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

Prof. Dr. Andreas Koch

Embedded Systems & Applications

"Optimierende Compiler"
Aufgabe 1: Interaktiver Compiler und Vereinfachung der Eingabesprache
Abgabe bis zum 04.05.2011, 23:59 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 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.

Zur Vereinfachung der weiteren Projektphasen sollen Sie den Compiler dann so modifizieren, dass er nur eine Untermenge der Sprache Triangle als Eingabe akzeptiert und nicht mehr unterstützte Konstrukte als Fehler ausgibt.

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`**.

2.1.1 Hinweise zur Shell

In erster Linie dient die Shell natürlich dazu, Ihren Compiler interaktiv zu steuern. Je nach Bedarf können einzelne Passes (ggf. mehrfach) aufgerufen oder weggelassen werden. Der von Ihnen entwickelten Compiler wird nach Abgabe von uns getestet, indem zahlreiche, meist kleine Testprogramme kompiliert und ausgeführt werden – jeweils mit und ohne Optimierungen.

Um diese aufwendige Prozedur zu beschleunigen, werden die meisten Tests automatisch von einem speziell dafür entwickelten Skript erledigt. Dieses Skript steuert Ihren Compiler über die Shell und ist daher auf die Einhaltung der geforderten Befehls- und Ausgabeformate angewiesen.

2.1.2 Allgemeine Funktionsweise

Die Shell liest Tastatureingaben und führt die eingegebenen Befehle aus. Statusmeldungen und sonstige Ausgaben werden auf den Bildschirm geschrieben. Das Befehlsformat ist bewusst sehr einfach gehalten:

- Jede eingegebene Zeile entspricht genau einem Befehl.
- Der eigentliche Befehl steht am Anfang der Zeile. Durch je ein Leerzeichen getrennt können ein oder mehrere Parameter folgen.
- Befehle und Parameter bestehen nur aus Buchstaben (a…z, keine Umlaute), Zahlen, Bindestrich, Unterstrich und Punkt. Insbesondere enthalten sie keine Leerzeichen. Es sind also keine Maßnahmen erforderlich, um Sonderzeichen zu “maskieren”.

Falls bei der Bearbeitung eines Befehls ein Fehler auftritt, muss dies deutlich gekennzeichnet werden, z. B. indem der Fehlermeldung der Text **`ERROR:`** vorangestellt und bei Bedarf ein Stack-Trace ausgegeben wird. Schwerwiegende Fehler (z. B. Typfehler bei der Kontextanalyse) können auch dazu führen, dass die Kompilierung ganz abgebrochen und kein Maschinencode erzeugt wird.

2.1.3 Befehlsklassen und Ausgabe

Die zu implementierenden Befehle unterscheiden sich teilweise enorm hinsichtlich ihrer Ausgaben: Während der Befehl **`check`** sich nur bei Fehlern kurz zu Wort meldet, liefert **`dumpast`** (siehe unten) Unmengen Text. Um die Auswertung der Ausgaben zu vereinfachen (und die eigentliche Shell-Ausgabe

übersichtlich zu halten), müssen besonders "geschwätzige" Befehle ihre Ausgabe in Dateien umleiten können. In Bezug auf die Shell lassen sich die zu implementierenden Befehle somit grob in drei Klassen unterteilen:

1. Shell- und Dateioperation: Dabei handelt es sich um die Befehle **`exit`** sowie **`read`** und **`write`**, mit denen Quelltext aus einer Datei gelesen bzw. Maschinencode in eine Datei geschrieben wird. Diese Befehle sollen nur eine kurze Statusmeldung ausgeben (eine Zeile genügt).
2. Analyse und Transformation: Befehle wie **`check`**, **`codegen`** und später **`dvnt`** führen mehr oder weniger aufwendige Berechnungen bzw. Transformationen mit dem gerade geladenen Programm durch. Dabei können kurze Statusmeldungen während oder nach der Bearbeitung durchaus nützlich sein (z. B. eine Statistik über die von **`dvnt`** optimierten Teilausdrücke). Die Ausgabe sollte 20 Zeilen jedoch nicht überschreiten.
3. Dump-Ausgabe: Befehle wie **`showast`** führen selbst keine Transformationen durch, sondern geben "nur" die internen Datenstrukturen zur Kontrolle aus. Da diese Ausgaben sehr umfangreich sein können, akzeptieren diese Befehle einen zusätzlichen Parameter, über den eine Datei angegeben werden kann, in die die Dump-Ausgabe umgeleitet wird (ausgenommen **`drawast`**)

Um das Debuggen zu erleichtern, steht es Ihnen natürlich frei, Ihre Shell geeignet zu erweitern, beispielsweise indem Sie für Ihre eigenen Tests ausführliche Statusmeldungen über einen einen zusätzlichen Parameter (wie z. B.**`-v`**) einschalten.

2.1.4 Befehlsabfolge

Die Befehle können gewöhnlich nicht in beliebiger Reihenfolge ausgeführt werden. Einige Befehle dürfen nur einmal aufgerufen werden, andere auch mehrfach. Eine robuste Shell muss daher erkennen, ob ein bestimmter Befehl zu einem Zeitpunkt erlaubt ist oder nicht. Ihre Shell muss jedoch mindestens folgende Abläufe beherrschen:

1. Nach dem Start werden **`read`** und **`check`** zuerst aufgerufen.
2. Falls das Programm als AST vorliegt (nach **`check`** bzw. **`ssa2ast`**, 3. Aufgabe), können AST-Transformationen und -Dumps ausgeführt sowie Maschinencode erzeugt werden.
3. Falls das Programm als CFG vorliegt (nach **`ast2ssa`**, 2. Aufgabe), können CFG-Transformationen und -Dumps ausgeführt werden.
4. Dumps können mehrfach durchgeführt werden, z. B. nach jeder Transformation.
5. **`codegen`** und **`write`** werden zuletzt aufgerufen.
6. Die Shell kann jederzeit durch **`exit`** beendet werden.

Je nach Art der Passes können weitere Einschränkungen bzw. Vorbedingungen gelten. Diese werden in den jeweiligen Aufgabenstellungen genauer festgelegt. Sie können vereinfachend davon ausgehen, dass pro Shell-Aufruf nur eine Quelltextdatei bearbeitet wird und dass nach **`codegen`** keine Transformationen oder Dumps mehr durchgeführt werden. Bei der Durchführung eines Befehls kann sich herausstellen, dass die Kompilierung wegen eines Fehlers abgebrochen werden muss (z. B. wegen Syntaxfehlern im Triangle-Programm, unzulässiger Befehlsabfolge in der Shell oder gar einem internen Fehler im Compiler). Ihre Shell muss diese Situation erkennen und die Ausführung weiterer Befehle ggf. verweigern. Je nach Schwere des Fehlers ist auch denkbar, dass sich Ihre Shell selbst beendet (natürlich mit einer aussagekräftigen Fehlermeldung und bei Bedarf mit Stack-Trace).

2.1.5 Hinweise zur Implementierung

Die Shell liest Befehle vom Eingabestrom stdin (**`System.in`** in Java) und schreibt Statusmeldungen auf den Ausgabestrom stdout (**`System.out`**) bzw. stderr (**`System.err`**). Bei einem "normalen", d. h.

interaktiven Aufruf der Shell werden stdin und stdout letzten Endes mit Tastatur bzw. Bildschirm verbunden. Unser Testskript klinkt sich an dieser Stelle ein und sendet selbst Befehle nach stdin bzw. liest und verarbeitet Statusmeldungen von stdout und stderr. Es ersetzt damit Tastatur und Bildschirm. Im Gegensatz zum "echten" interaktiven Aufruf sind einige Besonderheiten zu beachten: Das Testskript "tippt" schneller: Alle auszuführenden Befehle werden auf einmal an die Shell gesendet. Daher kann es passieren, dass ...

- ein Aufruf von `System.in.read()` möglicherweise mehrere Befehle enthält. Beim interaktiven Aufruf ist stdin üblicherweise zeilengepuffert und `read()` liefert immer nur eine Zeile. Tipp: Verwenden Sie `java.io.BufferedReader.readLine()` in Verbindung mit `java.io.InputStreamReader` zum Lesen der Befehle.
- sich noch zusätzliche, jedoch nicht mehr durchführbare Befehle im Eingabepuffer befinden, nachdem die Kompilierung wegen eines Fehlers abgebrochen wurde.
- Der Eingabestrom ist endlich, d. h. irgendwann wird beim Lesen das "Dateiende" (EOF) erreicht. Die Shell muss auch in diesem Fall "sauber" beendet werden und darf z. B. nicht in einer Endlosschleife hängen bleiben.

2.1.6 Abweichungen und Erweiterungen

Die hier beschriebenen Anforderungen sind nur die *Minimalanforderungen*. Ihre Shell darf natürlich schöner, größer, besser sein (was auch zur Notenverbesserung führen kann). Bitte vergessen Sie dabei nicht, alle unterstützten Befehle Ihrer Shell in der Readme-Datei zu dokumentieren (einschließlich der Vorbedingungen) und Abweichungen ggf. mit dem Betreuer abzusprechen.

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 textuell 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.

Beide Kommandos sollen einen optionalen Parameter akzeptieren, der die Ausgabe in eine Datei gleichen Namens erfolgen lässt.

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

```
(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**

Code-Erzeugung und syntaktische Analyse).

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

2.3 Änderungen der Eingabesprache

In dieser Phase sollen zum einen einige Misfeatures von Triangle korrigiert werden, zum anderen die akzeptierte Sprache weiter beschränkt werden. Letzteres soll den Implementierungsaufwand für die nachfolgenden Optimierungsverfahren reduzieren.

Folgende Änderungen sollen allgemein an Sprache und Semantik vorgenommen werden:

- Es soll Punkt- vor Strichrechnung beachtet werden. Wie in der Vorlesung beschrieben erfordert dies Änderungen an der Grammatik und damit auch des Parsers. Der Modulus-Operator `//` soll wie die Division binden.
- *Optional* können Sie für alle Operatoren (also auch Vergleiche, boolesche Operatoren etc.) die üblichen Präzedenzregeln implementieren (diese können z.B. der Java-Spezifikation entnommen werden).
- Der Typ für Zeichen soll nicht mehr **`Char`** sondern **`Character`** heissen (im gleichen Stil wie **`Boolean`** und **`Integer`**).
- Es sollen auch noch die Zeichen Unterstrich (_) und Dollar (**`$`**) in Bezeichnern erlaubt sein.

Hinweis: Der Triangle-Compiler akzeptiert derzeit keinen unären Minus-Operator. Dieser wird zwar korrekt geparsed, führt aber in der Kontextanalysephase zu einem Fehler. In der letzten Aufgabe wird Ihnen die Möglichkeit gegeben werden, auch diesen Operator korrekt einzubauen. Dazu müssen Sie allerdings leicht die Code-Erzeugung modifizieren, die in der Vorlesung erst gegen Ende behandelt wird.

Der Compiler soll des weiteren so modifiziert werden, dass er folgende Forderungen an die eingegebenen Programme stellt und in anderen Fällen einen Fehler ausgibt.

- Zugriffe auf nicht-lokale Variablen oder Konstanten sind nicht mehr erlaubt. Prozeduren können nur noch auf ihre eigenen lokalen Variablen und Konstanten, ihre formalen Parameter sowie auf globale Konstanten zugreifen. Damit sind auch globale Variablen illegal.
- Keine Funktionen- oder Prozeduren als formale Parameter (*closures*).
- Funktions- und Prozedurdefinitionen dürfen nicht geschachtelt sein.
- **`let`**-Blöcke dürfen nur noch (müssen aber nicht) direkt unterhalb von **`Program`** und **`ProcDeclaration`**-Knoten auftauchen. Damit haben wir eine Sprache mit flacher Blockstruktur erzwungen.
- Keine **`let`** und **`if`**-Expressions (wohlgemerkt: **`if`**-Commands sollen natürlich noch erlaubt sein!).
- Konstanten dürfen nicht mehr von Variablen abhängen,

Es gibt mehrere Möglichkeiten, diese Restriktionen zum implementieren. Die einfachste ist sicherlich ein zusätzlicher Pass nach der kontextuellen Analyse **`Checker.check(theAST)`**. Dieser könnte dann den DAST durchgehen und bei Auftreten der jetzt illegal gewordenen Konstrukte eine aussagekräftige Fehlermeldung ausgeben. Andere Möglichkeiten würden eher eingreifen (z.B. schon im Parser), ziehen aber einen erhöhten Implementierungsaufwand nach sich.

Ihre Modifikationen können Sie anhand der bereitgestellten Sammlung von Triangle-Beispielprogramme überprüfen. Diese Programme können den Ausgangspunkt für Ihre eigene Suite von Testprogrammen für den Compiler bilden. Sie werden für die folgenden Aufgaben darüberhinaus noch gezielt Testprogramme für bestimmte Optimierungen entwickeln müssen, dafür sind die Beispielprogramme oftmals schon zu komplex oder teilweise ungeeignet. Die Triangle-Quellen der Testprogramme dürfen Sie dann auch zwischen den Gruppen austauschen (z.B. im Forum der Veranstaltung).

3 Programmierstil

Die von Ihnen erstellten Programme werden in der Endfassung erfahrungsgemäß zwischen 7.000 und 18.000 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 1.6 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

Grundsätzlich schickt jede Gruppe spätestens zum Abgabetermin in einem **`.jar`**- oder **`.zip`**-Archiv die Quelldateien Ihrer Version des Triangle-Compilers an

`oc@esa.informatik.tu-darmstadt.de`

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).

Wichtig: Um unseren Testaufwand überschaubar zu halten, lesen Sie bitte die letzten drei Absätze nocheinmal. Sie sollen abgeben:

- Die *vollständigen* Quellen (also die `.java`-Dateien!)
- Aber *nur diese*, nicht noch irgendwelche Altdaten Ihrer eigenen Testläufe (soweit nicht explizit in der Aufgabenstellung angefordert). Räumen Sie also Ihre Verzeichnisse auf, bevor Sie sie in ein Archiv packen!
- Die eventuell benötigten Fremdbibliotheken
- Alles zusammen in einem `.jar` oder `.zip`-Archiv. Also kein `.7z`, `.tar.gz` oder `.tbz`

Daneben enthält das Abgabearchiv eine Datei `README.txt`, die enthält

- die Namen der Gruppenmitglieder und die jeweils bearbeiteten Themen
- 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.).
- 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.

7 Gruppenarbeit

Vom Arbeitsaufkommen her ist die Änderung der Eingabesprache (Beschränkung, Punkt- vor Strichrechung, etc.) 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. 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 zu wenig (oder zu viel!) 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 von der FG ESA Web-Seite zu Optimierende Compiler 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.