Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik

Programmieren in C++

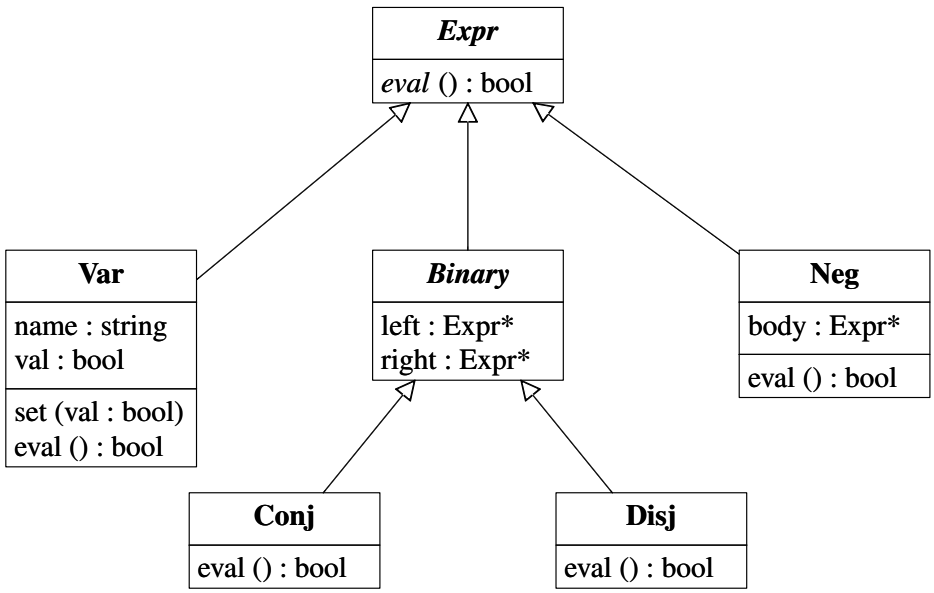
Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

5. Übungsblatt (23. November 2017)

Aufgabe 9: Virtuelle Funktionen, Besuchermuster

Teilaufgabe 9.a)

Implementieren Sie die in der Abbildung dargestellte Klassenhierarchie zur Repräsentation aussagenlogischer Formeln/Ausdrücke!



Eine logische Variable `Var` besitzt einen Namen `name` und einen Wert `val`, der mittels `set` verändert werden kann (Anfangswert `false`). Negationen `Neg`, Konjunktionen `Conj` und Disjunktionen `Disj` haben die übliche Bedeutung. `eval` ermittelt den Wert eines Ausdrucks.

Anwendungsbeispiel:

```
Var* a = new Var("a");
Var* b = new Var("b");
Expr* x = new Disj(new Conj(new Neg(a), b), new Conj(a, new Neg(b)));
a->set(true);
b->set(false);
cout << (x->eval() ? "T" : "F") << endl; // Ausgabe: T
```

Teilaufgabe 9.b)

Integrieren Sie das Besuchermuster (visitor pattern) wie folgt in die obige Klassenhierarchie:

- Definieren Sie die folgende abstrakte Klasse `Visitor`:

```
struct Visitor {
    virtual void visit (Var* x) = 0;
    virtual void visit (Neg* x) = 0;
    virtual void visit (Conj* x) = 0;
    virtual void visit (Disj* x) = 0;
};
```

  Alternativ könnten die Elementfunktionen auch `visitVar`, `visitNeg`, `visitConj` und `visitDisj` heißen.

- Fügen Sie die folgende rein virtuelle Funktion `accept` zur Klasse `Expr` hinzu:

```
virtual void accept (Visitor& v) = 0;
```

- Fügen Sie zu jeder konkreten Klasse folgende Implementierung von `accept` hinzu:

```
virtual void accept (Visitor& v) { v.visit(this); }
```

  Achtung: Obwohl `accept` textuell immer gleich ist, muss es zu *jeder* konkreten Klasse hinzugefügt werden. Eine einmalige Definition in der Wurzelklasse `Expr` würde nicht funktionieren. Warum? (Wenn die vier Elementfunktionen von `Visitor` unterschiedliche Namen besitzen, müssen diese hier anstelle von `visit` verwendet werden.)

Hinweis: Die wechselseitigen Abhängigkeiten zwischen `Visitor` und den anderen Klassen können durch Vorabdeklarationen aufgelöst werden, zum Beispiel:

```
struct Var;
```

Anschließend kann der Name `Var` eingeschränkt verwendet werden, z. B. als Zieltyp von Zeiger- und Referenztypen.

Teilaufgabe 9.c)

Implementieren Sie eine Klasse `PrintVisitor` sowie eine globale Funktion `print` zur Ausgabe von Ausdrücken:

```
struct PrintVisitor : Visitor {
    virtual void visit (Var* x) { ..... }   // Variable x ausgeben.
    virtual void visit (Neg* x) { ..... }   // Negation x (rekursiv) ausgeben.
    virtual void visit (Conj* x) { ..... }  // Konjunktion x (rekursiv) ausgeben.
    virtual void visit (Disj* x) { ..... }  // Disjunktion x (rekursiv) ausgeben.
};

// Ausdruck x ausgeben.
void print (Expr* x) {
    static PrintVisitor v;
    x->accept(v);
}
```

Fortsetzung des obigen Anwendungsbeispiels:

```
print(x); cout << endl; // Ausgabe: ((!a&b)|(a&!b))
```

Teilaufgabe 9.d)

Implementieren Sie entsprechend eine Klasse `CollectVarsVisitor`, deren Elementfunktionen alle in einem Ausdruck enthaltenen Variablen in einem Feld o. ä. sammeln:

```
struct CollectVarsVisitor : Visitor {
    ......
    virtual void visit (Var* x) { ...... }
    virtual void visit (Neg* x) { ...... }
    virtual void visit (Conj* x) { ...... }
    virtual void visit (Disj* x) { ...... }
};
```

Implementieren Sie dann eine globale Funktion `solve`, die für einen Ausdruck eine oder mehrere *erfüllende Belegungen* seiner Variablen ermittelt und ausgibt, indem sie nacheinander alle möglichen Belegungen ausprobiert:

```
void solve (Expr* x) {
    CollectVarsVisitor v;
    x->accept(v);
    // Jetzt enthält das Objekt v
    // alle im Ausdruck x enthaltenen Variablen.

    // Alle Belegungen dieser Variablen durchlaufen und für jede Belegung
    // mittels x->eval() überprüfen, ob der Ausdruck x mit dieser Belegung
    // wahr wird.
    ......
}
```

Fortsetzung des obigen Anwendungsbeispiels:

```
solve(x);
// Ausgabe: a = T, b = F
// Und/oder: a = F, b = T
```

Lösung

```
#include <string>
#include <set>
#include <iostream>
using namespace std;

// Notwendige Vorabdeklaration.
struct Visitor;

// Allgemeiner logischer Ausdruck.
struct Expr {
    // Ausdruck auswerten.
    virtual bool eval () const = 0;

    // Besucher v "akzeptieren".
    virtual void accept (Visitor& v) = 0;
};
```

```
// Logische Variable.
struct Var : Expr {
    // Name.
    const string name;

    // Aktueller Wert.
    bool val;

    // Initialisierung mit Name name und Wert false.
    Var (const string& name) : name(name), val(false) {}

    // Wert auf val setzen.
    void set (bool val) { this->val = val; }

    // Wert liefern.
    virtual bool eval () const { return val; }

    // Besucher v "akzeptieren".
    virtual void accept (Visitor& v);
};

// Logische Negation.
struct Neg : Expr {
    // Operand.
    Expr* const body;

    // Initialisierung mit Operand body.
    Neg (Expr* body) : body(body) {}

    // Negation auswerten.
    virtual bool eval () const { return !body->eval(); }

    // Besucher v "akzeptieren".
    virtual void accept (Visitor& v);
};

// Binärer logischer Ausdruck.
struct Binary : Expr {
    // Linker und rechter Operand.
    Expr* const left;
    Expr* const right;

    // Initialisierung mit Operanden left und right.
    Binary (Expr* left, Expr* right) : left(left), right(right) {}
};

// Konjunktion.
struct Conj : Binary {
    // Konstruktor von Binary erben.
    using Binary::Binary;

    // Konjunktion auswerten.
    virtual bool eval () const { return left->eval() && right->eval(); }
```

```
    // Besucher v "akzeptieren".
    virtual void accept (Visitor& v);
};

// Disjunktion.
struct Disj : Binary {
    // Konstruktor von Binary erben.
    using Binary::Binary;

    // Disjunktion auswerten.
    virtual bool eval () const { return left->eval() || right->eval(); }

    // Besucher v "akzeptieren".
    virtual void accept (Visitor& v);
};

// Allgemeiner Besucher.
struct Visitor {
    virtual void visit (Var* x) = 0;
    virtual void visit (Neg* x) = 0;
    virtual void visit (Conj* x) = 0;
    virtual void visit (Disj* x) = 0;
};

// Implementierungen von accept.
void Var::accept (Visitor& v) { v.visit(this); }
void Neg::accept (Visitor& v) { v.visit(this); }
void Conj::accept (Visitor& v) { v.visit(this); }
void Disj::accept (Visitor& v) { v.visit(this); }

// Ausdruck x rekursiv ausgeben.
void print (Expr* x);

// Besucher zum Ausgeben von Ausdrücken.
struct PrintVisitor : Visitor {
    // Variable x ausgeben.
    virtual void visit (Var* x) {
        cout << x->name;
    }

    // Negation x ausgeben.
    virtual void visit (Neg* x) {
        cout << "!";
        print(x->body);
    }
```

```
    // Binären Ausdruck x mit Operatorsymbol oper ausgeben.
    void visit (Binary* x, char oper) {
        cout << "(";
        print(x->left);
        cout << oper;
        print(x->right);
        cout << ")";
    }

    // Konjunktion x ausgeben.
    virtual void visit (Conj* x) {
        visit(x, '&');
    }

    // Disjunktion x ausgeben.
    virtual void visit (Disj* x) {
        visit(x, '|');
    }
};

// Ausdruck x rekursiv ausgeben.
void print (Expr* x) {
    static PrintVisitor v;
    x->accept(v);
}

// Besucher zum Sammeln von Variablen in Ausdrücken.
struct CollectVarsVisitor : Visitor {
    // Menge der Variablen.
    set<Var*> vars;

    // Variable x sammeln.
    virtual void visit (Var* x) {
        vars.insert(x);
    }

    // Variablen im Ausdruck x sammeln.
    void collect (Expr* x) {
        x->accept(*this);
    }
    virtual void visit (Neg* x) {
        collect(x->body);
    }
    virtual void visit (Conj* x) {
        collect(x->left);
        collect(x->right);
    }
    virtual void visit (Disj* x) {
        collect(x->left);
        collect(x->right);
    }
};
```

```
// Eine erfüllende Belegung der Variablen des Ausdrucks x suchen
// und ausgeben.
void solve (Expr* x) {
    // Variablen im Ausdruck x sammeln.
    CollectVarsVisitor v;
    x->accept(v);

    while (true) {
        // Wenn der Ausdruck mit der aktuellen Variablenbelegung
        // erfüllt ist: Variablenwerte ausgeben.
        if (x->eval()) {
            for (Var* var : v.vars) {
                cout << var->name << " = "
                                << (var->eval() ? "T" : "F") << endl;
            }
            break;
        }

        // Die Menge v.vars der Variablen wird als Dualzahl mit
        // Anfangswert 0 (alle Variablen sind false) aufgefasst.
        // Um alle möglichen Belegungen zu durchlaufen, wird in
        // jedem Schritt 1 zu dieser Dualzahl addiert.
        int n = 0;
        for (Var* var : v.vars) {
            // Wenn die aktuelle "Stelle" var der Dualzahl 1 ist,
            // wird sie auf 0 gesetzt und die Schleife fortgesetzt
            // ("Übertrag").
            if (var->eval()) {
                var->set(false);
                n++;
            }
            // Andernfalls (aktuelle "Stelle" ist 0) wird sie auf 1
            // gesetzt und die Schleife abgebrochen (kein "Übertrag").
            else {
                var->set(true);
                break;
            }
        }

        // Wenn alle "Stellen" von 1 auf 0 gesetzt wurden, hat man
        // wieder den Anfangswert 0 erreicht, d. h. alle Belegungen
        // durchlaufen.
        if (n == v.vars.size()) break;
    }
}
```

```
// Testprogramm.
int main () {
    // (!a & b) | (a & !b)
    Var* a = new Var("a");
    Var* b = new Var("b");
    Expr* x =
            new Disj(new Conj(new Neg(a), b), new Conj(a, new Neg(b)));
    print(x); cout << endl;
    solve(x);
};
```