TENTAMEN COMPILERCONSTRUCTIE
Dinsdag 17 december 2013, 10:00 – 13:00 uur

Dit tentamen bestaat uit 6 opgaven.

Als je het antwoord op een onderdeel niet weet, en je hebt dat antwoord nodig bij een later onderdeel, dan kun je het antwoord 'kopen' bij de docent.

Als er bij een opgave gevraagd wordt om uitleg of motivatie van je antwoord, is het belangrijk dat je die ook geeft.

---

1. Noem drie redenen waarom het nuttig is om de lexicale analyse en de syntax analyse door afzonderlijke componenten van de compiler uit te laten voeren.

---

2. (a) Noem twee onderdelen van de syntax van C++ die niet vastgelegd kunnen worden met een context-vrije grammatica.
   Licht ook kort toe *waarom* deze onderdelen van de syntax niet vastgelegd kunnen worden met een context-vrije grammatica.
   (b) In welke fase van het compileren worden deze onderdelen van de syntax gecontroleerd?

---

3. (a) Leg uit hoe je in het algemeen directe (*immediate*) links-recursie elimineert uit een context-vrije grammatica G, zonder de gegenereerde taal te veranderen.
   (b) Elimineer op een systematische wijze de (directe en indirecte) links-recursie uit de grammatica G_1 met variabelen $\{S, A, B, C, D\}$, terminalen $\{a, b\}$, startvariabele S en de volgende producties:

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & \epsilon \mid Ca \\ B & \rightarrow & bD \mid Bb \\ C & \rightarrow & Sb \mid a \\ D & \rightarrow & Sa \mid b \end{array}$$

   Leg uit hoe je te werkt gaat en geef tussenresultaten.
   N.B.: hoewel G_1 een ϵ-productie bevat, werkt het algoritme uit het boek wel gewoon.
   (c) Laat G_2 het resultaat van het vorige onderdeel zijn. Construeer vanuit G_2 een nieuwe context-vrije grammatica G_3 door (indien van toepassing) links-factorisatie toe te passen.
   (d) Bepaal voor elke variabele in de resulterende grammatica G_3 de FIRST-verzameling.
   (e) Is G_3 een LL(1) grammatica? Motiveer je antwoord.

---

4. (a) Bij het SLR parsen maken we gebruik van een LR(0)-automaat. De toestanden in deze automaat bevatten items van de (algemene) vorm $A \rightarrow \beta_1 \cdot \beta_2$.
   Leg uit wat het betekent wanneer we tijdens het SLR parsen in een toestand met daarin het item $A \rightarrow \beta_1 \cdot \beta_2$ zijn aangekomen.
   (b) Leg met behulp van o.a. items uit een LR(0)-automaat uit *wanneer* we tijdens het SLR parsen kunnen reduceren naar een productie $A \rightarrow \beta$

---

5. Tijdens het genereren van drie-adres code voor boolese expressies en *flow-of-control* instructies weten we bij Goto-instructies vaak niet onmiddellijk waar we naartoe moeten springen. We kunnen *backpatching* gebruiken om dit achteraf op te lossen. Hierbij krijgt de variabele B in de grammatica (overeenkomend met een boolese expressie) attributen *truelist* en *falselist*. De variabele S (overeenkomend met een instructie) krijgt een attribuut *nextlist*.

   Naast deze variabelen gebruiken we hieronder hulpvariabelen M (met een attribuut *instr*) en N (met een attribuut *nextlist*).

   Beschouw het volgende *translation scheme* voor het genereren van de genoemde drie-adres code:

   $S \rightarrow \textbf{if}\ (B)\ M_1 S_1 N\ \textbf{else}\ M_2 S_2$
   $\{\ backpatch(B.truelist, M_1.instr);$
   $backpatch(B.falselist, M_2.instr);$
   $temp = merge(S_1.nextlist, N.nextlist);$
   $S.nextlist = merge(temp, S_2.nextlist);\ \}$

   $S \rightarrow \textbf{id}\ = \textbf{num};$ $\{\ S.nextlist = \textbf{null};$
   $gen(\textbf{id}.addr\ '='\ \textbf{num}.val);\ \}$

   $B \rightarrow B_1\ ||\ MB_2$ $\{\ backpatch(B_1.falselist, M.instr);$
   $B.truelist = merge(B_1.truelist, B_2.truelist);$
   $B.falselist = B_2.falselist;\ \}$

   $B \rightarrow \textbf{id}_1\ \textbf{rel}\ \textbf{id}_2$ $\{\ B.truelist = makelist(nextinstr);$
   $B.falselist = makelist(nextinstr + 1);$
   $gen('\texttt{if}'\ \textbf{id}_1.addr\ \textbf{rel}.op\ \textbf{id}_2.addr\ '\texttt{goto}\ \_');$
   $gen('\texttt{goto}\ \_');\ \}$

   $M \rightarrow \epsilon$ $\{\ M.instr = nextinstr;\ \}$

   $N \rightarrow \epsilon$ $\{\ N.nextlist = makelist(nextinstr);$
   $gen('\texttt{goto}\ \_');\ \}$

   (a) Teken de afleidingsboom (*parse tree*) bij bovenstaande grammatica (met startvariabele S) voor het volgende 'programma':

   ```
   if (a==b || c==d)
     x=1;
   else
     x=2;
   ```

   (b) Pas bij de afleidingsboom van het vorige onderdeel de semantische acties toe zoals beschreven in het translation scheme. Geef bij elke variabele in de boom aan wat de attributen (*truelist*, *falselist*, *nextlist* en/of *instr*) worden. Geef ook de resulterende drie-adres code. Ga ervanuit dat deze drie-adres code begint op instructienummer 100.

   (c) Leg uit wat er volgens het translation scheme gebeurt (de semantische actie) bij de productie

   $$S \rightarrow \textbf{if}\ (B)\ M_1 S_1 N\ \textbf{else}\ M_2 S_2$$

   Leg ook uit waarom dat gebeurt.

   N.B.: het gaat er hier om wat er in het algemeen gebeurt bij deze productie, en niet zozeer wat er gebeurt in het geval van ons voorbeeld'programma'.

6. (a) Wat zijn *reaching definitions* op een bepaald punt in een programma?

   De algemene opzet van een iteratief algoritme van voorwaartse *dataflow* analyse is als volgt (waarbij de knopen de *basic blocks* zijn):

   OUT[ENTRY] = ...
   for each node B other than ENTRY
       OUT[B] = ...

   while (changes to any OUT occur)
       for each node B other than ENTRY
       { IN[B] = $\ldots_{\text{predecessors } P \text{ of } B}$ OUT[P]
           (i.e., combine the OUT-sets of the predecessors in some way)

           OUT[B] = ... (some function of IN[B])
       }

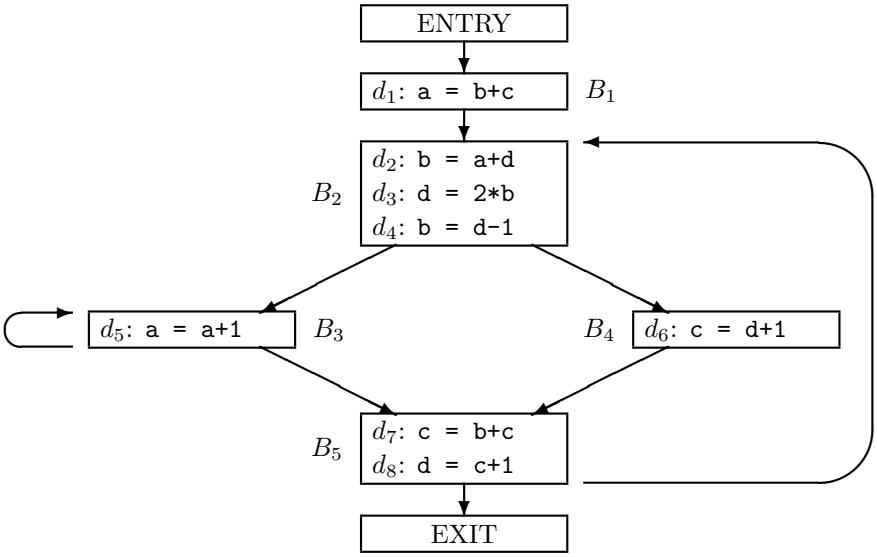
   (b) Leg uit waarom we een *voorwaartse* (dus niet een achterwaartse) dataflow analyse gebruiken voor het berekenen van reaching definitions in een stroomdiagram (*flow graph*).

   (c) Vul de vier '...' in de algemene opzet van het iteratieve algoritme in voor het berekenen van reaching definitions. Voor iedere knoop B in het stroomdiagram moeten IN[B] en OUT[B] aan het eind van het algoritme alle reaching definitions aan het begin, respectievelijk einde van B bevatten.

   Wees met name ook precies in je beschrijving van 'some function'.

Z.O.Z.

Beschouw nu het volgende stroomdiagram:



(d) Wanneer we het iteratieve algoritme willen gebruiken om de reaching definitions in een stroomdiagram te berekenen, moeten we een volgorde kiezen waarin we de basic blocks aflopen in de binnenste for-lus (dwz: de for-lus binnen de while-lus).

Wat is een gunstige volgorde van de basic blocks bij bovenstaand stroomdiagram? Motiveer je antwoord.

(e) Pas nu het iteratieve algoritme om de reaching definitions te berekenen toe op bovenstaand stroomdiagram. Laat met een overzichtelijke tabel zien wat de IN en OUT-verzameling van elk basic block (op ENTRY na) is na de initialisatie en na iedere iteratie. De laatste iteratie, waarin je zou constateren dat er toch niets veranderd is, hoef je niet uit te voeren.