UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME

Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Tel +44(0)1865 283525 Fax +44(0)1865 283531
info@softeng.ox.ac.uk www.softeng.ox.ac.uk

Part-time postgraduate study in software engineering

Functional Programming, FPR

15th – 19th September 2014

ASSIGNMENT

The purpose of this assignment is to test the extent to which you have achieved the learning objectives of the course. As such, your answer must be substantially your own original work. Where material has been quoted, reproduced, or co-authored, you should take care to identify the extent of that material, and the source or co-author.

Your answers to the questions on this assignment should be submitted to:

Software Engineering Programme
Department of Computer Science
Wolfson Building
Parks Road
Oxford OX1 3QD

Alternatively, you may submit using the Software Engineering Programme website — `www.softeng.ox.ac.uk` — following the submission guidelines. The deadline for submission is 12 noon on Tuesday, 4th November 2014. If you have not already returned a signed assignment acceptance form, you must do so before the deadline, or your work may not be considered. The results and comments will be available after the next examiners' meeting, during the week commencing Tuesday, 23rd December 2014.

ANY QUERIES OR REQUESTS FOR CLARIFICATION REGARDING THIS ASSIGNMENT SHOULD, IN THE FIRST INSTANCE, BE DIRECTED TO THE PROGRAMME OFFICE WITHIN THE NEXT TWO WEEKS.

---

1 Introduction

This assignment concerns *turtle graphics*, which is a very simple language for controlling a virtual robot that has a pen. The robot moves around on an unbounded rectangular grid, staying on integer coordinates. It can be facing north, west, south, or east; it can turn by $90°$. When it moves, it moves by distance 1 in the direction it is facing. The robot also has a pen, which can be 'up' or 'down'; if the robot moves while the pen is down, it draws a line from its old position to its new position.

The robot is controlled via a little language of robot programs, which can be represented as follows:

```
data Prog = Move Prog
          | Turn Prog
          | SwapPen Prog
          | Stop
```

The idea is that

- the program *Move x* moves the robot forward one square (in the direction it is currently facing), then continues with program *x*;
- the program *Turn x* turns the robot by $90°$ anticlockwise, then continues with *x*;
- the program *SwapPen x* flips the state of the pen (it puts the pen down if it was currently up, and lifts the pen up if it was currently down), the continues with *x*;
- the program *Stop* does nothing.

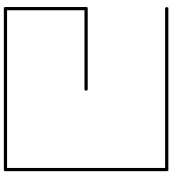
The robot is initially at position $(0, 0)$, facing north, with the pen up. Note that programs are written in the order of execution; so for example the program

```
simple = Turn (Move (SwapPen (Move (SwapPen Stop))))
```

turns, then moves forward, then puts the pen down, then moves again, then lifts the pen up, then stops—it draws a line from $(-1, 0)$ to $(-2, 0)$. The slightly longer program

```
snail = SwapPen (Move (Turn (Move (Turn (Move (Move (Turn
          (Move (Move (Turn (Move (Move Stop))))))))))))
```

draws a square spiral, like this, starting in the centre:



2 Exercises

1. Define a function

   type *PenDown* = *Bool*
   penDown :: *Prog* → *PenDown* → *PenDown*

   so that *penDown x b* represents the final state of the pen for the program *x* starting initially with pen state *b*, where *b* = *True* represents the pen being down and *b* = *False* the pen being up.

2. Define a function

   dir :: *Prog* → *Dir* → *Dir*

   so that *dir x d* represents the final direction the robot is facing after executing program *x*, if it starts facing direction *d*. Here, the directions are the compass points:

   data *Dir* = *North* | *West* | *South* | *East*

3. Define a function

   forwards :: *Dir* → *Pos* → *Pos*

   so that *forwards d p* computes the position one step 'forwards' from position *p* in direction *d*.

4. You can view the datatype *Dir* as the syntactic representation of a very simple domain-specific language (DSL), having only four possible 'programs'. Then the function *forwards* provides a semantics for that DSL of directions. In DSL circles, datatype *Dir* is called a 'deep embedding' of the language of directions. An alternative approach is to use a 'shallow embedding', in which programs in the DSL are represented directly by their semantics:

   type *Dir*2 = *Pos* → *Pos*

   Provide implementations of the four directions using this representation.

   north, *west*, *south*, *east* :: *Dir*2

5. Define a function

   $pos :: Prog \to Pos \to Dir \to Pos$

   so that $pos\ x\ p\ d$ represents the final position of the robot after executing program x, if it starts in position p facing direction d. Here, positions are integer pairs:

   type $Pos = (Integer, Integer)$

   (Hint: you might want to use *forwards*).

6. You can view *pos* as a semantics for the deeply-embedded language *Prog*. Provide an alternative representation $Prog2$ of programs as a shallow embedding, using this semantics.

   type $Prog2 = \ldots$
   $move2, turn2, pen2 :: Prog2 \to Prog2$
   $stop2 :: Prog2$

7. We gave a shallow embedding $Dir2$ of directions in Exercise 4. But we can't use this to replace the type *Dir* in the rest of our code—in particular, we can't use $Dir2$ in the function *pos*. Why not? How would you provide a shallow embedding of directions that does allow us to write a function like *Pos* (without also using the deep embedding *Dir*)?

8. Define a function

   $plot :: Prog \to Pos \to Dir \to PenDown \to Picture$

   so that $plot\ x\ p\ d\ b$ produces the picture drawn by the robot following program x, from initial position p facing direction d with the pen in state b. Here, a *Picture* is the list of lines the robot draws while the pen is down, where each line segment is given by its two endpoints:

   type $Picture = [LineSegment]$
   type $LineSegment = (Pos, Pos)$

9. Provide another shallow embedding $Prog3$ of robot programs using their *plot* semantics.

   type $Prog3 = \ldots$
   $move3, turn3, swapPen3 :: Prog3 \to Prog3$
   $stop3 :: Prog3$

10. It is inconvenient to have two different shallow embeddings *Prog*2 and *Prog*3 of robot programs, with correspondingly different implementations of the constructors *move*, *turn* etc. Fortunately, it is possible to provide a single generic shallow embedding that encompasses both *Prog*2 and *Prog*3:

$$\begin{aligned}
&\textbf{type } \mathit{ProgS}\ a = (a \to a, a \to a, a \to a, a) \to a\\
&\mathit{move}, \mathit{turn}, \mathit{swapPen} :: \mathit{ProgS}\ a \to \mathit{ProgS}\ a\\
&\mathit{stop} :: \mathit{ProgS}\ a\\
&\mathit{move}\ x \quad = \lambda(m,t,p,s) \to m\ (x\ (m,t,p,s))\\
&\mathit{turn}\ x \quad = \lambda(m,t,p,s) \to t\ (x\ (m,t,p,s))\\
&\mathit{swapPen}\ x = \lambda(m,t,p,s) \to p\ (x\ (m,t,p,s))\\
&\mathit{stop} \quad = \lambda(m,t,p,s) \to s
\end{aligned}$$

For example,

$$\mathit{turn}\ (\mathit{move}\ (\mathit{swapPen}\ (\mathit{move}\ (\mathit{swapPen}\ \mathit{stop})))) :: \mathit{ProgS}\ a$$

represents the robot program from the introduction. Show how to extract both the *pos* semantics and the *plot* semantics from such a program. (This is a little tricky, but there's a short answer.)

11. Write a function to convert the *Picture* produced by the *plot* semantics into Scalable Vector Graphics (SVG), which you can then view in an SVG viewer such as Google Chrome. I used my answer to this to produce the following SVG rendition of the plot of the *snail* program:

```
<svg width="204" height="204" viewBox="-102,-102,204,204"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
<g transform="scale(1,-1)" stroke-width="4" stroke-linecap="round"
   stroke="black" fill="none">
<line x1="0" y1="0" x2="0" y2="100"/>
<line x1="0" y1="100" x2="-100" y2="100"/>
<line x1="-100" y1="100" x2="-100" y2="0"/>
<line x1="-100" y1="0" x2="-100" y2="-100"/>
<line x1="-100" y1="-100" x2="0" y2="-100"/>
<line x1="0" y1="-100" x2="100" y2="-100"/>
<line x1="100" y1="-100" x2="100" y2="0"/>
<line x1="100" y1="0" x2="100" y2="100"/>
</g>
</svg>
```

which I then included as the picture at the start of this document. Hopefully, the format of SVG is self-explanatory from this example; but if you need more

guidance, the specification is at `http://www.w3.org/TR/SVG/`. I suggest that you define a simple datatype of content-free XML:

> **data** *XML* = *Element String* [*Attr*] [*XML*]
> **type** *Attr* = (*String*, *String*)

and then define a function to generate *XML* and a way of printing it:

> *svg* :: *Picture* → *XML*
> **instance** *Show XML* **where** ...

Then you can produce the SVG file above by

> *writeFile* `"snail.svg"` (*show* (*svg* (*plot snail* (0, 0) *North False*)))

You may find this helpful in answering the other questions.

3 Guidance

The purpose of the assignment is for you to demonstrate your understanding of functional programming. In principle, you can do that without solving all of the questions above; and there is no need for you to avail yourself of any of the hints, if you can see a better way to do it. But if you're not sure, I advise you to follow the steps above.

You may be able to find partial solutions to the problems on the web. I recommend that you don't use them, or at least, don't rely on them: they are often of dubious quality, they are likely to implement slightly different specifications, they typically won't help your critical review, and they won't help you learn about functional programming. Whatever you do, do make sure you *make clear the source and the extent* of any derivative material.

Submit your answers as a *single* PDF file, formatted using LaTeX, Word or some other text processor of your choice. I want to see the code and commentary interleaved. Additionally, submit the solutions in the form of a literate script, a plain-text LHS file (or several files, if you want to use modules). Also submit some test data, integrated into the literate script(s), along with a short explanation of how to invoke the program on the data.

Finally, please structure the PDF file so that there is a cover sheet which contains only your name, the subject and date, and a note of the total number of pages. Do not put any answer material on the cover sheet; begin your answer on a fresh page. Avoid putting your name on any page except the cover page. Please number the pages and sections.

4 Assessment criteria

The assignment is intended to measure the following, in order of decreasing importance:

- Have you understood the fundamental tools of functional programming: algebraic data types, pattern matching, higher-order functions, parametric polymorphism, and lazy evaluation?
- Have you demonstrated an ability to apply these fundamental tools and accompanying techniques to a particular case study?
- Do you have the ability to present clear arguments supporting design decisions and discussing trade-offs, concisely and precisely?
- How fluent is your expression in Haskell, and how elegant is the resulting code?