11.41

1 (a)

Per scope kunnen er in een programmeertaal als C variabelen gedeclareerd worden, en scopes kunnen genest zijn

1) Een variabele x kan in verschillende geneste scopes gedeclareerd zijn. Met een symbol table per scope kun je bepalen welke declaratie van toepassing is op een gebruik van x in een bepaalde scope.

2) Met een symbol table per scope kunnen we aan het eind van een scope de betreffende symbol table weggooien, en vervolgens vaststellen of een variabele x die daarna gebruikt wordt, nog wel gedeclareerd is.

11.47 / 11.49

(b) Met de klasse Env vormen we een keten van hash tabellen: de op een bepaald moment geldende symbol tables per scope. De variabele top wijst naar het eind van de keten, naar de symbol table van de huidige scope.

Vóór de productie program → block hebben we nog geen block, en dus ook geen symbol table, laat staan een keten van symbol tables. Daarom wordt top op null gezet.

Met de accolade { in de productie block → { decls stmts } wordt een nieuw block geopend. Het oude uiteinde van de keten van symbol tables wordt opgeslagen in de lokale variabele saved, en de keten wordt verlengd met een nieuw object van klasse Env.
Met de accolade } in deze productie wordt het block weer afgesloten. De voor dit block toegevoegde symbol table is nu niet meer geldig. We zetten top daarom weer terug naar zijn oude waarde, die opgeslagen was in saved.

12.02

(c) De methode get van de klasse Env kijkt of de gezochte string s in zijn eigen hashtabel voorkomt. Zo nee, dan kijkt hij in de hashtabel van zijn voorganger prev, en zo nodig in de voorganger van de voorganger prev. Net zo lang totdat s gevonden wordt (in dat geval wordt het bijbehorende Symbol gereturneerd) of totdat er geen voorganger meer is (in dat geval wordt null gereturneerd).

12.06

12.09 Uitwerking tentamen Compilerconstructie, donderdag 20 december 2018

2(a)

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

12.12 (b)

FIRST (AcB) = {a, c}
FIRST (Bb) = {a}

12.13 (c)

Er geldt FIRST (bA) = {b}, zodat FIRST (AcB) ∩ FIRST (bA) = ∅
Bij S bepaalt het invoersymbool dus ondubbelzinnig welke productie gebruikt moet worden.

[en ε zit niet in FIRST (Bb)]

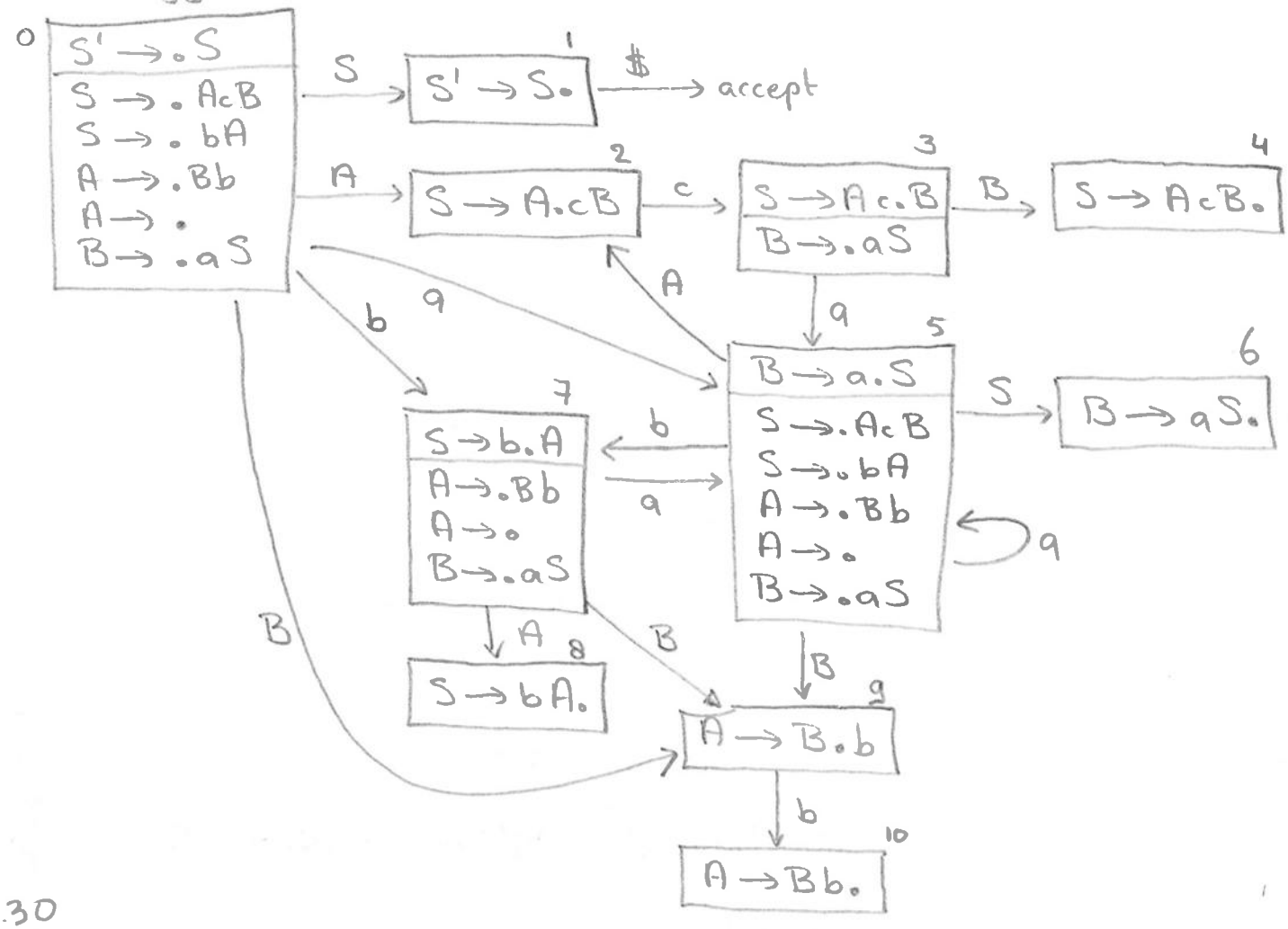
De tweede productie van A is een ε-productie! We moeten nog wel FIRST (Bb) vergelijken met FOLLOW (A) = {c, $, b}. Die twee verzamelingen zijn ook disjunct. Ook bij A bepaalt het invoersymbool dus ondubbelzinnig welke productie gebruikt moet worden.
= {a}

Bij B is er maar één productie.

Dus ja, G is LL(1).

12.18

(d) We wegen een speciale startproductie $S' \to S$ toe en krijgen dan:



12.30

(e) De SLR parsing table, waarbij we de producties als volgt nummeren:

1: $S \to AcB$
2: $S \to bA$
3: $A \to Bb$
4: $A \to \epsilon$
5: $B \to aS$

toestand	Action a	b	c	$	Goto S	A	B
0	s5	s7,r4	r4	r4	1	2	9
1				accept			
2			s3				
3	s5						4
4		r1		r1			
5	s5	s7,r4	r4	r4	6	2	9
6		r5		r5			
7	s5	r4	r4	r4		8	9
8		r2		r2			
9		s10					
10		r3	r3	r3			

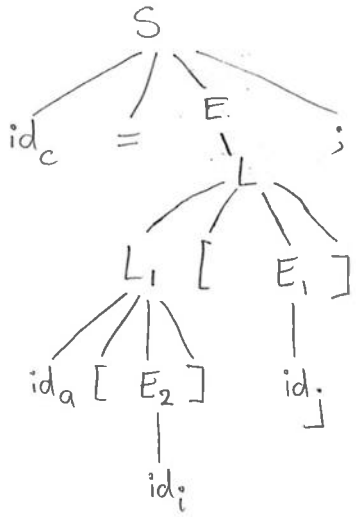
12.39

(f) Nee, G is geen SLR grammatica, want er zijn twee entries in de SLR parsing table met twee (dus meer dan één) acties: toestand 0 en toestand 5, met invoer b. In beide gevallen is er een shift-reduce conflict.

12.41

19.51

3(a)



19.55

(b) Attributen in de volgorde waarin ze hun waarde krijgen

$E_2.addr = i$
$L_1.array = a$
$L_1.type = array(4, integer)$
$L_1.addr = t1$
$E_1.addr = j$
$L.array = a$
$L.type = integer$
$t = t2$
$L.addr = t3$
$E.addr = t4$

Gegenereerde code (met pijl naar moment van genereren)

$t_1 = i * 16$
$t2 = j * 4$
$t_3 = t_1 + t2$
$t4 = a[t3]$
$C = t4$

Dat kan b.v. integer zijn, maar ook array (4, integer)

20.10

(c)

De variabele L genereert bij beide mogelijke producties een expressie E tussen blokhaken, die een index in het array oplevert (hierboven i of j). L.type is het soort / type elementen waar die index over gaat.

Bij de vertaling naar drie-adrescode van het attribuut type tweeledig:

* L.type.width wordt vermenigvuldigd met de betreffende index, als onderdeel van de som die een positie in het array berekent (in bytes vanaf het begin van het array)
* L.type.elem wordt (indien van toepassing) toegekend aan het attribuut type van de variabele L direct boven 'onze' L in de parse tree. Dan weten we tenminste ook van die L het soort / type elementen waar zijn index over gaat.

20.21

(d) Het attribuut type van de variabele L is een synthesized attribuut. Zijn waarde wordt namelijk bepaald door de waarde van een attribuut

* van een kind in de parse tree, in $L.type = L_1.type.elem$ bij productie $L \rightarrow L_1[E]$
* van L zelf, en indirect een kind in de parse tree, in $L.array = top.get(id.lexeme)$; en $L.type = L.array.type.elem$ bij productie $L \rightarrow id[E]$.

20.27

14.12

4(a)

De register descriptor van een register bevat (de namen van) de variabelen waarvan de huidige waarde zich bevindt in dat register

De address descriptor van een variabele bevat de locatie(s) (registers en/of geheugenlocaties) waar de huidige waarde van de variabele te vinden is.

14.16

(b)

Door de assembly instructie wordt de waarde in register R_x overschreven door de nieuwe waarde van x

1) * voor elke variabele V in de register descriptor van R_x (wiens waarde dus in R_x te vinden was) moet R_x uit de address descriptor van V verwijderd worden.
3) * de register descriptor van R_x moet alleen nog maar x bevatten (alle elementen worden dus uit (de descriptor) verwijderd en x wordt toegevoegd)
2) * voor elk register R in de address descriptor van x wordt x uit de register descriptor van R verwijderd (want de waarde van x is alleen nog maar in R_x te vinden)
4) * de address descriptor van x moet alleen nog maar R_x bevatten (alle elementen worden dus uit de descriptor verwijderd en R_x wordt toegevoegd)

In volgorde 1, 2, 3, 4 (voor het geval dat de oude waarde van x zich al in R_x bevond)

14.27.

Uitwerking tentamen Compilerconstructie, donderdag 20 december 2018

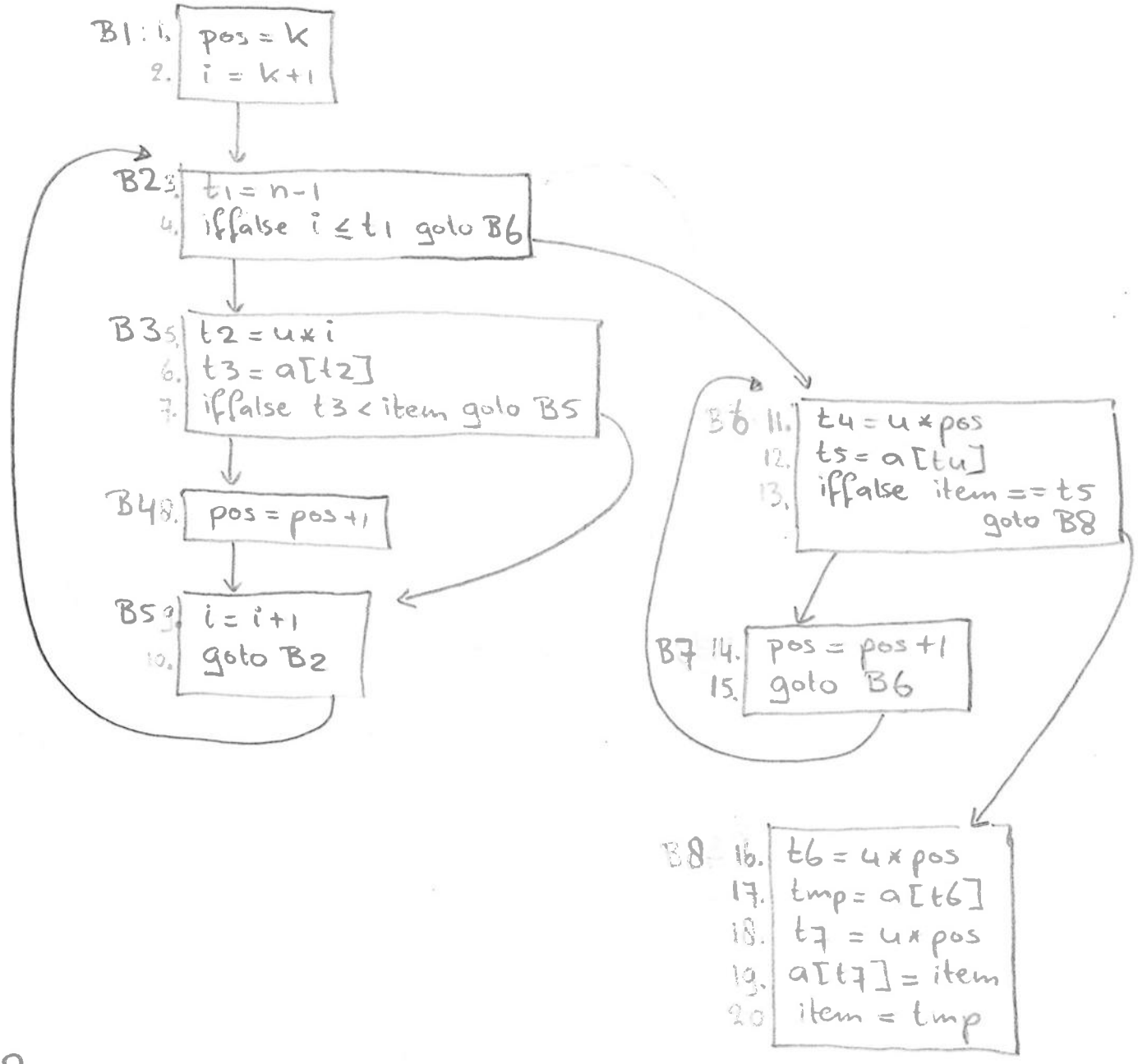
12.47

5 (a) De leaders zijn:
1, 11, 9, 3, 16, 5, 8, 14

In volgorde:
1, 3, 5, 8, 9, 11, 14, 16

12.50

(b)



12.58

(c) ① local common subexpression elimination
In B8 wordt twee keer w × pos berekend. De tweede keer elimineren we, en we vervangen t7 door t6:

B8
```
t6 = w * pos
tmp = a[t6]
a[t6] = item
item = tmp
```

(2) global common subexpression elimination.
In B6 en B8 wordt allebei 4x pos berekend.
De tweede keer elimineren we, en we vervangen t6 door t4

B8
```
tmp = a[t4]
a[t4] = item
item = tmp
```

(3) global common subexpression elimination
In B6 en B8 wordt allebei a[t4] berekend. We vullen de tweede keer gewoon t5 in

B8
```
tmp = t5
a[t4] = item
item = tmp
```

(4) copy propagation
De copy-instructie tmp = t5 in B8 propageren we:

B8
```
tmp = t5
a[t4] = item
item = t5
```

(5) dead-code elimination
De copy-instructie tmp = t5 in B8 is nu dead code, omdat gegeven is dat tmp aan het eind niet live is.
We elimineren de instructie dus:

B8
```
a[t4] = item
item = t5
```

(6) code motion
De instructie t1 = n-1 in B2 wordt iedere iteratie van de loop met B2, B3, B4, B5 uitgevoerd, terwijl n niet verandert.
We verplaatsen de instructie daarom naar B1

B1:
```
pos = k
i = k+1
t1 = n-1
```

B2
```
iffalse i ≤ t1 goto B6
```

13.15.