FIT2102 Assignment II - Report

Simon Frédéric Paul RACAUD - 33287430

October 2022

1 Design of the code

1.1 Lambda calculus parser

For the first part, I decided to create a hybrid parser managing long and short forms of lambda expressions. The choice between the two modes is done by giving a Boolean as parameter. True for long form and False for short form.

My lambda parser follows this BNF:

```
# Long and Short form parser
<lambda> ::= <wrappedFunction> <lambda> | <wrappedFunction> | <function>
<wrappedFunction> ::= "(" <function> ")"
<function> ::= "\" <variables> "." <applicationTerm>
<applicationTerm> ::= <item> | <item> <applicationTerm>
<item> ::= <terms> | "(" <expression> ")" | <function>
<expression> ::= <applicationTerm> | <function>
<terms> ::= <letter> <terms> | <letter>
<letter> ::= 'a' ... 'z'

# Short form:
<variables> ::= <letter> <variables> | <letter>

# Long form:
<variables> ::= <letter>
```

Each BNF expression corresponds to a small function. Since the parser starts with the *lambda* expression, only functions can be parsed. Isolated terms like "(xx)" or "x" are not managed. That restriction was added since the builder engine doesn't support expressions with free variables.

1.2 Logic parser

My logic parser is an improvement of the previous one with the usage of intermediate data structures.

The code flow is separated into two sections:

- Parsing of the string to an expression Tree
- Resolution of the Tree to reduce it to only one lambda Builder

The parser roughly follows the following BNF:

```
<stmt> ::= <ifCond> |  "(" <ifCond> ")" | <expr>
<ifCond> ::= "if" <expr> "then" <stmt> "else" <stmt>
<expr> ::= <param> <duop> <expr> | <unop> <expr> | "(" <expr> ")" | <bool>
<param> ::=  "(" <expr> ")" | <bool>

<bool> ::= "True" | "False"
<unop> ::= "not"
<duop> ::= "and" | "or"
```

The expressions are parsed using the LogicExpr and Stmt data structures. The Stmt data structure allows to declare if conditions at the root of the expression. The operations, values and sub-expressions are parsed using LogicExpr.

The tree resolver simply applies operation resolver functions in a specific order to resolve the whole tree. The *parseBracket* function start by resolving every sub-expressions inside brackets. The *parseNot* function browse the tree and resolve every 'not' operation it encounter. Same thing for *parseAnd* and *parseOr*.
At the end of the process, the remaining Var element will contain the final lambda builder.

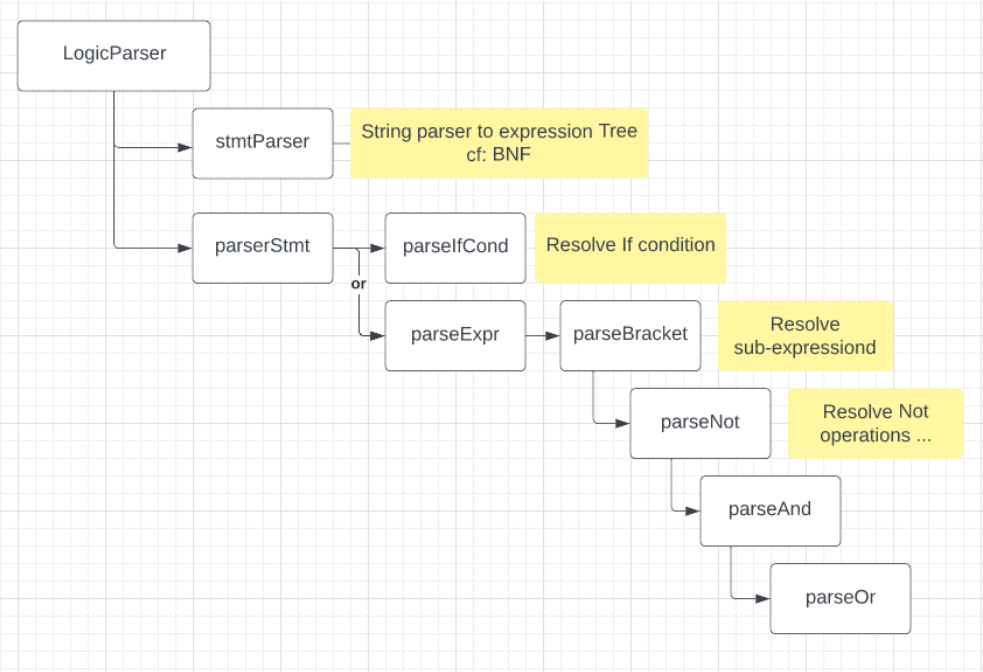

Figure 1: Architecture of the logic parser

1.3 Arithmetic parser

The arithmetic parser follows the same method as the logic parser.
The integers are extracted during the parsing of the string to become lambda builders using *intToLam*. The data structure used to create the expression is simpler than previously as I only need to stock dual operand operation, single value and sub-expression.
The tree resolver has been improved to merge every operator resolver functions into one generic resolver. The operations are resolved using the *Op* enum with the *solver* function.

The parser has also been improved with more generic functions. Instead of having one function to parse each operator, I only have the *parseOperation* function which takes the operator to parse as parameter.

The parser follows this BNF architecture:

```
<expr> ::= <operation> | <bracket> | <number>
<bracket> ::= "(" <expr> ")"
<operation> ::= <param> <op> <expr>
<param> ::=  <bracket> | <number>
<number> ::= <digit> <number> | <digit>
```

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<op> ::= "+" | "-" | "**" | "*"
```

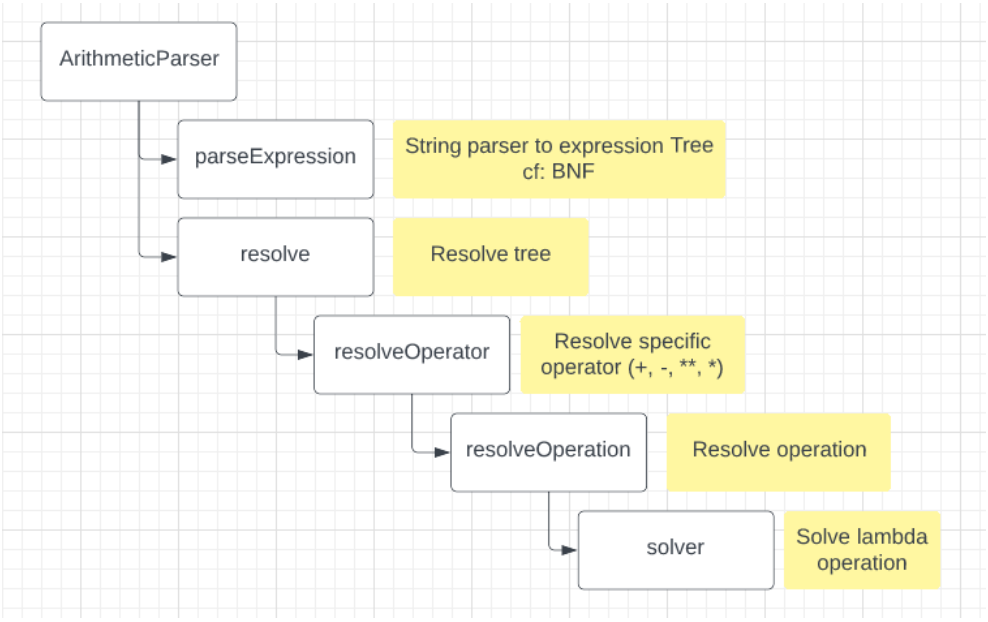

Figure 2: Architecture of the arithmetic parser

1.4 Complex parser

The complex parser still uses an expression Tree like the two previous ones. However, big improvements have been made to create a code as generic as possible. This parser merges the functionalities of the Logic and Arithmetic parsers. Adding the new possibility to parse comparison expressions.
The Tree generator has been created using the following BNF:

```
<stmt> ::= <ifCond> |  "(" <ifCond> ")" | <logicExpr> | <arithmExpr>
<ifCond> ::= "if" <logicExpr> "then" <stmt> "else" <stmt>

<logicExpr> ::= <logicDuop> | <logicUnop> | "(" <logicExpr> ")" | <logicParam>
<logicDuop> ::= <logicPrevParam> <duop> <logicExpr>
<logicUnop> ::= <unop> <logicParam>
<logicPrevParam> ::=  "(" <logicExpr> ")" | <logicParam>
<logicParam> ::= <bool> | <comparisonExpr>

<comparisonExpr> ::= <comparisonOperation>
<comparisonOperation> ::= <comparisonPrevParam> <cmpOp> <comparisonParam>
```

```
<comparisonPrevParam> ::= "(" <comparisonExpr> ")" | <number> | <arithmExpr>
<comparisonParam> ::= <comparisonExpr> | <number> | <arithmExpr>

<arithmExpr> ::= <operation> | <number> | "(" <arithmExpr> ")"
<operation> ::= <arithmPrevParam> <arithmOp> <arithmExpr>
<arithmPrevParam> ::= "(" <arithmExpr> ")" | <number>

<number> ::= <digit> <number> | <digit>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<bool> ::= "True" | "False"

<arithmOp> ::= "+" | "-" | "**" | "*"
<cmpOp> ::= ">=" | "<=" | "==" | "!=" | "<" | ">"
<unop> ::= "not"
<duop> ::= "and" | "or"

<spaces> ::= " " <spaces> | " "
```

The tree resolver extends the possibilities of the one in the Arithmetic parser to parse both dual operand and unary operand operations.

1.5 List parser

The list parser is very basic compared to the previous exercises. The *listParser* function parse a list of expression located under square brackets and separated by commas. As you can see in *itemParser*, a list can contain other lists, complex expressions or short form lambda expressions. The list is parsed using the *array* function (from Builder.hs) which allows performing this task efficiently.

The function *functionListParser* can parse and append a list of function before the array. The functions are appended from the right to the left starting with the array.

2 Parsing

In this part, I will list the new parser functionalities used in each exercise.

For each parser, monads were used in order to chain the usage of parsing functions. The same argument is valid for ||| to chain parsing functions with a conditional application.
Functors were mainly used to apply a function to a Builder inside a Parser.
To develop each parser (except for the list which is very easy), I wrote a BNF (cf previous section). The BNF is a convenient tool to conceive a parser. With this tool, I was able to split the development process into conception and realization parts.

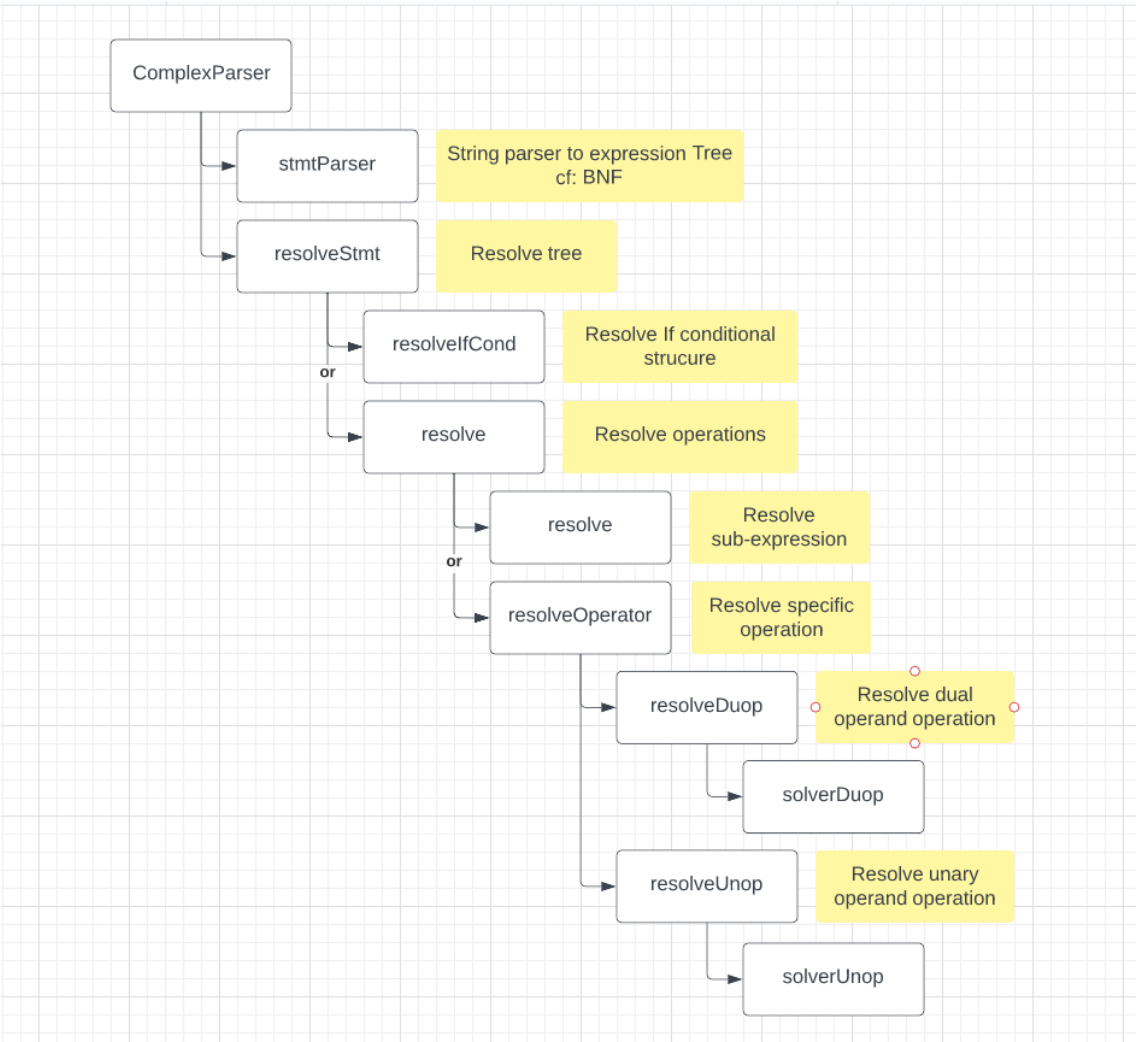

Figure 3: Architecture of the complex parser

2.1 Lambda parser

In part 1, the parser is generated mainly using |||, 'is', 'oneof' and 'list1' functions from the builder module. To extract terms and variables, I used oneof in order to only extract letters. list1 was used to chain the letter parser and foldl was used to combine the resulting array into one Builder. The *foldBuilders* function has been created to append a list of builders into one.

2.2 Logic parser

For the logic parser, I started using *stringTok* which is an improvement of the *string* function skipping spaces after the token. I also used the *bracket* function which is simply using the *between* function to apply a parser on an expression between brackets.

2.3 Arithmetic parser

The arithmetic parser is using *munch1* to extract number digits and *readMaybe* has been used to convert that string to an integer.
I used pattern matching in the *getOp* function to convert operators' symbols to their Enum value.

2.4 Complex parser

The complex parser can be sliced into three layers:

1. Logical expression parser
2. Comparison expression parser
3. Arithmetic expression parser

Some common generic functions have also been realised to avoid code duplication between each layer.
Each layer can use a lower one as an expression parameter. E.g. The parameter of a logical expression can be a comparison and the parameter of a comparison expression can be an arithmetic one.

The complex parser use the *int* function, in *numberParser*, to extract the numbers. The *parseEach* function, in *parseEachOperator*, is used to chain the usage of ||| using a list of operator.

2.5 List parser

As mentioned in the first section, the list parser uses the *array* function to parse a list of expressions separated by commas.

3 Functional Programming

The parsers functions have been created to follow the sentence: "One function, one responsibility". For example, in the first task the *function* function only parse one lambda, the *expression* function parses one expression, ... That makes the code far more readable and testable. Indeed, since each function parses a sub-section of the expression, I can just call them individually to test their behaviours in an atomic way.

Since I was still learning parsing in Haskell, the modularity evolve at each exercise. In the complex parser, generic functions like *duoOperationParser* have been created to parse generic patterns of expression. For instance, *parseEachOperator* use a list of operator and a callback returning a parser given as parameter and call this parser for each operator in the operator array until the operation succeeds. That function allows me to parse generically any operation.

First-class and higher-order functions haven't been much used in my project since I mostly need parser combinators to resolve the tasks. But callbacks were used like in *parseEachOperator* (Part2Complex) in order to create more generic code managing a wide range of operations.
Every function of the parsers is pure. They will always return the same output for the same input. Recursion has been used everywhere to parse dynamic expressions or just for simple operations like combining Builders with fold.

4 Haskell Language Features Used

To define expression Trees, I defined multiple data structures. Enums were used to define operators. Expression trees were implemented to separate the parsing part of the expression resolution.

Monads were used to chain and combine parser functions. That was necessary to parse formatted expressions like mathematical operations *"(operand) (operator) (operand)"*.

$* >$ and $< *$ functions were useful in order to skip spaces during the parsing.

Pattern matching was heavily used by the Tree resolvers in order to process specific expressions from the tree (e.g. Var, SubExpr or Calc).

liftA2 was used for example to append two builders which were located inside a Parser. (e.g. cf: Part1.hs, lambda function)