10:25 Uitwerking tentamen Compilerconstructie, dinsdag 17 december 2013

1.
(*) De opbouw van de compiler wordt overzichtelijker, met verschillende componenten voor verschillende taken. Daardoor wordt de kans op fouten kleiner en de onderhoudbaarheid gemakkelijker.
(*) We kunnen dan efficiënter compileren, door speciale technieken (reguliere expressies, eindige automaten) in te zetten voor de lexicale analyse
(*) De compiler wordt meer 'portable': alleen de lexical analyser hoeft dan nog rekening te houden met eigenaardigheden in/van de invoer.

10:32

2 (a) (*) Dat een variabele gedeclareerd moet zijn, voordat hij wordt gebruikt. Dat betekent dat je programma van de vorm $X_1 w X_2 w X_3$ moet zijn voor de variabele w. Dat is echter niet af te dwingen met een context-vrije grammatica. De taal $\{www \mid w \in \{a,b\}^*\}$ is al niet context-vrij.
(*) Dat een functie bij aanroep evenveel parameters moet hebben als bij haar definitie.
Neem twee functies f_1 en f_2 met m, resp. n parameters.
Stel dat we in een programma achtereenvolgens
- de definitie van f_1
- de definitie van f_2
- een aanroep van f_1
- een aanroep van f_2

hebben. Dan ziet het programma er feitelijk uit als $X_1 a^m X_2 b^n X_3 a^m X_4 b^n X_5$. De gelijkheid van de aantallen a's en de aantallen b's is echter niet af te dwingen met een context-vrije grammatica. De taal $a^m b^n a^m b^n$ is al niet context-vrij.

10:43

(b) Deze onderdelen van de syntax kunnen (dus) niet tijdens het parsen gecontroleerd worden. Daarom gebeurt dat tijdens de semantische analyse (ook al zijn het syntax controles). Samen met bepaalde (echte) semantische controles vormt dit de Static Checker.

10:47

3 (a) Er is sprake van directe links-recursie als G producties van de vorm $A \to A\alpha$ bevat.
Stel dat dat het geval is voor de variabele A en dat de (alle) producties van A de volgende zijn:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

waarbij $\beta_1, \beta_2, \ldots, \beta_n$ niet met A beginnen.
Dan kunnen we deze producties vervangen door

$$A \to \beta_1 X \mid \beta_2 X \mid \cdots \mid \beta_n X$$
$$X \to \alpha_1 X \mid \alpha_2 X \mid \cdots \mid \alpha_m X \mid \in$$

waarbij X een nieuwe variabele is

Dan zijn we van de links-recursie af (aannemende dat de α_i's niet leeg zijn, maar goed, een productie $A \to A.B$ kunnen we sowieso weggooien).

10:53

(b) G_1 kent directe links-recursie ($B \to Bb$) en indirecte links-recursie ($S \Rightarrow AB \Rightarrow CaB \Rightarrow SbaB$).
Een handigere volgorde is D, S, A, C, B zie blz. (7)
We lopen de variabelen in de volgorde S, A, B, C, D af, en we zorgen ervoor dat ze alleen nog maar 'vooruit kunnen wijzen'.
Dat wil zeggen: als er een productie $A_i \to A_j\alpha$ is, moet A_j ná A_i komen in de volgorde

$S \to AB$ geen probleem
$A \to \in \mid Ca$ geen probleem
$B \to bD \mid Bb$ directe links-recursie ⇒ vervang de producties
door $B \to bDX \quad X \to bX \mid \in$
$C \to Sb \mid a$ C wijst terug naar S ⇒ vervang de producties
door $C \to ABb \mid a$ C wijst terug naar A ⇒ vervang de producties
door $C \to Bb \mid CaBb \mid a$
C wijst terug naar B ⇒ vervang de productie
door $C \to bDXb \mid CaBb \mid a$
directe links-recursie ⇒ vervang de producties
door $C \to bDXbY \mid aY \quad Y \to aBbY \mid \in$
$D \to Sa \mid b$ D wijst terug naar S ⇒ vervang de producties
door $D \to ABa \mid b$
D " " " A ⇒ " " "
door $D \to Ba \mid CaBa \mid b$
D wijst " " B ⇒ " " "
door $D \to bDXa \mid CaBa \mid b$
D wijst " " C ⇒ " " "
door $D \to bDXa \mid bDXbYaBa \mid aYaBa \mid b$.

11:07.

11:18

(c) Alleen bij de variabele D komen we producties tegen met rechterkanten die met zelfde letter(s) beginnen.
⇒ vervang de producties van D eerst door

$D \rightarrow bV \mid aYaBa$
$V \rightarrow DXa \mid DXbYaBa \mid \epsilon$

Nu heeft variabele V nog twee producties waar rechterkanten hetzelfde beginnen. ⇒ vervang de producties van V door

$V \rightarrow DXW \mid \epsilon$
$W \rightarrow a \mid bYaBa$

11:23

(d) First (D) = {a, b}
First (V) = {a, b, ε}
First (W) = {a, b}
First (C) = {b, a}
First (Y) = {a, ε}
First (B) = {b}
First (X) = {b, ε}
First (A) = {ε, b, a}
First (S) = {b, a}

11:27

(e). Nee, G_3 is geen LL(1) grammatica.
Bij bepalen van First (S) zagen we dat de b zowel bij A vandaan kan komen als (wanneer $A \rightarrow \epsilon$) bij B

Concreet:

$S \Rightarrow AB \Rightarrow Ca \Rightarrow bDXba$
$S \Rightarrow AB \Rightarrow B \Rightarrow bDX$

11:30.

4 (a) (1) Dat betekent dat we op dit moment bezig kunnen zijn met het opbouwen van de rechterkant $\beta_1\beta_2$ van de productie $A \rightarrow \beta_1\beta_2$, (2) dat we daarbij al genoeg letters in de invoer zijn tegengekomen om de deelstring β_1 te matchen. (en ook geen letters daarna meer),
zodat we (3) nog zouden moeten proberen om de deelstring β_2 met (de eerste letters van) de resterende invoer te matchen

11:37

In de eerste regel staat 'kunnen', omdat het ook best mogelijk is dat we uiteindelijk de rechterkant van een andere productie aan het opbouwen waren

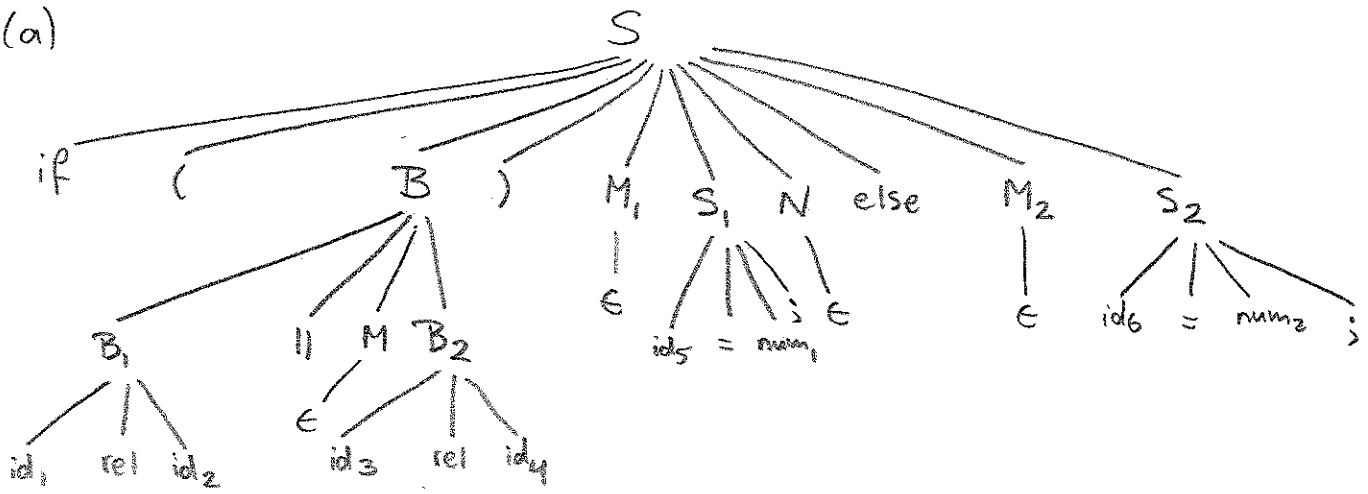
ontdekken dat we

11:39

(b) We kunnen tijdens het SLR parsen reduceren naar de productie $A \rightarrow \beta$, wanneer
* we in een toestand zitten met daarin het item $A \rightarrow \beta.$
* en de eerstvolgende letter in de invoer in Follow(A) zit

11:43

5 (a)



11:47.

(b) We bepalen de attributen bottom-up (LRW)

B_1.truelist = {100}
B_1.falselist = {101}
M.instr = 102
B_2.truelist = {102}
B_2.falselist = {103}

100. if a == b goto _→104
101. goto _→102
102. if c = d goto _→104
103 goto _→106
104 x = 1
105 goto _
106 x = 2

11:53

12:05

B.truelist = {100,102}
B.falselist = {103}
M_1.instr = 104
S_1.nextlist = null
N.nextlist = {105}
M_2.instr = 106
S_2.nextlist = null
temp = {105}
S.nextlist = {105}

12:12

(c)

backpatch (B.truelist, M_1.instr);
De Goto's uit B.truelist gaan naar M_1.instr
Immers, op die momenten dat je binnen B vaststelt dat B true is/wordt, moet je S_1 uit gaan voeren. Het instructienummer waar de code van S_1 begint is in M_1.instr opgeslagen.

Daarvoor staan al goto's in code voor B, maar nog zonder goto-adres. De instructienummers van deze goto's staan in B.truelist

backpatch (B.falselist, M_2.instr)
De Goto's uit B.falselist gaan naar M_2.instr
Immers, op die momenten dat je binnen B vaststelt dat B false is/wordt, moet je S_2 uit gaan voeren. Het instructienummer waar de code van S_2 begint is in M_2.instr opgeslagen.

temp = merge (S_1.nextlist, N.nextlist);
S.nextlist = merge (temp, S_2.nextlist);
Resultaat: S.nextlist = S_1.nextlist ∪ N.nextlist ∪ S_2.nextlist

Als je tijdens S_1 of S_2 ontdekt dat je klaar bent met S_1 / S_2 (terwijl de code van S_1 / S_2 nog verder gaat), moet je springen naar de code na S. Deze (nog onbepaalde) Goto's staan in S_1.nextlist en S_2.nextlist.

12:22 Net zo: als we de code van S_1 helemaal afgelopen hebben,
12:36 komen we bij de Goto die in N.nextlist staat. Ook dan moeten we naar de code na S.

Zo gauw we weten naar welke regel we moeten na S, kunnen we al deze Goto's invullen. Daarom voegen we ze bij elkaar in S.nextlist. Die verzameling moet immers alle Goto's binnen S bevatten naar de code na S.

12:40

09:59 **Uitwerking tentamen Compilerconstructie, dinsdag 17 december 2013**

6 (a) Reaching definitions op een bepaald punt in een programma zijn die definities (toekenningen van een waarde aan een variabele) die nog geldig kunnen zijn op dat punt in het programma

10:02

(b) We gebruiken een voorwaartse dataflow analyse voor het berekenen van reaching definitions, omdat de reaching definitions op een bepaald punt in een programma definities zijn die eerder (in het programma) zijn uitgevoerd (van kracht zijn geworden), en in de tussentijd niet per se ongeldig zijn geworden.

10:09 Om informatie over eerdere definities door te geven naar latere punten in het programma (via paden met instructies voor de tussentijd) is juist voorwaartse dataflow analyse geschikt

10:15

(c)

1. $OUT[ENTRY] = \emptyset$
2. $OUT[B] = \emptyset$
3. $IN[B] = \bigcup_{\text{predecessors } P \text{ of } B} OUT[P]$.
4. $OUT[B] = gen_B \cup (IN[B] - kill_B)$

Hierbij is gen_B de verzameling van definities in block B die niet later in datzelfde block B ongeldig zijn geworden ('ge-kill-ed zijn'). Hierbij is $kill_B$ de verzameling van alle definities in het (hele) programma die in block B ongeldig worden, dwz: alle definities van een variabele u waarvoor block B ook een (andere) definitie bevat.

10:23 (andere: een definitie van een variabele u maakt alle andere definities van u in het hele programma ongeldig, maar natuurlijk niet zichzelf.

10:25

10:28

(d) Een gunstige volgorde is B_1, B_2, B_3, B_4, B_5 (EXIT), want dat is een 'voorwaartse volgorde' in het stroomdiagram. De reaching definitions aan het eind van een bepaald basic block kunnen meteen gebruikt worden voor de berekening van reaching definitions in de opvolger(s) van dat basic block.
Die opvolger(s) is/zijn immers (ongeweer) als volgende aan de beurt in deze volgorde. (die we net berekend hebben,

10:37

(e)

B	na initialisatie: OUT[B]	na iteratie 1: IN[B]	na iteratie 1: OUT[B]	na iteratie 2: IN[B]	na iteratie 2: OUT[B]
B_1	$\emptyset$	$\emptyset$	$\{d_1\}$	$\emptyset$	$\{d_1\}$
B_2	$\emptyset$	$\{d_1\}$	$\{d_1, d_3, d_4\}$	$\{d_1, d_4, d_5, d_7, d_8\}$	$\{d_1, d_3, d_4, d_5, d_7\}$
B_3	$\emptyset$	$\{d_1, d_3, d_4\}$	$\{d_3, d_4, d_5\}$	$\{d_1, d_3, d_4, d_5, d_7\}$	$\{d_3, d_4, d_5, d_7\}$
B_4	$\emptyset$	$\{d_1, d_3, d_4\}$	$\{d_1, d_3, d_4, d_6\}$	$\{d_1, d_3, d_4, d_5, d_7\}$	$\{d_1, d_3, d_4, d_5, d_6\}$
B_5	$\emptyset$	$\{d_1, d_3, d_4, d_5, d_6\}$	$\{d_1, d_4, d_5, d_7, d_8\}$	$\{d_1, d_3, d_4, d_5, d_6, d_7\}$	$\{d_1, d_4, d_5, d_7, d_8\}$
EXIT	$\emptyset$	$\{d_1, d_4, d_5, d_7, d_8\}$	$\{d_1, d_4, d_5, d_7, d_8\}$	$\{d_1, d_4, d_5, d_7, d_8\}$	$\{d_1, d_4, d_5, d_7, d_8\}$

In de 3e iteratie verandert er niets meer aan de OUT-verzamelingen (en ook niets meer aan de IN-verzamelingen).

10:53 Alternatief.

3(b) We lopen de variabelen in de volgorde D, S, A, C, B af, en we zorgen ervoor dat ze alleen nog maar 'vooruit kunnen wijzen'.
Dat wil zeggen: als er een productie $A_i \to A_j\alpha$ is, moet A_j nà A_i komen in de volgorde

$D \to Sa \mid b$ geen probleem
$S \to AB$ geen probleem
$A \to \epsilon \mid Ca$ geen probleem.
$C \to Sb \mid a$ C wijst terug naar S $\Rightarrow$ vervang de producties d'r.
door $C \to ABb \mid a$ C wijst terug naar A $\Rightarrow$ vervang de producties
door $C \to Bb \mid CaBb \mid a$
directe links-recursie $\Rightarrow$ vervang de producties
door $C \to BbX \mid aX$ $\quad X \to aBbX \mid \epsilon$
$B \to bD \mid Bb$ directe links-recursie $\Rightarrow$ vervang de producties
door $B \to bDY$ $\quad Y \to bY \mid \epsilon$

(c) Er is geen enkele variabele met verschillende producties waarvan de rechter kant met zelfde niet-lege substring begint.
$\Rightarrow$ geen linksfactorisatie mogelijk.

(d)
First (Y) = {b, ε}
First (B) = {b}
First (X) = {a, ε}
First (C) = {b, a}
First (A) = {ε, b, a}
First (S) = {b, a}
First (D) = {b, a}

(e) Niet LL(1), want we hebben (b.v.) producties
$D \to Sa \mid b$
waarbij First (Sa) = {b, a}
en First (b) = {b}
$\Rightarrow$ de First-verzamelingen van de rechterkanten van deze producties (bij dezelfde variabele D) zijn niet disjunct.