TENTAMEN COMPILERCONSTRUCTIE
Maandag 14 december 2015, 14:00 – 17:00 uur

Dit tentamen bestaat uit 5 opgaven, waarbij steeds tussen [en] staat hoeveel punten er ongeveer mee te verdienen zijn. In totaal zijn er 100 punten te verdienen.

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. [10 pt] We kunnen in het compileerproces de volgende zeven fasen onderscheiden (in alfabetische volgorde):

   - Code Generator
   - Intermediate Code Generator
   - Lexical Analyser
   - Machine-Dependent Code Optimizer
   - Machine-Independent Code Optimizer
   - Semantic Analyser
   - Syntax Analyser

   (a) Plaats deze fasen in de juiste volgorde.

   (b) Ervanuitgaande dat de fasen keurig na elkaar worden uitgevoerd, levert elk van de fasen een representatie van het *source program* (bijvoorbeeld: syntax tree, assembly,. . .). Noem het soort representatie dat elk van de fasen produceert.

   N.B.: dit zijn niet per se allemaal verschillende soorten representatie.

---

2. [25 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{array}{rcl} S & \rightarrow & M \mid O \\ M & \rightarrow & \textbf{if expr then } M \textbf{ else } M \mid \textbf{id} = \textbf{num} \\ O & \rightarrow & \textbf{if expr then } S \mid \textbf{if expr then } M \textbf{ else } O \end{array}$$

G is een eenvoudige grammatica voor een instructie in een programmeertaal, waarin de *dangling-else*-dubbelzinnigheid is geëlimineerd. S, M, O zijn de variabelen (overeenkomend met *Statement*, *Matched statement* en *Open statement*) en **if**, **expr**, **then**, **else**, **id**, $=$, **num** zijn de terminalen in G.

(a) Leg uit waarom je zo al (zonder grondige analyse) kunt concluderen dat G geen LL(1) grammatica is.

(b) Construeer vanuit G een context-vrije grammatica G' door
- (indien van toepassing) links-recursie te elimineren
- (indien van toepassing) links-factorisatie toe te passen.

Leg uit hoe je G' uit G verkrijgt.

(c) Bepaal voor elke variabele in de nieuwe grammatica G' zowel de FIRST- als de FOLLOW-verzameling.

(d) Construeer de top-down *parsing table* bij de nieuwe grammatica G'. Wellicht ten overvloede: dit is dus iets anders dan de SLR parsing table.

(e) Is de nieuwe grammatica G' een LL(1) grammatica? Motiveer je antwoord.

---

3. [20 pt] Beschouw de volgende context-vrije grammatica (met startvariabele T) om type expressies te genereren voor integers, floats en arrays daarvan.

$$
\begin{array}{lcl}
T & \rightarrow & BC \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

   (a) Teken een afleidingsboom (*parse tree*) voor de type expressie **int**[4][50].

   Alle drie variabelen in bovenstaande grammatica hebben attributen *type* en *width*. Deze attributen worden ingevuld met behulp van het volgende *translation scheme*:

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\} \\
 & & C & \{\ T.type = C.type;\ T.width = C.width;\} \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\}
\end{array}
$$

   (b) Annoteer de afleidingsboom van onderdeel (a), dat wil zeggen: vul de attributen *type* en *width* in, zoals gespecificeerd door het translation scheme.

   (c) Teken de met de resulterende type expressie corresponderende boom ('syntax tree').

   (d) Leg nu uit wat er volgens het translation scheme gebeurt (de semantische acties) bij de producties

$$T \rightarrow BC \qquad C \rightarrow \epsilon \qquad C \rightarrow [\ \textbf{num}\ ]\ C$$

   Leg ook uit waarom dat gebeurt.

   N.B.: het gaat er hier om wat er in het algemeen gebeurt bij deze producties, en niet zozeer wat er gebeurt in het geval van onze voorbeeld type expressie.

---

4. [20 pt] Stel dat we assembly code voor de drie-adres instructie $x = y+z$ willen genereren, waarbij x, y en z drie (niet per se verschillende) integer variabelen zijn. Dan moeten de argumenten (*operands*) van de operator + in een register geladen worden, voordat de operator kan worden toegepast.

   (a) Voor de selectie van een register voor een variabele kunnen we gebruik maken van *register descriptors* en *address descriptors*. Wat verstaan we onder deze twee 'descriptors'?

   (b) Leg uit hoe we de register descriptors en address descriptors kunnen gebruiken om te bepalen of y al in een register zit, en of er nog een register vrij is.

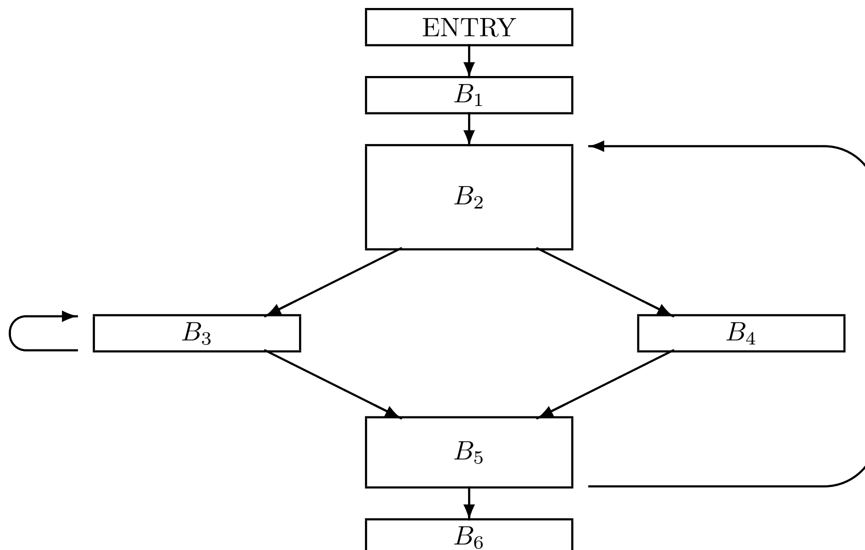
   (c) Beschrijf een algoritme om een register R_y voor y te bepalen wanneer y nog niet in een register zit en er ook geen register vrij is. Dit algoritme moet (minstens) dezelfde onderdelen bevatten als het algoritme in het boek, maar deze onderdelen hoeven niet per se in dezelfde volgorde te staan.

---

5. [25 pt]

   (a) Laat G een stroomdiagram (*flow graph*) zijn met beginknoop *entry*. Wanneer noemen we een knoop d in G een dominator van een knoop n in G?

   Beschouw nu het volgende stroomdiagram G:



   (b) Geef de *dominator tree* voor stroomdiagram G.

   (c) Geef een *depth-first spanning tree* T voor G.

   (d) Wat zijn de *retreating edges* van stroomdiagram G bij boom T ? En wat zijn de *cross edges*? En wat zijn de *back edges*? Motiveer je antwoorden.

   (e) Geef een *depth-first ordering* van de knopen in G, aansluitend bij de boom T. Leg uit hoe je aan je antwoord komt.

   (f) Leg uit hoe je (in het algemeen) uit een depth-first spanning tree T voor een stroomdiagram G kunt afleiden, hoe snel een iteratief algoritme voor voorwaartse data flow analyse (dat gebruik maakt van een depth-first ordering bij T) kan convergeren?