

# Lambda Cases (lcases)

Dimitris Saridakis

## Contents

|          |                                      |           |
|----------|--------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                  | <b>1</b>  |
| <b>2</b> | <b>Language Description</b>          | <b>2</b>  |
| 2.1      | Value Expressions . . . . .          | 3         |
| 2.1.1    | Literals . . . . .                   | 3         |
| 2.1.2    | Identifiers . . . . .                | 3         |
| 2.1.3    | Operators . . . . .                  | 4         |
| 2.1.4    | Tuples . . . . .                     | 4         |
| 2.1.5    | Lists . . . . .                      | 5         |
| 2.1.6    | Function Expressions . . . . .       | 5         |
| 2.1.7    | Special Function Arguments . . . . . | 6         |
| 2.1.8    | Expressions . . . . .                | 6         |
| 2.2      | Value Definitions . . . . .          | 6         |
| 2.3      | Types . . . . .                      | 7         |
| 2.3.1    | Type expressions . . . . .           | 7         |
| 2.3.2    | Tuple Types . . . . .                | 8         |
| 2.3.3    | Or Types . . . . .                   | 9         |
| 2.4      | Type Logic . . . . .                 | 11        |
| 2.4.1    | Type Predicate . . . . .             | 11        |
| 2.4.2    | Type Theorem . . . . .               | 11        |
| 2.5      | Grammar . . . . .                    | 11        |
| 2.5.1    | Tokens . . . . .                     | 11        |
| 2.5.2    | Core Grammar . . . . .               | 11        |
| <b>3</b> | <b>Parser implimentation</b>         | <b>12</b> |
| 3.1      | AST Types . . . . .                  | 12        |
| 3.2      | Parsers . . . . .                    | 12        |
| <b>4</b> | <b>Translation to Haskell</b>        | <b>12</b> |
| <b>5</b> | <b>Running examples</b>              | <b>12</b> |
| <b>6</b> | <b>Conclusion</b>                    | <b>12</b> |
| <b>7</b> | <b>To be removed or incorporated</b> | <b>12</b> |

## 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 enough students and corporations 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 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 value, type and predicate definitions along with type theorems. 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 algorithm

```
// type definitions

tuple_type PrevCoeffs
value (prev_prev, prev : Int, Int)

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

// algorithm

extended_euclidean: (Int, Int) -> GcdAndCoeffs
  = (init_a_coeffs, init_b_coeffs) ==> ee_recursion

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

ee_recursion: (PrevCoeffs, PrevCoeffs, Int, Int) -> GcdAndCoeffs
  = (a_coeffs, b_coeffs, x, cases) ->
    0 -> (x, a_coeffs.prev_prev, b_coeffs.prev_prev)
    y ->
      ee_recursion(next <== a_coeffs, next <== b_coeffs, y, x ==> mod <== y)
      where
        next: PrevCoeffs -> PrevCoeffs
          = fields -> (prev, prev_prev - x ==> div <== y * prev)

// reading, printing and main

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

print_gcd_and_coeffs : GcdAndCoeffs -> (Empty)WithIO
  = fields -> print("Gcd: " + gcd + "\nCoefficients: a = " + a + ", b = " + b)

main : (Empty)WithIO
```

```
= read_two_ints >>= ints ->
  extended_euclidean(ints.1st, ints.2nd) ==> print_gcd_and_coeffs
```

## 2.1 Value Expressions

### 2.1.1 Literals

Literals are the same as haskell

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

TODO add the grammar from the haskell report

### 2.1.2 Identifiers

#### Examples

x

y

z

-

funny\_identifier

unnecessarily\_long\_identifier

#### Description

An identifier is a string of lower case letters or underscore. It is used to give names to values when they are defined.

QUESTION maybe shouldn't let '\_' be an identifier

#### Grammar

$\langle identifier \rangle ::= ([a-z\_])^*$

### 2.1.3 Operators

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

| Operator                                | Precedence | Associativity |
|---|------------|---------------|
| <code>==&gt;</code>                     | 10         | Left          |
| <code>&lt;==</code>                     | 9          | Right         |
| <code>o&gt; &lt;o</code>                | 8          | Left          |
| <code>^</code>                          | 7          | Right         |
| <code>* /</code>                        | 6          | Left          |
| <code>+ -</code>                        | 5          | Left          |
| <code>= /= &gt; &lt; &gt;= &lt;=</code> | 4          | None          |
| <code>&amp;</code>                      | 3          | Left          |
| <code> </code>                          | 2          | Left          |
| <code>&gt;&gt;= ;</code>                | 1          | Left          |

### Grammar

$\langle operator-expression \rangle ::= \langle operator-argument \rangle \langle operator \rangle \langle operator-argument \rangle$

$\langle operator-argument \rangle ::= \langle literal \rangle \mid \langle identifier \rangle \mid \langle parenthesis-function-application \rangle$

$\langle operator \rangle ::= \text{'\_'} \langle op \rangle \text{'\_}'$

$\langle op \rangle ::= \text{'==>'} \mid \text{'<=='} \mid \text{'o>'} \mid \text{'<o'} \mid \text{'^'} \mid \text{'*'} \mid \text{'/'} \mid \text{'+'} \mid \text{'-'} \mid \text{'='} \mid \text{'/='} \mid \text{'>'} \mid \text{'<'} \mid \text{'>='} \mid \text{'<='} \mid \text{'&'} \mid \text{'|'} \mid \text{'>>='} \mid \text{';'}$

#### 2.1.4 Tuples

##### Examples

(1, 2)

(1, 2, 3.14)

(1, 'a', "hello")

(a, my\_val, 1)

(x -> x + 1, my\_function, x -> x \* 2)

##### Description

Tuples are used to group many values into one. The type of the tuple is either the product of the types of the subvalues or a defined `tuple_type` which is equivalent to the aforementioned product type i.e. the product type is in the definition of the `tuple_type` (see "tuple\_type" section 2.3.2).

##### Grammar

$\langle tuple \rangle ::= ' (' \langle value-expression \rangle ( ', \sqcup ' \langle value-expression \rangle ) + ' ) '$

#### 2.1.5 Lists

##### 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.

##### Grammar

$\langle list \rangle ::= '[' [ \langle value-expression \rangle ( ', \sqcup ' \langle value-expression \rangle ) * ] ' ] '$

#### 2.1.6 Function Expressions

##### Examples

a -> 17 \* a + 42

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

##### Description

Function expressions are used to define functions or pass anonymous functions as arguments to other functions. They are comprised by their parameters and their body. The parameters are either only one in which case a single identifier is used, or they are many in which case many identifiers are used in parenthesis, separated by a comma. The parameters and the body are separated by an arrow. The body is an operator expression.

##### Grammar

$\langle function-expression \rangle ::= \langle parameters \rangle ' \sqcup -> \sqcup ' \langle operator-expression \rangle$

$\langle parameters \rangle ::= \langle identifier \rangle | ' (' \langle identifier \rangle ( ', \sqcup ' \langle identifier \rangle ) + ' ) '$

### 2.1.7 Special Function Arguments

```
x -> body
(x, y, z) -> body
cases -> body
(x, cases, z) -> body
```

### 2.1.8 Expressions

#### Examples

```
42

x

funny_identifier

[1, 2, 3]

"Hello world!"

x ==> mod <== y

1.61 * 2.71 + 3.14

a -> 17 * a + 42

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

n ==> +1 ==> ^2 ==> *3 ==> print

f(x, y, z) + g(1, 2, 3)
```

#### Description

The base of expressions, are literals and identifiers, those can be combined either with operators, or by normal function application with mathematical notation. Finally, on top of that there can be added one or more abstractions (parameters) in the beginning of the expressions with an arrow.

#### Grammar

## 2.2 Value Definitions

#### 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 separating 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

$$\langle \textit{value-definitions} \rangle ::= \langle \textit{identifiers} \rangle \text{'\texttt{:}\texttt{'}} (\langle \textit{types} \rangle \mid \text{'\texttt{all}\texttt{'}} \langle \textit{type} \rangle) \text{'\texttt{=}\texttt{'}} \langle \textit{value-expressions} \rangle$$

$$\langle \textit{identifiers} \rangle ::= \langle \textit{identifier} \rangle (\text{'\texttt{,}\texttt{'}} \langle \textit{identifier} \rangle)^*$$

$$\langle \textit{types} \rangle ::= \langle \textit{type} \rangle (\text{'\texttt{,}\texttt{'}} \langle \textit{type} \rangle)^*$$

$$\langle \textit{value-expressions} \rangle ::= \langle \textit{value-expression} \rangle (\text{'\texttt{,}\texttt{'}} \langle \textit{value-expression} \rangle)^*$$

## 2.3 Types

### 2.3.1 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<br>Char<br>String                  | Base types  |
| A -> A<br>(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

$\langle type \rangle ::= \langle type-application \rangle \mid \langle product-type \rangle \mid \langle function-type \rangle$

$\langle type-application \rangle ::= [ \langle types-in-paren \rangle ] \langle type-identifier \rangle ( \langle types-in-paren \rangle ( [A-Za-z] )^* )^* [ \langle types-in-paren \rangle ]$

$\langle types-in-paren \rangle ::= ' ( \langle type \rangle ( ' , ' \langle type \rangle )^* ' )'$

$\langle type-identifier \rangle ::= [A-Z] ( [A-Za-z] )^*$

$\langle product-type \rangle ::= \langle product-subtype \rangle ( ' \_x \_ ' \langle product-subtype \rangle )^+$

$\langle product-subtype \rangle ::= ' ( ( \langle function-type \rangle \mid \langle product-type \rangle ) ' ) ' \mid \langle type-application \rangle$

$\langle function-type \rangle ::= \langle input-types-expression \rangle ' \_ \rightarrow \_ ' \langle one-type \rangle$

$\langle input-types-expression \rangle ::= \langle one-type \rangle \mid \langle two-or-more-types-in-paren \rangle$

$\langle one-type \rangle ::= \langle type-application \rangle \mid \langle product-type \rangle \mid ' ( \langle function-type \rangle ' )'$

$\langle two-or-more-types-in-paren \rangle ::= ' ( \langle type \rangle ( ' , ' \langle type \rangle )^+ ' )'$

#### 2.3.2 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 -> (Empty)WithIO
  = fields -> print(name ==> name_to_string + "\nNationality: " + nationality)

print_error_in_expr : (SomeDefinedExprT)WithPosition -> (Empty)WithIO
  = 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 sub-values 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 they can just be appended to their argument.

## Definition Grammar

$\langle \text{tuple-type-definition} \rangle ::=$   
 $\text{'tuple\_type\_'} \langle \text{type-application} \rangle \text{'\nvalue\_'} \text{'('} \langle \text{identifier} \rangle \text{'(, ' } \langle \text{identifier} \rangle \text{'*) ' } \text{'\_:'} \langle \text{product-type} \rangle$

### 2.3.3 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 : A, tail : 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 function 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

$\langle or\_type\_definition \rangle ::=$   
 $\text{'or\_type\_'} \langle type\_application \rangle \text{'\nvalues\_'} \langle identifier \rangle [ \text{' : ' } \langle type \rangle ] ( \text{' | ' } \langle identifier \rangle [ \text{' : ' } \langle type \rangle ] )^*$

## 2.4 Type Logic

### 2.4.1 Type Predicate

### 2.4.2 Type Theorem

## 2.5 Grammar

### 2.5.1 Tokens

#### Keywords

cases use\_fields tuple\_type or\_type

#### Value names

#### Type names

### 2.5.2 Core Grammar

#### Program

$\langle program \rangle ::= (\langle value\_definitions \rangle \mid \langle type\_def \rangle)^+$

$\langle value\_definitions \rangle ::= \langle identifiers \rangle ' \sqcup : \sqcup ' (\langle types \rangle \mid 'all' \langle type \rangle) ' \backslash n \sqcup \sqcup = ' \langle value\_expressions \rangle$

$\langle identifiers \rangle ::= \langle identifier \rangle ( ' , \sqcup ' \langle identifier \rangle )^*$

$\langle types \rangle ::= \langle type \rangle ( ' , \sqcup ' \langle type \rangle )^*$

$\langle value\_expressions \rangle ::= \langle value\_expression \rangle ( ' , \sqcup ' \langle value\_expression \rangle )^*$

#### Types

#### Value Expressions

$\langle value\_expression \rangle ::= [ \langle input\_expr \rangle ] \langle cases\_or\_where \rangle \mid \langle op\_expr \rangle$

$\langle cases\_or\_where \rangle ::= \langle cases\_expr \rangle \mid \langle where\_expr \rangle$

$\langle where\_expr \rangle ::= 'let' \langle spicy\_nl \rangle (\langle value\_definitions \rangle \langle spicy\_nls \rangle)^+ 'in' \langle value\_expression \rangle \langle spicy\_nl \rangle$

$\langle cases\_expr \rangle ::= 'cases' ( \langle case \rangle )^+ [ \langle default\_case \rangle ]$

### 3 Parser implimentation

The parser was implemented using the parsec library.

#### 3.1 AST Types

#### 3.2 Parsers

### 4 Translation to Haskell

### 5 Running examples

### 6 Conclusion

### 7 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 (NonEmptyListOf_s a) | Cempty

data NonEmptyListOf_s a =
  CNonEmptyListOf_s a (ListOf_s a)

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

get_head :: ListOf_s a -> Possibly a
get_head = \case
  Cempty -> Cnothing
  Cnon_empty (CNonEmptyListOf_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:

