TENTAMEN COMPILERCONSTRUCTIE
Donderdag 19 december 2019, 14.15 – 17.15 uur

Dit tentamen bestaat uit zes 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. [11 pt] Voor de specificatie van patronen waaraan de lexemes van een token moeten voldoen kunnen we gebruik maken van een reguliere definitie. De lexical analyser generator Lex (of Flex) doet dat ook.

   (a) Leg uit hoe een reguliere definitie er (in het algemeen) uit ziet.
   N.B.: we vragen niet naar een reguliere *expressie*, maar naar een reguliere *definitie*.

   (b) Geef een voorbeeld van een reguliere definitie die de identifiers in de programmeertaal C beschrijft.

   (c) Beschrijf (duidelijk en volledig) een algoritme om een reguliere definitie voor een token om te bouwen tot een reguliere expressie over alleen het oorspronkelijke alfabet Σ.

---

2. [22 pt]

   (a) Leg duidelijk uit aan welke voorwaarden de FIRST- en FOLLOW-verzamelingen moeten voldoen om ervoor te zorgen dat een context-vrije grammatica LL(1) is.

   Beschouw nu de context-vrije grammatica G met startvariabele S en de volgende producties:

$$
\begin{aligned}
S &\rightarrow AcB \mid bA \\
A &\rightarrow Bb \mid Ac \\
B &\rightarrow Sa
\end{aligned}
$$

   De terminalen in G zijn dus a, b, c. Laat je niet afleiden door het feit dat $L(G) = \emptyset$. Pas gewoon de gebruikelijke technieken toe.

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

   Leg uit hoe je te werk gaat, en geef, indien van toepassing, ook tussenresultaten. Bij de uitleg licht je ook toe waarom je een stap uitvoert, maar je hoeft de stappen niet voor algemene grammatica's te beschrijven – geef wel het resultaat van elke stap voor deze specifieke grammatica.

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

   (d) Beredeneer, gebruikmakend van je antwoord bij onderdelen (a) en (c) of G' een LL(1) grammatica is. Wellicht ten overvloede: het is niet de bedoeling om een complete top-down parsing table te construeren.

---

3. [9 pt] Beschouw de context-vrije grammatica G met startvariabele S en de volgende producties:

$$\begin{array}{ll} (1) & S \to TB \\ (2) & T \to aTb \\ (3) & T \to ab \\ (4) & B \to bB \\ (5) & B \to b \end{array}$$

(a) Geef (ad hoc) een *parse tree* in G voor de string *abbb*

(b) Gegeven is dat de SLR parsing table bij grammatica G er als volgt uit ziet:

State	Action			Goto		
	a	b	\$	S	T	B
0	s6			1	2	
1			acc			
2		s4				3
3			r1			
4		s4	r5			5
5			r4			
6	s6	s9			7	
7		s8				
8		r2				
9		r3				

Parse de string *abbb* met deze tabel. Laat bij iedere stap duidelijk zien wat je doet, bijvoorbeeld met behulp van een tabel van de volgende vorm:

States on stack	Corresponding symbols on stack	Input	Action
0	\$	...	...
...	...	...	...

---

4. [17 pt] Bij declaratie van een variabele wordt ook zijn type opgeslagen in de symbol table.

(a) Wat verstaan we onder de *width* van een type?

Behalve basis types als *integer* en *float* kunnen we ook samengestelde types hebben. We geven de structuur van een type weer met een *type expression*, bijvoorbeeld *array*(3, *array*(4, *integer*)) (een array van 3 arrays van elk 4 integers). Dit type kan er in een programmeertaal uitzien als `int[3][4]` Dergelijke array types kunnen gegenereerd worden met het volgende stukje van een context-vrije grammatica, met startvariabele T:

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

(b) Teken een *parse tree* voor het type `int[3][4]`

Om de juiste type expression op te bouwen, met de juiste width, krijgen de drie nonterminals T, B en C allemaal een attribuut *type* en een attribuut *width*. Deze attributen

krijgen hun waarde met het volgende (nog onvolledige) translation scheme:

$$
\begin{array}{llll}
T & \rightarrow & B & \{ \; t = B.type; \; w = B.width; \} \\
 & & C & \{ \; T.type = C.type; \; T.width = C.width; \} \\
B & \rightarrow & \textbf{int} & \{ \; B.type = integer; \; B.width = 4; \} \\
C & \rightarrow & \epsilon & \{ \; C.type = \ldots; \; C.width = \ldots; \} \\
C & \rightarrow & [\; \textbf{num} \;] \; C_1 & \{ \; C.type = \ldots; \\
 & & & \;\;\; C.width = \ldots; \}
\end{array}
$$

(c) Vul de (totaal) vier '. . . ' bij de producties $C \rightarrow \epsilon$ en $C \rightarrow [\; \textbf{num} \;] \; C_1$ in.
Hint: bedenk wat voor het voorbeeld `int[3][4]` het uiteindelijke attribuut $T.type$ moet worden.

(d) Pas bij de parse tree uit onderdeel (b) de semantische acties toe zoals beschreven in het translation scheme. Geef bij elk (voorkomen van een) variabele in de boom aan wat de waarde van zijn attributen *type* en *width* wordt. Vermeld de attributen in de volgorde waarin ze hun waarde krijgen.

---

5. [15 pt] Bij register allocatie kun je ervoor kiezen om aan het eind van elk basic block in de drie-adres code alle variabelen, die live zijn aan het eind van het block en waarvan de huidige waarde alleen in een register te vinden is, op te slaan (te *storen*) in het geheugen. Aan het begin van een basic block ga je er dan vanuit dat de registers leeg zijn.

   Als alternatief kun je ervoor kiezen om, gedurende een loop L, een register te reserveren voor een variabele die binnen de loop vaak gebruikt wordt.

   (a) Voor de keuze van de betreffende variabele kun je kijken welke variabele de meeste besparingen aan loads en stores oplevert, wanneer je voor hem een register reserveert. Als schatting voor de besparing voor een variabele x wordt in het boek de volgende formule gegeven:

   $$\text{totale besparing} \approx \sum_{\text{blocks } B \in L} use(x,B) + 2 * live(x,B)$$

   i. Leg uit wat $use(x,B)$ en $live(x,B)$ in deze formule betekenen.
   ii. Beredeneer waarom, gegeven de betekenis van $use(x,B)$ en $live(x,B)$, bovenstaande formule een redelijke schatting is van de totale besparing voor variabele x.

   (b) Wat is een nadeel van het reserveren van een register voor een variabele?

---

6. [26 pt]

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

   (b) Beschouw een willekeurige definitie $d : u = v + w$ binnen een programma van drie-adres instructies. Wat zijn, in het kader van het berekenen van reaching definitions, (concreet) de verzamelingen gen_d en $kill_d$ voor deze instructie?

Z.O.Z.

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

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

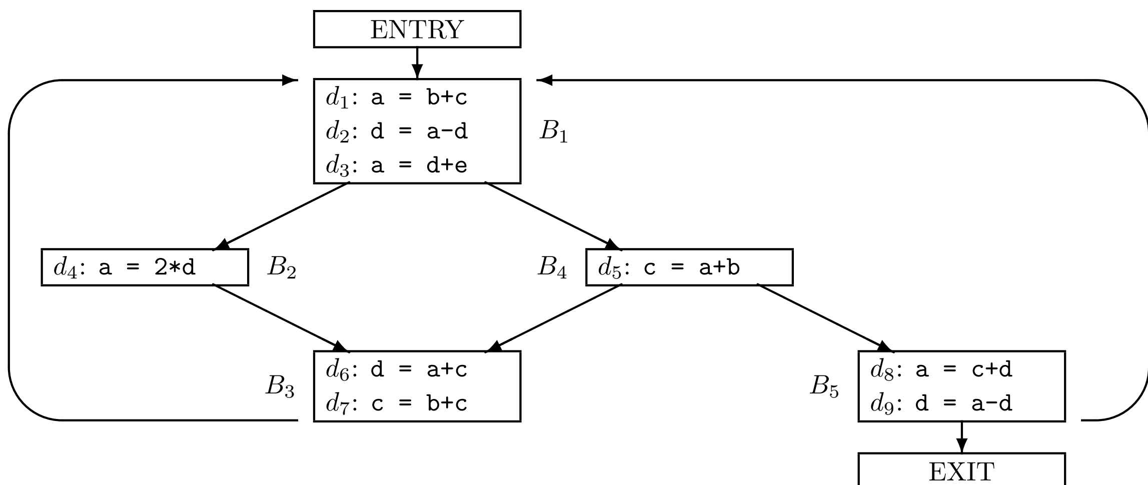
while (changes to any OUT occur)
    for each node B other than ENTRY
    {   IN[B] = $\dots_{\text{predecessors } P \text{ of } B}$ OUT[P]
        (i.e., combine the OUT-sets of the predecessors in some way)

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

(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'.

Beschouw nu het volgende stroomdiagram:



(d) Geef voor elk blok B in bovenstaand stroomdiagram (afgezien van ENTRY en EXIT) de verzamelingen gen_B en $kill_B$.

(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 van de while-lus. De laatste iteratie, waarin je zou constateren dat er toch niets veranderd is, hoef je niet uit te voeren.

Kies een gunstige volgorde om de blokken af te lopen in de binnenste for-lus (dwz: de for-lus binnen de while-lus in het algoritme), en vermeld deze volgorde expliciet.