Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 14/15
Aufgabenblatt 3 - Lösungsvorschlag

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?
Das Routinenprotokoll (oder auch engl. "calling conventions") legt fest, wie die Funktionsparameter und der Rückgabewert übergeben werden. Die Parameter können beispielsweise über den Stack, in Registern oder als Mischform der Varianten übergeben werden. Natürlich muss auch die Reihenfolge der Parameter auf dem Stack, bzw. die Zuordnung von Parameternummer zu Register festgelegt werden.

Zusätzlich bestimmt es, ob der Aufrufer (caller) oder Aufgerufene (callee) Register sichert und wer den Stack und die Register bei Routinenende aufräumt.

Die Einhaltung des Routinenprotokolls ist inbesondere dann wichtig, wenn der Compiler nicht das vollständige Programm übersetzt, sondern beispielsweise Funktionen aus einer vorab kompilierten Bibliothek aufrufen muss. (Ein Compiler kann aber als Optimierung für Funktionen, die nicht extern sichtbar sind, andere calling conventions benutzen.)

In der Praxis gibt es mehr als ein Routinenprotokoll, weil je nach Plattform andere Kompromisse eingegangen werden müssen. Die Parameterübergabe in Registern ist schneller als die Übergabe über den Stack, jedoch müssen diese Register exklusiv für diesen Zweck reserviert werden und fehlen deshalb für die übrigen Berechnungen.

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.
Der Stack wird dazu verwendet alle statisch allozierbaren Variablen zu speichern, die nur bis zum Ende der deklarierenden Routine gebraucht werden. Dabei ist aber nur die Größe dieses Bereichs für jede einzelne Routine immer die gleiche. Die Größe des Stacks ist nicht statisch, da die Reihenfolge der Aufrufe während des Programmablaufs dynamisch varieren kann. Da die TAM eine Stackmaschine ist liegen auch alle Operanden auf dem Stack. Der Heap hingegen wird verwendet um Datenstrukturen abzuspeichern, deren Größe zur Übersetzungszeit nicht bekannt ist und die unabhängig von der Aufrufhierachie weiterbestehen sollen, wie z.b. verkettete Listen.

c) Wofür wird der dynamische bzw. statische Link innerhalb der Speicherverwaltung verwendet?
Der dynamic link zeigt auf den Beginn des Stackframes der aufrufenden Routine. Dies ist notwendig, um nach dem Zurückkehren aus der aufgerufenen Routine das LB-Register wieder auf das Stackframe des Aufrufers zurückzusetzen.

Der static link hingegen ist ein Verweis auf den Stackframe des letzten Aufrufs der (im Quelltext) umgebenden Routine, welcher verwendet wird, um auf nicht-lokale Variablen zuzugreifen.

d) Warum ist der Zugriff auf nicht-lokale Variablen teuer? Wie kann man den Zugriff durch zusätzliche Hardware beschleunigen?
Weil dazu (gegebenfalls mehrfach) der static link verfolgt werden muss. Abhilfe schafft der Einsatz sogenannter Display-Register, die für eine kleine Anzahl an Schachtelungstiefen den Wert des static links direkt speichern.

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.
Addition des drittletzten Funktionsparameters mit der zweiten lokalen Variable der umgebenden Funktion.

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.
Es wird getestet, ob der Wert an der Spitze des Stacks größer als 42 ist. Ist das der Fall, so liegt eine 1 auf dem Stack; dies ist ungleich der Sprungbedingung (`JUMPIF(0)`), und wir führen einfach die nächste Instruktion aus, nämlich den CALL, der die bei 3[CB] beginnende Prozedur aufgeruft, die 2 Schachtelungsebenen außerhalb der aktuellen Prozedur liegt.

Ansonsten wird der Sprung zu 24[CB] genommen und die Prozedur bei 9[CB] aufgerufen, die eine Schachtelungsebene außerhalb der aktuellen Prozedur liegt.

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 3.3 Lösung

a) 1.

```
LOAD b
LOAD c
ADD
LOAD a
MUL
LOAD b
LOAD c
MUL
ADD
LOADL 2
MUL
STORE x
```

2.

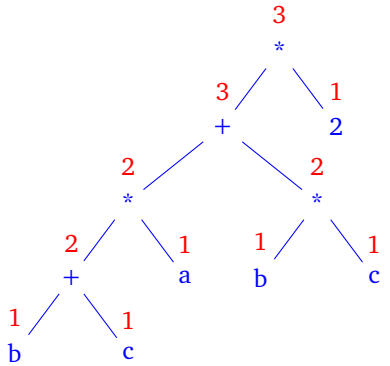
```
LOADL 2
LOAD a
MUL
LOADL 4
ADD
LOAD a
MUL
LOADL 5
ADD
LOAD a
MUL
LOADL 7
ADD
LOAD a
MUL
LOADL 11
ADD
STORE y
```

b) 1.

```
r1 = load a
r2 = load b
r3 = load c
r4 = loadc 2
r5 = add r2, r3
r6 = mul r5, r1
r7 = mul r2, r3
r8 = add r6, r7
r9 = mul r8, r4
store r9, x
```

2. Im Ausdrucksbaum rechts sind die Ershov-Zahlen rot dargestellt. Man benötigt also mindestens 3 Register.

```
r1 = load b
r2 = load c
r1 = add r1, r2
r2 = load a
r1 = mul r1, r2
r2 = load b
r3 = load c
r2 = mul r2, r3
r1 = add r1, r2
r2 = loadc 2
r1 = mul r1, r2
store r1, x
```



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 3.4 Lösung

Adresse	Variablenname	Typ	Strukturhierarchie
0	moves	Integer	
1	next.number	Integer	
2	board.board[0].empty	Boolean	
3	board.board[0].owner.number	Integer	
...	...		
722	board.board[360].empty	Boolean	
723	board.board[360].owner.number	Integer	

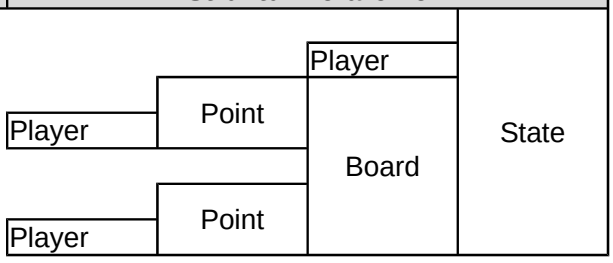


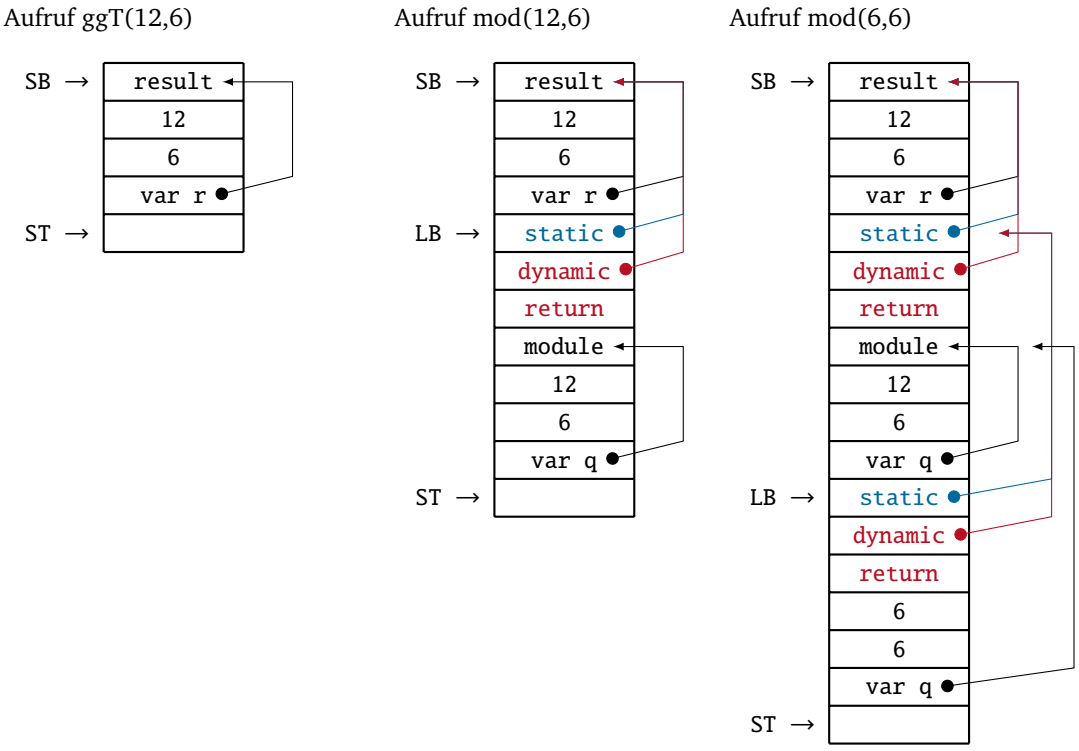
Abbildung 1: Lösung zu Aufgabe 3.4

Aufgabe 3.5 Routinenprotokoll

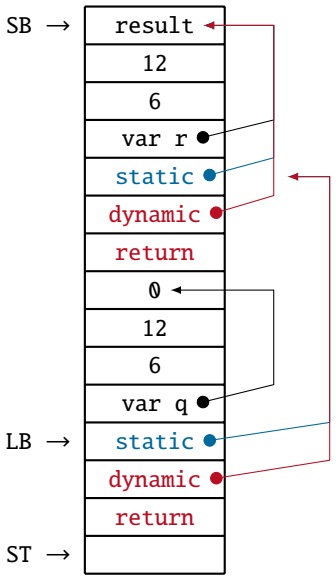
Das nachfolgende Programm berechnet den größten gemeinsamen Teiler zweier Zahlen mittels des euklidischen Algorithmus. Zeichnen Sie hierfür Stack Frames (also die Argumente, Resultate und Verwaltungsinfos) vor Eintritt in eine Funktion und nach Austritt aus einer Funktion für die Auswertung des Ausdrucks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

```
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
```

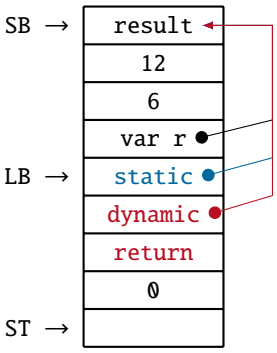
Listing 2: Beispielprogramm zum Routinenprotokoll zu Aufgabe 35a



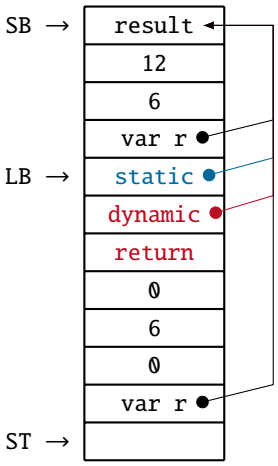
Austritt mod(6,6) Aufruf



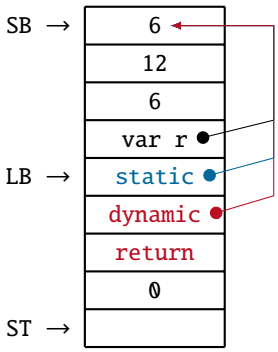
Austritt mod(12,6) Aufruf



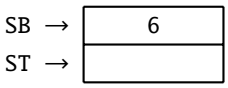
Aufruf ggT(6,0)



Austritt ggt(6,0) Aufruf



Austritt ggT(12,6) Aufruf



Diese Grafik wurde uns von einer Übungsgruppe freundlicherweise zur Verfügung gestellt. Vielen Dank!

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
```

Aufgabe 3.6 Lösung

```
0:  JUMP        7[CB]
1:  LOAD  (1)   -1[LB]; func twice; load first parameter
```

```
2:  LOAD  (2)   -3[LB]; load closure
3:  CALLI             ; call loaded closure
4:  LOAD  (2)   -3[LB]; load closure
5:  CALLI             ; call loaded closure
6:  RETURN(1)   3
7:  JUMP        12[CB]
8:  LOAD  (1)   -1[LB]; func double
9:  LOADL       2
10:  CALL  (SB)  10[PB]; call mult
11:  RETURN(1)   1
12:  PUSH        1; main program
13:  LOADA       0[SB]; closure start - static link
14:  LOADA       8[CB]; closure end   - address of double
15:  LOADL       10
16:  CALL  (SB)  1[CB]; call twice
17:  STORE (1)   0[SB]
18:  LOAD  (1)   0[SB]
19:  CALL  (SB)  26[PB]; call putint
20:  POP   (0)   1
21:  HALT
```