

Übung zur Vorlesung Compiler 1: Grundlagen

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Wintersemester 13/14
Aufgabenblatt 4

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 14.02.2014 um 23:59 MET an oc@esa.informatik.tu-darmstadt.de ein.

Aufgabe 4.1 Codeschablonen I

Erzeugen Sie mit Hilfe der Codeschablonen aus den Folien von Block 05, Seite 8 ff. die TAM-Befehle für das gegebene Programm. Gehen Sie dabei schrittweise, vergleichbar einem Visitors, vor. Jeder Aufruf einer Code-Schablone sollte ein Schritt Ihrer detaillierten Lösung darstellen. Markieren Sie hierbei, wann eine Sprungadresse nicht direkt eingetragen werden kann. In einem solchen Fall markieren Sie auch den Schritt, in dem die Adresse durch Backpatching eingetragen wird. Die konkreten Adressen der Variablen und Sprungmarken sind nicht notwendig, es reicht wenn Sie diese als Namen angeben.

```
let
  const MAX ~ 10;
  var n: Integer
in begin
  getint(var n);
  if (n>0) /\ (n<=MAX) then
    while n > 0 do begin
      putint(n); puteol();
      n := n - 1;
    end
  else
  end
end
```

Listing 1: Programm zur Übersetzung mittels Code-Schablonen

Aufgabe 4.2 Codeschablonen II

Triangle soll um ein "foreach ARRAY do C" Kommando erweitert werden. Dieser Befehl soll auf jedes Element des Arrays das Kommando C ausführen. Hierbei wird das aktuelle Element von ARRAY innerhalb C durch current dargestellt. In size steht die Anzahl der Elemente von ARRAY. Beachten Sie hierbei, dass C auch ein "begin ... end" Block sein kann. Ein return beendet ganz normal die Ausführung der aktuellen Prozedur.

Bestimmen Sie für diesen Befehl die Codeschablone, welche das Kommando ausführt. Sie dürfen hierzu bestehende Schablonen verwenden.

Adresse	Routine
1	id
2	not
3	and
4	or
5	succ
6	pred
7	neg
8	add
9	sub
10	mult
11	div
12	mod
13	lt
14	le
15	ge
16	gt
17	eq
18	ne
19	eol
20	eof
21	get
22	put
23	geteol
24	puteol
25	getint
26	putint
27	new
28	dispose
29	fopen
30	fgetint
31	fputint
32	fget
33	fput
34	fgeteol
35	fputeol
36	feol
37	feof
38	fclose

Tabelle 1: Primitive Routine IDs

Aufgabe 4.3 Adressen von Konstanten und Variablen

Erweitern Sie den gegebenen TAM-Code um Befehle für die Speicherverwaltung und ersetzen Sie die Variablen-, Prozedur- und Funktionsnamen durch ihre konkreten Adressen gemäß den Beispiel in Abbildung 1. Die Adressen der primitiven Routinen können sie Tabelle 1 entnehmen.

<pre> let var x : Integer; const n ~ 0-1 in x := n + 1; </pre> <p>(a) Triangle-Code</p>	<pre> 0 ... 1 LOADL 0 2 LOADL 1 3 CALL sub 4 LOAD (1) n 5 LOADL 1 6 CALL add 7 STORE (1) x 8 ... 9 HALT </pre> <p>(b) TAM-Code</p>	<pre> 0 PUSH 1 1 LOADL 0 2 LOADL 1 3 CALL (SB) 9[PB] 4 LOAD (1) 1[SB] 5 LOADL 1 6 CALL (SB) 8[PB] 7 STORE (1) 0[SB] 8 POP (0) 2 9 HALT </pre> <p>(c) Lösung TAM-Code</p>
--	--	--

Abbildung 1: Beispiel

<pre> let const MAX ~ 10; proc test (a : Integer) ~ let var b : Integer in begin b := a / 2; b := b + 3; putint (b) end in test (MAX) </pre> <p>(a) Triangle-Code</p>	<pre> 0 JUMP 14[CB] 1 ... 2 LOAD (1) a 3 LOADL 2 4 CALL (SB) div 5 STORE (1) b 6 LOAD (1) b 7 LOADL 3 8 CALL (SB) add 9 STORE (1) b 10 LOAD (1) b 11 CALL (SB) putint 12 ... 13 RETURN(0) 1 14 LOADL 10 15 CALL (SB) test 16 HALT </pre> <p>(b) TAM-Code</p>
---	--

Abbildung 2: Aufgabe a

	0	...	
	1	JUMP	23[CB]
	2	...	
	3	LOAD (1)	x
	4	LOADI (1)	
	5	LOADL	2
let	6	CALL (SB)	mult
var a : Integer ;	7	STORE (1)	b
proc test (var x : Integer) ~	8	LOAD (1)	x
let	9	LOADI (1)	
var b : Integer ;	10	LOADL	10
const A ~ 10	11	CALL (SB)	add
in	12	LOAD (1)	b
begin	13	CALL (SB)	add
b := x * 2;	14	LOAD (1)	x
x := x + A + b;	15	STOREI(1)	
putint (A + x)	16	LOADL	10
end	17	LOAD (1)	x
in	18	LOADI (1)	
begin	19	CALL (SB)	add
a := 4;	20	CALL (SB)	putint
test (var a)	21	...	
end	22	RETURN(0)	1
	23	LOADL	4
	24	STORE (1)	a
	25	LOADA	a
	26	CALL (SB)	test
	27	...	
	28	HALT	

(a) Triangle-Code

(b) TAM-Code

Abbildung 3: Aufgabe b

	0 ...	
	1 ...	
	2 JUMP	7[CB]
	3 LOAD (1)	i
	4 LOAD (1)	i
	5 CALL	mult
let	6 RETURN(1)	1
var a : Integer;	7 JUMP	26[CB]
var b : Integer;	8 JUMP	20[CB]
	9 LOAD (1)	u
func f(i : Integer)	10 LOADI (1)	
: Integer ~	11 CALL (SB)	f
i * i;	12 CALL	putint
	13 CALL	puteol
proc p(l : Integer,	14 LOAD (1)	u
var u : Integer) ~	15 LOADI (1)	
begin	16 LOADL	1
while u > 1 do begin	17 CALL	sub
putint(f(u));	18 LOAD (1)	u
puteol();	19 STOREI(1)	
u := u - 1;	20 LOAD (1)	u
end	21 LOADI (1)	
end	22 LOAD (1)	l
in	23 CALL	gt
begin	24 JUMPIF(1)	9[CB]
a := 10;	25 RETURN(0)	2
b := 0;	26 LOADL	10
p(b, var a);	27 STORE (1)	a
end	28 LOADL	0
	29 STORE (1)	b
	30 LOAD (1)	b
	31 LOADA	a
	32 CALL (SB)	p
	33 ...	
	34 HALT	

(a) Triangle-Code

(b) TAM-Code

Abbildung 4: Aufgabe c

Implementieren Sie mit Hilfe von ANTLRv4 einen Compiler, der ein Brainfuck-Programm einliest und Code für die Triangle Abstract Machine (in der Assemblerdarstellung) erzeugt.

Hinweise:

- <http://de.wikipedia.org/wiki/Brainfuck>
- Eine Beschreibung des TAM-Befehlssatzes finden Sie hier: <http://goo.gl/IqmTUd>
- Lassen Sie sich von ANTLR einen Visitor erzeugen.

Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Weitere Infos unter www.informatik.tu-darmstadt.de/plagiarism