Uitwerking tentamen Compilerconstructie, maandag 19 december 2016

18.36

1. Drie soorten informatie over een variabele

* naam – tijdens syntax analyse, want dan is pas bekend in welke scope de variabele zit, dus in welk deel van de symbol table hij moet komen
* type – tijdens syntax analyse
* adres – tijdens code generation (òf het absolute adres voor een globale variabele, òf het relatieve adres, t.o.v. de framepointer, voor een lokale variabele van een functie)

18.43 Er zijn ook andere antwoorden mogelijk.

2 (a)

FIRST(α) bevat de terminalen $a \in \Sigma$ waarvoor $\alpha \Rightarrow^* a\beta$ met $\beta \in (V \cup \Sigma)^*$, ofwel de terminalen die als eerste terminaal kunnen voorkomen in een string die afgeleid wordt van α.
Daarnaast bevat FIRST(α) ook ϵ, als $\alpha \Rightarrow^* \epsilon$, dat wil zeggen, als ϵ afgeleid van worden uit α.

FOLLOW(A) bevat de terminalen $a \in \Sigma$ waarvoor $S \Rightarrow^* \beta_1 A a \beta_2$ met $\beta_1, \beta_2 \in (V \cup \Sigma)^*$, ofwel de terminalen die direct na A voor kunnen komen in een string afgeleid van het startsymbool S.
Daarnaast bevat FOLLOW(A) ook \$, als $S \Rightarrow^* \beta_1 A$ met $\beta_1 \in (V \cup \Sigma)^*$, dat wil zeggen, als A voor kan komen aan het eind van een string afgeleid van S. In het bijzonder is \$ ∈ FOLLOW(S).

18.53

(b)

	FIRST	FOLLOW
S	{a, b, c}	{\$, b}
A	{a, c, ε}	{a, \$, b}
B	{a, c}	{\$, b}

18.58

(c) Per variabele en letter in de invoer bekijken we welke productie mogelijk is

	a	b	c	\$
S	S → AaB	S → bA	S → AaB	
A	A → Bb A → ε	A → ε	A → Bb	A → ε
B	B → aS		B → cB	

Als $\alpha \Rightarrow^* \epsilon$, dan is productie A → α mogelijk bij elke terminaal / \$ in FOLLOW(A)

19.04

(d) Nee, G is geen LL(1) grammatica, want de top-down pasing table bevat twee producties bij variabele A en terminaal a.

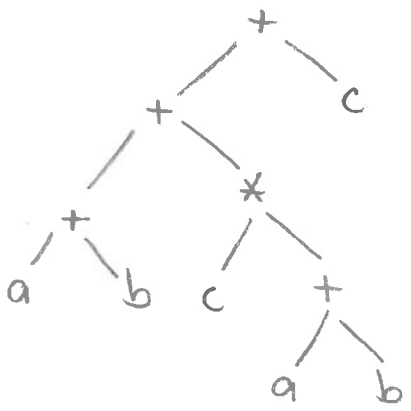
19.06

3 (a)

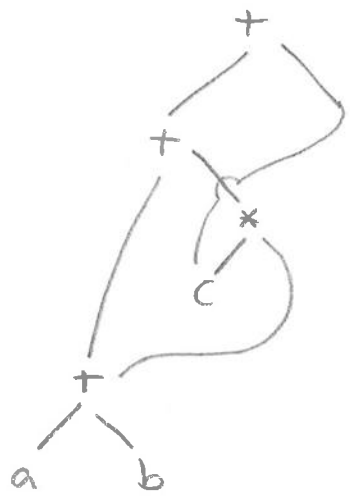
* Je hoeft minder knopen op te slaan in een gerichte acyclische graaf, omdat een knoop 'meerdere keren gebruikt kan worden' als kind van een andere knoop.
* Wanneer een knoop meerdere keren gebruikt wordt als kind van een andere knoop, hoef je zijn waarde maar één keer te berekenen. (common subexpressie). Dit is dus nuttig bij optimalisatie.

19.11

(b) Syntax tree

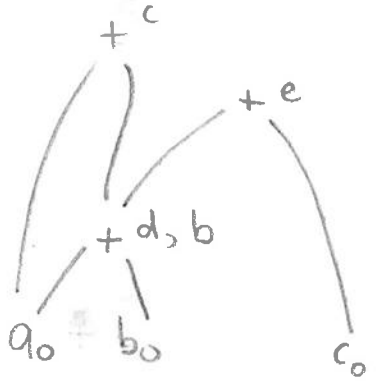


Gericht acyclische graaf



19.15

(c)



19.17

Voor elke variabele wier oorspronkelijke waarde we gebruiken in de instructies, hebben we een knoop met subscript 0, voor die oorspronkelijke waarde.

Voor elke toegepaste operator hebben we een knoop met daaraan gekoppeld de variabelen wier nieuwe waarde bepaald wordt door de toepassing van die operator

19.21

(d) Voor elke wortel in de gerichte acyclische graaf kun je kijken welke variabelen daaraan gekoppeld zijn. Als dat alleen variabelen zijn die niet live zijn (voor een bepaalde wortel), is de berekening die in die wortel plaatsvindt dead-code. De wortel kan dan ook verwijderd worden met zijn uitgaande takken (de berekening wordt dan niet uitgevoerd). Hierdoor kunnen nieuwe wortels ontstaan, die ook onderzocht kunnen worden, enzovoort, totdat aan elke wortel in de graaf minstens één live variabele is gekoppeld.

19.28

Dode variabelen kunnen ook losgemaakt worden van niet-wortels, zolang er maar minstens een variabele over blijft bij zo'n knoop.

4(a)

B.truelist bevat (de nummers van) de goto-instructies die moeten springen naar de code die uitgevoerd moet worden als B true is. Op de plaats van die goto-instructies kondem we namelijk concluderen dat B true was.

B.falselist analoog, maar dan voor false in plaats van true.

S.nextlist bevat (de nummers van) de goto-instructies die moeten springen naar de code die uitgevoerd moet worden na statement S. Op de plaats van die goto-instructies konden we namelijk concluderen dat we klaar waren met S

19.36

Uitwerking tentamen Compilerconstructie, maandag 19 december 2016

19.41
(b)

We maken een bottom up wandeling door de afleidingsboom, waarbij we de attributen invullen en de drie-adres code genereren

$M_1.instr = 100$
$E_1.addr = x$ (in plaats van pointer naar symbol in symbol table)
$E_2.addr = z$
$B.truelist = \{100\}$
$B.falselist = \{101\}$
$M_2.instr = 102$
$E_4.addr = x$
$E_5.addr = y$
$E_3.addr = new\ Temp() = t1$
$S_1.nextlist = null$
$S.nextlist = B.falselist = \{101\}$

100. if x < z goto 102
101. goto _
102. t1 = x + y
103. x = t1
104. goto 100
105

19.51
(c) backpatch ($S_1.nextlist$, $M_1.instr$):
de goto's in $S_1.nextlist$ worden ingevuld met instructie $M_1.instr$
Als je namelijk tijdens S_1 ontdekt dat je klaar bent met S_1, moet je terug naar de conditie van de while-lus. Die conditie staat / begint op instructie $M_1.instr$. (want voordat we de conditie gingen vertalen, hebben we het nummer van de eerstvolgende instructie opgeslagen in $M_1.instr$

19.56
20.20
backpatch ($B.truelist$, $M_2.instr$)
de goto's in $B.truelist$ worden ingevuld met $M_2.instr$.
Als je namelijk tijdens het evalueren van B kan concluderen dat B true is, moet statement S_1 uitgevoerd worden. De code voor dat statement staat / begint op instructie $M_2.instr$ (want voordat we het statement gingen vertalen, hebben we het nummer van de eerstvolgende instructie opgeslagen in $M_2.instr$.

20.24
$S.nextlist = B.falselist$.
de goto's in $B.falselist$ komen nu ook in $S.nextlist$.
Als je namelijk tijdens het evalueren van B kan concluderen dat B false is, moet je uit de while-lus springen

Je bent dan klaar met S_1 en moet verder gaan met de instructie die na S_1 aan de beurt is. Wanneer we weten welke instructie dat is, kunnen we die invullen.

20.29

gen('goto M_1.instr')

Na het uitvoeren van S_1 (als je 'door de code van S_1 bent heengezakt'), moet je weer terug naar de conditie van de while-lus. Die conditie staat/begint op instructie M_1.instr.

In tegenstelling tot het geval dat je tijdens het uitvoeren van S_1 ontdekt dat je klaar bent met S_1

20.33

20.35

5 (a)

De register descriptor van een register bevat (de namen van) de variabelen wier huidige waarde in dat register te vinden is.
De address descriptor van een variabele houdt bij waar (in welk register en/of welke geheugenlocatie) de huidige waarde van die variabele te vinden is.

20.39

(b)

	Register descriptors			Address descriptors				
	R1	R2	R3	a	b	c	d	e
	–	–	–	a	b	c	d	e

d = a+b
LD R1, a
LD R2, b
ADD R1, R1, R2
a en b stoppen we in de eerste-de-beste lege registers. Omdat we de waarde van a niet meer nodig hebben (wordt in derde 3-adres instructie overschreven), gebruiken we R1 ook voor d.
	d	b	–	a	b, R2	c	R1	e

e = d+c (2) LD R3, c ADD R3, R1, R3	d	b	e	a	b, R2	c	R1	R3

Uitwerking tentamen Compilerconstructie, maandag 19 december 2016

	R1	R2	R3	a	b	c	d	e
(3) a = e (geen assembly)	d	b	a, e	R3	b, R2	c	R1	R3
c = a + b								
(4) { ST d, R1	c	b	a, e	R3	b, R2	R1	d	R3
ADD R1, R3, R2								
a = b - c								
SUB R2, R2, R1	c	a	e	R2	b	R1	d	R3

(3): e staat al in register R3. We hoeven dus alleen descriptors aan te passen.

(2): b in R2 hebben we straks nog nodig, dus we kiezen R3 voor c. De waarde van c hebben we na deze instructie niet meer nodig, en dus kiezen we R3 ook voor het resultaat e.

(4) De twee operanden a en b staan al in registers, maar we moeten een register vrijmaken voor het resultaat c. We kiezen voor R1, omdat
* we b in R2 zo meteen nog nodig hebben
(we hadden ook R3 kunnen kiezen, omdat we daarvoor alleen e zouden hoeven storen, want a krijgt zometeen een nieuwe waarde.

(5) De twee operanden b en c staan al in registers.
We kunnen R2 gebruiken voor het resultaat a, omdat we b toch ook nog in geheugen hebben.

21.03