Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
CAU Kiel, Institut für Informatik
Prof. Dr. M. Hanus

24118 Kiel
Christan-Albrechts-Platz 4
Tel.: 0431 / 880-7271

12. Übung „Übersetzerbau“

Bearbeitung bis zum 8. Juli 2008

Bitte senden Sie die Programmieraufgaben zusätzlich zur Abgabe in ausgedruckter Form auch per Email an Axel Stronzik (`axs@informatik.uni-kiel.de`)!

Aufgabe 35

In der Vorlesung wurde die Funktion `munch` zur Erzeugung lokal optimalen Zielcodes skizziert. Hierbei wurde jedoch auf die Verwendung konkreter Register verzichtet. Verfeinern Sie die Funktion `munch`, so dass alle Zielcodebefehle die verwendeten Register als Parameter erhalten.
Gehen Sie zunächst davon aus, dass beliebig viele Register zur Verfügung stehen. Versuchen Sie aber dennoch mit möglichst wenigen Registern bei der Übersetzung auszukommen (Wiederverwendung von Registern).

Aufgabe 36

Führen Sie die in der Vorlesung vorgestellte Lebendigkeitsanalyse für folgendes Programm durch:

```
var i,x,y,z,o;

begin
  i := read(); % Einlesen von z
  z := i;
  y := 10;
  while y>0 do
    begin
      y := y-1;
      z := z*2;
    end;
  x := z;
  if z<>0 then
    x := z*2;
  o := x+1;
  print (o);  % Ausgabe von o
```

Erzeugen Sie zunächst einen Flussgraphen für dieses Programm und wenden Sie dann die in der Vorlesung vorgestellte Lifeness-Analyse an.
Mit wievielen Registern käme man bei der Übersetzung dieses Programms aus?
Wie verteilen sich die Variablen auf die Register?

Aufgabe 37
Wir erweitern den Zielcode um die Befehle „`JMP` m“ und „`JMC` r m“, wobei $m \in \mathbb{N}$ die Programmadresse ist, zu der ggf. verzweigt werden soll. Beim bedingten Sprung `JMC` verzweigt das Programm nur, falls das Register r nicht den Wert Null enthält. Sonst wird die Ausführung mit dem nächsten Befehl fortgesetzt.
Erweitern Sie die Funktion `munch` für alle möglichen Konstrukte der Ausdrucksbäume außer `CALL`. Gehen Sie von einer Basisblockdarstellung aus, bei der die Basisblöcke bereits gemäß der 3. Phase umgeordnet sind.

Aufgabe 38
Vervollständigen Sie einen Ihrer MPS-Parser (ohne Prozeduren), so dass er direkt Zielcode erzeugt. Ihr Compiler soll das Quellprogramm aus einer Datei einlesen und das Compilat in eine andere Datei schreiben. Verwenden Sie die Funktionen `readFile` und `writeFile` zum Lesen und Schreiben von Dateien:

```
compile inFile outFile = do
   str <- readFile inFile
   writeFile outFile (show (translate (parse (scan str))))
```

Hierbei soll die Funktion `translate` den Syntaxbaum in Zielcode übersetzen, welcher dann mittels `show` in einen String umgewandelt und in eine Datei mit Namen `outFile` geschrieben wird.