1. The program name identifier has no restrictions.
2. All identifiers that appear in a block must be declared in that block or in an enclosing block.
3. No identifier may be declared more than once at the top level of a block.
4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right side must be of the same type.
5. An identifier occurring as an (integer) element must be an integer variable or an integer constant.
6. An identifier occurring as a Boolean element must be a Boolean variable or a Boolean constant.
7. An identifier occurring in a read command must be an integer variable.
8. An identifier used in a procedure call must be defined in a procedure declaration with the same (zero or one) number of parameters.
9. The identifier defined as the formal parameter in a procedure declaration is considered to belong to the top level declarations of the block that forms the body of the procedure.
10. The expression in a procedure call must match the type of the formal parameter in the procedure's declaration.

Figure 9.19: Context Conditions for Pelican

Since environments map identifiers to types, we need a semantic domain Sort to assemble the possible types. Note that we distinguish between constants (*integer* and *boolean*) and variables (*intvar* and *boolvar*). It is important to remember that every domain is automatically augmented with an *error* value, and every semantic function and auxiliary function propagates *error*.

Semantic Domains

Boolean = { true, false }

Sort = { *integer*, *boolean*, *intvar, boolvar, program, unbound* }

Environment = Identifier → Sort

Semantic Functions

validate	:	Program → Boolean
examine	:	Block → Environment → Boolean
elaborate	:	Declaration → (Environment x Environment) → (Environment x Environment)
check	:	Command → Environment → Boolean
typify	:	Expression → Environment → Sort

Figure 9.20: Semantic Domains and Functions for Context Checking

We need two environments to elaborate each block:

1. One environment (locenv) holds the identifiers local to the block so that duplicate identifier declarations can be detected. It begins the block as an empty envirnoment with no bindings.
2. The other environment (env) collects the accumulated bindings from all of the enclosing blocks. This environment is required so that the expressions in constant declarations can be typified.

Both type environments are built in the same way by adding a new binding using *extendEnv* as each declaration is elaborated.

The semantic equations in Figure 9.21 show that each time a block is initialized, we build a local type environment starting with the empty environment. The first equation indicates that the program identifier is viewed as lying in a block of its own, and so it does not conflict with any other occurrences of identifiers. This alteration in the context conditions for program identifiers as compared to Wren makes the denotational specification much simpler.

```
validate [[program I is B]] =
        examine [[B]] extendEnv(emptyEnv,I,program)

examine [[D begin C end]] env = check [[C]] $env_1$
        where (locenv, $env_1$) = elaborate [[D]] (emptyEnv, env)

elaborate [[ε]] (locenv, env) = (locenv, env)
elaborate [[$D_1$ $D_2$]] = (elaborate [[$D_2$]]) ∘ (elaborate [[$D_1$]])
elaborate [[const I = E]] (locenv, env) = if applyEnv(locenv,I) = unbound
    then (extendEnv(locenv,I,typify [[E]] env),extendEnv(env,I,typify [[E]] env))
    else error
elaborate [[var I : T]] (locenv, env) = if applyEnv(locenv,I) = unbound
    then (extendEnv(locenv,I,type (T)),extendEnv(env,I,type (T)))
    else error
elaborate [[var I, L : T]] = (elaborate [[var L : T]]) ∘ (elaborate [[var I : T]])
```

Figure 9.21: Checking Context Constraints in Pelican (Part 1)

As declarations are processed, the environment for the current local block (locenv) and the cumulative environment (env) are constructed incrementally, adding a binding of an identifier to a type for each individual declaration while checking for multiple declarations of an identifier in the local environment. If an attempt is made to declare an identifier that is not *unbound* locally, the *error* value results. We assume that all semantic functions propagate the *error* value.

```
check [[C₁ ; C₂]] env = (check [[C₁]] env) and (check [[C₂]] env)
check [[skip]] env = true
check [[I := E]] env =
                (applyEnv (env,I) = intvar and typify [[E]] env = integer)
             or (applyEnv (env,I) = boolvar and typify [[E]] env = boolean)
check [[if E then C]] env = (typify [[E]] env = boolean) and (check [[C]] env)
check [[if E then C₁ else C₂]] env =
    (typify [[E]] env = boolean) and (check [[C₁]] env) and (check [[C₂]] env)
check [[while E do C]] env = (typify [[E]] env = boolean) and (check [[C]] env)
check [[declare B]] env = examine [[B]] env
check [[read I]] env = (applyEnv(I, env) = intvar)
check [[write E]] env = (typify [[E]] env = integer)

typify [[I]] env = case applyEnv(env,I) of
                        intvar, integer    : integer
                        boolvar, boolean   : boolean
                        program            : program
                        unbound            : error
typify [[N]] env = integer
typify [[true]] env = boolean
typify [[false]] env = boolean
typify [[E₁ + E₂]] env =
             if (typify [[E₁]] env = integer) and (typify [[E₂]] env = integer)
                  then integer else error
   :
typify [[E₁ and E₂]] env =
             if (typify [[E₁]] env = boolean) and (typify [[E₂]] env = boolean)
                  then boolean else error
   :
typify [[E₁ < E₂]] env =
             if (typify [[E₁]] env = integer) and (typify [[E₂]] env = integer)
                  then boolean else error
   :
```

Figure 9.21: Checking Context Constraints in Pelican (Part 2)

Checking commands involves finding Boolean or integer expressions where required and recursively checking sequences of commands that might occur. The semantic function *check* applied to a **declare** command just calls the *examine* function for the block. Simple expressions have their types determined directly. When we *typify* a compound expression, we must verify that its operands have the proper types and then specify the appropriate result type. If any part of the verification fails, *error* becomes the type value to be propagated.

A program satisfies the context-sensitive syntax of Pelican if *validate* produces true when applied to it. A final value of false or *error* means that the program does not fulfill the context constraints of the programming language.

The elaboration of the following Pelican program suggests the need for the local environment for context checking. Observe the difference if the Boolean variable is changed to “b”. Note that the expressions “m+21” cannot be typified without access to the global environment, env.

```
                                  locenv              env
program bad is                      [ ]               [ ]
    const m = 34;             [ m |→ integer ]    [ m |→ integer ]
  begin
    declare                         [ ]           [ m |→ integer ]
      var c : boolean;        [ c |→ boolvar ]  [ c |→ boolvar, m |→ integer]
      const c = m+21;            error
    begin
      write m+c;
    end
  end
```

Exercises

1. Apply the *validate* semantic function to these Pelican programs and elaborate the definitions that check the context constraints for Pelican.

a)
```
program a is
    const c = 99;
    var n : integer;
  begin
    read n;
    n := c-n;
    write c+1;
    write n
  end
```

b)
```
program b is
    const c = 99;
    var b : boolean;
  begin
    b := false;
    if b and true
       then b := c end if;
    b := c>0
  end
```

c) **program** c **is**
    var x,y,z : **integer**;
  begin
    read x;
    y := z;
    declare
      var x,z : **integer**;
    begin
      while x>0 **do**
        x := x-1 **end while**;
      declare
        var x,y : **boolean**;
        const y = **false**;
      begin
        skip
      end
    end
  end

d) **program** d **is**
    var b : **boolean**;
    const c = **true**;
  begin
    b := **not**(c) or **false**;
    read b;
    write 1109
  end

e) **program** e **is**
    var m,n : **integer**;
  begin
    read m;
    n := m/5;
    write n+k
  end

2. Extend the denotational semantics for context checking Pelican to include procedure declarations and calls.

3. Extend the result in exercise 2 to incorporate procedures with an arbitrary number of parameters.

4. Reformulate the denotational semantics for context checking Pelican using false in place of *error* and changing the signature of *elaborate* to

   elaborate : Declaration → Environment x Environment
   → Environment x Environment x Boolean

   Let *typify* applied to an expression with a type error or an unbound identifier take the value *unbound*.

5. Following the denotational approach in this section, implement a context checker for Pelican in Prolog.

9.7 CONTINUATION SEMANTICS

All the denotational definitions studied so far in this chapter embody what is known as **direct denotational semantics**. With this approach, each semantic equation for a language construct describes a transformation of argument domain values, such as environment and store, directly into results in some semantic domain, such as a new environment, an updated store, or an expressible value. Furthermore, the results from one construct pass directly