FIT2102 Programming Paradigms 2022

Assignment 2: Lambda Calculus Parser

Due Date: 23:55, October 23rd, 2022

Weighting: 30% of your final mark for the unit
Overview: Implement an interpreter that translates input strings into lambda calculus expressions using parser combinators in Haskell. This should highlight your understanding of Haskell, functional programming, and its application; as well as lambda calculus. You will also need to write a report to demonstrate your knowledge and understanding.
Building and using the code: The code bundle is packaged the same way as tutorial code. To compile the code, run: `stack build`. To run the main function, run: `stack run`

Submission Instructions

Submit a zipped file named <studentNo>_<name>.zip which extracts to a folder named <studentNo>_<name>

- **It must contain all the code that will be marked** including the **report** and ***submission/LambdaParser.hs***
- Your assignment code and your report in PDF format go in the ***submission/*** folder of the code bundle. To submit you will zip up **just** the contents of this **submission** *folder*
- It should include sufficient **documentation** so that we can appreciate everything you have done (readme.txt or other supplementary documentation)
- You also need to include a report describing your design decisions. The report must be named **<studentNo>_<name>.pdf**.
- Only Haskell built in libraries should be used (i.e. no additional installation should be required)
- **Before zipping, run `stack clean --full` (to ensure a small bundle)**
- **Make sure the code you submit executes properly.**

The marking process will look something like this:

- Extract **<studentNo>_<name>.zip**
- Copy the **submission** folder contents into the assignment code bundle submission folder
- Execute `stack build`, `stack test`, `stack run`

Please ensure that you test this process before submitting. Any issues during this process will make your marker unhappy. **Failure to follow these instructions may result in deductions.**

Table of Contents

Introduction

In this assignment, we will use Haskell to develop an interpreter that parses a string into the data types provided in `Builder.hs and Lambda.hs.` Using this we can create Lambda Calculus expressions which we are then able to normalise into a simplified but equivalent expression (evaluating the expression).

This assignment will be split into multiple parts, of increasing difficulty. While many lambda expressions for earlier tasks have been provided to you in the course notes and in this document, you may have to do your own **independent research** to find church encodings for more complicated constructs.
You are welcome to use any of the material covered in the previous weeks, including solutions for tutorial questions, to assist in the development of your parser. **Please reference ideas and code constructs obtained from external sources**, as well as anything else you might find in your independent research, for this assignment.

Goals / Learning Outcomes

The purpose of this assignment is to highlight lambda calculus as a method of computation and apply the skills you have learned to a practical exercise (parsing).

Use functional programming and parsing effectively
Understand and be able to use key functional programming principles (HOF, pure functions, immutable data structures, abstractions)
Apply Haskell and lambda calculus to implement non-trivial programs

Scope of assignment

It is important to note that **you will not need to implement code to evaluate lambda calculus expressions**. Functions to evaluate lambda calculus expressions are provided. Rather, you are only required to **parse** an expression into the data types provided in `Builder.hs` and then use the given functions to evaluate the expression.

You will need to have some understanding of lambda calculus, particularly in the latter parts of the assignment. Revise the notes on Lambda Calculus and come to consultations early if you have any uncertainty.

Lambda Calculus syntax

Lambda Calculus can be written in an explicit way (long form), using brackets to show ordering of all operations, or in an implicit way (short form), by removing unnecessary syntax.
For example, consider the expression for a Church Encoded IF in short form
λbtf.b t f
We introduced this short syntax for lambda expressions as it's a little easier to read. However, let's also consider the long form version. First we need to include all lambdas
λb.λt.λf.b t f
Then, we need to include brackets around every expression (note that applying the function "b" to two parameters "t" and "f" is one expression)
(λb.(λt.(λf.b t f)))
It may be simpler to parse the long-form syntax of lambda calculus compared to the short form.

Getting started

The first step which we recommend is to play around with the code base, and see how you can build Lambda Expressions.

Step 1. Try to use the builder and GHCI to construct and normalise the following expressions:

(λb.(λt.(λf.f t b)))
(λx.(λy.xx(λz.z)x))
For extra practice build the lambda expressions in the Week 5 Tutorial Sheet. See which ones successfully build and think about why!

Step 2. Try to describe the syntax for a verbose Lambda Calculus using a BNF Grammar.
Step 3. Try to construct parsers for and test each part of this grammar separately.
Step 4. Combine and test your code!

What is provided

Provided to you is an engine which can Beta/Eta reduce Lambda Calculus into beta-normal form.
This engine is built around the `Builder` type. This `Builder` type allows you to create Lambda Calculus expressions, which can then be normalised.
First, how do you build a lambda expression? Let's consider the expression

λx.x

`Builder` representation

```
build $ lam 'x' (term 'x')
>>> \x.x
```

We use `build` to construct the lambda expression
We use `lam` , similar to the way you use λ in lambda expressions.
The first argument is the function input, in this case `x`
The second argument is the return value of the function. We use `term` to
identify this is a variable.

Let's look at the types of all these expressions a little more.

```
lam :: Char -> Builder -> Builder
```

Takes a `char`, which represents the input variable
Takes an expression of builder type, which represents the return value of the Lambda Expression
Returns a value of `Builder` type.

```
term :: Char -> Builder
```

Takes a `char`, which represents the variable
Returns a value of the `Builder` type.

```
build :: Builder -> Lambda
```

This should be the last step, which takes a builder and constructs the `Lambda` expression.

Takes a `Builder` type
Returns a `Lambda` type.

Note: this fails if the expression contains free variables.

```
ap :: Builder -> Builder -> Builder
```

Combines two `Builder` expressions by applying one builder to another

Takes a builder
Takes another builder
Returns a builder, where the second builder will be applied to the first builder

For example,

(λx.x)(λx.x)
```
let id = lam 'x' (term 'x') -- Builder
expression
build $ id `ap` id
>>> (\x.x)\x.x
```

A more complex example:

(λb.(λt.(λf.b t f)))
```
build $ lam 'b' $ lam 't' $ lam 'f' ((term 'b')
`ap` (term 't') `ap` (term 'f'))
>>> \btf.btf
```

```
normal :: Lambda -> Lambda:
```

Normalises a Lambda expression by reducing it to Beta-Normal Form.
NOTE: This will cause an infinite loop if you try to normalise a divergent expression.

```
let k = lam 'x' $ lam 'y' (term 'x')
let i = lam 'x' (term 'x')
normal $ build $ k `ap` i
>>> \yx.x
```

While you are not required to evaluate your lambda calculus expressions, we also provide you with some evaluator functions to aid in testing:

```
lamToBool :: Lambda -> Maybe Bool:
```

Normalises a Lambda expression and then returns its Boolean evaluation (if it has one).

```
lamToInt :: Lambda -> Maybe Int:
```

Normalises a Lambda expression and then returns its numeric evaluation (if it has one).

Additional functions and types

Feel free to have a look at the '`Builder.hs`' file, which will provide some tests showing more usage of this type. Note that it is not important that you understand how these functions work, just that you understand how to **use** them.
There are many other well-documented functions that you can look at and use. Remember that it is not necessary to understand the implementation, only their usage (think of it like a library that you use).

Exercises

These exercises provide a structured approach for creating an interpreter.

Part 1: parsing lambda expressions
Part 2: simple arithmetic and boolean operations
Part 3: extending the interpreter to handle more programmatic operations

IMPORTANT: In each of the exercises, there will be

Deliverables: Functions/parsers that you must implement or documentation you have to complete to successfully complete the exercise

The functions/parsers **must** be named and have the same type signature as specified in the exercise otherwise they will break our tests
These functions must be implemented in *submission/LambdaParser.hs* as per the code bundle otherwise they will break out tests

Recommended steps: How to get started on the exercise. These are suggestions and you may wish to use a different approach

Basic tests will be provided, however it is important to **construct your own unit tests** and add to the existing tests for each task to aid in your development. Similarly, the tests provided will **not** be a proof of correctness as they may not be exhaustive, so it's important you ensure that your code is provably correct.
Marks for the tasks will come from

Correct implementations (i.e. passes the tests provided and our own tests)
Effective usage of course content (HOF, Functor, Applicative, Monad, etc.)
Good code quality (functional/declarative style, readability, structure, documentation etc.).

Please refer to the Marking rubric section for more information.

Part 1 (10 marks)

By the end of this section, we will have a parser for lambda calculus expressions.

Exercise 1 (2 marks): Construct a BNF Grammar for lambda calculus expressions

Deliverables

At the end of this exercise, we should have the following:

A BNF grammar to demonstrate the structure of the lambda expression parser, representing both short and long form in **one** grammar (as your parser should also handle both short and long form).

Recommended steps

Construct parsers for lambda calculus expression components ("λ", ".", "(", ")", "x", "y", "z", etc.)
Use the component parsers to create parsers for simple combinators to get familiar with parsing lambda expressions and their structure
Construct a BNF grammar for short form and long form lambda expressions

Exercise 2 (4 marks): Construct a parser for long form lambda calculus expressions

Deliverables

At the end of this exercise, we should have at least the following parser:

`longLambdaP :: Parser Lambda`

Parses a long form lambda calculus expression

Your BNF grammar must match your parsers. It is highly recommended to use the same name for your parsers and non-terminals (i.e. a non-terminal like <lambdaChar> should correspond to a `lambdaChar` parser) so your marker can easily validate the grammar.

Recommended steps

- Build a general purpose lambda calculus parser combinator which:
  - Parses general multi variable lambda expressions/function bodies
    - **Note** default associativity, e.g. λxy.xxy = λxy.(xx)y
  - Parses general multi variable lambda expressions/function bodies **with brackets**
    - E.g λxy.x(xy)
  - Parses any valid lambda calculus expression using **long-form syntax**
    - E.g. (λb.(λt.(λf.b t f)))

Exercise 3 (4 marks): Construct a parser for short form lambda calculus expressions

Deliverables

At the end of this exercise, we should have at least the following parsers:

`shortLambdaP :: Parser Lambda`

Parses a short form lambda calculus expression

`lambdaP :: Parser Lambda`

Parses both long form and short form lambda calculus expressions

Similar to Exercise 2, your **parser must match your BNF grammar.**

Recommended steps

- Build a general purpose lambda calculus parser combinator which:
  - Parses any valid lambda expression using **short-form syntax**
    - E.g. λbtf.b t f

Part 2 (8 marks)

By the end of this section, we should be able to parse arithmetic and logical expressions into their equivalent lambda calculus expressions.

Exercise 1 (2 marks): Construct a parser for logical statements

Deliverables

At the end of this exercise, you should have the following parsers:

`logicP :: Parser Lambda`

Parse simple to complex logical clauses

Recommended steps

Construct a parser for logical literals (“true”, “false”) and operators (“and”, “or”, “not”, “if”) into their church encoding

Use the logical component parsers to build a general logical parser combinator into the equivalent church encoding, which:

- Correctly negates a given expression
  - E.g. not not True
- Parses complex clauses with nested expressions
  - E.g. not True and False or True
- Parses expressions with the correct order of operations (“()” -> “not” -> “and” -> “or”)

Exercise 2 (4 marks): Construct a parser for arithmetic expressions

Requirements

At the end of this exercise, you should have the following parsers:

`basicArithmeticP ::  Parser Lambda`

Parses simple arithmetic expressions (+, -)

`arithmeticP :: Parser Lambda`

Parses complex arithmetic expressions (+, -, *, **, ()) with correct order of operations

Recommended steps

Construct a parser for natural numbers into their church encoding (“1”, “2”, …)

Construct a parser for **simple** arithmetic operators with **natural numbers** into equivalent lambda expressions. (“+”, “-”)

- See the Parser combinators section of the notes for some examples

Construct a parser for **complex** mathematical expressions with **natural numbers** into their equivalent lambda expressions. (“*”, “**”, “()”)

It may be useful to write a BNF for this

Using the component parsers built previously to build a parser combinator for complex arithmetic expressions.

Note: the correct order of operations, e.g. 5 + 2 * 3 - 1 = 5 + (2 * 3) - 1

Exercise 3 (2 marks): Construct a parser for comparison expressions

Requirements

At the end of this exercise, you should have the following parsers:

`complexCalcP :: Parser Lambda`

Parses expressions with logical connectives, arithmetic and in/equality operations

Recommended steps

Construct a parser for complex conditional expressions into their equivalent church encoding (>, <, <=, >=, ==, !=)

Using the parsers from the previous exercises, create a parser which parses arithmetic expressions separated by logical connectives and in/equality operations

E.g. (1 + 3 < 5 * 3) == (7 * 2 != 4) and (3 - 1 == 5 - 3)

Part 3 (7 marks)

This section of the assignment will include a sequence of exercises that may build to parse basic Haskell functionality, which can be used to build more complex Haskell expressions or structures.

Exercise 1 (3 marks): Construct a parser for basic Haskell constructs

Requirements

At the end of this exercise, you should have the following parsers:

`listP :: Parser Lambda`

Parses a haskell list of arbitrary length into its equivalent church encoding

`listOpP :: Parser Lambda`

Parse simple list operations into their church encoding

Recommended steps

- Construct a parser for Haskell lists (empty lists, lists of n elements, arbitrary sized lists)
- Construct parsers for simple list operations (`null`, `isNull`, `head`, `tail`, `cons`)
  - **Hint:** Haskell uses cons lists to represent lists

Exercise 2 (4 marks): Construct a parser for more advanced concepts

Requirements

At the end of this exercise you should have parsers that:

- Parse some complex language features (of your choice) that bring you closer to defining your own language.

Recommended steps

You may choose a number of the below examples, or you can come up with your own (but check with the teaching team if they are considered complex enough). Implementing each of the following may earn you up to 2 marks (depending on the quality of the solution). Choose two or more to achieve up to 4 marks total.
Some suggestions include parsers for:

- Recursive list functions (e.g. map, filter, foldr, etc.)
  - These will involve implementing some form of recursion
  - Note the discussion of recursive lambda calculus functions in the notes, there are many ways to implement recursive functions, including but not limited to fixed point combinators (e.g. Y or Z combinator). Try to notice the downside of each approach.
- Other functions such as:
  - Fibonacci
  - Factorial
  - Euclidean algorithm
  - Euler's problem
  - Division
- Variables
- Parsers with error handling
  - See week 11 tutorial for how to handle errors that come up in parsing
- Known algorithms
  - Binary search
  - Quick/Insertion/Selection sort
- Negative numbers/Decimal numbers
- Create your own language!

Report (5 marks)

You are required to provide a report in PDF format of max. 1200 words (markers will not mark beyond this word limit), description of extensions can use up to 600 words per extension feature. You should summarise the intention of the code, and highlight the interesting parts and difficulties you encountered.
In particular, describe how your strategy (and thus your code) evolved. You should focus on the **"why" not** the **"how".**
Additionally, just posting screenshots of code is **heavily discouraged**, unless it contains something of particular importance. Remember, markers will be looking at your code alongside your report, so we do not need to see your code twice.
Importantly, this report must include a BNF grammar and a description about why and how parser combinator helped you complete the parsing.
In summary, your report should include the following sections:

- (0.5 mark) Design of the code (including data-structures)
  - High level description of approach
  - High level structure of code
  - Code architecture choices
- (1.5 marks) Parsing
  - BNF grammar, but the BNF grammar marks are given in the earlier section
  - Usage of parser combinators
  - Choices made in creating parsers and parser combinators
  - How parsers and parser combinators were constructed using the Functor, Applicative, and Monad typeclasses
- (1 marks) Functional Programming (focusing on the **why**)
  - Small modular functions
  - Composing small functions together
  - Declarative style (including point free style)
- (2 mark) Haskell Language Features Used (focusing on the **why**)
  - Typeclasses and Custom Types
  - fmap, apply, bind
  - Higher order functions
  - Function composition
  - Leveraging built in functions
- Description of Extensions (if applicable)
  - What you intended to implement
  - What you did implement
  - What is cool/interesting/complex about it

Marking breakdown

There are two main evaluation criteria for your assignment. For each exercise there will be **1 mark** designated for the **quality** of the provided solution, and the **remaining marks** will be allocated to **correctness.**

Correctness

You will be provided with a handful of tests for each exercise (excluding Part 3 - Exercise 2). On top of these tests, tutors will run an additional test suite to measure the robustness of your code. Marks will be awarded proportionally for passing the tests for each exercise. It is highly recommended that you create your own tests as you go, on top of those provided, and that you consider possible edge cases. Correctness also relates to the correctness of your approach. That is, how well you've applied concepts covered from the unit content.
You must apply concepts from the course. The important thing here is that you need to use what we have taught you effectively. For example, defining a new type and its `Monad` instance, but then never actually needing to use it will not give you marks. (Note: using bind `(>>=)` for the sake of **using the `Monad`** when it is not needed will not count as "effective usage.")
Most importantly, code that does not utilise Haskell's language features, and that attempts to code in a more imperative style will not be awarded high marks.

Code quality

Code quality will relate more to how understandable your code is. You must have readable and **functional** code, commented when necessary. Readable code means that you keep your lines at a reasonable length (< 80 characters), that you provide comments above non-trivial functions, and that you comment sections of your code whose function may not be clear.

Additional information

Common Mistakes

- Haskell is a functional language. Do **not** write very large do blocks which handle all of your logic. Think carefully about your context and only use do

notation when applicable.
- Please do not write unnecessary functions reproducing functions from the prelude.

, e.g., the following is just map.

```
applyToList f (x:xs) = f x : applyToList f xs
applyToList f [] = []
```

- Try to use appropriate Prelude functions when you can. For examples of this, please see the 'Exercises' files that have been included since Week 6
- Eta-reduce when easy. The add2List function should be eta-reduced to remove the l.

```
add2ToList l = map (+2) l
add2ToList = map (+2)
```

- Do **not** write excessively point-free code like this (please):

```
find' = (. ((find .) . (.) . (==))) . (.) . (.) . maybe -1
```

Remember, the point of documentation is desecribe how a function is **used** (i.e. a manual) rather than describing the implementation details. In the case of a function, you would explain how to use it rather than obvious parameters, return types, etc.
The point of section/block comments are to describe blocks of code at a high level to aid in readability and support the overall logical flow of your code.
The point of inline comments are to justify the usage of particular constructs (e.g. using a `foldr` instead of `foldl`) or to explain how a particularly non-obvious part of the code works (e.g. describing what a complex maths expression does; note that excessive inline comments may indicate overly complex or poorly designed code).

Writing reports

The focus of the report should not be describing the code as that's what documentation (function headers, comments, etc.) are for. The purpose of the report is to demonstrate that you understand the code you have written and help your marker appreciate the work you've done.

The parts about Functional Programming and Haskell Language Features would be about the choices you have made in your code. It should also justify why those choices were made and how they were useful. Some examples here that reference parts of your code is okay, but the focus shouldn't be on the particular code.

The examples below reference assignment 1 as it will not conflict with the topics you will discuss in your report.

Good

"the Model-View-Controller architecture helps separate pure components like data and data transformation from impure svg updates, so errors in the code can be quickly identified"

"I limit side effects and maintain purity as much as possible because ..."

"I use/do X to Y because Z"

Can include an example: "I use small module functions (e.g. range) ..."

Y is identifying the FRP principle or highlighting a particular piece of knowledge

Z is the justification of why that is relevant to your code (e.g. more understandable, easier to read, extensible, testable, etc.)

Not so good

"In this screenshot, I use X on line 100, which does Y and then Z which then returns ...."

"... because we have to use pure code"

"... because we can't use let"

"... because the unit said so"

Revision History

28/09/2022: First published with notice of potential (likely) updates and clarifications
08/10/2022: Fixed allocation of marks.
Part 2 is worth 8 marks (up from 7)
Report is worth 5 marks (down from 6)
Design decisions section is worth 0.5 marks (down from 1)
Parsing section is worth 1.5 marks (down from 2)
19/10/2022: 2 day extension