

```

--- The version with display added so we can do error diagnostics
--- note that code only needed to change in five easy to spot places
--- Everything else is EXACTLY as it was in the previous not so good
version
--- and I have deleted all reference to the now unnecessary value Error

-- This is closely based on Robert D. Cameron's code
-- www.cs.sfu.ca/~cameron

-- 1. Syntactic and Semantic Domains of TINY
--     Syntactic Domains are Ide, Exp and Cmd

type Ide = String

data Exp = Zero | One | TT | FF |
          Read | I Ide | Not Exp |
          Equal Exp Exp | Plus Exp Exp
          deriving Show

data Cmd = Assign Ide Exp | Output Exp |
          IfThenElse Exp Cmd Cmd |
          WhileDo Exp Cmd |
          Seq Cmd Cmd
          deriving Show

-- Semantic Domains

data Value = Numeric Integer | Boolean Bool | ERROR
            deriving Show

data MemVal = Stored Value | Unbound
             deriving Show

-- Here we use a pair to represent memory. The first element is a list of
-- variables that are currently defined or bound to, The second is, as
-- originally, a function to represent memory. Looking up
-- an identifier is thus still function application, but we can also tell
-- what identifiers are defined, which allows a better diagnostic facility
-- than before.

-- ***Change here*****
type Memory = ([Ide], Ide -> MemVal)

type Input = [Value]

type Output = [Value]

type State = (Memory, Input, Output)

--

```

```

-- 2. Signatures of semantic functions.
--
-- First, we need auxiliary types to represent the possible
-- results of expression evaluation or command execution.
--

data ExpVal = OK Value State

data CmdVal = OKc State | Errorc

exp_semantics :: Exp -> State -> ExpVal

cmd_semantics :: Cmd -> State -> CmdVal

display :: Memory -> String
-- ***Change here***
display (d,f) = foldr (++) " " (map (\i -> i ++ " = " ++ (show (f i)) ++ ";") d)

--
-- 3. Semantic Equations defining the semantic functions
-- Haskell's equational definition is similar but not
-- identical to the equational style used in the mathematical
-- semantics.

exp_semantics Zero s = OK (Numeric 0) s

exp_semantics One s = OK (Numeric 1) s

exp_semantics TT s = OK (Boolean True) s

exp_semantics FF s = OK (Boolean False) s

exp_semantics Read (m, [], o) = error (display m ++ "Input: " ++ "[]" ++
"Output: " ++ show o)

exp_semantics Read (m, (i:is), o) = OK i (m, is, o)

-- *** Changes here ***
exp_semantics (I ident) ((d,m), i, o) =
  case (m ident) of
    Stored v  -> OK v ((d,m), i, o)
    Unbound  -> error (display (d,m) ++ "Input: " ++ show i ++ " " ++
"Output: " ++ show o)

exp_semantics (Not exp) s =
  case (exp_semantics exp s) of
    OK (Boolean True) s1 -> OK (Boolean False) s1
    OK (Boolean False) s1 -> OK (Boolean True) s1

```

```

    OK (Numeric v) s1 -> error (display m ++ "Input: " ++ show i ++ " " ++
"Output: " ++ show o)
        where (m,i,o) = s

```

```

exp_semantics (Equal exp1 exp2) s =
  case (exp_semantics exp1 s) of
    OK (Numeric v1) s1 -> case (exp_semantics exp2 s1) of
      OK (Numeric v2) s2 -> OK (Boolean (v1 == v2))
s2
      OK (Boolean v2) s2 -> OK (Boolean False) s2
    OK (Boolean v1) s1 -> case (exp_semantics exp2 s1) of
      OK (Boolean v2) s2 -> OK (Boolean (v1 == v2))
s2
      OK (Numeric v2) s2 -> OK (Boolean False) s2

```

```

exp_semantics (Plus exp1 exp2) s =
  case (exp_semantics exp1 s) of
    OK (Numeric v1) s1 -> case (exp_semantics exp2 s1) of
      OK (Numeric v2) s2 -> OK (Numeric (v1 + v2)) s2
      OK (Boolean v2) s2 -> error (display m ++
"Input: " ++ show i ++ " " ++ "Output: " ++ show o)
        where (m,i,o) = s1
    OK (Boolean v1) s1 -> error (display m ++ "Input: " ++ show i ++ " " ++
"Output: " ++ show o)
        where (m,i,o) = s

```

```

-- Assignment statements perform a memory updating operation.
-- A memory is represented as a function which returns the
-- value of an identifier. To update a memory with a new
-- identifier-value mapping, we return a function that will
-- return the value if given the identifier or will use the
-- original memory function to retrieve values associated with
-- other identifiers.

```

```

-- *** Changes here ***

```

```

-- If ide is not already stored then (1) ide is added to the domain list d
-- and (2) the function is updated too, just like in the original "fixed"
-- version

```

```

update (d,f) ide val =
  (upd d, \ide2 -> if ide == ide2 then Stored val else f ide2)
  where upd [] = [ide]
        upd (x:xs) = if x == ide then (x:xs) else x : (upd xs)

```

```

-- We will later need a function to initialize an "empty" memory
-- that returns Unbound for every identifier.

```

```

emptyMem ide = Unbound

```

```
cmd_semantics (Assign ident exp) s =
  case (exp_semantics exp s) of
    OK v1 (m1, i1, o1) -> OKc (update m1 ident v1, i1, o1)
```

```
cmd_semantics (Output exp) s =
  case (exp_semantics exp s) of
    OK v1 (m1, i1, o1) -> OKc (m1, i1, o1 ++ [v1])
```

```
cmd_semantics (IfThenElse exp cmd1 cmd2) s =
  case (exp_semantics exp s) of
    OK (Boolean True) s1 -> cmd_semantics cmd1 s1
    OK (Boolean False) s1 -> cmd_semantics cmd2 s1
    OK (Numeric v) s1 -> Errorc
```

```
cmd_semantics (WhileDo exp cmd) s =
  case (exp_semantics exp s) of
    OK (Boolean True) s1 ->
      case (cmd_semantics cmd s1) of
        OKc s2 -> cmd_semantics (WhileDo exp cmd) s2
        Errorc -> Errorc
    OK (Boolean False) s1 -> OKc s1
    OK (Numeric v) s1 -> Errorc
```

```
cmd_semantics (Seq cmd1 cmd2) s =
  case (cmd_semantics cmd1 s) of
    OKc s1 -> cmd_semantics cmd2 s1
    Errorc -> Errorc
```

```
-- 4. Demo/Semantic Change/Demo
```

```
--
```

```
-- To demo the semantics in action, we use the following
-- "run" function to execute a TINY program for a given input.
-- (Note that the memory is initialized to empty, as is the output).
```

```
run program input =
-- *** Change here ***
  case (cmd_semantics program (([],emptyMem), input, [])) of
    OKc (m, i, o) -> o
    Errorc -> [ERROR]
```

```
-- Test programs
```

```
testprog1 =
  Seq (Output (Plus Read Read))
      (Output Zero)
```

```
input1 = [Numeric 1, Numeric 2]
```

```
input2 = [Numeric 1, Numeric 3]
```

```
input3 = [Boolean True, Numeric 2]
```

--- testprog2 is parsed version of the example in Gordon, section 2.3

```
testprog2 =  
  Seq (Assign "sum" Zero)  
  (Seq (Assign "x" Read)  
    (Seq (WhileDo (Not (Equal (I "x") TT))  
      (Seq (Assign "sum" (Plus (I "sum") (I "x")))) (Assign "x" Read)  
    )  
  )  
  (Output (I "sum"))  
)
```

```
input4 = [Numeric 1, Numeric 2, Boolean True]
```

```
input5 = [Numeric 1, Numeric 2, Numeric 3, Boolean True]
```

--- testprog3 computes sum from 0 to n (n is read as input) and writes sum as output

```
testprog3 =  
  Seq (Assign "sum" Zero)  
  (Seq (Assign "n" Read)  
    (Seq (Assign "j" Zero)  
      (Seq (WhileDo (Not (Equal (I "j") (I "n")))  
        (Seq (Assign "sum" (Plus (I "sum") (Plus (I "j") One)))  
          (Assign "j" (Plus (I "j") One))  
        )  
      )  
    )  
  (Output (I "sum"))  
)
```

--- testprog4 computes product of two inputs and writes this to output

```
testprog4 =  
  Seq (Assign "prod" Zero)  
  (Seq (Assign "m" Read)  
    (Seq (Assign "p" Read)  
      (Seq (Assign "i" Zero)  
        (Seq (WhileDo (Not (Equal (I "i") (I "m")))  
          (Seq (Assign "prod" (Plus (I "prod") (I "p")))  
            (Assign "i" (Plus (I "i") One))  
          )  
        )  
      )  
    )  
  (Output (I "prod"))  
)
```

