

Lambda Cases (lcases)

Dimitris Saridakis

Contents

1	Introduction	2
2	Language Description	2
2.1	Basic Expressions	3
2.1.1	Literals and Identifiers	3
2.1.2	Tuples and Lists	4
2.2	Operators	5
2.2.1	Function Application Operators	5
2.2.2	Function Composition Operators	5
2.2.3	Arithmetic Operators	6
2.2.4	Comparison and Boolean Operators	7
2.2.5	Environment Operators	7
2.2.6	Complete Table, Precedence, Associativity and Grammar	9
2.3	Functions	10
2.3.1	Parenthesis Function Application	10
2.3.2	Function Expressions	10
2.3.3	Special Function Parameters	11
2.3.4	Expressions	11
2.4	Value Definitions	12
2.5	Types	12
2.5.1	Type expressions	12
2.5.2	Tuple Types	13
2.5.3	Or Types	14
2.6	Type Logic	16
2.6.1	Type Predicate	16
2.6.2	Type Statement	16
2.6.3	Type Theorem	16
2.7	Grammar	16
2.7.1	Tokens	16
2.7.2	Core Grammar	16
3	Notable differences from Haskell	17
4	Parser implimentation	17
4.1	AST Types	17
4.2	Parsers	17
5	Translation to Haskell	17
6	Running examples	17
7	Conclusion	17

1 Introduction

Haskell is a delightful language. Yet, for some reason, 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 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 Coeffs
value (previous, current) : Int x Int

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

// algorithm

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

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

ee_recursion: (Coeffs, Coeffs, Int, Int) => GcdAndCoeffs
  = (a_coeffs, b_coeffs, x, cases) =>
    0 => (x, a_coeffs.previous, b_coeffs.previous)
    y =>
      ee_recursion(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)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_io
  ... => io_error <- "You didn't give me 2 ints"

print_gcd_and_coeffs : GcdAndCoeffs => (EmptyValue)WithIO
= fields => print("Gcd: " + gcd + "\nCcoefficients: a = " + a + ", b = " + b)

main : (EmptyValue)WithIO
= read_two_ints ;> ints =>
  extended_euclidean(ints.1st, ints.2nd) -> print_gcd_and_coeffs

```

2.1 Basic 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 numbers, characters and strings. The exact specification of literals is the same as in the Haskell report.

QUESTION integers are different?

- *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 is also possible to have a pair of parentheses in the middle of an identifier (see "parenthesis function application" section 2.3.1). Finally, an identifier can be ended with a digit.

- *Grammar*

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

2.1.2 Tuples and Lists

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 aforementioned 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*

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

Lists

- *Examples*

```
[1, 2, 17, 42, -100]
[1.61, 2.71, 3.14, -1234.567]
["Hello World!", "What's up, doc?", "Alrighty then!"]
[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 three examples are:

```
ListOf(Int)s
ListOf(Real)s
ListOf(String)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*

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

2.2 Operators

2.2.1 Function Application Operators

The function application operators " \rightarrow " and " \leftarrow " are a different way to apply functions to arguments than the usual parenthesis function application. Each one applies the function from the corresponding direction. The operators 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 : (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(is_odd) -> sum_ints
```

Ofcourse this can be done equivalently using the other operator:

```
chars_in_odd_length_strings : Int
= sum_ints <- filter(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)
```

Operator	Type
\rightarrow	$(A, A \Rightarrow B) \Rightarrow B$
\leftarrow	$(A \Rightarrow B, A) \Rightarrow B$

2.2.2 Function Composition Operators

The function composition operators " $\circ>$ " and " $\circ<$ " are used to compose funtions, 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_word: 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_word)to_all o> merge_words
```

Ofcourse this can be done equivalently using the other operator:

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

Operator	Type
o>	$(A \Rightarrow B, B \Rightarrow C) \Rightarrow (A \Rightarrow C)$
<o	$(B \Rightarrow C, A \Rightarrow B) \Rightarrow (A \Rightarrow C)$

2.2.3 Arithmetic Operators

The usual arithmetic operators work as they are expected, similarly to mathematics and other programming languages. 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"
```

The generality can also be seen from their types in the table below:

Operator	Type
\sim	$(A)ToThe(B)Gives(C) \dashrightarrow (A, B) \Rightarrow C$
$*$	$(A)And(B)MultiplyTo(C) \dashrightarrow (A, B) \Rightarrow C$
$/$	$(A)Divides(B)To(C) \dashrightarrow (B, A) \Rightarrow C$
$+$	$(A)And(B)AddTo(C) \dashrightarrow (A, B) \Rightarrow C$
$-$	$(A)SubtractsFrom(B)To(C) \dashrightarrow (B, A) \Rightarrow C$

Let's analyze further the example of addition. The type can be read as such: the '+' operator has the type $(A, B) \Rightarrow C$, provided that the type statement $(A)And(B)AddTo(C)$ holds. This statement 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 see 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 they 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

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 (`== && ||`).

Operator	Type	Description
<code>= /=</code>	<code>(A)HasEquality --> (A, A) => Bool</code>	Equality operators
<code>> < >= <=</code>	<code>(A)HasOrder --> (A, A) => Bool</code>	Order operators
<code>& </code>	<code>(Bool, Bool) => Bool</code>	Boolean operators

2.2.5 Environment Operators

Example program

```

main : (EmptyValue)WithIO
  = 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 + "!"
    );

```

The example above demonstrates the use of the environment operators in the "WithIO" environment, 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)WithIO
get_line    : (String)WithIO

print_string <- "Hello! ... " : (EmptyValue)WithIO
print_string("Oh hi...")    : (EmptyValue)WithIO
print_string("Oh that's crazy...") : (EmptyValue)WithIO

; : ((EmptyValue)WithIO, (String)WithIO) => (String)WithIO
print_string("Oh hi...") ; get_line : (String)WithIO

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

;> : ((String)WithIO, String => (EmptyValue)WithIO) => (EmptyValue)WithIO

print_string("Oh hi...") ; get_line ;> age => print_string("Oh that's crazy...")
  : (EmptyValue)WithIO

```

```
name => print_string("Oh hi ... (till the end)  :  String => (EmptyValue)WithIO
```

```
print_string <- "Hello..." ; get_line  :  (String)WithIO
```

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

```
therefore (main  :  (EmptyValue)WithIO) 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 occurrences of the ";">" operator. Also, the (actual/most general) types of the operators are shown in the table below:

Operator	Type	Equivalent in Haskell
;>	$(E)IsAnEnvironment \rightarrow (E(A), A \Rightarrow E(B)) \Rightarrow E(B)$	$\gg=$
;	$(E)IsAnEnvironment \rightarrow (E(A), E(B)) \Rightarrow E(B)$	\gg

In this particular case we have:

- For ";">":

```
E = WithIO
A = String
B = EmptyValue
```

- For ";"":

```
E = WithIO
A = EmptyValue
B = String
```

- $(WithIO)IsAnEnvironment$ is a true statement

2.2.6 Complete Table, Precedence, Associativity and Grammar

Operator	Type	Description
\rightarrow	$(A, A \Rightarrow B) \Rightarrow B$	Right function application
\leftarrow	$(A \Rightarrow B, A) \Rightarrow B$	Left function application
$\circ\rightarrow$	$(A \Rightarrow B, B \Rightarrow C) \Rightarrow (A \Rightarrow C)$	Right function composition
$\leftarrow\circ$	$(B \Rightarrow C, A \Rightarrow B) \Rightarrow (A \Rightarrow C)$	Left function composition
\wedge	$(A)\text{ToThe}(B)\text{Gives}(C) \rightarrow (A, B) \Rightarrow C$	General exponentiation
$*$	$(A)\text{And}(B)\text{MultiplyTo}(C) \rightarrow (A, B) \Rightarrow C$	General multiplication
$/$	$(A)\text{Divides}(B)\text{To}(C) \rightarrow (B, A) \Rightarrow C$	General division
$+$	$(A)\text{And}(B)\text{AddTo}(C) \rightarrow (A, B) \Rightarrow C$	General addition
$-$	$(A)\text{SubtractsFrom}(B)\text{To}(C) \rightarrow (B, A) \Rightarrow C$	General subtraction
$=$ $/=$	$(A)\text{HasEquality} \rightarrow (A, A) \Rightarrow \text{Bool}$	Equality operators
$>$ $<$ \geq \leq	$(A)\text{HasOrder} \rightarrow (A, A) \Rightarrow \text{Bool}$	Order operators
$\&$ $ $	$(\text{Bool}, \text{Bool}) \Rightarrow \text{Bool}$	Boolean operators
$;>$	$(E)\text{IsAnEnvironment} \rightarrow (E(A), A \Rightarrow E(B)) \Rightarrow E(B)$	Monad bind
$;$	$(E)\text{IsAnEnvironment} \rightarrow (E(A), E(B)) \Rightarrow E(B)$	Monad then

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

Grammar

$$\langle operator-expression \rangle ::= \langle operator-argument \rangle \langle operator \rangle \langle right-operator-argument \rangle$$
$$\langle \textit{operator-argument} \rangle ::= \langle \textit{literal} \rangle \mid \langle \textit{identifier} \rangle \mid \langle \textit{parentheses-function-application} \rangle \mid \langle \textit{parentheses-expression} \rangle$$
$$\langle \textit{right-operator-argument} \rangle ::= \langle \textit{operator-argument} \rangle \mid \langle \textit{function-expression} \rangle$$
$$\langle operator \rangle ::= \text{'_'} \langle op \rangle \text{'_}'$$
$$\langle op \rangle ::= \text{'-'} \mid \text{'<'} \mid \text{'o'} \mid \text{'<o'} \mid \text{'\^{'}} \mid \text{'*'} \mid \text{'/'} \mid \text{'+'} \mid \text{'-'} \mid \text{'='} \mid \text{'/='} \mid \text{'>'} \mid \text{'<'} \mid \text{'>='} \mid \text{'<='} \mid \text{'\&'} \mid \text{'|'} \mid \text{'>'} \mid \text{';'}$$

2.3 Functions

2.3.1 Parenthesis Function Application

Examples

```
f(x)
f(x, y, z)
(x)to_string
apply(f)to_all
apply(f)to_all(1)
```

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 separated 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 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.

It is possible that many parentheses 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

```
<parenthesis-function-application> ::=
  <arguments> ( <identifier-with-arguments> | <identifier> )
  | ( <identifier-with-arguments> | <identifier> ) <arguments>
  | <identifier-with-arguments>

<arguments> ::= ' (' <value-expression> ( ' , ' <value-expression> )* ' ) '

<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_] '
```

2.3.2 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 parentheses, separated by a comma. The parameters and the body are separated by an arrow. The body is an operator expression.

Grammar

$\langle \text{function-expression} \rangle ::= \langle \text{parameters} \rangle \text{'\(\Rightarrow\)' } \langle \text{operator-expression} \rangle$

$\langle \text{parameters} \rangle ::= \langle \text{identifier} \rangle \mid \text{'(' } \langle \text{identifier} \rangle \text{' , ' } \langle \text{identifier} \rangle \text{') + '}'$

2.3.3 Special Function Parameters

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

2.3.4 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

2.4 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

$$\langle \text{value-definitions} \rangle ::= \langle \text{identifiers} \rangle \text{'_:_'} (\langle \text{types} \rangle \mid \text{'all'} \langle \text{type} \rangle) \text{'_=_'} \langle \text{value-expressions} \rangle$$
$$\langle \text{identifiers} \rangle ::= \langle \text{identifier} \rangle (\text{'_,'} \langle \text{identifier} \rangle)^*$$
$$\langle \text{types} \rangle ::= \langle \text{type} \rangle (\text{'_,'} \langle \text{type} \rangle)^*$$
$$\langle \text{value-expressions} \rangle ::= \langle \text{value-expression} \rangle (\text{'_,'} \langle \text{value-expression} \rangle)^*$$

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)HasStringRepresentation --> 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

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

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

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

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

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

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

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

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

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

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

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

print_error_in_expr : (SomeDefinedExprT)WithPosition => (EmptyValue)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

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

2.5.3 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 : 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

```

⟨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 Statement

2.6.3 Type Theorem

2.7 Grammar

2.7.1 Tokens

Keywords

cases use_fields tuple_type or_type

Value names

Type names

2.7.2 Core Grammar

Program

$\langle \text{program} \rangle \quad ::= (\langle \text{value-definitions} \rangle \mid \langle \text{type-def} \rangle)^+$

$\langle \text{value-definitions} \rangle \quad ::= \langle \text{identifiers} \rangle \text{'}_:_ \text{' } (\langle \text{types} \rangle \mid \text{'all' } \langle \text{type} \rangle) \text{'}\backslash \mathbf{n_}_ \text{' } \langle \text{value-expressions} \rangle$

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

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

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

Types

Value Expressions

$\langle \text{value-expression} \rangle \quad ::= [\langle \text{input-expr} \rangle] \langle \text{cases-or-where} \rangle \mid \langle \text{op-expr} \rangle$

$\langle \text{cases-or-where} \rangle \quad ::= \langle \text{cases-expr} \rangle \mid \langle \text{where-expr} \rangle$

$\langle \text{where-expr} \rangle \quad ::= \text{'let' } \langle \text{spicy-nl} \rangle (\langle \text{value-definitions} \rangle \langle \text{spicy-nls} \rangle)^+ \text{'in' } \langle \text{value-expression} \rangle \langle \text{spicy-nl} \rangle$

$\langle \text{cases-expr} \rangle \quad ::= \text{'cases' } (\langle \text{case} \rangle)^+ [\langle \text{default-case} \rangle]$

3 Notable differences from Haskell

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*

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