

Lambda Cases (lcases)

Dimitris Saridakis

Contents

1	Introduction	3
2	Language Description: General	3
2.1	Program Structure	3
2.2	Keywords	4
3	Language Description: Values	5
3.1	Basic Expressions	5
3.1.1	Literals and Identifiers	5
3.1.2	Parenthesis, Tuples and Lists	6
3.1.3	Parenthesis Function Application	10
3.1.4	Prefix and Postfix Functions	12
3.2	Operators	15
3.2.1	Function Application and Function Composition Operators	15
3.2.2	Arithmetic, Comparison and Boolean Operators	17
3.2.3	Environment Action Operators	19
3.2.4	Operator Expressions	22
3.2.5	Complete Table, Precedence and Associativity	24
3.3	Function Expressions	26
3.3.1	Regular Function Expressions	26
3.3.2	"cases" Function Expressions	27
3.4	Value Definitions and "where" Expressions	30
3.4.1	Value Definitions	30
3.4.2	"where" Expressions	31
4	Language Description: Types and Type Logic	32
4.1	Types	32
4.1.1	Type Expressions	32
4.1.2	Type Definitions	37
4.2	Type Logic	40
4.2.1	Proposition Definitions	40
4.2.2	Theorems	43
5	Language Description: Predefined	47
5.1	Values	47
5.2	Types	47
5.3	Type Propositions	48

6	Parser Implementation	50
6.1	Full grammar and indentation system	50
6.1.1	Full grammar	50
6.1.2	Indentation system	56
6.2	High level structure	57
6.2.1	Parsec library	57
6.2.2	File structure	57
6.3	Parser Examples	58
6.3.1	Parser Class and Example 0: Literal	58
6.3.2	Example 1: List	59
6.3.3	Example 2: Change	59
6.3.4	Example 3: Value Definition	60
7	Translation to Haskell	61
7.1	High Level Overview	61
7.2	Translation Phase	62
7.3	Translation Phase: Basic Expressions	63
7.3.1	Literals and Identifiers	63
7.3.2	Parenthesis, Tuples and Lists	63
7.3.3	Parenthesis Function Application	65
7.3.4	Prefix and Postfix Functions	65
7.4	Translation Phase: Operators	67
7.4.1	Operators	67
7.4.2	Operator Expressions	68
7.5	Translation Phase: Function Expressions	69
7.5.1	Regular Function Expressions	69
7.5.2	"cases" Function Expressions	70
7.6	Translation Phase: Value Definitions and "where" Expressions	71
7.7	Translation Phase: Types	73
7.7.1	Type Expressions	73
7.7.2	Type Definitions	76
7.8	Translation Phase: Type Logic	79
7.8.1	Proposition Definitions	79
7.8.2	Theorems	80
8	Running	82
9	Conclusion	83

1 Introduction

Haskell is a delightful language. 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 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: General

2.1 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 3.2.3) 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)) -> (_)&with_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'
```

2.2 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
<code>cases</code>	3.3.2 "cases" Function Expressions
<code>all where</code>	3.4 Value Definitions and "where" Expressions
<code>tuple_type value or_type values type_nickname</code>	4.1 Types
<code>type_proposition needed equivalent type_theorem proof</code>	4.2 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 [3.1.1](#)).

3 Language Description: Values

3.1 Basic Expressions

3.1.1 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*

$$\langle literal \rangle ::= \langle int-lit \rangle \mid \langle real-lit \rangle \mid \langle char-lit \rangle \mid \langle string-lit \rangle$$

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 3.1.3 for why this can be useful).

A simple identifier is an identifier without any underscores in parenthesis. It is used in expressions where underscores in parenthesis don't make sense (e.g. "Prefix and Postfix Functions" 3.1.4).

- *Grammar*

$$\langle identifier \rangle ::= [\langle unders-in-paren \rangle] \langle id-start \rangle \langle id-cont \rangle^* [[0-9]] [\langle unders-in-paren \rangle]$$
$$\langle simple-id \rangle ::= \langle id-start \rangle [[0-9]]$$
$$\langle id-start \rangle ::= [a-z] [a-z_]*$$
$$\langle id-cont \rangle ::= \langle unders-in-paren \rangle [a-z_]+$$
$$\langle unders-in-paren \rangle ::= '(_' (\langle comma \rangle ' _ ')^* ')'$$
$$\langle comma \rangle ::= ',' ['\u00a0']$$

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

3.1.2 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 over multiple lines new values must be defined.

- *Grammar*

$$\langle \text{paren-expr} \rangle ::= ' (' [_] \langle \text{line-op-expr} \rangle | \langle \text{line-func-expr} \rangle [_] ')'$$

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 4.1.2 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 [3.4.1](#)). 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" [6.1.2](#).

- *Tuples with empty fields*

Example

```
(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*

$$\begin{aligned}
\langle tuple \rangle &::= \langle ' [\text{'_'}] \langle line\text{-}expr\text{-}or\text{-}under \rangle \langle comma \rangle \langle line\text{-}expr\text{-}or\text{-}unders \rangle [\text{'_'}] \rangle' \\
\langle line\text{-}expr\text{-}or\text{-}unders \rangle &::= \langle line\text{-}expr\text{-}or\text{-}under \rangle (\langle comma \rangle \langle line\text{-}expr\text{-}or\text{-}under \rangle)^* \\
\langle line\text{-}expr\text{-}or\text{-}under \rangle &::= \langle line\text{-}expr \rangle | \text{'_'} \\
\langle line\text{-}expr \rangle &::= \langle basic\text{-}or\text{-}app\text{-}expr \rangle | \langle line\text{-}op\text{-}expr \rangle | \langle line\text{-}func\text{-}expr \rangle \\
\langle basic\text{-}or\text{-}app\text{-}expr \rangle &::= \langle basic\text{-}expr \rangle | \langle pre\text{-}func\text{-}app \rangle | \langle post\text{-}func\text{-}app \rangle \\
\langle basic\text{-}expr \rangle &::= \langle literal \rangle | \langle paren\text{-}func\text{-}app\text{-}or\text{-}id \rangle | \langle special\text{-}id \rangle | \langle tuple \rangle | \langle list \rangle \\
\\
\langle big\text{-}tuple \rangle &::= \\
&\langle ' [\text{'_'}] \langle line\text{-}expr\text{-}or\text{-}under \rangle [\langle nl \rangle \langle indent \rangle] \langle comma \rangle \langle line\text{-}expr\text{-}or\text{-}unders \rangle \\
&(\langle nl \rangle \langle indent \rangle \langle comma \rangle \langle line\text{-}expr\text{-}or\text{-}unders \rangle)^* \langle nl \rangle \langle indent \rangle \rangle'
\end{aligned}$$

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 [3.4.1](#)). 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" [6.1.2](#).

- *Grammar*

$$\langle list \rangle ::= '['['\] [\langle line\text{-}expr\text{-}or\text{-}unders \rangle] ['\]]'$$

$$\langle big\text{-}list \rangle ::= '['['\] \langle line\text{-}expr\text{-}or\text{-}unders \rangle (\langle nl \rangle \langle indent \rangle \langle comma \rangle \langle line\text{-}expr\text{-}or\text{-}unders \rangle)^* \langle nl \rangle \langle indent \rangle]'$$

3.1.3 Parenthesis Function Application

- *Examples*

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

- *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*

$$\langle \text{paren-func-app-or-id} \rangle ::= [\langle \text{arguments} \rangle] \langle \text{id-start} \rangle (\langle \text{arguments} \rangle [\text{a-z_}]^+)^* [[0-9]] [\langle \text{arguments} \rangle]$$

$$\langle \text{arguments} \rangle ::= ' ([_] \langle \text{line-expr-or-unders} \rangle [_])'$$

3.1.4 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 [4.1.2](#)). 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*

$\langle pre-func \rangle ::= \langle simple-id \rangle \text{':'}$

$\langle pre-func-app \rangle ::= \langle pre-func \rangle \langle operand \rangle$

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 4.1.2). 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: `"_.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*

$\langle post\text{-}func \rangle ::= \text{'.'} (\langle simple\text{-}id \rangle | \langle special\text{-}id \rangle)$

$\langle special\text{-}id \rangle ::= \text{'1st'} | \text{'2nd'} | \text{'3rd'} | \text{'4th'} | \text{'5th'}$

$\langle post\text{-}func\text{-}app \rangle ::= (\langle basic\text{-}expr \rangle | \langle paren\text{-}expr \rangle | \text{'_'}) (\langle dot\text{-}change \rangle | \langle post\text{-}func \rangle + [\langle dot\text{-}change \rangle])$

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 on 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> ::= [ '␣' ] '=' [ '␣' ]
```

3.2 Operators

3.2.1 Function Application and Function Composition Operators

Function Application Operators

Operator	Type
<code>-></code>	$T1 \times (T1 \Rightarrow T2) \Rightarrow T2$
<code><-</code>	$(T1 \Rightarrow T2) \times T1 \Rightarrow T2$

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
<code>o></code>	$(T1 \Rightarrow T2) \times (T2 \Rightarrow T3) \Rightarrow (T1 \Rightarrow T3)$
<code><o</code>	$(T2 \Rightarrow T3) \times (T1 \Rightarrow T2) \Rightarrow (T1 \Rightarrow T3)$

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 'o' 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
```


3.2.2 Arithmetic, Comparison and Boolean Operators

Arithmetic Operators

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

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" [4.2](#) 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)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
<code>==</code>	<code>(@A)And(@B)Can_Be_Equal --> @A x @B => Bool</code>
<code>!=</code>	<code>(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool</code>
<code>></code>	<code>(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool</code>
<code><</code>	<code>(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool</code>
<code>>=</code>	<code>(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool</code>
<code><=</code>	<code>(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool</code>
<code>&</code>	<code>(@A)Has_And --> @A^2 => @A</code>
<code> </code>	<code>(@A)Has_Or --> @A^2 => @A</code>

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 (`|`).

3.2.3 Environment Action Operators

Operator	Type
<code>></code>	$(@E)Has_Use \dashrightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$
<code>;</code>	$(@E)Has_Then \dashrightarrow @E(T1) \times @E(T2) \Rightarrow @E(T2)$

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: `;` : $(@E)Has_Then \dashrightarrow @E(T1) \times @E(T2) \Rightarrow @E(T2)$

In the following expression: `print_string("I'll repeat the line") ; get_line`
the "then" operator has type: $(EmptyVal)FromIO \times (String)FromIO \Rightarrow SomeType$

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: `>` : $(@E)Has_Use \dashrightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$

In the following expression: `print_string("I'll repeat the line") ; get_line ;> print_string(_)`
the "use" operator has type: $(String)FromIO \times (String \Rightarrow (EmptyVal)FromIO) \Rightarrow SomeType$

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 actions are type functions that take a type argument and produce a type (just like `ListOf(_)`s). These type functions 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 that 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 of type `T2` and therefore the new value is of type `@E(T2)`.

3.2.4 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 use operators. 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 Table, Precedence and Associativity") in Table 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 it begins at the column where the first line of the operator expression began.
- The precise indentation rules are described in the section "Indentation System" [6.1.2](#).

- *Grammar*

$$\langle op\text{-}expr \rangle ::= \langle line\text{-}op\text{-}expr \rangle \mid \langle big\text{-}op\text{-}expr \rangle$$
$$\langle op\text{-}expr\text{-}start \rangle ::= (\langle operand \rangle \langle op \rangle) +$$
$$\langle line-op-expr \rangle ::= \langle op-expr-start \rangle (\langle operand \rangle \mid \langle line-func-expr \rangle)$$
$$\langle \textit{big-op-expr} \rangle ::= \langle \textit{big-op-expr-op-split} \rangle \mid \langle \textit{big-op-expr-func-split} \rangle$$
$$\langle big-op-expr-op-split \rangle ::= \langle op-split-line \rangle + [\langle op-expr-start \rangle] (\langle operand \rangle \mid \langle func-expr \rangle)$$
$$\langle op-split-line \rangle ::= (\langle op-expr-start \rangle (\langle nl \rangle \mid \langle oper-fco \rangle) \mid \langle oper-fco \rangle) \langle indent \rangle$$
$$\langle oper-fco \rangle ::= \langle operand \rangle \text{ '}_\sqcup\text{' } \langle func-comp-op \rangle \text{ '\n'}$$
$$\langle \textit{big-op-expr-func-split} \rangle ::= \langle \textit{op-expr-start} \rangle (\langle \textit{big-func-expr} \rangle \mid \langle \textit{cases-func-expr} \rangle)$$
$$\langle operand \rangle ::= \langle basic\text{-}or\text{-}app\text{-}expr \rangle \mid \langle paren\text{-}expr \rangle \mid \text{'_'} \mid$$
$$\langle op \rangle ::= \text{'_'} \langle func-comp-op \rangle \text{'_'} \mid [\text{'_'}] \langle optional-spaces-op \rangle [\text{'_'}]$$
$$\langle func-comp-op \rangle ::= 'o>' \mid '<o'$$
$$\langle optional-spaces-op \rangle ::= \text{'>'} \mid \text{'<'} \mid \text{'\^{'}} \mid \text{'*'} \mid \text{'/'} \mid \text{'+'} \mid \text{'-'} \mid \text{'=='} \mid \text{'!='} \mid \text{'>'} \mid \text{'<'} \mid \text{'>='} \mid \text{'<='} \mid \text{'&'} \mid \text{'|'} \mid \text{'>'} \mid \text{';'}$$

3.2.5 Complete Table, Precedence and Associativity

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

Operator	Type	Description
\rightarrow	$T1 \times (T1 \Rightarrow T2) \Rightarrow T2$	Right Function Application
\leftarrow	$(T1 \Rightarrow T2) \times T1 \Rightarrow T2$	Left Function Application
$\circ>$	$(T1 \Rightarrow T2) \times (T2 \Rightarrow T3) \Rightarrow (T1 \Rightarrow T3)$	Right Function Composition
$<\circ$	$(T2 \Rightarrow T3) \times (T1 \Rightarrow T2) \Rightarrow (T1 \Rightarrow T3)$	Left Function Composition
\wedge	$(@A)\text{To_The}(@B)\text{Is}(@C) \dashrightarrow @A \times @B \Rightarrow @C$	General Exponentiation
$*$	$(@A)\text{And}(@B)\text{Multiply_To}(@C) \dashrightarrow @A \times @B \Rightarrow @C$	General Multiplication
$/$	$(@A)\text{Divided_By}(@B)\text{Is}(@C) \dashrightarrow @A \times @B \Rightarrow @C$	General Division
$+$	$(@A)\text{And}(@B)\text{Add_To}(@C) \dashrightarrow @A \times @B \Rightarrow @C$	General Addition
$-$	$(@A)\text{Minus}(@B)\text{Is}(@C) \dashrightarrow @A \times @B \Rightarrow @C$	General Subtraction
$==$	$(@A)\text{And}(@B)\text{Can_Be_Equal} \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Equality
$!=$	$(@A)\text{And}(@B)\text{Can_Be_Unequal} \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Inequality
$>$	$(@A)\text{Can_Be_Greater_Than}(@B) \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Greater Than
$<$	$(@A)\text{Can_Be_Less_Than}(@B) \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Less Than
$>=$	$(@A)\text{Can_Be_Gr_Or_Eq_To}(@B) \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Greater Than or Equal To
$<=$	$(@A)\text{Can_Be_Le_Or_Eq_To}(@B) \dashrightarrow @A \times @B \Rightarrow \text{Bool}$	General Less Than or Equal To
$\&$	$(@A)\text{Has_And} \dashrightarrow @A^2 \Rightarrow @A$	General And
$ $	$(@A)\text{Has_Or} \dashrightarrow @A^2 \Rightarrow @A$	General Or
$; >$	$(@E)\text{Has_Use} \dashrightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$	"Use" Environment Action
$;$	$(@E)\text{Has_Then} \dashrightarrow @E(T1) \times @E(T2) \Rightarrow @E(T2)$	"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 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

3.3 Function Expressions

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

$\langle func\text{-}expr \rangle ::= \langle line\text{-}func\text{-}expr \rangle \mid \langle big\text{-}func\text{-}expr \rangle \mid \langle cases\text{-}func\text{-}expr \rangle$

3.3.1 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 (5th 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 it 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" [6.1.2](#).

- *Grammar*

$\langle line\text{-}func\text{-}expr \rangle ::= \langle parameters \rangle [\text{'_'}] \text{'=>'} \langle line\text{-}func\text{-}body \rangle$

$\langle big\text{-}func\text{-}expr \rangle ::= \langle parameters \rangle [\text{'_'}] \text{'=>'} \langle big\text{-}func\text{-}body \rangle$

$\langle parameters \rangle ::= \langle identifier \rangle \mid \text{'*'} \mid \text{'('} [\text{'_'}] \langle parameters \rangle (\langle comma \rangle \langle parameters \rangle) + [\text{'_'}] \text{'}'$

$\langle line\text{-}func\text{-}body \rangle ::= [\text{'_'}] (\langle basic\text{-}or\text{-}app\text{-}expr \rangle \mid \langle line\text{-}op\text{-}expr \rangle \mid \text{'('} [\text{'_'}] \langle line\text{-}func\text{-}expr \rangle [\text{'_'}] \text{'})$

$\langle big\text{-}func\text{-}body \rangle ::= \langle nl \rangle \langle indent \rangle (\langle basic\text{-}or\text{-}app\text{-}expr \rangle \mid \langle op\text{-}expr \rangle \mid \text{'('} [\text{'_'}] \langle line\text{-}func\text{-}expr \rangle [\text{'_'}] \text{'})$

3.3.2 "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)

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 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*

$\langle \text{cases-func-expr} \rangle ::= \langle \text{cases-params} \rangle \langle \text{case} \rangle + [\langle \text{end-case} \rangle]$

$\langle \text{cases-params} \rangle ::=$
 $\langle \text{identifier} \rangle \mid \text{'cases'} \mid \text{'*'} \mid \text{'('} [\text{'_'}] \langle \text{cases-params} \rangle (\langle \text{comma} \rangle \langle \text{cases-params} \rangle) + [\text{'_'}] \text{'}'$

$\langle \text{case} \rangle ::= \langle \text{nl} \rangle \langle \text{indent} \rangle \langle \text{outer-matching} \rangle [\text{'_'}] \text{'=>' } \langle \text{case-body} \rangle$

$\langle \text{end-case} \rangle ::= \langle \text{nl} \rangle \langle \text{indent} \rangle (\text{'...'} \mid \langle \text{identifier} \rangle) [\text{'_'}] \text{'=>' } \langle \text{case-body} \rangle$

$\langle \text{outer-matching} \rangle ::= \langle \text{simple-id} \rangle \mid \langle \text{matching} \rangle$

$\langle \text{matching} \rangle ::= \langle \text{literal} \rangle \mid \langle \text{pre-func} \rangle \langle \text{inner-matching} \rangle \mid \langle \text{tuple-matching} \rangle \mid \langle \text{list-matching} \rangle$

$\langle \text{inner-matching} \rangle ::= \text{'*'} \mid \langle \text{identifier} \rangle \mid \langle \text{matching} \rangle$

$\langle \text{tuple-matching} \rangle ::= \text{'('} [\text{'_'}] \langle \text{inner-matching} \rangle (\langle \text{comma} \rangle \langle \text{inner-matching} \rangle) + [\text{'_'}] \text{'}'$

$\langle \text{list-matching} \rangle ::=$
 $\text{'['} [\text{'_'}] [\langle \text{inner-matching} \rangle (\langle \text{comma} \rangle \langle \text{inner-matching} \rangle)^* [\langle \text{rest-list-matching} \rangle]] [\text{'_'}] \text{'}'$

$\langle \text{rest-list-matching} \rangle ::= \langle \text{comma} \rangle [\langle \text{simple-id} \rangle \langle \text{equals} \rangle] \text{'...}'$

$\langle \text{case-body} \rangle ::= \langle \text{line-func-body} \rangle \mid \langle \text{big-func-body} \rangle [\langle \text{where-expr} \rangle]$

3.4 Value Definitions and "where" Expressions

3.4.1 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⟩ )*
```

3.4.2 "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" (3.4.1) or "'cases' Function Expressions" (3.3.2) 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⟩
```

4 Language Description: Types and Type Logic

4.1 Types

The constructs regarding types are **type expressions**, **type definitions** and **type nicknames** and they are described in the following sections.

4.1.1 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:

$$\langle type \rangle ::= [\langle condition \rangle] \langle simple-type \rangle$$
$$\langle simple-type \rangle ::= \langle param-t-var \rangle \mid \langle type-app-id-or-ahv \rangle \mid \langle power-type \rangle \mid \langle prod-type \rangle \mid \langle func-type \rangle$$

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*

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

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*

$\langle \text{param-t-var} \rangle ::= \text{'T'} [0-9]$

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 4.2). 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*

$\langle \text{ad-hoc-t-var} \rangle ::= \text{'@'} [A-Z]$

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*

$$\begin{aligned}\langle \text{prod-type} \rangle &::= \langle \text{field-type} \rangle (\text{'_x_'} \langle \text{field-type} \rangle)^+ \\ \langle \text{field-type} \rangle &::= \langle \text{power-base-type} \rangle \mid \langle \text{power-type} \rangle \\ \langle \text{power-base-type} \rangle &::= \langle \text{param-t-var} \rangle \mid \langle \text{type-app-id-or-ahv} \rangle \mid \text{'('} [\text{'_'}] (\langle \text{prod-type} \rangle \mid \langle \text{func-type} \rangle) [\text{'_'}] \text{'('} \\ \langle \text{power-type} \rangle &::= \langle \text{power-base-type} \rangle \text{'^'} \langle \text{int-greater-than-one} \rangle\end{aligned}$$

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*

$$\begin{aligned}\langle \text{func-type} \rangle &::= \langle \text{in-or-out-type} \rangle \text{'_=>_'} \langle \text{in-or-out-type} \rangle \\ \langle \text{in-or-out-type} \rangle &::= \\ &\quad \langle \text{param-t-var} \rangle \mid \langle \text{type-app-id-or-ahv} \rangle \mid \langle \text{power-type} \rangle \mid \langle \text{prod-type} \rangle \mid \text{'('} [\text{'_'}] \langle \text{func-type} \rangle [\text{'_'}] \text{'('}\end{aligned}$$

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 (seperately) 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 seperated 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 (4.2).

- *Grammar*

$\langle condition \rangle ::= \langle prop-name \rangle ' _ --> _ '$

4.1.2 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:

$\langle type-def \rangle ::= \langle tuple-type-def \rangle \mid \langle or-type-def \rangle$

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 convinience. 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)OnError(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)OnError(String)
= cases
  no_value => throw_err("There is no value!")
  the_value:val => result:val

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

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>
```

4.2 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.

4.2.1 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 truthvalue 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

$\langle type-prop-def \rangle ::= \langle atom-prop-def \rangle \mid \langle renaming-prop-def \rangle$

$\langle atom-prop-def \rangle ::= \langle prop-name-line \rangle \langle nl \rangle \text{'needed'} (\text{'_'} \mid \langle nl \rangle \text{'_'}) \langle identifier \rangle [\text{'_'}] \text{'.'} [\text{'_'}] \langle simple-type \rangle$

$\langle renaming-prop-def \rangle ::=$
 $\langle prop-name-line \rangle \langle nl \rangle \text{'equivalent'} (\text{'_'} \mid \langle nl \rangle \text{'_'}) \langle prop-name \rangle (\langle comma \rangle \langle prop-name \rangle)^*$

$\langle prop-name-line \rangle ::= \text{'type_proposition_'} \langle prop-name \rangle$

$\langle prop-name \rangle ::=$
 $[A-Z] (\langle name-part \rangle \langle types-in-paren \rangle) + [\langle name-part \rangle]$
 $\mid (\langle types-in-paren \rangle \langle name-part \rangle) + [\langle types-in-paren \rangle]$

$\langle name-part \rangle ::= ([A-Za-z] \mid \text{'_'} [A-Z]) +$

4.2.2 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(_))sHas_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(_))sHas_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

$\langle type-theo \rangle ::= \text{'type_theorem_'} \langle prop-name-with-sub \rangle [\text{'_-->_'} \langle prop-name-with-sub \rangle] \langle nl \rangle \text{'proof'} \langle proof \rangle$

$\langle prop-name-with-sub \rangle ::=$
 $[A-Z] (\langle name-part \rangle \langle subs-in-paren \rangle) + [\langle name-part \rangle]$
 $| (\langle subs-in-paren \rangle \langle name-part \rangle) + [\langle subs-in-paren \rangle]$

$\langle subs-in-paren \rangle ::= \text{'('} [\text{'_'}] \langle t-var-sub \rangle (\langle comma \rangle \langle t-var-sub \rangle)^* [\text{'_'}] \text{'}'$

$\langle t-var-sub \rangle ::= \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \langle power-type-sub \rangle | \langle prod-type-sub \rangle | \langle func-type-sub \rangle$

$\langle type-app-id-or-ahtv-sub \rangle ::= [\langle subs-or-unders-in-paren \rangle] \langle taioas-middle \rangle [\langle subs-or-unders-in-paren \rangle]$

$\langle taioas-middle \rangle ::= \langle type-id \rangle (\langle subs-or-unders-in-paren \rangle [A-Za-z]^+)^* | \langle ad-hoc-t-var \rangle$

$\langle subs-or-unders-in-paren \rangle ::= \text{'('} [\text{'_'}] \langle sub-or-under \rangle (\langle comma \rangle \langle sub-or-under \rangle)^* [\text{'_'}] \text{'}'$

$\langle sub-or-under \rangle ::= \langle t-var-sub \rangle | \text{'_'}$

$\langle power-type-sub \rangle ::= \langle power-base-type-sub \rangle \text{'^'} \langle int-greater-than-one \rangle$

$\langle power-base-type-sub \rangle ::=$
 $\text{'_'} | \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \text{'('} [\text{'_'}] (\langle prod-type-sub \rangle | \langle func-type-sub \rangle) [\text{'_'}] \text{'}'$

$\langle prod-type-sub \rangle ::= \langle field-type-sub \rangle (\text{'_x_'} \langle field-type-sub \rangle) +$

$\langle field-type-sub \rangle ::= \langle power-base-type-sub \rangle | \langle power-type-sub \rangle$

$\langle func-type-sub \rangle ::= \langle in-or-out-type-sub \rangle \text{'_=>_'} \langle in-or-out-type-sub \rangle$

$\langle in-or-out-type-sub \rangle ::=$
 $\text{'_'} | \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \langle power-type-sub \rangle | \langle prod-type-sub \rangle |$
 $\text{'('} [\text{'_'}] \langle func-type-sub \rangle [\text{'_'}] \text{'}'$

$\langle proof \rangle ::= \text{'_'} \langle id-or-op-eq \rangle \text{'_'} \langle line-expr \rangle | \langle nl \rangle \text{'_'} \langle id-or-op-eq \rangle \langle tt-value-expr \rangle$

$\langle id-or-op-eq \rangle ::= \langle identifier \rangle [\langle op \rangle \langle identifier \rangle] \text{'_='}$

$\langle tt-value-expr \rangle ::= \text{'_'} \langle line-expr \rangle | \langle nl \rangle \langle indent \rangle \langle value-expr \rangle [\langle where-expr \rangle]$

5 Language Description: Predefined

5.1 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(_)`,

5.2 Types

- Basic: `Int`, `Real`, `Char`, `String`, `(_)FromIO`, `(_)FState(_)``Man`
- Or Types: `EmptyVal`, `Bool`, `Possibly(_)`, `ListOf(_)``s`, `Result(_)``OnError(_)`
- Type Nicknames: `IO`, `State(_)``Man`, `Z`, `R`

5.3 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

6 Parser Implementation

6.1 Full grammar and indentation system

6.1.1 Full grammar

$\langle literal \rangle ::= \langle int-lit \rangle \mid \langle real-lit \rangle \mid \langle char-lit \rangle \mid \langle string-lit \rangle$

$\langle identifier \rangle ::= [\langle unders-in-paren \rangle] \langle id-start \rangle \langle id-cont \rangle^* [[0-9]] [\langle unders-in-paren \rangle]$

$\langle simple-id \rangle ::= \langle id-start \rangle [[0-9]]$

$\langle id-start \rangle ::= [a-z] [a-z_]^*$

$\langle id-cont \rangle ::= \langle unders-in-paren \rangle [a-z_]^+$

$\langle unders-in-paren \rangle ::= '(_' (\langle comma \rangle ' _ ')^* ')'$

$\langle comma \rangle ::= ',' [' _ ']$

$\langle paren-expr \rangle ::= '(' [' _ '] \langle line-op-expr \rangle \mid \langle line-func-expr \rangle [' _ '] ')'$

$\langle tuple \rangle ::= '(' [' _ '] \langle line-expr-or-under \rangle \langle comma \rangle \langle line-expr-or-unders \rangle [' _ '] ')'$

$\langle line-expr-or-unders \rangle ::= \langle line-expr-or-under \rangle (\langle comma \rangle \langle line-expr-or-under \rangle)^*$

$\langle line-expr-or-under \rangle ::= \langle line-expr \rangle \mid ' _ '$

$\langle line-expr \rangle ::= \langle basic-or-app-expr \rangle \mid \langle line-op-expr \rangle \mid \langle line-func-expr \rangle$

$\langle basic-or-app-expr \rangle ::= \langle basic-expr \rangle \mid \langle pre-func-app \rangle \mid \langle post-func-app \rangle$

$\langle basic-expr \rangle ::= \langle literal \rangle \mid \langle paren-func-app-or-id \rangle \mid \langle special-id \rangle \mid \langle tuple \rangle \mid \langle list \rangle$

$\langle big-tuple \rangle ::=$
 $'(' [' _ '] \langle line-expr-or-under \rangle [\langle nl \rangle \langle indent \rangle] \langle comma \rangle \langle line-expr-or-unders \rangle$
 $(\langle nl \rangle \langle indent \rangle \langle comma \rangle \langle line-expr-or-unders \rangle)^* \langle nl \rangle \langle indent \rangle ')'$

$\langle list \rangle ::= '[' [' _ '] [\langle line-expr-or-unders \rangle] [' _ '] ']'$

$\langle big-list \rangle ::= '[' [' _ '] \langle line-expr-or-unders \rangle (\langle nl \rangle \langle indent \rangle \langle comma \rangle \langle line-expr-or-unders \rangle)^* \langle nl \rangle \langle indent \rangle ']'$

$\langle paren-func-app-or-id \rangle ::= [\langle arguments \rangle] \langle id-start \rangle (\langle arguments \rangle [a-z_]^+)^* [[0-9]] [\langle arguments \rangle]$

$\langle arguments \rangle ::= '(' [' _ '] \langle line-expr-or-unders \rangle [' _ '] ')'$

$$\langle pre\text{-}func \rangle ::= \langle simple\text{-}id \rangle \text{ ‘:’}$$
$$\langle pre\text{-}func\text{-}app \rangle ::= \langle pre\text{-}func \rangle \langle operand \rangle$$
$$\langle post\text{-}func \rangle ::= \text{'.'} \ (\ \langle simple\text{-}id \rangle \ | \ \langle special\text{-}id \rangle \)$$
$$\langle special-id \rangle ::= '1st' \mid '2nd' \mid '3rd' \mid '4th' \mid '5th'$$
$$\langle post\text{-}func\text{-}app \rangle ::= (\langle basic\text{-}expr \rangle \mid \langle paren\text{-}expr \rangle \mid \text{'_'}) (\langle dot\text{-}change \rangle \mid \langle post\text{-}func \rangle + [\langle dot\text{-}change \rangle])$$
$$\langle \textit{dot-change} \rangle ::= \textit{'change'} [\textit{'_'}] \langle \textit{field-change} \rangle (\langle \textit{comma} \rangle \langle \textit{field-change} \rangle)^* [\textit{'_'}] \textit{'}'}$$
$$\langle field-change \rangle ::= (\langle simple-id \rangle \mid \langle special-id \rangle) \langle equals \rangle \langle line-expr-or-under \rangle$$
$$\langle equals \rangle ::= [\text{'_'}] \text{'='} [\text{'_'}]$$
$$\langle op\text{-}expr \rangle ::= \langle line\text{-}op\text{-}expr \rangle \mid \langle big\text{-}op\text{-}expr \rangle$$
$$\langle op\text{-}expr\text{-}start \rangle ::= (\langle operand \rangle \langle op \rangle) +$$
$$\langle line-op-expr \rangle ::= \langle op-expr-start \rangle (\langle operand \rangle \mid \langle line-func-expr \rangle)$$
$$\langle big-op-expr \rangle ::= \langle big-op-expr-op-split \rangle \mid \langle big-op-expr-func-split \rangle$$
$$\langle big-op-expr-op-split \rangle ::= \langle op-split-line \rangle + [\langle op-expr-start \rangle] (\langle operand \rangle \mid \langle func-expr \rangle)$$
$$\langle op-split-line \rangle ::= (\langle op-expr-start \rangle (\langle nl \rangle \mid \langle oper-fco \rangle) \mid \langle oper-fco \rangle) \langle indent \rangle$$
$$\langle oper-fco \rangle ::= \langle operand \rangle \text{ '}' \langle func-comp-op \rangle \text{ '\n'}$$
$$\langle \textit{big-op-expr-func-split} \rangle ::= \langle \textit{op-expr-start} \rangle (\langle \textit{big-func-expr} \rangle \mid \langle \textit{cases-func-expr} \rangle)$$
$$\langle operand \rangle ::= \langle basic-or-app-expr \rangle \mid \langle paren-expr \rangle \mid \text{'_'} \mid$$
$$\langle op \rangle ::= \text{'}\sqcup\text{' } \langle func-comp-op \rangle \text{'}\sqcup\text{' } \mid [\text{'}\sqcup\text{' }] \langle optional-spaces-op \rangle [\text{'}\sqcup\text{' }]$$
$$\langle func-comp-op \rangle ::= 'o>' \mid '<o'$$
$$\langle optional-spaces-op \rangle ::= \text{'->' | '<- ' | '^' | '*' | '/' | '+' | '-' | '==' | '!=' | '>' | '<' | '>=' | '<=' | '&' | '|' | ';' | '>' | ';'}$$

$\langle \text{func-expr} \rangle ::= \langle \text{line-func-expr} \rangle \mid \langle \text{big-func-expr} \rangle \mid \langle \text{cases-func-expr} \rangle$

$\langle \text{line-func-expr} \rangle ::= \langle \text{parameters} \rangle [\text{'_'}] \text{'=>'} \langle \text{line-func-body} \rangle$

$\langle \text{big-func-expr} \rangle ::= \langle \text{parameters} \rangle [\text{'_'}] \text{'=>'} \langle \text{big-func-body} \rangle$

$\langle \text{parameters} \rangle ::= \langle \text{identifier} \rangle \mid \text{'*'} \mid \text{'('} [\text{'_'}] \langle \text{parameters} \rangle (\langle \text{comma} \rangle \langle \text{parameters} \rangle) + [\text{'_'}] \text{'}'$

$\langle \text{line-func-body} \rangle ::= [\text{'_'}] (\langle \text{basic-or-app-expr} \rangle \mid \langle \text{line-op-expr} \rangle \mid \text{'('} [\text{'_'}] \langle \text{line-func-expr} \rangle [\text{'_'}] \text{'}')$

$\langle \text{big-func-body} \rangle ::= \langle \text{nl} \rangle \langle \text{indent} \rangle (\langle \text{basic-or-app-expr} \rangle \mid \langle \text{op-expr} \rangle \mid \text{'('} [\text{'_'}] \langle \text{line-func-expr} \rangle [\text{'_'}] \text{'}')$

$\langle \text{cases-func-expr} \rangle ::= \langle \text{cases-params} \rangle \langle \text{case} \rangle + [\langle \text{end-case} \rangle]$

$\langle \text{cases-params} \rangle ::= \langle \text{identifier} \rangle \mid \text{'cases'} \mid \text{'*'} \mid \text{'('} [\text{'_'}] \langle \text{cases-params} \rangle (\langle \text{comma} \rangle \langle \text{cases-params} \rangle) + [\text{'_'}] \text{'}'$

$\langle \text{case} \rangle ::= \langle \text{nl} \rangle \langle \text{indent} \rangle \langle \text{outer-matching} \rangle [\text{'_'}] \text{'=>'} \langle \text{case-body} \rangle$

$\langle \text{end-case} \rangle ::= \langle \text{nl} \rangle \langle \text{indent} \rangle (\text{'...'} \mid \langle \text{identifier} \rangle) [\text{'_'}] \text{'=>'} \langle \text{case-body} \rangle$

$\langle \text{outer-matching} \rangle ::= \langle \text{simple-id} \rangle \mid \langle \text{matching} \rangle$

$\langle \text{matching} \rangle ::= \langle \text{literal} \rangle \mid \langle \text{pre-func} \rangle \langle \text{inner-matching} \rangle \mid \langle \text{tuple-matching} \rangle \mid \langle \text{list-matching} \rangle$

$\langle \text{inner-matching} \rangle ::= \text{'*'} \mid \langle \text{identifier} \rangle \mid \langle \text{matching} \rangle$

$\langle \text{tuple-matching} \rangle ::= \text{'('} [\text{'_'}] \langle \text{inner-matching} \rangle (\langle \text{comma} \rangle \langle \text{inner-matching} \rangle) + [\text{'_'}] \text{'}'$

$\langle \text{list-matching} \rangle ::= \text{'['} [\text{'_'}] [\langle \text{inner-matching} \rangle (\langle \text{comma} \rangle \langle \text{inner-matching} \rangle)^* [\langle \text{rest-list-matching} \rangle]] [\text{'_'}] \text{'}'$

$\langle \text{rest-list-matching} \rangle ::= \langle \text{comma} \rangle [\langle \text{simple-id} \rangle \langle \text{equals} \rangle] \text{'...}'$

$\langle \text{case-body} \rangle ::= \langle \text{line-func-body} \rangle \mid \langle \text{big-func-body} \rangle [\langle \text{where-expr} \rangle]$

$\langle \text{value-def} \rangle ::=$
 $\quad \langle \text{indent} \rangle \langle \text{identifier} \rangle ([\text{'_'}] \text{'::'} [\text{'_'}] \mid \langle \text{nl} \rangle \langle \text{indent} \rangle \text{'::_'}) \langle \text{type} \rangle$
 $\quad \langle \text{nl} \rangle \langle \text{indent} \rangle \text{'=_'} \langle \text{value-expr} \rangle [\langle \text{where-expr} \rangle]$

$\langle \text{value-expr} \rangle ::= \langle \text{basic-or-app-expr} \rangle \mid \langle \text{op-expr} \rangle \mid \langle \text{func-expr} \rangle \mid \langle \text{big-tuple} \rangle \mid \langle \text{big-list} \rangle$

$\langle \text{grouped-value-defs} \rangle ::=$
 $\quad \langle \text{indent} \rangle \langle \text{identifier} \rangle (\langle \text{comma} \rangle \langle \text{identifier} \rangle) +$
 $\quad ([\text{'_'}] \text{'::'} [\text{'_'}] \mid \langle \text{nl} \rangle \langle \text{indent} \rangle \text{'::_'}) (\langle \text{type} \rangle (\langle \text{comma} \rangle \langle \text{type} \rangle) + \text{'all_'} \langle \text{type} \rangle)$
 $\quad \langle \text{nl} \rangle \langle \text{indent} \rangle \text{'=_'} \langle \text{line-exprs} \rangle (\langle \text{nl} \rangle \langle \text{indent} \rangle \langle \text{comma} \rangle \langle \text{line-exprs} \rangle)^*$

$\langle \text{line-exprs} \rangle ::= \langle \text{line-expr} \rangle (\langle \text{comma} \rangle \langle \text{line-expr} \rangle)^*$

$\langle where\text{-}expr \rangle ::= \langle nl \rangle \langle indent \rangle \text{'where'} \langle nl \rangle \langle value\text{-}def\text{-}or\text{-}defs \rangle (\langle nl \rangle \langle nl \rangle \langle value\text{-}def\text{-}or\text{-}defs \rangle)^*$

$\langle value\text{-}def\text{-}or\text{-}defs \rangle ::= \langle value\text{-}def \rangle \mid \langle grouped\text{-}value\text{-}defs \rangle$

$\langle type \rangle ::= [\langle condition \rangle] \langle simple\text{-}type \rangle$

$\langle simple\text{-}type \rangle ::= \langle param\text{-}t\text{-}var \rangle \mid \langle type\text{-}app\text{-}id\text{-}or\text{-}ahtv \rangle \mid \langle power\text{-}type \rangle \mid \langle prod\text{-}type \rangle \mid \langle func\text{-}type \rangle$

$\langle type\text{-}id \rangle ::= [A-Z] [A-Za-z]^*$

$\langle param\text{-}t\text{-}var \rangle ::= \text{'T'} [0-9]$

$\langle ad\text{-}hoc\text{-}t\text{-}var \rangle ::= \text{'@'} [A-Z]$

$\langle type\text{-}app\text{-}id\text{-}or\text{-}ahtv \rangle ::= [\langle types\text{-}in\text{-}paren \rangle] \langle taioa\text{-}middle \rangle [\langle types\text{-}in\text{-}paren \rangle]$

$\langle taioa\text{-}middle \rangle ::= \langle type\text{-}id \rangle (\langle types\text{-}in\text{-}paren \rangle [A-Za-z]^+)^* \mid \langle ad\text{-}hoc\text{-}t\text{-}var \rangle$

$\langle types\text{-}in\text{-}paren \rangle ::= \text{'('} [\text{'_'}] \langle simple\text{-}type \rangle (\langle comma \rangle \langle simple\text{-}type \rangle)^* [\text{'_'}] \text{'}'$

$\langle prod\text{-}type \rangle ::= \langle field\text{-}type \rangle (\text{'_x_'} \langle field\text{-}type \rangle)^+$

$\langle field\text{-}type \rangle ::= \langle power\text{-}base\text{-}type \rangle \mid \langle power\text{-}type \rangle$

$\langle power\text{-}base\text{-}type \rangle ::= \langle param\text{-}t\text{-}var \rangle \mid \langle type\text{-}app\text{-}id\text{-}or\text{-}ahtv \rangle \mid \text{'('} [\text{'_'}] (\langle prod\text{-}type \rangle \mid \langle func\text{-}type \rangle) [\text{'_'}] \text{'}'$

$\langle power\text{-}type \rangle ::= \langle power\text{-}base\text{-}type \rangle \text{'^'} \langle int\text{-}greater\text{-}than\text{-}one \rangle$

$\langle func\text{-}type \rangle ::= \langle in\text{-}or\text{-}out\text{-}type \rangle \text{'_=>_'} \langle in\text{-}or\text{-}out\text{-}type \rangle$

$\langle in\text{-}or\text{-}out\text{-}type \rangle ::= \langle param\text{-}t\text{-}var \rangle \mid \langle type\text{-}app\text{-}id\text{-}or\text{-}ahtv \rangle \mid \langle power\text{-}type \rangle \mid \langle prod\text{-}type \rangle \mid \text{'('} [\text{'_'}] \langle func\text{-}type \rangle [\text{'_'}] \text{'}'$

$\langle condition \rangle ::= \langle prop\text{-}name \rangle \text{'_-->_'} \langle prop\text{-}name \rangle$

$\langle type-def \rangle ::= \langle tuple-type-def \rangle \mid \langle or-type-def \rangle$

$\langle tuple-type-def \rangle ::=$
 $\quad \text{'tuple_type_'} \langle type-name \rangle \langle nl \rangle$
 $\quad \text{'value' } (\text{'_'} \mid \langle nl \rangle \text{'_'}) \langle id-tuple \rangle [\text{'_'}] \text{'::'} [\text{'_'}] (\langle prod-type \rangle \mid \langle power-type \rangle)$

$\langle type-name \rangle ::=$
 $\quad [\langle param-vars-in-paren \rangle] \langle type-id \rangle (\langle param-vars-in-paren \rangle [A-Za-z]^+)^* [\langle param-vars-in-paren \rangle]$

$\langle param-vars-in-paren \rangle ::= \text{'(' } [\text{'_'}] \langle param-t-var \rangle (\langle comma \rangle \langle param-t-var \rangle)^* [\text{'_'}] \text{'}'}$

$\langle id-tuple \rangle ::= \text{'(' } [\text{'_'}] \langle simple-id \rangle (\langle comma \rangle \langle simple-id \rangle)^+ [\text{'_'}] \text{'}'}$

$\langle or-type-def \rangle ::=$
 $\quad \text{'or_type_'} \langle type-name \rangle \langle nl \rangle$
 $\quad \text{'values' } (\text{'_'} \mid \langle nl \rangle \text{'_'})$
 $\quad \langle simple-id \rangle [\text{'::'} \langle simple-type \rangle] ([\text{'_'}] \text{'|'} [\text{'_'}] \langle simple-id \rangle [\text{'::'} \langle simple-type \rangle])^*$

$\langle t-nickname \rangle ::= \text{'type_nickname_'} \langle type-name \rangle \langle equals \rangle \langle simple-type \rangle$

$\langle type-prop-def \rangle ::= \langle atom-prop-def \rangle \mid \langle renaming-prop-def \rangle$

$\langle atom-prop-def \rangle ::= \langle prop-name-line \rangle \langle nl \rangle \text{'needed' } (\text{'_'} \mid \langle nl \rangle \text{'_'}) \langle identifier \rangle [\text{'_'}] \text{'::'} [\text{'_'}] \langle simple-type \rangle$

$\langle renaming-prop-def \rangle ::=$
 $\quad \langle prop-name-line \rangle \langle nl \rangle \text{'equivalent' } (\text{'_'} \mid \langle nl \rangle \text{'_'}) \langle prop-name \rangle (\langle comma \rangle \langle prop-name \rangle)^*$

$\langle prop-name-line \rangle ::= \text{'type_proposition_'} \langle prop-name \rangle$

$\langle prop-name \rangle ::=$
 $\quad [A-Z] (\langle name-part \rangle \langle types-in-paren \rangle)^+ [\langle name-part \rangle]$
 $\quad \mid (\langle types-in-paren \rangle \langle name-part \rangle)^+ [\langle types-in-paren \rangle]$

$\langle name-part \rangle ::= ([A-Za-z] \mid \text{'_'} [A-Z])^+$

$\langle type-theo \rangle ::= \text{'type_theorem_'} \langle prop-name-with-subs \rangle [\text{'_-->'} \langle prop-name-with-subs \rangle] \langle nl \rangle \text{'proof'} \langle proof \rangle$
 $\langle prop-name-with-subs \rangle ::=$
 $\quad [A-Z] (\langle name-part \rangle \langle subs-in-paren \rangle) + [\langle name-part \rangle]$
 $\quad | \quad (\langle subs-in-paren \rangle \langle name-part \rangle) + [\langle subs-in-paren \rangle]$
 $\langle subs-in-paren \rangle ::= \text{'('} [\text{'_'}] \langle t-var-sub \rangle (\langle comma \rangle \langle t-var-sub \rangle)^* [\text{'_'}] \text{'}'$
 $\langle t-var-sub \rangle ::= \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \langle power-type-sub \rangle | \langle prod-type-sub \rangle | \langle func-type-sub \rangle$
 $\langle type-app-id-or-ahtv-sub \rangle ::= [\langle subs-or-unders-in-paren \rangle] \langle taioas-middle \rangle [\langle subs-or-unders-in-paren \rangle]$
 $\langle taioas-middle \rangle ::= \langle type-id \rangle (\langle subs-or-unders-in-paren \rangle [A-Za-z]^+)^* | \langle ad-hoc-t-var \rangle$
 $\langle subs-or-unders-in-paren \rangle ::= \text{'('} [\text{'_'}] \langle sub-or-under \rangle (\langle comma \rangle \langle sub-or-under \rangle)^* [\text{'_'}] \text{'}'$
 $\langle sub-or-under \rangle ::= \langle t-var-sub \rangle | \text{'_'}$
 $\langle power-type-sub \rangle ::= \langle power-base-type-sub \rangle \text{'^'} \langle int-greater-than-one \rangle$
 $\langle power-base-type-sub \rangle ::=$
 $\quad \text{'_'} | \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \text{'('} [\text{'_'}] (\langle prod-type-sub \rangle | \langle func-type-sub \rangle) [\text{'_'}] \text{'}'$
 $\langle prod-type-sub \rangle ::= \langle field-type-sub \rangle (\text{'_x_'} \langle field-type-sub \rangle) +$
 $\langle field-type-sub \rangle ::= \langle power-base-type-sub \rangle | \langle power-type-sub \rangle$
 $\langle func-type-sub \rangle ::= \langle in-or-out-type-sub \rangle \text{'_=>'} \langle in-or-out-type-sub \rangle$
 $\langle in-or-out-type-sub \rangle ::=$
 $\quad \text{'_'} | \langle param-t-var \rangle | \langle type-app-id-or-ahtv-sub \rangle | \langle power-type-sub \rangle | \langle prod-type-sub \rangle |$
 $\quad \text{'('} [\text{'_'}] \langle func-type-sub \rangle [\text{'_'}] \text{'}'$
 $\langle proof \rangle ::= \text{'_'} \langle id-or-op-eq \rangle \text{'_'} \langle line-expr \rangle | \langle nl \rangle \text{'_'} \langle id-or-op-eq \rangle \langle tt-value-expr \rangle$
 $\langle id-or-op-eq \rangle ::= \langle identifier \rangle [\langle op \rangle \langle identifier \rangle] \text{'_='}$
 $\langle tt-value-expr \rangle ::= \text{'_'} \langle line-expr \rangle | \langle nl \rangle \langle indent \rangle \langle value-expr \rangle [\langle where-expr \rangle]$
 $\langle program \rangle ::= \langle nl \rangle^* \langle program-part \rangle (\langle nl \rangle \langle nl \rangle \langle program-part \rangle)^* \langle nl \rangle^*$
 $\langle program-part \rangle ::= \langle value-def \rangle | \langle grouped-value-defs \rangle | \langle type-def \rangle | \langle t-nickname \rangle | \langle type-prop-def \rangle | \langle type-theo \rangle$
 $\langle nl \rangle :: (\text{'_'} | \text{'\t'})^* \text{'\n'}$

6.1.2 Indentation system

The $\langle indent \rangle$ nonterminal is not a normal BNF nonterminal. 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 $\langle indent \rangle$ nonterminal 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: $il \leftarrow il - 1$.

6.2 High level structure

6.2.1 Parsec library

The parser was implemented using the **parsec** library [1]. 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 6.1.2).

6.2.2 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.

6.3 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.

6.3.1 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:

```
 $\langle literal \rangle ::= \langle int-lit \rangle \mid \langle real-lit \rangle \mid \langle char-lit \rangle \mid \langle string-lit \rangle$ 
```

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".

6.3.2 Example 1: List

The grammar rule for the list is the following:

$\langle list \rangle ::= '[' ['_ '] [\langle line\text{-}expr\text{-}or\text{-}unders \rangle] ['_ '] '['$

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`.

6.3.3 Example 2: Change

The grammar rule for the "change" expression is the following:

$\langle dot\text{-}change \rangle ::= '.change{' ['_ '] \langle field\text{-}change \rangle (\langle comma \rangle \langle field\text{-}change \rangle)^* ['_ '] '}'$

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).

6.3.4 Example 3: Value Definition

The grammar rule for the value definition is the following:

```
 $\langle value-def \rangle ::=$   
   $\langle indent \rangle \langle identifier \rangle ( [ \langle ' \_ ' \rangle \langle : ' \rangle [ \langle ' \_ ' \rangle ] \mid \langle nl \rangle \langle indent \rangle \langle : ' \_ ' \rangle ) \langle type \rangle$   
   $\langle nl \rangle \langle indent \rangle \langle = \_ \rangle \langle value-expr \rangle [ \langle where-expr \rangle ]$ 
```

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 $\langle 2 * \text{the indentation level} \rangle$ spaces, getting the indentation level from the state (see Indentation System 6.1.2). "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 (6.1.2). "has_type_symbol" parses the following part of the grammar rule: $([\langle ' _ ' \rangle \langle : ' \rangle [\langle ' _ ' \rangle] \mid \langle nl \rangle \langle indent \rangle \langle : ' _ ' \rangle)$. "we_are_in_equal_line" and "we_are_not_in_equal_line" change the state to enforce rule 6 of the Indentation System.

7 Translation to Haskell

7.1 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 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 then 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 to be converted as follows:
`x.change{f1 = f1 + 1} \implies x.change{f1 = x.f1 + 1}`

The same applies to special identifiers:

`x.change{1st = 1st + 1} \implies 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.

7.2 Translation Phase

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 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 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.

7.3 Translation Phase: Basic Expressions

7.3.1 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 ==> x1
apply(_)to_all_in(_) ==> apply'to_all_in'
(_)to_string ==> a'to_string
f(_, _, _) ==> 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.

7.3.2 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*

ftn function

Every tuple is passed through the **ftn** 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 7.7.2 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 `ftn` function.

7.3.3 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 7.3.2).

7.3.4 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}
 $\xrightarrow{\text{preprocessing}}$  state.change{counter = state.counter + 1}
 $\Rightarrow$  c0counter(counter(state) !+ (1 :: Integer)) state

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

tuple.change{x = _, y = _}  $\Rightarrow$  (\ (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.

7.4 Translation Phase: Operators

7.4.1 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 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
<code>-></code>	<code>&></code>
<code><-</code>	<code><&</code>
<code>o></code>	<code>.></code>
<code><o</code>	<code><.</code>
<code>^</code>	<code>!^</code>
<code>*</code>	<code>!*</code>
<code>/</code>	<code>!/</code>
<code>+</code>	<code>!+</code>
<code>-</code>	<code>!-</code>
<code>==</code>	<code>!==</code>
<code>!=</code>	<code>!!=</code>
<code>></code>	<code>!></code>
<code><</code>	<code>!<</code>
<code>>=</code>	<code>!>=</code>
<code><=</code>	<code>!<=</code>
<code>&</code>	<code>!&</code>
<code> </code>	<code>!!</code>
<code>; ></code>	<code>!>>=</code>
<code>;</code>	<code>!>></code>

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

```

7.4.2 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 then a new lambda abstraction is introduced, similarly to how this is done for tuples in [7.3.2](#).

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 [7.3.2](#).

7.5 Translation Phase: Function Expressions

7.5.1 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 + "\nCcoefficients: a = " + gac.a + ", b = " + gac.b
```

In that case, the `"where"` expression becomes a `"let-in"` expression as described in section 7.6. 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) !+ "\nCcoefficients: a = " !+ a(gac) !+ ", b = " !+ b(gac)  
in  
print'(message)
```

where the `"gac"` parameter is `"visible"` to the expression of `"message"`.

7.5.2 "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 paramters> of". For the last case, if we have "..." then it is translated to "_". When matching on the first few elements of a list the translations is done as follows:

```
[x1, x2, ...] => <case body> ⇒ x1 : x2 : _ -> <case body translation>
```

```
[x1, x2, xs = ...] => <case body> ⇒ x1 : x2 : xs -> <case body translation>
```

where the 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.

7.6 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

```

⇒

```

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 indentaion 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 " <translation of the parametric type variables seperated by spaces> '.'
```

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" expresssion, the "where" expresssion is translated to a "let-in" expresssion 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" expresssion is placed between the parameters and the body (last example).

7.7 Translation Phase: Types

7.7.1 Type Expressions

Type Identifiers

- *Examples*

`Int` \implies `Integer`

`Real` \implies `Double`

`String` \implies `String`

`SelfReferencingType` \implies `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` \implies `a1`

`T2` \implies `a2`

`T3` \implies `a3`

– *Description*

The '`T`' becomes an '`a`'.

- Ad Hoc Type Variables

– *Examples*

`@A` \implies `b0`

`@B` \implies `b1`

`@C` \implies `b2`

– *Description*

The '`@`' becomes a '`b`' and the capital letters map like so: `A` \implies 0, `B` \implies 1, etc

Type Application Types

- *Examples*

$\text{ListOf}(\text{Int})\text{s} \implies \text{ListOf's Integer}$

$\text{Error}(\text{String})\text{OrResult}(\text{Int}) \implies \text{Error'OrResult' String Integer}$

$(\text{Int})\text{Tree} \implies \text{A'Tree Integer}$

$\text{ListOf}(\text{Int} \Rightarrow \text{Int})\text{s} \implies \text{ListOf's (Integer} \rightarrow \text{Integer)}$

$\text{Before}(\text{B}, \text{C})\text{After} \implies \text{Before''After B C}$

$\text{A}(\text{B}(\text{C})) \implies \text{A' (B' C)}$

- *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*

$\text{Int} \times \text{Real} \times \text{String} \implies (\text{Integer}, \text{Double}, \text{String})$

$\text{Int}^2 \times \text{Int}^2 \implies ((\text{Integer}, \text{Integer}), (\text{Integer}, \text{Integer}))$

$(\text{A}^2 \Rightarrow \text{A}) \times \text{A} \times \text{ListOf}(\text{A})\text{s} \implies ((\text{A}, \text{A}) \rightarrow \text{A}, \text{A}, \text{ListOf's A})$

- *Description*

For the product types, all the field types are translated, separated by commas and put in parenthesis.

Function Types

- *Examples*

$\text{T1} \Rightarrow \text{T1} \implies \text{a1} \rightarrow \text{a1}$

$\text{Int}^2 \Rightarrow \text{Int} \implies (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$

$(\text{A}^2 \Rightarrow \text{A}) \times \text{A} \times \text{ListOf}(\text{A})\text{s} \Rightarrow \text{A} \implies ((\text{A}, \text{A}) \rightarrow \text{A}, \text{A}, \text{ListOf's A}) \rightarrow \text{A}$

- *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  $\implies$  A'Has_Str_Rep b19 => b19 -> String
```

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

```
(@T)Has_Internal_App --> (T1 => T2) x @T(T1) => @T(T2)  
 $\implies$   
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 "=>".

7.7.2 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
```

\Rightarrow

```
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" \Rightarrow "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` which is required for the translation of tuples for reasons explained in section 7.3.2. 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 FromTuplen "
 - (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 commas are appended.
6. The type annotations for the change function of each field (described in the ".change" section of 7.3.4) 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 substeps 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
```

\implies

```
data Bool =
  Ctrue |
  Cfalse
```

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

\implies

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

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

\implies

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

```
or_type Comparison
values lesser | equal | greater
```

\implies

```
data Comparison =
  Clesser |
  Cequal |
  Cgreater
```

- *Description*

For `or_types` the translation has the following steps:

1. "`tuple_type`" \implies "`data`"
2. The type name is translated the same way as it is in tuple types.
3. "`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 (seperated by a space).
 - (c) If it is not the last value, "`|`" is appended.

Type Nicknames

- *Examples*

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

```
type_nickname ErrOrRes(T1) = Error(String)OrResult(T1)
 $\implies$  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.

7.8 Translation Phase: Type Logic

7.8.1 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_A_Wrapper b19 where  
  wrap' :: a1 -> b19 a1
```

```
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 of the appearances of the ad hoc type variables).
- " 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 then the translation of its type

7.8.2 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(_))sHas_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
        no_value -> no_value
        Cthe_value x -> Cthe_value(f'(x))
```

```
type_theorem (ListOf(_))sHas_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 -> Cnon_empty_l(ft2(f <& head, apply'inside'(f, 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 →

```

# this is a representation of the AST after preprocessing not lcases syntax
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

```

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.

8 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`

9 Conclusion

A translator from lambda-cases to Haskell has been implemented. It is written in Haskell. The main differences from 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 show in the following example (and in the examples of the "cases" function expressions section [3.3.2](#))

```
- (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:

"Int²" 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.

References

- [1] Daan Leijen. *Parsec, a fast combinator parser*. 2001. URL: <https://web.archive.org/web/20120401040711/http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf>.