -> True
    _ -> False
  ```

- Powers can be used on types. For example:

  ”`Int^2`” instead of ”`Int x Int`” in lcases or ”`(Integer, Integer)`” in Haskell.

- Operators are generalized.

  Examples:

  – ”`”Hello ” + ”World!”`” instead of ”`”Hello ” ++ ”World!”`”
  – ”`5 * ’a’`” instead of ”`replicate 5 ’a’`”

Future Work

Very little semantic analysis has been done and any error that is not a parsing error will be thrown by the Haskell compiler. For a more complete compiler/translator, a semantic analysis step is needed and error messages should refer to lambda-cases program and not the Haskell translation. This has not been implemented due to the complexity of the Haskell type system and the enormous work it would require to reimplement it. However, because lambda-cases forces the user to use type annotations, it might be possible to implement a simpler type system that is enough for what lambda-cases needs and/or throw an error message if type checking is not possible, to force the user to provide the relevant type annotations. This is what we will be working on in the future of the project.

Βιβλιογραφία

[1] S. Marlow *et al.*, “Haskell 2010 language report,” 2010.

[2] D. Leijen, “Parsec, a fast combinator parser,” 2001.