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 \land x \geq -7 \}

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

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

(e) \{ z = 0 \} \text{ if } x = 0 \text{ then } w := \text{True} \text{ else } z := 1/x \text{ 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:

\[
\begin{align*}
n &:= 1 \\
r &:= 0 \\
\text{while } s \leq n \text{ do} \\
& \quad r := r + 1 \\
& \quad s := s + 2 \cdot r + 1 \\
& \quad \text{println}(r + " " + s) \\
\text{od}
\end{align*}
\]

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

(b) Give an equation relating the values of \( r \) and \( s \) in each \texttt{println} statement.

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

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) 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:

\[
\begin{align*}
&\text{( } k, \ i \text{ ) := ( } 1, \ -1 \text{ ) ; } \\
&\text{while } k \leq n \text{ do} \\
&\quad ( k, \ i ) := ( k \times 2, \ i + 1 ) \\
&\text{od ;}
\end{align*}
\]

This program fragment is intended to calculate the base-2-logarithm of \( n \), as expressed by the following post-condition \( \text{Post} \):

\[
2^i \leq n \land n < 2^{i+1}
\]

(b) 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) 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) Formally prove that P is partially correct with respect to the precondition you stated in (c) and the above postcondition:

\[
\text{Post} : \iff 2^i \leq n \land n < 2^{i+1}
\]

Choose whether you consider \( \square \) only the version with simultaneous assignments (left), or \( \square \) 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:

```plaintext
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:

```plaintext
Stmt ::= skip
    | var Type Id ;
    | Id ::= Expr ;
    | if Expr then Stmt else Stmt
    | while Expr do Stmt
    | begin Stmt end
```

For notation, we use the following conventions:

- “\(A \rightarrow B\)” denotes the set of total functions from the set \(A\) to the set \(B\).
- “\(A \rightarrow\)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:

```plaintext
Val     = Bool + Num     values     data Value  = ValBool Bool | Vallnt Int
SVal    = Val + {Ω}      storable values  type SVal  = Maybe Value
Env     = Id → SVal      environments  type Env   = Map Variable SVal
State   = [ Env ]        states       type State = [ Env ]
```

We denote the elements of \(Val\) by \(True, False, 0, 1, 2, \ldots\)
We denote the elements of \(SVal\) by \(Ω, True, False, 0, 1, 2, \ldots\)

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 \( \Omega \)), 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):

<table>
<thead>
<tr>
<th>skip ;</th>
<th>([ { } ])</th>
</tr>
</thead>
<tbody>
<tr>
<td>var int ( k );</td>
<td>([ { k \rightarrow \Omega } ])</td>
</tr>
<tr>
<td>begin</td>
<td>([ { }, { k \rightarrow \Omega } ])</td>
</tr>
<tr>
<td>var int ( q );</td>
<td>([ { q \rightarrow \Omega }, { k \rightarrow \Omega } ])</td>
</tr>
<tr>
<td>( k := 9 );</td>
<td>([ { q \rightarrow \Omega }, { k \rightarrow 9 } ])</td>
</tr>
<tr>
<td>var int ( r );</td>
<td></td>
</tr>
<tr>
<td>( q := 5 \times k );</td>
<td></td>
</tr>
<tr>
<td>var int ( k );</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>var int ( r );</td>
<td></td>
</tr>
<tr>
<td>( r := q - 8 );</td>
<td></td>
</tr>
<tr>
<td>( k := r + 5 )</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>( q := q + k );</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

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 \( \rightarrow \) State \( \rightarrow \) Maybe Value
- **interpStmt** :: Statement \( \rightarrow \) State \( \rightarrow \) Maybe State

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

   (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.