Assignment 3

Modifying a denotational semantic definition

Goals
To read a denotational definition of a simple imperative language, and to modify it to add a new feature.

Procedure
Take the following abstract syntax definition of a language, and its denotational semantics and augment the syntax and semantics to allow for the multiple assignment statement type found in BCPL, and other languages.

Abstract Syntax:

\[ P ::= S \]
\[ S ::= I = E \mid S_1 ; S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \mid \text{while } B \text{ do } S \text{ end} \]
\[ B ::= I == E \mid I > E \mid I < E \mid B_1 \text{ or } B_2 \mid B_1 \text{ and } B_2 \mid \text{not } B \]
\[ E ::= N \mid I \mid E_1 + E_2 \mid E_1 - E_2 \]

Semantic domains:

\( Z \) (the integers, with addition, subtraction and comparison operations)
\( I \in \text{Id} \) (identifiers)
\( s \in \text{Store: Id} \rightarrow Z \) (update operation \( \text{a} \))
\( B = \{ \text{true, false} \} \) (operations and, or not)

Semantics (valuation functions):

\[ M[N] = n, \text{ where } n \in Z \] (integers)
\[ M[I]s = s(I) \]
\[ M[E_1 + E_2]s = M[E_1]s + M[E_2]s \]
\[ M[E_1 - E_2]s = M[E_1]s - M[E_2]s \]
\[ M[I==E]s = \text{true if } s(I) = M[E]s \text{ else false} \]
\[ M[I > E]s = \text{true if } s(I) > M[E]s \text{ else false} \]
\[ M[I < E]s = \text{true if } s(I) < M[E]s \text{ else false} \]
\[ M[B_1 \text{ or } B_2]s = \text{true if one or both of } M[B_1]s \text{ and } M[B_2]s \text{ is true else false} \]
\[ M[B_1 \text{ and } B_2]s = \text{false if one of } M[B_1]s \text{ and } M[B_2]s \text{ is false else true} \]
\[ M[\text{not } B]s = \text{true if } M[B]s \text{ is false else true} \]
\[ M[I=E]s = s[I \mapsto M[E]s] \]
\[ M[S_1 ; S_2]s = M[S_2](M[S_1]s) \]
\[ M[\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end}]s = M[S_1]s \text{ if } M[B]s = \text{true else } M[S_2]s \]
\[ M[\text{while } B \text{ do } S \text{ end}]s = M[S; \text{while } B \text{ do } S]s \text{ if } M[B]s \text{ is true else } s \]
**Hints**

- Add a new statement type to the abstract syntax to handle any number of identifiers and the same number of expressions. For instance:
  
  \[ x, y, z = 1, 2, 3 \]

  will assign 1 to \( x \), 2 to \( y \) and 3 to \( z \). Can the syntax specify that the number of expressions should equal the number of identifiers?

- Add a meaning function or functions that map your new syntax to the appropriate functional forms. You will have to solve the problem of processing the sequence of identifiers (and expressions) in a functional manner. Typically this is done with two operations on a list – head which returns the first item in the list, and tail which returns everything else except the first item. Recursion then handles the list one item at a time. Don’t try to use an iterative solution since the functional world of lambda calculus doesn’t have any iteration.

**Grading**

Total 30 points. Partial credit will be available for answers that are along the right lines.

**Due Date**

September 30th. by 5:00pm.