Lehrstuhl für Informatik 2, FAU
uebersetzerbau@i2.cs.fau.de

Übungsblatt 11

Aufgabe 11.1: Code-Generierung mittels Dynamischer Programmierung

Es sei eine Zielarchitektur mit den zwei Allzweckregistern `r[12]` sowie den folgenden Operationen gegeben (r_i bezeichne ein Register, M_x den Speicherplatz an der symbolischen Adresse `x`, `c` eine ganzzahlige Konstante):

Name	Operation	Semantik
L	$r_i := M_x$	lade Wert von Speicherstelle M_x in Register r_i
S	$M_x := r_i$	speichere Wert von Register r_i an Speicherstelle M_x
M	$r_i := r_j$	kopiere Wert von Register r_j in Register r_i
C	$r_i := c$	lade Konstante `c` in Register r_i
RoR	$r_i := r_i \circ r_j$	verknüpfe die Werte in den Registern r_i und r_j mit dem arithmetischen Operator $\circ$ und speichere das Ergebnis in r_i
RoM	$r_i := r_i \circ M_x$	verknüpfe den Wert in Register r_i mit dem Wert an Speicherstelle M_x mit dem arithmetischen Operator $\circ$ und speichere das Ergebnis in r_i

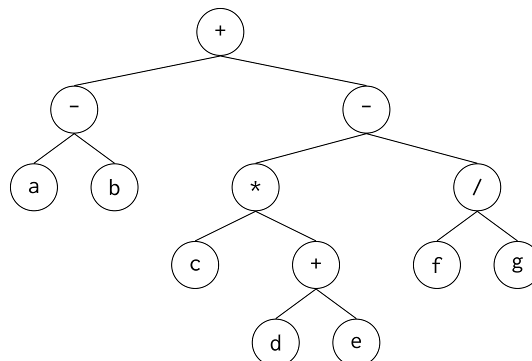
Wenden Sie das aus der Vorlesung bekannte Verfahren zur Code-Generierung mittels *Dynamischer Programmierung* an, um Code für das folgende Programmfragment in Zwischencodedarstellung zu erzeugen:

```
%v0 = add b, c
%v1 = div a, %v0
%v2 = mul e, f
%v3 = sub d, %v2
g = mul %v1, %v3  // g sei nach dieser Instruktion lebendig
```

Nehmen Sie dabei ein *uniformes Kostenmaß* an.

Aufgabe 11.2: Code-Generierung mittels Baumtransformationen

Es sei dieselbe Zielarchitektur wie in Aufgabe 11.1 gegeben. Erzeugen Sie mit Hilfe des aus der Vorlesung bekannten *Baumtransformationsverfahrens* Code für den folgenden Ausdruck:



Aufgabe 11.3: Code-Generierung mit dem Verfahren von Graham & Glanville

Es sei dieselbe Zielarchitektur wie in Aufgabe 11.1 gegeben. Erzeugen Sie mit Hilfe des aus der Vorlesung bekannten Verfahrens von *Graham & Glanville* Code für den Ausdruck aus Aufgabe 11.2.

Geben Sie dazu zunächst eine *Maschinengrammatik* für obige Architektur an.

Projektübung 7 (Abgabe bis 08.02.2018 (fest!))

Im Rahmen dieser (letzten ☺) Projektübung erweitern Sie Ihren e2-Übersetzer um die Code-Generierung für x86-64/Linux. Dabei wird der in einer vorherigen Projektübung erzeugte Zwischencode in menschenlesbaren Assembler-Code[1] übersetzt. Dieser wird anschließend mit den Standardwerkzeugen von Linux assembliert und zusammen mit der zu schreibenden Standardbibliothek zu einem ausführbaren Programm gebunden.

Hinweis: Wie auch schon in der vorherigen Projektübung müssen Sie in dieser Aufgabe *keine* Unterstützung für Fließkommaberechnungen implementieren.

a) Implementieren Sie zunächst die Funktionen der e2-Standardbibliothek im Unterordner `runtime_library/`:

- Schreiben Sie ein Modul `startup.s`, das die `_start`-Funktion beinhaltet. Diese soll die `main`-Funktion aus dem Eingabeprogramm aufrufen und deren Rückgabewert mittels `exit`-Systemaufruf an das Betriebssystem zurückliefern.
- Die Funktionen `readChar()`, `writeChar()`, `time()` und `exit()` können nicht in e2 selbst implementiert werden und müssen deshalb in Assembler-Code geschrieben werden. Implementieren Sie diese in einem weiteren Modul `library_functions.s`. Eine exemplarische Implementierung von `time()` finden Sie auf der Internetseite zur Übung.
- Für die Funktionen `readInt()` und `writeInt()` haben Sie zwei Möglichkeiten:
  - Sie implementieren diese wie `readChar()` und `writeChar()` in Assembler-Code.
  - Sie implementieren diese unter Verwendung von `readChar()` und `writeChar()` in e2 und übersetzen sie mit Hilfe Ihres Übersetzers (sobald dieser in der Lage ist, Code zu generieren). Die Namensanalyse in Ihrem Übersetzer gibt (hoffentlich) eine Fehlermeldung aus, falls das Eingabeprogramm Funktionen der Standardbibliothek zu redefinieren versucht. Fügen Sie deshalb eine weitere Kommandozeilenoption „`--runtime`“ zu Ihrem Übersetzer hinzu und sorgen Sie dafür, dass bei gesetzer Option eine Definition von `readInt()` und `writeInt()` erlaubt ist. Eine exemplarische Implementierung von `writeInt()` in e2 finden Sie auf der Internetseite zur Übung.

  Tipp: Wählen Sie die zweite Option ☺

  Hinweis: Um die Implementierung zu vereinfachen, konsumiert `readInt()` auch das Zeichen nach der eingelesenen Zahl. Wird die Funktion beispielsweise auf eine Eingabe „`13ab`“ angewendet, so würde ein nachfolgender Aufruf von `readChar()` das Zeichen `b` zurückliefern.

Wichtig: Achten Sie darauf, dass die in Assembler-Code geschriebenen Funktionen dieselbe *Aufrufkonvention* wie die von Ihrem Übersetzer generierten Funktionen verwenden.

b) Fügen Sie im Ordner `runtime_library/` ein `Makefile` hinzu, mit dem die Bibliotheksfunktionen übersetzt bzw. assembliert werden können. Sorgen Sie außerdem dafür, dass zum Schluss aus allen Objektmodulen der Standardbibliothek ein Archiv `e2lib.a` erzeugt wird, das später gegen das Eingabeprogramm gebunden wird. Verwenden Sie dazu das Werkzeug `ar`: „`ar crs output.a foo.o bar.o`“ erzeugt beispielsweise aus den beiden Modulen `foo.o` und `bar.o` das Archiv `output.a`.

mehr auf der nächsten Seite

[1] Es ist Ihnen überlassen, ob der von Ihnen geschriebene bzw. erzeugte Code die Intel- oder die AT&T-Syntax verwendet.

c) Beim Übersetzen des e2-Übersetzers mittels Gradle soll auch die Standardbibliothek gebaut werden. Fügen Sie dazu die folgenden Zeilen zu der Datei `build.gradle` hinzu:

```
// builds the e2 runtime library (if necessary)
task buildLibrary(type : Exec) {
  def subdir = 'runtime_library'

  workingDir(subdir)

  inputs.files("${subdir}/Makefile",
      new FileNameByRegexFinder().getFileNames(subdir, '/.*\\.s|e2$'))
  outputs.file("${subdir}/e2lib.a")

  commandLine 'make'
}
build.dependsOn(buildLibrary)
```

d) Die Übersetzung nach Assembler-Code soll mit Hilfe eines Zwischencodebesuchers erfolgen. Dieser verwendet die in der vorherigen Projektübung implementierte Speicherplatzvergabe und schreibt den erzeugten Assembler-Code in eine Ausgabedatei. Denken Sie auch daran, die globalen Variablen des Eingabeprogramms abzubilden (Tipp: `.lcomm`-Direktive).

Tipp 1: Stellen Sie für eine einfachere Fehlersuche den Assembler-Instruktionen die jeweils zugehörige Instruktion im Zwischencode als Kommentar (`#`) voran.

Tipp 2: Wenn Sie sich unsicher sind, wie die Übersetzung eines bestimmten Code-Fragments aussehen sollte, schreiben Sie ein entsprechendes C-Fragment und betrachten Sie die Ausgabe eines C-Übersetzers.

e) Die Vorlage beinhaltet bereits Teile einer Infrastruktur, um die Code-Generierung anzustoßen und anschließend den Assemblierer und Binder des Systems aufzurufen. Erweitern Sie diese wie folgt:

- Erstellen Sie eine Klasse `e2c.backend.codegen.x64.BackendX64`, die von der abstrakten Klasse `e2c.backend.Backend` erbt und die fehlenden Methoden implementiert:
  - `generateCode()` soll den implementierten Zwischencodebesucher so aufrufen, dass dieser den erzeugten Assembler-Code in die Datei mit dem übergebenen Dateinamen schreibt.
  - `runAssembler()` und `runLinker()` sollen den Assemblierer `as` bzw. den Binder `ld` geeignet aufrufen. Die Klasse `FileUtil` stellt die Methode `exec()` zum Ausführen externer Anwendungen bereit.
- „Registrieren“ Sie das implementierte Backend in `e2c.backend.BackendSelector` wie folgt:

```
public enum BackendSelector {

  BACKEND_X64("x64", ".s", ".o", "") {

    @Override
    public final Backend createBackend() {
      return new e2c.backend.codegen.x64.BackendX64();
    }

  };

  // ...
}
```

Wenn Sie Ihren Übersetzer nun ausführen, sollte die Code-Generierung angestoßen sowie Assemblierer und Binder ausgeführt werden. Sie können nun die folgenden Kommandozeilenoptionen verwenden:

- `-o <Dateiname>`: Setzt den Dateinamen der Ausgabedatei auf `<Dateiname>`. Standardmäßig entspricht der Name der Ausgabedatei dem der Eingabedatei mit entfernter Dateiendung.

- `-S`: Stoppt die Übersetzung vor dem Aufruf des Assemblierers. Falls die Option „`-o`“ *nicht* gesetzt ist, entspricht der Name der Ausgabedatei dem der Eingabedatei mit ersetzter Dateiendung (`*.s`).
- `-c`: Stoppt die Übersetzung vor dem Aufruf des Binders. Falls die Option „`-o`“ *nicht* gesetzt ist, entspricht der Name der Ausgabedatei dem der Eingabedatei mit ersetzter Dateiendung (`*.o`).
- `--keep`: Bei gesetzter Option werden die Zwischenergebnisse der Übersetzung (generierter Assembler-Code, erzeugtes Objektmodul) *nicht* entfernt.

Ihr Übersetzer sollte nun in der Lage sein, ausführbare Programme zu erzeugen ☺

Testen Sie Ihre Implementierung mit Hilfe der zur Verfügung gestellten Beispielprogramme (Unterordner `examples/`) sowie den Testfällen zum fünften Meilenstein (siehe unten).

Meilenstein 5 (Abgabe bis 08.02.2018 (fest!))

Stellen Sie sicher, dass Ihr Übersetzer alle zum fünften Meilenstein gehörenden Testfälle korrekt behandelt, indem Sie den Gradle-Task „`milestone5`“ ausführen und kontrollieren, dass am Ende die Ausgabe „`BUILD SUCCESSFUL`“ erscheint. **Hinweis**: Die Ausführung der Testfälle bricht (anders als bei den vorherigen Aufgaben) beim ersten fehlgeschlagenen Testfall ab; verwenden Sie die zusätzliche Kommandozeilenoption „`--continue`“, damit Gradle bei einem Fehler auch die retlichen Testfälle ausführt. Versehen Sie den finalen Stand mittels „`git tag`“ mit der Markierung „`milestone5`“ und pushen Sie Ihre Änderungen auf den `git`-Server des Lehrstuhls.