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

SFWR ENG 3E03
Exercise Sheet 10

Design and Selection of Programming Languages

23 November 2005

Exercise 10 — Imperative Programs with Nested Scopes — 25% of Final 2004

We consider a simple imperative programming language with variable declarations (**var**) and nested blocks (delimited by **begin** and **end**).

The **abstract syntax** of this programming language is the following:

Stmt	::=	**skip**	*Expr*	::=	*Id*
	|	**var** *Type Id*		|	*Num*
	|	*Id* **:=** *Expr*		|	*Bool*
	|	*Stmt* **;** *Stmt*		|	*Expr Op Expr*
	|	**if** *Expr* **then** *Stmt* **else** *Stmt* **fi**			
	|	**while** *Expr* **do** *Stmt* **od**	*Op*	::=	+ | − | ∗ | / | ≤ | ≥ | < | >
	|	**begin** *Stmt* **end**			

(a) For each of the following, indicate whether it denotes a **syntactically correct** program (i.e., statement) of this language, and if not, ***visibly mark the problem***:

1. True: ☐ False: ☐ **if** $a < b$ **then** b **:=** a **else var int** a **fi ; while** $a > 0$ **do** a **:=** $a - b$ **od**
2. True: ☐ False: ☐ **if** $a < b$ **then begin else end fi ; while** $a > 0$ **do** a **:=** $a - b$ **od**
3. True: ☐ False: ☐ **if** $a < b$ **then** b **:=** a **else skip fi ;**
   while $a > 0$ **do**
   begin var int a **;** a **:=** $a - 1$ **end od**
4. True: ☐ False: ☐ **var int** a **;** a **:=** 10 **;**
   while $a > 0$ **do var int** b **;** b **:=** $b * a$ **;** a **:=** $a - 1$ **end od**
5. True: ☐ False: ☐ **if** $a < b$ **then** b **:=** a
   else begin a **:=** $a - b$ **;** b **:=** $b + 1$ **; end fi**

(b) Draw the **abstract syntax tree** for **one of the longest** syntactically correct statements from (a).

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

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

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

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.

(c) Besides **each** line of the following program, 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 have the choice** whether you want to do this by providing derivation rules for operational semantics, similar to those on the extra rules sheet, or whether you want to do this by providing the relevant alternatives of definitions of the Haskell interpreter functions

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

In the latter case, the following is the data type for the abstract syntax category *Stmt* of statements:

```
data Statement  = Skip
                | Decl        Type   Variable
                | Seq         Statement  Statement
                | Assignment  Variable   Expression
                | Conditional Expression Statement Statement
                | Loop        Expression Statement
                | BeginEnd    Statement
```

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

(If you choose Haskell, define *interpStmt* (*BeginEnd stmt*) for an arbitrary *stmt* :: *Statement*.)

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

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

(f) $\boxed{\text{mostly } \textbf{new}}$ Define the remaining cases of statement semantics. Also adapt the expression semantics from `Expr.hs` on the course page to this setting.