Programming in Haskell – Project Assignment

UNIZG FER, 2017/2018

Handed out: December 22 2018. Due: January 18 2018

1 Introduction

This year you have two projects to choose from. You can earn maximum of 30 points for each project and you will have to present your project in person to teaching assistants.

In the first project you will have to implement an interpreter for simple programming language to get 15 points. In this part you will have instructions guiding you through the process, but to earn full 30 points you will have to implement a parser for our little programming language and you can implement it in any way you want as long it is implemented in Haskell.

Second project is more "practical" and open-ended in nature and it entails writing a manga reader web application that will scrape manga from some popular manga web site of your choosing and present it in a nice clean format without all the ads and popups on a local web server. You are free to implement it how ever you want as long as it fulfills its purpose.

2 SImPL (Project 01)

Interpreters

An interpreter is a program that takes another program as an input and evaluates it. Many modern languages such as Java, JavaScript, Python, and PHP are interpreted. Although interpreted languages are generally slower than compiled ones such as Haskell, OCaml, and C, they offer many advantages such as platform independence.

Haskell's type system makes it an excellent *metalanguage*. A metalanguage is a language that is used to reason about a different language. In this assignment, you will implement an interpreter for a simple imperative language in Haskell.

Meet SImPL

SImPL is the **S**imple **Im**perative **P**rogramming **L**anguage that you will be interpreting. This language is comprised of seven kinds of statements: assignments, increments, if statements, while loops, for loops, sequences, and skips. The only kinds of values that exists in SImPL are integers. Integers can be represented in expressions as literals `(ie -1, 0, 5)`, variables, or the result of a computation involving two sub-expressions. An example SImPL program that calculates the value of `B` to the power of `E` is:

```
Out := 1;
for (I := 0; I < E; I++) {
    Out := Out * B
}
```

At the end of the program's execution, the value of `B` to the power `E` is stored in the variable `Out`. This SImPL program is nicely readable by humans, however is not so easy for Haskell to work with. In Haskell, we will represent SImPL programs using *Abstract Syntax Trees* or ASTs. An AST is just a Haskell datatype that represents the structure of a piece of code. The datatypes that we will be working with in this assignment are defined in project files that you can download from the repository. The SImPL program above could be represented in Haskell as follows:

```
Sequence
    (Assign "Out" (Val 1))
    (For (Assign "I" (Val 0))
        (Op (Var "I") Lt (Var "E"))
        (Incr "I")
        (Assign (Var "Out") (Op (Var "Out") Times (Var "B"))))
```

Before you continue, make sure you understand the correspondence between the SImPL program and its representation in Haskell. All of this assignment will be based on working with and manipulating the AST.

Part 01

Before we can start evaluating Expressions and Statements we need some way to store and look up the state of a variable. We define a `State` to be a function of type `String -> Int`. This makes it very easy to look up the value of a variable; to look up the value of `"A"` in state, we simply call `state "A"`. Whenever we assign a variable, we want to update the program `State`. Implement the following function:

```
extend :: State -> String -> Int -> State
```

Hint: You can use the input State as a black box for variables other than the one you are assigning.

Example:

```
let st' = extend st "A" 5
in st' "A" == 5
```

In addition to the ability to extend States, we need to have an empty State that we can use to evaluate programs where no variables are assigned. In the empty State, it might make sense to throw an exception when you try to access a variable that has not been defined. Unfortunately, SImPL has no notion of exceptions *(SImPL has no type errors either; everything is an integer, so type checking isn't necessary!)*. For this reason, we will define the empty State to set all variables to 0. Define the empty State:

```
empty :: State
```

Example:

```
empty "A" == 0
```

Part 02

We are now ready to evaluate expressions! This can be implemented as a fairly straightforward recursive function. The value that an `Expression` evaluates to depends on the state of the variables that appear in the `Expression`. For this reason, a `State` must be provided along with the `Expression` that is being evaluated. Implement the function:

```
evalE :: State -> Expression -> Int
```

Note: some of the binary operators (Bops) in SImPL are `Int -> Int -> Int` functions, but others have type `Int -> Int -> Bool`. Since SImPL only has integer types, we need to modify these functions to returning `Ints`. We will do so by returning `0` in place of `False` and `1` in place of `True`.

Example:

```
evalE empty (Val 5) == 5
```

Example:

```
evalE empty (Op (Val 1) Eql (Val 2)) == 0
```

Part 03

It turns out that SImPL isn't so simple after all. The syntax of SImPL contains some repetition. For example, `Incrs` are just special cases of `Assigns`. Syntax like this makes it easier for programmers to reason about code, however it makes the internal representation of the language more complicated. This is called *Syntactic Sugar*. SImPL is a little too sweet for my taste, so before we evaluate SImPL `Statements` we are going to desugar them.

A `DietStatement` is a new datatype that we will use to represent desugared SImPL `Statements`. You may notice that `DietStatements` are very similar to `Statements`. In fact, the `DietStatement` type is the same as the `Statement` type with two constructors removed: `Incr` and `For`. This is because `Incr` is Syntactic Sugar for an assignment and `For` is Syntactic Sugar for a while loop.

For Loop in SImPL

```
for (A := 0; A < N; A++) {...}
```

For Loop represented as an AST

```
For (Assign "A" (Val 0)) (Op (Var "A") Lt (Var "N")) (Incr "A") (...)
```

Explanation

1. Initialization: Executed once at the very beginning
2. Loop Condition: Expression dictating whether or not the body of the loop should be executed.
3. Update: Executed every iteration after the body of the loop

Now, implement the following function:

```
desugar :: Statement -> DietStatement
```

Example:

```
desugar (Incr "A") == DAssign "A" (Op (Var "A") Plus (Val 1))
```

Part 04

In this exercise, you will write a function that evaluates desugared `Statements`. Unlike `Expressions`, `Statements` do not evaluate to a single value. Instead, they perform actions that mutate the initial State. Since we cannot actually mutate the State in Haskell, we will return a new `State` that is the result of evaluating the program. Implement the function:

```
evalSimple :: State -> DietStatement -> State
```

Note: `If` statements and `While` loops both represent conditionals as an `Expression`. Since `Expressions` in SImPL are integers, we will consider `0` to be `False` and every other value to be `True`.

Example:

```
let s = evalSimple empty (DAssign "A" (Val 10))
in s "A" == 10
```

We also want to be able to run programs that have not already been desugared. Implement the function:

```
run :: State -> Statement -> State
```

Note: `run` should be defined in terms of `desugar` and `evalSimple`. It should not be implemented from scratch.

Running Programs

Congratulations! You have written an interpreter for a simple imperative language. But what good is an interpreter without any programs to test it on? We have provided three programs to test your code on: *factorial*, *square root*, and *fibonacci.*

You can use these programs to test your interpreter. In each case, the input is assumed to be assigned to the variable `"In"` and the output is assign to `"Out"`. So, for example, if you wanted to run the `factorial` program with input `4`, you should run the program with a `State` that binds the variable `"In"` to `4`.

Example:

```
let s = run (extend empty "In" 4) factorial
in s "Out" == 24
```

Part 05

This part is for everyone who wants to win full 30 points for the project and it requires a bit more research and effort but it should be very interesting.

To complete our little interpreter, it would be nice to have a parser which can convert our imagined SImPL syntax from simple string into AST.

For that purpose you can use `parsec` library, but I first recommend you to read this very informative tutorial on implementing a simple parser and than using actual production ready parser library like aforementioned `parsec`.

3 Manga Reader (Project 02)

For this project you will have to make a web scraper that will scrape manga from your favorite manga source (or you can use this one if you don't have a personal favorite) and present it in nice and clean way without all the ads and noise.

Idea is to have a local web server that has 3 types of pages / views. One page type should list all available titles, another page should list all available chapters of a specific title and a third page type should show page within a certain chapter with "back" and "forward" links that allow user to flip through pages.

Here some examples of pages that you need to scrape data from:

1. Title list
2. List of chapters from a specific title
3. Page within a chapter with option to go back and forth

You have the freedom to do this project any way you like as long as it presents only the relevant information (don't just save the page on the disk and display it back).

Here are some recommended libraries that you might find useful:

1. **wreq** - Nice library for making HTTP requests and retrieving web pages from URLs.
2. **scalpel-core** - Useful for extracting data from HTML (something akin to jQuery / CSS selectors but for Haskell).

3. **servant** - Very elegant library for defining web APIs and servers. You might want to read the tutorial.

Basically you just need to scrape the list of manga titles and display it on the front page of your web application, if user clicks on one of the titles it will lead him to a list of available chapters for that title and than once a user clicks on one of the chapters he will be presented with first page of that chapter and options to navigate through it.

Scraping should happen when user requests the page from your web application since scraping everything in advance would take up a lot of space and time.

If you have any questions or need help with your project than ask on Google groups so that everyone can benefit from your question.