The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 4 MODULE, SPRING SEMESTER 2012–2013

MATHEMATICAL FOUNDATIONS OF PROGRAMMING

ANSWERS

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer QUESTION ONE and THREE other questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Note: ANSWERS

Question 1 (Compulsory)

(a) See appendix A for the grammars and operational semantics relevant to this question. (You do not need the typing rules for this question.) Use the operational semantics to evaluate the following term until a value is obtained:

```
if (if (iszero (pred 0))
       then (iszero (succ 0))
       else (iszero (pred 0)))
   then (succ 0)
   else 0
```

The validity of *the first two* evaluation steps should be proved by applying the rules of the operational semantics. For the remaining steps, just show the sequence of terms in the right order. (8)

Answer: *Straightforward, will only sketch the answer. The sequence of evaluation steps is as follows:*

```
if (if (iszero (pred 0)) then (iszero (succ 0)) else (iszero (pred 0)))
   then (succ 0)
   else 0
⟶ if (if (iszero 0) then (iszero (succ 0)) else (iszero (pred 0)))
        then (succ 0)
        else 0
⟶ if (if true then (iszero (succ 0)) else (iszero (pred 0)))
        then (succ 0)
        else 0
⟶ if (iszero (succ 0)) then (succ 0) else 0
⟶ if false then (succ 0) else 0
⟶ 0
```

For full marks, the first two steps should be proved using the rules of the operational semantics. As an example, the first step should be

proved as follows:

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred 0} \longrightarrow \texttt{0}}\ \text{E-PREDZERO}}{\texttt{iszero (pred 0)} \longrightarrow \texttt{iszero 0}}\ \text{E-ISZERO}}{\begin{array}{l}\texttt{if (iszero (pred 0)) then (iszero (succ 0)) else (iszero (pred 0))} \\ \longrightarrow \texttt{if (iszero 0) then (iszero (succ 0)) else (iszero (pred 0))}\end{array}}\ \text{E-IF}$$

$$\dfrac{\begin{array}{l}\texttt{if (if (iszero (pred 0)) then (iszero (succ 0)) else (iszero (pred 0)))} \\ \quad \texttt{then (succ 0)} \\ \quad \texttt{else 0} \\ \longrightarrow \texttt{if (if (iszero 0) then (iszero (succ 0)) else (iszero (pred 0)))} \\ \qquad \texttt{then (succ 0)} \\ \qquad \texttt{else 0}\end{array}}{\text{E-IF}}$$

(b) Give λ-terms satisfying the following descriptions:

(i) a *closed* term containing at least one β-redex; (2)

(ii) an *open* term containing at least two λ-abstractions; (2)

(iii) a term containing at least two *bound* and two *free* occurrences of *distinct* variables, clearly indicating which variable occurrences are free and which are bound. (3)

Answer: *For example:*

- $(\lambda x.x)\ (\lambda x.x)$
- $\lambda x.\lambda y.z$
- $\lambda x.\lambda y.\underline{x}\ \underline{y}\ \boxed{u}\ \boxed{v}$
  where the bound variable occurrences have been underlined and the free ones have been enclosed in a box.

(c) Given the definitions:

$$\begin{aligned} \mathrm{I} &\equiv \lambda x.x \\ \mathrm{S} &\equiv \lambda x.\lambda y.\lambda z.x\ z\ (y\ z) \\ \mathrm{K} &\equiv \lambda x.\lambda y.x \\ \omega &\equiv \lambda x.x\ x \end{aligned}$$

reduce the following λ-calculus term to normal form if possible:

$$\mathrm{I\ S\ K\ K}\ (\omega\ \omega)$$

Show each unfolding step (expansion of a definition) and each individual β-reduction step. If it is not possible to reach a normal form, explain why. (5)

Answer: *Normal-order reduction ensures a normal form will be found, if it exists. So carry out normal-order reduction (outermost, left-most redex first) until either a normal form has been reached, or it becomes evident that reduction isn't going to yield a normal form:*

$$\begin{aligned} &\mathrm{I\ S\ K\ K}\ (\omega\ \omega) \\ &= (\lambda x.x)\ \mathrm{S\ K\ K}\ (\omega\ \omega) \\ &\underset{\beta}{\rightarrow} \mathrm{S\ K\ K}\ (\omega\ \omega) \\ &= (\lambda x.\lambda y.\lambda z.x\ z\ (y\ z))\ \mathrm{K\ K}\ (\omega\ \omega) \\ &\underset{\beta}{\rightarrow} (\lambda y.\lambda z.\mathrm{K}\ z\ (y\ z))\ \mathrm{K}\ (\omega\ \omega) \\ &\underset{\beta}{\rightarrow} (\lambda z.\mathrm{K}\ z\ (\mathrm{K}\ z))\ (\omega\ \omega) \\ &\underset{\beta}{\rightarrow} \mathrm{K}\ (\omega\ \omega)\ (\mathrm{K}\ (\omega\ \omega)) \\ &= (\lambda x.\lambda y.x)\ (\omega\ \omega)\ (\mathrm{K}\ (\omega\ \omega)) \\ &\underset{\beta}{\rightarrow} (\lambda y.(\omega\ \omega))\ (\mathrm{K}\ (\omega\ \omega)) \\ &\underset{\beta}{\rightarrow} \omega\ \omega \\ &= (\lambda x.x\ x)\ \omega \\ &\underset{\beta}{\rightarrow} \omega\ \omega \end{aligned}$$

Thus, $\omega\ \omega$ reduces to the very same term, $\omega\ \omega$, by normal-order reduction; it is thus clear that no normal form can be reached as it will always be possible to carry out another β-reduction.

(d) See appendix A for the grammars and typing rules for this question. Prove that the following term is well-typed:

$$\texttt{if}\ (\texttt{iszero}\ (\texttt{pred}\ \texttt{0}))\ \texttt{then}\ (\texttt{succ}\ \texttt{0})\ \texttt{else}\ \texttt{0}$$

Your proof should be structured as a proof tree. (5)

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{0} : \texttt{Nat}}\ \text{T-ZERO}}{\texttt{pred}\ \texttt{0} : \texttt{Nat}}\ \text{T-PRED}}{\texttt{iszero}\ (\texttt{pred}\ \texttt{0}) : \texttt{Bool}}\ \text{T-ISZERO} \qquad \dfrac{\dfrac{}{\texttt{0} : \texttt{Nat}}\ \text{T-ZERO}}{\texttt{succ}\ \texttt{0} : \texttt{Nat}}\ \text{T-SUCC} \qquad \dfrac{}{\texttt{0} : \texttt{Nat}}\ \text{T-ZERO}}{\texttt{if}\ (\texttt{iszero}\ (\texttt{pred}\ \texttt{0}))\ \texttt{then}\ (\texttt{succ}\ \texttt{0})\ \texttt{else}\ \texttt{0} : \texttt{Nat}}\ \text{T-IF}$$

Question 2
This question concerns the pure, untyped λ-calculus, enriched with λ-definable constants where indicated.

(a) Consider the following definition of a function for summing the integer values carried by the leaves and nodes in a binary tree:

```
sumtree(t) = if isleaf(t) then
                 leafval(t)
             else
                 sumtree(left(t))
                 + nodeval(t)
                 + sumtree(right(t))
```

Show how to translate this function into the pure λ-calculus, explaining the key ideas of the translation. You may assume normal-order evaluation and use the following λ-definable constants:

IF	Usual three-argument conditional; first argument, the condition, assumed to be of Boolean type
ISLEAF	Test for leaf
LEAFVAL	Value carried by a leaf
NODEVAL	Value carried by a node
LEFT	Left subtree
RIGHT	Right subtree
PLUS	Numerical addition (two arguments)

Any other constants needed in your translation have to be defined and explained.

(10)

Answer: *The idea of the translation is to abstract out the recursively called function as an extra, first, argument. The resulting residual function is thus a function that will* return *a function that computes the sum of the values in a tree if applied to a function that computes this sum. Or, in other words, the desired function is the* fixed point *of the residual function. If we introduce a fixed-point combinator* Y *for computing fixed points, the above function* `fib` *can be translated into the* λ*-calculus as follows, where* SUMTREE$'$ *is the residual and* SUMTREE *the desired translation:*

$$\begin{aligned}\text{SUMTREE}' &\equiv \lambda f.\lambda t.\text{IF } (\text{ISLEAF } t)\\ &\qquad\qquad (\text{LEAFVAL } t)\\ &\qquad\qquad (\text{PLUS } (f\ (\text{LEFT } t))\ (\text{PLUS } (\text{NODEVAL } t)\ (f\ (\text{RIGHT } t))))\\ \text{SUMTRE} &\equiv \text{Y SUMTREE}'\end{aligned}$$

Y *is the call-by-name fixed point combinator and works under call-by-name or normal-order evaluation. It has the following definition:*

$$\mathrm{Y} \equiv \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

(An answer along the lines above suffices for full marks; the following are additional explanations/suggests possible variations of a good answer.) What makes this work is the fact that Y *satisfies the fixed point equation*

$$\mathrm{Y\ F} = \mathrm{F\ (Y\ F)}$$

In other words, Y *ensures that a function* F *to which* Y *is applied gets applied to the fixed point of the function* F *itself, which is computed by applying* Y *to* F*.*

Another way to understand this is that Y *enables a recursive definition to be unfolded on demand, thus simulating the jump to a specific code sequence, which is how function calls and recursion usually are handled, by inlining a copy of the called function at the call site.*

(b) A *discriminated union* is a way to form the union of two sets of values in such a way that it for each element in the union is possible to tell from which of the two sets this element initially came. One way to realise this idea is through two injection functions, *left* and *right*, that takes an element from one or the other set into the discriminated union, and a function *case* that applies one of two given functions to an element from the discriminated union depending on whether that element was injected using *left* or *right*. Using $\uplus$ to denote discriminated union, the types for these functions for some specific sets A, B, and C would be:

$$\begin{aligned} \mathit{left} &: A \to A \uplus B \\ \mathit{right} &: B \to A \uplus B \\ \mathit{case} &: (A \to C) \to (B \to C) \to (A \uplus B) \to C \end{aligned}$$

Further, for arbitrary $a : A$, $b : B$, $f : A \to C$, and $g : B \to C$, the following algebraic laws must hold:

$$\begin{aligned} \mathit{case}\ f\ g\ (\mathit{left}\ a) &= f\ a \\ \mathit{case}\ f\ g\ (\mathit{right}\ b) &= g\ b \end{aligned}$$

Provide λ-calculus definitions for these three functions that meet the above specification. Explain your construction, and use only the basic calculus; if you wish to use some auxiliary constants, you must define them first. Further, demonstrate the validity of your definition by proving that one of the two algebraic laws holds. (10)

Answer: *One possible encoding is as follows. The idea is essentially the same as the Church encoding of the Booleans, except that we associate a "payload" with each element, essentially forming one-field records:*

$$\begin{aligned} \text{LEFT} &\equiv \lambda a.\lambda f.\lambda g.f\ a \\ \text{RIGHT} &\equiv \lambda b.\lambda f.\lambda g.g\ b \\ \text{CASE} &\equiv \lambda f.\lambda g.\lambda c.c\ f\ g \end{aligned}$$

That this encoding satisifies the first law can be proved as follows:

$$\begin{aligned} &\text{CASE}\ f\ g\ (\text{LEFT}\ a) \\ &= (\lambda f.\lambda g.\lambda c.c\ f\ g)\ f\ g\ (\text{LEFT}\ a) \\ &\xrightarrow[\beta]{*} (\text{LEFT}\ a)\ f\ g \\ &= ((\lambda a.\lambda f.\lambda g.f\ a)\ a)\ f\ g \\ &\xrightarrow[\beta]{} (\lambda f.\lambda g.f\ a)\ f\ g \\ &\xrightarrow[\beta]{*} f\ a \end{aligned}$$

Another possibility is to make use of the Church encodings of Booleans and pairs. In essence the idea is to tag elements by pairing them either with true or false, enabling a choice of which function to be applied to be done in the encoding of case:

$$\begin{aligned} \text{LEFT} &\equiv \text{PAIR T} \\ \text{RIGHT} &\equiv \text{PAIR F} \\ \text{CASE} &\equiv \lambda f.\lambda g.\lambda c.\text{IF (FST } c) \text{ } (f \text{ (SND } c)) \text{ } (g \text{ (SND } c)) \end{aligned}$$

Note that for full marks, the definitions of all of these constants must also be given.

(c) Explain the problem of *name capture*. Illustrate your answer with an example. (5)

Answer: *Name capture occurs when a term with free variables is naively substituted for a variable in a context where one or more of the variables that are free in the term are bound. For example, if the application*

$$(\lambda x.\lambda y.x)\ y$$

were to be reduced naively by substituting y for x in $\lambda y.x$, then the result would be $\lambda y.y$ which is wrong. The initially free y has been confused with a different bound variable that just happened to have the same name.

Question 3
Consider the following expression language:

$$
\begin{array}{lll}
e & \to & \textit{expressions:} \\
 & n & \textit{natural numbers, } n \in \mathbb{N} \\
 & \mid\ x & \textit{variables, } x \in \text{Name} \\
 & \mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \textit{conditional}
\end{array}
$$

where Name is the set of variable names. The types are given by the following grammar:

$$
\begin{array}{lll}
t & \to & \textit{types:} \\
 & \texttt{Nat} & \textit{natural numbers} \\
 & \mid\ \texttt{Bool} & \textit{Booleans}
\end{array}
$$

The ternary relation $\Gamma \vdash e : t$ says that expression e has type t in the typing context Γ and it is defined by the following typing rules:

$$\Gamma \vdash n : \texttt{Nat} \qquad \text{(T-NAT)}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t} \qquad \text{(T-COND)}$$

A typing context, Γ in the rules above, is a comma-separated sequence of variable-name and type pairs, like

$$\texttt{x} : \texttt{Nat},\ \texttt{y} : \texttt{Bool},\ \texttt{z} : \texttt{Nat}$$

or empty, denoted $\emptyset$. Typing contexts are extended on the right, e.g. Γ, $\texttt{z} : \texttt{Nat}$, the membership predicate is denoted by $\in$, and lookup is from right to left, ensuring recent bindings hides earlier ones.

(a) Use the typing rules given above to formally prove that the expression

$$\texttt{if b then 3 else x}$$

has type `Nat` in the typing context

$$\Gamma_1 = \texttt{b} : \texttt{Bool},\ \texttt{x} : \texttt{Nat}$$

(5)

Answer:

$$\dfrac{\dfrac{\texttt{b} : \texttt{Bool} \in \Gamma_1}{\Gamma_1 \vdash \texttt{b} : \texttt{Bool}}\ \text{T-VAR} \quad \dfrac{}{\Gamma_1 \vdash \texttt{3} : \texttt{Nat}}\ \text{T-NAT} \quad \dfrac{\texttt{x} : \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{x} : \texttt{Nat}}\ \text{T-VAR}}{\Gamma_1 \vdash \texttt{if b then 3 else x} : \texttt{Nat}}\ \text{T-COND}$$

(b) The expression language defined above is to be extended with lists of elements of some specific type (i.e. homogeneous lists) as follows:

e	$\rightarrow$		*expressions:*
	...	...	...
	|	`null`	*empty list*
	|	`cons` e e	*list construction*
	|	`head` e	*head of list*
	|	`tail` e	*tail of list*
	|	`isnull` e	*test for empty list*
t	$\rightarrow$		*types:*
	...	...	...
	|	`List` t	*list of elements of type* t

Provide a typing rule for each of the new expression constructs in the same style as the existing rules. The constant `null` has type `List` t for any type t; `cons` takes a value of a type t and a list of elements of the same type t, i.e. `List` t, as arguments, and returns a list of type `List` t; and `head`, `tail`, and `isnull` all take a list of elements of a type t, `List` t, as argument and return, respectively, a result of type t, `List` t, and `Bool`. (5)

Answer:

$$\Gamma \vdash \texttt{null} : \texttt{List}\ t \qquad \text{(T-NULL)}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : \texttt{List}\ t}{\Gamma \vdash \texttt{cons}\ e_1\ e_2 : \texttt{List}\ t} \qquad \text{(T-CONS)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{List}\ t}{\Gamma \vdash \texttt{head}\ e_1 : t} \qquad \text{(T-HEAD)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{List}\ t}{\Gamma \vdash \texttt{tail}\ e_1 : \texttt{List}\ t} \qquad \text{(T-TAIL)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{List}\ t}{\Gamma \vdash \texttt{isnull}\ e_1 : \texttt{Bool}} \qquad \text{(T-ISNULL)}$$

(c) Extend the language further with function abstraction and application as follows:

e	$\rightarrow$		*expressions:*
	$\ldots$	$\ldots$	$\ldots$
	$\mid$	$\lambda x{:}t\ .\ e$	*function abstraction*
	$\mid$	$e\ e$	*function application*
t	$\rightarrow$		*types:*
	$\ldots$	$\ldots$	$\ldots$
	$\mid$	$t \rightarrow t$	*function (arrow) type*

For example, the function abstraction λ`x:Nat . x` is the identity function on `Nat`, the natural numbers, and (λ`x:Nat . x`) `42` is the application of this function to the number `42`.

Provide a typing rule for each of the new expression constructs, in the same style as the existing rules, reflecting the standard notions of function abstraction and function application. (6)

Answer:

$$\frac{\Gamma,\ x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x{:}t_1\ .\ e) : t_1 \rightarrow t_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash e_1 : t_2 \rightarrow t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t_1} \quad \text{(T-APP)}$$

(d) Use the extended set of typing rules to formally prove that the expression

$$\texttt{tail}\ ((\lambda \texttt{f}{:}\texttt{Nat} \to \texttt{Bool}\ .\ \texttt{cons}\ (\texttt{f}\ 7)\ \texttt{null})\ \texttt{even})$$

has type `List Bool` in the typing context

$$\Gamma_2 = \texttt{even} : \texttt{Nat} \to \texttt{Bool} \tag{9}$$

Answer: Let $g = \lambda \texttt{f} : \texttt{Nat} \to \texttt{Bool}\ .\ \texttt{cons}\ (\texttt{f}\ 7)\ \texttt{null}$. Let us first determine the type of the expression g:

$$\dfrac{\dfrac{\dfrac{\texttt{f} : \texttt{Nat} \to \texttt{Bool} \in \Gamma_3}{\Gamma_3 \vdash \texttt{f} : \texttt{Nat} \to \texttt{Bool}}\ \text{T-VAR} \quad \dfrac{}{\Gamma_3 \vdash 7 : \texttt{Nat}}\ \text{T-NAT}}{\Gamma_3 \vdash \texttt{f}\ 7 : \texttt{Bool}}\ \text{T-APP} \quad \dfrac{}{\Gamma_3 \vdash \texttt{null} : \texttt{List Bool}}\ \text{T-NULL}}{\Gamma_3 = \Gamma_2,\ \texttt{f} : \texttt{Nat} \to \texttt{Bool} \vdash \texttt{cons}\ (\texttt{f}\ 7)\ \texttt{null} : \texttt{List Bool}}\ \text{T-CONS}$$
$$\dfrac{}{\Gamma_2 \vdash g : (\texttt{Nat} \to \texttt{Bool}) \to \texttt{List Bool}}\ \text{T-ABS}$$

We then check the type of the application g `even`

$$\dfrac{\dfrac{}{\Gamma_2 \vdash g : (\texttt{Nat} \to \texttt{Bool}) \to \texttt{List Bool}}\ \text{above} \quad \dfrac{\texttt{even} : \texttt{Nat} \to \texttt{Bool} \in \Gamma_2}{\Gamma_2 \vdash \texttt{even} : \texttt{Nat} \to \texttt{Bool}}\ \text{T-VAR}}{\Gamma_2 \vdash g\ \texttt{even} : \texttt{List Bool}}\ \text{T-APP}$$

Finally, we turn to the complete expression:

$$\dfrac{\dfrac{}{\Gamma_2 \vdash g\ \texttt{even} : \texttt{List Bool}}\ \text{above}}{\Gamma_2 \vdash \texttt{tail}\ \ (g\ \texttt{even}) : \texttt{List Bool}}\ \text{T-HEAD}$$

Question 4

The aim of this question is to give an *operational* semantics for the language in Appendix B. As it is an imperative language, the evaluation relation(s) will have to relate a term c or e along with a store σ to the resulting term after one step of evaluation and, if necessary, an updated store. As described in the informal semantics, there is only one type of value, integers, and the subset of terms representing values is thus just n, the syntactic category of integers. Take a store to be a function that maps a variable name to a value; i.e.

$$\sigma : x \to n$$

Further, use the notation $[x \mapsto n]\sigma$ for "updating" a store; i.e., to denote a store that is like σ except that x is mapped to n.

(a) Give an operational semantics for expressions e. It is permissible to use predicates on the integers as premises in the inference rules. For example, if you need the sum n' of two integers n_1 and n_2 in the conclusion of a rule, use a predicate like $n' = n_1 + n_2$ as a premise, or if a rule should only be applicable if two numbers n_1 and n_2 are not equal, use a premise like $n_1 \neq n_2$. (8)

Answer:

$$\frac{\sigma(x) = n}{x \mid \sigma \longrightarrow n} \quad \text{(E-VAR)}$$

$$\frac{e_1 \mid \sigma \longrightarrow e_1'}{e_1 \texttt{ + } e_2 \mid \sigma \longrightarrow e_1' \texttt{ + } e_2} \quad \text{(E-ADD1)}$$

$$\frac{e_2 \mid \sigma \longrightarrow e_2'}{n_1 \texttt{ + } e_2 \mid \sigma \longrightarrow n_1 \texttt{ + } e_2'} \quad \text{(E-ADD2)}$$

$$\frac{n = n_1 + n_2}{n_1 \texttt{ + } n_2 \mid \sigma \longrightarrow n} \quad \text{(E-ADDN)}$$

$$\frac{e_1 \mid \sigma \longrightarrow e_1'}{e_1 \texttt{ = } e_2 \mid \sigma \longrightarrow e_1' \texttt{ = } e_2} \quad \text{(E-EQ1)}$$

$$\frac{e_2 \mid \sigma \longrightarrow e_2'}{n_1 \texttt{ = } e_2 \mid \sigma \longrightarrow n_1 \texttt{ = } e_2'} \quad \text{(E-EQ2)}$$

$$\frac{n_1 = n_2}{n_1 \texttt{ = } n_2 \mid \sigma \longrightarrow 1} \quad \text{(E-EQEQ)}$$

$$\frac{n_1 \neq n_2}{n_1 \texttt{ = } n_2 \mid \sigma \longrightarrow 0} \quad \text{(E-EQNE)}$$

(b) Give an operational semantics for commands c. (12)

Answer:

$$\frac{c_1 \mid \sigma \longrightarrow c_1' \mid \sigma'}{c_1 \texttt{ ; } c_2 \mid \sigma \longrightarrow c_1' \texttt{ ; } c_2 \mid \sigma'} \quad \text{(E-SEQ)}$$

$$\texttt{skip ; } c_2 \mid \sigma \longrightarrow c_2 \mid \sigma \quad \text{(E-SEQSKIP)}$$

$$\frac{e \mid \sigma \longrightarrow e'}{x \texttt{ := } e \mid \sigma \longrightarrow x \texttt{ := } e' \mid \sigma} \quad \text{(E-ASSIGN)}$$

$$x \texttt{ := } n \mid \sigma \longrightarrow \texttt{skip} \mid [x \mapsto n]\sigma \quad \text{(E-ASSIGNN)}$$

$$\frac{e \mid \sigma \longrightarrow e'}{\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \sigma \longrightarrow \texttt{if } e' \texttt{ then } c_1 \texttt{ else } c_2 \mid \sigma} \quad \text{(E-IF)}$$

$$\frac{n \neq 0}{\texttt{if } n \texttt{ then } c_1 \texttt{ else } c_2 \mid \sigma \longrightarrow c_1 \mid \sigma} \quad \text{(E-IFTRUE)}$$

$$\texttt{if } 0 \texttt{ then } c_1 \texttt{ else } c_2 \mid \sigma \longrightarrow c_2 \mid \sigma \quad \text{(E-IFFALSE)}$$

$$\frac{e_1 \mid \sigma \longrightarrow e_1'}{\texttt{for } x \texttt{ := } e_1 \texttt{ to } e_2 \texttt{ do } c \mid \sigma \longrightarrow \texttt{for } x \texttt{ := } e_1' \texttt{ to } e_2 \texttt{ do } c \mid \sigma} \quad \text{(E-FOR1)}$$

$$\frac{e_2 \mid \sigma \longrightarrow e_2'}{\texttt{for } x \texttt{ := } n_1 \texttt{ to } e_2 \texttt{ do } c \mid \sigma \longrightarrow \texttt{for } x \texttt{ := } n_1 \texttt{ to } e_2' \texttt{ do } c \mid \sigma} \quad \text{(E-FOR2)}$$

$$\frac{n_1 \leq n_2 \quad n_1' = n_1 + 1}{\texttt{for } x \texttt{ := } n_1 \texttt{ to } n_2 \texttt{ do } c \mid \sigma \longrightarrow c \texttt{ ; for } x \texttt{ := } n_1' \texttt{ to } n_2 \texttt{ do } c \mid [x \mapsto n_1]\sigma} \quad \text{(E-FORDO)}$$

$$\frac{n_1 > n_2}{\texttt{for } x \texttt{ := } n_1 \texttt{ to } n_2 \texttt{ do } c \mid \sigma \longrightarrow \texttt{skip} \mid [x \mapsto n_1]\sigma} \quad \text{(E-FORDONE)}$$

(c) Use your rules to formally evaluate the program

```
i := i + 7
```

given an initial store σ_1 such that $\sigma_1(\texttt{i}) = \texttt{3}$. (5)

Answer: *Straightforward, will only sketch the answer. The sequence of evaluation steps is as follows:*

$$\begin{aligned} & \texttt{i := i + 7} \mid \sigma_1 \\ \longrightarrow\ & \texttt{i := 3 + 7} \mid \sigma_1 \\ \longrightarrow\ & \texttt{i := 10} \mid \sigma_1 \\ \longrightarrow\ & \texttt{skip} \mid [\texttt{i} \mapsto \texttt{10}]\sigma_1 \end{aligned}$$

For full marks, the steps should be proved using the rules of the operational semantics. As an example, the first step should be proved as follows:

$$\dfrac{\dfrac{\dfrac{\sigma_1(\texttt{i}) = \texttt{3}}{\texttt{i} \mid \sigma_1 \longrightarrow \texttt{3}}\ \text{E-VAR}}{\texttt{i + 7} \mid \sigma_1 \longrightarrow \texttt{3 + 7}}\ \text{E-ADD1}}{\texttt{i := i + 7} \mid \sigma_1 \longrightarrow \texttt{i := 3 + 7} \mid \sigma_1}\ \text{E-ASSIGN}$$

Question 5

The aim of this question is to give a *denotational* semantics for the language in Appendix B. As it is an imperative language, the notion of a store mapping a variable name to the current value of that variable is needed. There is only one type of value, integers. Thus, take a store σ to be a function of type Σ mapping a variable name (in the syntactic category x) to its current value (in $\mathbb{Z}$):

$$\begin{array}{rcl} \Sigma & = & x \to \mathbb{Z} \\ \sigma & : & \Sigma \end{array}$$

Further, use the notation $[x \mapsto n]\sigma$ for "updating" a store; i.e., to denote a store that is like σ except that x is mapped to n.

(a) Suggest an appropriate type signature for a semantic function $\mathrm{E}[\![\cdot]\!]$ that maps an expression to its meaning. Use the syntactic category e as the type of expressions. Provide a brief explanation of your answer. Note that evaluation of expressions in this language has *no* side effects and is total and terminating. (2)

Answer: *An appropriate type signature for* $\mathrm{E}[\![\cdot]\!]$ *is:*

$$\mathrm{E}[\![\cdot]\!] : e \to (\Sigma \to \mathbb{Z})$$

As evaluation of expressions is total and terminating, there is no need for a bottom element in the range of the semantic function.

(b) Define the semantic function $\mathrm{E}[\![\cdot]\!]$ for expressions. (4)

Answer: *Definition:*

$$\begin{array}{rcl} \mathrm{E}[\![n]\!]\ \sigma & = & n \\ \mathrm{E}[\![x]\!]\ \sigma & = & \sigma\ x \\ \mathrm{E}[\![e_1 \texttt{ + } e_2]\!]\ \sigma & = & \mathrm{E}[\![e_1]\!]\ \sigma\ +\ \mathrm{E}[\![e_2]\!]\ \sigma \\ \mathrm{E}[\![e_1 \texttt{ = } e_2]\!]\ \sigma & = & \begin{cases} 1, & \text{if } \mathrm{E}[\![e_1]\!]\ \sigma\ =\ \mathrm{E}[\![e_2]\!]\ \sigma \\ 0, & \text{otherwise} \end{cases} \end{array}$$

(c) What is a suitable type signature for a semantic function $\mathrm{C}[\![\cdot]\!]$ mapping a command to its meaning? Use the syntactic category c as the type of commands. Provide a brief explanation of your answer. Note that execution of a command in the language as given always terminates; in particular, the body of a `for`-loop is repeated a finite number of times. (4)

Answer: *A suitable type signature is:*

$$\mathrm{C}[\![\cdot]\!] : c \to (\Sigma \to \Sigma)$$

That is, a command is mapped to a state transformer, a function mapping a state represented by a store to a new state. The language does include a loop construct, but the number of iterations is always finite meaning that execution will never diverge (fail to terminate). The state transformer is thus a total function, and its range should therefore not include $\perp$ (no lifting).

(d) Define the semantic function $\mathrm{C}[\![\cdot]\!]$ for commands. (10)

Answer:

$$\begin{aligned}
\mathrm{C}[\![\texttt{skip}]\!]\ \sigma &= \sigma \\
\mathrm{C}[\![x\ \texttt{:=}\ e]\!]\ \sigma &= [x \mapsto \mathrm{E}[\![e]\!]\ \sigma]\sigma \\
\mathrm{C}[\![c_1\ \texttt{;}\ c_2]\!]\ \sigma &= \mathrm{C}[\![c_2]\!]\ (\mathrm{C}[\![c_1]\!]\ \sigma) \\
\mathrm{C}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ \sigma &= \begin{cases} \mathrm{C}[\![c_1]\!]\ \sigma, & \text{if } \mathrm{E}[\![e]\!]\ \sigma \neq 0 \\ \mathrm{C}[\![c_2]\!]\ \sigma, & \text{otherwise} \end{cases} \\
\mathrm{C}[\![\texttt{for}\ x\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ c]\!]\ \sigma &= (\mathrm{fix}_{\mathbb{Z}\to(\Sigma\to\Sigma)}\ g)\ (\mathrm{E}[\![e_1]\!]\ \sigma)\ \sigma \\
\text{where } g = \lambda f.\lambda n.\lambda\sigma_1.&\begin{cases} [x \mapsto n]\sigma_1, & \text{if } n > \mathrm{E}[\![e_2]\!]\ \sigma \\ f\ (n+1)\ (\mathrm{C}[\![c]\!]\ ([x \mapsto n]\sigma_1)), & \text{otherwise} \end{cases}
\end{aligned}$$

Further comments on the solution:

A denotational semantics should always be compositional, i.e. the meaning of the whole (syntactic construct) given by meaning of the parts (subterms). This is not just a matter of style, but is deeply connected to the question of whether denotational semantics actually has any well-defined meaning.

In this particular case, given that the for-loop as defined is guaranteed to terminate, one might argue that a non-compositional "operational-style" definition, where for each iteration the loop is syntactically translated into a sequence of the loop body followed by a loop with one fewer iterations, should be fine as it really just amounts to a finite unfolding of the loop; i.e. a kind of "macro expansion".

However, besides breaking compositionality, it is a very precarious approach. For example, it is very easy to inadvertently violate the specificed semantics by opening up for non-termination by re-evaluating *the loop expressions at each iteration. Note that the specification clearly says that the loop expressions are to be evaluated* before *the loop begins. This is a critical part of ensuring termination. Because otherwise a loop like e.g. the following will not terminate:*

```
y = 1;
for x = 0 to y do
    y := y + 1
```

Or other variations on this theme.

Thus the right approach is to ensure that the loop expressions are only evaluated with respect to the initial *store (note the distinction between* σ *and* σ_1 *above), and then define a suitable semantic function by means of a fixed-point construction.*

(e) How would the *type* of $\mathrm{C}[\![\cdot]\!]$ have to be changed if a `while`-loop (with the conventional imperative semantics) were to be added to the language? Explain your answer, and provide an updated definition of $\mathrm{C}[\![c_1 \texttt{ ; } c_2]\!]$ as an illustration. (5)

Answer: *Adding a conventional* `while`*-loop means that execution of commands now may diverge (fail to terminate). The range (co-domain) of the state transformer that is the meaning of a command must therefore be extended to include* $\perp$*, denoting divergence. The new type signature becomes:*

$$\mathrm{C}[\![\cdot]\!] : c \to (\Sigma \to \Sigma_\perp)$$

When the result of one state transformer is passed to another, the domain of the second state transformer needs to be extended to account for the fact that the result from the first one may be $\perp$*. This can be done by a* source lifting *that extends a function by mapping* $\perp$ *to* $\perp$*. For example, the denotation of sequencing becomes:*

$$\mathrm{C}[\![c_1 \texttt{ ; } c_2]\!]\,\sigma \;=\; (\mathrm{C}[\![c_2]\!]_{\perp\!\perp})\,(\mathrm{C}[\![c_1]\!]\,\sigma)$$

Appendix A

This appendix contains the abstract syntax, operational semantics, and typing rules for the small example language (from Pierce's book *Types and Programming Languages*) that has been discussed in the module.

Abstract Syntax:

t	$\rightarrow$		*terms:*
		`true`	*constant true*
	$\mid$	`false`	*constant false*
	$\mid$	`if` t `then` t `else` t	*conditional*
	$\mid$	`0`	*constant zero*
	$\mid$	`succ` t	*successor*
	$\mid$	`pred` t	*predecessor*
	$\mid$	`iszero` t	*zero test*

Values:

v	$\rightarrow$		*values:*
		`true`	*true value*
	$\mid$	`false`	*false value*
	$\mid$	nv	*numeric value*
nv	$\rightarrow$		*numeric values:*
		`0`	*zero value*
	$\mid$	`succ` nv	*successor value*

Types:

T	$\rightarrow$		*types:*
		`Bool`	*Boolean type*
	$\mid$	`Nat`	*Numeric type*

Operational Semantics: (call-by-value)

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IFTRUE)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IFFALSE)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-IF)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \qquad \text{(E-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PREDZERO)}$$

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \qquad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \qquad \text{(E-PRED)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-ISZEROZERO)}$$

$$\texttt{iszero (succ } nv_1\texttt{)} \longrightarrow \texttt{false} \qquad \text{(E-ISZEROSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \qquad \text{(E-ISZERO)}$$

Typing Rules:

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-IF)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$

Appendix B

This appendix gives the abstract syntax and informal semantics for a small imperative language. In the following, x is some suitable syntactic category of variable names, and n that of integers ($\mathbb{Z}$):

c	$\rightarrow$		*Commands:*
	$\mid$	`skip`	*Do nothing*
	$\mid$	c `;` c	*Sequencing*
	$\mid$	x `:=` e	*Assignment*
	$\mid$	`if` e `then` c `else` c	*Conditional*
	$\mid$	`for` x `:=` e `to` e `do` c	`for`-*loop*
e	$\rightarrow$		*Expressions:*
	$\mid$	n	*Literal integer*
	$\mid$	x	*Variable*
	$\mid$	e `+` e	*Addition*
	$\mid$	e `=` e	*Comparison*

Expressions are interpreted in the conventional way; e.g., `+` is integer addition. The order of evaluation is call-by-value. However, for simplicity, there is only one type of value in the language, integers, and the integer values 0 and 1 are used to represent the Boolean values *false* and *true*, respectively. Thus, `3 = 7` evaluates to 0 and `7 = 7` evaluates to 1. Additionally, when what notionally is a Boolean value is used, any non-zero value is treated as representing *true*. Note that evaluation of expressions in this language has *no* side effects and is total and terminating.

Commands have the conventional, imperative interpretation, along with the following refinement for the `for`-loop:

> The two expressions are first evaluated, the first one to n_1 and the second one to n_2; if $n_1 \leq n_2$, the loop body c is repeated $n_2 - n_1 + 1$ times with x being assigned to consecutive values from n_1 to n_2 prior to each iteration; the value of x after the loop is the greater of n_1 and $n_2 + 1$.

Also, as implied by the description of Boolean values above, the conditional treats any non-zero value as *true*. Note that commands in general *have* side effects.