McMaster University
Department of Computing and Software
Dr. W. Kahl

SFWR ENG 3E03
Exercise Sheet 10

Design and Selection of Programming Languages

13 November 2005

This will be discussed in class

Exercise 10.1 — Correctness Proof for Bisection — 65% of Midterm 3, 2005

Written in the simple imperative programming language for which axiomatic semantics rules are available on the distributed rule sheet, the following program fragment implements the bisection method for finding a root of the function $f : \mathbb{R} \to \mathbb{R}$, where e, m, s, x, u are all variables of type $\mathbb{R}$:

```
s := signum(f(x)) ;
while u > x + e do
      m := u − (u − x)/2 ;
      if signum(f(m)) ≡ s then x := m else u := m fi
od
```

signum is the usual signum (sign) function:

$$signum(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0 \end{cases}$$

(a) ≈59% Using the ***global assumptions*** that f is a ***continuous function*** and that $e > 0$ (i.e., no need to carry this explicitly through every proof step; just ***mention where you use it***), **prove partial correctness** of the above program with respect to

- the **precondition** $u > x \wedge signum(f(x)) \neq signum(f(u))$
- and the **postcondition** $\exists\, w \in [x, x + e] \bullet f(w) = 0$.

  (The bracket notation $[x, x + e]$ here denotes the **interval** containing exactly those real numbers z with $x \leq z$ and $z \leq x + e$.)

> - **Hint:** Induce the invariant from the **precondition** in this case!
>   (I.e., **not** from the postcondition as in most previous examples.)
> - **Use the big sheet for this proof!**

(b) ≈6% (***independent from the solution of (a)!***)

The postcondition given in (a) asserts that the resulting x is an approximation to **an arbitrary** root. However, the root produced by this program fragment will actually be between the **starting values** of x and u. Provide a precondition-postcondition specification that includes this fact.

Exercise 10.2 — Abstract Syntax in Haskell — 35% of Midterm 3, 2005

The following abstract syntax datatypes for a variant of Jay adds procedure declarations and procedure calls to the language we have seen so far, and removes while loops.

Procedure parameters are implicitly declared to be of type int.

```
-- Datatypes for expressions etc.
-- are essentially as before:
type Variable = String
data Type = IntType | BoolType
data Expression
   = Num Integer
   | Var Variable
   | BinOp Expression Operator Expression
data Operator = Op String
data Program
   = Prog [Declaration] [Statement]
```

```
-- New and changed datatypes:
type ProcName = String

data Declaration
   = VarDecl Variable Type
   | ProcDecl ProcName [Variable] Program

data Statement
   = Assignment Variable Expression
   | ProcCall ProcName [Expression]
   | Conditional Expression [Statement] [Statement]
```

(a) ≈10% With mostly C-like concrete syntax, the following is a program of this language variant:

```
int x;
bool b;
void addToX(a) {
  x = x + a;
}

{                   // top-level block would be body of  main()  in C
 x = 15;
 addToX(27);
}
```

Define a Haskell value *prog1* :: *Program* to represent the abstract syntax tree of this program.

(b) ≈7% This abstract syntax datatype allows the representation of programs that cannot be directly transliterated into ANSI C. Which feature does it introduce that ANSI C does not have?

Explain, and give an example for a program using this feature (preferably in C-like concrete syntax as the example given in (a)).

(c) ≈18% Implement the Haskell function *tailCallsProg* :: *Program* → [(*ProcName*, *ProcName*)] such that for a program *p* and two procedure names *f* and *g*, the pair (*f*, *g*) is in *tailCallsProg p* exactly if in the declaration of procedure *f*, there is a tail call to procedure *g* (i.e., in one branch of the body of This abstract syntax datatype *f*, a call to *g* is the last executed statement).

(Define auxiliary functions as necessary.)