Final Examination Solutions

The AVAJ language has the following abstract syntax:

P ∈ Program
K ∈ Class-declaration
T ∈ Type-structure
D ∈ Declaration
C ∈ Command
E ∈ Expression
I ∈ Identifier
N ∈ Numeral

P ::= K | KP
K ::= class I { D }
T ::= nat | bool | I
D ::= D₁; D₂ | var I:T | method I₁:T (I₂:T) C
C ::= C₁; C₂ | I ::= E | return E | if E then C₁ else C₂
E ::= E₁+E₂ | E₁*E₂ |E₁-E₂ | I | N | new I | E.I | E₁(E₂) | …

**PROBLEM 1** (20 Points)
Show the abstract syntax tree for this program:

class foo {
  var x:foo;
  
  method test:foo (x:foo)
  return x
}

\[
\begin{array}{l}
\text{P} \\
\downarrow \\
\text{K} \\
\downarrow \\
\text{class I} \{ D \} \\
\downarrow \\
\text{foo D ;} \\
\downarrow \\
\text{var I : T} \\
\downarrow \\
\text{method I : T ( I : T )} \\
\downarrow \\
\text{C} \\
\end{array}
\]

\[
\begin{array}{l}
X \\
\downarrow \\
\text{test} \\
\downarrow \\
I \\
\downarrow \\
x \\
\downarrow \\
\text{return} \\
\downarrow \\
E \\
\end{array}
\]

\[
\begin{array}{l}
\text{foo} \\
\downarrow \\
\text{foo} \\
\downarrow \\
\text{foo} \\
\end{array}
\]
PROBLEM 2 (15 Points)
Find and give an example of each type of ambiguity in the AVAJ language syntax, or explain why none exist.

Declarations
The rule D ::= D₁; D₂ leads to ambiguity when parsing D₁ D₂ D because the declarations can associate to the right or left.

Commands
The rule C ::= C₁; C₂ leads to ambiguity when parsing C₁ C₂ C because the commands can associate to the right or left. The rule if E then C₁ else C₂ leads to ambiguity when parsing if E then C₁ else C₂ C because the else clause might consist of one or two commands.

Expressions
There is ambiguity in every expression rule that involves subexpressions. When parsing E op E op E, it is possible to associate to the left or right, where op can be +, *, -, or ().

PROBLEM 3 (10 Points)
Assuming that AVAJ supports recursive methods, write a factorial method declaration in AVAJ.

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot \text{fact}(n - 1) & \text{otherwise}
\end{cases}
\]

method fact:nat (n:nat)
if (n = 0)
    return 1
else
    return n * fact(n - 1)

PROBLEM 4 (25 Points)
Handling the return command correctly poses some difficulty. One way to do so is to define semantic domain Poststore as:

Domain \( p \in \text{Poststore} = \text{OK} + \text{ReturnValue} + \text{Err} \)
where \( \text{OK} + \text{Err} = \text{Store} \) and \( \text{ReturnValue} = \text{Expressible} \times \text{Store} \)

The idea is that whenever a return command is encountered, a ReturnValue is returned. Commands detect this value and keep returning it until it is detected by a method invocation.

What are the semantic equations for
\[
C[[C₁; C₂]]
C[[\text{return } E]]
E[[E₁(E₂)]]
\]

Explain any assumptions you make.

C[[C₁; C₂]] must detect the ReturnValue type from C₁ and return it, otherwise it behaves as before. Using the language of Schmidt figure 7.2, this is easily accomplished by modifying the check function.

\[
\text{check} : (\text{Store} \to \text{Poststore}_\perp) \to (\text{Poststore}_\perp \to \text{Poststore}_\perp)
\]

\[
\text{check} f = \lambda p . \text{cases } p \text{ of}
\]
isOK(s) → (f s)
isReturnValue(r) → p
[[isErr(s) → p

C[[return E]] must convert the Expressible from E into a ReturnValue. Recall that commands have valuation function signature

C: Command → Environment → Store → Poststore⊥

and that expressions are

E: Expression → Environment → Store → (Expressible × Poststore)⊥

So all we need to do is “retag” the result of evaluating expression E.

\[ C[[\text{return } E]] = \lambda e.\lambda s.\text{let } (v, p) = E[[E]] e s \text{ in } \]
\[ \text{inReturnValue}(v, p) \]

E[[E₁(E₂)]] acts like a function call.

\[ E[[E₁(E₂)]] = \lambda e.\lambda s.\text{let } (v, p) = E[[E₁]] e s \text{ in } \]
\[ \text{cases } p \text{ of } \]
\[ \text{isOK}(s') \rightarrow \text{cases } v \text{ of } \]
\[ \text{isMethod}(m) \rightarrow (m(E[[E₂]] e s') s') \]
\[ \text{inErrvalue}(\text{end}) \]
\[ \text{inErrvalue}(\text{end}) \]

**PROBLEM 5** (15 Points)
Show that \( C[[\text{return } E; C]] = C[[\text{return } E]] \)

\[ C[[\text{return } E; C]] = \lambda e.(\text{check}(C[[C]] e)) \circ (C[[\text{return } E]] e) \]
\[ = \lambda e.(\text{check}(C[[C]] e)) \circ \lambda s.\text{let } (v, p) = E[[E]] e s \text{ in } \text{inReturnValue}(v, p) \]
\[ = \lambda e.\lambda p.\text{cases } p \text{ of } \ldots \text{isReturnValue}(r) \rightarrow p \ldots \circ \lambda s.\text{let } (v, p) = E[[E]] e s \text{ in } \text{inReturnValue}(v, p) \]
\[ = \lambda e.\lambda s.\text{let } (v, p) = E[[E]] e s \text{ in } \text{inReturnValue}(v, p) \]
\[ = C[[\text{return } E]] \]

**PROBLEM 6** (15 Points)
As it now stands, all fields and methods in an AVAJ class are public. That is, they may be accessed from other classes. It is proposed to add an optional modifier **private** to the language such that private fields and methods may not be accessed from outside the class.

Describe how environments may be used to maintain private and public access. You do not need to supply semantic equations. Hint: Consider keeping 2 environments for each class declaration.
The main idea is that class definitions use two environments, a public environment and a private environment. Only the public environment is “exported” for use by other classes. That is, when a method call or field access from another class is made to an instance of this class, only the public environment is used to lookup the identifier corresponding to the method or field.

When a method call or field access is made from within the current class, both the public and private environments are used.

There are a number of complicated details that would need to be addressed in a complete denotation. For example, one should assure that method and field identifiers are distinct in both the public and private environments. Also, in order to support recursive public and private methods, methods should be considered to be fixedpoints over both public and private environments.