Tutorial #2 - Working on your assignment

Habib Ghaffari

2022-09-19 Mon

Contents

1 Introduction

The objective of this tutorial is to work on some examples that will help you with your assignments. For this tutorial, I am still going to continue working with `Haskell`. The solutions we are presenting may/may not be related to your assignment. It is your responsibility to find the relevance and try to figure out how you could use it in your submissions. I am also going to give you a brief introduction to the `Hakell` while I am presenting this tutorial.

2 Objective

- Becoming familiar with `Haskell`.
- Learn how to expand the regular expression based on what we defined last week.
- Learn to apply `BNF` for our definitions.

- Learn how to use Haskell for the implementation of a recursive parser to parse our expressions.

3 Learn how to work with Haskell

3.1 Installing Haskell interpreter

GHCI is Haskell interpreter It is an interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreter. But in order to have the most stable installation, we recommend you install GHCI using Stack.

Stack is a cross-platform program for developing Haskell projects. It is aimed at Haskellers both new and experienced. In order to install Stack you can download it for your machine and the OS you are using from here.

After installation of Stack there are some other things that you can do in order to take the best out of it. One of the best IDEs for Haskell implementation is IntelliJ IDEA. In order to make Stack work properly with this IDE follow the instruction below.

- Open your Terminal or Command Prompt and run the following command

```
stack install hindent stylish-haskell
```

- Download and install IntelliJ IDEA. Since you are a student you can use IDEA and other IDEs of IntelliJ for free as long as you use your academic email address. The process is straightforward. Just register for an account in IntelliJ's website and apply for an activation license. You may ask to renew your license annually.
- Open IDEA and install Intellij-Haskell plugin and restart the IDE.
- Open the IDEA again and make a new project.
  - Select Stack binary.
  - In the next step choose the proper SDK. Most of the time IDEA automatically going to detect the stack path. If not just manually select it.

Now you are ready to start working with IDEA and Haskell programming.

Note: You can also use other IDEs such as VSCode while using Stack. For that just try to find the proper instruction to configure your IDE.

Now it is time to do some Haskell programming.

3.2 Introduction to Haskell programming

The best resources to learn `haskell` programming are listed below:

- `http://learnyouahaskell.com`
- `https://wiki.haskell.org/Learning_Haskell`

Here we are going to do some live coding which will introduce you to the `types` and `typeclasses` as well as `Algebraic Data types`.

You can check the example hs file later if you want to review the material.

Let's define `Efficiency` and `Safety`.

- **Efficiency** means ease of use for the programmer, and ease of achieving the programming goals.
- **Safety** means no unintended errors.
- `Haskell` is a statically typed language. This means that the type errors can be caught at compile time. This makes `Haskell` a very safe language to work with.
- The `Haskell` interpreter can usually infer the types of expressions based on the definitions you are making. In this manner, `Haskell` can be consider `Efficient` programming language. However, for more complicated functions and in most of the cases, it is better to write out the type signature of a function. It means more work for the developer which means less efficient programming language.
- Indention is mandatory in `Haskell`. Without `Indention` the interpreter may not able to detect the function definition, pattern matchings, etc without indention. Mandatory indention makes programming harder unless you use a proper IDE. Therefore in this matter, `Haskell` is not an `Efficient` programming language.
- `Haskell` can be written using braces and semi-colons, just like `C`. However, no one does. Instead, the "layout" rule is used, where spaces represent scope. The general rule is: always indent. When the compiler complains, indent more. Based on this we can say `Haskell` has some sort of `Efficiency` for programmers as well.
- `Haskell` uses Generational garbage collection (GC). In fact, it has one of the fastest garbage collection strategies known as nursery. However,

it use a bit more space to handle the garbage collection. But it is not important for us when we are talking about `safety`. So we could consider `Haskell` as a very `safe` programming language. For more information about garbage collection read the following link:

https://www.channable.com/tech/lessons-in-managing-haskell-memory

Here I tried to cover some of the tasks you need to do for questions number 1 to number 5. It is just a matter of investigation and figuring out how could you compare `efficiency` vs `safety` when it comes to `Python`.

3.3 Naming Conventions in Haskell

Names in Haskell must satisfy the following simple rules:

- Types and typeclasses must start with an uppercase letter
- Functions and variables must start with a lowercase letter
- Top-level operator functions must start with any allowed symbol except for `:`.
- Constructors as operators must start with `:`.

Additionally, functions follow the `lowerCamelCase` style and types follow the `UpperCamelCase` style.

Lets consider Types / typeclasses / Functions / variables as identifiers. Now lets define `Haskell` identifiers using the `EBNF`:

```
<H_ID> ::= <head> {<tail>}
<head> ::= <upper> | <lower>
<tail> ::= <upper> | <lower> | <special> | <digits> | '
<lower> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s
<upper> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S
<digits> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  (0-9) is accepted
```

What if we want to use `BNF`:

```
Functions and Variables  # <H_ID> ::= <head>[<tail>]
                           <head> ::= <lower>
                           <tail> ::= <lower> | <upper>
                           ...
```

Exercise: Consider how we define the variable above, and try to define the following token:

- Unsigned integers with leading zeros allowed (Such as 0, 0001, 1, 200, 0220 etc.)

Make sure you check the `Help` page when you staring answering the questions. You may find lots of useful information. Here is the link:
`http://www.cas.mcmaster.ca/~franek/courses/cs3mi3/help/help.cgi`
But you need to sign-in to see the information.

4 Language Definition

Let's jump to the language we started to define last week. Here is what we did two weeks ago to refresh your mind:

```
t ::=                               Term
        true                        Constant True
        false                       Constant False
        if t then t else t          Conditional Expression
        0                           Zero
        succ t                      Successor
        pred t                      Predecessor
        iszero t                    Zero Test
```

The only thing we do not have here is a strategy to write this language as a text file such that an interpreter is able to read it and then evaluate it for us. I am going to use the same strategy introduce in this file for tokens:

```
EOI = 0 # Enf of input
T = 1 # True
F = 2 # False
Z = 3 # Zero
Succ = 4 # Successor
Pred = 5 # Predecessor
IsZero = 6 # Zero Test
IfThenElse = 7  # Conditional Expression
```

Here is my grammar:

```
expersion   : term | termTail
termTail    : Succ term | Pred term | isZero term | ifThenElse term term term
term        : T | F | Z
```

Now it is more clear. Right!

Let's see how we could recursively read an expression and evaluate it in `Haskell`:

```
data Term =
    T
  | F
  | Z
  | Succ term
  | Pred term
  | IsZero term
  | IfThenElse term term term
```

Here are some expressions defined based on the above languge:

```
expr1 :: Term
expr1 = Succ (Succ (Succ Z))

expr2 :: Term
expr2 = ifThenElse T Z (Suc Z)
```