Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 12/13
Aufgabenblatt 2 - Lösungsvorschlag

Abgabemodalitäten

Die Aufgabe sind in 2er Gruppen zu bearbeiten. Ihre Lösungen reichen Sie als Zip-Datei per E-Mail bis zum 23.12.2012 um 23:59 MEZ an der Adresse oc@esa.informatik.tu-darmstadt.de ein. Die Zip-Datei solte Ihre Implementierung der Klassen *SymbolTable* und *ConstantFolding* jeweils als Java-Quelldatei enthalten. Die E-Mail muss als Betreff "Compiler 1 Aufgabe 2" haben. Geben Sie in Ihrem Lösungsdokument den Namen und die Matrikelnummer aller Gruppenmitglieder an.

Aufgabe 2.1 Effiziente Implementierung der Symboltabelle — 25 Punkte

In Vorlesungsblock drei wurde die im Rahmen der Kontextanalyse stattfindende Erstellung der Symboltabelle thematisiert. Die Datenstruktur der im Triangle-Compiler per se implementierten Variante basiert auf verketteten Listen. Wie aufgezeigt sind weitaus effizientere Implementierungen denkbar (vgl. Skript: Block 3, S. 20-22). Im Rahmen dieser Aufgabe sollen Sie diese effizientere Fassung implementieren.

Laden Sie sich hierzu zunächst den auf der Website der Vorlesung verfügbaren Quellcode des Triangle-Compilers herunter. Sie werden feststellen, dass die für diese Aufgabe notwendigen Dateien im Package *Triangle.ContextualAnalyzer* zu finden sind. Die angesprochene Variante einer Symboltabelle auf Basis verketteter Listen liegt in der Klasse *IdentificationTable* vor.

Implementieren Sie nun auf Basis des bereitgestellten Grundgerüsts die Klasse *SymbolTable*, die als Ersatz der Klasse *IdentificationTable* dienen soll. Verwenden Sie bei Ihrer Implementierung das in der Vorlesung vorgestellte Konzept. Achten Sie insbesondere darauf, die bereitgestellte API nicht zu verändern, um Ihre Klasse zum Triangle-Compiler kompatibel zu halten. Hierzu zählen alle öffentlichen Methoden der bereitgestellten Klasse.

Um Ihre Lösung zu testen, können Sie Ihre Implementierung in den Triangle-Compiler einfügen. Bearbeiten Sie hierzu die Klasse *Checker*. Lokalisieren Sie das Objekt-Attribut *idTable* und ändern Sie dessen Typ von *IdentifcationTable* auf *SymbolTable*. Hiernach wird der Triangle-Compiler auf Ihre Implementierung der Symboltabelle zurückgreifen.

Aufgabe 2.1 Lösung

Den Lösungsvorschlag zu dieser Aufgabe finden Sie in Form der Datei *SymbolTable.java* in der Anlage.

Aufgabe 2.2 Vereinfachung von konstanten Ausdrücken — 50 Punkte

Einige einfache Optimierungen können bereits auf dem abstrakten Syntaxbaum realisiert werden. In dieser Aufgabe sollen Sie die Vereinfachung von konstanten Ausdrücken durch ein Visitor Pattern kennenlernen und implementieren. Hierbei werden Ausdrücke deren Ergebniss bereits zur Kompilierungszeit berechnet werden kann zusammengefasst. Dadurch kann ein Programm wie in Listing 1 zu dem Programm in Listing 2 vereinfacht werden.

Listing 1: Beispielprogramm

```
let
  var   a : Boolean;
  var   b : Boolean;
  var   x : Integer;
  var   y : Integer;
  var   z : Integer;
  const n ~ 42 + 1;
  const m ~ n + 1
in
  a := true /\ false;
  b := true /\ (1 < 2);
  x := n * 1;
  y := (4 + 7) * 5;
  z := y + 1;
```

Listing 2: Optimiertes Beispielprogramm

```
let
  var   a : Boolean;
  var   b : Boolean;
  var   x : Integer;
  var   y : Integer;
  var   z : Integer;
  const n ~ 43;
  const m ~ n + 1
in
  a := false;
  b := true;
  x := n;
  y := 55;
  z := y + 1;
```

Implementieren Sie hierzu folgende Vereinfachungsregeln

- C_1 op C_2 wird ersetzt durch das Ergebniss der binären Operation op
  - Wenn op eine Operation mit neutralem Element ist und C_1 oder C_2 eben dieses neutrale Element ist, dann ersetzen Sie die Operation durch den jeweils anderen Operanden.
  - Bei logischen Operationen beachten Sie die Fälle false /\ x = false bzw. true \/ x = true
- op C_1 wird ersetzt durch das Ergebniss der unären Operation op

mit Hilfe eines Visitor Patterns (siehe Skript S.54ff).

Die folgenden binären Operatoren müssen behandelt werden: ”+”, ”-”, ”*”, ”/”, ”//”, ”<”, ”>”, ”<=”, ”>=”, ”/\”, ”\/”

Der folgende unäre Operator muss behandelt werden: ”\”

Beachten Sie hierbei dass das Ergebnis einer Operation einen anderen Typ wie das der Operatoren haben kann, wie zum Beispiel bei $1 < 2$ was durch *true* ersetzt werden kann.

Ihre Lösung implementieren Sie in der Klasse *ConstantFolding* welche das Interface *Visitor* implementieren muss. Geben Sie dabei so vor das Ihre Optimierung den veränderten AST als Ergebnis zurückliefert. Wenn Sie dies konsequent durchführen reicht es die unveränderten Elemente einfach zurückzugeben. Zur Implementierung können Sie die Hilfsfunktionen in der Datei *constant_folding_helper_functions.java* verwenden, welche die Erkennung und Erzeugung neuer Konstanten erleichtern soll.

Um Ihre Implementierung zu testen, bauen Sie den Aufruf Ihrer Optimierung in der Klasse *Compiler* nach dem Aufruf des Checkers in der Methode *compileProgram* ein. Vergessen Sie auch nicht zu überprüfen ob die resultierenden Programme mit der Optimierung immer noch das gleiche Verhalten haben.

Aufgabe 2.2 Lösung

Die Musterlösung dieser Aufgabe finden Sie in Form der Datei *ConstantFolding.java* in der Anlage.

Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weitere Infos unter www.informatik.tu-darmstadt.de/plagiarism