Uitwerking tentamen Compilerconstructie, donderdag 21 december 2017



10.27

1(a)

De front end analyseert het source program, d.w.z. het
* splitst de invoer (minus commentaar en whitespace) op in tokens
* controleert of die stroom van tokens te genereren valt met de grammatica van de programmeertaal, en zo ja, maakt daar een syntax tree van
* controleert of aan de semantische eisen van de programmeertaal is voldaan, en past zonodig de syntax tree aan
* genereert intermediate code

10.33

De back end
* genereert target code
* voert daar eventueel optimalisaties op uit

10.35

(b) Als je voor m verschillende programmeertalen en n verschillende target machines compilers zou willen bouwen, heb je in principe m * n verschillende compilers. Met een geschikte intermediate representation heb je voldoende aan m front ends (van source program in de respectievelijke talen naar intermediate representation) en n back ends (van intermediate representation naar target machine), oftwel m + n 'halve' compilers, in plaats van m * n hele compilers

10.40

2. Het algoritme gaat als volgt:

* for alle terminalen a ; FIRST (a) = {a}
* for alle variabelen A, FIRST (A) = ∅ // initialisatie.
* for alle variabelen A, als er productie A → ϵ is, voeg ϵ toe aan FIRST (A)
* do
  for alle producties $A \to Y_1 Y_2 \cdots Y_n$ in G,
  - for alle terminalen a in FIRST ($Y_1$), voeg a toe aan FIRST (A)
  - als ϵ ∈ FIRST ($Y_1$), voeg alle terminalen a in FIRST ($Y_2$) toe aan FIRST (A)
  - als ϵ ∈ FIRST ($Y_1$) en ϵ ∈ FIRST ($Y_2$), voeg alle terminalen in FIRST ($Y_3$) toe aan FIRST (A)
  - enzovoort.
  - als ϵ ∈ FIRST ($Y_1$), ϵ ∈ FIRST ($Y_2$), ...., ϵ ∈ FIRST ($Y_n$), voeg ϵ toe aan FIRST (A).

  while er nog iets veranderd is in een FIRST-verzameling.

10.50

3 (a)

G is geen LL(1) grammatica, ~~er bij de~~

* omdat er bij de variabele A linksrecursie optreedt (door productie $A \to AdB$)
* omdat variabele B twee producties heeft die allebei met b beginnen

Beide zaken zijn, los van elkaar, al reden om te concluderen dat G geen LL(1) grammatica is.

10.54

(b) Linksrecursie verwijderen:

We vervangen de producties van A (waar dus linksrecursie optreedt) door

$A \to B A'$

$A' \to dBA' \mid \epsilon$

Links factorisatie toepassen.

We vervangen de producties van B (die dus beide met b beginnen) door

$B \to bB'$

$B' \to cb \mid \epsilon$

Naast deze zes producties bevat G' ook nog de producties van S:

$S \to aAS \mid bb$.

10.59

(c)

| X | FIRST(X) | FOLLOW(X) |
|---|---|---|
| S | {a,b} | {$} |
| A | {b} | {a,b} |
| A' | {d, ε} | {a,b} |
| B | {b} | {d,a,b} |
| B' | {c, ε} | {d,a,b} |

11.03.

(d) De top-down parsing table

| | a | b | c | d | $ |
|---|---|---|---|---|---|
| S | $S \to aAS$ | $S \to bb$ | | | |
| A | | $A \to BA'$ | | | |
| A' | $A' \to \epsilon$ | $A' \to \epsilon$ | | $A' \to dBA'$ | |
| B | | $B \to bB'$ | | | |
| B' | $B' \to \epsilon$ | $B' \to \epsilon$ | $B' \to cb$ | $B' \to \epsilon$ | |

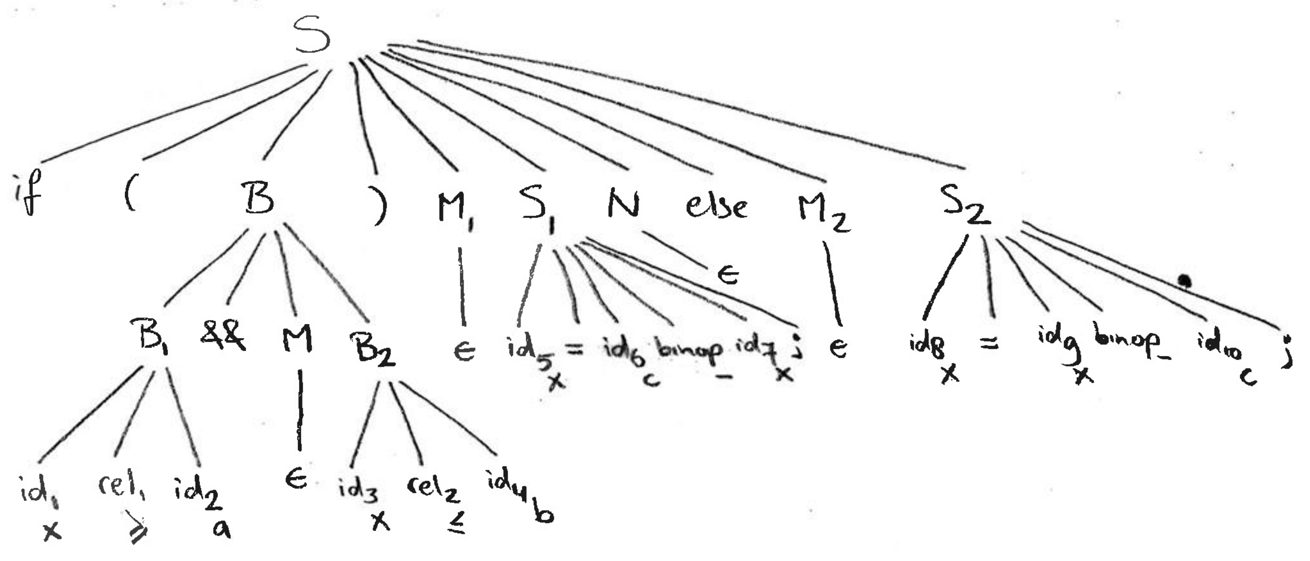
11.09

(e) Ja, de nieuwe grammatica G' is een LL(1) grammatica, want elk vakje in de top-down parsing table bevat hoogstens één productie.

11.11

11.18

4(a)




11.24

(b) We voeren een postorder wandeling (LRW) uit door de parse tree.

$B_1$.truelist = {100}
$B_1$.falselist = {101}
M.instr = 102
$B_2$.truelist = {102}
$B_2$.falselist = {103}
backpatch ({100}, 102)
B.truelist = {102}
B.falselist = {101, 103}
$M_1$.instr = 104
$S_1$.nextlist = null
N.nextlist = {105}

$M_2$.instr = 106
$S_2$.nextlist = null.
backpatch ({102}, 104)
backpatch ({101, 103}, 106)
temp = {105}
S.nextlist = {105}

100 if x ≥ a goto _102
101. goto _106
102. if x ≤ b goto _104
103 goto _106
104 x = c − x
105 goto _
106 x = x − c

(d)

Er gebeuren feitelijk drie dingen

* B.truelist wordt gebackpatched naar $M_1$.instr, dwz op de goto's in B.truelist wordt $M_1$.instr ingevuld. De reden is dat als B true is, moet $S_1$ uitgevoerd worden. Het nummer van de eerste instructie van $S_1$ is opgeslagen in $M_1$.instr, dus daar moeten we naartoe springen.

11.39

* B.falselist wordt gebackpatched naar $M_2$.instr, dwz op de goto's in B.falselist wordt $M_2$.instr ingevuld. De reden is dat als B false is, moet $S_2$ uitgevoerd worden. Het nummer van de eerste instructie van $S_2$ is opgeslagen in $M_2$.instr, dus daar moeten we naartoe springen.

11.41

* $S_1$.nextlist, N.nextlist en $S_2$.nextlist worden samengevoegd (in twee stappen, met behulp van temp) tot S.nextlist. De reden is, respectievelijk,
  - dat als je tijdens het uitvoeren van $S_1$ ontdekt dat je klaar bent met $S_1$, je ook klaar bent met S (hiervoor staan goto's in $S_1$.nextlist)
  - dat als je gewoon $S_1$ helemaal uitvoert, tot en met de laatste drie-adres instructie, je ook klaar bent met S (hiervoor staat een goto in N.nextlist)
  - dat als je tijdens het uitvoeren van $S_2$ ontdekt dat je klaar bent met $S_2$, je ook klaar bent met S (hiervoor staan goto's in $S_2$.nextlist)

11.47.

Merk op dat als je gewoon $S_2$ helemaal uitvoert, tot en met de laatste drie-adres instructie, je ook de laatste drie-adres instructie van S hebt gehad. (daarmee) Je gaat dan automatisch verder met de code die na S moet worden uitgevoerd.

11.49.

12.01

5(a) We noemen een variabele x op een bepaald punt in een programma live, als de waarde die x op dat punt in het programma heeft, later in het programma mogelijk nog gebruikt wordt.

12.03

(b)

* Als x een nieuwe waarde krijgt toegekend, en x is direct na deze toekenning niet live, dan heeft de toekenning geen zin gehad, d.w.z. het is dead code. Dan kunnen we de toekenning schrappen.
* Als we een register nodig hebben om een waarde in te laden of te berekenen, moeten we soms een register vrijmaken. Als daar de waarde van een variabele x staat die niet live is, kunnen we x gewoon overschrijven. Als x weliswaar live is, maar zijn next use is ver in de toekomst, is het ook niet zo erg om de waarde van x in het register te overschrijven (eventueel na een store).

Als het next use van x echter alweer snel is, dan is het niet handig om zijn waarde in het register nu te overschrijven en straks alweer in een register te moeten laden. Dan kan het handiger zijn om nu een ander register vrij te maken

12.14 / 12.16

(c)

| Next use. | a | b | c | d | e |
|---|---|---|---|---|---|
| na regel 5 (on exit) | ° | ° | ° | ° | ° |
| vóór regel 5 | – | 5 | 5 | • | ° |
| vóór regel 4 | 4 | 4 | – | ° | ° |
| vóór regel 3 | – | 4 | – | • | 3 |
| vóór regel 2 | – | 4 | – | 2 | 2 |
| vóór regel 1 (on entry) | 1 | 1 | – | – | 2 |

12.21

Door de toekenning $d = a + b$ in regel 1 krijgt d een nieuwe waarde. Zijn waarde vóór regel 1 is op dat moment dus niet live (dus – in tabel). De waardes van de variabelen a en b worden voor de toekenning gebruikt. Hun next use direct vóór regel 1 is dus regel 1 zelf.

[want wordt overschreven.]

12.25.

6 (a)

begin | bestemming van goto | direct na voorwaardelijke goto.

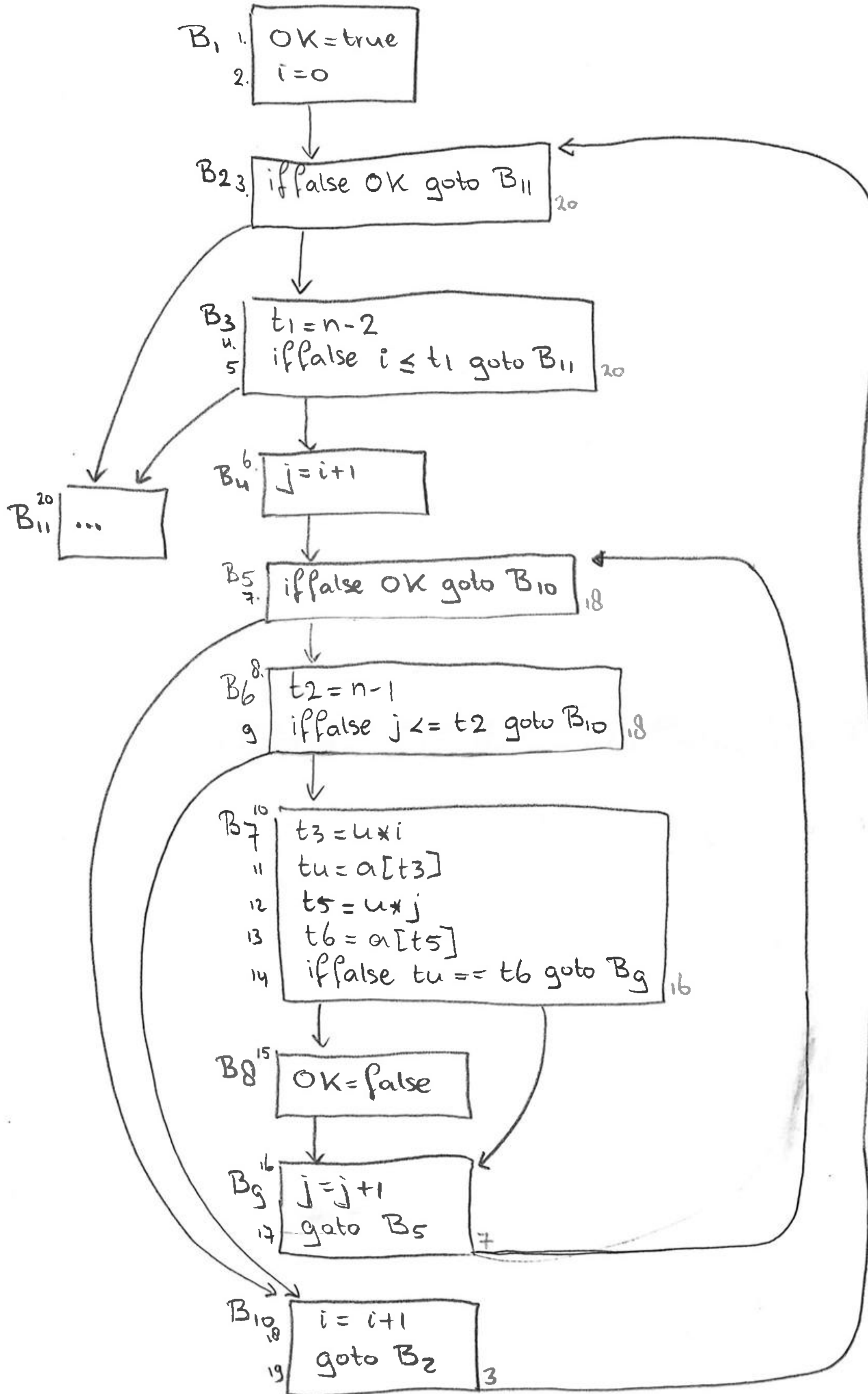
De leaders zijn instructies: 1, 20, 18, 16, 7, 3, 4, 6, 8, 10, 15

In volgorde: 1, 3, 4, 6, 7, 8, 10, 15, 16, 18, 20

[18 en 20 (na onvoorwaardelijke goto's) hebben we al.]

12.29

(b) De flow graph:

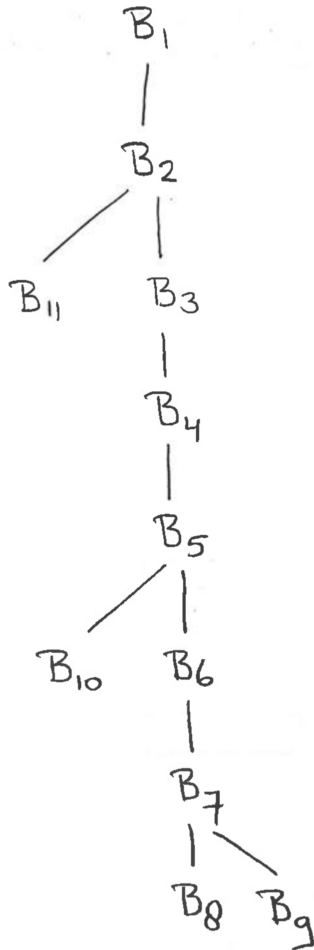



12 3 8

(c) We noemen een knoop d een dominator van knoop n, als elk pad in G vanaf de entry naar knoop n ook knoop d bevat.
In het bijzonder is elke knoop een dominator van zichzelf.

12.41

(d) De dominator tree:




12.43.

(e) We noemen een tak van een knoop a naar een knoop b een back edge, als b een dominator is van knoop a

12.44

(f) Back edges zijn: $B_9 \rightarrow B_5$
$B_{10} \rightarrow B_2$.

12.45

(g) De natural loops zijn:

bij $B_9 \rightarrow B_5$: $B_5, B_9, B_8, B_7, B_6$

bij $B_{10} \rightarrow B_2$: $B_2, B_{10}, B_6, B_5, B_9, B_8, B_7, B_4, B_3$

12.48