Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik

Programmieren in C++

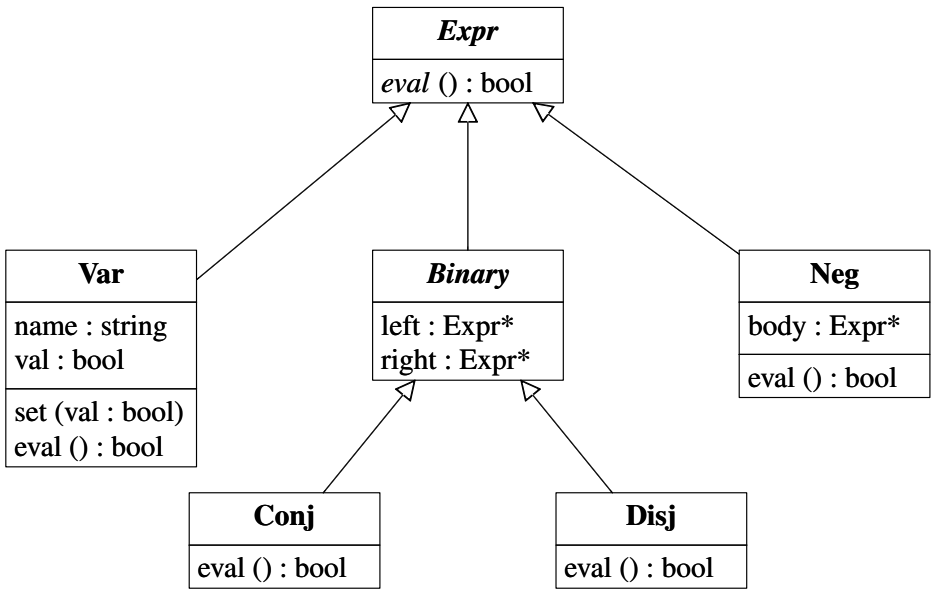
Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

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
```