

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
      / var Type Id
      / Id := Expr
      / Stmt ; Stmt
      / if Expr then Stmt else Stmt fi
      / while Expr do Stmt od
      / begin Stmt end

Expr ::= Id
      / Num
      / Bool
      / Expr Op Expr

Op ::= + | - | * | / | ≤ | ≥ | < | >
```

(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*:

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

Solution Hints

No fixed way to organise the tree prescribed.

For below:

```
module Nested where
```

```
import Expr
import qualified Data.Map as Map hiding ( Map )
import Data.Map ( Map )
```

```
data Value = ValBool Bool | ValInt Int
type SVal = Maybe Value
type Env = Map Variable SVal
type State = [ Env ]
```

```
type Variable = String
type Expression = Expr
```

type $Type = String$

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

- “ $A \rightarrow B$ ” denotes the set of *total functions* from the set A to the set B .
- “ $A \mapsto B$ ” 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 $Value = ValBool Bool \mid ValInt Int$
$SVal$	$= Val + \{\Omega\}$	storable values	type $SVal = Maybe Value$
Env	$= Id \mapsto SVal$	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 ;	[{ $q \mapsto \Omega$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
$q := 5 * k$;	[{ $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
var int k ;	[{ $k \mapsto \Omega$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
begin	[{ }, { $k \mapsto \Omega$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
var int r ;	[{ $r \mapsto \Omega$ }, { $k \mapsto \Omega$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
$r := q - 8$;	[{ $r \mapsto 37$ }, { $k \mapsto \Omega$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
$k := r + 5$	[{ $r \mapsto 37$ }, { $k \mapsto 42$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
end ;	[{ $k \mapsto 42$, $q \mapsto 45$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
$q := q + k$	[{ $k \mapsto 42$, $q \mapsto 87$, $r \mapsto \Omega$ }, { $k \mapsto 9$ }]
end	[{ $k \mapsto 9$ }]

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

$evalExpr :: Expression \rightarrow State \rightarrow Maybe Value$

$interpStmt :: Statement \rightarrow State \rightarrow Maybe State$

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 $stmt :: Statement$.)

Solution Hints

$interpStmt (BeginEnd stmt) = \lambda state \rightarrow$

case $interpStmt stmt (Map.empty : state)$ **of**

$Nothing \rightarrow Nothing$

$Just (_ : state') \rightarrow Just state'$

(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 $v :: Variable$.)

Solution Hints

$interpStmt (Decl ty v) = \lambda state \rightarrow$

case $state$ **of**

$[] \rightarrow Nothing$

$(env : envs) \rightarrow$ **case** $Map.lookup v env$ **of**

$Nothing \rightarrow Just (Map.insert v Nothing env : envs)$

$Just sval \rightarrow Nothing$ $--$ redefinition!

(f) mostly new Define the remaining cases of statement semantics. Also adapt the expression semantics from `Expr.hs` on the course page to this setting.

Solution Hints

$interpStmt (Seq s1 s2) = \lambda state \rightarrow$

case $interpStmt s1 state$ **of**

$Nothing \rightarrow Nothing$

$Just state' \rightarrow interpStmt s2 state'$

$interpStmt (Assignment v e) = \lambda state \rightarrow$

case $evalExpr e state$ **of**

$Nothing \rightarrow Nothing$

$Just val \rightarrow updVar v val state$

where

$updVar v val (env : envs) =$ **case** $Map.lookup v env$ **of**

$-- Nothing \rightarrow fmap (env :) (updVar v val envs)$

$Nothing \rightarrow$ **case** $updVar v val envs$ **of**

$Nothing \rightarrow Nothing$

$Just envs' \rightarrow Just (env : envs')$

$Just env' \rightarrow Just (Map.insert v (Just val) env : envs)$

$interpStmt (Conditional c s1 s2) = \lambda state \rightarrow$

case $evalExpr c state$ **of**

$Just (ValBool b) \rightarrow interpStmt (if b then s1 else s2) state$

$_ \rightarrow Nothing$

$interpStmt s \equiv (Loop c body) = interpStmt (Conditional c (Seq body s) Skip)$

```
lookupVar :: Variable → State → Maybe SVal
lookupVar v [] = Nothing
lookupVar v (env : envs) = case Map.lookup v env of
  Nothing → lookupVar v envs
  Just val → Just val

evalExpr (Var v) state = maybe Nothing id (lookupVar v state)
evalExpr (Num k) state = Just (Vallnt (fromInteger k))
evalExpr (Bin e1 (MkOp op) e2) state = case (evalExpr e1 state, evalExpr e2 state) of
  (Just val1, Just val2) → evalOp op val1 val2
  _ → Nothing

evalOp "+" (Vallnt m) (Vallnt n) = Just (Vallnt (m + n))
evalOp "-" (Vallnt m) (Vallnt n) = Just (Vallnt (m - n))
evalOp "*" (Vallnt m) (Vallnt n) = Just (Vallnt (m * n))
evalOp "/" (Vallnt m) (Vallnt 0) = Nothing
evalOp "/" (Vallnt m) (Vallnt n) = Just (Vallnt (m `div` n))
-- etc...
evalOp _ _ _ = Nothing
```
