Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 14/15
Aufgabenblatt 3

Abgabemodalitäten

Gruppenarbeit ist erlaubt und erwünscht. Bitte geben Sie dann nur eine Lösung pro Gruppe ab. Ihre Lösungen reichen Sie als PDF per E-Mail bis zum 25.01.2015 um 23:59 MET an oc@esa.informatik.tu-darmstadt.de ein. Die E-Mail muss als Betreff "Compiler 1 Aufgabe 3" haben.

Aufgabe 3.1 Allgemeine Fragen

a) Beschreiben Sie kurz die Aufgabe des Routinenprotokolls. Begründen Sie auch warum ein Routinenprotokoll notwendig ist. Warum gibt es auf unterschiedlichen System verschiedene Routinenprotokolle?

b) Wieso ist es notwendig, einen Stack und einen Heap in der Speicherverwaltung zu verwenden? Beschreiben Sie hierzu auch kurz welche Daten im Stack bzw. Heap gespeichert werden.

c) Wofür wird der dynamische bzw. statische Link innerhalb der Speicherverwaltung verwendet?

d) Warum ist der Zugriff auf nicht-lokale Variablen teuer? Wie kann man den Zugriff durch zusätzliche Hardware beschleunigen?

Aufgabe 3.2 TAM

Beschreiben Sie die Semantik der Instruktionen in den folgenden TAM-Instruktionsschnipseln:

a)
```
...
LOAD (1) -3[LB]
LOAD (1)  4[L1]
CALL (SB) 8[PB]
...
```

Beschreiben Sie möglichst genau, welche Operanden die Operation verwendet.

b)
```
...
19:  LOADL        42
20:  CALL         gt
21:  JUMPIF(0)    24[CB]
22:  CALL   (L2)  3[CB]
23:  JUMP         25[CB]
24:  CALL   (L1)  9[CB]
25:  ...
```

Nehmen Sie an, dass der Stack anfangs nicht leer ist. Beschreiben Sie möglichst genau, wie der Ablauf innerhalb dieses Schnipsels ist.

Aufgabe 3.3 Ausdrucksauswertung

a) Geben Sie für die folgenden Ausdrücke die zur Auswertung notwendigen Instruktionen in der richtigen Reihenfolge an. Gehen Sie analog zu den Folien 32 ff. des 4. Vorlesungsblocks vor. Optimieren Sie, falls möglich, in Richtung einer möglichst kurzen Instruktionssequenz. Erlaubt sind alle Operationen von Folie 33.

1. $x := ((b + c) * a + b * c) * 2$
2. $y := 11 + a * (7 + a * (5 + a * (4 + a * 2)))$

b) Gegeben sei eine (imaginäre) Registermaschine mit unendlich vielen Registern, die Ihnen folgende Instruktionen zur Verfügung stellt:

`r = loadc imm`	Lädt die Konstante `imm` in das Register `r`
`r = load addr` `store r, addr`	Lädt den Speicherwert an der Adresse `addr` in das Register `r` Schreibt den Inhalt von Register `r` an die Adresse `addr`

`r = add r1, r2`
`r = sub r1, r2`
`r = mul r1, r2`
	Addiert `r1` und `r2` und schreibt das Ergebnis in Register `r`
	Subtrahiert `r1` und `r2` und schreibt das Ergebnis in Register `r`
	Multipliziert `r1` und `r2` und schreibt das Ergebnis in Register `r`
	Das Ergebnis- und die Zielregister dürfen gleich sein.

1. Erzeugen Sie nun für den ersten Ausdruck aus der vorherigen Teilaufgabe eine möglichst kurze Instruktionssequenz.
2. Erzeugen Sie für eben diesen Ausdruck ein Instruktionssequenz, die mit der minimal erforderlichen Anzahl an Registern auskommt. *Hinweis:* Informieren Sie sich über die "Ershov-Zahlen".

Aufgabe 3.4 Statische Speicherverwaltung

Nachfolgend finden Sie die zur Definition eines Go-Spiels notwendigen Datentypen. Geben Sie für dieses Programm die Speicherorganisation der Variable *state* gemäß Folie 40 des 4. Vorlesungsblocks an. Ihre Lösung sollte für jeden Wort-Index des Speichers, beginnend ab der fiktiven Anfangsadresse 0, dessen benutzerdefinierten- (Player, Point,...) und Basistyp (Integer, Boolean) angeben. Markieren Sie auch welche Worte zu welchem Record bzw. Array gehören. Die Arrayelemente 1..359 können sich durch "..." abkürzen.

```
let
  type Player ~ record
    number : Integer
  end;
  type Point ~ record
    empty : Boolean,
    owner : Player
  end;
  type Board ~ record
    board : array 361 of Point
  end;
  type State ~ record
    moves : Integer
    next : Player,
    board : Board,
  end;
  var state : State
in
  ...
```

Listing 1: Beispiel zur statischen Speicherverwaltung

Aufgabe 3.5 Routinenprotokoll

Das nachfolgende Programm berechnet den größten gemeinsamen Teiler zweier Zahlen mittels des euklidischen Algorithmus. Zeichnen Sie hierfür Stack Frames (also die Argumente, Resultate und Verwaltungsinfos) vor Eintritt in eine Funktion und nach Austritt aus einer Funktion für die Auswertung des Ausdrucks $ggT(12, 6)$.

```
let
  proc ggT(a: Integer, b: Integer, var r: Integer) ~
  let
    proc mod(n: Integer, m: Integer, var q: Integer) ~
      begin
        if n < m then
          q := n
        else if n = m then
          q := 0
        else
          mod(n − m, m, var q)
      end ;

    var module: Integer
  in
    begin
      if b = 0 then
        r := a
      else
        begin
          mod(a, b, var module);
          ggT(b, module, var r)
        end
    end ;

  var result : Integer
in
  begin
    ggT(12, 6, var result);
    putint(result)
  end
```

Listing 2: Beispielprogramm zum Routinenprotokoll zu Aufgabe 35a

Aufgabe 3.6 Routinen als Parameter

Triangle bietet die Möglichkeit, auch Funktionen oder Prozeduren als Parameter bei einem Aufruf zu verwenden. Hierzu wird ein Paar bestehend aus Static Link und der Anfangsadresse der Routine übergeben. Dieses Paar nennt man Closure.

Untersuchen Sie das folgende Programm mit Hilfe des Triangle Disassemblers. Kommentieren Sie im disassemblierten Code die Position, an welcher die Clossure erzeugt wird und wie die parametrisierte Funktion aufgerufen wird. Geben Sie auch an, wo die einzelnen Routinen beginnen.

```
let
  func twice(func doit(x : Integer): Integer, i : Integer): Integer ~ doit(doit(i));
  func double(d : Integer): Integer ~ d*2;
  var x: Integer
in begin
  x := twice(func double, 10);
  putint(x)
end
```