IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)
Aufgabe 05

Hinweise
Abgabe: Stand des Git-Repositories am 31.5.2022 um 9 Uhr.
Die Dateien zur Bearbeitung dieser Aufgabe erhalten Sie, indem Sie die neue Aufgabe aus dem Aufgabenrepository in Ihr lokales mergen. Dies geschieht mit `git pull common main` innerhalb Ihres Repositories. Die Lösungen committen Sie bitte in Ihr lokales Repository und pushen sie in Ihr Repository auf den Gitlab-Server.

Teilaufgabe 1 (4 Punkte)

In dieser Aufgabe soll ein Programm vervollständigt werden, das boolsche Logikfunktionen in Auswertungsbäume überführt. Auf der Grundlage des Auswertungsbaums einer logischen Funktion soll für beliebige Belegungen der enthaltenen Variablen der resultierende Logikwert, `true` oder `false`, berechnet werden können. Darüber hinaus soll der Auswertungsbaum die Ausgabe der abgebildeten Logikformel in unterschiedlichen Notationen ermöglichen.

Um die Auswertung der einzelnen logischen Operationen durchzuführen, wird für jeden Operator ein polymorphes `Calculator`-Objekt erzeugt. Die `Calculator`-Objekte sind die Knoten des Baums. Knoten binärer Operatoren haben zwei Nachfolger, unäre Operatoren einen Nachfolger. Die Auswertung des Baums erfolgt durch einen Methodenaufruf auf dem Wurzelknoten.

a) Vervollständigen Sie die Klasse `NotCalculator` in der Datei `calculator.h`! Dazu müssen Sie alle Memberfunktionen, die in `UnaryCalculator` abstrakt sind, überschreiben und in `calculator.cpp` implementieren.

b) Definieren Sie in der Datei `calculator.h` Klassen zur Repräsentation (und Berechnung) der AND- und der OR-Verknüpfung! Implementieren Sie die dazugehörigen Memberfunktionen getrennt von der Klassendefinition in der Datei `calculator.cpp`.
Hinweise:
Sie können dies ähnlich lösen wie bei der Klasse `UnaryCalculator` und der abgeleiteteten Klasse `NotCalculator`. Allerdings müssen Sie bedenken, dass AND und OR zwei Operanden besitzen und nicht nur einen. Die Ausdrücke für die Argumente sollten wie in `UnaryCalculator` als Smartpointer verwaltet werden. Die

Appelstraße 4 | 30167 Hannover
fon 0511 762-19691 | fax 0511 762-19694
www.ims.uni-hannover.de | info@ims.uni-hannover.de

abstrakte Methode `print` müssen Sie überschreiben. Die Implementation dieser Methode ist aber erst in Teilaufgabe 3 gefordert, für diese Teilaufgabe reicht ein leerer Methodenrumpf aus.

c) Erklären Sie, warum die Methoden der Klasse `Calculator` `virtual` sind und erläutern Sie die Auswirkungen!

Teilaufgabe 2 (1 Punkt)

Die Logikfunktion wird durch eine Formel in Präfixnotation ausgedrückt. In dieser Formel steht '&' für eine AND-Verküpfung, '|' für eine OR-Verküpfung und '~' für eine NOT-Operation. Variablen werden jeweils durch einzelne Ziffern ausgedrückt, die den Index in der Belegung angeben. Als Beispiel beschreibt `"&&01~2"` eine Funktion, bei der eine AND-Verknüpfung von dem Logikwert an Index 0 mit dem von Index 1 und der Invertierung (NOT) des Werts an Index 2 durchgeführt wird.

Insgesamt ergibt sich somit die folgende Grammatik in EBNF für einen logischen Ausdruck:

```
<expression>  = <and-expr> | <or-expr> | <not-expr> | <input-index>
<and-expr>    = "&", <expression>, <expression>
<or-expr>     = "|", <expression>, <expression>
<not-expr>    = "~", <expression>
<input-index> = <digit>
<digit>       = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Das Parsen der Logikfunktion und das Erzeugen der entsprechenden `Calculator` geschieht in den Funktionen `parseExpression` und `parseExpressionRecursive` in der Datei `parser.cpp`. Vervollständingen Sie die Funktion `parseExpressionRecursive` um die Behandlung von AND ('&') und OR ('|')!

Teilaufgabe 3 (2 Punkte)

Ergänzen Sie das Programm so, dass ein Aufruf der in Calculator definierten `print`-Methode abhängig vom übergebenen `TraversalType` eine Ausgabe in Präfix-, Infix- und Postfix-Notation durchführt.

Hinweise:

Den Typ `BonusInfix` müssen Sie in dieser Teilaufgabe nicht berücksichtigen. Die Ausgabe darf keine Leerzeichen enthalten und soll für die Operatoren dieselben Zeichen wie bei der Eingabe verwenden (`~` für NOT, `&` für AND und `|` für OR). Setzen Sie bei der Ausgabe in Infix-Notation grundsätzlich runde Klammern um die Argumente eines Operators. Die Ausgabe ist also beispielsweise `"((a)&(b))|(~(c))"` anstatt `"a&b|~c"`. Für Präfixnotation entspricht die Ausgabe genau dem in Teilaufgabe 2 spezifizierten Eingabeformat.

Teilaufgabe 4 (1 Punkt)

Sehen Sie sich die Klasse `CalculatorPrinter` an! Welche Aufgabe erfüllt diese Klasse? `CalculatorPrinter` ruft lediglich eine einzelne Memberfunktion von `Calculator` auf. Warum ist diese Klasse dennoch sinnvoll?

Bonusaufgabe (2 Punkte)

Erweitern Sie `print` derart, dass auch `TraversalType::BonusInfix` funktioniert. Dieser `TraversalType` soll eine möglichst lesbare Ausgabe der Formel bewirken. Die Ausgabe soll in Infixnotation erfolgen, für diesen Ausgabetyp dürfen aber auch Leerzeichen vorkommen. Setzen Sie für dieses Ausgabeformat nur Klammern, wenn dies auch tatsächlich notwendig ist. Dafür können Sie die folgenden Regeln verwenden:

- `~` bindet am stärksten, `&` am zweitstärksten und `|` am schwächsten. `"~a&b|c"` ist also äquivalent zu `"((~a)&b)|c"`.
- Binäre Operatoren sind assoziativ, `"a&b&c"` ist also identisch zu `"(a&b)&c"` und zu `"a&(b&c)"`.

Die Formel `"(((~a)&b)|~(c|d))|e"` könnte also beispielsweise als `"~a & b | ~(c | d)| e"` ausgegeben werden.

Hinweise:
`dynamic_cast` kann verwendet werden, um zu prüfen, ob ein Pointer auf eine Basisklasse auf ein Objekt eines bestimmten Typs zeigt. Die Verwendung von Leerzeichen steht Ihnen völlig frei, sie sollten aber die Lesbarkeit eher unterstützen als sie behindern (`"~a&b"` als `"~ a&b"` auszugeben, ist der Lesbarkeit eher abträglich).