4 Compiler Construction (TGG)

Suppose that we are to implement a compiler for the following simple, strongly-typed language with types t, expressions e, and programs p.

$$
\begin{array}{rcll}
t & ::= & \textbf{int} & \\
 & | & t * t & \text{(product type)} \\
 & & & \\
e & ::= & n & \text{(integer)} \\
 & | & \textbf{?} & \text{(read integer input by user)} \\
 & | & e + e & \text{(addition)} \\
 & | & e - e & \text{(subtraction)} \\
 & | & (e, e) & \text{(pair)} \\
 & | & \textbf{fst}\ e & \text{(first projection)} \\
 & | & \textbf{snd}\ e & \text{(second projection)} \\
 & | & f(e) & \text{function application} \\
 & | & \textbf{let}\ x : t = e\ \textbf{in}\ e\ \textbf{end} & \text{(let binding)} \\
 & & & \\
p & ::= & e & \\
 & | & \textbf{fun}\ f(x : t) : t = e\ \textbf{;}\ p & \text{(function definition, recursion allowed)}
\end{array}
$$

In the above x and f range over identifiers. For example, here is a simple program:

```
fun swap (p : int * int) : int * int = (snd p, fst p) ;

fun swizzle (p : int * (int * int)) : (int * int) * int =
     (swap (snd p), fst p) ;

swizzle (?, (?, ?))
```

You are asked to implement this language on a stack machine **that has no heap**. All stack entries are simple words (integers or pointers). Hint: consider using type information.

(a) Describe how your compiler will use the stack to implement function calls and returns. Describe any auxiliary pointers that you might need. Is there anything about the language above that makes this especially easy? [5 marks]

(b) Describe how you allocate space on the stack for a value of type t. [5 marks]

(c) Describe how your compiler will implement expressions of the form $(e_1,\ e_2)$. Explain how the order of evaluation (left-to-right, or right-to-left) impacts your choices. [5 marks]

(d) Describe how your compiler will implement expressions of the form **fst** e and **snd** e. [5 marks]