Exercise 11: Language Extension: Data Type boolean

What changes would be necessary in the compiler if MicroJava had a data type boolean with the values true and false? It should also be possible to store the result of a Boolean expression in a boolean variable, for example:

```
boolVar = a < b && b < c;
```

Boolean variables should be usable in conditions and as operands of Boolean expressions, for example:

```
if (boolVar) ...
if (boolVar || a < b) ...
```

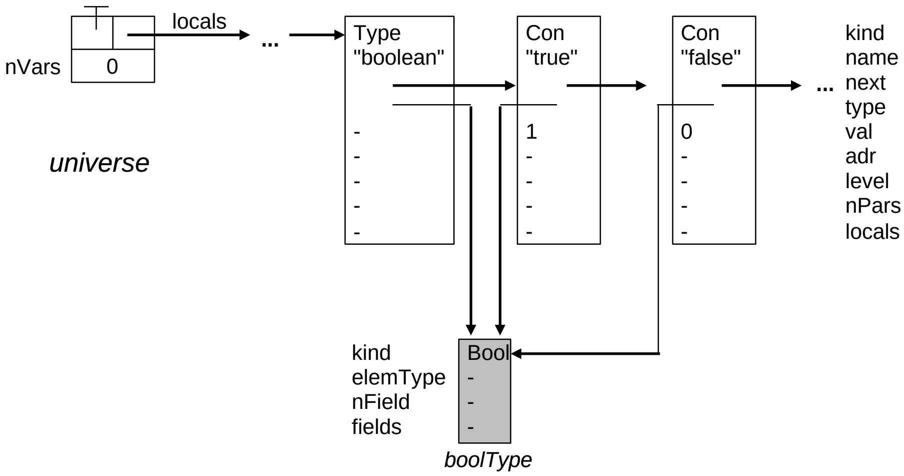
Finally, it should also be possible to output Boolean expressions using the print statement.

Solution

Although the introduction of a type boolean seems simple, extensive changes are necessary in the compiler.

Like int and char, the type boolean is a predeclared type (i.e., not a keyword) whose name has to be entered into the universe. In the same way as the constant null, two predeclared Boolean constants true and false with the values 1 and 0 have to be entered into the universe.

There must also be a new structure node of kind Bool, which is exported from the Tab class as boolType. Just like variables of types int and char, variables of type boolean have word size (4 bytes).



Declarations, assignments and comparisons work the same for boolean variables as for int or char variables. This means that no changes need to be made in the compiler. However, to make it possible to assign a Boolean expression to a boolean variable

```
boolVar = a < b && b < c;
```

you must be able to convert a Cond operand to true (1) or false (0). Conversely, you must be able to convert a boolean variable into a Cond operand in conditions so that you can write

```
if (boolVar) ...
```

The conversion of a Cond operand into the values true or false happens in the method Code.load():

```
public static void load (Operand x) {
   switch (x.kind) {
      ...
      case Operand.Cond:                   // load true or false
         Code.fJump(x.op, x.fLabel);       // jump across true
         x.tLabel.here();
         Code.put(Code.const1);            // true
         Label end = new Label();
         Code.jump(end);                   // jump across false
         x.fLabel.here();
         Code.put(Code.const0);            // false
         end.here();
         x.type = Tab.boolType;
         break;
      ...
   }
   x.kind = Operand.Stack;
}
```

For example, the assignment

```
boolVar = a > 5;
```

in which the right-hand side is loaded by the method load(), results in the following code:

```
100  load0     // load a
101  const5    // load 5
102  jle 7     // false jump => 109
105  const1    // true
106  jmp 4     // => 110
109  const0    // false
110  store1    // store to boolVar
```

For this to work, however, a Boolean expression must be allowed on the right-hand side of an assignment. We must therefore change the MicroJava grammar so that Expr can also be a Boolean expression:

```
Expr         = Term {"||" Term}.
Term         = Factor {"&&" Factor).
Factor       = SimpleExpr [RelOp SimpleExpr].
SimpleExpr   = SimpleTerm {AddOp SimpleTerm}.
SimpleTerm   = SimpleFactor {MulOp SimpleFactor}.
SimpleFactor = Designator [ActPars] | number | charConst | "new" ident ["[" Expr "]"] | "(" Expr ")".
```

This grammar also allows for parentheses in compound Boolean expressions, for example:

```
if ((a == 0 || a == 1) && b > 0) ...
```

We show the (attributed) productions of Expr, Term and Factor here. The productions of SimpleExpr, SimpleTerm and SimpleFactor are the same as the previous productions of Expr, Term and Factor. In exchange, we no longer need the productions of Condition, CondTerm and CondFactor.

```
Expr <↑x>            (. Operand x, y; .)
= Term <↑x>
   {  "||"           (. x = makeCond(x);
                        Code.tJump(x.op, x.tLabel);
                        x.fLabel.here(); .)
      Term <↑y>      (. y = makeCond(y);
                        x.op = y.op;
                        x.fLabel = y.fLabel;
                        x.tLabel.merge(y.tLabel); .)
   } .
```

The method makeCond() converts an operand x, which might represent a boolean variable, into a Cond operand by generating code that loads x and compares it with 1(true).

```
private static Operand makeCond (Operand x) {
   if (x.kind == Operand.Cond) return x; // is already Cond => nothing to do
   if (x.type == Tab.boolType) { // only boolean variables can be converted to Cond
      Code.load(x);
      Code.put(Code.const1); // true
   } else error("boolean expected");
   return new Operand(Operand.Cond, Code.eq, Tab.boolType); // compare with true
}
```

For example, the expression

```
boolVar || ...
```

is translated into

```
100  load0      // load boolVar
101  const1     // load true
102  jeq ...    // jump if boolVar == true
```

Since there can now also be nested Boolean expressions (in parentheses), the second Term in Expr provides a Cond operand that can contain both unresolved *true jumps* and unresolved *false jumps*. The *false jumps* of the first Term are resolved in the attributed grammar, so only the *false jumps* of the second Term (y.fLabel) need to be propagated as the *false jumps* of Term || Term, while the *true jumps* of the second Term need to be added to the *true jumps* of the first Term, which is done using the method merge() in the Label class.

```
public void merge (Label other) {
   this.fixupList.addAll(other.fixupList);
}
```

Here are the attributed productions of Term and Factor. The explanations on makeCond() and merge() apply here analogously to the production of Expr.

```
Term <↑x>              (. Operand x, y; .)
= Factor <↑x>
   { "&&"              (. x = makeCond(x);
                          Code.fJump(x.op, x.fLabel);
                          x.tLabel.here(); .)
      Factor <↑y>      (. y = makeCond(y);
                          x.op = y.op;
                          x.tLabel = y.tLabel;
                          x.fLabel.merge(y.fLabel); .)
   } .

Factor <↑x>            (. Operand x, y; int op; .)
= SimpleExpr <↑x>
   [ RelOp <↑op>       (. Code.load(x); .)
      SimpleExpr <↑y>  (. Code.load(y);
                          if (!x.type.compatibleWith(y.type)) error("type mismatch");
                          if (x.type.isRefType() && op != Code.eq && op != Code.ne) error("invalid compare");
                          x = new Operand(Operand.Cond, op, Tab.boolType); .)
   ] .
```

If Factor is a comparison, the two operands are loaded and a Cond operand with the comparison operator is returned. If Factor is just a SimpleExpr, the attribute of SimpleExpr is returned by Factor.

We also have to change the if statement so that it also works if the condition is a simple boolean variable:

```
Statement                  (. Operand x; .)
=  ...
|  "if" "(" Expr <↑x> ")"  (. x = makeCond(x);
                              Code.fJump(x.op, x.fLabel);
                              x.tLabel.here(); .)
   Statement
   (  "else"               (. Label end = new Label();
                              Code.jump(end);
                              x.fLabel.here(); .)
      Statement            (. end.here(); .)
   |                       (. x.fLabel.here(); .)
   )
|  ...
```

The only difference to before is that the condition in the if statement can be an Expr and that its attribute is converted into a Cond operand with makeCond() if necessary. The if statement

```
if (boolVar) a = 0;
```

results in the following code:

```
200  load1    // load boolVar
201  const1   // load true
202  jne 4    // false jump => 206
205  const0   // a = 0;
206  store0
206  ...
```

Analogous to the if statement, the while statement must also be changed so that its condition is an Expr whose attribute is converted into a Cond operand with makeCond() if necessary.