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

SFWR ENG 3E03
Exercise Sheet 9

Design and Selection of Programming Languages

8 November 2005

This exercise sheet has four pages and four exercises.

Exercise 9.1 — Partial Correctness

For each of the following Hoare triples, determine whether it holds; if yes, prove it using the rules of axiomatic semantics, and if no, prove a counter-example using the rules of operational semantics (**you may abbreviate each *expression* evaluation into a single step**).

(a) $\{x \geq -5\}\ z := 5 - x\ \{z \leq 11\}$

(b) $\{x \geq -5\}\ z := 5 - x\ \{z \leq 11 \wedge x \geq -7\}$

(c) $\{x \geq -5\}\ z := 5 - x\ \mathbf{;}\ x := x + z\ \{z \leq 11 \wedge x = 2\}$

(d) $\{z = abs(x)\}$ **if** $x \geq 0$ **then** $z := -z$ **fi** $\{xz = -x^2\}$

(e) $\{z = 0\}$ **if** $x = 0$ **then** $w :=$ True **else** $z := 1/x$ **fi** $\{\neg w \rightarrow xz = 1\}$

Exercise 9.2 — Partial Correctness Proof — 50% of Midterm 4, 2003

Consider the following program in a language providing a Java-like printing statement:

```
s := 1 ;
r := 0 ;
while s ≤ n do
      r := r + 1 ;
      s := s + 2 * r + 1 ;
      println(r + "  " + s)
od
```

(a) What is the output of this program for $n = 30$?

(b) Give an equation relating the values of r and s in each **println** statement.

(c) For this program **without the println** statement, **prove partial correctness** with respect to the **precondition** $\{\ n \geq 0\ \}$ and the **postcondition** $\{\ r^2 \leq n\ \wedge\ n < (r+1)^2\ \}$.

Hint: ***Use the equation*** from (b) **as part of the invariant!**

Exercise 9.3 — Operational and Axiomatic Semantics: Base-2-Logarithm — 30% of Final 2005

(a) $\boxed{\approx 3\%}$ Define what the phrase "Statement **S** is partially correct with respect to the precondition Q and the postcondition R" means in terms of operational semantics.

Written in the simple imperative programming language for which operational and axiomatic semantics rules are available on the distributed rule sheet, let the following program fragment **P**, with `int` variables i, k, and n, be given in two variants, one with simultaneous assignments, and one without::

$(k,\ i) := (1,\ -1)$;
while $k \leq n$ **do**
  $(k,\ i) := (k * 2,\ i + 1)$
od;
	$k := 1$;
	$i := -1$;
	while $k \leq n$ **do**
	  $k := k * 2$;
	  $i := i + 1$
	od;

This program fragment is intended to calculate the base-2-logarithm of n, as expressed by the following post-condition *Post*:

$$2^i \leq n \quad \wedge \quad n < 2^{i+1}$$

(b) $\boxed{\approx 6\%}$ Provide a derivation in the operational semantics that shows that the precondition "*True*" is too weak, i.e., that the program **P** (in the **right** variant **without** simultaneous assignments) is **not** partially correct with respect to the precondition "*True*" and the above postcondition. (You may omit the details for expression evaluation.)

Explain why your derivation shows that.

(c) $\boxed{\approx 4\%}$ Identify the weakest precondition for which the program fragment P can be proven partially correct with respect to the above postcondition. **Explain!**

(*True* is "weaker" than every other condition Q, since $Q \Rightarrow True$ holds for every Q.)

(d) $\boxed{\approx 18\%}$ Formally prove that P is partially correct with respect to the precondition you stated in (c) and the above postcondition:

$$Post \quad :\Leftrightarrow \quad 2^i \leq n \quad \wedge \quad n < 2^{i+1}$$

Choose whether you consider
- ☐ only the version with simultaneous assignments (left), or
- ☐ only the version without (right).

Include all intermediate steps of the proof, and **show** also the **implications** used.

Exercise 9.4 — Imperative Programs with Nested Scopes — 20% of Final 2004 (adapted)

For this question, the **abstract** syntax of statements of SImPL-0.0 will be replaced by the following definitions that allow declarations to occur anywhere:

```
data Program   = MkProgram  Block

type Block     = [ Statement ]

data Statement
        = Decl        Variable    Type
        | MkBlock     Block
        | Assignment  Variable    Expression
        | Conditional Expression Statement Statement
        | Loop        Expression Statement
```

(a) Change the SImPL lexer and parser to the following **concrete** statement syntax where variable declarations are introduced by the keyword **var**, and blocks are delimited by the keywords **begin** and **end** instead of by braces; we also introduce the keyword **skip** for the empty block:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \textbf{skip} \\
 & | & \textbf{var}\ \mathit{Type}\ \mathit{Id}\ \textbf{;} \\
 & | & \mathit{Id}\ \textbf{:=}\ \mathit{Expr}\ \textbf{;} \\
 & | & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt} \\
 & | & \textbf{while}\ \mathit{Expr}\ \textbf{do}\ \mathit{Stmt} \\
 & | & \textbf{begin}\ \mathit{Stmt}^{*}\ \textbf{end}
\end{array}
$$

For **notation**, we use the following conventions:

- "$A \longrightarrow B$" denotes the set of *total functions* from the set A to the set B.
- "$A \nrightarrow B$" denotes the set of *partial functions* from the set A to the set B.
- "$[\ A\ ]$" denotes the set of finite sequences (lists) of elements from the set A.

We choose the following **basic semantic domains**:

Val	$= \mathit{Bool} + \mathit{Num}$	**values**	**data** *Value* = *ValBool Bool* | *ValInt Int*
SVal	$= \mathit{Val} + \{\Omega\}$	**storable values**	**type** *SVal* = *Maybe Value*
Env	$= \mathit{Id} \nrightarrow \mathit{SVal}$	**environments**	**type** *Env* = *Map Variable SVal*
State	$= [\ \mathit{Env}\ ]$	**states**	**type** *State* = [*Env*]

We denote the elements of *Val* by True, False, 0, 1, 2, …

We denote the elements of *SVal* by Ω, True, False, 0, 1, 2, …

From an *operational point of view*, a program **starts** executing in a **state** consisting of a **single, empty environment**.

At any time, the first element of the environment list that is the state is called the **current environment**.

A variable declaration "**var** *ty v*" produces a run-time error if v is in the domain of the current environment, and otherwise enters v associated with Ω (marking *uninitialised* variables) into the current environment.

When execution moves past a **begin**, a new, empty current environment is added to the state. When execution moves past the matching **end**, this current environment is dropped.

A reference to a variable (in assignments or expressions) named v refers to the first environment in the current state that has v in its domain of definition.

Assignments and references to non-existing variables give rise to run-time errors. In expressions, variable references to uninitialised variables (i.e., associated with Ω), also give rise to run-time errors.

(b) Besides **each** line of the following statement sequence, write down the *State* that is reached ***after*** execution of the respective line (it has been written down for you in the first few lines):

skip ;	$[\{\}]$
var int k **;**	$[\{k \mapsto \Omega\}]$
begin	$[\{\}, \{k \mapsto \Omega\}]$
var int q **;**	$[\{q \mapsto \Omega\}, \{k \mapsto \Omega\}]$
k **:=** 9 **;**	$[\{q \mapsto \Omega\}, \{k \mapsto 9\}]$
var int r **;**	
q **:=** $5 * k$ **;**	
var int k **;**	
begin	
var int r **;**	
r **:=** $q - 8$ **;**	
k **:=** $r + 5$	
end	
q **:=** $q + k$ **;**	
end	

In the following, you are asked to define the operational semantics of selected syntactic constructs. You may want to provide derivation rules for operational semantics assertions, similar to those presented in the lecture. In any case, also modify the definitions of the Haskell interpreter functions in the module *SImPLEval*:

```
evalExpr   :: Expression → State → Maybe Value
interpStmt :: Statement  → State → Maybe State
```

(c) Define the statement semantics of **begin** S **end** for an arbitrary statement $S : \mathit{Stmt}$.

(In Haskell, define *interpStmt* (*BeginEnd stmt*) for an arbitrary *stmt* :: *Statement*.)

(d) Define the statement semantics of **var** ty v for an arbitrary type ty and an arbitrary variable name v.

(In Haskell, define *interpStmt* (*Decl v ty*) for arbitrary *ty* :: *Type* and *v* :: *Variable*.)

(e) Define the remaining cases of statement semantics. Also adapt the expression semantics in *SImPLEval* *.hs* to this setting.