Uitwerking tentamen Compilerconstructie, maandag 15 december 2014

09:57

1) * een token is een tweetal (token-name, attribuut) dat overeenkomt met de kleinste bij elkaar horende (lexicale) eenheid in een programma. De token-name is het soort token (identifier, numerieke constante, relationele operator, ...), en het (optionele) attribuut is/verwijst naar (de waarde van) het feitelijke attribuut (welke identifier, welke numerieke constante, welke relationele operator).

* een lexeme is het (letterlijke) stukje programmacode dat het een token overeenkomt, bijv "count" (voor een identifier), "7.83" (voor een numerieke constante) of "<=" (voor een relationele operator)

* een pattern is een voorschrift waaraan de lexemes van een bepaald token moeten voldoen, b.v.: een identifier is een string van letters en cijfers, die begint met een letter.

10:12

17:41

2) First(α) is de verzameling terminalen die als eerste/voorste letter kunnen ontstaan bij een afleiding in de contextvrije grammatica vanuit α, het lege woord ϵ als dat uit α is af te leiden.

$$\text{First}(\alpha) = \{a \in \Sigma \mid \alpha \Rightarrow^* a\beta \text{ voor zekere } \beta \in (V \cup \Sigma)^*\}$$
$$\cup \{\epsilon\} \text{ als } \alpha \Rightarrow^* \epsilon$$

Hierin is Σ het terminaal alfabet van de CFG en V is de verzameling variabelen.

17:48

Laat S het startsymbool van de CFG zijn.
Dan is Follow(A) de verzameling terminalen die in een afleiding vanuit S direct achter de variabele A kunnen ontstaan, aangevuld met het end-of-string symbool \$ als het mogelijk is om A als laatste letter te krijgen van een string die uit S is af te leiden.

$$\text{Follow}(A) = \{a \in \Sigma \mid S \Rightarrow^* \alpha_1 A a \alpha_2 \text{ voor zekere } \alpha_1, \alpha_2 \in (V \cup \Sigma)^*\}$$
$$\cup \{\$\} \text{ als } S \Rightarrow^* \alpha_1 A \text{ voor zekere } \alpha_1 \in (V \cup \Sigma)^*.$$

17:54

18:42

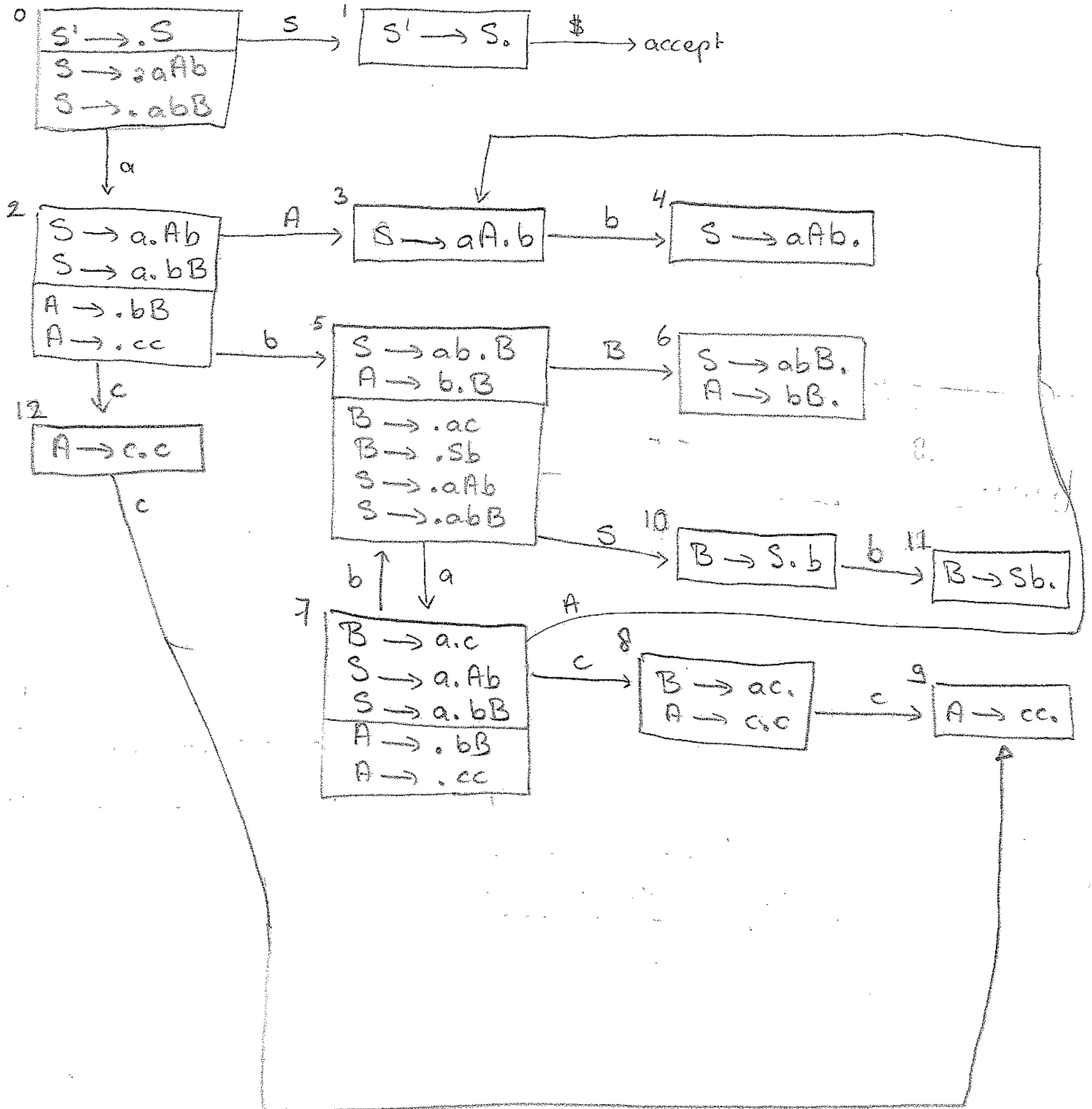
3(a) Follow(S) = {b, $}

Follow(A) = {b}

Follow(B) = Follow(A) ∪ Follow(S) = {b, $}.

18:44

(b) We voegen speciale startproductie S' → S toe aan G



Uitwerking tentamen Compilerconstructies maandag 15 december 2014

18:45

(c)

Toestand	Action a	b	c	$	Goto S	A	B
0	s2				1		
1				accept			
2		s5	s12			3	
3		s4					
4		r1		r1			
5	s7				10		6
6		r2/r3		r2			
7		s5	s8			3	
8		r5	s9	r5			
9		r4					
10		s11					
11		r6		r6			
12			s9				

18:57
19:03

We nummeren de oorspronkelijke producties als volgt:

(1) $S \rightarrow aAb$
(2) $S \rightarrow abB$
(3) $A \rightarrow bB$
(4) $A \rightarrow cc$
(5) $B \rightarrow ac$
(6) $B \rightarrow Sb$

19:05
controle
19:08

(d)

Nee, G is geen SLR grammatica, want de parsing tabel bevat een entry met twee mogelijke acties: als we in toestand 6 een b in de invoer zien staan, kunnen we reduceren naar $S \rightarrow abB$ of naar $A \rightarrow bB$.

19:10

4 (a) Het attribuut code is/wordt de drie-adrescode die overeenkomt met (de vertaling is van) de programmacode die wordt afgeleid uit P, B, of S.

Het attribuut next van S is/wordt een label van de drie-adresinstructie die uitgevoerd moet worden (waar we zo nodig naar toe moeten springen) als we klaar zijn met de instructie die afgeleid wordt uit S.

Het attribuut true van B is/wordt een label van de drie-adresinstructie die uitgevoerd moet worden (en waar we zo nodig naar toe moeten springen) als we bij het evalueren van de boolese expressie die afgeleid wordt uit B, tot de conclusie zijn gekomen dat die expressie true is.

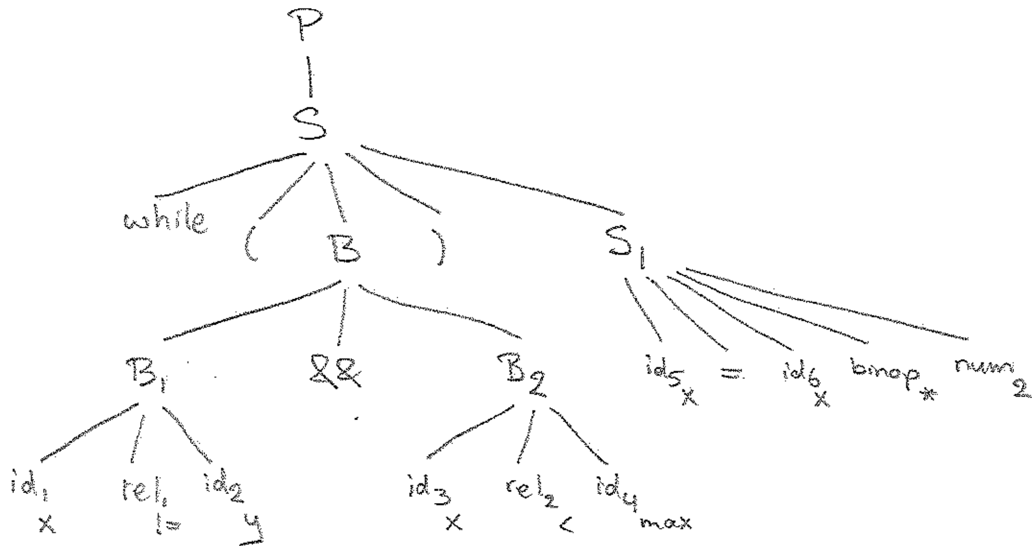
Het attribuut false van B is volstrekt analoog.

19:24

(b) Het attribuut code is synthesized: het wordt bottom-up opgebouwd
De attributen next, true en false zijn inherited: ze worden top-down doorgegeven

19:26

(c)



19:31
11:46

(d) We voeren een depth-first wandeling uit door de afleidingsboom, waarbij we van boven naar beneden labels doorgeven, en van beneden naar boven code opbouwen

```
S.next = L1
begin = L2
B.true = L3
B.false = L1
S1.next = L2
B1.true = L4
B1.false = L1
B2.true = L3
B2.false = L1
B1.code = if x != y goto L4
          goto L1
B2.code = if x < max goto L3
          goto L1
B.code = if x!=y goto L4
         goto L1
      L4 if x < max goto L3
         goto L1
S1.code = x = x * 2
S.code = L2  if x!=y goto L4
             goto L1
         L4  if x < max goto L3
             goto L1
         L3  x = x * 2
             goto L2

P.code = L2  if x!=y goto L4
             goto L1
         L4  if x < max goto L3
             goto L1
         L3  x = x * 2
             goto L2
         L1
```

11:59
18:57 Verkorte notatie:

```
B.code = B1.code || L4 || B2.code
S.code = L2 || B.code || L3 || S1.code || goto L2
P.code = S.code || L1
```

19:01

(e) In de vertaling van while (B) S_1 moet eerst B geëvalueerd worden. Op het moment dat we concluderen dat B waar is, moet er naar S_1 gesprongen worden. Daarom moeten we in ieder geval een label vóór S_1 hebben staan, dat gelijk is aan B.true.
Dit verklaart

B.true = newlabel

S.code = ... || B.code || label (B.true) || S_1.code || ...

Als we klaar zijn met S_1, moeten we terug naar het begin van de while-lus. Dit verklaart

begin = newlabel

S_1.next = begin ⇝ voor als we 'halverwege' de code van S_1 ontdekken dat we klaar zijn

S.code = label (begin) || ... || gen ('goto' begin)

⇝ voor als we de code voor S_1 gewoon tot het eind doorlopen

Als we bij het evalueren van B concluderen dat B false is, zijn we klaar met de instructie S. Dan moeten we naar de instructie die na S uitgevoerd moet worden. Die instructie krijgt het label S.next (dit label is al gegenereerd, de instructie die dan uitgevoerd moet worden moet nog gemaakt worden). Dit verklaart

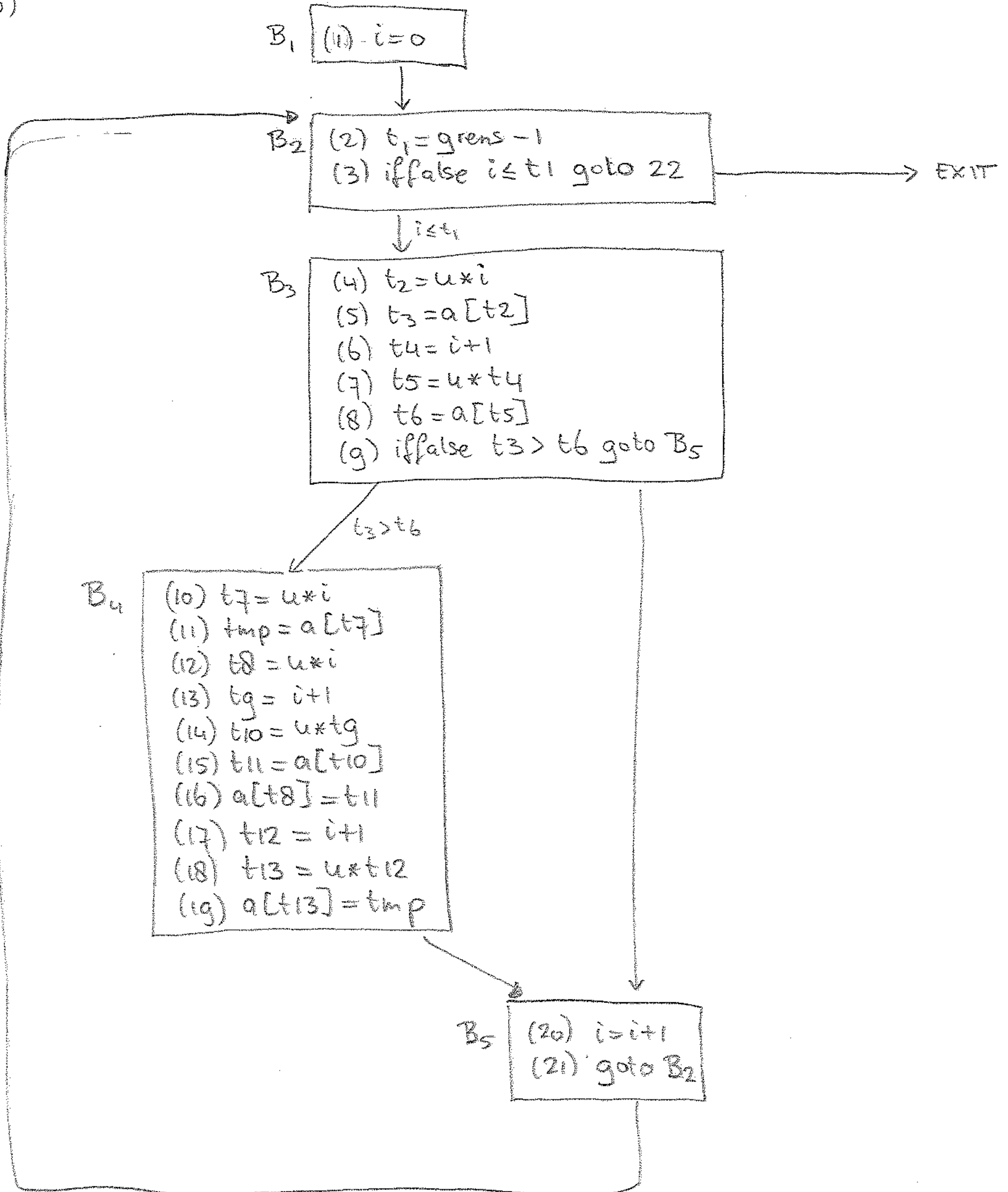
B.false = S.next.

12:13

5 (a) De leaders in de drie-adrescode zijn de instructies 1, 22, 4, 20, 10, 2 ; in oplopende volgorde: 1, 2, 4, 10, 20, 22.

12:17

(b)



12:25

4 [21]

(c) (1) Local common-subexpression elimination (LCSE) voor expressies u*i en i+1 in B_4:
(u*i: t_7 en t_8; i+1: t_9 en t_{12})

B_4

```
t7 = 4*i
tmp = a[t7]
t9 = i+1
t10 = u*t9
t11 = a[t10]
a[t7] = t11
t13 = u*t9
a[t13] = tmp
```

(2) LCSE voor expressie u*t9 (t_{10} en t_{13}) in B_4:

B_4

```
t7 = 4*i
tmp = a[t7]
t9 = i+1
t10 = u*t9
t11 = a[t10]
a[t7] = t11
a[t10] = tmp
```

(3) Global common-subexpression elimination (GCSE) voor expressie u*i (t_2 en t_7) in B_3 en B_4 en expressie i+1 in B_3, B_4 en B_5 (t_4, t_9 en i)

B_4

```
tmp = a[t2]
t10 = u*t4
t11 = a[t10]
a[t2] = t11
a[t10] = tmp
```

B_5

```
i = t4
goto B2
```

14:33
14:35

(4) GCSE voor expressies a[t2] (t_3 en tmp) en u*t4 (t_5 en t_{10}) in B_3 en B_4:

B_4

```
tmp = t3
t11 = a[t5]
a[t2] = t11
a[t5] = tmp
```

(5) GCSE voor expressie a[t5] (t_6 en t_{11}) in B_3 en B_4:

B_4

```
tmp = t3
a[t2] = t6
a[t5] = tmp
```

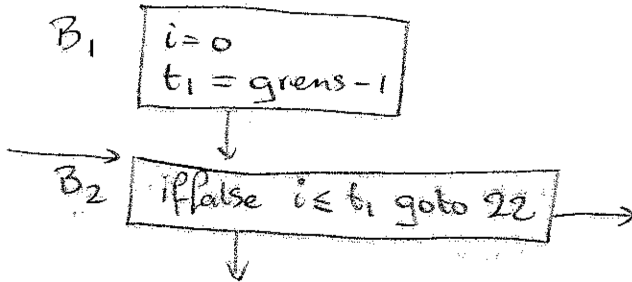
(6) Copy Propagation vanwege copy-instructie tmp = t3 in B_4:

B_4
```
tmp = t3
a[t2] = t6
a[t5] = t3
```

(7) Dead-code elimination voor instructie tmp = t3 in B_4 (ervanuitgaande dat we weten dat waarde van tmp ook buiten onze code ~~niet meer~~ gebruikt wordt.

B_4
```
a[t2] = t6
a[t5] = t3
```

(8) Code motion op instructie t_1 = grens − 1 : van B_2 naar B_1



(9) Reduction in strength op instructies t2 = 4*i en t5 = 4*t4 in B3
Zowel i als t4 wordt iedere iteratie van de while-lus opgehoogd, zodat t2 en t5 met 4 worden opgehoogd.

B_1
```
i = 0
t1 = grens - 1
t2 = 4*i
t5 = 4*i
```

B_3
```
t3 = a[t2]
t4 = i+1
t5 = t5+4
t6 = a[t5]
iffalse t3 > t6 goto B5
```

B_5
```
i = t4
t2 = t2+4
goto B2
```

(10) Copy Propagation op instructie i = 0 in B_1:

B_1
```
i = 0
t1 = grens - 1
t2 = 4*0
t5 = 4*0
```

(11) Constant folding op expressie 4*0 (t_2 en t_5) in B_1:

B_1
```
i=0
t1 = grens - 1
t2 = 0
t5 = 0
```

(12) Copy propagation op instructie $i = t_4$ in B_5 (werkt door in B_2 en B_3, en we moeten t_4 initialiseren in B_1):

B_1
```
i=0
t1 = grens - 1
t2 = 0
t5 = 0
t4 = i
```

B_3
```
t3 = a[t2]
t4 = t4 + 1
t5 = t5 + 4
t6 = a[t5]
iffalse t3 > t6 goto B5
```

B_2
```
iffalse t4 <= t1 goto 22
```

(13) Copy propagation op instructie $i=0$ in B_1:

B_1
```
i=0
t1 = grens - 1
t2 = 0
t5 = 0
t4 = 0
```

(14) Dead code elimination voor instructies $i=0$ in B_1 en $i = t_4$ in B_5 (ervanuitgaande dat we weten dat de waarde van i ook buiten onze code niet meer gebruikt wordt):

B_1
```
t1 = grens - 1
t2 = 0
t5 = 0
t4 = 0
```

B_5:
```
t2 = t2 + 4
goto B2
```

(15) Induction variable elimination.
De tijdelijke variabele t_4 wordt alleen nog gebruikt voor test
iffalse $t_4 \leq t_1$ goto 22
in B_2.
Er geldt: $t_4 \leq t_1 \Leftrightarrow 4 * t_4 \leq 4 * t_1$
En $4 * t_4 = t_5$.
We kunnen inductievariabele t_4 dus elimineren, als we (eenmalig) $4 * t_1$ uitrekenen, met een nieuwe tijdelijke variabele t_{14}

B1:
```
t1 = grens - 1
t14 = 4 * t1
t2 = 0
t5 = 0
```

B2:
```
iffalse t5 ≤ t14 goto 22
```

B3:
```
t3 = a[t2]
t5 = t5 + 4
t6 = a[t5]
iffalse t3 > t6 goto B5
```

De complete flow graph ziet er als volgt uit: