# Lambda Cases (lcases)

Dimitris Saridakis

## Contents

# 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))from_io
      ... => throw_err("You didn't give me 2 ints")

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

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

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

**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 |
|---|---|
| `cases` | 3.3.2 "cases" Function Expressions |
| `all where` | 3.4 Value Definitions and "where" Expressions |
| `tuple_type value or_type values type_nickname` | 4.1 Types |
| `type_proposition needed equivalent type_theorem proof` | 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*

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

#### Identifiers

- *Examples*

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

- *Description*

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

  An identifier starts with a lower case letter and is followed by lower case letters or underscores. It is also possible to have pairs of parentheses in the middle of an identifier (see "Parenthesis Function Application" section 3.1.3 for why this can be useful). Finally, an identifier can be ended with a digit.

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

- *Grammar*

  ⟨*identifier*⟩ ::= [ ⟨*unders_in_par*⟩ ] ⟨*id-start*⟩ ⟨*id-cont*⟩* [ [0-9] ] [ ⟨*unders_in_par*⟩ ]

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

  ⟨*id-start*⟩ ::= [a-z] [a-z_]*

  ⟨*id-cont*⟩ ::= ⟨*unders_in_par*⟩ [a-z_]+

  ⟨*unders_in_par*⟩ ::= '(_' (',' [ '␣' ] '_' )* ')'

  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.

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

- *Grammar*

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

#### Tuples

- *Examples*

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

- *Description*

  Tuples are used to group many values (of possibly different types) into one. The type of a tuple can be either the product of the types of the fields or a defined `tuple_type` which is equivalent to the afformentioned 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 seperate 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 demonstated 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(questions)

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

- *Grammar*

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

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

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

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

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

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

  ⟨*comma*⟩ ::= ',' [ '␣' ]

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

**Lists**

- *Examples*

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

- *Description*

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

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

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

```
ListOf(T)s
```

- *Big Lists*

  **Example**

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

  **Description**

  It is possible to stretch a (big) list expression over multiple lines (only) in a seperate 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*

  ⟨*list*⟩ ::= '[' [ '␣' ] [ ⟨*line-expr-or-unders*⟩ ] [ '␣' ] ']'

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

### 3.1.3 Parenthesis Function Application

- *Examples*

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

- *Description*

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

  In the first two examples, we have the usual mathematical function application 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 seperated 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, below is the definition of "`apply(_)to_all_in(_)`":

```
apply(_)to_all_in(_) : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = (f, cases)
    empty_l => empty_l
    non_empty_l:l => non_empty_l:(f <- l.head, apply(f)to_all_in(l.tail))
```

  The actual definition doesn't matter at this point, what matters is that 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 give a function only some of the arguments by putting an underscore to all the missing arguments. The resulting expression is a function that expects the missing arguments to return the final result. Let's see this in action:

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

  f(c, i, r) : String

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

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

- *Grammar*

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

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

#### 3.1.4 Prefix and Postfix Functions

**Prefix Functions**

- *Examples*

```
the_value:1
non_empty_l:l
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 this type inside. For example in the first example above we have:

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

  Where the function `thevalue:_` 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 function with an underscore in their argument. An illustration of the aforementioned is the last example, where the function `the_value:_` is an argument of the function `apply(_)to_all_in(_)`.

- *Grammar*

  ⟨*pre-func*⟩ ::= ⟨*simple-id*⟩ '`:`'

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

**Postfix Functions**

- *Examples*

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

- *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 function with an underscore in their argument. An illustration of the aforementioned is the last example, where the function "`_.1st`" is an argument of the function "`apply(_)to_all_in(_)`".

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

- *Grammar*

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

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

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

### The ".change" Function

- *Examples*

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

- *Description*

The "`_.change`" function is a special postfix function that works an all tuples. It returns a new tuple that is the same as the input tuple except for some fields that change. 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: "`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 can be 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
  : (@A)And(Int)AddTo(@A), (@A)Is(@B)s_1st --> ListOf(@B)s => ListOf(@B)s


assuming  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 seperated 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*⟩ ) [ '␣' ] '`=`' [ '␣' ] ⟨*line-expr-or-under*⟩

## 3.2 Operators

### 3.2.1 Function Application and Function Composition Operators

**Function Application Operators**

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

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

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

And a list of strings:

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

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

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

This can be done equivalently using the other operator:

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

**Function Composition Operators**

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

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

```
split(_)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 |
|---|---|
| ˆ | `(@A)To_The(@B)Is(@C) --> @A x @B => @C` |
| * | `(@A)And(@B)Multiply_To(@C) --> @A x @B => @C` |
| / | `(@A)Divided_By(@B)Is(@C) --> @A x @B => @C` |
| + | `(@A)And(@B)Add_To(@C) --> @A x @B => @C` |
| - | `(@A)Minus(@B)Is(@C) --> @A x @B => @C` |

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

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

Let's analyze further the example of addition. The type can be read as such: the '+' operator has the type `@A x @B => @C`, provided that the type proposition `(@A)And(@B)Add_To(@C)` holds. This proposition being true, means that addition has been defined for these three types (see section "Type Logic" 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 |
|---|---|
| `==` | `(@A)And(@B)Can_Be_Equal --> @A x @B => Bool` |
| `!=` | `(@A)And(@B)Can_Be_Unequal --> @A x @B => Bool` |
| `>` | `(@A)Can_Be_Greater_Than(@B) --> @A x @B => Bool` |
| `<` | `(@A)Can_Be_Less_Than(@B) --> @A x @B => Bool` |
| `>=` | `(@A)Can_Be_Gr_Or_Eq_To(@B) --> @A x @B => Bool` |
| `<=` | `(@A)Can_Be_Le_Or_Eq_To(@B) --> @A x @B => Bool` |
| `&` | `(@A)Has_And --> @A^2 => @A` |
| `\|` | `(@A)Has_Or --> @A^2 => @A` |

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

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

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

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

### 3.2.3 Environment Action Operators

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

**Simple Example Program**

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

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

```
print_string(_): String => (EmptyVal)FromIO
print_string("I'll repeat the line"): (EmptyVal)FromIO

get_line: (String)FromIO

# for the "then" operator we have:
; : (@E)Has_Then --> @E(T1) x @E(T2) => @E(T2)

# therefore in the following expression:
#     print_string("I'll repeat the line") ; get_line
# the only way to match the types is if @E = FromIO, T1 = EmptyVal, T2 = String
# from which it follows that (FromIO)Has_Then
# and the "then" operator (in this particular case) has type
#     (EmptyVal)FromIO x (String)FromIO => (String)FromIO
# for the whole expression we have:
print_string("I'll repeat the line") ; get_line
  : (String)FromIO

print_string: String => (EmptyVal)FromIO

# for the "use" operator we have:
;> : (@E)Has_Use --> @E(T1) x (T1 => @E(T2)) => @E(T2)

# therefore in the following expression:
#     print_string("I'll repeat the line") ; get_line ;> print_string(_)
# the only way to match the types is if @E = FromIO, T1 = String, T2 = EmptyVal
# from which it follows that (FromIO)Has_Use
# and the "use" operator (in this particular case) has type
#     (String)FromIO x (String => (EmptyVal)FromIO) => (EmptyVal)FromIO
# 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("Nice to meet you " + name + "!")

print_string(_): String => (EmptyVal)FromIO
"Hello! What’s your name?": String
<- : (T1 => T2) x T1 => T2

# matching the types: T1 = String, T2 = (EmptyVal)FromIO
print_string <- "Hello! What’s your name?"
  : (EmptyVal)FromIO

# Similarly to the previous example program
print_string <- "Hello! What’s your name?" ; get_line
  : (String)FromIO

name => print_string("Nice to meet you " + name + "!")
  : String => (EmptyVal)FromIO

# Here we have a function expression as the second operand
# of the ";>" operator:
#   print_string <- "Hello! What’s your name?" ; get_line ;> name =>
#   print_string("Nice to meet you " + name + "!")
# Whenever this happens the function extends to the end of the expression.
# So the expression:
#   name => print_string("Nice to meet you " + name + "!")
# is the second operand.
# This is important to note as it could be longer and have other subexpressions
# in it. For example it could be:
#   name => print_string("Nice to...) ; print_string("How old are you?") ;
#   get_line ;> age => print_string(...) ; ...
# Here again the whole expression from "name => ..." till the end would be the
# second operand to the ";>" operator (and as you can see it has more ";>"
# operators and operands of those operators inside)
#
# Putting all together (similarly to the previous example program):
print_string <- "Hello! What’s your name?" ; get_line ;> name =>
print_string("Nice to meet you " + 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 the does both actions (provided the first did not result in an error). The overall effect is a value which is an environment action of type `@E` (the combination of the "smaller" actions) which produces a value of type `T2` (the one produced by the second action) and therefore it is of type `@E(T2)`.

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

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

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

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

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

### 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"), but it is also possible with the following operators: "->", "o>", "<o".

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 the examples but they are compatible and good enough for their illustration purposes.

- *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 argument).

  – Every line after the first must be indented so that in begins at the column where the first line of the operator expression begun.

  – The precise indentation rules are described in the section ”Indentation System” 6.1.2.

- *Grammar*

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

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

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

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

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

  ⟨*op-split-line*⟩ ::= ⟨*op-expr-start*⟩ ( ⟨*nl*⟩ | ⟨*operand*⟩ ‘␣’ ⟨*func-comp-op*⟩ ‘`\n`’ ) ⟨*indent*⟩

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

  ⟨*operand*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*paren-expr*⟩ | ‘`_`’

  ⟨*op*⟩ ::= ‘␣’ ⟨*func-comp-op*⟩ ‘␣’ | [ ‘␣’ ] ⟨*optional-spaces-op*⟩ [ ‘␣’ ]

  ⟨*func-comp-op*⟩ ::= ‘`o>`’ | ‘`<o`’

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

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

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

⟨*func-expr*⟩ ::= ⟨*line-func-expr*⟩ | ⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩

### 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, seperated by commas. If a parameter is not needed it can be left empty by having an asterisk instead of an identifier (3rd and 4th 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 seperated by the funciton arrow ("`=>`"). The body is an operator or basic expression.

- *Big Function Expressions*

  **Example**

```
(value1, value2, value3, value4, value5, value6, value7) =>
print_line("value1 = " + value1 + ", value2 = " + value2 + ", value3 = " + value3) ;
print_line("value4 = " + value4 + ", value5 = " + value5 + ", value6 = " + value6) ;
print_line("value7 = " + value7)
```

  **Description**

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

  - The function expression must split in a new line after the function arrow ("`=>`").
  - Every line after the first must be indented so that in begins at the column where the first character of the parameters was in the first line.
  - The precise indentation rules are described in the section "Indentation System" 6.1.2.

- *Grammar*

  ⟨*line-func-expr*⟩ ::= ⟨*parameters*⟩ [ '␣' ] '`=>`' ⟨*line-func-body*⟩

  ⟨*big-func-expr*⟩ ::= ⟨*parameters*⟩ [ '␣' ] '`=>`' ⟨*big-func-body*⟩

  ⟨*parameters*⟩ ::= ⟨*identifier*⟩ | '`*`' | '`(`' [ '␣' ] ⟨*parameters*⟩ ( ⟨*comma*⟩ ⟨*parameters*⟩ )+ [ '␣' ] '`)`'

  ⟨*line-func-body*⟩ ::= [ '␣' ] ( ⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ )

  ⟨*big-func-body*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ( ⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ )

### 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

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

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

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

traffic_lights_match(_, _): TrafficLight^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))

(_)is_empty: ListOf(T1)s => Bool
  = cases
    empty_l => true
    non_empty_l:* => false

apply(_)to_all_in(_): (T1 => T2) x ListOf(T1)s => ListOf(T2)s
  = (f, cases)
    empty_l => empty_l
    non_empty_l:list => non_empty_l:(f <- list.head, apply(f)to_all_in(list.tail))
```

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

  It is also possible to use a "`where`" expression below a particular case. The "`where`" expression must be indented two spaces more than than the line where that particular case begins.

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

- *Grammar*

  ⟨*cases-func-expr*⟩ ::= ⟨*cases-params*⟩ ⟨*case*⟩+ [ ⟨*end-case*⟩ ]

  ⟨*cases-params*⟩ ::=
  ⟨*identifier*⟩ | '`cases`' | '`*`' | '`(`' [ '␣' ] ⟨*cases-params*⟩ ( ⟨*comma*⟩ ⟨*cases-params*⟩ )+ [ '␣' ] '`)`'

  ⟨*case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ⟨*outer-matching*⟩ [ '␣' ] '`=>`' ⟨*case-body*⟩

  ⟨*end-case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ('`...`' | ⟨*identifier*⟩) [ '␣' ] '`=>`' ⟨*case-body*⟩

  ⟨*outer-matching*⟩ ::= ⟨*simple-id*⟩ | ⟨*matching*⟩

  ⟨*matching*⟩ ::= ⟨*literal*⟩ | ⟨*pre-func*⟩ ⟨*inner-matching*⟩ | ⟨*tuple-matching*⟩ | ⟨*list-matching*⟩

  ⟨*inner-matching*⟩ ::= '`*`' | ⟨*identifier*⟩ | ⟨*matching*⟩

  ⟨*tuple-matching*⟩ ::= '`(`' [ '␣' ] ⟨*inner-matching*⟩ ( ⟨*comma*⟩ ⟨*inner-matching*⟩ )+ [ '␣' ] '`)`'

  ⟨*list-matching*⟩ ::= '`[`' [ '␣' ] [ ⟨*inner-matching*⟩ ( ⟨*comma*⟩ ⟨*inner-matching*⟩ )* ] [ '␣' ] '`]`'

  ⟨*case-body*⟩ ::= ⟨*line-func-body*⟩ | ⟨*big-func-body*⟩ [ ⟨*where-expr*⟩ ]

## 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 seperating 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
    empty_l => empty_l
    non_empty_l:l => sort(less_l) + l.head + sort(greater_l)
      where
      less_l, greater_l: all ListOf(Int)s
        = filter(l.tail)with(_ < l.head), filter(l.tail)with(_ >= l.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
        empty_l => 0
        non_empty_l:l => l.head + sum_list(l.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*

  $\langle$*where-expr*$\rangle$ ::= $\langle$*nl*$\rangle$ $\langle$*indent*$\rangle$ '`where`' $\langle$*nl*$\rangle$ $\langle$*value-def-or-defs*$\rangle$ ( $\langle$*nl*$\rangle$ $\langle$*nl*$\rangle$ $\langle$*value-def-or-defs*$\rangle$ )*

  $\langle$*value-def-or-defs*$\rangle$ ::= $\langle$*value-def*$\rangle$ | $\langle$*grouped-value-defs*$\rangle$

# 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:

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

$\langle simple\text{-}type \rangle$ ::= $\langle type\text{-}id\text{-}or\text{-}var \rangle$ | $\langle type\text{-}app \rangle$ | $\langle power\text{-}type \rangle$ | $\langle prod\text{-}type \rangle$ | $\langle func\text{-}type \rangle$

$\langle type\text{-}id\text{-}or\text{-}var \rangle$ ::= $\langle type\text{-}id \rangle$ | $\langle type\text{-}var \rangle$

**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*

  $\langle type\text{-}id \rangle$ ::= [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.

*Grammar*

$\langle type\text{-}var \rangle$ ::= $\langle param\text{-}t\text{-}var \rangle$ | $\langle ad\text{-}hoc\text{-}t\text{-}var \rangle$

### Parametric Type Variables

- *Examples*

```
T1     T2     T3
```

- *Examples of parametric type variables inside bigger type expressions*

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

- *Description*

  Parametric type variables can be substituted with any type and the program will type check. The simplest example of a polymorphic type with a parametric type variable is the type of the identity function where we have:

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

id(1): Int
  where T1 is substituted by Int and id gets the type Int => Int

id("Hello"): String
  where T1 is substituted by String and id gets the type String => String
```

  A parametric type variable is written with capital "T" followed by a digit.

- *Grammar*

  ⟨*param-t-var*⟩ ::= 'T' [0-9]

### Ad Hoc Type Variables

- *Examples*

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

- *Examples of ad hoc type variables inside bigger type expressions*

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

- *Description*

  Ad hoc type variables are like parametric type variables with the difference that any type by which they are substituted must satisfy certain conditions in order for the program to type check. These conditions come in the form of type propositions (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*

  ⟨*ad-hoc-t-var*⟩ ::= '@' [A-Z]

### Type Application Types

- *Examples*

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

- *Description*

  Type application types are types that are produced by passing arguments to a type function generated by a `tuple_type` or an `or_type` definition. For example, given the definition of `ListOf(T1)s`:

```
or_type ListOf(T1)s
values non_empty_l:NonEmptyListOf(T1)s | empty_l
```

  We have that `ListOf(_)s` is a type function that receives one type parameter and returns a resulting type. For example `ListOf(Int)s` is the result of passing the type argument `Int` to `ListOf(_)s`.

  Type application types have the same form as the name in the `tuple_type` or `or_type` definition, with the difference that type parameters are substituted by the expressions of the type arguments.

- *Grammar*

  ⟨*type-app*⟩ ::=
  [ ⟨*types-in-paren*⟩ ] ⟨*type-id-with-args*⟩ [ ⟨*types-in-paren*⟩ ]
  | ⟨*types-in-paren*⟩ ⟨*t-id-or-ad-hoc-t-var*⟩ [ ⟨*types-in-paren*⟩ ]
  | ⟨*t-id-or-ad-hoc-t-var*⟩ ⟨*types-in-paren*⟩

  ⟨*type-id-with-args*⟩ ::= ⟨*type-id*⟩ ( ⟨*types-in-paren*⟩ [A-Za-z]+ )+

  ⟨*t-id-or-ad-hoc-t-var*⟩ ::= ⟨*type-id*⟩ | ⟨*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 seperated by the string " `x` " (space 'x' space) because 'x' is very similar the symbol used in the cartesian product. If any of the fields is of a product or a function type then the corresponding type expression must be inside parentheses. A product type where all the fields are of the same type can be abbreviated with a power type expression which is comprised of the type, the power symbol '^' and the number of times the type is repeated.

- *Grammar*

  ⟨*prod-type*⟩ ::= ⟨*field-type*⟩ ( '␣x␣' ⟨*field-type*⟩ )+

  ⟨*field-type*⟩ ::= ⟨*power-base-type*⟩ | ⟨*power-type*⟩

  ⟨*power-base-type*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app*⟩ | '(' [ '␣' ] ( ⟨*prod-type*⟩ | ⟨*func-type*⟩ ) [ '␣' ] ')'

  ⟨*power-type*⟩ ::= ⟨*power-base-type*⟩ '^' ⟨*int-greater-than-one*⟩

### Function Types

- *Examples*

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

- *Description*

  A function type expression is comprised of the input type expression and the output type expression seperated by the function arrow ("`=>`"). The input and output type expressions are type expressions which are put in parentheses only if they are function type expressions.

- *Grammar*

  ⟨*func-type*⟩ ::= ⟨*in-or-out-type*⟩ '␣=>␣' ⟨*in-or-out-type*⟩

  ⟨*in-or-out-type*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | '(' [ '␣' ] ⟨*func-type*⟩ [ '␣' ] ')'

### Conditional Types

- *Examples*

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

- *Description*

Conditional types are the types of values that are polymorphic not because of their structure but because they have been defined (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*

⟨*condition*⟩ ::= ⟨*prop-name*⟩ '␣-->␣'

### 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:

⟨*type-def*⟩ ::= ⟨*tuple-type-def*⟩ | ⟨*or-type-def*⟩

#### Tuple Types

- *Definition Examples*

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

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

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

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

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

- *Usage Examples*

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

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

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

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

- *Description*

  A tuple type is equivalent to a product type with a new name and names for the fields for 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*

  $\langle$*tuple-type-def*$\rangle$ ::=
  ‘`tuple_type␣`’ $\langle$*type-name*$\rangle$ $\langle$*nl*$\rangle$
  ‘`value`’ ( ‘␣’ | $\langle$*nl*$\rangle$ ‘␣␣’ ) $\langle$*id-tuple*$\rangle$ [ ‘␣’ ] ‘:’ [ ‘␣’ ] ( $\langle$*prod-type*$\rangle$ | $\langle$*power-type*$\rangle$ )

  $\langle$*type-name*$\rangle$ ::=
  [ $\langle$*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*$\rangle$ ::= ‘(’ [ ‘␣’ ] $\langle$*param-t-var*$\rangle$ ( $\langle$*comma*$\rangle$ $\langle$*param-t-var*$\rangle$ )* [ ‘␣’ ] ‘)’

  $\langle$*id-tuple*$\rangle$ ::= ‘(’ [ ‘␣’ ] $\langle$*simple-id*$\rangle$ ( $\langle$*comma*$\rangle$ $\langle$*simple-id*$\rangle$ )+ [ ‘␣’ ] ‘)’

**Or Types**

- *Definition Examples*

```
or_type Bool
values true | false

or_type Possibly(T1)
values the_value:T1 | no_value

// needed tuple_type for ListOf(T1)s
tuple_type NonEmptyListOf(T1)s
value (head, tail) : T1 x ListOf(T1)s

or_type ListOf(T1)s
values non_empty_l:NonEmptyListOf(T1)s | empty_l

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

- *Usage Examples*

```
(_)is_empty: ListOf(T1)s => Bool
  = cases
    empty_l => true
    non_empty_l:* => false

(_)head: ListOf(T1)s => Possibly(T1)
  = cases
    empty_l => no_value
    non_empty_l:list => the_value:list.head

sum_list(_): ListOf(Int)s => Int
  = cases
    empty_l => 0
    non_empty_l:l => l.head + sum_list(l.tail)

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

- *Description*

  The values of an `or_type` are split into cases. Some cases have other values inside. The cases which have other values inside are followed by a colon and the type of the internal value. Similar syntax can be used for matching that particular case in a function using the "cases" syntax. An `or_type` definition automatically creates prefix functions for each case with an internal value (which are simply conversions from the type of the internal value to the `or_type`). For example, for the case "`non_empty_l`" of a list, the function "`non_empty_l:_`" is automatically created from the definition for which we can say:

  ```
  non_empty_l:_ : NonEmptyListOf(T1)s => ListOf(T1)s
  ```

  For example:

  ```
  non_e_l : NonEmptyListOf(Int)s
    = (1, [2, 3, 4])
  >> non_empty_l:non_e_l
    : ListOf(Int)s
    ==> [1, 2, 3, 4]
  ```

  Similarly:

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

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

  ```
  non_empty_ls(_)to_ls : ListOf(NonEmptyListOf(T1)s)s => ListOf(ListOf(T1)s)s
    = apply(non_empty_l:_)to_all_in(_)
  ```

- *Grammar*

  ⟨*or-type-def*⟩ ::=
  ‘`or_type␣`’ ⟨*type-name*⟩ ⟨*nl*⟩
  ‘`values`’ ( ‘␣’ | ⟨*nl*⟩ ‘␣␣’ )
  ⟨*simple-id*⟩ [ ‘`:`’ ⟨*simple-type*⟩ ] ( [ ‘␣’ ] | [ ‘␣’ ] ⟨*simple-id*⟩ [ ‘`:`’ ⟨*simple-type*⟩ ] )*

### 4.1.3 Type Nicknames

- *Examples*

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

- *Description*

  Type nicknames are used to abbreviate or give a more descriptive name to a type. They start with the keyword "`type_nickname`", followed by the nickname, then an equal sign and they end with the type to be nicknamed. Parametric type variables can be used in the nickname.

- *Grammar*

  ⟨*t-nickname*⟩ ::= ‘`type_nickname␣`’ ⟨*type-name*⟩ [ ‘␣’ ] ‘`=`’ [ ‘␣’ ] ⟨*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 definining 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
          empty_l => show_err("Tried to take the first element of an empty list")
          non_empty_l:l => l.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 seperated 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 (i.e. AND of all of them) into one 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 seperated by commas (where the commas essentially mean "and").

### Grammar for Proposition Definitions

⟨*type-prop-def*⟩ ::= ⟨*atom-prop-def*⟩ | ⟨*renaming-prop-def*⟩

⟨*atom-prop-def*⟩ ::= ⟨*prop-name-line*⟩ ⟨*nl*⟩ '`needed`' ( '␣' | ⟨*nl*⟩ '␣␣' ) ⟨*identifier*⟩ [ '␣' ] '`:`' [ '␣' ] ⟨*simple-type*⟩

⟨*renaming-prop-def*⟩ ::=
    ⟨*prop-name-line*⟩ ⟨*nl*⟩ '`equivalent`' ( '␣' | ⟨*nl*⟩ '␣␣' ) ⟨*prop-name*⟩ ( ⟨*comma*⟩ ⟨*prop-name*⟩ )*

⟨*prop-name-line*⟩ ::= '`type_proposition`␣' ⟨*prop-name*⟩

⟨*prop-name*⟩ ::=
    [A-Z] ( ⟨*name-part*⟩ ⟨*ad-hoc-vars-in-paren*⟩ )+ [ ⟨*name-part*⟩ ]
  | ( ⟨*ad-hoc-vars-in-paren*⟩ ⟨*name-part*⟩ )+ [ ⟨*ad-hoc-vars-in-paren*⟩ ]

⟨*ad-hoc-vars-in-paren*⟩ ::= '`(`' [ '␣' ] ⟨*ad-hoc-t-var*⟩ ( ⟨*comma*⟩ ⟨*ad-hoc-t-var*⟩ )* [ '␣' ] '`)`'

⟨*name-part*⟩ ::= ( [A-Za-z] | '`_`'[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(_)s)Has_A_Wrapper
proof wrap(_) = [_]

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

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

- *Usage*

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

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

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

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

- *Description*

  A theorem of an atomic proposition proves the proposition for specific type arguments, by implementing the value associated to the proposition for these type arguments. Therefore, the value associated with the proposition can be used with all the combinations of type arguments for which the proposition is true, i.e. the combinations of type arguments for which the value has been implemented.

  A proof of a theorem of an atomic proposition is correct when the implementation of the value associated with the proposition follows the form of the type given to the value by the definition of the proposition, i.e. the only difference between the type of the value in the theorem and the type of the value in the definition is that the type parameters of the proposition are substituted by the type arguments of the theorem.

  A theorem of an atomic proposition begins with the keyword "`type_theorem`" followed by the name of the proposition with the type parameters substituted by the specific types for which the proposition will be proven. The second line is the keyword "`proof`". The third line is indented once and it is the line in which the proof begins. The proof begins with the identifier of the value associated with the proposition and is followed by an equal sign and the value expression which implements the value.

### Implication Propositions

- *Examples*

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

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

type_propositon (@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 very similar to a theorem of an atomic proposition in the sense that it also implements a value in the proof. The difference is that the implementation uses another ad hoc polymorphic value (or many). Therefore, the implementation does not prove the proposition associated to the value it implements, because it assumes that the polymorphic value(s) used in the implementation is(are) already defined. In other words it proves the following: "if this(these) ad hoc polymorphic value(s) is(are) defined then we can also define this other one". This can be translated into the following implication proposition: "if the proposition associated to the value(s) we are using is 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.

  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 propostition is comprised by two atomic propositions seperated by the condition arrow (" `-->` ") arrow.

**Grammar for Theorems**

⟨*type-theo*⟩ ::= ‘`type_theorem␣`’ ⟨*prop-name-with-subs*⟩ [ ‘`␣-->␣`’ ⟨*prop-name-with-subs*⟩ ] ⟨*nl*⟩ ‘`proof`’ ⟨*proof*⟩

⟨*prop-name-with-subs*⟩ ::=
    [A-Z] ( ⟨*name-part*⟩ ⟨*subs-in-paren*⟩ )+ [ ⟨*name-part*⟩ ]
  | ( ⟨*subs-in-paren*⟩ ⟨*name-part*⟩ )+ [ ⟨*subs-in-paren*⟩ ]

⟨*subs-in-paren*⟩ ::= ‘`(`’ [ ‘`␣`’ ] ⟨*t-var-sub*⟩ ( ⟨*comma*⟩ ⟨*t-var-sub*⟩ )* [ ‘`␣`’ ] ‘`)`’

⟨*t-var-sub*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩

⟨*type-app-sub*⟩ ::=
    [ ⟨*subs-or-unders-in-paren*⟩ ] ⟨*type-id-with-subs*⟩ [ ⟨*subs-or-unders-in-paren*⟩ ]
  | ⟨*subs-or-unders-in-paren*⟩ ⟨*t-id-or-ad-hoc-t-var*⟩ [ ⟨*subs-or-unders-in-paren*⟩ ]
  | ⟨*t-id-or-ad-hoc-t-var*⟩ ⟨*subs-or-unders-in-paren*⟩

⟨*type-id-with-subs*⟩ ::= ⟨*type-id*⟩ ( ⟨*subs-or-unders-in-paren*⟩ [A-Za-z]+ )+

⟨*subs-or-unders-in-paren*⟩ ::= ‘`(`’ [ ‘`␣`’ ] ⟨*sub-or-under*⟩ ( ⟨*comma*⟩ ⟨*sub-or-under*⟩ )* [ ‘`␣`’ ] ‘`)`’

⟨*sub-or-under*⟩ ::= ⟨*t-var-sub*⟩ | ‘`_`’

⟨*power-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ ‘`^`’ ⟨*int-greater-than-one*⟩

⟨*power-base-type-sub*⟩ ::=
    ‘`_`’ | ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ‘`(`’ [ ‘`␣`’ ] ( ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩ ) [ ‘`␣`’ ] ‘`)`’

⟨*prod-type-sub*⟩ ::= ⟨*field-type-sub*⟩ ( ‘`␣x␣`’ ⟨*field-type-sub*⟩ )+

⟨*field-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ | ⟨*power-type-sub*⟩

⟨*func-type-sub*⟩ ::= ⟨*in-or-out-type-sub*⟩ ‘`␣=>␣`’ ⟨*in-or-out-type-sub*⟩

⟨*in-or-out-type-sub*⟩ ::=
    ‘`_`’ | ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ‘`(`’ [ ‘`␣`’ ] ⟨*func-type-sub*⟩ [ ‘`␣`’ ] ‘`)`’

⟨*proof*⟩ ::= ‘`␣`’ ⟨*id-or-op-eq*⟩ ‘`␣`’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ‘`␣␣`’ ⟨*id-or-op-eq*⟩ ⟨*tt-value-expr*⟩

⟨*id-or-op-eq*⟩ ::= ⟨*identifier*⟩ [ ⟨*op*⟩ ⟨*identifier*⟩ ] ‘`␣=`’

⟨*tt-value-expr*⟩ ::= ‘`␣`’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ⟨*indent*⟩ ⟨*value-expr*⟩ [ ⟨*where-expr*⟩ ]

# 5 Language Description: Predefined

## 5.1 Values

- Constants: `undefined, pi`
- 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(_), leave(_)from(_), zip(_)and(_), unzip(_), zip(_)and(_)with(_)`
  - IO:
    - Input: `get_char, get_line, get_input, read_file(_)`
    - Output: `print(_), print_string(_), print_line(_), write(_)in_file(_)`
  - Ad Hoc Polymorphic:
    `(_)first, (_)second, (_)third, (_)fourth, (_)fifth, wrap(_),`
    `(_)to_string, from_string(_), apply(_)inside(_),`
    `apply_wrapd(_)inside(_), (_)to_list, from_list(_)`

## 5.2 Types

- Basic: `Int, Real, Char, String`
- Or Types: `EmptyVal, Bool, Possibly(_), ListOf(_)s, Result(_)OrError(_)`
- Tuple Types: `NonEmptyListOf(_)s`
- Type Nicknames: `IO, Z, R, C`

## 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)`
  - `(@T)Has_Use`
  - `(@T)Has_Then`
- Function Propositions:
  - `(@A)Is(@B)s_First`
  - `(@A)Is(@B)s_Second`
  - `(@A)Is(@B)s_Third`
  - `(@A)Is(@B)s_Fourth`
  - `(@A)Is(@B)s_Fifth`
  - `(@T)Has_A_Wrapper`
  - `(@T)Has_Str_Rep`
  - `(@T)Can_Be_Parsed`
  - `(@T)Has_Internal_App`
  - `(@T)Has_Wrapd_Intern_App`
- Renaming Propositions:
  - `(@T)Has_Exponentiation`
  - `(@T)Has_Multiplication`
  - `(@T)Has_Division`
  - `(@T)Has_Addition`
  - `(@T)Has_Subtraction`
  - `(@T)Has_Equality`
  - `(@T)Has_Non_Equality`
  - `(@T)Has_Greater_Or_Equal`
  - `(@T)Has_Less_Or_Equal`
  - `(@T)Has_Greater`
  - `(@T)Has_Less`

- Theorems:
  - (T1)Is(T1 x T2)s_First
  - (T1)Is(T1 x T2 x T3)s_First
  - (T1)Is(T1 x T2 x T3 x T4)s_First
  - (T1)Is(T1 x T2 x T3 x T4 x T5)s_First
  - (T1)Is(ListOf(T1)s)s_First
  - (T2)Is(T1 x T2)s_Second
  - (T2)Is(T1 x T2 x T3)s_Second
  - (T2)Is(T1 x T2 x T3 x T4)s_Second
  - (T2)Is(T1 x T2 x T3 x T4 x T5)s_Second
  - (T1)Is(ListOf(T1)s)s_Second
  - (T3)Is(T1 x T2 x T3)s_Third
  - (T3)Is(T1 x T2 x T3 x T4)s_Third
  - (T3)Is(T1 x T2 x T3 x T4 x T5)s_Third
  - (T1)Is(ListOf(T1)s)s_Third
  - (T4)Is(T1 x T2 x T3 x T4)s_Fourth
  - (T4)Is(T1 x T2 x T3 x T4 x T5)s_Fourth
  - (T1)Is(ListOf(T1)s)s_Fourth
  - (T5)Is(T1 x T2 x T3 x T4 x T5)s_Fifth
  - (T1)Is(ListOf(T1)s)s_Fifth
  - TODO wrapper
  - (Int)Has_Str_Rep
  - (Char)Has_Str_Rep
  - (Real)Has_Str_Rep
  - (@A)Has_Str_Rep --> (ListOf(@A)s)Has_Str_Rep
  - TODO apply()inside
  - TODO wrapd_app()inside
  - TODO ;>
  - TODO ;

# 6 Parser Implementation

## 6.1 Full grammar and indentation system

### 6.1.1 Full grammar

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

⟨*identifier*⟩ ::= [ ⟨*unders_in_par*⟩ ] ⟨*id-start*⟩ ⟨*id-cont*⟩* [ [0-9] ] [ ⟨*unders_in_par*⟩ ]

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

⟨*id-start*⟩ ::= [a-z] [a-z_]*

⟨*id-cont*⟩ ::= ⟨*unders_in_par*⟩ [a-z_]+

⟨*unders_in_par*⟩ ::= '(_' (',' [ '␣' ] '_' )* ')'

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

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

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

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

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

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

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

⟨*comma*⟩ ::= ',' [ '␣' ]

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

⟨*list*⟩ ::= '[' [ '␣' ] [ ⟨*line-expr-or-unders*⟩ ] [ '␣' ] ']'

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

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

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

⟨*pre-func*⟩ ::= ⟨*simple-id*⟩ ‘`:`’

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

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

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

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

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

⟨*field-change*⟩ ::= ( ⟨*simple-id*⟩ | ⟨*special-id*⟩ ) [ ‘␣’ ] ‘`=`’ [ ‘␣’ ] ⟨*line-expr-or-under*⟩

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

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

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

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

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

⟨*op-split-line*⟩ ::= ⟨*op-expr-start*⟩ ( ⟨*nl*⟩ | ⟨*operand*⟩ ‘␣’ ⟨*func-comp-op*⟩ ‘`\n`’ ) ⟨*indent*⟩

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

⟨*operand*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*paren-expr*⟩ | ‘`_`’

⟨*op*⟩ ::= ‘␣’ ⟨*func-comp-op*⟩ ‘␣’ | [ ‘␣’ ] ⟨*optional-spaces-op*⟩ [ ‘␣’ ]

⟨*func-comp-op*⟩ ::= ‘`o>`’ | ‘`<o`’

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

⟨*func-expr*⟩ ::= ⟨*line-func-expr*⟩ | ⟨*big-func-expr*⟩ | ⟨*cases-func-expr*⟩

⟨*line-func-expr*⟩ ::= ⟨*parameters*⟩ [ ‘␣’ ] ‘`=>`’ ⟨*line-func-body*⟩

⟨*big-func-expr*⟩ ::= ⟨*parameters*⟩ [ ‘␣’ ] ‘`=>`’ ⟨*big-func-body*⟩

⟨*parameters*⟩ ::= ⟨*identifier*⟩ | ‘`*`’ | ‘`(`’ [ ‘␣’ ] ⟨*parameters*⟩ ( ⟨*comma*⟩ ⟨*parameters*⟩ )+ [ ‘␣’ ] ‘`)`’

⟨*line-func-body*⟩ ::= [ ‘␣’ ] ( ⟨*basic-or-app-expr*⟩ | ⟨*line-op-expr*⟩ )

⟨*big-func-body*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ( ⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ )

⟨*cases-func-expr*⟩ ::= ⟨*cases-params*⟩ ⟨*case*⟩+ [ ⟨*end-case*⟩ ]

⟨*cases-params*⟩ ::= ⟨*identifier*⟩ | ‘`cases`’ | ‘`*`’ | ‘`(`’ [ ‘␣’ ] ⟨*cases-params*⟩ ( ⟨*comma*⟩ ⟨*cases-params*⟩ )+ [ ‘␣’ ] ‘`)`’

⟨*case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ⟨*outer-matching*⟩ [ ‘␣’ ] ‘`=>`’ ⟨*case-body*⟩

⟨*end-case*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ (‘`...`’ | ⟨*identifier*⟩) [ ‘␣’ ] ‘`=>`’ ⟨*case-body*⟩

⟨*outer-matching*⟩ ::= ⟨*simple-id*⟩ | ⟨*matching*⟩

⟨*matching*⟩ ::= ⟨*literal*⟩ | ⟨*pre-func*⟩ ⟨*inner-matching*⟩ | ⟨*tuple-matching*⟩ | ⟨*list-matching*⟩

⟨*inner-matching*⟩ ::= ‘`*`’ | ⟨*identifier*⟩ | ⟨*matching*⟩

⟨*tuple-matching*⟩ ::= ‘`(`’ [ ‘␣’ ] ⟨*inner-matching*⟩ ( ⟨*comma*⟩ ⟨*inner-matching*⟩ )+ [ ‘␣’ ] ‘`)`’

⟨*list-matching*⟩ ::= ‘`[`’ [ ‘␣’ ] [ ⟨*inner-matching*⟩ ( ⟨*comma*⟩ ⟨*inner-matching*⟩ )* ] [ ‘␣’ ] ‘`]`’

⟨*case-body*⟩ ::= ⟨*line-func-body*⟩ | ⟨*big-func-body*⟩ [ ⟨*where-expr*⟩ ]

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

⟨*value-expr*⟩ ::= ⟨*basic-or-app-expr*⟩ | ⟨*op-expr*⟩ | ⟨*func-expr*⟩ | ⟨*big-tuple*⟩ | ⟨*big-list*⟩

⟨*grouped-value-defs*⟩ ::=
  ⟨*indent*⟩ ⟨*identifier*⟩ ( ⟨*comma*⟩ ⟨*identifier*⟩ )+
  ( [ ‘␣’ ] ‘`:`’ [ ‘␣’ ] | ⟨*nl*⟩ ⟨*indent*⟩ ‘`:`␣’ ) ( ⟨*type*⟩ ( ⟨*comma*⟩ ⟨*type*⟩ )+ | ‘`all`␣’ ⟨*type*⟩ )
  ⟨*nl*⟩ ⟨*indent*⟩ ‘`=`␣’ ⟨*line-exprs*⟩ ( ⟨*nl*⟩ ⟨*indent*⟩ ⟨*comma*⟩ ⟨*line-exprs*⟩ )*

⟨*line-exprs*⟩ ::= ⟨*line-expr*⟩ ( ⟨*comma*⟩ ⟨*line-expr*⟩ )*

⟨*where-expr*⟩ ::= ⟨*nl*⟩ ⟨*indent*⟩ ‘`where`’ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩ ( ⟨*nl*⟩ ⟨*nl*⟩ ⟨*value-def-or-defs*⟩ )*

⟨*value-def-or-defs*⟩ ::= ⟨*value-def*⟩ | ⟨*grouped-value-defs*⟩

⟨*type*⟩ ::= [ ⟨*condition*⟩ ] ⟨*simple-type*⟩

⟨*simple-type*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | ⟨*func-type*⟩

⟨*type-id-or-var*⟩ ::= ⟨*type-id*⟩ | ⟨*type-var*⟩

⟨*type-id*⟩ ::= [A-Z] [A-Za-z]*

⟨*type-var*⟩ ::= ⟨*param-t-var*⟩ | ⟨*ad-hoc-t-var*⟩

⟨*param-t-var*⟩ ::= ‘`T`’ [0-9]

⟨*ad-hoc-t-var*⟩ ::= ‘`@`’ [A-Z]

⟨*type-app*⟩ ::=
   [ ⟨*types-in-paren*⟩ ] ⟨*type-id-with-args*⟩ [ ⟨*types-in-paren*⟩ ]
 | ⟨*types-in-paren*⟩ ⟨*t-id-or-ad-hoc-t-var*⟩ [ ⟨*types-in-paren*⟩ ]
 | ⟨*t-id-or-ad-hoc-t-var*⟩ ⟨*types-in-paren*⟩

⟨*type-id-with-args*⟩ ::= ⟨*type-id*⟩ ( ⟨*types-in-paren*⟩ [A-Za-z]+ )+

⟨*t-id-or-ad-hoc-t-var*⟩ ::= ⟨*type-id*⟩ | ⟨*ad-hoc-t-var*⟩

⟨*types-in-paren*⟩ ::= ‘`(`’ [ ‘␣’ ] ⟨*simple-type*⟩ ( ⟨*comma*⟩ ⟨*simple-type*⟩ )* [ ‘␣’ ] ‘`)`’

⟨*prod-type*⟩ ::= ⟨*field-type*⟩ ( ‘␣`x`␣’ ⟨*field-type*⟩ )+

⟨*field-type*⟩ ::= ⟨*power-base-type*⟩ | ⟨*power-type*⟩

⟨*power-base-type*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app*⟩ | ‘`(`’ [ ‘␣’ ] ( ⟨*prod-type*⟩ | ⟨*func-type*⟩ ) [ ‘␣’ ] ‘`)`’

⟨*power-type*⟩ ::= ⟨*power-base-type*⟩ ‘`^`’ ⟨*int-greater-than-one*⟩

⟨*func-type*⟩ ::= ⟨*in-or-out-type*⟩ ‘␣`=>`␣’ ⟨*in-or-out-type*⟩

⟨*in-or-out-type*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app*⟩ | ⟨*power-type*⟩ | ⟨*prod-type*⟩ | ‘`(`’ [ ‘␣’ ] ⟨*func-type*⟩ [ ‘␣’ ] ‘`)`’

⟨*condition*⟩ ::= ⟨*prop-name*⟩ ‘␣`-->`␣’

⟨type-def⟩ ::= ⟨tuple-type-def⟩ | ⟨or-type-def⟩

⟨tuple-type-def⟩ ::=
  `tuple_type␣` ⟨type-name⟩ ⟨nl⟩
  `value` ( '␣' | ⟨nl⟩ '␣␣' ) ⟨id-tuple⟩ [ '␣' ] ':' [ '␣' ] ( ⟨prod-type⟩ | ⟨power-type⟩ )

⟨type-name⟩ ::=
  [ ⟨param-vars-in-paren⟩ ] ⟨type-id⟩ ( ⟨param-vars-in-paren⟩ [A-Za-z]+ )* [ ⟨param-vars-in-paren⟩ ]

⟨param-vars-in-paren⟩ ::= '(' [ '␣' ] ⟨param-t-var⟩ ( ⟨comma⟩ ⟨param-t-var⟩ )* [ '␣' ] ')'

⟨id-tuple⟩ ::= '(' [ '␣' ] ⟨simple-id⟩ ( ⟨comma⟩ ⟨simple-id⟩ )+ [ '␣' ] ')'

⟨or-type-def⟩ ::=
  `or_type␣` ⟨type-name⟩ ⟨nl⟩
  `values` ( '␣' | ⟨nl⟩ '␣␣' )
  ⟨simple-id⟩ [ ':' ⟨simple-type⟩ ] ( [ '␣' ] | [ '␣' ] ⟨simple-id⟩ [ ':' ⟨simple-type⟩ ] )*

⟨t-nickname⟩ ::= `type_nickname␣` ⟨type-name⟩ [ '␣' ] '=' [ '␣' ] ⟨simple-type⟩

⟨type-prop-def⟩ ::= ⟨atom-prop-def⟩ | ⟨renaming-prop-def⟩

⟨atom-prop-def⟩ ::= ⟨prop-name-line⟩ ⟨nl⟩ `needed` ( '␣' | ⟨nl⟩ '␣␣' ) ⟨identifier⟩ [ '␣' ] ':' [ '␣' ] ⟨simple-type⟩

⟨renaming-prop-def⟩ ::=
  ⟨prop-name-line⟩ ⟨nl⟩ `equivalent` ( '␣' | ⟨nl⟩ '␣␣' ) ⟨prop-name⟩ ( ⟨comma⟩ ⟨prop-name⟩ )*

⟨prop-name-line⟩ ::= `type_proposition␣` ⟨prop-name⟩

⟨prop-name⟩ ::=
  [A-Z] ( ⟨name-part⟩ ⟨ad-hoc-vars-in-paren⟩ )+ [ ⟨name-part⟩ ]
  | ( ⟨ad-hoc-vars-in-paren⟩ ⟨name-part⟩ )+ [ ⟨ad-hoc-vars-in-paren⟩ ]

⟨ad-hoc-vars-in-paren⟩ ::= '(' [ '␣' ] ⟨ad-hoc-t-var⟩ ( ⟨comma⟩ ⟨ad-hoc-t-var⟩ )* [ '␣' ] ')'

⟨name-part⟩ ::= ( [A-Za-z] | '_'[A-Z] )+

⟨*type-theo*⟩ ::= ‘`type_theorem␣`’ ⟨*prop-name-with-subs*⟩ [ ‘`␣-->␣`’ ⟨*prop-name-with-subs*⟩ ] ⟨*nl*⟩ ‘`proof`’ ⟨*proof*⟩

⟨*prop-name-with-subs*⟩ ::=
  [A-Z] ( ⟨*name-part*⟩ ⟨*subs-in-paren*⟩ )+ [ ⟨*name-part*⟩ ]
| ( ⟨*subs-in-paren*⟩ ⟨*name-part*⟩ )+ [ ⟨*subs-in-paren*⟩ ]

⟨*subs-in-paren*⟩ ::= ‘`(`’ [ ‘`␣`’ ] ⟨*t-var-sub*⟩ ( ⟨*comma*⟩ ⟨*t-var-sub*⟩ )* [ ‘`␣`’ ] ‘`)`’

⟨*t-var-sub*⟩ ::= ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩

⟨*type-app-sub*⟩ ::=
  [ ⟨*subs-or-unders-in-paren*⟩ ] ⟨*type-id-with-subs*⟩ [ ⟨*subs-or-unders-in-paren*⟩ ]
| ⟨*subs-or-unders-in-paren*⟩ ⟨*t-id-or-ad-hoc-t-var*⟩ [ ⟨*subs-or-unders-in-paren*⟩ ]
| ⟨*t-id-or-ad-hoc-t-var*⟩ ⟨*subs-or-unders-in-paren*⟩

⟨*type-id-with-subs*⟩ ::= ⟨*type-id*⟩ ( ⟨*subs-or-unders-in-paren*⟩ [A-Za-z]+ )+

⟨*subs-or-unders-in-paren*⟩ ::= ‘`(`’ [ ‘`␣`’ ] ⟨*sub-or-under*⟩ ( ⟨*comma*⟩ ⟨*sub-or-under*⟩ )* [ ‘`␣`’ ] ‘`)`’

⟨*sub-or-under*⟩ ::= ⟨*t-var-sub*⟩ | ‘`_`’

⟨*power-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ ‘`^`’ ⟨*int-greater-than-one*⟩

⟨*power-base-type-sub*⟩ ::=
  ‘`_`’ | ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ‘`(`’ [ ‘`␣`’ ] ( ⟨*prod-type-sub*⟩ | ⟨*func-type-sub*⟩ ) [ ‘`␣`’ ] ‘`)`’

⟨*prod-type-sub*⟩ ::= ⟨*field-type-sub*⟩ ( ‘`␣x␣`’ ⟨*field-type-sub*⟩ )+

⟨*field-type-sub*⟩ ::= ⟨*power-base-type-sub*⟩ | ⟨*power-type-sub*⟩

⟨*func-type-sub*⟩ ::= ⟨*in-or-out-type-sub*⟩ ‘`␣=>␣`’ ⟨*in-or-out-type-sub*⟩

⟨*in-or-out-type-sub*⟩ ::=
  ‘`_`’ | ⟨*type-id-or-var*⟩ | ⟨*type-app-sub*⟩ | ⟨*power-type-sub*⟩ | ⟨*prod-type-sub*⟩ | ‘`(`’ [ ‘`␣`’ ] ⟨*func-type-sub*⟩ [ ‘`␣`’ ] ‘`)`’

⟨*proof*⟩ ::= ‘`␣`’ ⟨*id-or-op-eq*⟩ ‘`␣`’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ‘`␣␣`’ ⟨*id-or-op-eq*⟩ ⟨*tt-value-expr*⟩

⟨*id-or-op-eq*⟩ ::= ⟨*identifier*⟩ [ ⟨*op*⟩ ⟨*identifier*⟩ ] ‘`␣=`’

⟨*tt-value-expr*⟩ ::= ‘`␣`’ ⟨*line-expr*⟩ | ⟨*nl*⟩ ⟨*indent*⟩ ⟨*value-expr*⟩ [ ⟨*where-expr*⟩ ]

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

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

⟨*nl*⟩ :: ( ‘`␣`’ | ‘`\t`’ )* ‘`\n`’

### 6.1.2 Indentation system

The <*indent*> nonterminal in 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 <*indent*> nonterminal corresponds to $2 * il$ space characters. The indentation level follows the rules below:

**Indentation Rules**

1. At the beginnng: $il = 0$
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 the paremeters 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 correspoding (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 respresentations for each AST type
- `Parsers.hs`: Parsers for each AST type
- `Testing.hs`: Runs the parsers on the examples and prints the result

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 correspoding 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:

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

The AST type for the literal is:

```
data Literal =
  Int Int | 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 seperately:

```
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 and the ”`<|>`” operator means ”this parser or that parser”. Finally, the ”`<?>`” operator means ”if all the parsers fail show this in the error message”.

### 6.3.2 Example 1: List

The grammar rule for the list is the following:

⟨*list*⟩ ::= '`[`' [ '␣' ] [ ⟨*line-expr-or-unders*⟩ ] [ '␣' ] '`]`'

The AST type for the list is:

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

And the parser for the list is:

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

Here we use the "`*>`" and "`<*`" operators which parse both of the parsers that they have as operands (from left to right) but only keep the result of the parser that the "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 the 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:

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

The AST type for the "change" expression is:

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

And the parser for the "change" expression is:

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

Here we use the "`+++`" operator which is defined in the parser. It takes two parsers as operands and creates a parser that uses them sequentially and puts the two results in a tuple. "field_change_p" parses a single field change. "many" is defined in the library, it parses with the argument parser as many times as possible and puts the results in a list (the Kleene star of the parser world). Lastly, we also use the "try" parser combinator (defined in the library) which is used to avoid comsuming any input if the parser fails. For example if the input was "`.change ...`", the parser would fail because we have a space instead of an opening bracket. Without the "try" combinator it would consume the "`.change`" part and leave only "` ...`" after failing. This would result in the PostFuncAppEnd alternative parser below failing as well (the parser after "`DC1 <$>`" is the Change parser). Whereas with the "try" combinator we avoid this problem and the parser after "`PFsMDC <$>`" successfully parses ".change" as an alternative postfunc (get the "change" field of a tuple type value).

```
instance HasParser PostFuncAppEnd where
  parser = DC1 <$> parser <|> PFsMDC <$> many1 parser +++ optionMaybe parser
```

### 6.3.4 Example 3: Value Definition

The grammar rule for the value definition is the following:

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

The AST type for the value definition is:

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

And the parser for the value definition is:

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

    increase_il_by 1 >>

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

    increase_il_by 1 >> we_are_in_equal_line >>

    parser >>= \value_expr ->

    we_are_not_in_equal_line >>

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

    decrease_il_by 2 >>

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

In this example we see how the state of the parser is used to enforce the indentation rules. The "indent" parser parses <2 * the indentation level> spaces, getting the indentation level from the state (see Indentation System 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: ( [ ' ' ] ':' [ ' ' ] — ⟨*nl*⟩ ⟨*indent*⟩ ': '). "we_are_in_equal_line" and "we_are_not_in_equal_line" change the state to enforce rule 6 of the Indentation System.

# 7 Semantic Analysis

## 7.1 Name resolution

## 7.2 Type checking

## 7.3 Error messages

# 8 Haskell Generation

## 8.1 High Level Overview

Lcases is essentially a subset of Haskell with different syntax and naming, sprinkled with few new constructs. Anything that is not mentioned in the following sections is more or less the same in Haskell and lcases.

### 8.1.1 Correspondaces with Haskell

Here is the list of lcases constructs that have a Haskell correspondent:

- Prefix Functions ⟺ Data Constructors (with arguments)
- Postfix Functions (except ”.change” and ”.<special-id>”) ⟺ Accessor Functions generated by the Record Syntax
- ”.change” function ⟺ Accessor Functions generated by the Record Syntax
- Operators:
  Operators are not actually compiled to the corresponding ones in Haskell because the are some difference either in the precedence or in the types (in lcases they are more general). Nevertheless, they are similar enough to be considered ”correspondances”. Everything is compiled into function application in Haskell. (TODO will it?)
  - Left function application operator ”<-” ⟺ ”$” operator
    (althought it’s not compiled to that due to difference in precedence)
  - Left function application operator ”->” ⟺ ”&” operator from Data.Function
    (althought it’s not compiled to that due to difference in precedence)
  - Left function composition operator ”<o” ⟺ ”.” operator
  - Right function composition operator ”o>” ⟺ ”>.>” operator from GHC.Core.Map.Expr (or other equivalent ones from other esoteric libraries, why Haskell?)
  - ”!=” operator ⟺ ”\=” operator
  - ”&” operator ⟺ ”&&” operator
  - ”|” operator ⟺ ”||” operator
  - ”;>” operator ⟺ ”>>=” operator
  - ”;” operator ⟺ ”>>” operator
- one parameter (one lambda abstraction) ”a => ...” ⟺ ”\a -> ...”
- multiple parameters (multiple lambda abstractions) ”(a, b) => ...” ⟺ ”\a b -> ...”
- ”don’t care” parameter ”* => ...” ⟺ ”_ => ...”
  or one ”don’t care” in many ”(a, *, b) => ...” ⟺ ”\a _ b => ...”
- ”cases” function expressions ⟺ ”\case” syntax (LambdaCase extension)
  url: https://typeclasses.com/ghc/lambda-case

- "value definitions" are similar. In lcases the type and the expression are in one definition. This is technically possible in Haskell too, but the type must be after the expression and sometimes the expression must be put in parentheses for the type to extend to the whole expression. "value definitions" are compiled into a type annotation **and** a value definition in Haskell.
- "where" expressions are also similar. In lcases the values defined in a "where" expression can "see" the parameters of the outer expression when it is a function. "where" expressions in lcases are actually compiled to "let-in" expressions in Haskell.
- Type variables are almost the same. In lcases, there syntactic separation of ad hoc type variables (constrained type variables in Haskell) and parametric type variables whereas in Haskell they are all lower case letters.
- The condition (constaint in Haskell) arrow: "`-->`" $\Longleftrightarrow$ "`=>`"
- The product type: "`A x B x C`" $\Longleftrightarrow$ "`(A, B, C)`"
- Type definitions $\Longleftrightarrow$ "data" statements
- Type nicknames $\Longleftrightarrow$ "type" statements
- Atomic type propositions $\Longleftrightarrow$ Type classes
- Type theorems $\Longleftrightarrow$ Type class instances

# 9 Running Examples

# 10 Conclusion

# References

[1] Daan Leijen. *Parsec, a fast combinator parser*. 2001. URL: http://www.cs.uu.nl/~daan.