Einführung in den Compilerbau

Prof. Dr.-Ing. Andreas Koch
Julian Oppermann, Lukas Sommer

TECHNISCHE UNIVERSITÄT DARMSTADT

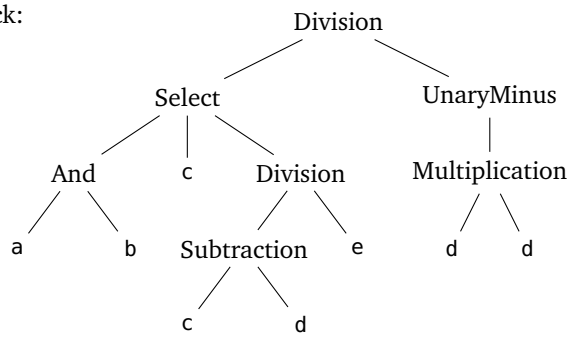
Wintersemester 18/19
Theorieblatt 3

Abgabe bis Sonntag, 27.01.2019, 18:00 Uhr (MEZ)

Aufgabe 3.1 Ausdrucksauswertung

6+3+6 = 15 P

Gegeben sei der folgende Ausdruck:



Hinweis: Der Select-Knoten hat folgende Kindknoten (von links nach rechts): “Condition”, “True”, “False”.

a) Erzeugen Sie für den Ausdruck eine Instruktionsfolge für eine typische Stackmaschine (vgl. Folie 33, Foliensatz “Laufzeitorganisation”). Nach der Verarbeitung der letzten Instruktion soll das Ergebnis der Ausdrucksauswertung als oberstes Element auf dem Stack liegen. Verwenden Sie folgende Instruktionen:

```
LOAD v   Hole den Wert der Variablen 'v' und lege ihn auf den Stack.

AND      Ersetze die zwei obersten Werte auf dem Stack durch ihr bitweises Und:
         value(ST') ← value(ST-1) & value(ST)

SUB      Ersetze die zwei obersten Werte auf dem Stack durch ihre Differenz:
         value(ST') ← value(ST-1) - value(ST)

MUL      Ersetze die zwei obersten Werte auf dem Stack durch ihr Produkt:
         value(ST') ← value(ST-1) * value(ST)

DIV      Ersetze die zwei obersten Werte auf dem Stack durch ihren Quotienten:
         value(ST') ← value(ST-1) / value(ST)

NEG      Ersetze den obersten Wert auf dem Stack durch seine Negation:
         value(ST') ← -1 * value(ST)

SEL      Ersetze die obersten drei Werte auf dem Stack durch den dritten Wert,
         falls der erste 0 ist, ansonsten durch den zweiten Wert:
         value(ST') ← value(ST-2) ? value(ST-1) : value(ST)
```

b) Informieren Sie sich über die sog. “Ershov-Zahlen”[1] und geben Sie diese für alle Knoten im oben gegebenen Ausdrucksbaum an.

[1] z.B. https://de.wikipedia.org/wiki/Ershov-Zahl

c) Erzeugen Sie für den Ausdruck eine Instruktionsfolge für eine imaginäre Registermaschine. Die Maschine habe 8 Register `r1, r2, ..., r8` und unterstütze die folgenden Instruktionen:

```
rx = ld v            Hole den Wert der Variablen 'v' und schreibe ihn in das Register 'rx'.

rx = and ra, rb      Schreibe das Ergebnis des bitweise Und 'ra'&'rb' nach 'rx'.

rx = sub ra, rb      Schreibe das Ergebnis der Subtraktion 'ra'-'rb' nach 'rx'.

rx = div ra, rb      Schreibe das Ergebnis der Division 'ra'/'rb' nach 'rx'.

rx = mul ra, rb      Schreibe das Ergebnis der Multiplikation 'ra'*'rb' nach 'rx'.

rx = neg ra          Schreibe das Ergebnis der Negation -1*'ra' nach 'rx'.

rx = sel ra, rb, rc  Schreibe das Ergebnis der Selektion 'ra'?'rb':'rc' nach 'rx'.

// Die Quell- und das Zielregister können identisch sein.
```

Verwenden Sie in Ihrer Instruktionsfolge **so wenig wie möglich** Register. Nach der Verarbeitung der letzten Instruktion soll das Ergebnis der Ausdrucksauswertung in `r1` stehen.

Aufgabe 3.2 Registerallokation *3+5 = 8 P*

Gegeben sei folgendes MAVL-Programmfragment (alle Variablen sind mit Typ `int` deklariert):

```
// ...
z = z * x;
w = x + y;
x = z / 2;
y = x * 5;
w = y - w;
z = w + 1;
// ...
```

Sie sollen in dieser Aufgabe einige Aspekte der Registerallokation anhand dieses Beispiels betrachten, indem Sie Code für eine imaginäre Registermaschine ähnlich der in Aufgabe 3.1 c) erzeugen. Anders als in der vorherigen Aufgabe sollen nicht nur Zwischenergebnisse, sondern auch die Variablen selbst in Register abgelegt werden. In dieser Aufgabe hat die Zielmaschine nur drei statt acht Register, es gibt also mehr Variablen als Register. Deshalb ist es potenziell erforderlich, einige der Variablen stattdessen in den Speicher auszulagern (sogenanntes *spilling*) und sie bei Benutzung temporär in ein Register zu duplizieren, um mit ihnen arbeiten zu können.

Zusätzlich zu den Registermachinen-Instruktionen aus der vorherigen Aufgabe benutzen wir hier Varianten mit einem konstanten Operand (z.B., `rx = ra + 2`) und schreiben den Inhalt eines Registers `ra` in den Speicher einer Variable `v` mit `st v, ra`.

a) Generieren Sie Code für das obige Programm unter der Annahme, dass jede Variable *entweder* in einem festen Register *oder* im Speicher abgelegt ist. Konkret soll `w` jederzeit in `r1` liegen und `x` in `r2`. Das Register `r3` ist reserviert, um *temporär* die Werte von Variablen zwischenzuspeichern, die im Speicher abgelegt sind. Beachten Sie, dass `r3` nur innerhalb einer Anweisung benutzt werden soll, d.h., eventuelle Optimierungen, die `r3` über mehrere Anweisungen hinweg benutzen, sollen (noch) nicht durchgeführt werden.

b) Deutlich effizienterer Code kann generiert werden, wenn die Zuordnung von Variablen zu Registern flexibler ist, also die gleiche Variable zu verschiedenen Zeitpunkten in verschiedenen Registern leben kann. Geben Sie unter der Annahme, dass eingangs `x` in `r1`, `y` in `r2` und `z` in `r3` abgelegt ist, möglichst kompakten Registermaschinen-Code für obiges Programm an. Machen Sie deutlich, wie sich bei jeder Instruktion die Zuordnung von Variablen zu Registern ändert, und geben Sie zusätzlich an, welche Variable am Ende in welchem Register liegt. Nehmen Sie an, dass `x` nach dem Ende des Programmfragments nicht mehr benutzt wird, also der Wert dieser Variable nicht bis zum Ende erhalten bleiben muss.

Aufgabe 3.3 MTAM-Routinenprotokoll *4+8 = 12 P*

In den folgenden Teilaufgaben sind MAVL-Programme und ihre Übersetzung in Assemblercode für die MATM gegeben. Sie sollen zu jedem dieser Programme den Stackaufbau **wortweise** zu einem gegebenen Zeitpunkt ermitteln. Geben Sie für jedes Stackelement dessen Inhalt **und** seine Bedeutung (z.B. "Speicherort von Variable v", "2. Argument für Funktion foo", usw.) an. Handelt es sich beim Inhalt um eine Adresse, so verwenden Sie die in der TAM-Assemblerdarstellung übliche Offsetform (z.B. "`42[SB]`", "`11[CB]`"). Der Stack soll von oben nach unten wachsen, d.h. die erste Zeile in Ihrer Lösung entspricht der Adresse `0[SB]`, die zweite Zeile hat die Adresse `1[SB]` usw..

Listing 1: MAVL-Programm a)

```
function int calculate(int x, int y) {
    return x + y;
}

function void main() {
    val int a = 39;
    var int b;
    b = calculate(157, a);
}
```

Listing 2: MTAM-Assemblercode a)

```
 0:  JUMP        5[CB]  main
# function INT calculate(INT, INT)
 1:  LOAD  (1)   -2[LB]
 2:  LOAD  (1)   -1[LB]
 3:  add
 4:  RETURN(1)   2
# function VOID main()
 5:  LOADL       39
 6:  PUSH        1
 7:  LOADL       86
 8:  LOAD  (1)   2[LB]
 9:  CALL  (CB)  1[CB]  calculate
10:  LOADA       3[LB]
11:  STOREI(1)
12:  HALT
```

Listing 3: MAVL-Programm b)

```
function int bar(int y) {
  return y + 17;
}

function void foo(int x) {
  bar(2);
}

function void main() {
  foo(1);
}
```

Listing 4: MTAM-Assemblercode b)

```
 0:  JUMP        15[CB] main
# function INT bar(INT)
 1:  LOAD  (1)   -1[LB]
 2:  LOADL       17
 3:  add
 4:  RETURN(1)   1
# function VOID foo(INT)
 5:  PUSH        1
 6:  LOAD  (1)   -1[LB]
 7:  LOADL       1
 8:  add
 9:  LOADA       2[LB]
10:  STOREI(1)
11:  LOAD  (1)   2[LB]
12:  CALL  (CB)  1[CB]  bar
13:  POP   (0)   1
14:  RETURN(0)   1
# function VOID main()
15:  LOADL       1
16:  CALL  (CB)  5[CB]  foo
17:  HALT
```

a) Betrachten Sie Listings 1 und 2. Geben Sie den Stackaufbau zum Zeitpunkt **vor** Ausführung der `CALL` instruction in `9[CB]` an.

b) Betrachten Sie Listings 3 und 4. Geben Sie den Stackaufbau zum Zeitpunkt **vor** Ausführung der `RETURN` instruction in `4[CB]` an.

Aufgabe 3.4 Speicherlayout *2+3 = 5 P*

Geben Sie für **alle** Zuweisungen in den folgenden MAVL-Funktionen die Adresse des Zuweisungsziels als Offset zum Beginn des Stackframes (z.B. `42[LB]`) an.

a)

```
void func_a() {
  val int i = 0;
  var float f;
  f = 0.0;
}
```

b)

```
void func_b() {
  var matrix<int>[3][5] m;
  var int i;
  m[2][1] = 0;
  m[1][2] = 0;
  i = 0;
}
```