6

Type Systems

Exercises

6.1 **Type Rule 6.2 expanded**: All declared variables must have unique names and each one must have a type taken from the set {`int, bool, char, float` }.

6.2 The following Java method extends the code on page 139.

```
public static void V (Declarations d) {
        for (int i=0; i<d.size() - 1; i++) {
            Declaration di = d.get(i);
            for (int j=i+1; j<d.size(); j++) {
                 Declaration dj = d.get(j);
                 check( ! (di.v.equals(dj.v)),
                       "duplicate declaration: " + dj.v);
            }
            check( di.t.equals("int") || di.t.equals("bool") ||
                  di.t.equals("float") || di.t.equals("char"),
                 "invalid declared type: " + di.t);
        }
}
```

6.3 Each time through the outer loop, the ith declaration (di) is checked for type correctness. When the inner loop is entered, di's name (di.v) is checked to be sure that it is different from every subsequent declaration's name (dj.v). This strategy guarantees uniqueness among declared names without making any unnecessary checks (i.e., if di.v != dj.v, then dj.v != di.v doesn't need to be checked).

6.4 An imaginative student will run the *Clite* interpreter with each of these assignments to obtain an abstract syntax tree. After type checking, the interpreter will discover a mixed mode assignment to i in part (g) of this question.

The Clite code for this problem is given in the file `ex64.cpp` in the *ch06code* directory. The interpreter output for that run is shown in the file `ex64.output`.

6.5 Students will need to obtain the sources for the Clite interpreter to answer this question.

6.6 Type Rule 6.5 would be modified as follows:

3(b) If its *BinaryOp* is arithmetic, then each of its *Expression*s must be either char, int, or float.

4(c) If op is - then term must be char, int, or float.

For 3(b), the four lines that begin with `if (b.op.ArithmeticOp())` in Fig. 6.3 would be changed as follows:

```
if (b.op.ArithmeticOp())
   check ( (typ1 == Type.INT || typ1 == Type.FLOAT || typ1 == Type.CHAR) &&
           (typ2 == Type.INT || typ2 == Type.FLOAT || typ2 == Type.CHAR)
           , "type error for " + b.op);
```

The conversions themselves would be accomplished by the type transformer discussed in Section 6.2 (see Exercise 6.9).

6.7 Students will need to obtain the sources for the Clite interpreter to answer this question.

6.8 Students will need to obtain the sources for the Clite interpreter to answer this question.

6.9 The needed modifications for the type transformer in Fig. 6.6 are substantial, since each one of the different combinations of argument types for a *Binary* will generate a different transformed subtree.

6.10 Change the last two lines on the bottom of p 149 and the first two lines on the top of p 150 as follows:

$$\begin{aligned}&= V(e.\text{term1}, tm) \wedge V(e.\text{term2}, tm)\\ &\quad \wedge\ \mathit{typeOf}(e.\text{term1}, \text{tm}) \in \{\text{float}, \text{int}, \text{char}\} \wedge \mathit{typeOf}(e.\text{term2}, \text{tm}) \in \{\text{float}, \text{int}, \text{char}\}\\ &\qquad\qquad\qquad \text{if } e \text{ is a } \mathit{Binary} \wedge e.op \in \{\mathit{ArithmeticOp}\}\end{aligned}$$

6.11 Type Rule 6.5, part 2 should be modified as follows:

A *Variable* or an *ArrayRef* is valid if its `id` appears in the type map. Also, if it is an *ArrayRef* its `index` must be valid and have type `int`.

6.12 Type Rule 6.6, part 2 should be modified as follows:

> If the *Expression* is a *Variable* or *ArrayRef*, its result type is the type of that *Variable* or *ArrayRef*.

6.13 The `V` method in Fig. 6.2 needs no changes. The only two places a Clite *ArrayRef* can appear are in an *Expression* or as the `target` in an *Assignment*. The extended `V` and `typeOf` methods (see below) will cover an *ArrayRef* in either of these two contexts.

The `V` method in Fig.6.3 needs the following code inserted before the third `if` statement:

```
if (e instanceof ArrayRef) {
    ArrayRef a = (ArrayRef) e;
    check(tm.containsKey(a), "undefined array reference. " + a);
    return;
}
```

The `typeof` method in Fig. 6.4 needs the following code inserted before the third `if` statement:

```
if (e instanceof ArrayRef) {
    ArrayRef a = (ArrayRef) e;
    check(tm.containsKey(a), "undefined array reference. " + a);
    return (Type) tm.get(a);
}
```