Aufgabe 11: Spracherweiterung: Datentyp boolean

Welche Änderungen wären im Compiler nötig, wenn es in MicroJava den Datentyp boolean mit den Konstanten true und false gäbe? Es soll auch möglich sein, das Ergebnis boolescher Ausdrücke in einer boolean-Variablen zu speichern, zum Beispiel:

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

Ebenso sollen boolean-Variablen in Abfragen und als Operanden boolescher Ausdrücke verwendbar sein, zum Beispiel:

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

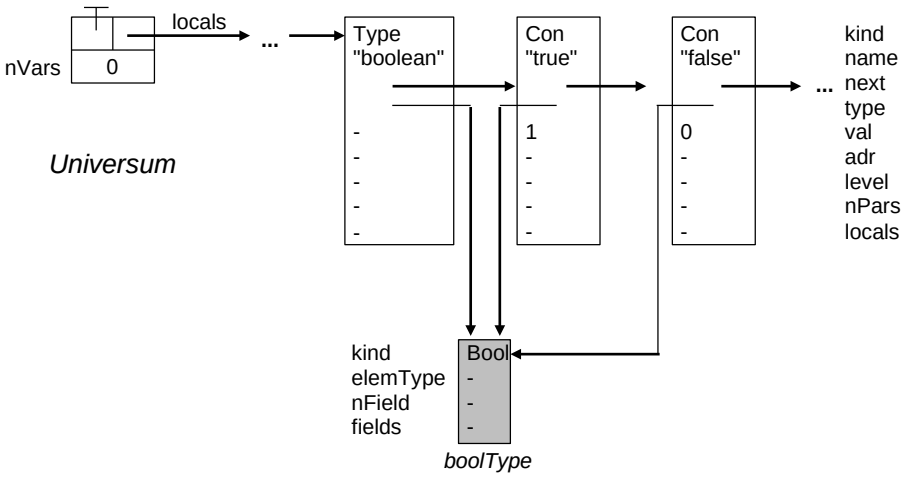
Es soll ferner möglich sein, boolesche Ausdrücke mit einer print-Anweisung auszugeben.

Lösung

Obwohl die Einführung eines Typs boolean einfach erscheint, sind doch umfangreiche Änderungen im Compiler nötig.

Der Typ boolean ist so wie int und char ein vordeklarierter Typ (also kein Schlüsselwort), dessen Name im Universum eingetragen wird. Ebenso werden so wie für die Konstante null zwei vordeklarierte Konstanten true und false mit den Werten 1 und 0 und dem Typ boolean ins Universum eingetragen.

Es muss auch eine neue Art von Strukturknoten (Bool) geben. Im Universum wird so ein Strukturknoten angelegt und von der Klasse Tab als boolType exportiert. So wie Variablen der Typen int und char werden auch Variablen des Typs boolean in einem Wort (4 Bytes) gespeichert.



Deklarationen, Zuweisungen und Vergleiche funktionieren für boolean-Variablen gleich wie für int- oder char-Variablen. Hier sind also keine Änderungen im Compiler nötig. Allerdings soll es möglich sein, einen booleschen Ausdruck einer boolean-Variablen zuzuweisen, zum Beispiel

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

Dazu muss man einen Cond-Operanden in true (1) oder false (0) umwandeln können. Umgekehrt muss man in Abfragen boolean-Variablen in Cond-Operanden umwandeln können, damit man zum Beispiel schreiben kann

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

Die Umwandlung eines Cond-Operanden in die Werte true oder false erfolgt in der Methode 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;
}
```

Die Zuweisung

```
boolVal = a > 5;
```

bei der die rechte Seite mit load geladen wird, führt zu folgendem 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
```

Damit das aber funktioniert, muss der Ausdruck auf der rechten Seite der Zuweisung ein Vergleichsausdruck sein dürfen. Wir müssen daher die Grammatik von MicroJava so ändern, dass Expr auch ein Vergleichsausdruck sein darf:

```
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 ")".
```

Das erlaubt nun sogar Klammern in zusammengesetzten booleschen Ausdrücken, zum Beispiel:

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

Wir geben hier die Attributierung der Produktionen von Expr, Term und Factor an. Die Attributierung von SimpleExpr, SimpleTerm und SimpleFactor ist gleich wie die bisherige Attributierung von Expr, Term und Factor. Dafür brauchen wir die Produktionen von Condition, CondTerm und CondFactor nicht mehr.

```
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); .)
   } .
```

Die Methode makeCond sorgt dafür, dass ein Operand x, der eine boolean-Variable darstellt, in einen Cond-Operanden umgewandelt wird, wobei Code erzeugt wird, der x lädt und mit true vergleicht.

```
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
}
```

Der Ausdruck

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

wird übersetzt zu

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

Da es nun auch geklammerte boolesche Ausdrücke geben kann, kann der zweite Term in Expr einen Cond-Operanden liefern, der sowohl unaufgelöste True-Jumps als auch unaufgelöste False-Jumps enthalten kann. Damit diese Fixuplisten nicht verloren gehen, müssen wir sie zu den True-Jumps und False-Jumps des ersten Terms hinzufügen. Die False-Jumps des ersten Terms wurden bereits aufgelöst, daher muss nur y.fLabel als Fixupliste von Term || Term übernommen werden. Die True-Jumps des zweiten Terms müssen aber zu den True-Jumps des ersten Terms hinzugefügt werden, was durch die Methode merge in der Klasse Label bewerkstelligt wird.

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

Hier folgen noch die attributierten Produktionen von Term und Factor. Die Erläuterungen zu makeCond und merge gelten hier analog zur Produktion von 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); .)
  ] .
```

Wenn Factor ein Vergleich ist, werden die beiden Operanden geladen, und es wird ein Cond-Operand mit dem Vergleichsoperator zurückgegeben. Wenn Factor nur ein SimpleExpr ist, wird das Attribut von SimpleExpr auch von Factor zurückgegeben.

Nun müssen wir aber auch noch die if-Anweisung so ändern, dass sie auch funktioniert, wenn der Vergleichsausdruck eine einfache boolesche Variable ist:

```
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(); .)
  )
| ...
```

Der einzige Unterschied zu bisher ist, dass die if-Bedingung ein Expr sein kann und dass dessen Attribut mit makeCond nötigenfalls in einen Cond-Operanden umgewandelt wird. Die if-Anweisung

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

führt dann zu folgendem Code:

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

Analog zur if-Anweisung ist auch die while-Anweisung so zu ändern, dass die while-Bedingung ein Expr ist, dessen Attribut mit makeCond nötigenfalls in einen Cond-Operanden umgewandelt wird.