Lambda Cases (lcases)

Dimitris Saridakis

Contents

1 Introduction

Haskell is a delightful language. Yet, for some reason, it doesn't seem to have it's rightful place in terms of popularity in industry. Why is it so? Is it inherently hard to learn and therefore only the brave dare to use it, 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).

2 Language Description

An lcases program consists of a set of definitions and theorems. Definitions are split into value definitions, type definitions, type predicate definitions and type proposition definitions. Theorems are proven type propositions. The definition of the "main" value determines the program's behaviour. Constants and functions are all considered values and they have no real distinction other than the fact that functions have a function type and constants don't. Functions (just like "values") can be passed to other functions as arguments or can be returned as a result of other functions.

Program example: extended euclidean alogirthm

```
// type definitions

tuple_type Coeffs
value (previous, current) : Int x Int

tuple_type GcdAndCoeffs
value (gcd, a, b) : Int x Int x Int

// algorithm

init_a_coeffs, init_b_coeffs: all Coeffs
  = (1, 0), (0, 1)

ext_euc: (Int, Int) => GcdAndCoeffs
  = ext_euc_rec(init_a_coeffs, init_b_coeffs)

ext_euc_rec: (Coeffs, Coeffs, Int, Int) => GcdAndCoeffs
  = (a_coeffs, b_coeffs, x, cases) =>
    0 => (x, a_coeffs.previous, b_coeffs.previous)
    y => ext_euc_rec(next <- a_coeffs, next <- b_coeffs, y, x -> mod <- y)
      where
      next: Coeffs => Coeffs
        = fields => (current, previous - x / y * current)
```

```
// reading, printing and main

read_two_ints : (Int x Int)IOAction
  = print <- "Please give me 2 ints";
    get_line ;> split_words o> apply(from_string)to_all o> ints =>
    ints -> length -> cases =>
      2 => ints -> with_io
      ... => io_error <- "You didn't give me 2 ints"

print_gcd_equation : (Int, Int, GcdAndCoeffs) => (EmptyValue)IOAction
  = (x, y, fields) => print("gcd = " + gcd + " = " + sum_string)
    where
    sum_string : String
      = a + " * " + x + " + " + b + " * " + y

main : (EmptyValue)IOAction
  = read_two_ints ;> ints =>
    print_gcd_and_coeffs(ints.1st, ints.2nd, ext_euc(ints.1st, ints.2nd))
```

2.1 Basic Value Expressions

2.1.1 Literals and Identifiers

Literals

- *Examples*

```
1  2  17  42  -100
1.61  2.71  3.14  -1234.567
'a'  'b'  'c'  'x'  'y'  'z'  '.'  ','  '\n'
"Hello World!"  "What's up, doc?"  "Alrighty then!"
```

- *Description*

  We have literals for the four basic types: Int, Real, Char, String. These are the usual integers, real numbes, characters and strings. The exact specification of literals is the same as in the Haskell report.

- *Grammar*

  $\langle literal \rangle$ ::= $\langle literal \rangle$

  TODO add the grammar from the haskell report

Identifiers

- *Examples*

```
x y z
a1 a2 a3
funny_identifier
unnecessarily_long_identifier
apply()to_all
```

- *Description*

  An identifier is a string used as the name of a value. It is first used in the definition of the value (see "value definition" section 2.4) and later used in the definition of other values that use that defined value. An identifier starts with a lower case letter and is followed by lower case letters or underscores. It also possible to have a pairs of parentheses in the middle of an identifier (see "parenthesis function application" section 2.1.3). Finally, an identifier can be ended with a digit.

- *Grammar*

  $\langle identifier \rangle$ ::= [a-z]([a-z_] | '()'[a-z_])* [[0-9]]

2.1.2 Parenthesis, Tuples and Lists

Parenthesis

- *Examples*

```
(1 + 2)
in
(1 + 2) * 3

(x => f(x) + 1) and (s => "f(val) + 1 is: " + s)
in
val -> (x => f(x) + 1) -> to_string -> (s => "f(val) + 1 is: " + s)

("Line is: " + line) and (get_line ;> line => print("Line is: " + line))
in
do(3)times <- (get_line ;> line => print("Line is: " + line))
```

- *Description*

  An expression is put in parenthesis to prioritize it or isolate it in a bigger expression. The expressions that can be placed in parethesis are "simple" operator or function expressions (or a mix). In this context, "simple" means that the "cases" syntax does not appear anywhere in the expression (see "cases" syntax section 2.3.3).

- *Grammar*

  ⟨*paren-expr*⟩ ::= '(' ⟨*simple-expr*⟩ ')'

  ⟨*simple-expr*⟩ ::= ⟨*simple-op-expr*⟩ | ⟨*simple-func-expr*⟩

  ⟨*simple-op-expr*⟩ ::= (⟨*op-arg*⟩ '␣' ⟨*op*⟩ '␣')+ (⟨*op-arg*⟩ | ⟨*simple-func-expr*⟩)

Tuples

- *Examples*

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

- *Description*

  Tuples are used to group many values (of possibly different types) into one. The type of the tuple can be either the product of the types of the subvalues or a defined tuple_type which is equivalent to the afformentioned product type i.e. the product type is in the definition of the tuple_type (see "tuple_type" section 2.5.2). For example, the type of the second example 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
```

- *Grammar*

  ⟨*tuple*⟩ ::= '(' ⟨*basic-or-simple-expr*⟩ (',␣' ⟨*basic-or-simple-expr*⟩)+ ')'

  ⟨*basic-or-simple-expr*⟩ := ⟨*basic-expr*⟩ | ⟨*simple-expr*⟩

  ⟨*basic-expr*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩ | ⟨*tuple*⟩ | ⟨*list*⟩ | ⟨*paren-func-app*⟩

Lists

- *Examples*

```
[1, 2, 17, 42, -100]
[1.61, 2.71, 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(A)s where A 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)AddTo(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
```

- *Grammar*

  ⟨*list*⟩ ::= '[' [⟨*basic-or-simple-expr*⟩ (',␣' ⟨*basic-or-simple-expr*⟩)*] ']'

2.1.3 Parenthesis Function Application

Examples

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

Description

Function application in lcases can be done in many different ways in an attempt to maximize readability. In this section, we discuss the ways function application can be done with parenthesis. In the first two examples, we have the usual mathematical function application which is also used in most programming languages and should be familiar to the reader. That is, function application is done with the arguments of the function in parenthesis seperated by commas and **appended** to the function identifier.

We extend this idea by allowing the arguments to be **prepended** to the function identifier (third example). Finally, it is also possible to to have the arguments **inside** the function identifier provided the function has been **defined with parentheses inside the identifier**. For example, below is the definition of "apply()to_all":

```
apply()to_all: (A => B, ListOf(A)s) => ListOf(B)s
  = (f, cases) =>
    empty => empty
    non_empty:l => non_empty:(f <- l.head, apply(f)to_all <- l.tail)
```

The actual definition doesn't matter at this point, what matters is that the identifier is "apply()to_all" with the parentheses **included**. This is very useful for defining functions where the argument in the middle makes the function application look and sound more like natural language.

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

Grammar

⟨*paren-func-app*⟩ ::=
    ⟨*arguments*⟩ (⟨*identifier-with-arguments*⟩ | ⟨*identifier*⟩)
  | (⟨*identifier-with-arguments*⟩ | ⟨*identifier*⟩) ⟨*arguments*⟩
  | ⟨*identifier-with-arguments*⟩

⟨*arguments*⟩ ::= '(' ⟨*basic-or-simple-expr*⟩ (',␣' ⟨*basic-or-simple-expr*⟩)* ')'

⟨*identifier-with-arguments*⟩ ::=
    [a-z] ⟨*id-char-or-paren-id-char*⟩* (⟨*arguments*⟩ [a-z_] ⟨*id-char-or-paren-id-char*⟩*)+ [[0-9]]

⟨*id-char-or-paren-id-char*⟩ ::= [a-z_] | '()'[a-z_]

2.2 Value Operators

2.2.1 Function Application Operators

Operator	Type
`->`	(A, A => B) => B
`<-`	(A => B, A) => B

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 : (A => B, ListOf(A)s) => ListOf(B)s
string_length: String => Int
filter_with : (A => Bool, ListOf(A)s) => ListOf(A)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 number of characters in all the strings that have odd length:

```
chars_in_odd_length_strings : Int
  = strings -> apply(string_length)to_all -> filter_with(is_odd) -> sum_ints
```

Ofcourse this can be done equivalently using the other operator:

```
chars_in_odd_length_strings : Int
  = sum_ints <- filter_with(is_odd) <- apply(string_length)to_all <- strings
```

These operators can also be used together to put a function between two arguments if that function is commonly used that way in math (or if it looks better for a certain function). For example the "mod" function can be used like so:

```
x -> mod <- y
```

Instead of:

```
mod(x, y)
```

2.2.2 Function Composition Operators

Operator	Type
`o>`	(A => B, B => C) => (A => C)
`<o`	(B => C, A => B) => (A => C)

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 '$\circ$' 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_words : String => ListOf(String)s
apply()to_all : (A => B, ListOf(A)s) => ListOf(B)s
reverse_string: String => String
merge_words : ListOf(String)s => String
```

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

```
reverse_words : String => String
  = split_words o> apply(reverse_string)to_all o> merge_words
```

Ofcourse this can be done equivalently using the other operator:

```
reverse_words : String => String
  = merge_words <o apply(reverse_string)to_all <o split_words
```

2.2.3 Arithmetic Operators

Operator	Type
^	(A)ToThe(B)Gives(C) --> (A, B) => C
*	(A)And(B)MultiplyTo(C) --> (A, B) => C
/	(A)Divides(B)To(C) --> (B, A) => C
+	(A)And(B)AddTo(C) --> (A, B) => C
-	(A)SubtractsFrom(B)To(C) --> (B, A) => 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
2
>> 1 + 3.14
4.14
>> ’a’ + ’b’
"ab"
>> ’w’ + "ord"
"word"
>> "Hello " + "World!"
"Hello World!"
>> 5 * ’a’
"aaaaa"
>> 5 * "hi"
"hihihihihi"
>> "1,2,3" - ’,’
"123"
```

Let’s analyze further the example of addition. The type can be read as such: the ’+’ operator has the type (A, B) => C, provided that the type proposition (A)And(B)AddTo(C) holds. This proposition being true, means that addition has been defined for these three types (see section ”type logic” 2.6 for more on type propositions). For example, by the examples above we can deduce that the following propositions are true (in the order of the examples):

```
(Int)And(Int)AddTo(Int)
(Int)And(Real)AddTo(Real)
(Char)And(Char)AddTo(String)
(Char)And(String)AddTo(String)
(Int)And(Char)MultiplyTo(String)
(Int)And(String)MultiplyTo(String)
(Char)SubtractsFrom(String)To(String)
```

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

2.2.4 Comparison and Boolean Operators

Operator	Type
= /=	(A)HasEquality --> (A, A) => Bool
> < >= <=	(A)HasOrder --> (A, A) => Bool
& |	(Bool, Bool) => Bool

The comparison and boolean operators behave the same as in Haskell and very similarly to most programming languages. The main difference is that in lcases the "equals", "and" and "or" operators have the symbol once (`= & |`) rather than twice (`== && ||`).

2.2.5 Environment Action Operators

Operator	Type
;>	(E)IsAnEnvAction --> (E(A), A => E(B)) => E(B)
;	(E)IsAnEnvAction --> (E(A), E(B)) => E(B)

Simple Example

```
print_string("I'll repeat the line.") ; get_line ;> print_string
```

The example above demonstrates the use of the environment action operators with the `IOAction` environment action, 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 => (EmptyValue)IOAction
print_string("I'll repeat the line.")  : (EmptyValue)IOAction
get_line  :  (String)IOAction

;  :  (E)IsAnEnvAction --> (E(A), E(B)) => E(B)
print_string("I'll repeat the line.") ; get_line  :  (String)IOAction
  where
  (IOAction)IsAnEnvAction, E = IOAction, A = EmptyValue, B = String

;>  :  (E)IsAnEnvAction --> (E(A), A => E(B)) => E(B)
print_string("I'll repeat the line.") ; get_line ;> print_string  :  (EmptyValue)IOAction
  where
  (IOAction)IsAnEnvAction, E = IOAction, A = String, B = EmptyValue
```

Example program

```
main : (EmptyValue)IOAction
  = print_string <- "Hello! What's your name?" ; get_line ;> name =>
    print_string("Oh hi " + name + "! What's your age?") ; get_line ;> age =>
    print_string("Oh that's crazy " + name + "! I didn't expect you to be " + age + "!");
```

In this bigger but similar example the types are:

```
print_string  :  String => (EmptyValue)IOAction
get_line  :  (String)IOAction

print_string <- "Hello! ... "  :  (EmptyValue)IOAction
```

```
print_string("Oh hi...)  :  (EmptyValue)IOAction
print_string("Oh that's crazy...)  :  (EmptyValue)IOAction

;  :  (E)IsAnEnvAction --> (E(A), E(B)) => E(B)
print_string("Oh hi...) ; get_line  :  (String)IOAction
  where
  (IOAction)IsAnEnvAction, E = IOAction, A = EmptyValue, B = String

age => print_string("Oh that's crazy...)  :  String => (EmptyValue)IOAction

;>  :  (E)IsAnEnvAction --> (E(A), A => E(B)) => E(B)

print_string("Oh hi...) ; get_line ;> age =>
print_string("Oh that's crazy...) :  (EmptyValue)IOAction
  where
  (IOAction)IsAnEnvAction, E = IOAction, A = String, B = EmptyValue

print_string <- "Hello..." ; get_line  :  (String)IOAction
name => print_string("Oh hi ... (till the end)  :  String => (EmptyValue)IOAction

print_string <- "Hello..." ; get_line ;> name =>
print_string("Oh hi ... (till the end)
   :  (EmptyValue)IOAction
```

Therefore, "`main : (EmptyValue)IOAction`" checks out. The key here is to remember that function expressions extend to the end of the whole expression. Therefore, we have "`name => ...  (till the end)`" and "`age => ... (till the end)`" as the second arguments of the two occurences of the "`;>`" operator.

Description

The environment action operators are used to combine values that have some environment action embedded. Environment actions are also represented by types. These types take other types as arguments and produce new types (just like ListOf()s). The values of the type `E(A)` where `(E)IsAnEnvAction` have inside both a value of type `A` together with an environment action of type `E`.

The effect of the operator "`;`" described in words is the following: given a value of type `E(A)` (which is a value of type `A` with an environment action of type `E`) and a value of type `E(B)` (which is a value of type `B` with an environment action of type `E`), combine those two values by producing the value of type `B` with a new environment action of type `E` which is the combination of the two environment actions. The overall effect produces a value of type `B` with an environment action of type `E` and therefore the overall value is of type `E(B)`.

Note that the value of type `A` inside the first value is not used anywhere. This is mostly used when `A = EmptyValue`. This happens because values of type `E(EmptyValue)` are used for their environment action only (e.g. `print_string(...) : (EmptyValue)IOAction`).

How the two environment actions of the `E(A)` and `E(B)` values are combined to produce the new environment action is specific to the envoriment action type `E`.

The effect of the operator "`;>`" described in words is the following: given a value of type `E(A)` (which is a value of type `A` with an environment action of type `E`) and a value of type `A => E(B)` (which is a function from a value of type `A` to a value of type `B` with an environment action of type `E`), combine those two values by passing the value of type `A` inside the first value as an argument to the second value. This results in a value of type `E(B)` (which is a value of type `B` with an environment action of type `E`). Then, produce this value of type `B` with a new environment action of type `E` that is produced by combining the first environment action with the resulting environment action.

The overall effect produces a value of type `B` with an environment action of type `E` and therefore the overall value is of type `E(B)`.

2.2.6 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)
```

Description

Operator expressions are expressions that use operators. Operators act like two-argument-functions that are placed in between their arguments. 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 Table, Precedence and Associativity") in Table 2.

Just like functions, the sub-expressions that act as arguments to an operator, must have types that match the types expected by the operator.

It is possible to end an operator expression with a function. This is mostly useful with the "`;>`" operator (see previous section: "Environment Operators"), but it is also possible with the following operators: "`->`", "`o>`", "`<o`".

Grammar

⟨*op-expr*⟩ ::= (⟨*op-arg*⟩ '␣' ⟨*op*⟩ '␣')+ (⟨*op-arg*⟩ | ⟨*simple-func-expr*⟩ | ⟨*cases-func-expr*⟩)

⟨*op-arg*⟩ ::= ⟨*basic-expr*⟩ | ⟨*paren-expr*⟩

⟨*op*⟩ ::= '`->`' | '`<-`' | '`o>`' | '`<o`' | '`^`' | '`*`' | '`/`' | '`+`' | '`-`' | '`=`' | '`/=`' | '`>`' | '`<`' | '`>=`' | '`<=`' | '`&`' | '`|`' | '`;>`' | '`;`'

2.2.7 Complete Table, Precedence and Associativity

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 assosiativity of the operators is shown in the table below.

Table 1: The complete table of lcases operators along with their types and their short descriptions.

Operator	Type	Description
`->`	(A, A => B) => B	Right function application
`<-`	(A => B, A) => B	Left function application
`o>`	(A => B, B => C) => (A => C)	Right function composition
`<o`	(B => C, A => B) => (A => C)	Left function composition
`^`	(A)ToThe(B)Gives(C) --> (A, B) => C	General exponentiation
`*`	(A)And(B)MultiplyTo(C) --> (A, B) => C	General multiplication
`/`	(A)Divides(B)To(C) --> (B, A) => C	General division
+	(A)And(B)AddTo(C) --> (A, B) => C	General addition
`-`	(A)SubtractsFrom(B)To(C) --> (B, A) => C	General subtraction
= /=	(A)HasEquality --> (A, A) => Bool	Equality operators
`> < >= <=`	(A)HasOrder --> (A, A) => Bool	Order operators
`& |`	(Bool, Bool) => Bool	Boolean operators
`;>`	(E)IsAnEnvAction --> (E(A), A => E(B)) => E(B)	Monad bind
`;`	(E)IsAnEnvAction --> (E(A), E(B)) => E(B)	Monad then

Table 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

2.3 Function Expressions

Function expressions are either simple function expresssions or function expressions that use the ”cases” syntax. For more details and examples of both see the following sections.

Grammar

⟨*func-expr*⟩ ::= ⟨*simple-func-expr*⟩ | ⟨*cases-func-expr*⟩

2.3.1 Simple Function Expressions

Examples

```
a => 17 * a + 42
```

```
(x, y, z) => (x ^ 2 + y ^ 2 + z ^ 2) ^ (1 / 2)
```

Description

Simple 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. A parameter is either an identifier or the keyword "fields" (see next section: "fields" Special Parameter). The parameters are either only one, in which case there is no parenthesis, or they are many, in which case they are in parenthesis, seperated by commas. The parameters and the body are seperated by an arrow (" `=>` "). The body is a basic expression or a "simple" operator expression meaning that there is no "cases" syntax inside the operator expression.

Grammar

⟨*simple-func-expr*⟩ ::= ⟨*parameters*⟩ '␣=>␣' (⟨*basic-expr*⟩ | ⟨*simple-op-expr*⟩)

⟨*parameters*⟩ ::= ⟨*parameter*⟩ | '(' ⟨*parameter*⟩ (',␣' ⟨*parameter*⟩)+ ')'

⟨*parameter*⟩ ::= ⟨*identifier*⟩ | '`fields`'

2.3.2 "fields" Special Parameter

Examples

```
add_ints_and_print_with_string: Int x Int x String
  = fields => print("Ints: " + (1st + 2nd) + "\nString: " + 3rd)

tuple_type GcdAndCoeffs
value (gcd, a, b) : Int x Int x Int

print_gcd_equation : (Int, Int, GcdAndCoeffs) => (EmptyValue)IOAction
  = (x, y, fields) => print("gcd = " + gcd + " = " + sum_string)
    where
    sum_string : String
      = a + " * " + x + " + " + b + " * " + y
```

Description

The **"fields"** special parameter can be used when the parameter is of a **product type** (e.g. `Int x Int x String`) or of a **tuple type** (see "Tuple Types" section 2.5.2). It allows the direct use of the **subvalues (fields)** of that parameter in the function body without the need to name the parameter itself. This is very useful when the whole parameter is not used but (some or all of) it's subvalues are.

In the case where the parameter is of a **product type** the special identifiers "1st, 2nd, 3rd, 4th and 5th" can be used for the respective subvalues. Similarly, when the parameter is of a **tuple type** the identifiers used in the type's definition can be used for the subvalues.

The use of the **"fields"** keyword in multiple parameters is possible as long as it is not used in two or more parameters of the same type, to avoid ambiguities.

2.3.3 "cases" Syntax

Examples

```
 print_sentimental_bool : Bool => (EmptyValue)IOAction
  = cases =>
    true => print <- "It's true!! :)"
    false => print <- "It's false... :("

or_type TrafficLight
values green | amber | red

print_sentimental_traffic_light : Bool => (EmptyValue)IOAction
  = cases =>
    green => print <- "It's green! Let's go!!! :)"
    amber => print <- "Go go go, fast!"
    red => print <- "Stop right now! You're going to kill us!!"

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

gcd : (Int, Int) => Int
  = (x, cases) =>
    0 => x
    y => gcd(y, x -> mod <- y)

is_empty : ListOf(A)s => Bool
  = cases =>
    empty => true
    non_empty:anything => false

apply()to_all: (A => B, ListOf(A)s) => ListOf(B)s
  = (f, cases) =>
```

```
    empty => empty
    non_empty:list => non_empty:(f <- list.head, apply(f)to_all <- list.tail)
```

Description

"cases" is a keyword that works as a special parameter. The difference is that instead of giving the name "cases" to that parameter, it let's you pattern match on the possible values of that parameter and return a different result for each particular case (hence the name!).

The "cases" keyword can only be used on parameters that have either one of the basic types (Int, Real, Char, String) or an or_type (e.g. Bool, ListOf(A)s).

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_identifier => (body of default case)`" to capture all remaining cases while being able to use the value with the name "`some_identifier`" (e.g. "y" in `gcd` example).

It is possible to use the "cases" keyword in multiple parameters to match on all of them. By doing that, each case represents a particular combination of values for the parameters involved (e.g. `traffic_lights_match` example).

A function expression that uses the "cases" syntax must contain the "cases" keyword in at least one parameter. The number of matching expressions in all cases must be the same as the number of parameters with the "cases" keyword.

Grammar

⟨*cases-func-expr*⟩ ::= ⟨*cases-parameters*⟩ '` => `' ⟨*cases*⟩

⟨*cases-parameters*⟩ ::= ⟨*cases-parameter*⟩ | '`(`' ⟨*cases-parameter*⟩ ('`, `' ⟨*cases-parameter*⟩)+ '`)`'

⟨*cases-parameter*⟩ ::= ⟨*parameter*⟩ | '`cases`'

⟨*cases*⟩ ::= ⟨*case*⟩+ [⟨*default-case*⟩]

⟨*case*⟩ ::= '`\n`' ⟨*indentation*⟩ ⟨*matching*⟩ ('`, `' ⟨*matching*⟩)* '` => `' ⟨*simple-op-expr*⟩

⟨*default-case*⟩ ::= '`\n`' ⟨*indentation*⟩ '`... => `' ⟨*simple-op-expr*⟩

⟨*matching*⟩ ::= ⟨*literal*⟩ | ⟨*identifier*⟩ ['`:`' ⟨*identifier*⟩]

2.4 Value Definitions

2.4.1 Basic Structure

Examples

```
foo : Int
  = 42

val1, val2, val3 : Int, Bool, Char
```

```
  = 42, true, 'a'

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

succ : Int => Int
  = +1

f : (Int, Int, Int) => Int
  = (a, b, c) => a + b * c
```

Description

To define a new value you give it a name, a type and an expression. It is possible to group value definitions by seperating the names, the types and the expressions with commas. It is also possible to use the keyword ”all” to give the same type to all the values.

Grammar

⟨*value-definitions*⟩ ::= ⟨*identifiers*⟩ ‘␣:␣’ (⟨*types*⟩ | ‘`all`’ ⟨*type*⟩) ‘`\n`␣␣`=`’ ⟨*value-expressions*⟩

⟨*identifiers*⟩ ::= ⟨*identifier*⟩ (‘`,`␣’ ⟨*identifier*⟩)*

⟨*types*⟩ ::= ⟨*type*⟩ (‘`,`␣’ ⟨*type*⟩)*

⟨*value-expressions*⟩ ::= ⟨*value-expression*⟩ (‘`,`␣’ ⟨*value-expression*⟩)*

2.4.2 ”where” Expressions

Examples

```
sort : (A)HasOrder --> ListOf(A)s => ListOf(A)s
  = cases =>
    empty => empty
    non_empty:l => sort(less_l) + l.head + sort(greater_l)
      where
      less_l : ListOf(A)s
        = filter_with(x => x < l.head, l.tail)
      greater_l : ListOf(A)s
        = filter_with(x => x >= l.head, l.tail)

tuple_type Coeffs
value (previous, current) : Int x Int

tuple_type GcdAndCoeffs
value (gcd, a, b) : Int x Int x Int

ext_euc: (Int, Int) => GcdAndCoeffs
  = ext_euc_rec((1, 0), (0, 1))

ext_euc_rec: (Coeffs, Coeffs, Int, Int) => GcdAndCoeffs
  = (a_coeffs, b_coeffs, x, cases) =>
    0 => (x, a_coeffs.previous, b_coeffs.previous)
    y => ext_euc_rec(next <- a_coeffs, next <- b_coeffs, y, x -> mod <- y)
```

```
    where
    next: Coeffs => Coeffs
      = fields => (current, previous - x / y * current)

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 (i.e. they are not used anywhere outside that definition). In particular when the enclosing definition is a function definition and the expression to be reused, uses one or more of the function parameters, the only option is a "where" expression (in contrast to the situation where no parameters are used, in which case the definition could be moved outside as a new seperated definition). "where" expressions begin by a line that only has word "where" in it and is indented so that itfollows the indentation rules. The definitions are placed below the "where" line and must have the same indentation.

Grammar

⟨*where-expr*⟩ ::= '`\n`' ⟨*indentation*⟩ '`where\n`' ⟨*indented-value-definitions*⟩

⟨*indented-value-definitions*⟩ ::= TODO

2.4.3 "main" Value

2.5 Types

2.5.1 Type expressions

Examples

```
Int

String => String

Int x Int

Int x Int => Real

A => A

(A => B, B => C) => (A => C)

((A, A) => A, A, ListOf(A)s) => A

((B, A) => B, B, ListOf(A)s) => B

(T)HasStringRepresantion --> T => String
```

Description

Examples	Description
Int Char String	Base types
A => A (A => B, B => C) => (A => C)	Polymorphic types. A, B, C ... are type variables

Differences from Haskell

lcases	haskell	difference description
A => A	a => a	Type variables for polymorphic types are

Grammar

⟨*type*⟩ ::= ⟨*type-application*⟩ | ⟨*product-type*⟩ | ⟨*function-type*⟩

⟨*type-application*⟩ ::= [⟨*types-in-paren*⟩] ⟨*type-identifier*⟩ (⟨*types-in-paren*⟩ ([A-Za-z])*)* [⟨*types-in-paren*⟩]

⟨*types-in-paren*⟩ ::= '(' ⟨*type*⟩ (', ' ⟨*type*⟩)* ')'

⟨*type-identifier*⟩ ::= [A-Z] ([A-Za-z])*

⟨*product-type*⟩ ::= ⟨*product-subtype*⟩ ('␣x␣' ⟨*product-subtype*⟩)+

⟨*product-subtype*⟩ ::= '(' (⟨*function-type*⟩ | ⟨*product-type*⟩) ')' | ⟨*type-application*⟩

⟨*function-type*⟩ ::= ⟨*input-types-expression*⟩ '␣=>␣' ⟨*one-type*⟩

⟨*input-types-expression*⟩ ::= ⟨*one-type*⟩ | ⟨*two-or-more-types-in-paren*⟩

⟨*one-type*⟩ ::= ⟨*type-application*⟩ | ⟨*product-type*⟩ | '(' ⟨*function-type*⟩ ')'

⟨*two-or-more-types-in-paren*⟩ ::= '(' ⟨*type*⟩ (', ' ⟨*type*⟩)+ ')'

2.5.2 Tuple Types

Definition Examples

```
tuple_type Name
value (first_name, last_name) : String x String

tuple_type ClientInfo
value (name, age, nationality) : Name x Int x String

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

```
tuple_type (A)And(B)
value (a_value, b_value) : A x B

tuple_type (ExprT)WithPosition
value (expr, line, column) : ExprT x Int x Int
```

Usage Examples

```
giorgos_info : ClientInfo
  = (("Giorgos", "Papadopoulos"), 42, "Greek")

john_info : ClientInfo
  = (("John", "Doe"), 42, "American")

name_to_string : Name => String
  = fields => "First Name: " + first_name + "\nLast Name: " + last_name

print_name_and_nationality : ClientInfo => (EmptyValue)IOAction
  = fields => print(name -> name_to_string + "\nNationality: " + nationality)

print_error_in_expr : (SomeDefinedExprT)WithPosition => (EmptyValue)IOAction
  = ewp =>
    print(
      "Error in the expression:" + es +
      "\nAt the position: (" + ls + ", " + cs + ")"
    )
    where
    es, ls, cs : all String
      = ewp.expr->to_string, ewp.line->to_string, ewp.column->to_string
```

Description

Tuple types group many values into a single value. They are specified by their name, the names of their subvalues and the types of their subvalues. They generate projection functions for all of their subvalues by using a '.' before the name of the subvalue. For example the ClientInfo type above generates the following functions:

```
.name : ClientInfo => String
.age : ClientInfo => Int
.nationality : ClientInfo => String
```

These functions shall be named "postfix functions" as the can just be appended to their argument.

Definition Grammar

⟨*tuple-type-definition*⟩ ::=
  '`tuple_type␣`' ⟨*type-application*⟩ '`\nvalue␣`' '`(`' ⟨*identifier*⟩ ('`,␣`' ⟨*identifier*⟩)* '`)`' '`␣:␣`' ⟨*product-type*⟩

2.5.3 Or Types

Examples

```
or_type Bool
values true | false
```

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

or_type ListOf(A)s
values non_empty:HeadAndTailOf(A)s | empty

tuple_type HeadAndTailOf(A)s
value (head, tail) : A x ListOf(A)s

is_empty : ListOf(A)s => Bool
  = cases =>
    empty => true
    non_empty:anything => false

get_head : ListOf(A)s => Possibly(A)
  = cases =>
    empty => no_value
    non_empty:list => the_value:list.head
```

Description

Values of an or_type are one of many cases that possibly have other values inside. The cases which have other values inside are followed by a semicolon and the type of the internal value. The same syntax can be used for matching that particular case in a fucntion using the "cases" syntax, with the difference that after the colon, we write the name given to the value inside. Or_types and basic types are the only types on which the "cases" syntax can be used. The cases of an or_type which have a value inside create functions. For example, the case "non_empty" of a list creates the function "non_empty:" for which we can say:

```
non_empty: : HeadAndTailOf(A)s => ListOf(A)s
```

Similarly:

```
the_value: : A => Possibly(A)
```

These functions shall be named "prefix functions" as they are prepended to their argument. For example:

```
head_and_tail : HeadAndTailOf(Int)s
  = (1, [2, 3, 4])

list : ListOf(Int)s
  = non_empty:head_and_tail
```

These functions can be used like any other function as arguments to other functions. For example:

```
heads_and_tails_to_lists : ListOf(HeadAndTailOf(A)s)s => ListOf(ListOf(A)s)s
  = apply(non_empty:)to_each
```

Definition Grammar

⟨*or-type-definition*⟩ ::=
 '`or_type␣`' ⟨*type-application*⟩ '`\nvalues␣`' ⟨*identifier*⟩ ['`:`' ⟨*type*⟩] ('`␣|␣`' ⟨*identifier*⟩ ['`:`' ⟨*type*⟩])*

2.6 Type Logic

2.6.1 Type Predicate

2.6.2 Type Proposition

2.6.3 Type Theorem

2.7 Predefined

2.7.1 Functions

2.8 Grammar

2.8.1 Tokens

Keywords

`cases use_fields tuple_type or_type`

Value names

Type names

2.8.2 Core Grammar

Program

⟨*program*⟩ ::= (⟨*value-definitions*⟩ | ⟨*type-def*⟩)+

⟨*value-definitions*⟩ ::= ⟨*identifiers*⟩ '`␣:␣`' (⟨*types*⟩ | '`all`' ⟨*type*⟩) '`\n␣␣=`' ⟨*value-expressions*⟩

⟨*identifiers*⟩ ::= ⟨*identifier*⟩ ('`,␣`' ⟨*identifier*⟩)*

⟨*types*⟩ ::= ⟨*type*⟩ ('`,␣`' ⟨*type*⟩)*

⟨*value-expressions*⟩ ::= ⟨*value-expression*⟩ ('`,␣`' ⟨*value-expression*⟩)*

Types

Value Expressions

⟨*value-expression*⟩ ::= [⟨*input-expr*⟩] ⟨*cases-or-where*⟩ | ⟨*op-expr*⟩

⟨*cases-or-where*⟩ ::= ⟨*cases-expr*⟩ | ⟨*where-expr*⟩

⟨*where-expr*⟩ ::= ‘`let`’ ⟨*spicy-nl*⟩ (⟨*value-definitions*⟩ ⟨*spicy-nls*⟩)+ ‘`in`’ ⟨*value-expression*⟩ ⟨*spicy-nl*⟩

⟨*cases-expr*⟩ ::= ‘`cases`’ (⟨*case*⟩)+ [⟨*default-case*⟩]

3 lcases vs Haskell: Similarities and Differences

4 Parser implimentation

The parser was implemented using the parsec library.

4.1 AST Types

4.2 Parsers

5 Translation to Haskell

6 Running examples

7 Conclusion

8 To be removed or incorporated

Addition/Subtraction:

```
+ : (A)HasAddition => (A, A) => A
- : (A)HasSubtraction => (A, A) => A
```

Equality and ordering:

```
= : (A)HasEquality => (A, A) => Bool
<= : (A)HasOrder => (A, A) => Bool
>= : (A)HasOrder => (A, A) => Bool
```

(fmap)`<inside>` — (W)IsAWrapper `-->` (A `=>` B, W(A)) `=>` W(B) — Apply inside operator
`(<*>)<wrapped_inside>` — (W)IsAWrapper `-->` (W(A `=>` B), W(A)) `=>` W(B) — Order operators

better as postfix functions

Examples in Haskell

```
data ClientInfo =
  ClientInfoC String Int String

data WithPosition a =
  WithPositionC a Int Int

data Pair a b =
  PairC a b
```

Examples in Haskell

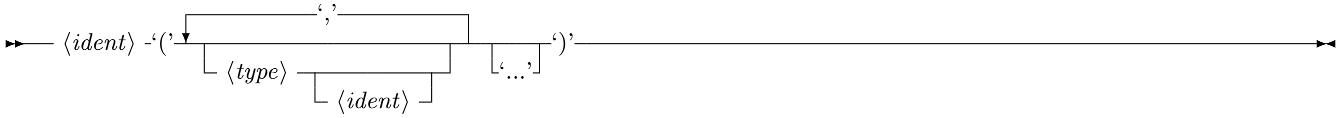
```
{-# language LambdaCase #-}

data Bool =
  Ctrue | Cfalse

data Possibly a =
  Cwrapper a | Cnothing

data ListOf_s a =
  Cnon_empty (NonEmptyValueListOf_s a) | Cempty

data NonEmptyValueListOf_s a =
  CNonEmptyValueListOf_s a (ListOf_s a)

is_empty :: ListOf_s a => Bool
is_empty = \case
  Cempty => Ctrue
  Cnon_empty (CNonEmptyValueListOf_s head tail) => Cfalse

get_head :: ListOf_s a => Possibly a
get_head = \case
  Cempty => Cnothing
  Cnon_empty (CNonEmptyValueListOf_s head tail) => Cwrapper head
```

Examples in Haskell

```
foo :: Int
foo = 42

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

int1 :: Int
int1 = 1
int2 :: Int
int2 = 2
int3 :: Int
int3 = 3

succ :: Int => Int
succ = \x => x + 1

f :: Int => Int => Int => Int
f = \a b c => a + b * c
```

Or Types the following have automatically generated functions:

```
is_case:
```



Hi

- *Examples*

- *Description*

  hi

- *Grammar*

  ⟨*identifier*⟩ ::= [a-z]([a-z_] | ‘()’[a-z_])* [[0-9]]