Einführung in den Compilerbau

Prof. Dr.-Ing. Andreas Koch
Tim Noack, Tammo Mürmann

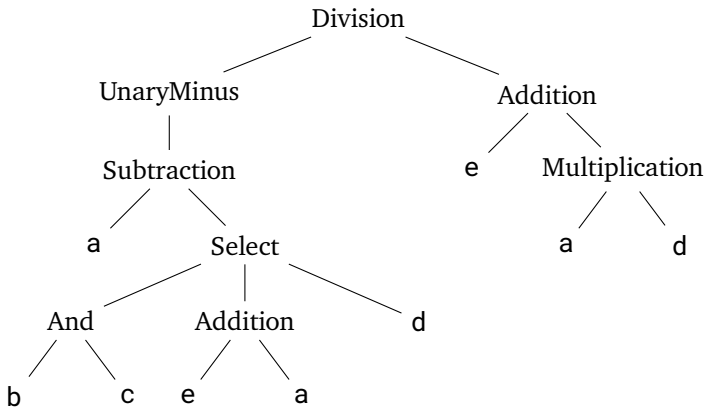
TECHNISCHE UNIVERSITÄT DARMSTADT

Theorie
Wintersemester 24/25
Übungsblatt 3

Abgabe bis Sonntag, 19.01.2025, 18:00 Uhr (MEZ)

Aufgabe 3.1: Ausdrucksauswertung (15 Punkte)

Gegeben sei der folgende Ausdruck:



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

3.1a) Typische Stackmaschine (6 Punkte)

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)


ADD      Ersetze die zwei obersten Werte auf dem Stack durch ihre Summe:
         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)
```

3.1b) Ershov-Zahlen (3 Punkte)

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

3.1c) Registermaschine (6 Punkte)

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 = add ra, rb     Schreibe das Ergebnis der Addition 'ra'+'rb' nach 'rx'.

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

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

rx = div ra, rb     Schreibe das Ergebnis der Division '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 wenige Register wie möglich**. Nach der Verarbeitung der letzten Instruktion, soll das Ergebnis der Ausdrucksauswertung in `r1` stehen.

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

Aufgabe 3.2: MTAM-Routinenprotokoll (12 Punkte)

In den folgenden Teilaufgaben sind MAVL-Programme und ihre Übersetzung in Assemblercode für die MTAM gegeben. Sie sollen zu jedem dieser Programme den Stackaufbau **wortweise** zu einem gegebenen Zeitpunkt ermitteln[2]. 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.

3.2a) MTAM-Routinenprotokoll (4 Punkte)

Betrachten Sie folgendes MAVL-Programm und die zugehörige Übersetzung in Assemblercode für die MTAM. Geben Sie den Stackaufbau zum Zeitpunkt **vor** Ausführung der `CALL` instruction in `12[CB]` an.

```
function int calculate(int a, int b) {
  return b * a;
}

function void main() {
  val int y = 42;
  var int x;
  x = calculate(y + y, 21);
}
```

```
 0:  CALL         6[CB]    ; main
 1:  HALT
# function int calculate(int, int)
 2:  LOAD   (1)  -1[LB]
 3:  LOAD   (1)  -2[LB]
 4:  mulI
 5:  RETURN (1)   2
# function void main()
 6:  LOADL        42
 7:  PUSH         1
 8:  LOAD   (1)   2[LB]
 9:  LOAD   (1)   2[LB]
10:  addI
11:  LOADL        21
12:  CALL         2[CB]    ; calculate
13:  LOADA        3[LB]
14:  STOREI (1)
15:  RETURN (0)   0
```

[2]Orientieren Sie sich hierbei an der MTAM Spezifikation

3.2b) MTAM-Routinenprotokoll (8 Punkte)

Betrachten Sie folgendes MAVL-Programm und die zugehörige Übersetzung in Assemblercode für die MTAM. Geben Sie den Stackaufbau zum Zeitpunkt **vor** Ausführung der `RETURN` instruction in `5[CB]` an.

```
function int bar(int y) {
  return y * y;
}

function void foo(int x) {
  var int z;
  z = 12;
  z = bar(x - z);
}

function void main() {
  foo(0);
  var int a;
  val int b = 99;
  a = b + 1;
}
```

```
 0:  CALL         17[CB]   ; main
 1:  HALT
# function int bar(int)
 2:  LOAD   (1)  -1[LB]
 3:  LOAD   (1)  -1[LB]
 4:  mulI
 5:  RETURN (1)   1
# function void foo(int)
 6:  PUSH         1
 7:  LOADL        12
 8:  LOADA        2[LB]
 9:  STOREI (1)
10:  LOAD   (1)  -1[LB]
11:  LOAD   (1)   2[LB]
12:  subI
13:  CALL         2[CB]   ; bar
14:  LOADA        2[LB]
15:  STOREI (1)
16:  RETURN (0)   1
# function void main()
17:  LOADL        0
18:  CALL         6[CB]   ; foo
19:  PUSH         1
20:  LOADL        99
21:  LOAD   (1)   3[LB]
22:  LOADL        1
23:  addI
24:  LOADA        2[LB]
25:  STOREI (1)
26:  RETURN (0)   0
```

Aufgabe 3.3: Speicherlayout (5 Punkte)

3.3a) Speicherlayout (2 Punkte)

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

```
function void main() {
  var matrix<int>[5][9] a;
  var vector<float>[7] b;
  a[2][6] = 42;
  b[3] = 3.0;
}
```

3.3b) Speicherlayout (3 Punkte)

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

```
record rec {
  val float x;
  val bool y;
  var matrix<int>[2][3] m;
  val int z;
}

function void main() {
  var vector<int>[13] v;
  var rec r;
  var int a;
  r = @rec[4.2, true, [[4, 2, 1], [3, 1, 0]], 21];
  a = 63;
  var int b;
  r@m = [[1, 2, 3], [4, 5, 6]];
}
```

Aufgabe 3.4: Syntaktischer Zucker (8 Punkte)

Programmiersprachen bieten oft sogenannten "syntaktischen Zucker" an. Dabei handelt es sich um Konstrukte, die einen häufig vorkommenden Spezialfall kürzer und prägnanter ausdrücken. In MAVL fällt die `foreach`-Schleife in diese Kategorie, da jedes Vorkommen dieser Schleifenart auch durch eine normale `for`-Schleife ersetzt werden kann.

Geben Sie für die `foreach`-Schleife, die mit einem **var-Iterator** über einen **Vektor** iteriert, äquivalenten MAVL-Code ohne `foreach` an.

```
foreach(var $T $ID_1 : $ID_2) $S
```

Ein Textersetzungssystem wird die mit einem Dollar-Symbol beginnenden Parameter für Sie ersetzen, und für eindeutige Bezeichner sorgern, falls Sie neue Hilfsvariablen deklarieren möchten.

Beispiel für die gewünschte Notation: Eine Verwendung des Select-Operators würden Sie folgendermaßen in eine äquivalente Verzweigung umschreiben:

```
val $T $ID_1 = $E_1 ? $E_2 : $E_3;

=>

var $T _tmp;
if ($E_1)
  _tmp = $E_2;
else
  _tmp = $E_3;
val $T $ID_1 = _tmp;
```

Es ist hier also *nicht* die Notation der Codeschablonen aus der Vorlesung (`execute[...]`, `elaborate[...]`, usw.) gefordert.