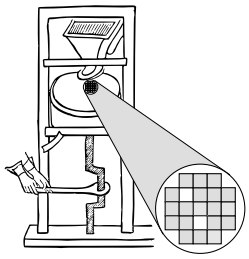
Technische Universität Darmstadt
FG Eingebettete Systeme und
ihre Anwendungen (ESA)

Prof. Dr. Andreas Koch



"Optimierende Compiler"
Aufgabe 1: Interaktiver Compiler und einfache Optimierungsschritte
Abgabe bis zum 17.05.2007, 18:00 Uhr MET DST

Um Erfahrungen im Umgang mit dem DAST zu bekommen, sollen Sie in Ihren Dreiergruppen verschiedene Operationen darauf implementieren. Zunächst gilt es aber, den Triangle-Compiler so zu modifizieren, dass er auch interaktiv bedienbar und somit für die folgenden Experimente besser geeignet ist.

1 Einleitung

In der Praxis interagieren verschiedene Optimierungsverfahren miteinander. So kann beispielsweise nach der Erkennung konstanter Ausdrücke eine Entfernung von "toten" Bedingungen (immer wahr oder falsch) mit größerem Erfolg vorgenommen werden, als vorher. Um diese Untersuchungen zu erleichtern, soll um den Triangle-Compiler herum eine interaktive Shell entwickelt werden, mit der durch zeilenorientierte Kommandos verschiedene Passes aufgerufen und Zwischenergebnisse dargestellt werden können.

Als erste Exkursion in den Bereich der Optimierung sollen Sie dann zwei neue Passes entwickeln. Der erste nimmt die oben angesprochene Evaluation von konstanten (Teil-)Ausdrücken vor, der zweite entfernt konstante Bedingungen.

2 Problemstellung

2.1 Interaktive Oberfläche

Implementieren Sie eine interaktive Oberfläche für den Triangle-Compiler in Form einer Kommandozeile. Aus dieser heraus sollen einzelne Passes dann gezielt durch den Benutzer aufrufbar sein, wobei gegebenenfalls auch noch zusätzliche Parameter auf der Kommandozeile angegeben werden können.

Die Oberfläche soll leicht in einer methodischen Form um weitere Kommandos erweiterbar sein (die im Laufe des praktischen Teils der Veranstaltung von Ihnen entwickelt werden).

Für die schon jetzt im Triangle-Compiler **vorhandenen** Passes sollen dabei folgende Kommandos verwendet werden:

`read` ***filename*** zum Einlesen einer Triangle-Quelldatei (lexikalische und syntaktische Analyse).

`check` zur Durchführung der Kontextanalyse.

`codegen` zur Code-Erzeugung.

`drawast` zur graphischen Ausgabe des ASTs (ja, auch das kann der Compiler jetzt schon).

`write` ***filename*** Ausgabe des TAM-Codes in Datei.

`exit` zum Beenden einer Sitzung.

Eine Beispielsitzung könnte also so aussehen:

```
$ java -cp classes Triangle.Shell
tc> read primes.pri
Syntactic Analysis ...
tc> check
Contextual Analysis ...
tc> codegen
Code Generation ...
tc> write primes.tam
Object file primes.tam written successfully
tc> exit
$
```

Beachten Sie, dass die Kommandos teilweise aufeinander aufbauen und dies auch überprüfen müssen. So ist zum Beispiel ein `check` nur nach einem `read` sinnvoll, ein `codegen` setzt ein erfolgreiches `check` voraus, und ein `write` ein erfolgreiches `codegen`.

2.2 Ausgabe des (D)AST

Der dekorierte AST ist *die* entscheidende Darstellung des aktuellen Programmes im Triangle-Compiler. Er wird durch verschiedene Optimierungsschritte transformiert. Zum Debugging ist es daher essentiell, sich vor/nach jedem Schritt eine entsprechende Anzeige zu verschaffen.

Grundsätzlich kann dabei unterschieden werden zwischen einer Ausgabe des ASTs wieder als gut lesbar formatierter Triangle-Quellcode (allerdings ohne Kommentare, diese wurden ja beim Scannen verworfen), und einer Anzeige der internen Struktur des ASTs (sogenannter *dump*) mit allen Attributen. Für beide Operationen sollen Sie entsprechende Kommandos in der Compiler-Shell implementieren.

Das Kommando `showast` soll den AST wieder textuelll als Triangle-Quellcode ausgeben. Beachten Sie hier die *formatierte* Ausgabe, die auch das korrekte Einrücken verschiedener Blockschachtelungen umfassen soll.

Auch das Kommando `dumpast` soll zu einer Ausgabe des AST führen. Hier sollen allerdings die Knoteninhalte in einer weniger übersichtlichen, aber vollständigen Version (mit fast allen Instanzvariablen, siehe unten) ausgegeben werden.

Beispiel: Der Sub-AST für den Ausdruck `a+1` könnte ausgegeben werden wie in Abbildung 1 vorgeschlagen.

Die konkrete Ausgabe kann durchaus etwas von dem Beispiel abweichen. Wichtig ist, dass Sie sowohl die jeweiligen Sub-ASTs (hier durch weitere Klammerebenen und Einrückungen gekennzeichnet) als auch die Instanzvariablen (hier in der Form *name*:*wert* gezeigt) jedes AST-Knotens ausgeben.

Zur besseren Übersicht sollten Sie allerdings auf die Ausgabe der Instanzvariablen aus den Klassen `RuntimeEntity` und `SourcePosition` verzichten (wie oben im Beispiel geschehen). Beide betreffen Teile des Compilers, die in den praktischen Arbeiten nur peripher behandelt werden (eigentliche Code-Erzeugung und syntaktische Analyse).

Eine solch detailierte Darstellung ist bei der Implementierung und der Fehlersuche in den nächsten beiden Aufgabenteilen sicherlich hilfreich.

```
(BinaryExpression
        E1:(VnameExpression
                V:(SimpleVname variable:true indexed:false offset:0
                        type:(IntTypeDenoter
                        )
                        I:(Identifier spelling:"a"
                                type:(IntTypeDenoter
                                )
                                decl:(VarDeclaration
                                        I:(Identifier spelling:"a"
                                        )
                                        T:(IntTypeDenoter
                                        )
                                )
                        )
                )
        )
        O:(Operator spelling:"+"
                decl:(BinaryOperatorDeclaration duplicated:false
                        O:(Operator spelling:"+"
                        )
                        ARG1:(IntTypeDenoter
                        )
                        ARG2:(IntTypeDenoter
                        )
                        RES:(IntTypeDenoter
                        )
                )
        )
        E2:(IntegerExpression
                type:(IntTypeDenoter
                )
                IL:(IntegerLiteral spelling:"1"
                )
        )
)
```

Abbildung 1: Beispielausgabe von `dumpast` von `a+1`

2.3 Optimierung: Evaluation konstanter Ausdrücke

Mit dem Compiler-Shell-Kommando `constantfold` sollen alle konstanten (Teil-)Ausdrücke berechnet und durch ihren Wert ersetzt werden. Dabei sollen sowohl Literale als auch Bezeichner mit konstantem Wert berücksichtigt werden.

Beispiel: In der Anweisung `a := 4*5+b` sei `b` eine Variable. Der Ausdruck soll nun umgestellt werden zu `20+b`. Falls `b` auch eine Konstante mit dem Wert 3 ist, würde der Ausdruck zu `23` "gefaltet". Beachten Sie dabei, dass ...

- in Triangle *nicht* Punkt- vor Strichrechnung gilt (warum?).
- primitive Funktionen mit konstanten Argumenten aufgelöst werden können, beispielsweise ist `chr(ord('A')+1)='B'`
- Sie alle primitiven Datentypen (Integer, Boolean, Character) berücksichtigen müssen.
- Sie *keine* Anweisungsgrenzen überschreiten dürfen. Im Beispiel oben darf also *nicht* für alle weiteren Verwendungen der Variable `a` der Wert 23 eingesetzt werden!

Ihr neuer Optimierungs-Pass soll auch ausgeben, wieviele Operationen er im ganzen Programm entfernen konnte. Im Beispiel `a := 4*5+b` oben wären das zunächst eine Multiplikation, im erweiterten Fall dann auch noch eine Addition. Bei der Charakter-Transformation würden 1x `chr()`, 1x `ord()` und eine Addition entfernt.

Verwenden Sie zum Testen die auf der Web-Seite bereitgestellten Triangle-Beispiele und geben Sie Statistiken für die einzelnen Eingabedateien aus.

2.4 Optimierung: Umstellen konstanter Bedingungen

Nach Eingabe des Kommandos `elimconstcond` soll der AST durchsucht werden nach allen Verzweigungskonstrukten mit einer konstanten Bedingung. Durch geeignete Transformation soll dann das Konstrukt durch eine gleichwertige AST-Darstellung ohne die überflüssige Verzweigung realisiert werden.

Abbildung 2 und 3 zeigen eine Beispieleingabe und das gewünschte Ergebnis der Optimierung.

Analoges gilt auch für `while`-Schleifen. Geben Sie auch hier Statistiken aus, wieviele `if` und `while`-Konstrukte Sie ersetzen konnten.

Hinweis: Beide Optimierungen dürfen die ausgeführte Funktion des eingegeben Programmes *nicht* beeinflussen. Sie sollten also durch Tests sicherstellen, dass bei den gleichen Eingaben optimierte und unoptimierte Fassung immer noch die gleichen Ausgaben haben!

```
...
if true then
  a := a + 1
else
  a := a + 2
...
```

Abbildung 2: Eingabe der Optimierung

```
...
a := a + 1
...
```

Abbildung 3: Ausgabe der Optimierung

3 Programmierstil

Die von Ihnen erstellten Programme werden in der Endfassung erfahrungsgemäß zwischen 3.000 und 12.000 Zeilen Java umfassen. Um dem Betreuer das Verständnis und Ihnen die Wartung zu erleichtern, sollen Sie von Anfang an einen sauberen und disziplinierten Programmierstil praktizieren.

Bei der Implementierung sind die Konventionen aus *Writing Robust Java Code* weitgehend einzuhalten. Dieses Dokument liegt als PDF auch auf der Web-Seite der Vorlesung. Ergänzend soll folgendes beachtet werden:

- Achten Sie darauf, dass Klassen nicht zu komplex werden (zu viele Instanzvariablen, zu viele Methoden). Bei deutlich mehr als 20 dieser Konstrukte sollten Sie die Klasse aufteilen.
- Analoges gilt für die Komplexität von einzelnen Methoden. Auch hier sollten Sie bei mehr als 100 Programmzeilen Länge die Methode aufteilen.
- Verwenden Sie statt Abfragen von **`instanceof`** echte objekt-orientierte Konstrukte (z.B. polymorphe Methoden).

Der Test und die Abnahme Ihrer Programme wird vom Betreuer auf Linux mit dem SUN Java Development Kit (JDK) Version 1.5/1.6 erfolgen.

4 Dokumentation

Die Lösungen werden nur durch das unten beschriebene **`README`** und die in das Java-Programm eingebetteten JavaDoc-Direktiven und Kommentare dokumentiert. Achten Sie daher darauf, dass Sie von diesen beiden Möglichkeiten ausreichend und aussagekräftig Gebrauch machen!

Kommentare sollen am Anfang jeder von Ihnen modifizierten oder neu erstellten Quelldatei, pro Klasse und pro Instanzvariable und Methode verfasst werden. Bei Verwendung relativ kurzer Methoden und aussagekräftiger Bezeichner können sich Kommentare innerhalb von Methoden auf wenige wirklich wichtige Stellen beschränken. Bei komplizierteren Methoden soll der Ablauf aber durch eine größere Anzahl an aussagekräftige Kommentaren im Methodenrumpf verdeutlicht werden.

Der Dateikopfkommentar muss neben einer allgemeinen Beschreibung auch eine Historie von Änderungen enthalten. Jeder Eintrag in dieser Historie beschreibt unter Angabe von Datum/Uhrzeit und Namen des Autors auf 1-2 Textzeilen die Natur der Änderungen. Alternativ kann hier auch das Log Ihres Versionskontrollsystems (z.B. CVS oder besser SVN) verwendet werden.

Diese Angaben sind für den Betreuer wichtig, damit im Kolloquium die für ein Thema passenden Ansprechpartner gefunden werden!

5 Abgabe

Jede Gruppe schickt spätestens zum Abgabezeitpunkt in einem `.jar`-Archiv alle Dateien ihrer Version des Triangle-Compilers an

mit dem Subject `Abgabe 1 Gruppe` *N*, wobei Ihnen *N* bereits in der Vorlesung mitgeteilt wurde. In dem Archiv sollen nicht nur die eigenen, sondern *alle* (auch unmodifizierten) Quellen des Triangle-Compilers enthalten sein. Ebenso legen Sie eventuell verwendete zusätzliche externe Bibliotheken in Form ihrer jeweiligen `.jar`-Dateien bei (aber siehe Abschnitt 8).

Neben den Java-Quelltexten enthält das Abgabearchiv eine Datei `README.txt`, die enthält

- die Namen der Gruppenmitglieder.
- eine Übersicht über die neuen und geänderten Dateien mit jeweils einer kurzen (eine Zeile reicht) Beschreibung ihrer Funktion.
- Hinweise zur Compilierung der Quellen. Geben Sie eine `javac`-Kommandozeile an bzw. verweisen Sie auf mitgelieferte Makefiles oder ANT Build-Dateien. *Nicht* ausreichend ist ein Hinweis auf eine von Ihnen verwendete IDE (wie Eclipse, NetBeans etc.).
- für alle Beispielprogramme die Statistiken über die durchgeführten Optimierungen.
- Angaben über weitere Bibliotheken (beispielsweise JSAP, log4j, JUnit etc.), die Sie eventuell verwendet haben. Diese Bibliotheken legen Sie bitte dann auch als .jar Dateien in das abgegebene Archiv.

6 Beurteilung

Die Beurteilung dieser ersten Abgabe erfolgt in jedem Fall via E-Mail. Kolloquien finden nur bei Bedarf statt, Vorträge in dieser Phase noch nicht

7 Gruppenarbeit

Vom Arbeitsaufkommen her sind die beiden Optimierungsteilaufgaben sicherlich anspruchsvoller als die Implementierung der Kommandozeilenumgebung und der AST-Ausgabe. Beide sind aber äußerst hilfreich für das Debugging. Es bietet sich daher an, zunächst mit vereinten Kräften zügig diese Hilfsfunktionen zu realisieren (z.B. 1. Mitglied: Shell, 2. Mitglied: `showast`, 3. Mitglied: `dumpast`), damit dann die anspruchsvolleren Teile in Angriff genommen werden können.

Ob hier allerdings zwei Mitglieder `constantfold` und eines `elimconstcond`, oder umgekehrt, bearbeiten soll, hängt stark von der Natur der Gruppe ab. Eine andere Aufteilung wäre, das je ein Mitglied einen der neuen Passes realisiert, während das dritte dann die Tests vornimmt.

Bei den Tests von Optimierungen ist es häufig so, dass auf den ersten Blick alles funktioniert. Aber dann in einem Quelltext ein “Sonderfall” auftaucht, der vom bisherigen Optimierungscode noch nicht oder nur fehlerhaft bearbeitet wird. Unterschätzen Sie also den Testaufwand nicht!

Falls in Ihrer Gruppe eine Situation entstehen sollte, in der einzelne Mitglieder deutlich zuwenig (oder zuviel!) der anfallenden Arbeitslast bewältigen, sprechen Sie den Betreuer bitte *frühzeitig* auf die Problematik an. Nur so kann durch geeignete Maßnahmen in Ihrem Interesse gegengesteuert werden. Nach der Abgabe ist es dafür **zu spät** und Sie tragen die Konsequenzen selber (z.B. wenn sich eines Ihrer Team-Mitglieder wegen seiner Verpflichtungen beim Wasser-Polo nur stark eingeschränkt den Mühen der Programmierung widmen konnte, und Sie daher eine unvollständige Lösung abgeben mussten).

8 Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe einer Lösung zu den Programmierprojekten bestätigen Sie, dass Ihre Gruppe die alleinigen Autoren des neuen Materials bzw. der Änderungen des zur Verfügung gestellten Codes sind. Im Rahmen dieser Veranstaltung dürfen Sie den Code des Triangle-Compilers vom OC07 Web-Site sowie Code-Bibliotheken für nebensächliche Programmfunktionen (Beispiele siehe oben) frei verwenden. Mit anderen Gruppen dürfen Sie sich über grundlegenden Fragen zur Aufgabenstellung austauschen. Detaillierte Lösungsideen dürfen dagegen *nicht vor Abgabe*, Artefakte wie Programm-Code oder Dokumentationsteile *überhaupt nicht* ausgetauscht werden. Bei Unklarheiten zu diesem Thema (z.B. der Verwendung weiterer Software-Tools oder Bibliotheken) sprechen Sie bitte Ihren Betreuer gezielt an.